CONSERT : Applying Semantic Web Technologies to Context Modeling in Ambient Intelligence

Alexandru Sorici[a,b,*], Gauthier Picard[b], Olivier Boissier[b], Antoine Zimmermann[b], Adina Florea[a]

[a] *University Politehnica of Bucharest, Department of Computer Science, 313 Splaiul Independentei, 060042 Bucharest, Romania*
[b] *Ecole Nationale Supérieure des Mines, FAYOL-EMSE, LSTI , F-42023 Saint-Etienne*

Abstract

Representation and reasoning about context information is a main research area in Ambient Intelligence (AmI). Context modeling in such applications is facing openness and heterogeneity. To tackle such problems, we argue that usage of semantic web technologies is a promising direction. We introduce CONSERT, an approach for context meta-modeling, which builds and improves upon related lines of work (SOUPA, CML, annotated RDF). It results in a consistent and uniform means for working with domain knowledge, as well as constraints and meta-properties thereof. We provide a formalization of the model and detail its innovative implementation using techniques from the semantic web community such as Ontology modeling and SPARQL. A stepwise example of modeling a commonly encountered AmI scenario showcases the expressiveness of our approach. Finally, the architecture of the representation and reasoning engine for CONSERT is presented and evaluated in terms of performance.

Keywords: Ambient Intelligence, Context Modeling, Ontology, Semantic Web, SPARQL, Rule-Based Reasoning

1. Introduction

Ambient Intelligence (AmI) is nowadays a well recognized area of research with work done in domains ranging from hardware (e.g. sensors, actuators) through middleware (e.g. information management, basic services) to innovative end applications and human computer interfaces. The industry is also starting to embrace scenarios and ideas from the ambient intelligence domain, most notably in activity areas such as home monitoring and automation, smart city sensing and monitoring infrastructures. There is a growing number of start-up enterprises that are active in the mentioned areas (e.g. Ninja Sphere[1], Nest[2], SmartThings[3]) and increasingly more cities are offering their support for installing prototype smart environment infrastructures.
The possibilities for AmI application development increase even further, given the emerging industry enterprises that offer entire development platforms for creating application and business logic in the IoT and M2M (machine-to-machine) domains (e.g. Xively[4], ThingWorx[5]). Such initiatives open up a trend that leads towards systems which promote anonymous social experiences and focus on models of group activity rather than just individual ones. It raises an AmI that is centered on enhancing human interaction apart from supporting individual needs and preferences.

In terms of information and situation supervision (the Context Management research branch of AmI), the perspectives listed above translate to requirements of being able to support interoperability and openness. They also demand an increased expressiveness of contextual information models and the accompanying reasoning and query solutions. The growing complexity of situation definitions in AmI scenarios as promoted by the consideration of models for social activities and experiences drives the need for model expressiveness, whereas the decentralized and heterogeneous nature of devices and applications found in the IoT and M2M domains imposes the necessity for interoperability through standards.

With this in mind, the information management middleware and the way in which it handles the notions of context representation and reasoning, as introduced by Dey [1], are of first importance. The past decade has seen many contributions in these particular fields of research [2, 3, 4]. Recognizing the need for interoperability and standards in terms of languages

*Principal corresponding author
Email addresses: sorici@emse.fr (Alexandru Sorici), picard@emse.fr (Gauthier Picard), boissier@emse.fr (Olivier Boissier), antoine.zimmermann@emse.fr (Antoine Zimmermann), adina.florea@cs.pub.ro (Adina Florea)

[1] http://ninjablocks.com/
[2] https://nest.com/
[3] http://www.smartthings.com/
[4] https://xively.com/
[5] http://www.thingworx.com/platform/#how-it-works

and approaches, works have started focusing on ontology models in support of context modeling. The representation potential of description logics, the ability to ensure knowledge consistency and the support for reasoning make for compelling benefits. However, the majority of approaches for ontology-based context modeling focus on building either a very generic (e.g [5]) or a highly specific (e.g. [6]) vocabulary for the domain knowledge of a given AmI application. Consequently, many such models cannot support important design expressiveness aspects of domain statements such as arbitrary arity of predicates or well-structured representation and reasoning on statement meta-properties (e.g. quality metrics, temporal validity).

Furthermore, an inference process based solely on the capabilities offered by description logics is not sufficient to address the challenges of having a well-structured means for concomitantly reasoning over context domain knowledge, time dependent meta-properties and integrity constraints.

In this article we propose to address the earlier mentioned shortcomings by developing CONSERT, a context representation meta-model implemented as an ontology, which extends and combines previous works [7, 8], providing extensive design support for expressing statement meta-properties (annotations) and constraints. We furthermore define a context representation and reasoning engine (CONSERT Engine) which tackles the problems of *(i)* combining rule-based and ontological reasoning for domain knowledge, *(ii)* structured manipulation of context annotations and *(iii)* detection of context integrity violations. Thereby we make use of the latest proposals of the semantic web community for standards such as RDF and SPARQL in order to address the storing, querying and reasoning aspects. As we will see later on, the latter reliance on semantic web standards is our solution to the requirements of interoperability with third party applications and general openness in AmI systems.

The remainder of this paper is structured as follows. In Section 2 we analyze related work in the domains of context ontology models and semantic web approaches to reasoning over context information and point out the works and concepts that influenced our own approach the most. Section 3 presents an application scenario which we further use to illustrate examples of the notions we define along the way. We then start a formal definition of our proposed context model in Section 4, present its implementation in Section 5 and provide, in Section 6, a designer's guidelines for using our approach to model the scenario exposed earlier. Sections 7 and 8 describe the architecture of our proposed representation and reasoning engine as well as the validation and performance tests we carried out. The paper concludes with Sections 9 and 10, where we discuss the contributions, existing limitations and future directions of work.

2. Related Works and Foundations

The field of context modeling has received a noticeable amount of contributions over the last decade [2, 3] ranging from simple key-value models, through mark-up and graphical models and down to different ontology models [5, 7, 9]. As mentioned in the introduction, the need for expressive modeling and reasoning support has led to ontologies for context modeling becoming a focus in many research approaches. In the following we provide an overview of ontology based context models going through domain centric proposals, works on context meta-models and approaches of reasoning about context. Throughout the analysis we try to point out how the presented works address the concerns of expressiveness and interoperability that we mentioned as focus elements in the introduction. At the end of the section we briefly explain how our own proposal tries to collect and combine relevant influences from the listed research works.

2.1. Domain-Centric Context Models

As mentioned in the introduction, many lines of work tackled the problem of expressiveness by focusing on developing extensible and generic context modeling ontologies that would cover as many context domains as possible. Some of the works tried to also address the dimension of quality of context information.

CONON [9] defines a set of 14 classes that constitute its core vocabulary, which focuses on modeling persons, locations, activities and computational entities. It further defines a QualityConstraint class that can represent different quality aspects of context information such as accuracy or freshness. However, the authors in [9] do not make it clear how this meta-information is assigned to domain knowledge or how it is used in inferencing.

The CoOL ontology [5] is based on the Aspect-Scale-Context model (ASC) and it is intended to enable context-awareness and contextual interoperability. CoOL focuses on defining a more abstract and overarching vocabulary. The authors point out that it could be used to achieve transfers between arbitrary context models thereby allowing the model to act as an interoperability and comparison layer.

SOUPA[7] is a well-known context ontology model [10] which achieves great genericity and reusability by creating its core

vocabulary in a modular way, based on upper-level consensus ontologies that cover aspects like person (FOAF[6]), time (OWL-Time[7]) or space (spatial ontologies in OpenCyc[8]). Though originally used in applications centered on user activity modeling, we considered it general enough for use as the default upper-ontology in our model (see Section 5.1), able to describe entities involved in a wider variety of context management applications. More recent works ([6] and [11]) apply ontology modeling for human activity recognition. However, as mentioned in the introduction, they represent examples of targeted domain modeling. Specifically, Chen and Nugent [6] focus on providing an extensive vocabulary for Activities of Daily Living (ADL) and a reasoning algorithm, based on description logic, which works in an incremental fashion. With every step it builds an increasingly precise realization of the activity of a person, given sensory information that describes usage of every-day objects. The authors note however that they leave all modeling and reasoning related to temporal and meta-properties (e.g. quality of information) to future work.

