Context-aware & privacy-preserving homecare monitoring through adaptive query derivation for IoT data streams with DIVIDE

Mathias De Brouwer [a,*], Bram Steenwinckel [a], Ziye Fang [a], Marija Stojchevska [a], Pieter Bonte [a], Filip De Turck [a], Sofie Van Hoecke [a] and Femke Ongenae [a]
[a] *IDLab, Ghent University – imec, Belgium*
E-mails: mrdbrouw.DeBrouwer@UGent.be, Bram.Steenwinckel@UGent.be, Ziye.Fang@UGent.be, Marija.Stojchevska@UGent.be, Pieter.Bonte@UGent.be, Filip.DeTurck@UGent.be, Sofie.VanHoecke@UGent.be, Femke.Ongenae@UGent.be

Abstract. Integrating Internet of Things (IoT) sensor data from heterogeneous sources with domain knowledge and context information in real-time is a challenging task in IoT healthcare data management applications that can be solved with semantics. Existing IoT platforms often have an issue with preserving the privacy of patient data. Moreover, the configuration and management of context-aware stream processing queries in semantic IoT platforms requires much manual, labor-intensive effort. Generic queries can deal with context changes but often lead to performance issues caused by the need for expressive real-time semantic reasoning. In addition, query window parameters are part of the manual configuration and cannot be made context-dependent. To tackle these problems, this paper presents DIVIDE, a component for a semantic IoT platform that automatically and adaptively derives and manages the queries of the platform's stream processing components in a privacy-preserving, context-aware and scalable manner. By performing semantic reasoning to derive the queries when context changes are observed, their real-time evaluation does require any reasoning. The results of an evaluation on a homecare monitoring use case demonstrate how activity detection queries derived with DIVIDE can be evaluated in on average less than 3.7 seconds and can therefore successfully run on low-end IoT devices.

Keywords: Context-aware query derivation, Homecare monitoring, Internet of Things, Cascading reasoning, Semantic reasoning, RDF stream processing

1. Introduction

In the healthcare domain, many applications involve a large collection of Internet of Things (IoT) devices and sensors [1]. Many of those systems typically focus on the real-time monitoring of patients in hospitals, nursing homes, homecare or elsewhere. In such systems, patients and their environment are being equipped with different devices and sensors for following up on the patients' conditions, diseases and treatments in a personalized, context-aware way. This is achieved by integrating the data collected by the IoT devices with existing domain knowledge and context information. As such, analyzing this combination of data sources jointly allows a system to extract meaningful insights and actuate on it [2].

Integrating and analyzing the generated IoT data with domain knowledge and context information in a real-time context is a challenging task. This is due to the typically high volume, variety and velocity of the different

*Corresponding author. E-mail: mrdbrouw.DeBrouwer@UGent.be.

IoT data sources [3]. To deal with these challenges, semantic IoT platforms can be deployed [4]. Semantic IoT platforms generally contain stream processing components that integrate and analyze the different data sources by continuously evaluating semantic queries. To deploy this, Semantic Web technologies are typically employed: ontologies are designed to integrate and model the data from different heterogeneous sources and its relationships and properties in a common, machine-interpretable format, and existing stream reasoning techniques are used by the data stream processing components [5].

In healthcare systems, preserving the privacy of the patients is of utmost importance [6]. In IoT platforms, lots of the data generated by the IoT devices can contain privacy-sensitive information. Depending on where the data processing components are being hosted, this privacy-sensitive data may have to be sent over the IoT network, potentially exposing it to the outside world. Hence, privacy preservation is an important requirement of a semantic IoT platform, that is ideally solved by processing the IoT data close to where it is generated so that the amount of information sent over the network is reduced as much as possible.

Currently, the configuration and management of queries that run on the stream processing components of a semantic IoT platform is a manual task that requires a lot of effort from the end user. In the typical IoT applications in healthcare, the queries installed on the stream processing components of the semantic IoT platform should be context-aware: the context information determines which sensors and devices should be monitored by the query, for example to filter specific events to send to other components for further analysis. For example, a patient's diagnosis in the Electronic Health Record (EHR) determines the monitoring tasks that should be performed in the patient's hospital room, while the indoor location (room) of the patient in a homecare monitoring environment implies what in-home activities can be monitored. Changes in this context information regularly occur. For example, the profile information of patients in their EHR can be updated, or the in-home location of the patient can evolve over time. Hence, the management of the queries should be able to deal with such context changes. Currently, no semantic IoT platform component exists that allows to configure, derive and manage the platform's queries in an automated, adaptive way. Therefore, platforms typically apply one of two existing approaches to achieve this.

The first approach to introduce context-awareness into semantic queries is by defining them in a generic fashion. A generic query uses generic ontology concepts in its definitions so that it can be used to perform several more specific tasks that are contextually relevant. This way, semantic reasoners will reason in real-time on all context data and domain knowledge, and the full data stream containing all generated sensor observations, to determine the contextually relevant sensors and devices to which the query is applicable. The advantage of this approach is that such queries are prepared to deal with contextual changes: due to their generic nature, they should not be updated often. However, the disadvantage of highly generic queries is the computational complexity of the semantic reasoning during their real-time evaluation: this complexity is typically high due to complex ontologies in IoT domains such as healthcare that require expressive reasoning [7]. In healthcare applications that involve a large number of sensors, it is practically challenging to do this in real-time: queries take longer to evaluate, causing lower performance and difficulty to keep up with the required query execution frequencies. Typically, central components in an IoT platform have more resources and are therefore more likely to overcome this challenge, but this would require all generated IoT streaming data to be sent over the network, causing the network to be highly congested all the time. In addition, the central server resources would be constantly in use, and local decision making would no longer be possible. More importantly, this also conflicts with the privacy preservation requirement introduced before. Looking at local and edge IoT devices to run those generic queries instead, resources are typically lower, making the performance challenges even a larger issue of the generic query approach.

An alternative approach that can be adopted is installing multiple specific queries on the stream processing components that filter the contextually relevant sensors for one specific task. Because they are no longer generic, evaluating such queries reduces the required semantic reasoning effort. However, while this approach solves the performance issues associated to deploying generic queries because of the reduced semantic reasoning, it even further increases the required manual query configuration and management effort for the end user: whenever the context changes, the queries should be manually updated. This is infeasible to do in practice. As a consequence, current platforms do not apply this approach often and mostly work with generic queries instead.

Currently, the definition of generic stream processing queries does not contain any means to make the window parameters of the query depend on the application context and domain knowledge. This means that the end user should define these parameters as a part of the query configuration, and cannot let the system automatically decide

on them based on the data. This can be a problem in some specific monitoring cases. For example, the size of the data window on which a monitoring task such as in-home activity detection should be executed, may depend on the type of task, and therefore be defined in the domain knowledge. Another example is when the execution frequency of a certain monitoring task is dependent on certain contextual events happening in the patient's environment.

A semantic IoT platform component that would solve the presented issues, should also be practically usable. Currently, existing semantic IoT healthcare platforms use semantic reasoners or stream reasoners that are configured with existing sets of generic semantic queries [2]. Defining such queries and ensuring their correctness is a delicate and time-consuming task. Hence, a new component should not introduce a completely different means of defining generic queries, but instead reduce the required changes to these definitions to a minimum. This implies that it should start from the generic definition of stream processing queries. Moreover, the other configuration tasks of the component should also be as minimal as possible to increase overall usability.

In summary, there is a need for a semantic IoT platform component that fulfills the different requirements tackled in the previous paragraphs, so that it can be applied in a healthcare data management system. Hence, we set the following research objectives for the design of such an additional semantic IoT platform component:

1. The component should be privacy-preserving, ensuring the amount of privacy-sensitive health-related and other information exposed to the outside world is limited as much as possible.
2. The component should reduce the manual, labor-intensive query configuration effort by managing the queries on the platform's stream processing components in an automated, adaptive and context-aware way.
3. The evaluation of queries managed by the component should be performant, also on low-end IoT edge devices with fewer resources. Network congestion and overuse of central resources should be avoided.
4. The component should allow for the query window parameters to be context-dependent.
5. The component should be practically usable, minimizing the effort to integrate it into an existing system.

This paper presents DIVIDE, a semantic IoT platform component that we have designed to achieve the presented research objectives. DIVIDE automatically and adaptively derives and manages the contextually relevant specific queries for the platform's stream processing components, by performing semantic reasoning with a generic query definition whenever contextual changes occur. As a result, the derived queries will efficiently monitor the relevant IoT sensors and devices in real-time, and still do not require any real-time reasoning during their evaluation.

The remainder of this paper is structured as follows. Section 2 discusses some related work. In Section 3, the eHealth use case scenario is further explained, translated into the technical system set-up, and semantically described with an ontology. Section 4 presents a general overview of the DIVIDE system. Further functional and algorithmic details of DIVIDE are provided in Section 5 and Section 6 using the running use case example, while Section 7 zooms in on the technical implementation of DIVIDE. Section 8 describes the evaluation set-up with the different evaluation scenarios and hardware set-up. Results of the evaluations are presented in Section 9, and further discussed in Section 10. Finally, Section 11 concludes the main findings of the paper and highlights future work.

2. Related work

Using Semantic Web technologies such as the Resource Description Framework (RDF) and the Web Ontology Language (OWL), heterogeneous data sources can be consolidated and semantically enriched into a machine-interpretable representation using ontologies [4]. An ontology is a model that semantically describes all domain-specific knowledge by defining domain concepts and their relations and attributes. Semantic reasoners can interpret semantic data to derive new knowledge based on the definitions in the ontologies. The complexity of the semantic reasoning depends on the expressivity of the underlying ontology [8]. Different ontology languages exist. They range from RDFS, which has the lowest expressivity, to OWL 2 DL, which has the highest expressivity.

RDFox [9] and VLog [10] are state-of-the-art OWL 2 RL reasoners. OWL 2 RL contains all constructs that can be evaluated by a rule engine. These constructs can be expressed by simple Datalog rules. By design, these engines are not able to handle streaming data. However, RDFox can also run on a Raspberry Pi, and any ARM-based IoT edge device in general. In addition, previous research has shown it can also successfully run on a smartphone [11].

Notation3 Logic (N_3) [12] is a rule-based logic that is often used to write down RDF. N_3 is a superset of RDF/-Turtle [13], which implies that any valid RDF/Turtle definitions are valid N_3 as well.

Stream Reasoning (SR) [5] state-of-the-art contains three main approaches: Continuous Processing (CP) engines, Reasoning Over Time (ROT) frameworks and Reasoning About Time (RAT) frameworks. CP engines have continuous semantics, high throughput, and low latency but do not perform reasoning. ROT frameworks solve reasoning tasks continuously with high throughput and low latency, but do not consider time. RAT frameworks do consider time in the reasoning task, but may lack reactivity due to the high latency. These various approaches each investigate the trade-off between the expressiveness of reasoning and the efficiency of processing [5].

RDF Stream Processing (RSP) identifies a family of CP engines that solve information needs over heterogeneous streaming data, which is typical in IoT applications. It addresses data variety by adopting RDF streams as data model, and solves data velocity by extending SPARQL with the continuous semantics [14]. Different RSP engines exist, such as C-SPARQL [15], CQELS [16], Yasper [17] and RSP4J [18]. Queries can be registered to these engines that are used to continuously filter the defined data streams. A data window is placed on top of the data stream. Parameters of the window definition include the size of the data window that is added to the query's data model, and the window's sliding step which directly influences the query's evaluation frequency.

RSP-QL [19] is a reference model that unifies the semantics of the existing RSP approaches. RSP has been extended to support ROT in various ways: (i) solutions incorporating efficient incremental maintenance of materializations of the windowed ontology streams [20–23], (ii) solutions for expressive Description Logics (DL) [24, 25], and (iii) a solution for Answer Set Programming (ASP) [26]. More central to ROT is the logic-based framework for analyzing reasoning over streams (LARS) [27] that extends ASP for analytical reasoning over data streams. LASER [28] is a system, based on LARS, that employs a tractable fragment of LARS that ensures uniqueness of models. BigSR [29] employs Big Data technologies (e.g., Apache Spark and Flink) to evaluate the positive fragment of LARS. C-Sprite [30] focuses on efficient hierarchical reasoning to improve the throughput and application on edge devices by efficiently filtering out unnecessary data in the stream. A similar approach to filter out unnecessary streaming data in ASP exists, by investigating the dependency graph of the input data [31]. RDF Event Processing (RSEP) identifies a family of approaches that extend CP over RDF Streams with event pattern matching [32]. RSEP extends RSP with a reactive RAT formalism with limited expressiveness [33]. RSEP-QL [34] is an extension of RSP-QL that incorporates the language features from CEP. StreamQR [35] rewrites continuous RSP queries to multiple parallel queries, allowing for the support of ontologies that are expressed in the $\mathcal{ELHIO}$ logic. The CityPulse project [36] presents the combination of RSP, CEP and expressive reasoning through ASP.

The most advanced attempts to develop expressive Stream Reasoning increased the reasoning expressiveness, but at the cost of limited efficiency. DyKnow [37] and ETALIS [38] combine RAT and ROT reasoning, but perform CP at an extremely slow speed. STARQL [39] is a first step in the right direction because it mixes RAT, and ROT reasoning utilizing a VKG approach to obtain CP. Cascading Reasoning [40] was proposed to solve the problem of expressive reasoning over high-frequency streams by introducing a hierarchical approach consisting of multiple layers. Although several of the presented approaches adopt a hierarchical approach [26, 38, 39], only recent attempt has laid the first fundamentals on realizing the full vision of cascading reasoning with Streaming MASSIF [41].

Today, different IoT platforms exist that extend big data platforms with IoT integrators [42, 43]. FIWARE [44] is a platform that offers different APIs that can be used to deploy IoT applications. Sofia2 [45] is a semantic middleware platform that allows different systems and devices to become interoperable for smart IoT applications. SymbIoTe [46] goes a step further and abstracts existing IoT platforms by providing a virtual IoT environment provisioned over various cloud-based IoT platforms. The Agile [47] and BIG IoT [48] platforms focus on flexible IoT APIs and gateway architectures, such as VICINITY [49] and INTER-IoT [50] which also provide an interoperability platform. bIoTope [51] addresses the requirement for open platforms within IoT systems development.

In general, many of the presented platforms are adopting a wide range of existing Semantic Web technologies to deal with the challenges associated to real-time IoT applications in complex IoT domains such as healthcare. These platforms typically combine different technologies that involve both stream processing and semantic reasoning components. They all have in common that the queries for the stream processing components are not yet configured and managed in a fully automated, adaptive and context-aware way.

3. Use case description and set-up

To demonstrate how DIVIDE enables to perform context-aware and privacy-preserving homecare monitoring, a detailed use case is presented in this section. First, the section describes the details of the use case itself, before discussing the architectural set-up corresponding with this use case and the constructed ontology that allows to apply the use case in a semantic platform.

3.1. Use case description

The homecare monitoring use case scenario presented in this paper focuses on a rule-based service that recognizes the activities of elderly people in their homes. This subsection zooms in on the problem statement that leads to a solution containing such a service, the details of this activity recognition service within this use case, and a running example that will be used in the description of DIVIDE in the following main sections of this paper.

Use case background More and more people live with chronic illnesses and are followed up at home by various healthcare actors such their General Practitioner (GP), nursing organization, and volunteers. Patients in homecare are increasingly equipped with monitoring devices such as lifestyle monitoring devices, medical sensors, localization tags, etc. The shift to homecare makes it important to continuously assess whether an alarming situation occurs at the patient and which intervention strategy is required, by reasoning on the measured parameters in combination with the medical know-how.

A core building block of a homecare monitoring solution is an autonomous activity detection and recognition service, that also monitors whether ongoing activities belong to a known regular routine of the patient or not. Such a service could make use of the data collected by the different sensors and devices installed in the patient's home environment, as well as knowledge about activity recognition rules and known routines of the patient. Given the heterogeneous nature of these different data sources, Semantic Web technologies are ideally suited to create this autonomous activity recognition service that will be the focus of the use case of this paper. These technologies can be employed in a cascading architecture.

Details of the activity recognition service The use case of routine and non-routine activity recognition has been designed together with the home monitoring company Z-Plus. To properly perform knowledge-driven activity recognition, activity recognition rules should be known by the system. Z-Plus has helped us with designing the rules.

An activity recognition rule can be defined as a set of one or more conditions defined on certain observable properties that are being analyzed for a certain entity type. An observable property is any property that can be measured by a sensor in the patient's environment, e.g., temperature, relative humidity, power consumption, door status (open vs. closed), etc. Every sensor in the environment is analyzing a specific entity of a certain type. An entity can be a certain room, for example for a temperature or humidity sensor in a room, but it can also be a specific object like a cooking stove or a television of which the power consumption is measured by a sensor, or a fridge or cupboard of which a sensor is observing the door status. Given this information for each sensor in the environment, a rule can unambiguously express by which sensors it can be triggered, by specifying for each condition which observable property and entity type it is relevant for. If one would only specify the observable property, this would not be the case, since this would trigger a rule's condition for every sensor observing that property, even if it is completely irrelevant to the activity.

Conditions can be defined on the exact value of an observation, or on a crossed lower or upper threshold of a property's observed value or an aggregation of multiple observation values. Moreover, conditions can be combined in multiple ways, e.g. by defining a logical *and*, logical *or*, or a time order.

By definition, the sensors installed in a patient's home environment define the possible activity rule conditions that can be defined. If no sensor is installed that analyzes an observable property for a specific entity type, a rule with a condition defined on such a combination will never be triggered. Therefore, to properly deploy this activity recognition service, a wide range of sensors should be installed. Examples of relevant sensor types to define activity rules are sensors measuring a person's indoor location, electrical power consumption of the electrical devices in the home, indoor temperature, relative humidity, air quality, light intensity, sound level, the status (open vs. closed) of

doors, windows and closets including fridges and cupboards, tap water consumption, etc. In fact, this list can be endless: whatever property can be measured, can be used in activity rule definitions.

In a realistic home environment with a wide range of sensors installed, many different specific activity rules can be defined. Such a high number of rules is required to have a proper activity recognition service that is able to recognize a wide range of in-home activities. However, in such realistic deployments with many different sensors and activity rules, it would be highly inefficient to continuously monitor all possible activities that can be recognized in the home at all times, since this would require the continuous monitoring of the observations made by all sensors that observe a certain property for an entity type associated to at least one rule. Hence, to reduce this number of sensors that is being monitored at a certain time, the activity recognition service makes the activity monitoring location dependent: it only observes activities that are relevant to the room that the patient is currently located in. This assumes that only activities can be observed that are actively being performed by the patient, which is obvious and of no limitation. To enable such location-based activity recognition, two requirements need to be fulfilled. First, an indoor location system should be installed in the patient's home. This sensor system should unambiguously know the location, i.e., room, of the patient in the home at every point in time. Second, to limit the observed sensors based on the location of the patient, it should be defined for each sensor to what room(s) this sensor is relevant. This definition can normally be implicitly derived from the entity that this sensor is analyzing. If the sensor is analyzing a property of a specific room or of an object contained in this room, it is automatically relevant to that room. If a sensor is a wearable of a person, this implies that the sensor is relevant to the current location of the person, i.e., that is always relevant.

If activities are being recognized by the service, it is important to include in the output of the service whether or not this activity belongs to the regular routine of this patient. If an ongoing activity in the patient's routine is recognized, the situation is normal and requires no more strict follow-up. Ideally, as long as an activity is going on, location changes in the home are less probable and should therefore be monitored less strictly, i.e., checks for location changes should happen less frequently. However, if an activity outside the routine of the patient is being detected, more strict location monitoring is required since the situation is abnormal. If necessary, an alarm should automatically be generated by the system. To implement such a system, knowledge on the existing routines of the patient at different times of the day should exist, such as a morning routine, lunch routine, dinner routine, evening routine and sleep routine. Again, for this activity recognition service, only the existence of this knowledge is important, while its source does not really matter.

Finally, an important requirement of the activity recognition service is that it preserves the privacy of the patient as much as possible. Hence, the information that leaves the patient's home environment should be kept to a minimum. This implies that no actual raw sensor data should be sent to over the network. To enable this, the activity recognition service should largely run in-home, so that only the actual outputs such as detected activities are being sent. Obviously, data that not contained in the HomeLab should always be sent over a secure encrypted connection.

Running example As a running example within the discussion of DIVIDE in the following main sections of this paper, consider a smart home with at least the following sensors and devices installed: an indoor location system detecting in which room of the house the patient is present, and an environmental sensor system measuring the relative humidity in every room of the home. The smart home consists of multiple rooms including one bathroom. The patient living in the home has a morning routine that includes showering. The current time in the running example is considered at 9 am, implying that a detected showering activity would be an expected routine activity.

To keep it simple, the activity recognition service of the running example consists of a single rule. This rule detects when a person is showering, and is formulated as follows:

A person is showering if he or she is present in a bathroom with a relative humidity of at least 57%.

This is a rule with a single condition, defined on a crossed lower threshold for the relative humidity observable property, for the bathroom entity type. Hence, given the presence of a humidity sensor in the patient's bathroom, the showering activity will be monitored by the activity recognition service *if* the patient is located in the bathroom. The patient's indoor location is also being continuously monitored. Note that this showering rule is used for illustration purposes only. It is probably too simplistic for a realistic environment as it has not been validated on realistic data.

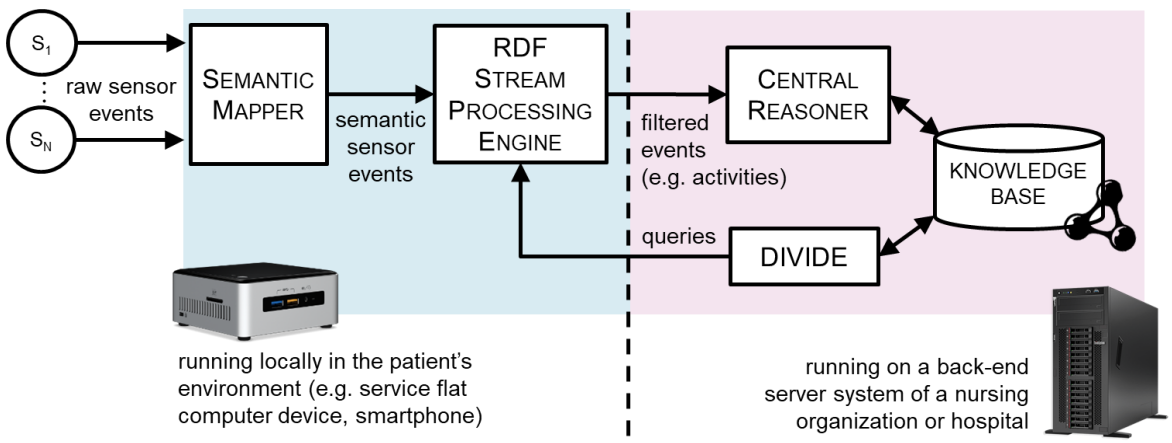


Figure 1. System set-up in the eHealth use case scenario

3.2. Architectural use case set-up

To implement the use case scenario of a knowledge-driven routine and non-routine activity recognition service that is privacy-preserving, as explained in Section 3.1, a cascading reasoning architecture is used [52]. An overview of the architectural cascading reasoning set-up for this use case is shown in Figure 1. This architecture is generic and can be applied to different eHealth use case scenarios in homecare monitoring that need to preserve privacy. In this paper, this architecture is employed for the activity recognition service.

The architecture of the system is split up in a local and a central part. The local part on the one hand consists of a set of components that are running on a local device in the patient's environment. This device could be any existing home gateway that is already installed in the patient's home, such as the device for a deployed nurse call system. The local components are the Semantic Mapper and an RDF Stream Processing (RSP) Engine. On the other hand, the components of the central part are deployed on a back-end server of an associated nursing home or hospital. They consist of a Central Reasoner, the DIVIDE system and a central Knowledge Base.

Knowledge Base The Knowledge Base contains the semantic representation of all knowledge in the system. This consists of both domain knowledge and contextual information. In the given use case scenario, this domain knowledge consists of the Activity Recognition ontology that is discussed in Section 3.3. It includes the activity recognition model with its different activity rules. The contextual information contains semantic data about the different smart homes and their installed sensors, and patient information including routine information.