Riboni and Bettini [11] examine the benefits of using the OWL 2 ontology language to build a vocabulary for human activity recognition. The authors analyze the way in which newer constructs available in OWL 2 (e.g. qualified cardinality restrictions, property composition) help to cover modeling efforts which had previously used a combination of OWL 1 and predicate logic, thereby reducing hybrid reasoning mechanisms to a single well-defined one. The work illustrates this by presenting an OWL 2 based model of ADLs for a smart home and smart workspace scenario. Still, the authors observe that their model cannot currently support the definition of context information quality metrics or easy handling of conflicting and incomplete information, while the *tree model property* condition [12] of OWL 2 limits the expressiveness of the language.

All of the works presented so far have put the *entities* of an application *domain* at the center of their model and tried to provide vocabularies that would cover as many context domain dimensions as possible. Only some of them ([5, 9]) offer support to characterize meta-properties of context information (e.g. quality information). However, they do not detail how these annotations are used during an inference process.

2.2. Context Meta Models

In contrast with the above mentioned approaches, other works propose focusing on *predicates* to describe the relations that exist between entities of an application domain. They apply the annotations (e.g. quality of information) aspect to entire context statements, rather than just entities. Most of them try to increase the expressiveness of their approaches by distinguishing between a base component (a meta-model realization), that provides a vocabulary for working with the different model elements and their properties, and an upper-component that, when extended, captures the different domain dimensions of a particular application field.

One such approach is mySAM [13] which introduces an ontology model able to define arbitrary context predicates. The principal model element is the ContextAttribute, which is a general construct that can express arbitrary statements of a context domain. The ContextAttribute has properties stating its arity, the list of entities over which it applies and the value(s) it returns. The model distinguishes between a context ontology and a domain ontology. The domain ontology is used to define the actual concepts that pertain to an application domain. The context ontology contains all the ContextAttributes that apply over the domain concepts. The approach is flexible in terms of its domain modeling expressiveness, but the work does not attempt to model quality (meta-properties) of the ContextAttributes, nor does it specify how reasoning is performed with the given constructs.

Fuchs et al. [14] propose a Context Meta Model capturing semantics of entities, properties and quality classes that characterize the properties. Similar to the ASC model [5], the meta model defines DataStructure classes and transformation rules that can convert from one DataStructure to another. Predicate dependencies and derivation rules are also specified in order to perform inferencing. However, the OWL-DL instantiation that the authors give to their Context Meta Model deals only with binary predicates and uses rules defined in SWRL to accomplish derivation of higher level context information, thereby limiting the expressiveness of the reasoning approach.

The work in [14] is similar to another proposal of particular interest, the Context Modeling Language (CML [8]). CML builds upon the Object-Role Model (ORM) conceptual language for data modeling used in the Relational Database domain and extends it with constructs specific to the area of context representation. The basic representational unit in CML is the *fact*, a relationship holding between one or more entities, categorized into static, sensed, profiled or derived depending on the acquisition type. It allows expression of uniqueness constraints and fact dependencies as well as annotation of facts with quality indicators. CML also introduces a form of first-order predicate logic used to derive higher-level information (called situations). The model's

[6] http://www.foaf-project.org/
[7] http://www.w3.org/TR/owl-time/
[8] http://www.cyc.com/opencyc

realization however was based on Relational Database schemas, thereby missing out on advantages given by implementations with a semantic dimension. A further result of this choice was that CML did not complement its context meta model with an upper-level component that would capture the concepts of a given application domain.

The shift towards predicates as first class elements might render the presented models more complex in practical use. However, we consider that the benefit of explicit access to meta-properties of context information outweighs the modeling overhead, since it introduces a more feature rich basis for inferring contextual situations. We analyze some of the different reasoning approaches in what follows.

2.3. Reasoning about Context

Most works presented above made use of ontology definition languages (more specifically OWL-DL) to represent their models. This immediately offered the benefits of the possible description logic entailments, most notably those of maintaining knowledge consistency and instance realization.

Apart from ontological approaches, many other works have taken a logic-supported rule-based view of context reasoning. For instance, SAGE [15] proposes a system using both forward chaining deductive reasoning and abductive reasoning to implement environment monitoring and control.

Toninelli et al. [16] develop an access control policy model that exploits context-awareness for the specification and evaluation of the defined policies. Context-awareness in their approach is implemented by a combination of ontology modeling and logic programming to overcome limitations of pure ontology reasoning.

Bikakis and Antoniou [17] use a reasoning mechanism based on the Multi-Context Systems paradigm extended with non-monotonic logic features (defeasible local rules, defeasible mapping rules) and a preference ordering mechanism in order to handle unknown, uncertain and conflicting context information. Whilst our reasoning approach does not handle non-monotonic reasoning, we address uncertain information by means of reasoning over explicit certainty annotations and manage conflicting information by detecting integrity constraint violations and using a policy-based resolution mechanism.

We note that the works listed above are not specific on how or whether annotations (e.g. timestamp of creation, duration, source, quality metrics) of the processed context information are represented or used during inference. Many AmI scenarios involve utilization of sensor data which may have inherent inaccuracies. Furthermore, complex activity recognition requires the capability to reason over the temporal order and duration of context situations. A mechanism to explicitly represent, query and reason about such annotations is therefore a clear benefit.

Additionally, many of the works that combine an ontology-based representation with a logic-based rule system require a transformation of knowledge representation to be performed before executing inference, thereby incurring additional runtime overhead.

A more recent approach for driving inference is found in [18], where an ontology for modeling complex activities is proposed. Though restricted to the domain of activity definition, the authors propose using SPARQL through its CONSTRUCT queries as a rule language that helps reason about composition of simpler activities into more complex ones. The inherent expressiveness of the SPARQL query syntax is the main advantage of this approach, while an additional benefit is its reliance on a standard of the semantic web community which favors interoperability.

Further works (e.g. EP-SPARQL [19]) considered enhancing the SPARQL standard with the ability to perform temporal reasoning tasks commonly found in event-processing systems. Others [20] made it an integral part of semantic event-driven systems, which combine static background knowledge modeled using ontologies with the complex dynamics of event processing systems.

Our reasoning engine proposes an architecture and process flow similar to that of the CROCO ontology-based context management service [21]. In contrast with Croco however, our approach exposes context information meta-properties and constraint detection rules directly as part of the vocabulary of the CONSERT ontology used to build the context model of an application, thereby providing a more unified design experience.

Lastly, both Croco and the majority of works in Section 2.2 offer support to further characterize context information with meta-properties (annotations) and some approaches [14] also provide vocabularies to specify the particular data types for each considered annotation. However, the mentioned works do not detail a means by which these meta-properties can be used in a structured and uniform way during an inference process (i.e., how to combine annotations of existing context information when deriving the ones for the inferred statement).

To complement this, we considered taking insights from the domain of annotated RDF [22, 23], itself inspired by Annotated Logic Programming [24], where formalization in terms of algebraic structures is provided for the annotation language and the corresponding deductive system. More specifically, Zimmermann et al. [22] defines representation forms and specific operators that are used to combine annotations of RDF statements during RDFS inferencing.

2.4. Foundations of our Approach

In the earlier three sections we have seen different aspects of context model approaches and what contributions each can bring with respect to expressiveness and interoperability. Our proposed CONSERT context model and associated reasoning engine detailed in the sections that follow tries to combine the best out of three worlds.

We present an ontology based solution that similar to [14] has an OWL-DL vocabulary to refer to meta-model elements including annotations and integrity constraints. Unlike [14] but similar to [8] we offer support to express context information statements of arbitrary arity and categorize them into sensed, profiled and derived ones. However, in contrast to [8], our meta-model elements are realized as an ontology which adds the inherent reasoning benefits. Additionally, we use an adapted version of SOUPA[7] as our default upper-ontology for giving a grounding to entities of the context domain of an application.

In our reasoning engine we employ a rule-based deduction approach using SPARQL CONSTRUCT queries coupled with ontology reasoning. The constructs that can be used within the SPARQL WHERE statements (e.g. term comparisons, aggregation, existence verification) alongside usual basic graph patterns ensure the expressiveness of the inference language. Similar to [18], we represent the rules in an RDF serialization using a semantic web proposal called SPIN[9] (SPARQL Inferencing Notation). The inference mechanism moves in the direction of semantic event processing and, while it currently cannot match the processing performance reported in works such as [19, 20], it offers important built-in modeling and reasoning capabilities not present in the cited works (e.g. determining the continuity of a context situation, detecting context integrity constraints - detailed in Section 7). These choices eliminate the need for knowledge representation transformations during inference and promote interoperability through use of standards.