Semantic Mapper The Semantic Mapper is responsible for semantically annotating all raw observations generated by the sensors in the patient's environment. This includes all sensors installed in the home as well as any wearable sensors. These semantic sensor observations are sent to the data streams of the RSP Engine.

RSP Engine The RSP engine continuously evaluates the registered queries on the RDF data streams, to filter relevant events. In this use case scenario, the filtered events are in-home locations and recognized activities both in and not in the patient's routine. Only these filtered events are sent over the network to the Central Reasoner. By applying the cascading reasoning principles and installing the RSP engine locally in the patient's environment, the privacy-preserving requirement of the activity recognition is met.

Central Reasoner The Central Reasoner is responsible for further processing the events received from the RSP engine, and acting upon them. For example, it can aggregate the filtered events and save them to use for future call enrichment, or send an alarm to the patient's caregivers when necessary. In general, any action is possible, depending on what additional components are deployed and implemented on the central node.

Importantly, the Central Reasoner will also update relevant contextual information in the Knowledge Base such as relevant contextual events occurring in the patients' environment. This information can then trigger a re-evaluation

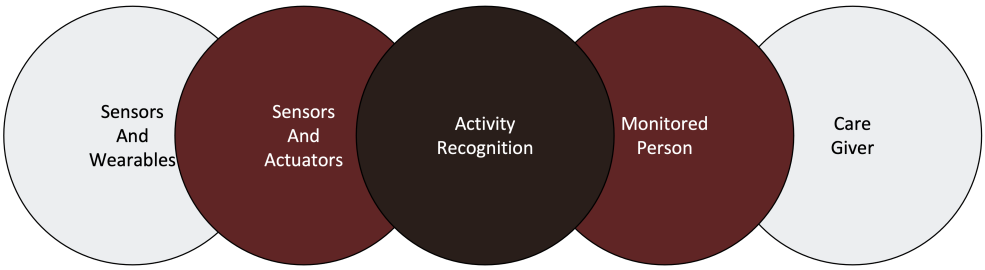


Figure 2. Overview of the different modules in the DAHCC ontology [53]

of the current queries deployed on the local components. In the given use case scenario, relevant context changes that trigger a possible change in the deployed RSP queries are location updates and detected activities. When the in-home location of the patient changes, the set of activities that need to be monitored by the set of RSP queries changes as well, since the activity recognition service is location-dependent as explained in Section 3.1. Moreover, context information about whether an activity is currently being recognized, and whether this activity is part of the patient's routine or not, directly defines the execution frequency of the location monitoring RSP query.

DIVIDE In this architecture, DIVIDE is the component that manages the queries that are being executed by the RSP Engine components on the patients' local devices. It updates the queries whenever triggered by context updates in the Knowledge Base. By aggregating contextual information with medical domain knowledge through semantic reasoning during the query derivation, the resulting RSP queries only involve filtering and do not require any more real-time reasoning. Moreover, it allows to dynamically manage the window parameters of the queries (i.e., the size of the data window and its sliding step) based on the current context. It is fully automated and adaptive, so that at all times, relevant queries are being executed given the important context information about the patient, as defined in the Knowledge Base. Sections 4, 5 and 6 further explain how DIVIDE internally works from a methodological point of view, using the running example to explain how it is applied in the use case scenario described in Section 3.1. Sections 7 zooms in on the implementation details of DIVIDE.

In the running example, DIVIDE will ensure that there is always a location monitoring query running on the RSP Engine component installed in the patient's home. The window parameters of this query will depend on whether or not an activity is currently going on, and whether or not this activity belongs to the current patient's routine. In addition, when the patient is located in the bathroom, an additional RSP query will be derived by DIVIDE and installed on the RSP Engine that monitors when the patient is showering. When the query detects this activity, this would be considered a recognized routine activity as showering is included in the patient's morning routine.

3.3. Activity recognition ontology

An Activity Recognition ontology has been designed to support the described use case scenario. This Activity Recognition ontology is linked to the DAHCC (Data Analytics for Heathcare and Connected Care) ontology [53], which is an in-house designed ontology that includes different modules connecting data analytics to healthcare knowledge. Specifically for the purpose of this semantic use case, it is extended with a module `KBActivityRecognition` supporting the knowledge-driven recognition of in-home activities.

Figure 2 shows the five main modules of the DAHCC ontology. The `SensorsAndActuators` and `SensorsAndWearables` modules describe the concepts that allow defining the observed properties, location, observations and/or actions of different sensors, wearables and actuators in a monitored environment such as a smart patient home. The `MonitoredPerson` and `CareGiver` modules contain concepts for the definition of a patient monitored inside his residence and its caregivers. The `ActivityRecognition` module allows describing the activities associated to a monitored person that are predicted by an activity recognition model.

The DAHCC ontology bridges the concepts of multiple existing ontologies in the data analytics and healthcare domains. These ontologies include SAREF (the Smart Applications REFerence ontology) [54] and its extensions SAREF4EHAW (SAREF extended with concepts of the eHealth Ageing Well domain) [55], SAREF4BLDG

(an extension for buildings and building spaces) and SAREF4WEAR (an extension for wearables), as well as the Execution-Executor-Procedure (EEP) ontology [56].

The following listings will detail some of the definitions within the Activity Recognition ontology that support the use case of the knowledge-driven activity recognition and its running example as described in Section 3.1. These listings will be used as a reference when explaining the details of DIVIDE in the following sections. For reference purposes, all prefixes used in the listings in the remainder of this paper are presented in Listing 1.

Listing 2 shows how some of the concepts of the `KBActivityRecognition` ontology module are defined. More specifically, it shows how a knowledge-based activity recognition model can be defined and configured. In the example, it is configured according to the use case's running example, i.e., with one activity rule for showering. Lines 13–17 of this listing contain the definition of the single condition of this rule. The condition is defined as a lower regular threshold that should be crossed, since it is an instance of `RegularThreshold` with the data property `isMinimumThreshold` set to `true`. The threshold value is set to 57. The relevant property (relative humidity) and analyzed entity type (bathroom) of this rule condition are described via the `forProperty` and `analyseStateOf` object properties, respectively.

Listing 3 further lists some ontology subclass and equivalence definitions within the Activity Recognition ontology. They define for the showering activity type when an activity prediction corresponds to the routine of a patient and when it does not, based on the routine activities defined in this patient's routine. Based on these definitions, this correspondence can be derived by a semantic reasoner through defining a recognized activity as an instance of either `RoutineActivityPrediction` or `NonRoutineActivityPrediction`. Instances of these classes, including the triples of which they are the subject, are the desired output of the semantic activity recognition service.

Listing 4 gives an example context description of a patient in the described use case scenario. In this context, there is a patient with ID 157 living in a service flat called the HomeLab. The morning routine of this patient consists of waking up followed by going to the toilet, showering, brushing teeth, eating breakfast and watching tv. The current location of this patient in the service flat is the bathroom. The description of the HomeLab service flat is given in the instantiated example modules `_Homelab` and `_HomelabWearable` of the DAHCC ontologies. The most relevant descriptions of these modules with respect to the running example are presented in Listing 5. As can be observed, for each sensor in the home, the observed properties are defined through the `measuresProperty` object property, and the analyzed entity is specified with the `analyseStateOf` property.

Listing 1: Overview of all prefixes used in the listings with semantic content in this document

```
# Activity Recognition ontology including DAHCC ontology modules
@prefix KBActivityRecognition: <https://dahcc.idlab.ugent.be/Ontology/ActivityRecognition/
    KBActivityRecognition/> .
@prefix ActivityRecognition: <https://dahcc.idlab.ugent.be/Ontology/ActivityRecognition/> .
@prefix MonitoredPerson: <https://dahcc.idlab.ugent.be/Ontology/MonitoredPerson/> .
@prefix SensorsAndActuators: <https://dahcc.idlab.ugent.be/Ontology/SensorsAndActuators/> .
@prefix SensorsAndWearables: <https://dahcc.idlab.ugent.be/Ontology/SensorsAndWearables/> .
@prefix Sensors: <https://dahcc.idlab.ugent.be/Ontology/Sensors/> .

# instances in use case scenario
@prefix : <http://divide.ilabt.imec.be/idlab.homelab/> .
@prefix patients: <http://divide.ilabt.imec.be/idlab.homelab/patients/> .
@prefix Homelab: <https://dahcc.idlab.ugent.be/Homelab/SensorsAndActuators/> .
@prefix HomelabWearable: <https://dahcc.idlab.ugent.be/Homelab/SensorsAndWearables/> .

# SAREF and extensions
@prefix saref-core: <https://saref.etsi.org/core/> .
@prefix saref4ehaw: <https://saref.etsi.org/saref4ehaw/> .
@prefix saref4bldg: <https://saref.etsi.org/saref4bldg/> .
@prefix saref4wear: <https://saref.etsi.org/saref4wear/> .

# other imports
@prefix time: <http://www.w3.org/2006/time#> .
@prefix eep: <https://w3id.org/eep#> .
```

```
# generic prefixes
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .

# definitions within DIVIDE
@prefix sd: <http://idlab.ugent.be/sensdesc#> .
@prefix sd-query: <http://idlab.ugent.be/sensdesc/query#> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
```

Listing 2: Example of how a knowledge-based activity recognition model with an activity rule for showering can be described through triples in the `KBActivityRecognition` ontology module. All definitions are listed in RDF/Turtle syntax. To improve readability, the `KBActivityRecognition:` prefix is replaced by the `:` prefix.

```
# define knowledge-based activity recognition model and its config with a specific rule
:KBActivityRecognitionModel rdf:type ActivityRecognition:ActivityRecognitionModel ;
    eep:implements :KBActivityRecognitionModelConfig1 .
:KBActivityRecognitionModelConfig1 rdf:type ActivityRecognition:Configuration ;
    :containsRule :showering_rule .

# define showering activity rule: detected by one specific condition
:showering_rule rdf:type :ActivityRule ;
    ActivityRecognition:forActivity [ rdf:type ActivityRecognition:Showering ] ;
    :hasCondition :showering_condition .

# define showering condition: relative humidity in the bathroom should be at least 57%
:showering_condition rdf:type :RegularThreshold ;
    :forProperty [ rdf:type SensorsAndActuators:RelativeHumidity ] ;
    Sensors:analyseStateOf [ rdf:type SensorsAndActuators:BathRoom ] ;
    :isMinimumThreshold "true"^^xsd:boolean ;
    saref-core:hasValue "57"^^xsd:float .

# define in system that conditions can be defined on relative humidity in a room
SensorsAndActuators:RelativeHumidity rdfs:subClassOf :ConditionableProperty .
SensorsAndActuators:Room rdfs:subClassOf :AnalyzableForCondition .
```

Listing 3: Example of how different subclass and equivalence relations between concepts are defined in the `KBActivityRecognition` ontology module, allowing a semantic reasoner to derive whether an activity prediction corresponds to a person's routine or not. To improve readability, all definitions are listed in Manchester syntax and the `KBActivityRecognition:` prefix is replaced by the `:` prefix.

```
:RoutineActivityPrediction SubClassOf: ActivityRecognition:ActivityPrediction
:NonRoutineActivityPrediction SubClassOf: ActivityRecognition:ActivityPrediction

:RoutineShoweringActivityPrediction SubClassOf: :RoutineActivityPrediction
:RoutineShoweringActivityPrediction EquivalentTo:
    :RoutineActivityPrediction and :ShoweringActivityPrediction
:RoutineShoweringActivityPrediction SubClassOf: :ShoweringActivityPrediction
:RoutineShoweringActivityPrediction EquivalentTo:
    :ShoweringActivityPrediction and
    (ActivityRecognition:activityPredictionMadeFor some :UserWithShoweringRoutine)
```

```
:NonRoutineShoweringActivityPrediction SubClassOf: :NonRoutineActivityPrediction
:NonRoutineShoweringActivityPrediction EquivalentTo:
    :NonRoutineActivityPrediction and :ShoweringActivityPrediction
:NonRoutineShoweringActivityPrediction SubClassOf: :ShoweringActivityPrediction
:NonRoutineShoweringActivityPrediction EquivalentTo:
    :ShoweringActivityPrediction and
    (ActivityRecognition:activityPredictionMadeFor some :UserWithoutShoweringRoutine)

:ShoweringActivityPrediction SubClassOf: ActivityRecognition:ActivityPrediction
:ShoweringActivityPrediction EquivalentTo:
    ActivityRecognition:ActivityPrediction and
    (ActivityRecognition:forActivity some ActivityRecognition:Showering)

:UserWithShoweringRoutine EquivalentTo:
    saref4ehaw:User and
    (MonitoredPerson:hasRoutine some (
        ActivityRecognition:Routine and
        (ActivityRecognition:consistsOf some ActivityRecognition:Showering)))
:UserWithoutShoweringRoutine EquivalentTo:
    saref4ehaw:User and
    (:doesNotHaveActivityInRoutine some ActivityRecognition:Showering)
```

Listing 4: Example context description of a patient in the eHealth use case scenario, in RDF/Turtle syntax. Only a selected set of context definitions are presented, some are omitted.

```
# patient with ID 157 lives in a smart home called the HomeLab
patients:patient157 rdf:type saref4ehaw:Patient ;
    MonitoredPerson:livesIn Homelab:homelab .

# patient has an Empatica wearable and location tag
patients:patient157 rdf:type saref4wear:Wearer .
HomelabWearable:empatica.E4.A03813 saref4wear:isLocatedOn patients:patient157 .
Homelab:AQURA_10_10_145_9 saref4wear:isLocatedOn patients:patient157 ;
    MonitoredPerson:hasLocation Homelab:homelab .

# patient has a morning routine consisting of a series of activities
patients:patient157 MonitoredPerson:hasRoutine :MorningRoutine_patient157 .
:MorningRoutine_patient157 rdf:type ActivityRecognition:MorningRoutine ;
    ActivityRecognition:consistsOf _:A1, _:A2, _:A3, _:A4, _:A5, _:A6 ;
    ActivityRecognition:nextActivity _:A1 .
_:A1 rdf:type ActivityRecognition:WakingUp ;
_:A2 rdf:type ActivityRecognition:Toileting .
_:A3 rdf:type ActivityRecognition:Showering .
_:A4 rdf:type ActivityRecognition:BrushingTeeth .
_:A5 rdf:type ActivityRecognition:EatingMeal .
_:A6 rdf:type ActivityRecognition:WatchingTVActively .
_:A1 ActivityRecognition:nextActivity _:A2 .
_:A2 ActivityRecognition:nextActivity _:A3 .
_:A3 ActivityRecognition:nextActivity _:A4 .
_:A4 ActivityRecognition:nextActivity _:A5 .
_:A5 ActivityRecognition:nextActivity _:A6 .

# patient is currently located in the bathroom
patients:patient157 MonitoredPerson:hasIndoorLocation Homelab:bathroom .
```

Listing 5: Example context description of the service flat of the example patient in the eHealth use case scenario, in Turtle syntax. Only a selected set of context definitions are presented, some are omitted.

```
# the HomeLab building consists of a bathroom on the first floor
Homelab:homelab rdf:type saref4bldg:Building .
Homelab:firstfloor rdf:type SensorsAndActuators:Floor ;
    saref4bldg:isSpaceOf Homelab:homelab .
Homelab:bathroom rdf:type SensorsAndActuators:BathRoom ;
    saref4bldg:isSpaceOf Homelab:firstfloor .

# the bathroom contains a Netatmo sensor that measures, among others, relative humidity
<https://dahcc.idlab.ugent.be/Homelab/SensorsAndActuators/70:ee:50:67:3e:78>
    rdf:type Homelab:Netatmo ;
    core:measuresProperty Homelab:org.dyamand.types.airquality.CO2 ,
                          Homelab:org.dyamand.types.common.AtmosphericPressure ,
                          Homelab:org.dyamand.types.common.Loudness ,
                          Homelab:org.dyamand.types.common.RelativeHumidity ,
                          Homelab:org.dyamand.types.common.Temperature ;
    Sensors:analyseStateOf Homelab:bathroom ;
    saref4bldg:isContainedIn Homelab:bathroom .
Homelab:Netatmo rdf:type owl:Class ;
    rdfs:subClassOf saref-core:Sensor .
Homelab:org.dyamand.types.common.RelativeHumidity
    rdf:type SensorsAndActuators:RelativeHumidity .

# the HomeLab consists of a location system that can detect the room in which
# the patient is located based on a tag system
Homelab:AQURA_10_10_145_9 core:consistsOf Homelab:AQURA_10_10_145_9.Tag .
Homelab:AQURA_10_10_145_9.Tag rdf:type saref4bldg:Sensor ;
    Sensors:analyseStateOf Homelab:AQURA_10_10_145_9 ;
    core:measuresProperty Homelab:org.dyamand.aqura.AquraLocationState_Protego_User .
Homelab:org.dyamand.aqura.AquraLocationState_Protego_User
    rdf:type SensorsAndActuators:Localisation .

# an Empatica wearable with multiple sensors is available in the HomeLab
HomelabWearable:empatica.E4.A03813 rdf:type SensorsAndWearables:Empatica ;
    core:consistsOf HomelabWearable:empatica.E4.A03813.Accelerometer ,
                    HomelabWearable:empatica.E4.A03813.BatteryLevelMeter ,
                    HomelabWearable:empatica.E4.A03813.EmpaticaTagButton ,
                    HomelabWearable:empatica.E4.A03813.GSRSensor ,
                    HomelabWearable:empatica.E4.A03813.OnWristDetector ,
                    HomelabWearable:empatica.E4.A03813.PPGSensor ,
                    HomelabWearable:empatica.E4.A03813.Thermopile .
```

4. Overview of the DIVIDE system

In Section 3, the general cascading reasoning architecture of the semantic system in the eHealth use case scenario is explained. This architecture is depicted in Figure 1. This section zooms in on DIVIDE, the architectural component responsible for managing the queries running on the local RSP engine components. It is the task of the DIVIDE system to ensure that these queries perform the relevant filtering given the current context, at any given time, for every RSP engine known to DIVIDE.

An overview of the DIVIDE system is shown in Figure 3. It shows a schematic overview of the different action steps, inputs and internal assets of DIVIDE. The figure makes a distinction between the two most important semantic aspects of the DIVIDE system from a methodological point of view: (i) the initialization of DIVIDE, involving the DIVIDE query parsing and ontology preprocessing steps, and (ii) and the core of DIVIDE which is the query derivation. The following two sections, Section 5 and Section 6, provide more information on this initialization and query derivation, respectively. Implementation details of DIVIDE are given in Section 7. Throughout the descrip-

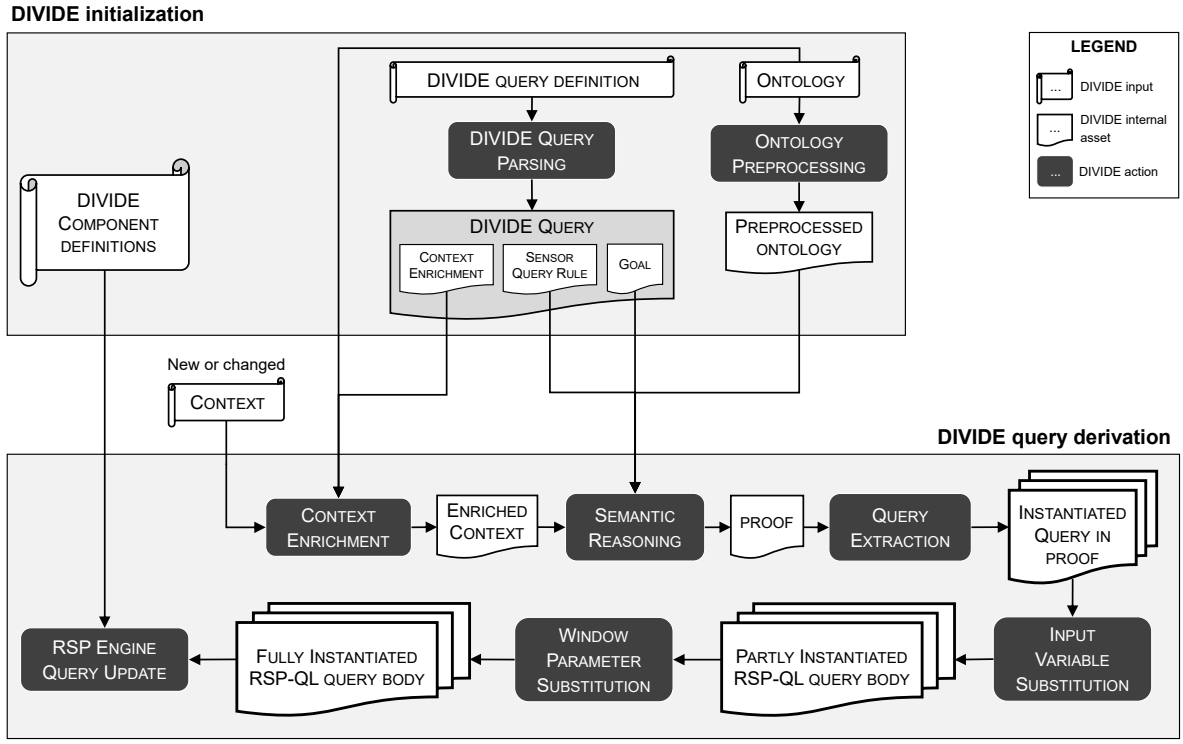


Figure 3. Schematic overview of the DIVIDE system. It shows all actions that can be performed by DIVIDE with their inputs and outputs. A distinction is made between internal assets and external inputs to the system. The overview is split up in the two major parts: the inputs, steps and assets of the DIVIDE initialization, and those of the DIVIDE query derivation.

tions of DIVIDE in these sections, the running eHealth use case example described in Section 3.1 is considered. The relevant ontology and contextual descriptions of this running example use case are listed in Section 3.3.

In terms of logic, DIVIDE works with the rule-based Notation3 Logic (N_3) [12]. The semantic reasoner used within DIVIDE should thus be a reasoner supporting N_3. Such a reasoner can reason within the OWL profile OWL 2 RL [8], which implies that a semantic system that uses DIVIDE in combination with an RSP engine is equivalent to a set-up involving a semantic OWL 2 RL reasoner. The reasoner should support the generation of all triples based on a set of input triples and rules, as well as generating a proof towards a certain goal rule. Such a proof contains the chain of all rules used by the reasoner to infer new triples based on its inputs, described in N_3 logic.

5. Initialization of the DIVIDE system

The core task of DIVIDE is the derivation and management of the queries running on the RSP engines of the semantic components in the system that are known to DIVIDE. To allow DIVIDE to effectively and efficiently perform the query derivation for one or more components upon context changes, different initialization steps are required. Three main steps can be distinguished from the upper part of the DIVIDE system overview in Figure 3: (i) parsing and initializing the DIVIDE queries, (ii) preprocessing the system ontology, and (iii) initializing the DIVIDE components. This section zooms in on each of these three initialization tasks.

5.1. Initialization of the DIVIDE queries

A DIVIDE query is a generic definition of an RSP query that should perform a real-time processing task on the RDF data streams generated by the different local components in the system. The goal of DIVIDE is to instantiate

this query in such a way that it can perform this task in a single query without performing semantic reasoning during its evaluation, i.e., by simple filtering on the RDF data streams. To this end, the internal representation of a DIVIDE query contains the following items: a goal, a sensor query rule with a generic query pattern, and a context enrichment. These three items are essential for correctly deriving the relevant queries with the semantic reasoner during the query derivation process. They will each be explained in detail in the first three subsections of this section.

As an end user of DIVIDE, it is not required to define a DIVIDE query according to its internal representation to properly initialize DIVIDE. Instead, the recommended way for an end user to define a DIVIDE query is by specifying an ordered collection of existing SPARQL queries that are applied in an existing (stream) reasoning system, or even from an already existing RSP-QL query. Through DIVIDE, this set of ordered queries will be replaced in the semantic platform by a single RSP query that performs a task that is its semantic equivalent. To enable this, DIVIDE contains a query parser. This parser converts such an external DIVIDE query definition into its internal representation. The goal of this approach is to make it easy for an end user to integrate DIVIDE into an existing semantic (stream) reasoning system, without having to know the details of how DIVIDE works. The parser and how to define such a DIVIDE query as an end user is explained in the final subsection of this section.

In the running example of the use case scenario described in Section 3, there is one RSP query that actively monitors the location of the patient in the home, and one query that detects when the patient is showering if he or she is located in the bathroom. This subsection will focus on the latter, which is an example of an actual activity detection query. Within DIVIDE, a generic DIVIDE query will be defined for each *type* of activity rule present in the system. This means that no dedicated DIVIDE query per activity should be defined, which would be too cumbersome and highly impractical in a real-world deployment. A rule type is defined as a specific combination of conditions and the type of value they are defined on, e.g., exact value, regular threshold or aggregation threshold. The exact observable properties that are being analyzed for each condition, and the entity types that they are analyzed for, are not part of the type definition and are hence later substituted into the generic DIVIDE query representing each rule type. For the showering rule, this means that the type is defined as follows: a rule with a single condition on a lower regular threshold that should be crossed. The fact the showering rule's condition is specified on the relative humidity of the bathroom is specific and not part of the generic definition. This implies that another rule of the same type, e.g., a rule for another activity with a single condition specifying that the electrical power consumption of the cooking stove should be crossed, would correspond to the same DIVIDE query. Hence, the detailed specific RSP queries corresponding to activity rules of the same type will all be derived from the same generic DIVIDE query. The generic DIVIDE query corresponding to the type of the showering activity rule will be used as the running example DIVIDE query in this section. Note that the running example will only focus on the detection of this activity *in* the patient's routine. Based on the definitions in the Activity Recognition ontology, a very similar DIVIDE query can be defined that is outputting the same detected activities when they are *not* in the patient's routine.

5.1.1. Goal

The goal of a DIVIDE query defines the semantic output that should be filtered by the resulting RSP query. This required query output is translated to a valid N_3 rule. How this translation exactly happens, will be explained in Section 5.1.4 about the DIVIDE query parser. This rule will be used in the DIVIDE query derivation to ensure that the resulting RSP query is filtering this required RSP query output.

For the generic query definition corresponding to the RSP query that detects the showering activity in the running example, the goal is specified in Listing 6. It is looking for any instance of a `RoutineActivityPrediction`.

Listing 6: Goal of the generic DIVIDE query detecting an ongoing activity in a patient's routine

```
{
    ?p rdf:type KBActivityRecognition:RoutineActivityPrediction ;
       ActivityRecognition:forActivity [ rdf:type ?activityType ] ;
       ActivityRecognition:activityPredictionMadeFor ?patient ;
       ActivityRecognition:predictedBy ?model ;
       saref-core:hasTimestamp ?t .
    ?activityType rdfs:subClassOf KBActivityRecognition:DetectableActivity .
}
=>
```

```
{
    _:p rdf:type KBActivityRecognition:RoutineActivityPrediction ;
        ActivityRecognition:forActivity [ rdf:type ?activityType ] ;
        ActivityRecognition:activityPredictionMadeFor ?patient ;
        ActivityRecognition:predictedBy ?model ;
        saref-core:hasTimestamp ?t .
} .
```

5.1.2. Sensor query rule with generic query pattern

The sensor query rule is the core of the DIVIDE query definition. As its name suggests, it is a complex N_3 rule. It defines the generic pattern of the RSP query, together with semantic information on when and how to instantiate it. Its usage by the semantic rule reasoner during the DIVIDE query generation defines whether or not this generic query should be instantiated given the involved context. If this is the case, the reasoner's output (proof) will contain all information on how exactly an RSP query should be instantiated from the generic query.

The formalism of the sensor query rule builds further on SENSdesc, which is the result of previous research [57]. This theoretical work was the first step in designing a format that describes an RSP query in a generic way that can be combined with formal reasoning to obtain the relevant queries that filter patterns of interest. By generalizing this format and integrating it into the DIVIDE system, it has become practically usable.

Each sensor query rule consists of three main parts: the relevant context in the rule's antecedence, and the generic query and ontology consequences defined in the rule's consequence. To explain the different parts, consider the DIVIDE query corresponding to the running example detecting the showering activity. Listing 7 defines the sensor query rule for the corresponding type of activity rule.

Listing 7: Sensor query rule of the generic DIVIDE query detecting an ongoing activity in a patient's routine, for an activity rule type with a single condition on a certain property of which a regular, observed value should cross a lower threshold

```
{
    ?model rdf:type ActivityRecognition:ActivityRecognitionModel ;
           <https://w3id.org/eep#implements> [
               rdf:type ActivityRecognition:Configuration ;
               KBActivityRecognition:containsRule ?a
           ] .
    ?a rdf:type KBActivityRecognition:ActivityRule ;
       ActivityRecognition:forActivity [ rdf:type ?activityType ] ;
       KBActivityRecognition:hasCondition [
           rdf:type KBActivityRecognition:RegularThreshold ;
           KBActivityRecognition:isMinimumThreshold "true"^^xsd:boolean ;
           saref-core:hasValue ?threshold ;
           Sensors:analyseStateOf [ rdf:type ?analyzed ] ;
           KBActivityRecognition:forProperty [ rdf:type ?prop ]
       ] .

    ?activityType rdfs:subClassOf KBActivityRecognition:DetectableActivity .

    ?sensor rdf:type saref-core:Device ;
            saref-core:measuresProperty ?prop_o ;
            Sensors:isRelevantTo ?room ;
            Sensors:analyseStateOf [ rdf:type ?analyzed ] .
    ?prop_o rdf:type ?prop .

    ?prop rdfs:subClassOf KBActivityRecognition:ConditionableProperty .
    ?analyzed rdfs:subClassOf KBActivityRecognition:AnalyzableForCondition .

    ?patient MonitoredPerson:hasIndoorLocation ?room .
}
```

```
=>
{
    _:q rdf:type sd:Query ;
        sd:pattern sd-query:pattern ;
        sd:inputVariables (("?sensor" ?sensor) ("?threshold" ?threshold)
                           ("?activityType" ?activityType) ("?patient" ?patient)
                           ("?model" ?model) ("?prop_o" ?prop_o)) ;
        sd:windowParameters (("?range" 30 time:seconds) ("?slide" 10 time:seconds)) .

    _:p rdf:type ActivityRecognition:ActivityPrediction ;
        ActivityRecognition:forActivity [ rdf:type ?activityType ] ;
        ActivityRecognition:activityPredictionMadeFor ?patient ;
        ActivityRecognition:predictedBy ?model ;
        saref-core:hasTimestamp _:t .
} .

sd-query:pattern
    rdf:type sd:QueryPattern ;
    sh:prefixes sd-query:prefixes-activity-showering ;
    sh:construct """
        CONSTRUCT {
            _:p a KBActivityRecognition:RoutineActivityPrediction ;
                ActivityRecognition:forActivity [ a ?activityType ] ;
                ActivityRecognition:activityPredictionMadeFor ?patient ;
                ActivityRecognition:predictedBy ?model ;
                saref-core:hasTimestamp ?now .
        }
        FROM NAMED WINDOW :win ON <http://protego.ilabt.imec.be/idlab.homelab>
                                  [RANGE ?{range} STEP ?{slide}]
        WHERE {
            BIND (NOW() as ?now)
            WINDOW :win {
                ?sensor saref-core:makesMeasurement [
                    saref-core:hasValue ?v ;
                    saref-core:hasTimestamp ?t ;
                    saref-core:relatesToProperty ?prop_o
                ] .
                FILTER (xsd:float(?v) > xsd:float(?threshold))
            }
        }
        ORDER BY DESC(?t)
        LIMIT 1
        """ .

sd-query:prefixes-activity-showering  rdf:type owl:Ontology ;
        sh:declare [ sh:prefix "xsd" ;
                 sh:namespace "http://www.w3.org/2001/XMLSchema#"^^xsd:anyURI ] ;
        sh:declare [ sh:prefix "saref-core" ;
                 sh:namespace "https://saref.etsi.org/core/"^^xsd:anyURI ] ;
        sh:declare [ sh:prefix "ActivityRecognition" ;
                 sh:namespace "https://dahcc.idlab.ugent.be/Ontology/ActivityRecognition/"^^xsd
                     :anyURI ] ;
        sh:declare [ sh:prefix "KBActivityRecognition" ;
                 sh:namespace "https://dahcc.idlab.ugent.be/Ontology/ActivityRecognition/
                     KBActivityRecognition/"^^xsd:anyURI ] .
```

Relevant context In the antecedence of the sensor query, the context in which the generic RSP query might become relevant is generically described. For each set of query variables for which the antecedence is valid, there is a chance

that the rule, instantiated with these query variables, will appear in the proof constructed by the semantic reasoner during the query derivation. If this is the case, the query will be instantiated for this set of variables.

In Listing 7, the rule's antecedence is described in lines 2–28. In short, it looks for an activity recognition model that contains a rule for a specific activity type that can be recognized by the system (lines 2–17). This rule should be of the generic type described before: it should consist of one condition that is defined on a certain property of which a regular, observed value should cross a lower threshold. This rule is specified on a certain analyzed entity. Moreover, the system should contain a sensor that is observing exactly that property for that entity on which the condition is specified (lines 19–26). This sensor should be relevant to the room that the patient is located in (line 28).

Generic query The generic query definition is contained inside the consequence of the sensor query rule. It consists of three main aspects: the generic query pattern, its input variables, and its static window parameters.

In Listing 7, the generic query definition is described in lines 32–37 and lines 46–82. More specifically, lines 32–33 and lines 46–82 define the generic query pattern, whereas lines 34–36 and line 37 define the input variables and static window parameters of the generic query, respectively.

The generic query pattern is a string representation of the actual RSP-QL query that will be the result of the DIVIDE query derivation. This pattern is however still generic: some of its query variables still need to be substituted by actual values (IRIs or literals) to obtain the correct and valid RSP-QL query. Similarly, the window parameters of the input stream windows of the RSP-QL query also need to be substituted.

The input variables that need to be substituted by the semantic reasoner in the generic query pattern are defined as a N_3 list. Every item in this list represents one input variable. This input variable is a list itself as well: the first item represents the string literal of the variable in the generic query pattern to be substituted, the second item is the query variable that should occur in the sensor query rule's antecedence so that it is instantiated by the semantic reasoner reasoner if the rule is applied in the proof during the query derivation.

Similarly, the definition of the window parameters is also a list of lists. Every item of the outer list is an inner list of three list items. The first item represents the string literal of the variable in *a* window definition of the generic query pattern. The second item can either be a query variable or literal defining the value of the window parameter. If this is a query variable, it will be substituted during the rule evaluation based on the matching value in the rule's antecedence, similarly to the input variables. The third item defines the unit of the value defined as the second item. This can either be a duration (`xsd:duration`; requires a valid string value such as `"PT5M"`) or one of seconds, minutes or hours (either `time:seconds`, `time:minutes` or `time:hours`; requires an integer value). Further explanation of how window parameters can be defined and how they are substituted during the query derivation follows in Section 5.1.4 and Section 6.5.

Inspecting the example in Listing 7 in further detail, the generic RSP-QL query pattern string is defined in lines 50–71. The query filters observations on the defined stream data window `:win` of a certain sensor `?sensor` with a value for the observed property `?prop_o` that is higher than a certain threshold `?threshold` (WHERE clause in lines 59–69). For every match of this pattern, output triples are constructed that represent an ongoing activity of type `?activityType` in the routine of a patient `?patient`, predicted by the activity recognition model `?model` (CONSTRUCT clause in lines 50–56). These six variables are exactly the six input variables as defined in lines 34–36: their values will be defined by the semantic reasoner by instantiating the second list items when the rule is applied during the reasoning process of the query derivation. Note that the triples in the CONSTRUCT clause (lines 51–55) are exactly those used in the consequence of the corresponding DIVIDE query goal shown in Listing 6. This will always be the case for CONSTRUCT queries, as this is essential for the DIVIDE query derivation to work correctly. This will be further explained in Section 6. Finally, note that the window parameter definitions specified in line 37 of Listing 7 define a window size of 30 seconds and a window sliding step of 10 seconds.

Ontology consequences The ontology consequences are the second main part of the sensor query rule's consequence. This part describes the *direct* effect of a query result in a real-time reasoning context, i.e., when a stream window of the generic RSP query would fulfill the pattern of the WHERE clause but no additional reasoning has been done (yet) to know the *indirect* consequences of this matching pattern. This is an essential aspect to understand: the whole purpose of DIVIDE is to derive queries that can make conclusions that are valid with the given context, through a single RSP-QL query without any reasoning involved. In a context without DIVIDE, these same *indirect* conclusions could only be made by performing an additional semantic reasoning step, based on the *direct*

conclusions that are directly known from the matching query pattern, without any additional reasoning involved. In other words, the triples defining the ontology consequences *can* be the same as the output of the generic RSP-QL query and thus the consequence of the rule representing the DIVIDE query's goal, but in practice, this will often *not* be the case: often, there will still be an additional semantic reasoning step required to see whether the ontology consequences actually imply the output of the generic RSP-QL query.

Applying this to the sensor query rule example in Listing 7, it can be noted that the ontology consequences in lines 39–43 indeed do not match the generic RSP-QL query output in lines 51–55. The *direct* consequences of a sensor observation matching the WHERE clause in lines 59–59 would be the fact that an ongoing activity of the given type is detected for the given patient. The *indirect* consequences represented by the definitions in the RSP-QL query output state that this is an activity *in the patient's routine*. Deriving these indirect consequences requires additional semantic reasoning on these ontology consequences, the patient's context and the domain knowledge defined in the Activity Recognition ontology.

Given these different parts of a sensor query rule, Section 6 will give further details using the running use case example on how the evaluation of such a sensor query rule by the DIVIDE query derivation can actually yield instantiated RSP-QL queries that are valid with the current patient's context.

5.1.3. Context enrichment

Prior to the start of the query derivation with the semantic reasoner, the current context can still be enriched by executing one or more SPARQL queries on this context. The context enrichment of a DIVIDE query consists of this ordered set of valid SPARQL queries, together with the context enrichment mode. If a DIVIDE query has no SPARQL queries in its context enrichment, the mode can be left undefined. There are four different modes, defined by the combination of the value of two boolean properties:

- The first property defines whether or not the TBox (ontology) triples should be added to the data model on which the context-enriching queries are executed. If not, the context-enriching queries are executed on a data model containing only the context (ABox) triples.
- The second property defines whether or not (incremental) rule-based reasoning should be applied on the model prior to the evaluation of each context-enriching query, using the rules extracted from the ontology definitions.

For the running example, no context-enriching queries are part of the DIVIDE query definition. However, when discussing the query parser in Section 5.1.4, an example for a related DIVIDE query will be given.

It is important to note that context-enriching queries are not only used to add general context to the model, but also for the dynamic window parameter substitution as will be explained in Section 6.5.

5.1.4. DIVIDE query parser

The recommended way to define a DIVIDE query is to define an ordered collection of existing SPARQL queries that are already applied in an existing (stream) reasoning system, or through an already existing RSP-QL query. This query definition will then be converted to the internal representation of a DIVIDE query by a dedicated component: the DIVIDE query parser. This internal representation is then used during the DIVIDE query derivation to convert the original set of SPARQL queries to a single RSP-QL query that is its semantic equivalent.

To ensure that an ordered set of existing SPARQL queries behaves semantically equivalent when used as a DIVIDE query in a system involving DIVIDE and a local RSP engine in the cascading system architecture, it is important to define what is considered the semantically equivalent task of evaluating this set of queries in a standard rule-based stream reasoning system, e.g., in a stream reasoner that uses a semantic reasoner such as RDFox or Apache Jena. Moreover, restrictions on the set of ordered SPARQL queries should be defined in order for this set of queries to be imported as a DIVIDE query. DIVIDE considers its equivalent regular (stream) reasoning system as a semantic reasoning engine in which this set of queries is executed sequentially on a data model containing the ontology (TBox) triples and rules, context (ABox) triples, and triples representing the sensor observations in the data stream, either extracted directly from the stream or via data windows. Each query in the ordered collection, except for the final one, should be a CONSTRUCT query, and its outputs are added to the data model on which (incremental) rule reasoning is applied before the next query in the chain is executed. To use this set of queries to define a single DIVIDE query, it is important that there is exactly one query in the chain that reads from the chain

of sensor observations, i.e., the triples that will be sent to the stream(s) that are defined as data window(s) in the resulting RSP-QL query. This query is called the *stream query*. In some cases, this query will be the first in the chain. If this is not the case, any preceding queries are defined as *context-enriching queries* in the DIVIDE query definition. The final query in the chain is called the *final query* in DIVIDE. This is the query that constructs the final result. If there are any other queries executed between the stream and final query, these are described as *intermediate queries*. In some cases, there will even be no final query, if no additional query outputs based on the sensor stream data are required as reasoning input to obtain the final results of the query chain. This is allowed in DIVIDE.

As introduced in the previous paragraph, the definition of a DIVIDE query as an ordered set of SPARQL queries includes a context enrichment with zero or more context-enriching queries, exactly one stream query, zero or more intermediate queries, and either no or exactly one final query. Besides these queries, such a DIVIDE query definition also includes a set of stream windows (required), a solution modifier (optional), and a variable mapping from stream query to final query (optional). The remainder of this subsection will discuss these different inputs in this DIVIDE query definition, and present this definition for the running use case example. This will demonstrate how this definition is translated by the DIVIDE query parser to the internal representation of this DIVIDE query discussed in the previous subsections, giving more insight into how both definitions relate.

Stream query The stream query has to be a CONSTRUCT query, unless it is the final query in the chain and no intermediate queries and final query are present; in that case, other SPARQL query forms are also allowed. A stream query should be a valid SPARQL query. Importantly, the WHERE clause conditions should be part of named graphs defined as data inputs with a `FROM` clause, except for special SPARQL constructs such as a FILTER or BIND clause: these *should* be put outside of the named graph patterns. To obtain a stream query that fulfills these requirements, an existing SPARQL query might have to be slightly modified. The IRIs used for the named graphs are important for DIVIDE to distinguish which data is considered as part of the context, and which data will be put on the data stream. For the data streams, the named graph IRI should reflect the stream IRI. This stream IRI should also be defined as a stream window.

The example stream query for the running use case example is shown in Listing 8.

Listing 8: Content of the stream query in the end-user definition of the DIVIDE query detecting an ongoing activity in a patient's routine, for an activity rule type with a single condition on a certain property of which a regular, observed value should cross a lower threshold. The full definition is given in the main text.

```
# stream-query.sparql
CONSTRUCT {
    _:p rdf:type ActivityRecognition:ActivityPrediction ;
        ActivityRecognition:forActivity [ rdf:type ?activityType ] ;
        ActivityRecognition:activityPredictionMadeFor ?patient ;
        ActivityRecognition:predictedBy ?model ;
        saref-core:hasTimestamp ?now .
}
FROM NAMED <http://protego.ilabt.imec.be/idlab.homelab>
FROM NAMED <http://protego.ilabt.imec.be/context>
WHERE {
    BIND (NOW() as ?now)

    GRAPH <http://protego.ilabt.imec.be/idlab.homelab> {
        ?sensor saref-core:makesMeasurement [
            saref-core:hasValue ?v ;
            saref-core:hasTimestamp ?t ;
            saref-core:relatesToProperty ?prop_o
        ] .
    }

    GRAPH <http://protego.ilabt.imec.be/context> {
        ?model rdf:type ActivityRecognition:ActivityRecognitionModel ;
               <https://w3id.org/eep#implements> [
```

```
                    rdf:type ActivityRecognition:Configuration ;
                    KBActivityRecognition:containsRule ?a
                ] .
        ?a rdf:type KBActivityRecognition:ActivityRule ;
           ActivityRecognition:forActivity [ rdf:type ?activityType ] ;
           KBActivityRecognition:hasCondition [
               rdf:type KBActivityRecognition:RegularThreshold ;
               KBActivityRecognition:isMinimumThreshold "true"^^xsd:boolean ;
               saref-core:hasValue ?threshold ;
               Sensors:analyseStateOf [ rdf:type ?analyzed ] ;
               KBActivityRecognition:forProperty [ rdf:type ?prop ]
           ] .

        ?activityType rdfs:subClassOf KBActivityRecognition:DetectableActivity .
    }

    FILTER (xsd:float(?v) > xsd:float(?threshold))

    GRAPH <http://protego.ilabt.imec.be/context> {
        ?sensor rdf:type saref-core:Device ;
                saref-core:measuresProperty ?prop_o ;
                Sensors:isRelevantTo ?room ;
                Sensors:analyseStateOf [ rdf:type ?analyzed ] .
        ?prop_o rdf:type ?prop .

        ?prop rdfs:subClassOf KBActivityRecognition:ConditionableProperty .
        ?analyzed rdfs:subClassOf KBActivityRecognition:AnalyzableForCondition .

        ?patient MonitoredPerson:hasIndoorLocation ?room .
    }
}
```

Final query A final query is optional. If it exists, it can either be a CONSTRUCT, SELECT, DESCRIBE or ASK query. A final query cannot have an input graph defined with `FROM`, have no special SPARQL constructs in the WHERE clause and have no solution modifier.

The example final query for the running use case example is shown in Listing 9.

Listing 9: Content of the final query in the end-user definition of the DIVIDE query detecting an ongoing activity in a patient's routine, for an activity rule type with a single condition on a certain property of which a regular, observed value should cross a lower threshold. The full definition is given in the main text.

```
# final-query.sparql
CONSTRUCT {
    _:p rdf:type KBActivityRecognition:RoutineActivityPrediction ;
        ActivityRecognition:forActivity [ rdf:type ?activityType ] ;
        ActivityRecognition:activityPredictionMadeFor ?patient ;
        ActivityRecognition:predictedBy ?model ;
        saref-core:hasTimestamp ?t .
}
WHERE {
    ?p rdf:type KBActivityRecognition:RoutineActivityPrediction ;
       ActivityRecognition:forActivity [ rdf:type ?activityType ] ;
       ActivityRecognition:activityPredictionMadeFor ?patient ;
       ActivityRecognition:predictedBy ?model ;
       saref-core:hasTimestamp ?t .
    ?activityType rdfs:subClassOf KBActivityRecognition:DetectableActivity .
}
```

Intermediate queries Intermediate queries should be defined as an ordered list of zero or more SPARQL queries. It is possible that no intermediate queries are defined. An intermediate query should be valid CONSTRUCT query without any special SPARQL constructs in the WHERE clause or a solution modifier, and cannot have an input graph defined with `FROM`.

There are no intermediate queries for the running use case example.

Stream windows Each stream window that should be included as an input window in the resulting RSP-QL query, i.e., each window defined on a sensor data stream, should be explicitly defined. It consists of a stream IRI, a window definition, and a set of default window parameter values.

The stream IRI represents the IRI of the data stream. This IRI should exactly match the name of a named graph defined in the stream query. The window definition defines the specification of how the windows are created on the stream. *If* the user wants to define variable window parameters, named variables should be inserted into the places that will be instantiated during the query derivation. In DIVIDE, two types of variable window parameters exist: static and dynamic window parameters. Static window parameters *might* be substituted similarly to an input variable during the DIVIDE query derivation. Hence, the variable name of this window parameter should appear in the WHERE clause of the stream query, in a named graph pattern of which the graph name does not refer to a stream IRI. This will imply that the variable name ends up in the antecedence of the sensor query rule of this DIVIDE query, ensuring it can be substituted as a regular input variable. A dynamic window parameter is defined via a context-enriching query: it will be substituted if a value for its corresponding variable is defined in the output of a context-enriching query. It is required to define a default value for a dynamic window parameter if its variable is *not* also used in the definition of a static window parameter. This default value is then used in case no context-enriching query yields a value for the dynamic window parameter variable. To achieve this, for each such variable that receives a default value, DIVIDE will define a static window parameter itself of which the value is set to the specified default instead of having the semantic reasoner substitute its value during the query derivation. For variables that are already defined as static window parameters by the end user, no default value should be defined, since their (default) values will be derived by the semantic reasoner during the query derivation. Detailed information on the window parameter substitution and the difference between static and dynamic window parameters, is provided in Section 6.5.