Lastly, in a manner similar to [22], we consider giving structure to the different context information annotations, setting the requirement to define specific operators that help to manipulate and combine annotation information in the derivation process. Our reasoning engine then presents the benefits this approach can bring with respect to information consistency and integrity.

3. AmI Demonstrative Scenario

Before detailing our formal context modeling approach and its implementation using semantic web technologies, we introduce a scenario which is further used throughout the paper to exemplify the subjects we discuss. We use a fairly known and sought after AmI scenario: the smart conference application. We formulate the description as a series episodes that capture functionality from several points of view: user profile settings, session management, networking.

Aspect 1: User Profiles

Rebecca is a first time participant to the AmI conference. Upon registration she also installs a smartphone application that can support all of her interactions during the event whether on site or remotely. During setup users have the option of building a personal profile, visible only to other conference participants, that includes online contact information (e.g. email, links to professional social networks), data about affiliation and research interests. This information helps to compute participant similarities, which are used to provide recommendations for contact or short ad-hoc discussion sessions between the participants. Rebecca can also indicate what *role* she will play in the conference: participant, presenter or session chair. She indicates she is a presenter and the application offers her the option to upload or provide a URI towards the content of her presentation. This will make it easier to retrieve it faster later on. Meanwhile, Ted arrives at the registration desk, installs the application and logs in as a session chair, which opens up further functionality for him.

The user smartphones also act as an indoor localization means, since the conference is equipped with low-energy bluetooth wireless location beacons, which allow estimating the whereabouts of conference participants down to the level of specific interest areas (e.g. speaker area within a session room, the reception desk, ad-hoc discussion areas).

Aspect 2: Session Management

The first session is about to start. Ted is chairing and he starts introducing Rebecca as the first speaker. He then marks her presentation as active through the application and this automatically sets up an internal clock which will help Ted to keep track Rebecca's presentation time. Rebecca walks to the front of the room and is detected by a location beacon as being in the speaker area. Since Ted marked her presentation as active, the application concludes that her presentation must start and automatically sends Rebecca's slides to the computer connected to the session room's projector.

[9] http://spinrdf.org/

During the presentation, all persons detected as being in a session room with an active presentation are deemed by the application as being busy and a default action is to automatically mute the ringtone of the phone.
Ted receives a notification as Rebecca's presentation nears its end. Her work was very interesting so she receives a lot of questions. Ted agrees to prolong her presentation time and indicates a 5 minute extension within the application, such that the following presentations get rescheduled correspondingly.

Aspect 3: Ad-Hoc Discussion

Using the similarity check functionality, the application determines that Rebecca and Steve, another conference participant, have a very high match. Both receive a recommendation to get in contact with each other. The application suggests they meet at an ad-hoc discussion area that is closest to their current position. As Rebecca and Steve arrive and start talking, based on the proximity data and noise level picked up by nearby embedded microphones the support application concludes that the users have started a discussion. This again automatically changes their smartphone availability status such that their conversation is not interrupted.

While most of the previously described situations do not represent totally new scenarios for conference management applications, they do contain a level of complexity that allows us to demonstrate the benefits of our approach in terms of modeling context domain statements, manipulating annotations, assigning constraints and expressive inference rules. Section 6 makes an explicit possible design of the scenario with help from our model.

4. CONSERT Context Model

We now present the formal model of context representation and reasoning (called CONSERT, an abbreviation from CONtext asSERTion), focusing on the key elements. Each concept is exemplified from the scenario of the previous section. The implementation of this model based on semantic Web Technologies is presented in Section 5.

4.1. Overview

The model of context that we propose comprises different elements that are related to each other (cf. Figure 1). Before giving a formal definition as well as properties of each element, let's give a global overview of this model.

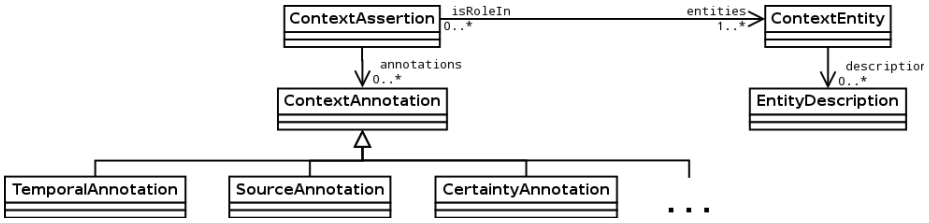

Figure 1: The defining concepts of our proposed context model.

A *ContextAssertion* represents the basic construct used to describe the situation of entities (e.g. a "person", a "place", an "object") "which are considered relevant for the interaction between a user and an application" (in the sense of Dey's definition of context [1]). Each assertion involves one or more *ContextEntities*. For instance, the two *ContextEntities*: *Person*(steve) (i.e., Steve is a person) and *SessionRoom*(sessionRoom1) (i.e., sessionRoom1 is a session room) may be related together through a *ContextAssertion* describing the situation of being located some place by the following expression: personLocatedAt(steve, sessionRoom1). *ContextAssertion* and a *ContextEntity* may be further characterized by *ContextAnnotation* and *EntityDescription* respectively.
A *ContextAnnotation* is a meta-property of a *ContextAssertion* and relates to information such as the source (author of the statement), the timestamp of its generation, the validity of the statement (time intervals for which the assertion is considered to be true) or the certainty with which the assertion is affirmed. These are the most often encountered annotation examples in the literature, but our model does not impose a limit on the type of possible *ContextAnnotations*. Other more complex properties can be imagined such as ownership (one or more entities which "hold control" of an assertion), access control (who is allowed to query or access the value of the assertion) or others.
An *EntityDescription* represents *static* information (i.e. does not vary with time) that provides additional characterization of a *ContextEntity* (e.g. spatial inclusion and distance relations, temporal relations, descriptive properties). It therefore holds between a *ContextEntity* and a literal value. For example, consider modeling the location of a user as personLocatedAt(steve,

ami_lab). We further know that includedIn(ami_lab, cs_building) and hasCoordinates(cs_building, geoCoordinates), where *Person*(steve), *SpatialStructure*(ami_lab, cs_building) are *ContextEntities* and geoCoordinates is a literal. The statement personLocatedAt represents a *ContextAssertion*, having dynamic value changes for each individual person, whereas includedIn and hasCoordinates are modeled as *EntityDescriptions* since they only provide additional static descriptions of a *SpatialStructure* instance.

Lastly, the model comprises a reasoning mechanism to derive higher-level *ContextAssertions* (i.e., situations with a more complex semantics) based on existing knowledge. The reasoning takes the form of so called *Context Derivation Rules*, a deduction rule system able to express conditions over *ContextAssertions*, *EntityDescriptions* and *ContextAnnotations*. The formal definition of *Context Derivation Rules* is given in Section 4.3.

4.2. Context Representation Elements

Let us now introduce the formal definitions and notations which will later serve in expressing the form of rules that are used to drive context situation inference in our model. Let's start with the definition of a context entity which is based on Dey's definition [1].

Definition 4.1 (ContextEntity)**.** *A ContextEntity is any physical, virtual or conceptual element that is considered relevant to the interaction between a user and an application, including the user and the application themselves.*
Formally, we denote with E the set of all *ContextEntities* that are considered within a model instance.

In order to properly introduce the remaining model elements we consider the following additional notations: V the alphabet of variables, L the alphabet of literals and A_d the one of a *ContextAnnotation* domain d (discussed further in definition 4.4). Further, let $\mathcal{F}$ be the set of all *ContextAssertions* and $\mathcal{A}$ be the union of all *ContextAnnotation* domains A_d.

Definition 4.2 (ContextAssertion)**.** *A ContextAssertion is an n-ary relation of the form* $F(x_1, x_2, \ldots, x_n) : \{\lambda_1, \lambda_2, \ldots, \lambda_m\} \in \mathcal{F}$, *where* $x_i \in E \cup L \cup V, i = 1..n$ and $\lambda_j \in A_{d_j}, j = 1..m$.

The function $entities : \mathcal{F} \rightarrow 2^{E \cup L \cup V}$ returns the set of *ContextEntities*, literals or variables which play a *role* in an assertion, i.e. $entities(F) = \{x_1, \ldots, x_n\}$. Similarly, the function $annotations : \mathcal{F} \rightarrow 2^{\mathcal{A} \cup V}$ returns all *ContextAnnotations* (instances or variables) which describe a *ContextAssertion* F, i.e. $annotations(F) = \{\lambda_1, \ldots, \lambda_m\}$.
Each *ContextAssertion* may also be qualified by *uniqueness* and *value constraints* which are similar to the ones defined for the Relational Database domain and serve the same purpose of maintaining the integrity of the knowledge base. It is important to note that we allow these constraints to be defined over a subset $K \subseteq entities(F) \cup annotations(F)$ of both context entities and annotations, meaning that annotations are important in determining the consistency and integrity of *ContextAssertions*.

Definition 4.3 (EntityDescription)**.** *An EntityDescription is a binary relation of the form* $D(x, y)$, *where* $x \in E \cup V$ *and* $y \in E \cup L \cup V$.

As explained earlier in the overview of our model elements, *EntityDescriptions* represent further characterization statements of a *ContextEntity* x. Notice also that the first parameter of D is always a *ContextEntity* or a variable holding its place, whereas the second one can also be a literal value.

Definition 4.4 (ContextAnnotation)**.** *A ContextAnnotation domain* A_d is an idempotent, commutative semi-ring $\langle A_d, \oplus, \otimes, \bot, \top \rangle$, where the operators $\oplus$ and $\otimes$ have the following properties:

- $\oplus$ is idempotent, commutative, associative;
- $\otimes$ is commutative and associative;
- $\bot \oplus \lambda = \lambda$, $\top \otimes \lambda = \lambda$, $\bot \otimes \lambda = \bot$, and $\top \oplus \lambda = \top$
- $\otimes$ is distributive over $\oplus$, i.e., $\lambda_1 \otimes (\lambda_2 \oplus \lambda_3) = (\lambda_1 \otimes \lambda_2) \oplus (\lambda_1 \otimes \lambda_3)$;

As we have mentioned in Section 2, the definition of a *ContextAnnotation domain* A_d is inspired from work on annotated RDF [22]. The reason for this choice relies in the ability to obtain a structured way of combining *ContextAssertions* during rule based inferencing by making use of the $\oplus$ and $\otimes$ operators. Following the observation in [22], we use $\oplus$ to combine information about the same *ContextAssertion*, whereas $\otimes$ models the conjunction of two different annotated statements.
For instance, in an example from the domain of temporal validity:

Example 4.5.
Let $f : \{[12{:}00,\ 12{:}05]\}$ and $f : \{[12{:}03,\ 12{:}08]\}$ be two instances of the same *ContextAssertion* with different temporal validity. We infer $f : \{[12{:}00,\ 12{:}05] \cup [12{:}03,\ 12{:}08]\} = f : \{[12{:}00,\ 12{:}08]\}$, where $\cup$ plays the role of $\oplus$.

On the other hand, when dealing with a conjunction of statements with different type, a similar example to the previous one gives:

Example 4.6.
Let $f_1 : \{[12{:}00,\ 12{:}05]\}$ and $f_2 : \{[12{:}03,\ 12{:}08]\}$ be two different *ContextAssertions*.
We can infer $f_1 \wedge f_2 : \{[12{:}00,\ 12{:}05] \cap [12{:}03,\ 12{:}08]\} = f_1 \wedge f_2 : \{[12{:}03,\ 12{:}05]\}$, where in this case $\cap$ plays the role of $\otimes$.

In what follows, we briefly present the form of the annotation domains we chose for the common settings (timestamp, time validity, certainty) which we discussed earlier in this section (partly adopted from [22]):

Annotation	Value set	Algebraic form
timestamp	set of time points $\cup \{-\infty, +\infty\}$	$\langle A_{timestamp}, max, min, -\infty, +\infty \rangle$
validity	set of sets of pairwise disjoint time intervals	$\langle A_{validity}, \cup, \cap, \emptyset, [-\infty, +\infty] \rangle$
certainty	$[0, 1]$	$\langle A_{certainty}, max, min, 0, 1 \rangle$

In the current version, the source annotation is kept simple (a URI identifying a service or actor that produces a *ContextAssertion* instance, or one that performs a derivation using our reasoning engine) and not modeled as a structured annotation ($\oplus$ and $\otimes$ are not defined). In section 5.1 we show it to be a subclass of a BasicAnnotation. The other provided annotation domain definitions have an intuitive interpretation of their respective $\oplus$ and $\otimes$ operators. The timestamp domain considers time points as its vocabulary. The additional $-\infty$ and $+\infty$ are used to complete the formal definition of the timestamp vocabulary. Based on the natural ordering of timestamps, *max* and *min* play the roles of $\oplus$ and $\otimes$.
We have already seen the form of the temporal validity annotation domain in Examples 4.5 and 4.6. The domain vocabulary consists of consecutive disjoint time intervals, where each end of an interval is a time point or $-\infty$ or $+\infty$. As noted in the examples, the roles of $\oplus$ and $\otimes$ are played by $\cup$ and $\cap$ respectively.
Lastly, our definition for the certainty annotation domain uses decimal values from the interval $[0, 1]$ as vocabulary for expressing certainty values. Using the natural order of real numbers, we can use *max* and *min* as the $\oplus$ and $\otimes$ operators. Note however that this choice is not unique, as another option for $\otimes$ could be the multiplication operator $\times$ for real numbers.

4.3. Context Derivation Rules

We now focus on the proposed reasoning model. It aims at combining *EntityDescriptions*, *ContextAssertions* and their annotations in order to obtain higher-level *ContextAssertions*. The reasoning is based on a deduction mechanism involving *Context Derivation Rules*. Each rule is made up of a head (the deduced *ContextAssertion*) and a body which expresses the conditions required for the rule head to be deduced.
The head of a derivation rule ρ is a *ContextAssertion* $F(x_1, \ldots, x_k) : \{\lambda_1, \ldots, \lambda_l\}$ where $x_i \in E \cup V \cup L$ and $\lambda_j \in A_{d_j} \cup V$. Notice that *entities*$(F)$ and *annotations*(F) can include variables which will be bound during the reasoning process.
The body consists of so called *ConditionExpressions* (detailed later in this section) and constrained forms of universal and existential quantification.
We next introduce three auxiliary functions that help us to better present the formal definition of a *Context Derivation Rule* ρ. Let $\mathcal{R}$ be the set of *Context Derivation Rules*. The function $head : \mathcal{R} \rightarrow \mathcal{F}$ retrieves the head of a rule, the *ContextAssertion* that is inferred. Similarly, the function $body : \mathcal{R} \rightarrow 2^{\mathcal{F}}$ retrieves the set of all *ContextAssertions* contained in the body of a *Context Derivation Rule*. Lastly, the function $constraint : \mathcal{R} \rightarrow \mathcal{F}$ gets the *ContextAssertion* that provides the expression for the constrained universal or existential quantification. A *Context Derivation Rule* is then defined as follows:

Definition 4.7 (Context Derivation Rule)**.**

$$\begin{aligned} \rho : F(x_1, x_2, \ldots, x_k) : \{\lambda_1 \lambda_2, \ldots, \lambda_l\} \leftarrow body \qquad & \text{where body may be:} \\ ConditionExpr & \\ or \quad \exists y_1, \ldots y_r \bullet F_c(z_1, \ldots, z_m) : \{\lambda_1, \ldots \lambda_p\} \bullet ConditionExpr \quad & (EQC) \\ or \quad \forall y_1, \ldots y_r \bullet F_c(z_1, \ldots, z_m) : \{\lambda_1, \ldots \lambda_p\} \bullet ConditionExpr \quad & (UQC) \end{aligned}$$

where $y_i \in V$, $Y_\rho = \{y_1, \ldots y_r\} \subseteq entities(constraint(\rho)) \cup annotations(constraint(\rho))$ and $entities(head(\rho)) \cap entities(constraint(\rho)) \neq \emptyset$.

In the above rule, EQC (resp. UCQ) refers to existential (resp. universal) quantification constraint, meaning that the possible values of the variables y_i are those for which the constraining *ContextAssertion* F_c is true:

- In the existential case, at least one value assignment for each y_i has to also observe the conditions set in *ConditionExpr*.
- In the universal case, all possible value assignments have to do so.