For the running example, the DIVIDE query definition contains one stream window with the following properties:

- Stream IRI: `http://protego.ilabt.imec.be/idlab.homelab`
- Window definition: `RANGE PT?rangeS STEP PT?slideS`
- Default window parameter values: `?range` has a default value of 30, `?slide` has default value 10

This window definition contains the two variable window parameters `?range` and `slide`. The definition of a default value for both window parameters implies that they are dynamic window parameters. This can be confirmed by observing their absence in the WHERE clause of the stream query in Listing 8.

Solution modifier If the resulting RSP-QL query should have a SPARQL solution modifier, this can be included in the DIVIDE query definition. If this solution modifier contains any unbound variable names, it is important that they are all defined in a named graph of the stream query's WHERE clause of which the graph name represents a stream window's stream IRI.

For the running use case example, the DIVIDE query definition contains the following solution modifier: `ORDER BY DESC(?t) LIMIT 1`. This solution modifier contains the unbound variable name `?t`, which is allowed since it is present in a stream graph of the stream query (Listing 8, line 17).

Variable mapping of stream to final query If a final query is specified, it often occurs that certain query variables in both the stream and final query actually refer to the same individuals. To make sure that DIVIDE parses the DIVIDE query input correctly, the mapping of these variable names should be explicitly defined. This is a manual required step. Often, they will have the same variable names, making this mapping trivial.

For the running use case example, the mapping consists of the stream query variables `?activityType`, `?patient`, and `?model`, which are all mapped to the same variable name in the final query. In addition, it contains the mapping of the variable `?now` in the stream query to the variable `?t` in the final query. The reason for this final mapping becomes clear when inspecting the corresponding generic RSP-QL query pattern in the internal

representation of this DIVIDE query (Listing 7, lines 50–71): the literal object of the `hasTimestamp` property in the resulting query output indeed corresponds to the `?now` variable.

Context enrichment The context enrichment in this DIVIDE query definition is identical to its internal representation in a DIVIDE query: it consists of an ordered set of SPARQL queries and a context enrichment mode.

For the running use case example, the context enrichment consists of an empty set of queries, since the stream query is the first query in the ordered set of SPARQL queries used in the stream reasoning system. Hence, the context enrichment mode is not defined.

However, as an example of a context-enriching query, consider the DIVIDE query equivalent to the DIVIDE query corresponding to the running example that is detecting an ongoing activity *not* in the patient's routine, instead of in-routine activities. The output of this DIVIDE query should contain instances of the class `NonRoutineActivityPrediction`. From the ontology definitions in Listing 3, it follows that the derivation of such instances requires the association between patient and activity with the `doesNotHaveActivityInRoutine` property for every activity type that is not in the patient's routine. However, such definitions are not present in the regular patient context described in Listing 4. Hence, in an existing stream reasoning system applying the DIVIDE query's equivalent as a set of ordered SPARQL queries, the evaluation of the stream query would be preceded by an additional query that is enriching the context with this information. In the DIVIDE query definition, this first SPARQL query would be defined as a context-enriching query. It is presented in Listing 10 for illustration purposes.

Listing 10: Example of a context-enriching query in the definition of the DIVIDE query detecting an ongoing activity that is *not* in a patient's routine. It enriches the context with information about all activity types that are not part of the patient's routines.

```
CONSTRUCT {
    ?p KBActivityRecognition:doesNotHaveActivityInRoutine [ rdf:type ?activityType ] .
}
WHERE {
    ?p rdf:type saref4ehaw:Patient .

    ?activityType rdf:type owl:Class ;
                  rdfs:subClassOf KBActivityRecognition:DetectableActivity .

    FILTER NOT EXISTS {
        ?p MonitoredPerson:hasRoutine ?routine .
        ?routine ActivityRecognition:consistsOf ?routineActivity .
        ?routineActivity rdf:type ?activityType .
    }
}
```

Parsing the end-user definition of a DIVIDE query to its internal representation Given the aforementioned items in the definition of a DIVIDE query, the DIVIDE query parser is able to construct the goal, sensor query rule and context enrichment of this DIVIDE query. For the context enrichment, no parsing is required. For the goal and sensor query rule, the parser needs to combine the different inputs.

The goal of the DIVIDE query is directly constructed from the final query, or the stream query if no final query is available. If a final query exists, the construction of the goal depends on the query form of this final query. If it is a CONSTRUCT query, the content of the WHERE clause is put in the antecedence of the goal, while the content of the CONSTRUCT clause represents the goal's consequence. For any other query form, the WHERE clause of the final query is used for both the goal's antecedence and consequence. If no final query is present, the goal is constructed in a similar way from the stream query if this is a CONSTRUCT query. However, if the stream query has another query form, the resulting SELECT, ASK or DESCRIBE clause is converted to a CONSTRUCT clause containing all unbound variables occurring in this clause. This clause is then used for both the antecedence and consequence of the DIVIDE query's goal.

The sensor query rule is the most complex part to construct. In the standard case, disregarding any exceptions, the antecedence of the rule is extracted from the WHERE clause of the stream query, more specifically from all named graph patterns of which the graph name does *not* refer to a stream IRI. The ontology consequences in the consequence of the sensor query rule are copied from the stream query's output. The generic RSP-QL query pattern is constructed from different parts. Its resulting clause (CONSTRUCT, SELECT, ASK or WHERE clause) is directly copied from the result clause of the final query (or the stream query if no final query is present), its input stream window definitions are constructed using the defined stream windows, and its WHERE clause consists of the remainder of the stream query's WHERE clause (i.e., the content of the named graph patterns of which the graph name *does* refer to a stream IRI, and the special SPARQL patterns that are not put inside a named graph pattern). If a solution modifier is specified, this solution modifier is appended to the generic RSP-QL query pattern. To define the input variables and window parameters of the sensor query rule, the combination of stream query, final query and the variable mapping between both is analyzed. Any intermediate queries are converted to additional semantic rules that are appended to the main sensor query rule.

Finally, it is worth noting that a DIVIDE query can alternatively also be defined through an existing RSP-QL query. Such a definition is quite similar to the definition described above, with a few differences. The main difference is that by definition, no intermediate and final queries will be present since the original system already uses RDF stream processing and individual RSP-QL queries. This means no variable mapping should be defined either. Hence, this definition is typically more simple than the definition of a DIVIDE query as a set of SPARQL queries.

5.2. Initialization of the DIVIDE ontology

To properly perform the query derivation, an ontology should be specified as input to DIVIDE by the end user. During initialization, this ontology will be loaded into the system. By definition, this ontology is considered not to change often during the system's lifetime, in contrast with the context data. Therefore, the ontology should be preprocessed by the semantic reasoner wherever possible. This will speed up the actual query derivation process, since it avoids that the full ontology is loaded and processed every time the DIVIDE query derivation is triggered. To what extent the ontology can be preprocessed depends on the semantic reasoner used.

For the running example, the ontology that will be preprocessed consists of the triples and axioms in the `KBActivityRecognition` module of the Activity Recognition ontology, including the definitions in all its imported ontologies. This includes the definitions in Listing 2 and Listing 3.

5.3. Initialization of the DIVIDE components

To properly initialize DIVIDE, it should have knowledge about the components it is responsible for. A component is defined as an entity in the IoT network on which a single RSP engine runs. For each DIVIDE component, the following information should be specified by an end user for the correct initialization of DIVIDE:

- The name of the graph (ABox) pattern in the knowledge base that contains the context specific for the entity that this component's RSP engine is responsible for. A typical example in the eHealth scenario is a graph pattern of a specific patient, containing all patient information.
- A list of any additional graph patterns in the knowledge base that contain context relevant to the entity that this component's RSP engine is responsible for. An example is generic information on the layout of the patient's environment in which his or her smart home is situated. Such context information is relevant to multiple components, and should therefore be stored in separate graphs in the knowledge base.
- The type of the RSP engine of this component (e.g., C-SPARQL).
- The base URL of the RSP engine's server API. This API should support registering and unregistering RSP queries, and pausing and restarting an RSP stream. It will be used during the DIVIDE query derivation to manage the active queries on the RSP engine.

Upon initialization, all component information is processed and saved by DIVIDE. For every graph (ABox) pattern associated to at least one component, DIVIDE should actively monitor for any updates to this ABox in the knowledge base during its lifetime, so that it can trigger the query derivation for the relevant components when any of these graphs is updated.

6. DIVIDE query derivation

Whenever DIVIDE is alerted of a context (ABox) change in the knowledge base, the DIVIDE query derivation is triggered. Based on the name of the updated ABox graph and the components known by the system, DIVIDE knows for which components, i.e., RSP engines, the query derivation process should be triggered. This process can execute independently for each RSP engine and can therefore be parallelized by DIVIDE. A context change requires the query derivation to be triggered for every DIVIDE query known to the system. Executing the query derivation steps can happen independently for each DIVIDE query and can therefore also be parallelized for the different DIVIDE queries. Hence, this section will focus on the query derivation task for a single RSP engine and a single DIVIDE query, that is triggered by a context change.

In the running example, two RSP queries may be deployed on the local component's RSP engine: a query that monitors the location of the patient in the home, and a query that detects when the patient is showering. The first query will always be running, while the second query is only active when the patient is located in the bathroom. Each of those two queries corresponds to one DIVIDE query. The generic DIVIDE query that allows to perform the showering activity detection has been used to explain the DIVIDE query initialization in Section 5.1, and will also be used as the running example in this section to illustrate the query derivation process in this section. It is important to specify what is considered as the context for the given RSP engine of which a change triggers the query derivation for the two DIVIDE queries. The main part of this context data, collected in the main ABox graph for this component in the knowledge base, consists of all information about the patient and the smart home, including the examples in Listing 4 and 5. Moreover, the output of all RSP queries derived from both DIVIDE queries is also part of the context as an additional graph in the knowledge base: the location in the house defines the relevant activity detection queries, while the detection of routine or non-routine activities influences the desired window parameters of the location monitoring query. For the latter, a non-routine activity detection requires a higher query execution frequency while a detected routine activity allows for a lower frequency.

The DIVIDE query derivation task for one RSP engine and one DIVIDE query consists of several steps, which are executed sequentially: (i) enriching the context, (ii) semantic reasoning on the enriched context to construct a proof containing the details of derived queries and how to instantiate them, (iii) extracting these derived queries from the proof, (iv) substituting the instantiated input variables in the generic RSP-QL query pattern for every derived query, (v) substituting the window parameters in a similar way, and (vi) finally updating the active RSP queries on the corresponding RSP engine. The input of the query derivation is the updated context, which consists of the set of triples in the context graph(s) of the knowledge base that are associated to the given component's RSP engine. Before starting the query derivation, DIVIDE constructs this data model from the knowledge base. In the following subsections, the DIVIDE query derivation action steps are further detailed. Figure 3 shows a schematic overview of these steps on the bottom part. For every step, the inputs and and outputs are detailed on the figure.

6.1. Context enrichment

Prior to actually deriving the RSP queries for the given DIVIDE query via semantic reasoning based on the context data model representing the input of the query derivation, this context can still be enriched. This is done by executing the ordered set of context-enriching queries associated to the DIVIDE query. As explained in Section 5.1, the context enrichment mode in the DIVIDE query definition defines whether these queries are either executed on a data model with only the context (ABox) triples or a model containing both the context and ontology (TBox) triples, and whether or not rule-based reasoning with the ontology axioms is applied on the model prior to the query evaluation. The ordered execution of the context-enriching queries can be done with any SPARQL query engine. In case reasoning is applied prior to the query evaluation, a semantic reasoner is required. The result of this context enrichment step is a data model containing the original context triples and all triples in the output of any of the context-enriching queries. If no context-enriching queries are defined for the DIVIDE query, this step skipped and the original context model is forwarded to the next step.

The generic DIVIDE query corresponding to the running example of detecting the showering activity in a patient's home, does not contain any context-enriching query, as explained when discussing the details of this DIVIDE query

in Section 5.1. Hence, when the query derivation is triggered for this DIVIDE query, the updated context will directly be sent to the input of the next step.

Note that context-enriching queries can also define dynamic window parameters to be used in the window parameter substitution. Section 6.5 will explain how such dynamic window parameters should be specified in the output of context-enriching queries, and how they will be substituted into the derived queries.

6.2. Semantic reasoning to derive queries

Starting from the context data model in the output of the context enrichment step, the semantic reasoner used within DIVIDE is run to perform the actual query derivation. This way, the reasoner will define whether the DIVIDE query should be initialized for the given context, and if so, with what values the input variables static window parameters, as defined in the query's sensor query rule consequence, should be substituted in the generic RSP-QL query pattern of the DIVIDE query.

The inputs of the semantic reasoner in this step consist of the preprocessed ontology including all ontology triples and rules extracted from the ontology axioms, the enriched context triples, the sensor query rule and the goal of the DIVIDE query. Given these inputs, the reasoner performs semantic reasoning to construct rule chains in which the goal of the DIVIDE query is the final rule applied. It does this for all possible instantiations of the goal, i.e., for all sets of instantiated query variables appearing in the goal's rule. For every such instantiation, a (partially) different rule chain will be followed. The output of this semantic reasoning step is a proof containing all rule chains that lead to an instantiation of the goal rule.

To allow the semantic reasoner to construct a rule chain that starts from the context and ends with the goal rule, the evaluation of the sensor query rule is crucial. If the inputs of the semantic reasoner allow it to derive the set of triples in the antecedence of the sensor query rule for a certain set of query variables, the required conditions in order for this sensor query rule to be evaluated are fulfilled. This means that the rule *can* be evaluated for this set of variables. However, the semantic reasoner *will* only actually evaluate the rule and include it in the rule chain, *if* the triples in the consequence of the sensor query rule (and more specifically, the part with the ontology consequences) allow the semantic reasoner to derive the antecedence of the goal rule. If this is not the case, the sensor query rule will not help the semantic reasoner in constructing a rule chain where the goal is the last rule applied, and the rule will therefore not be applied for the given set of sensor query rule variables. However, if the triples specifying the ontology consequences in the consequence of the sensor query rule lead either directly (i.e., without semantic reasoning) or indirectly (i.e., after rule-based semantic reasoning) to the antecedence of the goal, the semantic reasoner *will* evaluate the sensor query rule for this set of query variables. This means that this particular instantiation of the sensor query rule will appear in a rule chain of the proof constructed by the semantic reasoner. This automatically implies that the generic RSP-QL query of this DIVIDE query should be instantiated for this set of query variables, more specifically using those query variables of this set that are present in the list of input variables or window parameters of the sensor query rule's consequence. If this would not be true, the semantic reasoner would have never used this instantiation in its proof towards the instantiation of the goal rule.

The fact that this process works can be explained by considering the translation of the ordered set of SPARQL queries in the end-user DIVIDE query definition into the internal representation of the DIVIDE query by the DIVIDE query parser. If the original stream query in the SPARQL input would yield a query result, there is a possibility that the final query's WHERE clause has a matching pattern, and thus an output, either directly or indirectly after semantic reasoning. This is equivalent to the potential evaluation of the sensor query rule in the proof, depending on whether the rule's consequence directly or indirectly leads to a matching antecedence of the goal rule.

Reconsider the running use case example with the generic DIVIDE query that detects any ongoing activity in the patient's home that is defined as an activity rule with a single condition on a certain property of which a regular observed value should cross a lower threshold, as discussed in Section 5.1. If the query derivation would be executed for this DIVIDE query on the running example's context described in the introduction of this section, with the showering activity detection rule in Listing 2 defined in the preprocessed ontology, this DIVIDE query would be instantiated once for the showering activity in the proof constructed by the semantic reasoner, *if* the current location of the patient is the bathroom. The step in the rule chain of the reasoner's proof in which the corresponding sensor query rule of Listing 7 is instantiated, is shown as an illustration in Listing 11. This proof shows that the relative

humidity sensor with the given ID can detect the showering activity for patient with ID 157 if its value is 57 or higher. This is true for the current context, since patient 157 is located in the same room as this sensor, i.e., the bathroom. If the patient's location would be different, or showering would not be defined in the routine of the patient with ID 157, the proof would not contain this sensor query rule instantiation.

Listing 11: Part of the proof (one step containing one evaluated rule in the rule chain) constructed by the semantic reasoner used in DIVIDE during the DIVIDE query derivation for the running use example with the generic DIVIDE query that detects any ongoing activity in the patient's home that is defined as an activity rule with a single condition on a certain property of which a regular observed value should cross a lower threshold. The input of the query derivation contains the context of the running example (patient and smart home information, including the examples in Listing 4 and 5, the patient's location in the home, and any detected ongoing routine and non-routine activities) and the preprocessed ontology that includes the showering activity detection rule described in Listing 2. This part shows how the sensor query rule in Listing 7 is instantiated in the semantic reasoner's proof. `[...]` is a placeholder for omitted parts that are not of interest.

```
@prefix r: <http://www.w3.org/2000/10/swap/reason#>.

<#lemma3> a r:Inference;
  r:gives {
    _:sk_0 a sd:Query.
    _:sk_0 sd:pattern sd-query:pattern.
    _:sk_0 sd:inputVariables (
      ("?sensor" <https://dahcc.idlab.ugent.be/Homelab/SensorsAndActuators/70:ee:50:67:3e:78>)
      ("?threshold" "57"^^xsd:float)
      ("?activityType" ActivityRecognition:Showering)
      ("?patient" patients:patient157)
      ("?model" :KBActivityRecognitionModel)
      ("?prop_o" <https://dahcc.idlab.ugent.be/Homelab/SensorsAndActuators/org.dyamand.types.
          common.RelativeHumidity>)
    ).
    _:sk_0 sd:windowParameters (("?range" 30 time:seconds) ("?slide" 10 time:seconds)).
    _:sk_1 a ActivityRecognition:ActivityPrediction.
    _:sk_1 ActivityRecognition:forActivity _:sk_2.
    _:sk_2 a ActivityRecognition:Showering.
    _:sk_1 ActivityRecognition:activityPredictionMadeFor patients:patient157.
    _:sk_1 ActivityRecognition:predictedBy :KBActivityRecognitionModel.
    _:sk_1 saref-core:hasTimestamp _:sk_3.
  };
  r:evidence (
    <#lemma8>
    [...]
    <#lemma31>
  );
  r:binding [ r:variable [ n3:uri "http://josd.github.io/var#x_0"];
              r:boundTo [ n3:uri "https://dahcc.idlab.ugent.be/Ontology/ActivityRecognition/
                  KBActivityRecognition/KBActivityRecognitionModel"]];
  [...]
  r:binding [ r:variable [ n3:uri "http://josd.github.io/var#x_19"];
              r:boundTo [ a r:Existential; n3:nodeId "_:sk_3"]];
  r:rule <#lemma32>.
```

6.3. Query extraction

The output of the semantic reasoning step is a proof constructed by the semantic reasoner. This proof *can* contain instantiations of the sensor query rule. If not, the proof will be empty, since this means that the semantic reasoner has not found any rule chain that leads to an instantiation of the goal rule. Every sensor query rule instantiation in

the semantic reasoner's proof contains the list of input variables and window parameters that need to be substituted into the generic RSP-QL query of the considered DIVIDE query. In the query extraction step, DIVIDE will extract these definitions from every sensor query rule instantiation in the proof. Hence, the output of this step is a set of zero, one or more extracted queries. Note that even if the output contains no extracted queries, the following steps of the DIVIDE query derivation still need to be executed.

The query extraction happens through two forward reasoning steps with the semantic reasoner used in DIVIDE. The outputs of both reasoning steps are combined to construct the output of the query extraction. The first reasoning step extracts the relevant content from the sensor query rule instantiations in the proof returned by the semantic reasoner in the previous query derivation step. For each instantiation, this content includes the instantiated input variables and window parameters, as well as a reference to the query pattern in which they need to be substituted. Hence, the result basically is a cleaned version of the proof steps with a query rule instantiation. Note that the window parameters occurring in the query rule instantiations are considered as static window parameters, as explained before. The second forward reasoning step of the query extraction retrieves any defined window parameters from the enriched context that are associated to the instantiated RSP-QL query pattern. Such window parameters have been added to the enriched context during the context enrichment step. They are used as dynamic window parameters during the window parameter substitution step further in the query derivation process.

For the running use case example, the output of the extraction for the proof step in Listing 11 is presented in lines 4–15 of Listing 12. Line 18 of this listing presents the output of the second step. For this query example, there are no dynamic window parameters, which defaults the output of this second query extraction step to an empty list.

Listing 12: Output of the query extraction step of the DIVIDE query derivation, performed for the running example on the proof with a single sensor query rule instantiation presented in the proof step of Listing 11. The extraction of the dynamic window parameters (line 18) is done on the enriched context outputted by the context enrichment step.

```
1  @prefix : <file:///home/divide/.divide/query-derivation/10-10-129-31-8175-/activity-ongoing
       /20211220_194006/proof.n3#>.
2
3  # output of the first reasoning step of the query extraction
4  :lemma3 a sd:Query.
5  :lemma3 sd:inputVariables (
6      ("?sensor" <https://dahcc.idlab.ugent.be/Homelab/SensorsAndActuators/70:ee:50:67:3e:78>)
7      ("?threshold" "57"^^xsd:float)
8      ("?activityType" ActivityRecognition:Showering)
9      ("?patient" patients:patient157)
10     ("?model" :KBActivityRecognitionModel)
11     ("?prop_o" <https://dahcc.idlab.ugent.be/Homelab/SensorsAndActuators/org.dyamand.types.
           common.RelativeHumidity>)
12 ).
13 :lemma3 sd:staticWindowParameters (("?range" 30 time:seconds)
14                                   ("?slide" 10 time:seconds)).
15 :lemma3 sd:pattern sd-query:pattern.
16
17 # output of the second reasoning step of the query extraction
18 :lemma3 sd:dynamicWindowParameters ().
```

6.4. Input variable substitution

The next step of the DIVIDE query derivation is the input variable substitution. In this step, DIVIDE substitutes the input variables of each query from the query extraction output into the associated RSP-QL query pattern. To achieve this, a collection of N_3 rules have been defined that allow to substitute the input variables into the query body string in a deterministic way. These rules ensure that the correct substitution is performed, depending on whether the substituted value is a IRI or a literal, and depending on the correct data type in case of a literal. To perform the actual substitution, the semantic reasoner used in DIVIDE performs a forward reasoning step. The input of this reasoning

step consists of the collection of substitution rules, the output of the query extraction step and the query pattern of the considered DIVIDE query. For each query in the query extraction output, the output of this step consists of a series of triples that define the partially substituted RSP-QL query body.

The example result of the input variable substitution step for the running example is presented in Listing 13. This substitution is performed based on the generic RSP-QL query body that is referred to in the output of the query extraction in Listing 12. This query body is shown in Listing 7. In the input variable substitution output, lines 1–13 redefine the prefixes, which will be required in the final query derivation step, the RSP engine query update, to construct the full RSP-QL query. Line 16 shows the current state of the instantiated RSP-QL query body: input variables have already been substituted, but the window parameters still need to be substituted. The static and dynamic window parameters that will be used for substitution, are propagated from the output of the query extraction step (lines 19–20). They will be substituted in the following step of the DIVIDE query derivation.