Additionally, Y_ρ and all the variables that appear in the rule head ($entities(head(\rho))$, $annotations(head(\rho))$) must also appear in *ConditionExpr*, which we discuss next.

Definition 4.8 (*ConditionExpr*)**.** A *ConditionExpression* contains a domain expression (*DomExpr*) and an annotation expression (*AnnExpr*) as follows:

$$
\begin{aligned}
ConditionExpr &::= DomExpr \wedge AnnExpr \\
DomExpr &::= ComExpr \mid DomExpr \wedge ComExpr \\
ComExpr &::= SimExpr \mid AggExpr \\
SimExpr &::= AssertExpr \mid \neg AssertExpr \mid DescExpr \mid \neg DescExpr \mid TermExpr \\
AggExpr &::= aggregate(FuncExpr, FilterExpr, ResExpr) \\
FilterExpr &::= SimExpr \mid FilterExpr \wedge SimExpr \\
AssertExpr &::= F(x_1, \ldots, x_n) : \{\lambda_1, \ldots, \lambda_m\},\ x_i \in E \cup V \cup L,\ \lambda_j \in A_{d_j} \cup V \\
DescExpr &::= D(x, y),\ x \in E \cup V,\ y \in E \cup L \cup V
\end{aligned}
$$

Definition 4.9 (*DomExpr*)**.** A *DomExpr* is a conjunction of positive or negated *ContextAssertions* (*AssertExpr*) and *EntityDescriptions* (*DescExpr*), term expressions (*TermExpr*) and aggregations (*AggExpr*).

Definition 4.10 (*TermExpr*)**.** Term expressions contain terms $t \in E \cup L \cup V \cup A_{d_i}$ which are entities, literals, variables or annotations. Terms can be related by boolean comparators ($>$, $<$, $\geqslant$, $\leqslant$, $=$, $\neq$), logical connectors ($\wedge$, $\vee$, $\neg$) and system or user-defined functions $func(t_1, \ldots, t_n)$. Functions in term expressions act as predicates which return a truth value when all their arguments are bound. If the arguments contain free variables, the function call binds them to values that make the function true.

Definition 4.11 (*AggExpr*)**.** An *aggregation expression* contains three subexpressions: *FuncExpr*, *FilterExpr* and *ResExpr*. The *FuncExpr* is a list of one or more aggregation functions that take a single variable as their argument [$aggFunc_1(z_1), \ldots, aggFunc_k(z_k)$]. We employ the typical aggregation functions: $aggFunc \in \{count, sum, avg, min, max\}$. *FilterExpr* is the expression used to condition the values of the variables z_i over which we perform the aggregation. It takes the form of a conjunction of *SimExpr*. Therefore, all variables z_i must occur in *ContextAssertions*, *EntityDescriptions* or *ContextAnnotations* contained in *FilterExpr*. Finally, *ResExpr* is a list of variables [$aggRes_1, \ldots, aggRes_k$] which will store the result of the k $aggFunc_i(z_i)$ functions.

Definition 4.12 (*AnnExpr*)**.** An *annotation expression* is a conjunction of functions of the form $f_j(\lambda_{j1}, \ldots, \lambda_{jq})$ where each function f_j binds a free variable $\lambda_j^{head} \in annotations(head(\rho))$ (the annotations of the *ContextAssertion* in the rule head). All λ_{jk} and λ_j^{head} belong to the same annotation domain A_{d_j} and we additionally know that λ_{jk} belongs to the annotations of some *ContextAssertion* in *ConditionExpr*. The functions f_j are user-defined and can either directly bind λ_j^{head} to a value from the annotation domain A_{d_j}, or they can determine their output by computations using the $\oplus$ and $\otimes$ operators specific to annotation domain A_{d_j}.

Two examples of the usage of all the above definitions for *Context Derivation Rule* expressions can be found in Section 6.2 in the context of a smart conference scenario.

5. CONSERT Context Implementation

We stated in the introduction that one of our important contributions in this work was the definition of an ontology model for context representation which provides an expressive language to formulate context domain knowledge and support for expressing information about domain statement meta-properties (annotations) and constraints.

In this section, we present the CONSERT Ontology which gives an ontological form to all the key context model elements introduced in the previous section. We further show the way in which we employ additional semantic web technologies (as mentioned in the introduction) to handle constraint definitions and inference as well as the structured manipulation of domain and annotation knowledge. More specifically, we explore the usage of RDF quadstores and the concept of Named Graphs to identify individual *ContextAssertions* and state their *ContextAnnotations*, as well as the definition of constraints and *Context Derivation Rules* using the SPARQL Inferencing Notation (SPIN) proposal [**?**].

5.1. Ontology Definition

The CONSERT ontology has three important subparts which are depicted in Figure 2. We distinguish the core vocabulary,

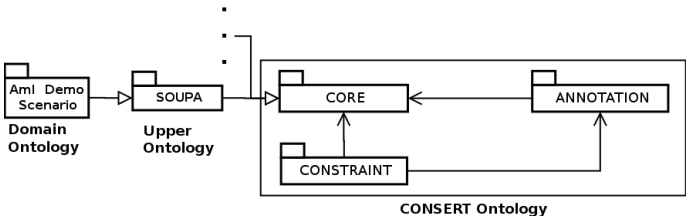

Figure 2: Components and their associations in the CONSERT ontology

which allows expressing context domain statements, the annotation vocabulary, which supports formulating *ContextAnnotations*, and the constraint vocabulary, which helps to express the causes of a uniqueness or value constraint violation. In Section 2 we discussed about SOUPA [7] as being our default choice for the upper-ontology that provides the grounding of the *ContextEntities* in our modeling of context. This fact is illustrated as well in Figure 2, but we remind that this is not a unique option and that given a specific context domain, a different ontology might be used as grounding. Based on the example of SOUPA, we show here how the definitions of the upper-ontology are extended in order to be coupled with the CONSERT ontology.

Core Vocabulary

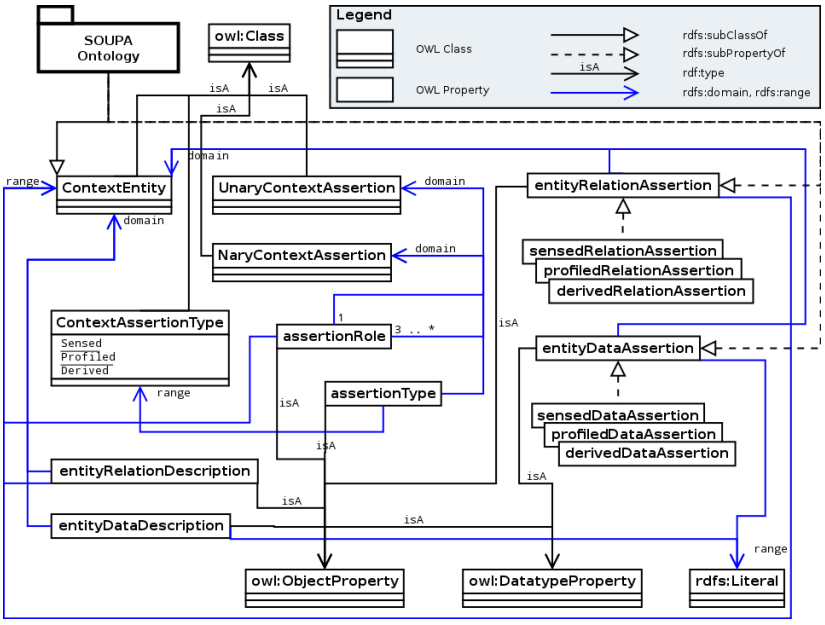

Figure 3: CONSERT ontology core vocabulary

Figure 3 shows a class-like diagram representation of the CONSERT core vocabulary. The ontology defines the generic class ContextEntity that becomes the new root for all classes in upper-ontology, in our case SOUPA.

In Section 4.1 we introduced the concepts of *ContextAssertion* and *EntityDescription* which are relations over *ContextEntities* and Literals. In order to characterize the binary assertions and descriptions which already exist between classes of SOUPA, in the CONSERT ontology we define two OWL object properties (entityRelationAssertion, entityRelationDescription) and two datatype properties (entityDataAssertion, entityDataDescription) that help us to "classify" SOUPA's object and datatype properties as either *ContextAssertion* with arity $n = 2$ (subproperties of $*$Assertion) or *EntityDescription* (subproperties of $*$Description).

Taking inspiration from work in [8], we further extend entityRelationAssertion and entityDataAssertion into properties that specify whether an assertion has been acquired from physical or virtual sensors (sensed), directly specified by the system or the user (profiled) or deduced by an inference (derived).
To express *ContextAssertions* with arities $n = 1$ or $n \geqslant 3$ we introduce two new classes within the CONSERT ontology: UnaryContextAssertion ($n = 1$) and NaryContextAssertion ($n \geqslant 3$). For both these cases we use a mechanism which is similar to reification of RDF statements[10]. In the CONSERT ontology we define the assertionRole property relating an instance of a UnaryContextAssertion or NaryContextAssertion to a ContextEntity or Literal which plays a role in the assertion.

Example 5.1 (UnaryContextAssertion example)**.**
For the unary case, a *ContextAssertion* like inAdHocDiscussion(*steve*) entails the creation of the inAdHocDiscussion subclass of UnaryContextAssertion and the assertion of the statements (in Turtle syntax):

```
{
    []  a :inAdHocDiscussion;
        :assertionRole :steve.
}
```

where [] represents a blank node.

Example 5.2 (NaryContextAssertion example)**.**
Taking a possible example from the scenario introduced in Section 3, in order to express a *ContextAssertion* like sensesBadgeWithIntensity(proxBeacon, userBadge, strong), where *ProximityBeacon*(proxBeacon) and *ConferenceBadge*(userBadge) are *ContextEntities* and strong is an instance from an enumeration (e.g. {weak, medium, strong}), we first create the sensesBadgeWithIntensity subclass of NaryContextAssertion together with subproperties of assertionRole specifying its roles (sensorRole, badgeRole, intensityRole). The *ContextAssertion* in our example would be then expressed as the following group of statements:

```
{
    []  a :sensesBadgeWithIntensity;
        :sensorRole :proxBeacon;
        :badgeRole  :userBadge;
        :intensityRole :strong.
}
```

For instances of UnaryContextAssertion and NaryContextAssertion, the CONSERT ontology also defines the assertionType property (whose range is an owl:unionOf class called ContextAssertionType) which states if they are sensed, profiled or derived (like in the $n = 2$ case).

Annotation Vocabulary

We move forward to detailing the concrete means of representing *ContextAnnotations* with a focus on the definitions for the annotation domains discussed throughout the article (source, timestamp, time validity and certainty). These are the ones most commonly used in the literature. They allow inspection of the temporal relationships between detected situations and support inference and query time decision making, by identifying the origin of the information and its quality metrics. Figure 4 presents the annotation vocabulary of the CONSERT ontology. The classes are defined with extensibility in mind, such that context model designers may have the ability to append new *ContextAnnotation* definitions according to their need. In Figure 4 we observe that two distinct types of annotations are modeled: BasicAnnotation and StructuredAnnotation. BasicAnnotation is the class meant to describe *ContextAnnotations* which do not have (or need) a structured manipulation during inference. At the end of Section 4.2 we mentioned that the source annotation is an example of a BasicAnnotation (as can be seen also in Figure 4). The hasUnstructuredValue functional property provides the actual value of a BasicAnnotation. It falls within the charge of the developer to specify the corresponding ontology class representing the range of this property for each subclass of BasicAnnotation defined, as well as providing the custom implementation of the annotation expression function (cf. Definition 4.12) that is used to specify the value of this type of annotation for a *ContextAssertion* derived during inference.
The other main annotation vocabulary component is the StructuredAnnotation. Its direct subclasses (called base structured

[10] http://www.w3.org/TR/rdf-schema/#ch_reificationvocab

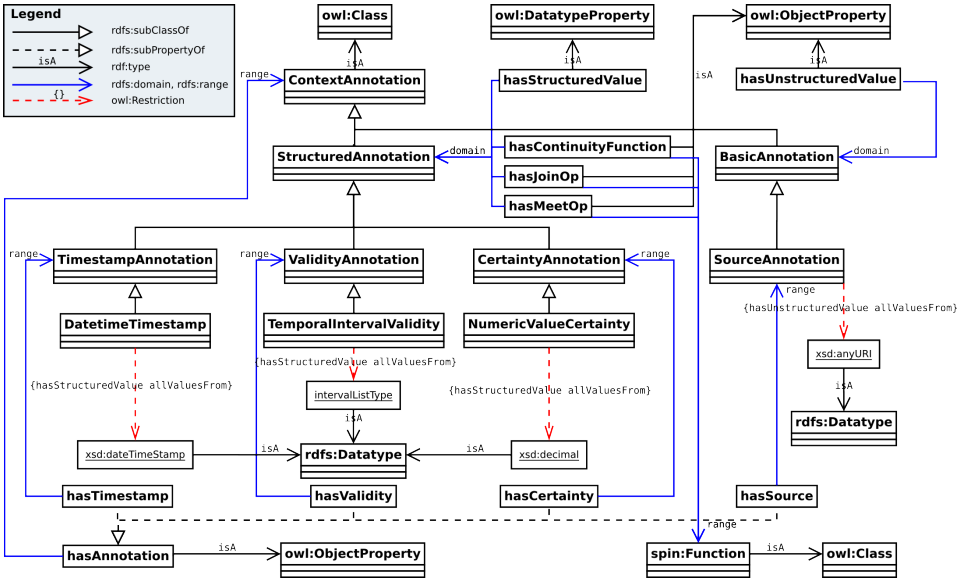

Figure 4: CONSERT ontology annotation vocabulary

annotation classes) represent a given annotation domain (cf. Definition 4.4). Derivatives of a base annotation class (e.g. DatetimeTimestamp, NumericValueCertainty, TemporalIntervalValditiy describe means of concretely expressing the value set of an annotation domain. For example, the default value set for the timestamp annotation domain is made up of xsd:dateTimeStamp instances. It may be however that for a particular application scenario the timestamps are much more suitably modeled as simple integer values, providing just relative order of events instead of an explicit time measurement. In this case, an application designer could extend the TimestampAnnotation base annotation class with one called IntegerTimestamp. This new class would define a restriction over the hasStructuredValue property, stating that all its values must be integers.

A *ContextAssertion* URI is bound to a *ContextAnnotation* of a particular domain by means of the corresponding subproperty of hasAnnotation (e.g. hasValidity for the validity domain, hasCertainty for the assertion certainty domain). The actual value of a *StructuredAnnotation* instance is given by the hasStructuredValue property. Each derivative of a base annotation class comprises in its definition an OWL allValuesFrom restriction which states the corresponding (standard or customly defined) rdfs:Datatype instances (e.g. xsd:datetime, ctx:intervalListType) that denote the value set of the given annotation domain. The restriction definition can be used at runtime to ensure that all the *ContextAnnotation* instances of a newly inserted *ContextAssertion* provide a correct value.

Next we introduce the rdfs:Datatype instance corresponding to each of the discussed annotation domains. The value for the *source* basic annotation consists in a URI (rdfs:Literal of type xsd:anyURI) identifying authors of *ContextAssertions* (services, actors). For the timestamp annotation domain, the vocabulary $A_{timestamp}$ consists of the set of date timestamp strings. The default CONSERT ontology specification defines xsd:dateTimeStamp as the datatype for timestamp literals.
In the case of the time validity annotation domain, an element of the vocabulary $A_{validity}$ is a set of pairwise disjoint time intervals. Though not shown in the annotation vocabulary figure, the CONSERT representation and reasoning engine defines a custom rdfs:Datatype called intervalListType which becomes the default for literals expressing values of the validity domain.
Lastly, for the accuracy annotation domain the vocabulary consists of decimal values in the interval $[0, 1]$. They are expressed using the xsd:decimal datatype.

One last noteworthy aspect is that StructuredAnnotation subclasses provide values for three important properties: hasJoinOp, hasMeetOp and hasContinuityFunction. The ranges of these properties are instances of SPIN functions[11]. The value of the first two properties represents the implementation of the $\oplus$ and $\otimes$ operators respectively, for the given StructuredAnnotation subclass. For the discussed annotation domains, the operators for timestamp and certainty default to existing SPARQL filter functions (i.e., afn:max, afn:min). For the validity domains a custom implementation is required.
The value of the hasContinuityFunction property is used during one of the steps performed by the CONSERT Engine when inserting a new *ContextAssertion* instance. It is further detailed in Section 7.2.
On startup, the CONSERT Engine creates an index of the basic and structured annotation types defined for the context model of

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

an application. For each StructuredAnnotation class it registers the functions that are the values of the three properties mentioned above. This way, at runtime it can automatically manipulate the annotations of *ContextAssertion* instances during the different steps in the execution cycle described in Section 7.2.