Listing 13: Output of the input variable substitution step of the DIVIDE query derivation, performed for the running example on the query extraction output presented in Listing 12. The substitution is done using the generic RSP-QL query body of the corresponding DIVIDE query presented in Listing 7.

```
1  sd-query:prefixes-activity-showering a owl:Ontology.
2  sd-query:prefixes-activity-showering sh:declare _:bn_1.
3  _:bn_1 sh:prefix "xsd".
4  _:bn_1 sh:namespace "http://www.w3.org/2001/XMLSchema#"^^xsd:anyURI.
5  sd-query:prefixes-activity-showering sh:declare _:bn_2.
6  _:bn_2 sh:prefix "saref-core".
7  _:bn_2 sh:namespace "https://saref.etsi.org/core/"^^xsd:anyURI.
8  sd-query:prefixes-activity-showering sh:declare _:bn_3.
9  _:bn_3 sh:prefix "ActivityRecognition".
10 _:bn_3 sh:namespace "https://dahcc.idlab.ugent.be/Ontology/ActivityRecognition/"^^xsd:anyURI.
11 sd-query:prefixes-activity-showering sh:declare _:bn_4.
12 _:bn_4 sh:prefix "KBActivityRecognition".
13 _:bn_4 sh:namespace "https://dahcc.idlab.ugent.be/Ontology/ActivityRecognition/
       KBActivityRecognition/"^^xsd:anyURI.
14
15 _:sk_20 a sd:Query.
16 _:sk_20 sd:queryBody " CONSTRUCT { _:p a KBActivityRecognition:RoutineActivityPrediction ;
       ActivityRecognition:forActivity [ a <https://dahcc.idlab.ugent.be/Ontology/
       ActivityRecognition/Showering> ] ; ActivityRecognition:activityPredictionMadeFor <http://
       protego.ilabt.imec.be/idlab.homelab/patients/patient157> ; ActivityRecognition:predictedBy
       <https://dahcc.idlab.ugent.be/Ontology/ActivityRecognition/KBActivityRecognition/
       KBActivityRecognitionModel> ; saref-core:hasTimestamp ?now . } FROM NAMED WINDOW :win ON <
       http://protego.ilabt.imec.be/idlab.homelab> [RANGE ?{range} STEP ?{slide}] WHERE { BIND (
       NOW() as ?now) WINDOW :win { <https://dahcc.idlab.ugent.be/Homelab/SensorsAndActuators/70:
       ee:50:67:3e:78> saref-core:makesMeasurement [ saref-core:hasValue ?v ; saref-core:
       hasTimestamp ?t ; saref-core:relatesToProperty <https://dahcc.idlab.ugent.be/Homelab/
       SensorsAndActuators/org.dyamand.types.common.RelativeHumidity> ] . FILTER (xsd:float(?v) >
       xsd:float(\"57\"^^xsd:float)) } } ORDER BY DESC(?t) LIMIT 1 ".
17 _:sk_20 sh:prefixes sd-query:prefixes-activity-showering.
18 _:sk_20 sd:pattern sd-query:pattern.
19 _:sk_20 sd:staticWindowParameters (("?range" 30 time:seconds) ("?slide" 10 time:seconds)).
20 _:sk_20 sd:dynamicWindowParameters ().
```

6.5. Window parameter substitution

The output of the input variable substitution consists of a set of partially instantiated RSP-QL queries. In this step, the window parameters are also substituted to obtain the resulting RSP-QL query bodies. This is the final step that is performed by the semantic reasoner used in DIVIDE.

In general, DIVIDE offers the possibility to define the window parameters of the RSP queries resulting from the query derivation using semantic definitions. Currently, context-dependent window parameters can be defined by an end user via the definition of DIVIDE queries. Through the previous steps of the DIVIDE query derivation, the contextually relevant window parameters are defined in the input of the window parameter substitution step. However, by separating the window parameter substitution from the other query derivation steps, DIVIDE offers the flexibility to trigger the window parameter substitution for the contextually relevant RSP queries from other external sources. An example of this could be a device or network monitor. Currently, to enable the substitution of *context-dependent* window parameters, DIVIDE makes the distinction between static and dynamic window parameters.

A static window parameter is a window parameter of which the variable behaves as a regular input variable: it should be defined as a query variable in the antecedence of a DIVIDE query's sensor query rule, so that it can be substituted during this step of the query derivation similarly to how input variables are substituted. Static window parameters should be defined in the sensor query rule with a triple similar to the following one:

```
_:q sd:windowParameters (("?range" ?var time:seconds)) .
```

This requires the variable `?var` to occur in the sensor query rule's antecedence. When defining a DIVIDE query as an end user using an ordered set of existing SPARQL queries, this can be achieved by ensuring that the variable name of this window parameter appears in the WHERE clause of the stream query, in a named graph pattern of which the graph name does not refer to the stream IRI of a defined stream window. By definition, static window parameter variables will always receive a value in the query extraction output that *can* be used for substitution. Therefore, no default window parameter value for the given parameter's variable name (`?range` in the example above) should be defined in the associated stream window definition of the end-user DIVIDE query definition.

Dynamic window parameters are window parameters that are dynamically defined through context-enriching queries. They can be defined in separate context-enriching queries, but they can also be defined together with other outputs in other context-enriching queries. Dynamic window parameters are defined specifically for one DIVIDE query since they are defined in the context enrichment of this DIVIDE query's definition. Importantly, dynamic window parameters will *always* overwrite static window parameters. This means that during the window parameter substitution, dynamic window parameters will be substituted first. Next, static window parameters are substituted for those window parameter variables in the RSP-QL query body that have not yet been substituted.

The substitution order of static and dynamic window parameters implies a few important things. Multiple dynamic window parameters can be defined in different context-enriching queries of the same DIVIDE query, to handle different situations. It is however the responsibility of the end user that no more than one definition occurs for each window parameter variable in the enriched context. If multiple values are defined for the same window parameter variable, typically originating from multiple context-enriching queries, it is undefined which one will be used by DIVIDE during the dynamic window parameter substitution: this choice will be arbitrary. It is not required that there is a value defined for each window parameter variable in the enriched context either, since static window parameters are still substituted after the dynamic window parameters. However, this imposes a problem if no static window parameter value is defined for a certain window parameter variable. Therefore, a default value should be provided for those variables in the end-user definition of a DIVIDE query. For each such variable, DIVIDE will define a static window parameter itself in its internal representation of the DIVIDE query, of which the value is set to the given default. This way, this default value will then also be regarded as a static window parameter in case no dynamic window parameter value is defined in the enriched context for that window parameter variable.

In the running example, the definition of the generic DIVIDE query associated to the detection of an ongoing showering activity does not contain any context-enriching query that defines a dynamic window parameter. However, the running example's DIVIDE query that corresponds to the monitoring of the patient's location in the home, does include two context-enriching queries that define multiple dynamic window parameters. These queries are shown in Listing 14. The dynamic window parameters defined in the output of these queries are constructed based on the current context concerning any ongoing activity for this patient. It makes a distinction between two scenarios: when an activity *not in* the patient's routine is ongoing (first query), and when an activity *in* the patient's routine is ongoing (second query). Note that for the default case when no activity is currently ongoing, no dynamic window parameters are defined: in those cases, default values for the window parameter variables will be substituted as

static window parameters. Moreover, note that the two graph patterns in the WHERE clauses of the queries are semantically distinct: there will never be more than one query for which the graph pattern in the WHERE clause has a matching set of variables. This ensures that there is at most one value defined for the two window parameter variables in the enriched context.

The actual substitution of window parameters is very similar to the input variable substitution. For both the static and dynamic window parameter substitution, a forward reasoning step is performed with the semantic reasoner used in DIVIDE, reasoning on the output of the previous step and a collection of N_3 rules that ensure the correct substitution in a deterministic way. The actual substitution differs slightly from the input variable substitution in the sense that the unit of the variable is also specified. As explained when discussing the sensor query rule in Section 5.1.2, four units are supported: `xsd:duration`, `time:seconds`, `time:minutes` or `time:hours`. The unit `xsd:duration` requires a valid XML Schema duration string value such as `"PT5M"`. The other units require an integer value which will be converted to a similar duration string in the window definition. For example, a value of 5 with unit `time:minutes` will also be converted during the substitution to the string `"PT5M"`. Such duration strings are used in the window definitions of valid RSP-QL queries.

Listing 14: Examples of dynamic window parameter queries. These two queries are used in the general running example as context-enriching queries for the DIVIDE query that performs the monitoring of the patient's location in the home. They define the window parameters of this location query based on the current context concerning any ongoing activity for this patient that *is* or *is not* part of the patient's known routine.

```
# window-query1.sparql
CONSTRUCT {
    sd-query:pattern sd:windowParameters (
        [ sd-window:variable "range" ; sd-window:value 30 ; sd-window:type time:seconds ]
        [ sd-window:variable "slide" ; sd-window:value 30 ; sd-window:type time:seconds ]
    )
}
WHERE {
    ?patient rdf:type saref4ehaw:Patient ;
             MonitoredPerson:livesIn ?home .

    ?prediction1 rdf:type KBActivityRecognition:RoutineActivityPrediction ;
                 ActivityRecognition:activityPredictionMadeFor ?patient .

    FILTER NOT EXISTS {
        ?prediction2 rdf:type KBActivityRecognition:NonRoutineActivityPrediction ;
                     ActivityRecognition:activityPredictionMadeFor ?patient .
    }
}


# window-query2.sparql
CONSTRUCT {
    sd-query:pattern sd:windowParameters (
        [ sd-window:variable "range" ; sd-window:value 5 ; sd-window:type time:seconds ]
        [ sd-window:variable "slide" ; sd-window:value 5 ; sd-window:type time:seconds ]
    )
}
WHERE {
    ?patient rdf:type saref4ehaw:Patient ;
             MonitoredPerson:livesIn ?home .

    ?prediction1 rdf:type KBActivityRecognition:NonRoutineActivityPrediction ;
                 ActivityRecognition:activityPredictionMadeFor ?patient .

    FILTER NOT EXISTS {
        ?prediction2 rdf:type KBActivityRecognition:RoutineActivityPrediction  ;
```

```
                    ActivityRecognition:activityPredictionMadeFor ?patient .
    }
}
```

6.6. RSP engine query update

The output of the window parameter substitution step is a set of instantiated, valid RSP-QL queries that are contextually relevant for the given component. These queries are however still presented as a series of semantic triples. The tasks of this final step are the following: construction the actual RSP-QL query string, translating the query to the correct query language and updating the registered queries at the component's RSP engine.

Query construction The query construction step is quite trivial: from the output of the window parameter substitution step, an actual RSP-QL query should be constructed. This string is directly constructed from the set of prefixes and instantiated query body defined as semantic triples in the window parameter substitution output.

For the running example involving the generic DIVIDE query that detects an ongoing activity in the patient's home defined as an activity rule type similar to the showering rule's type, the RSP-QL query resulting from the query construction step is presented in Listing 15. This query is the result of performing the window parameter substitution and query construction on the output of the input variable substitution step presented in Listing 13.

Query translation In the definition of every DIVIDE component, the query language of the RSP engine is defined. If this language is different from RSP-QL, e.g., C-SPARQL, the RSP-QL query is translated in this step to a valid query in this other query language.

Query registration update The output of the previous step is a set of translated RSP queries for the given DIVIDE query. Since the DIVIDE query derivation is triggered because of a detected context change relevant to this component, the queries on the RSP engine of this component should be updated to reflect this new situation. To do so, DIVIDE keeps track of the queries that are currently registered on the RSP engine for the given DIVIDE query. Queries resulting from other DIVIDE queries or even external sources that are also registered on the RSP engine, are left untouched. For the queries resulting from the given DIVIDE query, DIVIDE performs a semantic comparison of the new set of instantiated translated RSP queries with this existing set of registered queries. Based on this comparison, any registered queries that are no longer in the new set of contextually relevant RSP queries are unregistered. New queries that are not running yet on the RSP engine, are registered.

For completeness, it is important to mention that during the full DIVIDE query derivation, the query processing on the RSP engine of the corresponding component should ideally be temporarily paused. The goal of this is to avoid that incorrect filtering is done, since DIVIDE already knows that the active queries registered on the engine might no longer be contextually relevant as soon as DIVIDE is informed of a context change for this component. It is however important that during the pause, incoming observations on the RSP engine's streams are not just ignored, but buffered temporarily. This way, the queries can be restarted as soon as the RSP query update step finishes, and the buffered stream data can be further fed to the RSP engine with their original timestamps.

Listing 15: Final RSP-QL query that is the result of performing the window parameter substitution and query construction steps of the DIVIDE query derivation, performed for the running example on the input variable substitution output presented in Listing 13.

```
PREFIX ActivityRecognition: <https://dahcc.idlab.ugent.be/Ontology/ActivityRecognition/>
PREFIX KBActivityRecognition: <https://dahcc.idlab.ugent.be/Ontology/ActivityRecognition/
    KBActivityRecognition/>
PREFIX saref-core: <https://saref.etsi.org/core/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

CONSTRUCT {
    _:p a KBActivityRecognition:RoutineActivityPrediction ;
        ActivityRecognition:forActivity [
```

```
            a <https://dahcc.idlab.ugent.be/Ontology/ActivityRecognition/Showering> ] ;
        ActivityRecognition:activityPredictionMadeFor
            <http://protego.ilabt.imec.be/idlab.homelab/patients/patient157> ;
        ActivityRecognition:predictedBy
            <https://dahcc.idlab.ugent.be/Ontology/ActivityRecognition/KBActivityRecognition/
                KBActivityRecognitionModel> ;
        saref-core:hasTimestamp ?now .
}
FROM NAMED WINDOW :win ON <http://protego.ilabt.imec.be/idlab.homelab> [RANGE PT30S STEP PT10S]
WHERE {
    BIND (NOW() as ?now)
    WINDOW :win {
        <https://dahcc.idlab.ugent.be/Homelab/SensorsAndActuators/70:ee:50:67:3e:78> saref-core
            :makesMeasurement [
            saref-core:hasValue ?v ;
            saref-core:hasTimestamp ?t ;
            saref-core:relatesToProperty <https://dahcc.idlab.ugent.be/Homelab/
                SensorsAndActuators/org.dyamand.types.common.RelativeHumidity>
        ] .
        FILTER (xsd:float(?v) > xsd:float("57"^^xsd:float))
    }
}
ORDER BY DESC(?t)
LIMIT 1
```

7. Implementation of the DIVIDE system

The previous sections have described DIVIDE from a methodological point of view, irregardless of implementation details. This section zooms in on our implementation of DIVIDE.

7.1. Semantic reasoner

Our implementation of DIVIDE uses the EYE reasoner [58], which fulfills the requirements of the semantic reasoner explained in Section 4. This N_3 reasoner runs in a Prolog virtual machine.

7.2. Deploying the DIVIDE system

DIVIDE is implemented in Java as a set of Java JAR modules. These modules include the DIVIDE engine, which is the core of DIVIDE, and an EYE module that implements the initialization and preprocessing steps with the EYE reasoner. Moreover, the DIVIDE server module is an executable JAR module that starts up DIVIDE upon execution. It should be configured through a collection of JSON files. It also exposes a REST API through which many details of its configuration and internal assets can be requested and updated.

7.3. Configuration of DIVIDE

The configuration of the DIVIDE system is provided through a main JSON file. In addition, the DIVIDE components can be defined in a separate file. Many configuration items can later be updated through the REST API exposed by the DIVIDE server. An example of the JSON configuration of the DIVIDE system is provided in Appendix A.

7.3.1. Configuration of the DIVIDE system

The configuration of the DIVIDE system includes details about the knowledge base, the ontology, the DIVIDE queries, the reasoner and the DIVIDE engine.

Knowledge base The type of the knowledge base (e.g., Apache Jena, RDFox) should be configured, *if* it is deployed by the DIVIDE server. This is the preferred option when deploying new systems If DIVIDE is deployed in an existing system, an existing Knowledge Base can also be used. In that case, the system will be responsible for monitoring context updates relevant to components registered to DIVIDE, and triggering the query derivation in the DIVIDE engine for those components whenever such a context update occurs.

Ontology To configure the ontology used by DIVIDE, the relevant ontology files should be specified.

Reasoner and engine The configuration of the DIVIDE reasoner and engine consists of a series of flags that allow to change the default DIVIDE behavior. For example, DIVIDE can be configured to handle TBox definitions in context graphs during the query derivation. Moreover, the parser can be configured to automatically create a variable mapping between the stream and final query based on equal variable names.

DIVIDE queries For every DIVIDE query, a separate JSON file should be linked in the configuration. This file can include the items of the internal representation of a DIVIDE query, or the end user definition of a DIVIDE query. In the latter case, the implementation of the DIVIDE query parser ensures that the parsed DIVIDE queries result in valid RSP-QL queries after the query derivation. This is achieved by validating the inputs, renaming the query variables to avoid any mismatches, ordering the input variables and static window parameters to obtain a deterministic substitution, and handling query variables in special constructs such as GROUP BY clauses.

As an example, Appendix A contains the JSON configuration of the DIVIDE query for the running use case example discussed in Section 5.1.

7.3.2. Configuration of the DIVIDE server

For the DIVIDE server, the host and port of the exposed REST API is defined. If DIVIDE deploys the Knowledge Base as well, the port of the Knowledge Base REST API available for context updates is also specified.

7.3.3. Configuration of a DIVIDE component

The components known by DIVIDE should be defined in an additional CSV file, which contains one entry per component. The properties of every component entry are separated by a semicolon.

7.4. DIVIDE REST API

When the DIVIDE server is running, a REST API is exposed. The API allows to perform the following operations:

- DIVIDE components: requesting component information, adding components, deleting existing components
- DIVIDE queries: requesting DIVIDE query information, deleting existing DIVIDE queries, and adding new DIVIDE queries (both with the end user definition and the internal representation)

7.5. Implementation of the ontology preprocessing

During the initialization of DIVIDE, the configured ontology is preprocessed with the EYE reasoner in three steps. First, an N_3 copy of the full ontology is created. Second, specialized ontology-specific rules are created from the original rules taken from the OWL 2 RL profile description [59]. Starting the EYE reasoning process from these rules will reduce the computational complexity of the reasoning [60]. Third, an image of the EYE reasoner, which has already loaded the ontology and specialized rules, is compiled within Prolog. This precompiled Prolog image is the result of the ontology preprocessing. By starting the semantic reasoning step of the query derivation process from this image, the triples and rules do not need to be loaded into the EYE reasoner each time it is called during the DIVIDE query derivation. This allows to make the semantic reasoning step significantly more efficient.

Although considered infrequent, ontology changes can be handled by the DIVIDE system. If DIVIDE is hosting the Knowledge Base, ontology changes can be made by using the Knowledge Base REST API. Any TBox change will result in DIVIDE reloading the ontology, redoing the ontology preprocessing, and triggering the query derivation for all DIVIDE queries and components. This is a computationally intensive operation.

7.6. Implementation of the DIVIDE query derivation

The DIVIDE query derivation is managed by the DIVIDE engine. To decouple the scheduler of query derivation tasks from their actual parallel execution, the DIVIDE engine manages a blocking task queue and a dedicated processing thread for every RSP engine (i.e., DIVIDE component) in the system.

Different tasks can be scheduled by the DIVIDE engine in the blocking task queue of a specific DIVIDE component. The main task type is a query derivation for one or all DIVIDE queries. In case of a context change, the query derivation is scheduled for all DIVIDE queries. However, in some situations, for example when a new DIVIDE query is added to the engine via the DIVIDE server's API, the query derivation is only scheduled for the new DIVIDE query. In case the query execution should be performed for multiple DIVIDE queries, the query derivation steps are executed in parallel for every DIVIDE query in newly spawned threads by the dedicated execution thread. Another task type is a DIVIDE query removal for a component resulting in the unregistering of the related RSP queries from the component's RSP engine. This task is scheduled for all DIVIDE queries of a component when a component is removed from the system via the server's API, or for all components and one specific DIVIDE query when this DIVIDE query is unregistered from the DIVIDE system.

The following paragraphs present some further implementation details of some DIVIDE query derivation steps.

Context enrichment Since this step involves the execution of actual SPARQL queries prior to the actual query derivation, this is the only semantic step of the query derivation process that is not necessarily performed by the EYE reasoner. This is the case if the queries contain SPARQL constructs that cannot be translated to a valid N_3 rule. In this case, the queries are executed in Java by using Apache Jena. In the other case, the queries are translated to N_3 rules which are then applied on the set of triples and, if reasoning is enabled, ontology rules.

RSP engine query update This final step of the query derivation is, obviously, not performed with the EYE reasoner. To update the query registrations at the RSP engines, the REST APIs of the RSP engine servers are used.

8. Evaluation set-ups

This section presents the evaluations of the DIVIDE system. Two types of evaluations are performed. First, the performance of the DIVIDE system is evaluated by measuring the duration of the different key actions taken by DIVIDE during its initialization and query derivation. Second, the real-time execution of RSP-QL queries generated by the DIVIDE query derivation is evaluated. This is done by comparing the real-time DIVIDE set-up with other well-known real-time approaches.

General information about the collected data, the ontology and context, and activity rules used for these evaluations is presented in Section 8.1. The detailed set-ups of both individual evaluations are further described in Section 8.2 and Section 8.3, respectively.

Supportive information relevant to the evaluation set-ups of this paper is publicly available at https://github.com/IBCNServices/DIVIDE/tree/master/swj2022.

8.1. Evaluation scenarios

All evaluations are performed on the eHealth use case described in Section 3.1. This section zooms in on the details of the evaluation scenarios of this use case.

8.1.1. Ontology

The ontology of the evaluation system is the activity recognition ontology as an extension of the existing DAHCC ontology [53], as presented in Section 3.3. This includes the `KBActivityRecognition` ontology and its imports. The imported ontologies include the `ActivityRecognition`, `MonitoredPerson`, `SensorsAndActuators` and `SensorsAndWearables` modules of the DAHCC ontology and its imported ontologies.

8.1.2. Realistic dataset for rule extraction and simulation

To properly perform the evaluations presented in this paper, a realistic data set is used that is the result of a large scale data collection process. This data collection took place in the imec-UGent HomeLab from June 2021 until October 2021. The HomeLab is an actual standalone house located on the UGent Campus Zwijnaarde, offering a unique residential test environment for IoT services, as it is equipped with all kinds of sensors and actuators. It contains different rooms that represent a typical home: an entry hallway, a ground floor toilet, a living room and kitchen, a staircase to the first floor, and a bathroom, master bedroom, hallway and toilet on the first floor. Prior to the data collection period, a literature study, observational studies and interviews with caregivers were performed to derive the activity types that are important to detect in a patient's home. Based on these activities, a list of properties was derived that could be of relevance to observe in order to detect these activities. These properties were then translated to the required sensors, which were all installed in the HomeLab. The data collected during the data collection period in the HomeLab is used for the evaluation in two ways: to extract realistic rules for activity recognition, and to create a realistic data set for simulation during the real-time evaluations.

Throughout the data collection, data was obtained from two sources relevant to this evaluation: a wearable device, and the in-home contextual sensors. For the former, the patient was equipped with an Empatica E4 wearable device [61]. It has a 3-axis accelerometer (32 Hz) as well as different sensors to measure a person's physiological data: blood volume pulse (64 Hz) and derived inter beat interval of heart rate, galvanic skin response (4 Hz) and skin temperature (4 Hz). For the latter, as explained, a wide range of sensors was installed in the HomeLab. These sensors measure localization, the number of people in a room, relative humidity, indoor temperature, motion, light intensity, sound, air quality, usage of water, electric power consumption of multiple devices, interaction with light switches and other buttons, the state of windows and doors and blinds, and others. During the data collection, participants labeled their activities, which were mapped by the researchers to the activities in the DAHCC ontology.

8.1.3. Context

The context for the evaluations, as considered by DIVIDE, consists of three main parts. The first part is the description of a patient living in a smart home, including the patient's wearables and a routine. For this part, the exact definitions in Listing 4 are used. The second part is a single triple representing the patient's location in the home, which is normally derived by a specific query. For the evaluation scenarios, the location of the patient in the home will always be the bathroom. The third part is the description of all sensors, actuators and wearables of the patient's smart home with the DAHCC ontology concepts. The smart home used in this evaluation scenario is the HomeLab. The instantiated example modules `_Homelab` and `_HomelabWearables` of the DAHCC ontology contain an actual representation of all sensors, actuators and wearables used within the HomeLab. The ABox definitions in these ontology modules represent the second part of the context used for the evaluations. Note that for these evaluations, the small set of TBox definitions present in both modules are also considered part of the ontology.