Constraint Vocabulary

In Definition 4.2 we noted that *ContextAssertions* can be subject to uniqueness and value constraint definitions. Figure 5

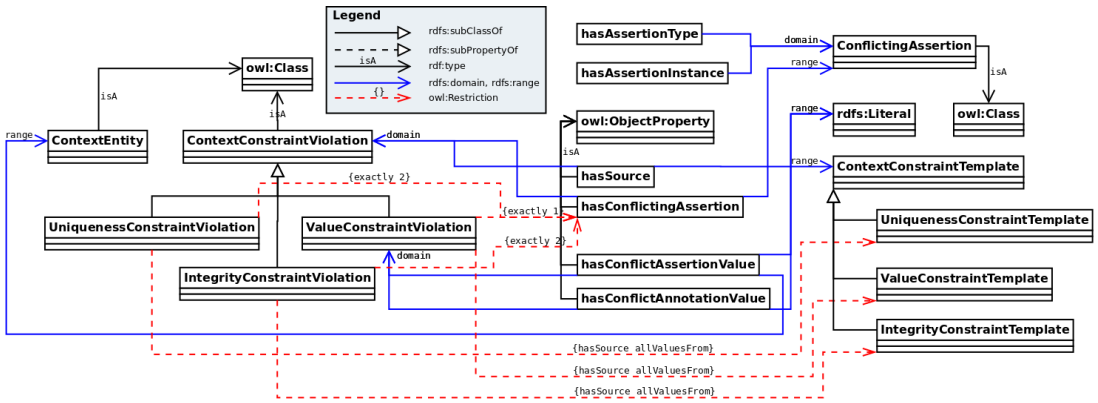

Figure 5: CONSERT ontology constraint vocabulary

shows the vocabulary used to define constraints. The UniquenessConstraintTemplate, ValueConstraintTemplate and IntegrityConstraintTemplate (and corresponding constraint violation classes) are the ones that realize the concept of context information constraints in the CONSERT ontology. The hasSource property is used on ContextConstraintViolation instances to refer to the ConstraintTemplate instance that triggered the conflict signalization. The conflicting *ContextAssertions* (one in the case of value constraints and two for uniqueness and integrity constraints) are given by the value of the hasConflictingAssertion property. The range of this property is an instance of the ConflictingAssertion class. Instances of this class have two properties. hasAssertionType specifies the ontology resource denoting the *ContextAssertion* on which the constraint is placed. hasAssertionInstance indicates identifier (the URI of the named graph that wraps its contents) of the conflicting *ContextAssertion* instance. The difference between uniqueness and integrity constraints is that the former are always specified for *ContextAssertions* of the same type (i.e. same value for the hasAssertionType property), while the latter capture unsatisfied dependencies between instances of two different *ContextAssertion* types. For value constraint violations, the CONSERT ontology also defines the possibility to express the annotation or assertion value that triggered the violation.
In Section 5.2 we detail how we attach constraints to a *ContextAssertion* class by using the introduced vocabulary.

5.2. Implementation using Semantic Web Technologies

The previous section introduced a realization of the CONSERT context model in terms of an ontology definition. We presented the vocabularies that express the core model elements, annotations and integrity constraints. We dedicate this section to explaining how we obtain an instantiation of a model expressed with the CONSERT ontology. We detail the logical storage of a model instance and the way in which we link a *ContextAssertion* with *ContextAnnotations*, *Context Constraints* and *Context Derivation Rules*.

Implementing ContextAssertions

We first discuss how we complete *ContextAssertions* expressed with the CONSERT ontology with the ability to become annotated statements. In doing so we make use of the concepts of quad stores and RDF named graphs [25]. RDF named graphs are a key concept of the Semantic Web. They allow a set of RDF statements (subject - predicate - object triples) called a graph to be grouped and associated with a URI.
We can use this facility to our advantage with the purpose of expressing *ContextAnnotations* of a *ContextAssertion*. We allow each individual RDF statement ($n = 1, 2$) or set of statements ($n \geqslant 3$) expressing a *ContextAssertion* to be wrapped within its own graph. Essentially, the graph URI becomes an identifier for the *ContextAssertion*. The *ContextAnnotations* are then expressed as RDF statements which have the graph URI (the identifier) as the subject.
The CONSERT representation and reasoning engine, which we detail in Section 7, uses such a named graph scheme to identify *ContextAssertion* instances and provide logical separation of *ContextEntity* class definition, *EntityDescription* instances and per-*ContextAssertion*-class *ContextAnnotations*.

Considering a context model expressed in terms of the CONSERT ontology and the chosen upper-ontology (e.g. SOUPA), the CONSERT engine maintains the following named graph schema structure:

- A named graph with the URI <baseURL/entityStore> contains all instances of ContextEntity subclasses as well as all *EntityDescriptions* expressed with subproperties of entityRelationDescription or entityDataDescription.
- For each class of *ContextAssertion* (subclasses of UnaryContextAssertion and NaryContextAssertion, subproperties of entityRelationAssertion and entityDataAssertion) we take the URI of the corresponding ontology resource definition and submit it to a transformation pattern that appends the word "Store" at the end. The resulting URIs identify named graphs that will hold all *ContextAnnotations* made for instances of their respective type of *ContextAssertions*.
- For a *ContextAssertion* instance that is about to be inserted, the following actions are taken:
  (i) issue a `CREATE GRAPH` request with a unique URI naming the graph that will wrap the new assertion and act as its identifier. The unique URI is obtained by appending the output of a UUID (universally unique identifier) generator to the ontology resource URI corresponding to the *ContextAssertion* instance class.
  (ii) use a `SPARQL INSERT` request to put the necessary RDF triples stating the *ContextAssertion* in the newly created named graph.
  iii) issue `SPARQL INSERT` requests for *ContextAnnotations* in the named graph corresponding to the name of our assertion as seen above. The graph URI created at step (i) serves as the subject of the RDF triples expressing the annotations.

Revisiting an earlier example from our smart conference scenario, the sensesBadgeWithIntensity *ContextAssertion*, the named graph structures that hold the instance of the assertion and the accompanying annotations are presented in the following figure.

```
GRAPH  <ex:senseBadge-UUID> {
 _:0 rdf:type ex:sensesBadgeWithIntensity.
 _:0 ex:sensorRole     ex:beacon1.
 _:0 ex:badgeRole      ex:userBadge.
 _:0 ex:intensityRole  ex:strong.

}
```

(a) A named graph identifying an instance of the sensesBadgeWithIntensity NaryContextAssertion

```
GRAPH <ex:sensesBadgeWithIntensityStore> {
 ex:senseBadge-UUID ctx:assertionType ctx:Sensed.
 ex:senseBadge-UUID ctx:hasTimestamp ctx:tsAnn.
 ctx:tsAnn ctx:hasValue
   "2013-12-03T12:00:05Z"^^xsd:datetime
 . . .
}
```

(b) The "store" named graph of the sensesBadgeWithIntensity *ContextAssertion* holding annotations

Figure 6: *ContextAssertion Store* and *identifier* Named Graphs

Implementing ContextAssertion Constraints

The implementation of *ContextAssertion* constraints and derivation rules (detailed in the next section) is achieved with help from the SPARQL Inferencing Notation (SPIN) proposal. SPIN currently has the status of a W3C Member Submission, but it has become the de facto industry standard to represent SPARQL rules and constraints on Semantic Web models[12]. Apart from providing an RDF serialization of SPARQL queries, SPIN offers a vocabulary that enables definition of inference and constraint rule templates and "attaching" instances of such templates to subclasses of owl:Class in an ontology model. In Section 5.1 we have seen how we take advantage of this fact within the CONSERT ontology vocabulary to define uniqueness and value constraints.