8.1.4. Activity rules

From the data collected during the large scale data collection in the HomeLab, data-driven rule mining algorithms were created that have extracted some realistic rules that can recognize some of the DAHCC activities from the data. For the evaluations of DIVIDE in this paper, rules for three bathroom activities are considered: toileting, showering and brushing teeth. Based on the analysis, the following rules were extracted:

- Toileting: the person present in the HomeLab is going to the toilet if a sensor that analyzes the energy consumption of the water pump has a value higher than 0.
- Showering: the person present in the HomeLab bathroom is showering if the relative humidity in the bathroom is at least 57%.
- Brushing teeth: the person present in the HomeLab bathroom is brushing his or her teeth if in the same time window (a) the sensor that analyzes the water running in the bathroom sink measures water running, and (b) the activity index value of the person's acceleration (measured by a wearable) is higher than 30. The activity index based on acceleration is defined as the mean variance of the acceleration over the three axes [62].

These three activity rules have been semantically described using the Activity Recognition ontology. The resulting descriptions are part of the `KBActivityRecognition` ontology module. These semantic descriptions of the activity rules are detailed in Appendix B.

The three given activity rules are the only rules present in the `KBActivityRecognition` ontology module during the evaluations. To represent each activity rule within DIVIDE, a DIVIDE query is created for each rule. Because of the completely similar definition, the generic DIVIDE query corresponding to the toileting and showering activity rules is the same. The DIVIDE queries are represented as a series of SPARQL queries.

8.2. Performance evaluation of DIVIDE

To derive the contextually relevant RSP queries, DIVIDE performs multiple steps, both during initialization and the query derivation. To evaluate the performance of DIVIDE, the duration of the main steps is measured for the given evaluation scenarios. In concrete, the duration of the following steps is measured:

1. the ontology preprocessing step with EYE of the activity recognition ontology;
2. the DIVIDE query parsing step with Java of the toileting DIVIDE query defined as SPARQL input;
3. the DIVIDE query derivation step for the toileting and brushing teeth DIVIDE queries separately, split up between the different EYE steps: semantic reasoning, query extraction, input variable substitution and window parameter substitution (note that the showering DIVIDE query is the same as the toileting DIVIDE query, and is therefore not evaluated twice).

The duration of these steps is measured from within the execution of the DIVIDE server Java JAR, which is configured with the scenario's ontology and DIVIDE queries to allow performing evaluation 1 and 2. To perform evaluation 3, the full context of the scenario is sent as new context to the DIVIDE Knowledge Base server.

Technical specifications The evaluation is performed on a device with a 2800 MHz quad-core Intel Core i5-7440HQ CPU and 16 GB DDR4-2400 RAM.

8.3. Real-time evaluation of derived DIVIDE queries

The task of DIVIDE is to manage the RSP queries on the registered RSP engines. These RSP queries are characterized by the fact that they do not require any more reasoning during their continuous evaluation, since semantic reasoning during the query derivation ensures that they are contextually relevant at every point in time. This section evaluates the real-time performance of evaluating these DIVIDE queries on the C-SPARQL RSP engine [15].

8.3.1. Evaluation of DIVIDE in comparison with real-time reasoning approaches

This evaluation compares the DIVIDE real-time approach, i.e., evaluating filtering RSP queries on an RSP engine, with other traditional approaches that do require real-time reasoning. The goal of the evaluation is to understand how DIVIDE compares to these traditional set-ups in terms of processing performance, as well as to understand the differences and the advantages and drawbacks of the different approaches.

To have a fair comparison, the real-time reasoning approaches should all reason within the same reasoning profile as DIVIDE, i.e., OWL 2 RL. Most of the approaches use RDFox [9], as this is known as one of the fastest OWL 2 RL (Datalog) reasoning engines that exist in the current state-of-the-art. Other approaches do OWL 2 RL reasoning via the Apache Jena rule reasoner.

Set-ups Different set-ups are considered for this evaluation. Most of the set-ups are streaming set-ups, meaning that they operate on windows taken from data streams. For every streaming set-up, Esper is the technology used to manage the windowing and to generate the window triggers [63].

1. DIVIDE approach using C-SPARQL without reasoning: regular C-SPARQL engine [15]. No ontology or context data is loaded into the engine, and no reasoning is performed during the continuous query evaluation.
2. Streaming RDFox: streaming version of RDFox. Consists of one engine that pipes Esper for windowing with RDFox for reasoning, via a processing queue. Initially, the ontology and context data are loaded into the data store of the RDFox engine, and a reasoning step is performed. Upon every window trigger generated by Esper, the window content is added as one event to a processing queue. When available, RDFox takes an event from the queue, incrementally adds it to the RDFox data store (i.e., it performs incremental reasoning with the event scheduled for addition), and executes the registered queries in order. If there are multiple queries

registered, query X incrementally adds its results to the data store, before query $X + 1$ is executed. Finally, RFDox performs incremental reasoning with the event and all previous query outputs scheduled for deletion (i.e., incremental deletion).

3. C-SPARQL piped with (non-streaming) RDFox: Initially, the RDFox data store contains the ontology and context data, and a reasoning step is performed. The queries registered on C-SPARQL listen to the observation stream, and run continuously on the stream window data and on the ontology and context triples. The axioms in the ontology are converted to a set of rules. Rule reasoning is performed during each query evaluation using these rules by C-SPARQL, which uses the Apache Jena rule reasoner with a hybrid forward and backward reasoning algorithm. C-SPARQL sends each query result to the event stream of a regular non-streaming RDFox engine, which adds it to a processing queue. Upon processing time, it incrementally adds the event to the data store, executes the registered queries in order, and incrementally deletes the event from the data store.
4. RDFox (non-streaming): RDFox engine wrapped into a server, that listens to the observation stream. Each incoming observation is added to a queue, which is processed by a separate thread. This thread takes an event from the queue, adds it to RDFox, performs incremental reasoning, and executes the registered queries in order. If there are multiple queries registered, query X incrementally adds its results to the data store, before query $X + 1$ is executed. Because this is a non-streaming version of RDFox, the event triples and triples constructed by the intermediate queries are not removed from the data store after processing.
5. Adapted Streaming RDFox: adapted streaming version of RDFox. This set-up only differs in one aspect from the original streaming RDFox set-up (2): before an event is added to RDFox, it checks the overlap between the event triples and existing triples in the data store. If overlapping triples are found, they are not added again to RDFox, and – most importantly – they are also not removed afterwards, so that no previously existing triples are removed from the data store after the event processing.
6. Semi-Streaming RDFox: mix between streaming RDFox set-up (2) and non-streaming RDFox set-up (4). This set-up only differs in one aspect from the original streaming RDFox set-up: the event triples and triples constructed by intermediate queries are not removed from the data store after processing. Hence, the only difference with the non-streaming RDFox set-up is that events are not added directly to the queue from the observation stream, but grouped together on Esper window triggers.
7. Streaming Jena: streaming version of the Apache Jena rule reasoner, similar to the streaming RDFox set-up (2). The only difference is the fact that during initialization, a set of rules is extracted from the ontology and loaded together with the ontology triples into the Apache Jena rule reasoner. Processing of events from the processing queue is done by this reasoner: it takes events, add them to the reasoner's data model, performs forward rule reasoning using the RETE algorithm, and executes the registered queries in order. Temporal query results are also added to the reasoner's data model, which are removed after processing of the event together with the event triples, followed by a final reasoning step. This set-up uses Apache Jena v3.7.

Each set-up is deployed with an associated WebSocket server to which an external component can connect to send data to the registered data streams. Each set-up involving RDFox uses RDFox v5.2.1, via the JRDFox Java jar, which is the Java bridge to the native RDFox engine. The RDFox data store used is the default `par-complex-nn` store, indicating a parallel data store using a complex indexing scheme with 32-bit integers.

Simulated data To create a simulation dataset to use in the evaluations, an anonymous representative portion is extracted from the dataset obtained with the large-scale data collection in the HomeLab. It contains real sensor observations of all HomeLab sensors and an Empatica E4 wearable worn by a real person living in the HomeLab for a day. Hence, the frequencies and values of the different observations are representative for a real smart home.

The data in the simulated specific scenarios is left unchanged except for two aspects: (i) obviously, the timestamps are shifted to real-time timestamps, and (ii) the values for the sensors relevant to the activity at hand are modified to ensure that the conditions for this activity are fulfilled all the time. In other words, the simulation for the brushing teeth scenario described below will lead to a detected brushing teeth activity throughout the full course of the scenario, and similarly for the other activities.

One hour of data from the anonymous data set used in this evaluation contains data of 231 different sensors, together producing 670,118 observations in this hour. These numbers can be further divided into data from the

HomeLab sensors versus data from the Empatica E4 wearable: 605,090 observations are produced by the 4 sensors of the Empatica E4 wearable, the remaining 65,028 observations are produced by 227 sensors in the HomeLab.

Specific scenarios Three specific scenarios, one for each activity rule in the general scenario, are constructed for this evaluation:

- Toileting scenario: Simulated HomeLab data for a period of 30 minutes is replayed at real rate, in batches of 1 second. For the streaming set-ups, the streaming queries are evaluated on a sliding window of 60 seconds with a sliding step of 10 seconds.
- Showering scenario: Simulated HomeLab data for a period of 20 minutes is replayed at real rate, in batches of 1 second. For the streaming set-ups, the streaming queries are evaluated on a sliding window of 60 seconds with a a sliding step of 10 seconds.
- Brushing teeth scenario: Simulated HomeLab data for a period of 30 minutes is replayed at real rate, in batches of 1 second. For the streaming set-ups, the streaming queries are evaluated on a sliding window of 30 seconds with a sliding step of 10 seconds.

The purpose of the evaluations is to measure and study the executions of the query evaluations and associated operations of the reasoner or engine, such as the semantic reasoning, both individually and progressively over time.

Replaying the data is performed by a data simulation component running on an external device in the same local network, to realistically represent the different sensor gateways. During simulation, this component connects as a client to the WebSocker server of the evaluated set-up, and sends the observations in each batch as a single message over the WebSocket connection to the appropriate data streams. This implies that one incoming event is registered by the set-ups every second. Hence, the streaming set-ups will add such an event to Esper for windowing every second, while the non-streaming set-up will trigger the in-order evaluation of the registered set-ups every second.

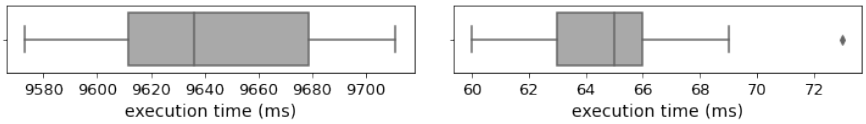
Evaluation queries To properly compare the different set-ups for each specific scenario, different versions of the SPARQL and C-SPARQL queries are created. In concrete, the following adaptations are made:

- For set-up 1, the C-SPARQL query as outputted by DIVIDE is registered.
- For set-ups 2, 4, 5, 6 and 7, the SPARQL definition of the DIVIDE query is modified to obtain two queries registered to the single reasoning service. The first reasoning query is the stream query of the SPARQL definition, from which the graph specifications are removed. The second query is the final query of this definition. Hence, the evaluated queries that are executed with RDFox or Apache Jena are regular SPARQL queries that involve semantic reasoning and are not rewritten by DIVIDE.
- For set-up 3, the SPARQL definition of the DIVIDE query is modified to obtain two queries. The first reasoning query is derived from the stream query of the SPARQL definition: the graph specifications are removed, and the query is translated to C-SPARQL syntax by adding the relevant FROM clauses that specify the query input: the static resources and the data stream window definition. This query is registered to the C-SPARQL engine. The second query is identical to set-up 2 and is registered to RDFox.

During an evaluation run, only the quer(y)(ies) related to the activity rule of the scenario are deployed on the engines. Queries related to other activity rules or aspects like location monitoring are not registered to the engines.

Measurements For each presented set-up, the *total execution time* metric is measured for each event. This metric is defined as the time starting from a *generated event* until the timestamp where an instance of the `RoutineActivityPrediction` is returned as output by the corresponding query. In a set-up with multiple queries that are executed in order, this is always the output of the final query in the chain. The definition of a *generated event* differs for each set-up: in the streaming set-ups, this is the time of an Esper window trigger; in the non-streaming set-up 4, this is the time of an incoming set of sensor observations.

Technical specifications All evaluations are run on a typical processing device in the IoT world: an Intel NUC, model D54250WYKH. It has a 1300 MHz dual-core Intel Core i5-4250U CPU (turbo frequency 2600 MHz) and 8 GB DDR3-1600 RAM.



(a) Ontology preprocessing (b) Query parsing for toileting query

Figure 4. Performance results of the initialization of the DIVIDE system: boxplot distributions of total execution times per step

8.3.2. Real-time evaluation of derived DIVIDE queries on a Raspberry Pi

DIVIDE is considered as a semantic component in a cascading reasoning set-up in an IoT network, which involves running RSP queries on local devices. These devices can be low-end devices with few resources in an IoT context. Hence, it is interesting to evaluate the real-time performance of continuously executing the RSP queries outputted by DIVIDE on a low-end device like a Raspberry Pi. This is the topic of the final evaluation.

For this evaluation, only the C-SPARQL baseline set-up (1) of the previous section is considered. The specific scenarios, evaluation queries, measurements and other properties of this evaluation are identical to those used for the evaluation in the previous section.

Technical specifications This evaluation is performed on a Raspberry Pi 3, Model B. This Raspberry Pi model has a Quad Core 1.2GHz Broadcom BCM2837 64bit CPU, 1GB RAM and MicroSD storage.

9. Evaluation Results

This section presents the results of the three evaluations described in Section 8. All results contain data of multiple evaluation runs, always excluding 3 warm-up and 2 cool-down runs.

9.1. Performance evaluation of DIVIDE

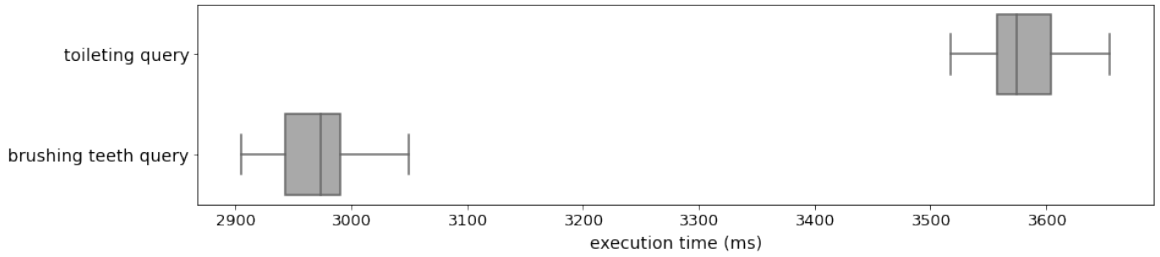
Figure 4 shows the distribution of the duration of two initialization steps of the DIVIDE system: the preprocessing of the Activity Recognition ontology, and the parsing of the toileting query specified as SPARQL input. The preprocessing of the ontology on average takes 9,640 ms, with a standard deviation (SD) of only 42 ms. The average duration of the query parsing is only 64.87 ms (SD 2.76 ms).

Figure 5 shows the performance results of the query derivation with DIVIDE, for the DIVIDE query corresponding to the toileting activity rule (also corresponds to the showering rule) and the DIVIDE query corresponding to the brushing teeth activity rule. Subfigure 5(a) shows the distribution of the duration of the query derivation for each individual query. The average durations of the query derivation are 3,578 ms (SD 38 ms) and 2,968 ms (SD 37 ms) for the toileting and brushing teeth DIVIDE queries, respectively.

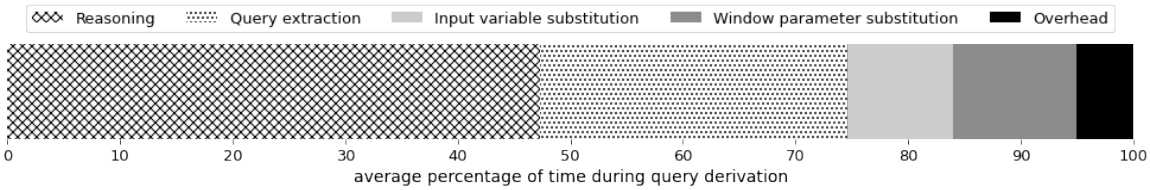
Subfigure 5(b) shows the percentage of time taken up by the different substeps, averaged over all runs for the three DIVIDE queries. These substeps include all steps performed with the EYE reasoner: the reasoning (47.27% on average), the query extraction (27.32% on average), the input variable substitution (9.44% on average) and the window parameter substitution (10.93% on average). The remaining time (5.04% on average) is overhead of the DIVIDE implementation, including internal threading and memory operations.

9.2. Evaluation of DIVIDE in comparison with real-time reasoning approaches

Figure 6 shows the results of the comparison of the real-time evaluation with DIVIDE on a C-SPARQL engine with different real-time reasoning approaches, for the toileting query. The results show the evolution over time of the total execution time from the event generation until the routine activity prediction is generated by the engine. The measurements included in the graphs are averaged over the evaluation runs. For three setups, there are no measurements shown for the full time course of the evaluation, which takes 30 minutes. These set-ups are the pipe of C-SPARQL with RDFox set-up (3), the adapted streaming RDFox set-up (5) and the streaming Jena set-up (7).



(a) Boxplot distribution of query derivation time for the toileting and brushing teeth DIVIDE query



(b) Relative times for query derivation substeps with EYE reasoner, averaged over all runs of the three DIVIDE queries

Figure 5. Performance results of the query derivation of the DIVIDE system

These missing measurements are caused by the systems running out of memory, causing them to stop evaluating the queries for the remainder of the scenario. The DIVIDE baseline set-up (1) has the lowest average total execution time from 960 seconds into the evaluation. Before this timestamp, the non-streaming RDFox set-up (4) is the quickest.

Figure 7 shows similar results for the real-time evaluation of the showering query. The same three set-ups run out of memory at a certain point, causing missing measurements for the remainder of the evaluation runs. Already after 550 seconds into the evaluation, the DIVIDE baseline set-up (1) has the lowest average total execution time.

Figure 8 shows similar results of the comparison of the real-time evaluation with DIVIDE with the real-time reasoning approaches, but for the toileting query. The properties of the graph are similar to those of the graph presenting the results for the brushing teeth query. In these results, only the non-streaming RDFox set-up (4) has no measurements for the full time course of the evaluation scenario due to the engine running out of memory.

In Appendix C, additional results of the evaluation runs over time are included. These results visualize the distribution of the total execution times for the different set-ups at different times during the evaluation runs.

9.3. Real-time evaluation of derived DIVIDE queries on a Raspberry Pi

Figure 9 shows the results of the evaluation of the DIVIDE set-up on the Raspberry Pi 3. These results visualize the distribution of the individual execution times of the RSP queries generated by DIVIDE with the C-SPARQL baseline set-up, for the toileting, showering and brushing teeth scenarios. For the toileting query, the average total execution time is 3,666 ms (SD 318 ms). This average number is 3,699 ms (SD 286 ms) and 3,001 ms (SD 174 ms) for the showering and brushing teeth scenarios, respectively.

10. Discussion

The inclusion of DIVIDE as a component in a semantic IoT platform allows to perform privacy-preserving monitoring of patients in homecare scenarios. This is possible because DIVIDE is designed to fit in a cascading architecture: it derives and manages contextually relevant RSP queries that require no additional reasoning while they are being executed, which makes them perfectly suitable to run on local low-end devices in the patient's home environment. By doing so, the processing of all privacy-sensitive data can be done locally, which means that it should not leave the home environment. By carefully designing the DIVIDE queries, one can precisely manage the data that is leaving the home environment, by specifying the data concepts that are being filtered by the local RSP engines and

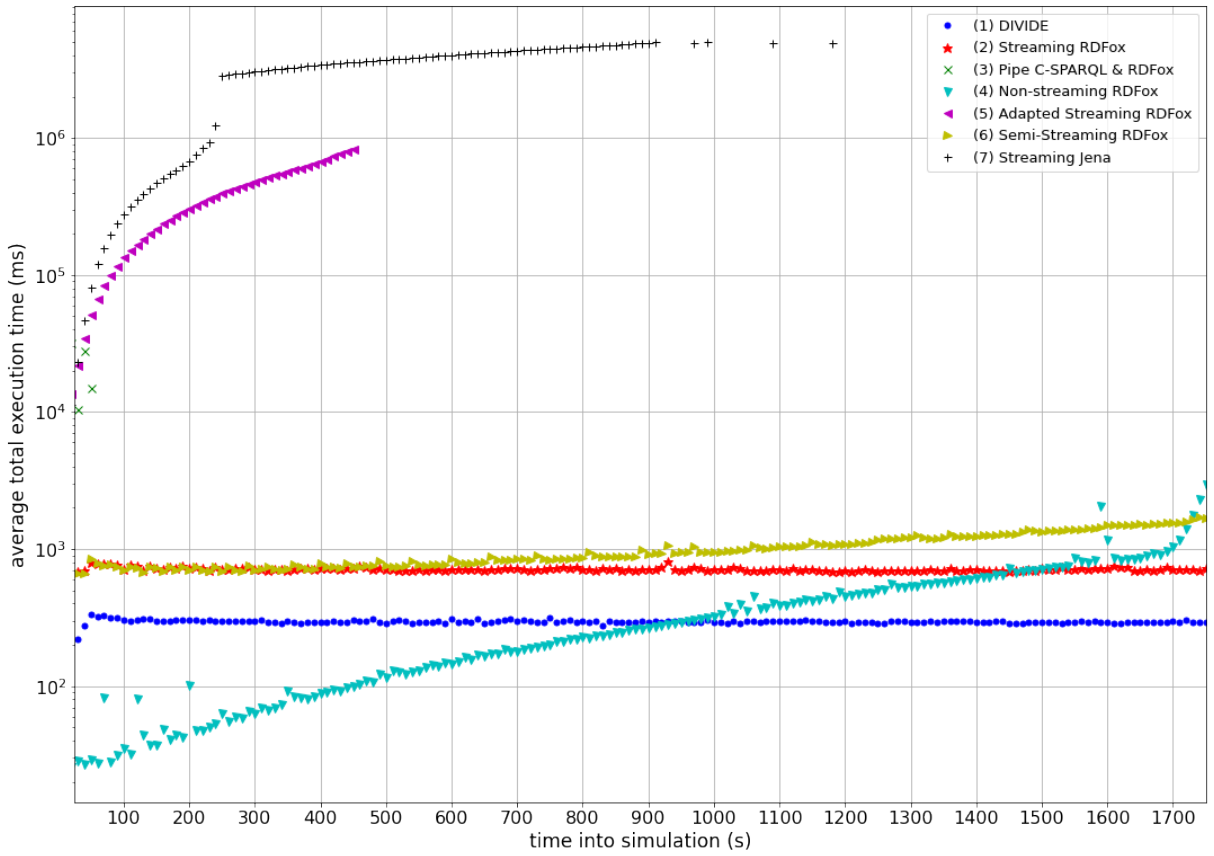


Figure 6. Results of the comparison of the DIVIDE real-time query evaluation approach with real-time reasoning approaches, for the toileting query. For each evaluation set-up, the results show the evolution over time of the total execution time from the generated event (either a windowed event in a streaming set-up or an incoming event in a non-streaming set-up) until the routine activity prediction as output of the final query. For all set-ups, measurements are shown for the processed event, either incoming or windowed, at every 10 seconds. All plotted execution times are averaged over the evaluation runs.

are therefore sent via the network to a central reasoner on a central server. In the described use case scenario, the patient's in-home location and detected activities comprise the only information that is leaving the home.

A DIVIDE query is generic by nature, which ensures that you should not define one DIVIDE query for every individual reasoning or filtering task that should be performed in the use case. In the activity recognition use case scenario discussed in this paper, one should only define a generic DIVIDE query per *type* of activity rule, instead of per activity rule individually. The generic nature of a DIVIDE query ensures that DIVIDE can derive the instantiated queries from it that are contextually relevant at any given point in time. This is achieved by listening to context updates in the knowledge base, and automatically triggering the query derivation upon a context change for all components that are affected by this context change. This is an improvement compared to systems where the management of the queries on the stream processing components of the IoT platform is still a manual, labor-intensive and thus highly impractical task. On the other hand, generic semantic queries can also be processed by reasoning engines, but while this is certainly feasible with current existing semantic reasoners for a single home environment, it might become more complex if this needs to be managed for a full network with for example many smart homes.

By deploying DIVIDE in a cascading architecture, more benefits are obtained than solely the privacy preservation, generic query definition and context-awareness. Since the high frequency and high volume data streams are processed locally, this data should not be transferred over the network. This significantly reduces network bandwidth usage and network delay impacting the system's performance. In addition, the data does not need to be processed by the central reasoner, which now only receives the outputs of the local RSP queries to do further processing. As such,

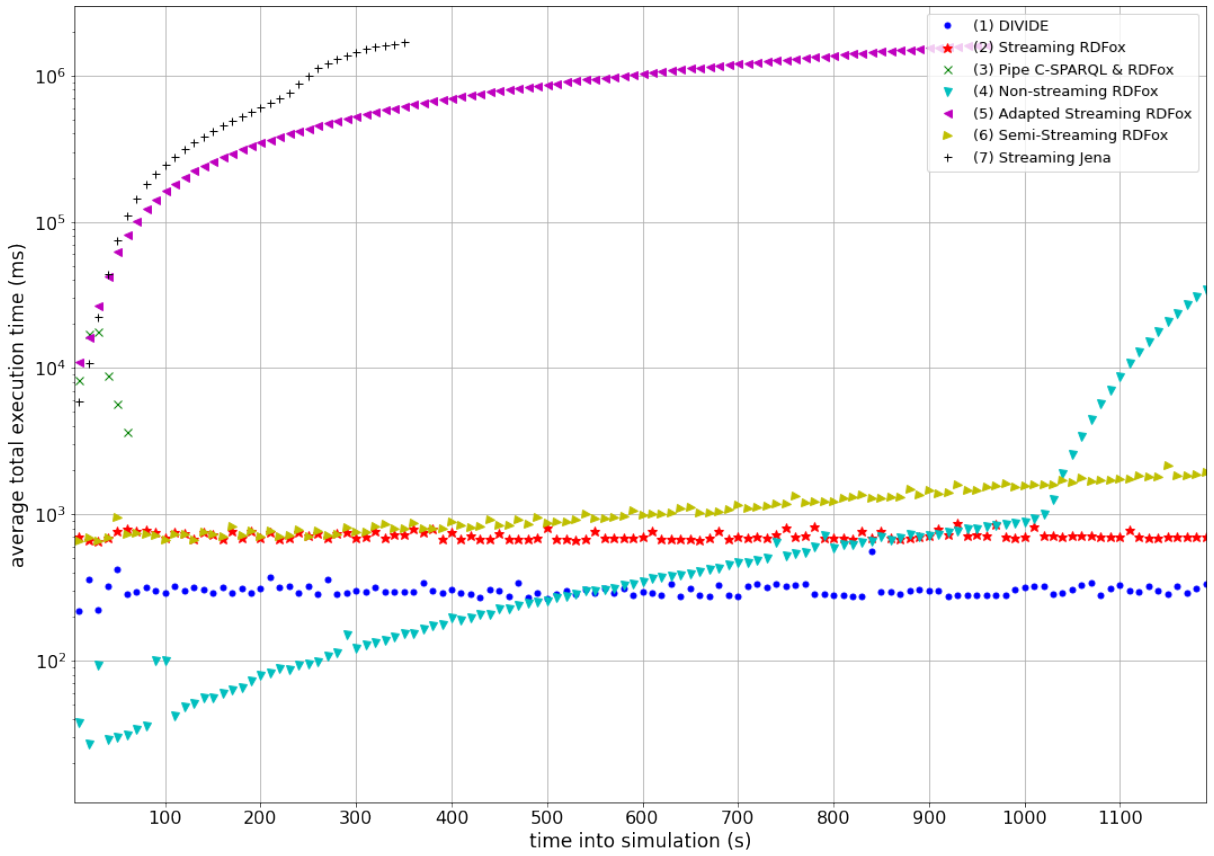


Figure 7. Results of the comparison of the DIVIDE real-time query evaluation approach with real-time reasoning approaches, for the showering query. For each evaluation set-up, the results show the evolution over time of the total execution time from the generated event (either a windowed event in a streaming set-up or an incoming event in a non-streaming set-up) until the routine activity prediction as output of the final query. For all set-ups, measurements are shown for the processed event, either incoming or windowed, at every 10 seconds. All plotted execution times are averaged over the evaluation runs.

the main resources of the server can be saved for the high-priority situations. In the presented use case scenario, an example is when an activity is detected that is not in the patient's routine: when this prediction is received by the central reasoner, it can investigate the cause of the issue and trigger further actions such as generating an alarm when needed. Meanwhile, the server resources can also be used by DIVIDE to derive the updated location monitoring query to ensure that the patient's location is followed up more closely.

When DIVIDE is used as a component in a semantic IoT platform to derive and manage the local RSP queries, it is of course important that the queries derived by DIVIDE have a good performance that is comparable to existing state-of-the-art stream reasoning systems. The results of this comparison with the C-SPARQL RSP engine running on an Intel NUC device demonstrate that the filtering RSP queries perform very well for the different activity detection queries that each correspond to a generic real-time reasoning set-up. The results show how the C-SPARQL queries are only outperformed by the classic non-streaming RDFox reasoning engine if you only look at the processing of single events. This can easily be explained by the fact that the events processed by this RDFox set-up contain fewer observations, and thus triples, than the events processed by C-SPARQL, which are larger batches of data grouped in data windows upon window triggers. Hence, due to the incremental reasoning in RDFox, this set-up initially performs best. However, looking at the evolution of the total execution times over time, the DIVIDE baseline set-up starts to perform better after a while. This is because the performance of the DIVIDE set-up stays constant over time, while the total execution time of the queries on the RDFox set-up increases over time because events are not removed from the data store, increasing the size of the data store on every execution. Therefore, we

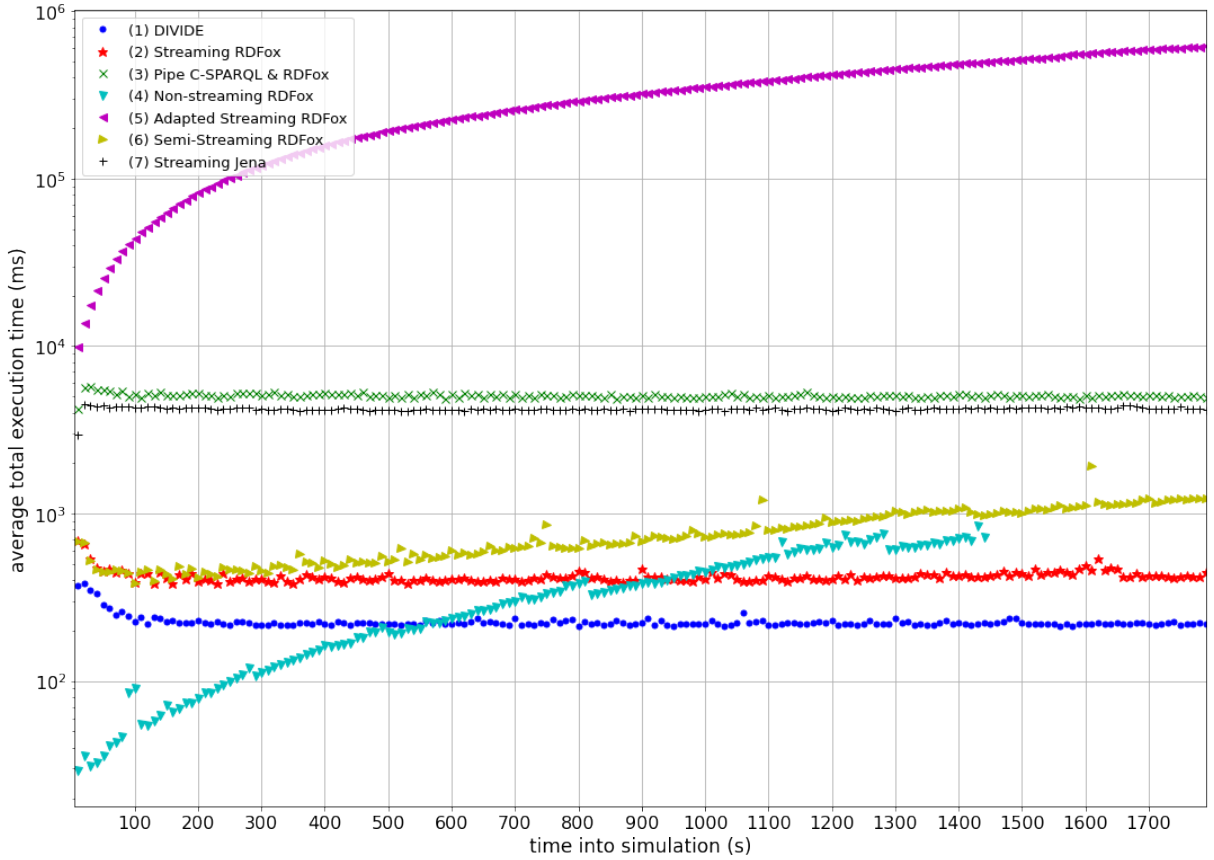


Figure 8. Results of the comparison of the DIVIDE real-time query evaluation approach with real-time reasoning approaches, for the brushing teeth query. For each evaluation set-up, the results show the evolution over time of the total execution time from the generated event (either a windowed event in a streaming set-up or an incoming event in a non-streaming set-up) until the routine activity prediction as output of the final query. For all set-ups, measurements are shown for the processed event, either incoming or windowed, at every 10 seconds. All plotted execution times are averaged over the evaluation runs.

have also included a comparison with a streaming version of RDFox. This set-up also performs constant over time, and is outperformed by a slight margin only by DIVIDE. This is mainly because RDFox still has to do some reasoning, which, even though this happens very efficiently with RDFox, is not required for the evaluation of the RSP queries with C-SPARQL in the baseline set-up. The streaming set-up of RDFox used in the evaluations makes a few assumptions that can still be optimized by looking at overlapping events and ensuring they are not removed after the processing of an event. However, in this adapted streaming RDFox setup, the processing of incoming events cannot keep up with the rate of the windowed events, causing the processing delay to build up, causing very long query execution times and memory issues in some cases. Moreover, looking at the results that involve reasoning with Apache Jena, it is clear that the set-ups using this semantic reasoner perform way worse than the DIVIDE and optimal RDFox set-ups. This is also true for the pipe of C-SPARQL with RDFox, in which C-SPARQL is performing rule-based reasoning with Apache Jena in the first query that causes the bad performance entirely on its own. This learns that using the built-in rule reasoning support of C-SPARQL is not efficient compared to alternative set-ups. As a conclusion, over time, DIVIDE performs comparable or even slightly better than the best RDFox set-ups, making it an ideal solution to integrate in a semantic platform to manage the local RSP queries, given the other main advantages. Ideally, this is combined in the cascading architecture with a central reasoner that does use a performant semantic reasoner such as RDFox.

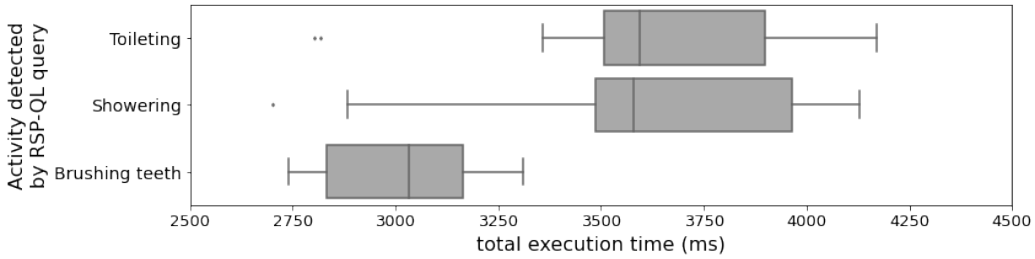


Figure 9. Results of evaluating the DIVIDE real-time query evaluation approach with the C-SPARQL baseline set-up (1), on a Raspberry Pi 3, Model B. The results show the total execution time distribution over the engine's runtime and multiple runs, for both the toileting and brushing teeth DIVIDE queries.

In IoT networks, devices with resources comparable to those of an Intel NUC are often unavailable locally. Therefore, it is important that the RSP queries can also be continuously executed on low-end devices with fewer resources. Otherwise, the data would still have to be sent to other devices with more resources running more centrally in the network that would then host the RSP engines, implying that all other advantages related to privacy, network usage and server resources do no longer apply. Therefore, the evaluation of the C-SPARQL baseline set-up was also performed on a Raspberry Pi. The results demonstrate that the queries can still be efficiently and consistently executed on such devices with way fewer resources than an Intel NUC. Specifically for this evaluation, the queries take approximately 10 times longer than on the Intel NUC, but take still well below the query execution frequency of 10 seconds. This is an additional advantage when deploying a system involving DIVIDE, as no large scale investment in expensive high-end hardware is required. In real set-ups, actually deploying a Raspberry Pi may however not be very practical or realistic. However, the resources of a Raspberry Pi are very comparable to other local devices such as wearable devices like the smartwatches in Samsung Galaxy Watch or Apple Watch series. Note that RDFox can also be used instead of C-SPARQL to run the queries derived by DIVIDE on a local low-end device, since RDFox can successfully run on an ARM based edge device like a Raspberry Pi or a smartphone as well [11]. This implies that the use of a RDFox set-up would also ensure that data can be processed locally instead of being sent to a server.

Up to now, we have only looked at the real-time evaluation of RSP queries derived by DIVIDE. They perform well in realistic homecare monitoring environments, but another important aspect is the performance of DIVIDE itself. The results of the DIVIDE performance evaluation show that the main portion of time during the initialization of DIVIDE is taken by the preprocessing of the ontology. Of course, the duration of the preprocessing depends on the number of triples and axioms defined in the ontology, which is use case specific. In any case, this is a task that should only happen once, given the assumption in DIVIDE that ontology updates do not happen. Nevertheless, DIVIDE does support ontology updates, but they require the ontology preprocessing to be redone. Besides the initialization, it is important to inspect the duration the query derivation process when a context change is observed. For this step, the performance results show that for the given evaluation use case scenario, the query derivation typically takes around 3 to 4 seconds. This is an order of magnitude higher than the time needed to perform real-time reasoning with RDFox during the query evaluation on an incoming event. However, the execution frequency of the query derivation is a few orders of magnitude smaller than the frequency of the event processing: events are processed on every window trigger or incoming observation, which is every 10 seconds or every second in the evaluation use case scenario. As you are not expecting a context change every 10 seconds, this shows that the performance results of the query derivation step are perfectly acceptable. In addition, the results show that the largest portion of the time is taken up by the different steps performed with the EYE reasoner. The biggest portion of the time, almost 50%, is spent on generating the proof with the EYE reasoner. The results show that only less than 5% of the query derivation step is overhead induced by the DIVIDE implementation.

To be able to use DIVIDE in a real IoT platform set-up, it is important that DIVIDE is practically usable. Therefore, we have implemented DIVIDE in a way that tries to maximize its practical usability. First, DIVIDE is available as an executable Java JAR component that can easily be run in a server environment, allowing for easy integration into an existing IoT platform. The main configuration of the server, engine, DIVIDE queries and components can

be easily created and modified with straightforward JSON and CSV files. Importantly, DIVIDE also does not hinder RSP engines to have active queries managed manually or by other system components, ensuring that the inclusion of DIVIDE into a semantic platform is not an all-or-nothing choice. In addition, the REST API exposed by the DIVIDE server implies that the configuration of DIVIDE is not fixed: components and DIVIDE queries can be easily added or removed, increasing the flexibility of the system. The internal implementation ensures that such changes are correctly handled and reflected on the RSP engines as well. Moreover, the implementation of the query parser allows the flexible and straightforward end-user definition of a DIVIDE query. This allows existing sets of queries to be used with DIVIDE to perform semantically equivalent tasks after only a small configuration effort, ensuring that no inner details of DIVIDE need to be known by end users who want to integrate it into their system. As a result, we believe that DIVIDE is perfectly suited in an IoT set-up where it is deployed in a cascading architecture.

11. Conclusion

This paper has presented the DIVIDE system. DIVIDE is designed as a semantic component that can automatically and adaptively derive and manage the queries of the stream processing components in a semantic IoT platform, in a context-aware manner. Through a specific homecare monitoring use case, this paper has shown how DIVIDE can divide the active queries across a cascading IoT set-up, and conquer the issues of existing systems by fulfilling important requirements related to data privacy preservation, performance, and usability.

Reaching back to the research objectives outlined in Section 1, we have achieved these in this paper with DIVIDE in the following ways:

1. By adopting a cascading reasoning architecture, DIVIDE manages the queries for the stream processing components that are running on local IoT devices. This ensures maximal privacy preservation: the stream processing queries process the privacy-sensitive data locally, only their outputs leave the home environment.
2. DIVIDE automatically triggers the derivation of the semantic queries of a stream processing component when changes are observed to context information that is relevant to that specific component. This way, DIVIDE automatically ensures that the active queries on each component are contextually relevant at all times. This process is context-aware and adaptive by design, minimizing the manual configuration effort for the end user to the initial query definition only. Once the system is deployed, no configuration changes are required anymore.
3. By performing semantic reasoning on the current context during the query derivation, DIVIDE ensures that the resulting stream processing queries can perform all relevant monitoring tasks without doing real-time reasoning. The evaluations on the use case scenario demonstrate how this ensures that DIVIDE performs comparable or even slightly better than state-of-the-art stream reasoning set-ups involving RDFox in terms of query execution times. This implies that the queries can also be executed in real-time on low-end devices with few resources, as demonstrated by the evaluations. The cascading architecture in which DIVIDE is adopted ensures minimal network congestion and optimal usage of the central resources of the network.
4. Through the definition of a DIVIDE query, an end user can make the window parameters of the stream processing queries context-dependent with DIVIDE.
5. Generic queries in DIVIDE can be easily defined by only slightly adapting existing SPARQL or RSP-QL queries, ensuring DIVIDE is practically usable.

There are multiple interesting future pathways related to DIVIDE that are worth investigating. First, the cascading architectural set-up in which DIVIDE is ideally deployed can be further exploited. By including the monitoring of device, network and/or stream characteristics into DIVIDE, the distribution of semantic stream processing queries across the IoT network could be dynamically adapted to optimize both local and global system performance. Such a monitor could also exploit the dynamic window parameter substitution functionality of DIVIDE to adapt these parameters to the monitored conditions. Second, the current implementation of DIVIDE only supports use cases that reason in the OWL 2 RL profile. However, the EYE reasoner used supports extending the rule set to obtain higher expressivity. Doing so would introduce support for higher expressivity use cases in DIVIDE.

Acknowledgements

This research is part of the imec.ICON project PROTEGO (HBC.2019.2812), co-funded by imec, VLAIO, Televic, Amaron, Z-Plus and ML2Grow. Bram Steenwinckel (1SA0219N) is funded by a strategic base research grant of Fund for Scientific Research Flanders (FWO), Belgium. Pieter Bonte (1266521N) is funded by a postdoctoral fellowship of FWO.

Appendix A. Configuration of the DIVIDE implementation

This appendix gives some examples of how our implementation of DIVIDE, which is presented in Section 7, should be concretely configured.

- Listing 16 shows an example of the JSON configuration of the DIVIDE system.
- Listing 17 contains the JSON configuration of the DIVIDE query for the running use case example discussed in Section 5.1. In other words, parsing the configured DIVIDE query with the DIVIDE query parser leads to the DIVIDE query goal in Listing 6 and the sensor query rule in Listing 7. The query files referred to in Listing 17 are shown in Listing 8 (stream query) and Listing 9 (final query).

Listing 16: Example JSON configuration of the DIVIDE system

```
{
  "divide": {
    "kb": {
      "type": "Jena",
      "baseIri": "http://protego.ilabt.imec.be/idlab.homelab/"
    },
    "ontology": {
      "dir": "definitions/ontology/",
      "files": [
        "KBActivityRecognition.ttl",
        "ActivityRecognition.ttl",
        "MonitoredPerson.ttl",
        "Sensors.ttl",
        "SensorsAndActuators.ttl",
        "SensorsAndWearables.ttl",
        "_Homelab_tbox.ttl",
        "_HomelabWearable_tbox.ttl",
        "imports/eep.ttl",
        "imports/affectedBy.ttl",
        "imports/cpannotationschema.ttl",
        "imports/saref.ttl",
        "imports/saref4bldg.ttl",
        "imports/saref4ehaw.ttl",
        "imports/saref4wear.ttl"
      ]
    },
    "queries": {
      "sparql": [
        "divide-queries/activity-showering.json"
      ]
    },
    "reasoner": {
      "handleTboxDefinitionsInContext": false
    },
    "engine": {
      "parser": {
```

```
        "processUnmappedVariableMatches": false,
        "validateUnboundVariablesInRspQlQueryBody": true
      },
      "stopRspEngineStreamsOnContextChanges": true
    }
  },
  "server": {
    "host": "localhost",
    "port":{
      "divide": 8342,
      "kb": 8343
    }
  }
}
```

Listing 17: Configuration of the DIVIDE query detecting an ongoing activity in a patient's routine, where the activity can be detected by a single independent sensor in the room that crosse a defined value threshold. The parsing of the given configuration by the DIVIDE query parser leads to the DIVIDE query with the goal in Listing 6 and the sensor query rule in Listing 7. The content of the file named `stream-query.sparql` in the configuration is presented in Listing 8, the content of the file named `final-query.sparql` is presented in Listing 9.

```
{
  "streamWindows": [
    {
      "streamIri": "http://protego.ilabt.imec.be/idlab.homelab",
      "windowDefinition": "RANGE PT?{range}S STEP PT?{slide}S",
      "defaultWindowParameterValues": {
        "?range": "30",
        "?slide": "10"
      }
    }
  ],
  "streamQuery": "stream-query.sparql",
  "finalQuery": "final-query.sparql",
  "solutionModifier": "ORDER BY DESC(?t) LIMIT 1",
  "streamToFinalQueryVariableMapping": {
    "?activityType": "?activityType",
    "?patient": "?patient",
    "?model": "?model",
    "?now": "?t"
  },
  "contextEnrichment": {
    "queries": [],
    "doReasoning": true,
    "executeOnOntologyTriples": true
  }
}
```

Appendix B. Semantic activity rules of the DIVIDE evaluation scenarios

This appendix contains the semantic description of the activity rules used in the evaluation of the DIVIDE system, as presented in Section 8.1.4. These rules include a rule for the toileting and brushing teeth activity. They are semantically defined using the Activity Recognition ontology presented in Section 3.3, in the `KBActivityRecognition` ontology module. To improve readability, the `KBActivityRecognition:` prefix is replaced by the `:` prefix in all semantic listings of this appendix.

– Toileting: the person present in the HomeLab is going to the toilet if a sensor that analyzes the energy consumption of the water pump has a value higher than 0. This translates into the following activity rule definition in the `KBActivityRecognition` module:

```
:toileting_rule rdf:type :ActivityRule ;
    ActivityRecognition:forActivity :_Toileting ;
    :hasCondition :toileting_condition01 .

:toileting_condition01 rdf:type :RegularThreshold ;
    :forProperty :_EnergyConsumption ;
    Sensors:analyseStateOf :_Pump ;
    :isMinimumThreshold "true"^^xsd:boolean ;
    saref-core:hasValue "1.0E-5"^^xsd:float .
```

– Showering: the person present in the HomeLab bathroom is showering if the relative humidity in the bathroom is at least 57%. This translates into the following activity rule definition in the `KBActivityRecognition` module:

```
:showering_rule rdf:type :ActivityRule ;
    ActivityRecognition:forActivity :_Showering ;
    :hasCondition :showering_condition01 .

:showering_condition01 rdf:type :RegularThreshold ;
    :forProperty :_RelativeHumidity ;
    Sensors:analyseStateOf :_BathRoom ;
    :isMinimumThreshold "true"^^xsd:boolean ;
    saref-core:hasValue "57.0"^^xsd:float .
```

– Brushing teeth: the person present in the HomeLab bathroom is brushing his or her teeth if in the same time window (a) the sensor that analyzes the water running in the bathroom sink measures water running, and (b) the activity index value of the person's acceleration (measured by a wearable) is higher than 30. The activity index based on acceleration is defined as the mean variance of the acceleration over the three axes. This translates into the following activity rule definition in the `KBActivityRecognition` module:

```
:brushing_teeth_rule rdf:type :ActivityRule ;
    ActivityRecognition:forActivity :_BrushingTeeth ;
    :hasCondition :brushing_teeth_condition01 .

:brushing_teeth_condition01 rdf:type :AndCondition ;
    :firstCondition :brushing_teeth_condition02 ;
    :secondCondition :brushing_teeth_condition03 .

:brushing_teeth_condition02 rdf:type :RegularThreshold ;
    :forProperty :_WaterRunning ;
    Sensors:analyseStateOf :_Room ;
    :isMinimumThreshold "true"^^xsd:boolean ;
    saref-core:hasValue "1.0E-5"^^xsd:float .

:brushing_teeth_condition03 rdf:type :MeanVarianceThreshold ;
    :forProperty :_WearableAcceleration ;
    Sensors:analyseStateOf :_Patient ;
    :isMinimumThreshold "true"^^xsd:boolean ;
    saref-core:hasValue "30.0"^^xsd:float .
```

Appendix C. Additional results of the evaluation of DIVIDE in comparison with real-time reasoning approaches

This appendix contains additional results of comparing the real-time evaluation of RSP queries derived by DIVIDE on a C-SPARQL engine, with the other evaluation set-ups that do involve real-time reasoning. These results are complementary to the results shown in Section 9.2, for the evaluation set-up as discussed in Section 8.3.1.

Figure 10 includes two boxplots that show the distribution of the total query execution times for the evaluation of the toileting DIVIDE query, for each set-up over the multiple evaluation runs. The distribution is shown for two timestamps corresponding to the mean values that are visualized in the timeline of Figure 6. Hence, the distributions correspond to the total execution times measured during the same corresponding evaluation runs. Subfigure 10(a) shows the distribution for the total execution times for the event, either streaming or incoming, generated 60 seconds after starting the data simulation. Subfigure 10(b) visualizes this distribution for the event generated 1300 seconds after the start of the data simulation. The results show how the non-streaming RDFox set-up has the smallest total execution times in the beginning of the simulation after only 60 seconds, while DIVIDE has smaller total execution times after 1300 seconds. Note that the boxplot distributions after 1300 seconds do not include results for the pipe of C-SPARQL with RDFox set-up (3), the adapted streaming RDFox set-up (5) and the streaming Jena set-up (7) due to those systems running out of memory before reaching this timestamp in the evaluation.

Figure 11 shows results completely similar to the results in Figure 10, but for the brushing teeth query. The distributions that are visualized correspond to the mean values that are visualized in the timeline of Figure 8. Additional results for the showering query are omitted due to their high similarity with the results of the other queries.

References

[1] K. Jaiswal and V. Anand, A survey on IoT-based healthcare system: Potential applications, issues, and challenges, in: *Advances in Biomedical Engineering and Technology*, Springer, pp. 459–471.

[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) (2015), 1844–1860.

[3] C.C. Aggarwal, N. Ashish and A. Sheth, The internet of things: A survey from the data-centric perspective, in: *Managing and mining sensor data*, Springer, 2013, pp. 383–428.

[4] 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) (2012), 1–21.

[5] D. Dell'Aglio, E. Della Valle, F. van Harmelen and A. Bernstein, Stream reasoning: A survey and outlook, *Data Science* **1**(1–2) (2017), 59–83.

[6] K. Abouelmehdi, A. Beni-Hssane, H. Khaloufi and M. Saadi, Big data security and privacy in healthcare: A Review **113**, 73–80.

[7] P. Bonte, F. Ongenae and F. De Turck, Subset reasoning for event-based systems, *IEEE Access* **7** (2019), 107533–107549.

[8] B. Motik, B.C. Grau, I. Horrocks, Z. Wu, A. Fokoue, C. Lutz et al., OWL 2 web ontology language profiles, *W3C recommendation* **27**(61) (2009), 61.

[9] Y. Nenov, R. Piro, B. Motik, I. Horrocks, Z. Wu and J. Banerjee, RDFox: A highly-scalable RDF store, in: *ISWC 2015*, Springer, 2015, pp. 3–20.

[10] J. Urbani, C. Jacobs and M. Krötzsch, Column-oriented datalog materialization for large knowledge graphs, in: *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 30, 2016.

[11] J. Lee, T. Hwang, J. Park, Y. Lee, B. Motik and I. Horrocks, A context-aware recommendation system for mobile devices, CEUR Workshop Proceedings, 2020.

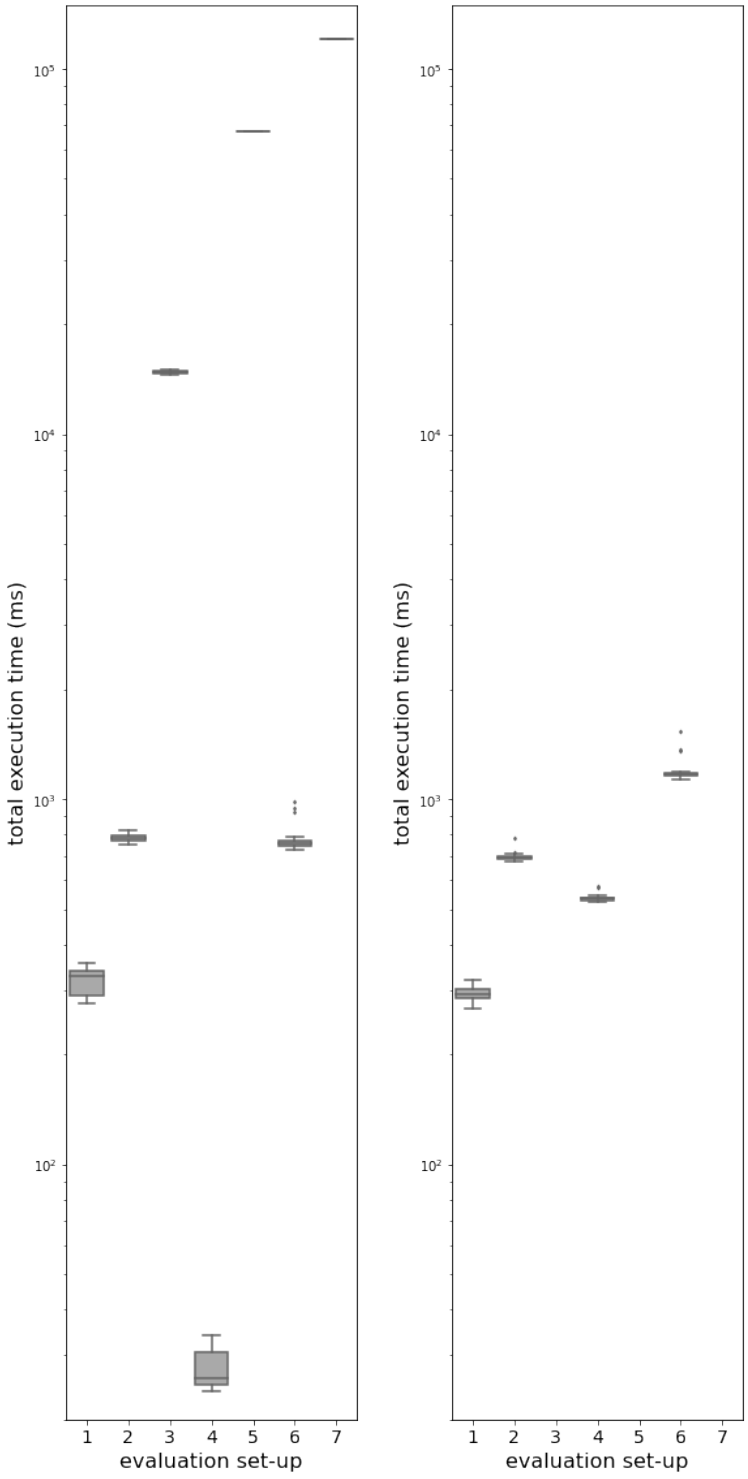
[12] T. Berners-Lee, D. Connolly, L. Kagal, Y. Scharf and J. Hendler, N3Logic: A logical framework for the World Wide Web, *Theory and Practice of Logic Programming* **8**(3) (2008), 249–269.

[13] R. Cyganiak, D. Wood, M. Lanthaler, G. Klyne, J.J. Carroll and B. McBride, RDF 1.1 concepts and abstract syntax, *W3C recommendation* **25**(02) (2014), 1–22.

[14] X. Su, E. Gilman, P. Wetz, J. Riekki, Y. Zuo and T. Leppänen, Stream reasoning for the Internet of Things: Challenges and gap analysis, in: *WIMS 2016*, ACM, 2016.

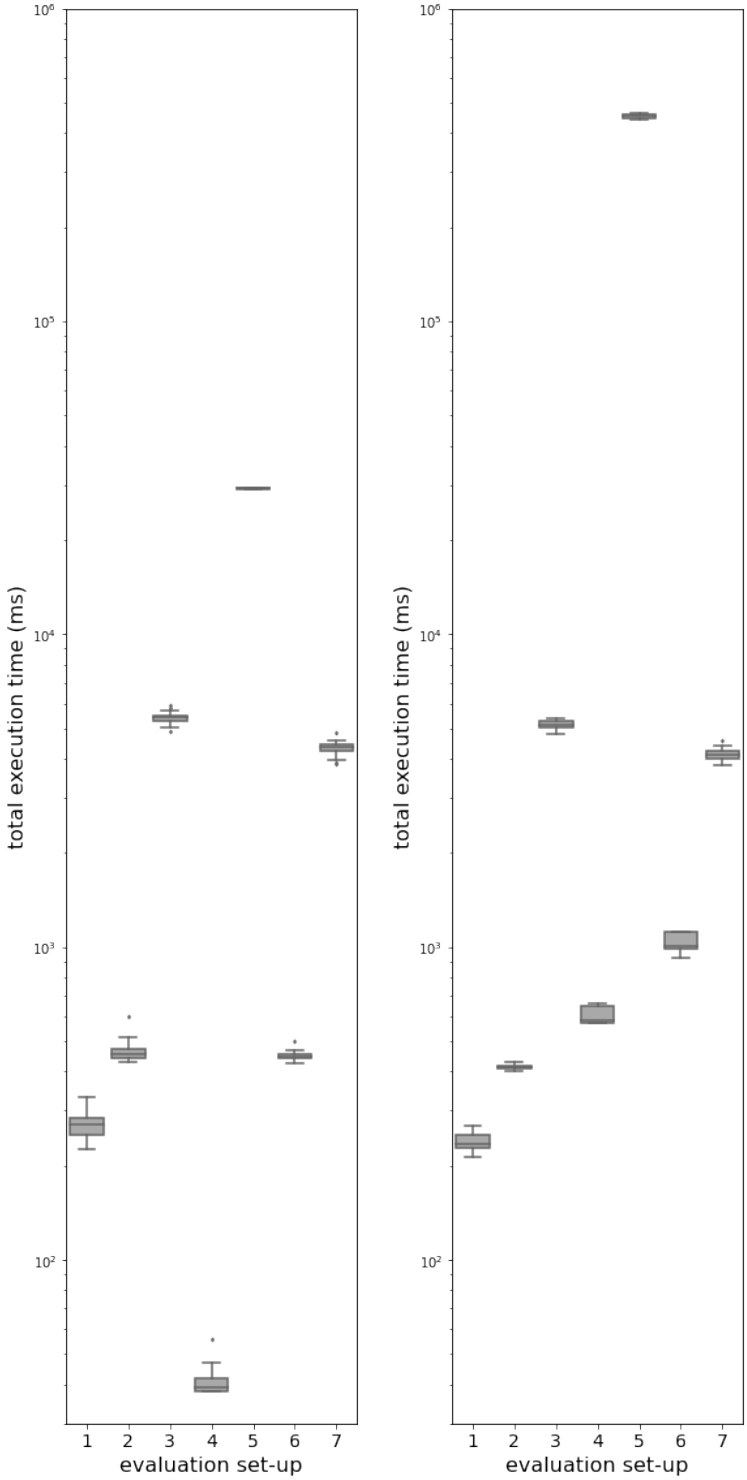
[15] D.F. Barbieri, D. Braga, S. Ceri, E. Della Valle and M. Grossniklaus, C-SPARQL: a continuous query language for RDF data streams, *International Journal of Semantic Computing* **4**(1) (2010), 3–25.

[16] D. Le-Phuoc, M. Dao-Tran, J.X. Parreira and M. Hauswirth, A native and adaptive approach for unified processing of linked streams and linked data, in: *ISWC 2011*, Springer, 2011, pp. 370–388.



(a) At 60 seconds (b) At 1300 seconds

Figure 10. Results of the comparison of the DIVIDE real-time query evaluation approach with real-time reasoning approaches, for the toileting query. For each evaluation set-up, the results show a boxplot distribution of the total execution time from the generation event (either a windowed event in a streaming set-up or an incoming event in a non-streaming set-up) until the routine activity prediction as output of the (final) query. The distribution is shown for two timestamps corresponding to the mean values for this timestamp plotted in Figure 6.



(a) At 60 seconds (b) At 1300 seconds

Figure 11. Results of the comparison of the DIVIDE real-time query evaluation approach with real-time reasoning approaches, for the brushing teeth query. For each evaluation set-up, the results show a boxplot distribution of the total execution time from the generation event (either a windowed event in a streaming set-up or an incoming event in a non-streaming set-up) until the routine activity prediction as output of the (final) query. The distribution is shown for two timestamps corresponding to the mean values for this timestamp plotted in Figure 8.

[17] R. Tommasini and E. Della Valle, Yasper 1.0: Towards an RSP-QL Engine, in: *International Semantic Web Conference (Posters, Demos & Industry Tracks)*, 2017.
[18] R. Tommasini, P. Bonte, F. Ongenae and E. Della Valle, RSP4J: An API for RDF Stream Processing, in: *European Semantic Web Conference*, Springer, 2021, pp. 565–581.
[19] D. Dell'Aglio, E. Della Valle, J.-P. Calbimonte and O. Corcho, RSP-QL semantics: A unifying query model to explain heterogeneity of RDF stream processing systems, *International Journal on Semantic Web and Information Systems (IJSWIS)* **10**(4) (2014), 17–44.
[20] D.F. Barbieri, D. Braga, S. Ceri, E.D. Valle and M. Grossniklaus, Incremental reasoning on streams and rich background knowledge, in: *Extended Semantic Web Conference*, Springer, 2010, pp. 1–15.
[21] 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*, 2012, pp. 58–68.
[22] B. Motik, Y. Nenov, R.E.F. Piro and I. Horrocks, Incremental update of datalog materialisation: the backward/forward algorithm, in: *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.
[23] J. Urbani, A. Margara, C. Jacobs, F.v. Harmelen and H. Bal, Dynamite: Parallel materialization of dynamic rdf data, in: *International Semantic Web Conference*, Springer, 2013, pp. 657–672.
[24] F. Lécué, Diagnosing changes in an ontology stream: A dl reasoning approach, in: *Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.
[25] E. Thomas, J.Z. Pan and Y. Ren, TrOWL: Tractable OWL 2 reasoning infrastructure, in: *Extended Semantic Web Conference*, Springer, 2010, pp. 431–435.
[26] 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*, Springer, 2013, pp. 247–252.
[27] H. Beck, M. Dao-Tran and T. Eiter, LARS: A logic-based framework for analytic reasoning over streams, *Artificial Intelligence* **261** (2018), 16–70.
[28] H.R. Bazoobandi, H. Beck and J. Urbani, Expressive stream reasoning with laser, in: *International Semantic Web Conference*, Springer, 2017, pp. 87–103.
[29] X. Ren, O. Curé, H. Naacke and G. Xiao, BigSR: real-time expressive RDF stream reasoning on modern Big Data platforms, in: *2018 IEEE International Conference on Big Data (Big Data)*, IEEE, 2018, pp. 811–820.
[30] P. Bonte, R. Tommasini, F. De Turck, F. Ongenae and E.D. Valle, C-sprite: efficient hierarchical reasoning for rapid RDF stream processing, in: *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*, 2019, pp. 103–114.
[31] T.-L. Pham, M.I. Ali and A. Mileo, Enhancing the scalability of expressive stream reasoning via input-driven parallelization, *Semantic Web* **10**(3) (2019), 457–474.
[32] D. Anicic, P. Fodor, S. Rudolph and N. Stojanovic, EP-SPARQL: a unified language for event processing and stream reasoning, in: *WWW 2011*, ACM, 2011, pp. 635–644.
[33] D. Luckham, *The Power of Events: An introduction to Complex Event Processing in Distributed Enterprise Systems*, Vol. 204, Addison-Wesley Reading, 2002.
[34] D. Dell'Aglio, M. Dao-Tran, J.-P. Calbimonte, D. Le Phuoc and E. Della Valle, A query model to capture event pattern matching in RDF stream processing query languages, in: *European Knowledge Acquisition Workshop*, Springer, 2016, pp. 145–162.
[35] J.-P. Calbimonte, J. Mora and O. Corcho, Query rewriting in RDF stream processing, in: *European Semantic Web Conference*, Springer, 2016, pp. 486–502.
[36] D. Puiu, P. Barnaghi, R. Tönjes, D. Kümper, M.I. Ali, A. Mileo, J.X. Parreira, M. Fischer, S. Kolozali, N. Farajidavar et al., Citypulse: Large scale data analytics framework for smart cities, *IEEE Access* **4** (2016), 1086–1108.
[37] F. Heintz, J. Kvarnström and P. Doherty, Bridging the sense-reasoning gap: DyKnow–stream-based middleware for knowledge processing, *Advanced Engineering Informatics* **24**(1) (2010), 14–26.
[38] D. Anicic, S. Rudolph, P. Fodor and N. Stojanovic, Stream reasoning and complex event processing in ETALIS, *Semantic web* **3**(4) (2012), 397–407.
[39] Ö.L. Özçep, R. Möller and C. Neuenstadt, A stream-temporal query language for ontology based data access, in: *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*, Springer, 2014, pp. 183–194.
[40] H. Stuckenschmidt, S. Ceri, E. Della Valle and F. Van Harmelen, Towards expressive stream reasoning, in: *Dagstuhl Seminar Proceedings*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2010.
[41] P. Bonte, R. Tommasini, E. Della Valle, F. De Turck and F. Ongenae, Streaming MASSIF: cascading reasoning for efficient processing of iot data streams, *Sensors* **18**(11) (2018), 3832.
[42] I. Kalamaras, N. Kaklanis, K. Votis and D. Tzovaras, Towards Big Data Analytics in Large-Scale Federations of Semantically Heterogeneous IoT Platforms, L. Iliadis, I. Maglogiannis and V. Plagianakos, eds, Springer International Publishing, Cham, 2018, pp. 13–23.
[43] P. Chamoso, A. González-Briones, F. De La Prieta, G.K. Venyagamoorthy and J.M. Corchado, Smart city as a distributed platform: Toward a system for citizen-oriented management, *Computer Communications* **152** (2020), 323–332. doi:https://doi.org/10.1016/j.comcom.2020.01.059. https://www.sciencedirect.com/science/article/pii/S0140366419321152.
[44] F. Cirillo, G. Solmaz, E.L. Berz, M. Bauer, B. Cheng and E. Kovacs, A standard-based open source IoT platform: FIWARE, *IEEE Internet of Things Magazine* **2**(3) (2019), 12–18.
[45] Sofia2. https://sofia2.com/.
[46] S. Soursos, I.P. Žarko, P. Zwickl, I. Gojmerac, G. Bianchi and G. Carrozzo, Towards the cross-domain interoperability of IoT platforms, in: *2016 European conference on networks and communications (EuCNC)*, IEEE, 2016, pp. 398–402.

[47] A. Felfernig, S.P. Erdeniz, P. Azzoni, M. Jeran, A. Akcay and C. Doukas, Towards configuration technologies for iot gateways, in: *18 th International Configuration Workshop*, Vol. 73, 2016.
[48] A. Bröring, S. Schmid, C.-K. Schindhelm, A. Khelil, S. Käbisch, D. Kramer, D. Le Phuoc, J. Mitic, D. Anicic and E. Teniente, Enabling IoT ecosystems through platform interoperability, *IEEE software* **34**(1) (2017), 54–61.
[49] A. Cimmino, V. Oravec, F. Serena, P. Kostelnik, M. Poveda-Villalón, A. Tryferidis, R. García-Castro, S. Vanya, D. Tzovaras and C. Grimm, VICINITY: IoT semantic interoperability based on the web of things, in: *2019 15th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, IEEE, 2019, pp. 241–247.
[50] M. Ganzha, M. Paprzycki, W. Pawłowski, P. Szmeja and K. Wasielewska, Semantic interoperability in the Internet of Things: An overview from the INTER-IoT perspective, *Journal of Network and Computer Applications* **81** (2017), 111–124.
[51] A. Javed, S. Kubler, A. Malhi, A. Nurminen, J. Robert and K. Främling, bIoTope: Building an IoT Open Innovation Ecosystem for Smart Cities, *IEEE Access* **8** (2020), 224318–224342. doi:10.1109/ACCESS.2020.3041326.
[52] M. De Brouwer, F. Ongenae, P. Bonte and F. De Turck, Towards a cascading reasoning framework to support responsive ambient-intelligent healthcare interventions
18(10), 3514.
[53] B. Steenwinckel, M.D. Brouwer and F. Ongenae, DAHCC: The Data Analytics in Healthcare and Connected Care ontology, IDLab, 2022, Available online: https://dahcc.idlab.ugent.be (accessed on 3 Feb 2022).
[54] L. Daniele, F. den Hartog and J. Roes, Created in close interaction with the industry: the smart appliances reference (SAREF) ontology, in: *International Workshop Formal Ontologies Meet Industries*, Springer, 2015, pp. 100–112.
[55] M. Girod-Genet, L.N. Ismail, M. Lefrançois, J. Moreira and M. Dragoni, ETSI TS 103 410-8 V1. 1.1 (2020-07)" SmartM2M; Extension to SAREF; Part 8: eHealth/Ageing-well Domain", PhD thesis, ETSI SmartM2M, 2020.
[56] I. Esnaola-Gonzalez, J. Bermúdez, I. Fernández and A. Arnaiz, Two Ontology Design Patterns toward Energy Efficiency in Buildings., in: *WOP@ ISWC*, 2018, pp. 14–28.
[57] D. Arndt, P. Bonte, A. Dejonghe, R. Verborgh, F. De Turck and F. Ongenae, SENSdesc: Connect sensor queries and context, in: *11th International Joint Conference on Biomedical Engineering Systems and Technologies*, pp. 1–8.
[58] R. Verborgh and J. De Roo, Drawing conclusions from linked data on the web: The EYE reasoner, *IEEE Software* **32**(3) (2015), 23–27.
[59] W3C, OWL 2 Web Ontology Language Profiles (Second Edition) – OWL 2 RL. https://www.w3.org/TR/owl2-profiles/#OWL_2_RL.
[60] D. Arndt, B. De Meester, P. Bonte, J. Schaballie, J. Bhatti, W. Dereuddre, R. Verborgh, F. Ongenae, F. De Turck, R. Van de Walle et al., Improving OWL RL reasoning in N3 by using specialized rules, in: *International Experiences and Directions Workshop on OWL*, Springer, 2015, pp. 93–104.
[61] Empatica, E4 wristband. https://www.empatica.com/research/e4/.
[62] J. Bai, C. Di, L. Xiao, K.R. Evenson, A.Z. LaCroix, C.M. Crainiceanu and D.M. Buchner, An activity index for raw accelerometry data and its comparison with other activity metrics, *PloS one* **11**(8) (2016), e0160644.
[63] EsperTech, Esper, Accessed: 2022-03-29. https://www.espertech.com/esper.