An instance of Uniqueness Constraint is given below (cf. Example 5.3) for the personLocatedAt *ContextAssertion* in our smart conference scenario. It expresses the fact that a person cannot be deemed as finding herself in two places at the same time (overlapping validity intervals) with high certainty in both affirmations. Notice also that one advantage of using SPARQL as a constraint definition language allows us to compose expressive constraint statements, as was also our goal. In this example, not only can we condition the triggering of a violation based on values of the annotations of a *ContextAssertion*, but the domain knowledge check includes a call to a SPARQL 1.1 Property Path (spc:spatiallySubsumedBy+) which states that physical spaces that lie in a spatial subsumption relation are excluded from the set of conflicting ones (e.g. if a user is in a session room, it is ok to have another *ContextAssertion* that says the user is also in the conference building).

[12] http://spinrdf.org/

Example 5.3 (personLocatedAt uniqueness constraint)**.**

```
CONSTRUCT {
  _:b0 a ctx:UniquenessConstraintViolation .
  _:b0 ctx:onContextAssertion person:locatedAt .
  _:b0 ctx:hasConflictingAssertion ?g1 .
  _:b0 ctx:hasConflictingAssertion ?g2 .
}
WHERE {
  GRAPH ?g1 {
    ?this person:locatedAt ?Loc1 .
  } .
  GRAPH ?g2 {
    ?this person:locatedAt ?Loc2 .
  } .
  GRAPH <http://pervasive.semanticweb.org/ont/2004/06/person/locatedAtStore> {
    ?g1 ctx:hasValidity ?valAnn1 . ?valAnn1 ctx:hasValue ?validity1 .
    ?g1 ctx:hasCertainty ?certAnn1 . ?certAnn1 ctx:hasValue ?cert1 .
    ?g2 ctx:hasValidity ?valAnn2 . ?valAnn2 ctx:hasValue ?validity2 .
    ?g2 ctx:hasCertainty ?certAnn2 . ?certAnn2 ctx:hasValue ?cert2 .
  } .
  FILTER (
    NOT EXISTS {?Loc1 (spc:spatiallySubsumedBy)+ ?Loc2 .} &&
    (?Loc1 != ?Loc2) && (?cert1 >= 0.75) && (?cert2 >= 0.75) &&
    cfn:validityIntervalsOverlap(?validity1, ?validity2)).
}
```

The SPIN specification allows such queries to be attached to an OWL class definition using the spin:constraint property. Since the constraints are conceptually defined on *ContextAssertions*, for subclasses of UnaryContextAssertion and NaryContextAssertion the SPIN Context Constraint Template instance is attached directly to the corresponding ontology class. For binary *ContextAssertions* (subproperties of entityRelationAssertion and entityDataAssertion) however, since owl:ObjectProperties are not a type of owl:Class, the constraint definition is attached to one of the two *ContextEntity* class definitions that play a role in the binary assertion (i.e., either the domain or the range of the ontology property). The CONSERT engine is automatically configured to look for both kinds of attachment schemes. In our example, the constraint for the personLocatedAt *ContextAssertion* is attached to the Person *ContextEntity* class which represents the domain of the property.

Implementing ContextAssertion Derivation Rules

To map the derivation rules which drive the reasoning process we again use SPIN. Recall from Section 4.3 that the head of a derivation rule ρ is a *ContextAssertion* $F_{head}(x_1, \ldots, x_k) : \{\lambda_1, \ldots, \lambda_l\}$. We use the spin:deriveassertion subproperty of spin:rule[13] to attach the corresponding body of ρ, expressed in SPARQL syntax, to the *ContextAssertion* of a derived type. As was the case for context constraints, the derivation rule is attached directly to subclasses of UnaryContextAssertion and NaryContextAssertion and to the domain or range *ContextEntity* class in the case of a binary *ContextAssertion*. In Section 7 we explain how the CONSERT engine creates auxiliary datastructures that build a derivation rule dictionary mapping every *ContextAssertion* to the list of *Context Derivation Rules* in the body of which it appears. The engine can then execute those rule bodies every time the value of a mapped *ContextAssertion* changes.

We now show how the elements of *ConditionExpr* are expressed in terms of SPARQL syntax:

- $AssertionExpr$: a *ContextAssertion* and its *ContextAnnotation* instances are expressed using RDF statements wrapped in named graphs as explained in Section 5.2. These form SPARQL basic graph patterns (BGP).
- $AggExpr$: are expressed using SPARQL aggregates[14]
- $TermExpr$: *EntityDescriptions* are expressed as RDF triples that reside within the entityStore named graph as detailed in Section 5.2. Boolean operations, logical connectives and functions on terms are implemented using the equivalent SPARQL syntax and are contained within SPARQL FILTER expressions.
- $AnnExpr$: the annotation assignment functions are user-defined. They are implemented based on the Jena API as custom code that runs during query evaluation in the software engine that we detail Section 7. The value they compute is bound to the corresponding λ_j variable in the rule head ($annotations(head(\rho))$) using a SPARQL BIND statement.

[13] http://spinrdf.org/spin.html#spin-rules
[14] http://www.w3.org/TR/sparql11-query/#aggregates

Finally, let us consider the existentially and universally constrained quantifications. For the existential case support is already provided in SPARQL by the EXISTS filter expression (see Figure 7). For the universal case the intuition behind the SPARQL query shown in Figure 7 is the following: consider a substitution $\sigma = \{y_1/t_1, \ldots, y_r/t_r\}$ which binds each variable $y_i \in Y_\rho$ to a *ContextEntity* or literal. Let us then denote by $\Sigma_{F_c \downarrow Y_\rho}$ and $\Sigma_{DerivExpr \downarrow Y_\rho}$ the sets of all substitutions σ binding variables in Y_ρ for which the constraining assertion F_c and the assertions in *ConditionExpr* are true respectively. The interpretation of the universal constrained quantification rule then implies that $\Sigma_{F_c \downarrow Y_\rho} \subseteq \Sigma_{DerivExpr \downarrow Y_\rho} \Leftrightarrow \Sigma_{F_c \downarrow Y_\rho} \setminus \Sigma_{DerivExpr \downarrow Y_\rho} = \emptyset$. The SPARQL MINUS[15] filter expression used in Figure 7 provides this exact semantics.

```
CREATE GRAPH <gURI>;
INSERT{
  GRAPH <gURI> {new assertion}
  GRAPH <newAssertionStore> {annotations}
}
WHERE {
 { constraining assertion } .
 FILTER (
  EXISTS { ConditionExpr }
 )
}
```

```
CREATE GRAPH <gURI>;
INSERT{ assertion and its annotations }
WHERE {
 { SELECT (COUNT(*) AS ?count)
    WHERE {
     {constraining assertion}
     MINUS
     {ConditionExpr}
    }
 } . FILTER (?count = 0)
}
```

Figure 7: SPARQL expressions for existentially (left) and universally (right) constrained quantifications

6. AmI Demonstrative Scenario: Modeling with CONSERT

Before we go on to present the CONSERT representation and reasoning engine that fully leverages the context model we presented in this article, we illustrate how the model can be used in the Smart Conference Scenario presented in Section 3.

6.1. Context Situation and Entity Identification

The first step in the effort to create the smart conference scenario model is accounting for situations that the application domain naturally exposes and which are of particular interest to the application designer. In the scenario, emphasis is placed on enhancing participant interaction (e.g. recommendations for participation in ad-hoc discussions) and automation of tasks related to setting up and holding a presentation (e.g. session management, automatic presentation configuration on speaker turn, default availability restriction actions for members of the audience). Therefore, the situations that are considered are: (i) establishing when a presentation starts, (ii) detecting if a user is attending a presentation, (iii) detecting if a user is giving a presentation, (iv) detecting if one or more users are in an ad-hoc discussion.

Secondly, considering the application capabilities and existing devices in the smart space, the relevant information that can be collected during runtime is:

- *static information*: information about user contact details, affiliation and research interests
- *sensed information*: strength of the signal perceived by a beacon for a user smartphone, noise level perceived by a microphone next to a discussion area
- *profiled information*: position of a location beacon, conference profile of a user, digital document (PDF, PowerPoint) containing a user's presentation
- *derived information*: the location of a user

Notice that we grouped the data by the way in which each particular piece of information is acquired (whether static – the information designates a particular truth – sensed, explicitly profiled or derived – deduced following an inference process that uses existing context information). Remember from explanations in Section 4.1 that the acquisition type is one of the rule of thumb criteria used to distinguish between pieces of context information that are to be modeled as *ContextAssertions* or *EntityDescriptions*.

Finally, in a third step, we identify the entities involved in the situations listed above. The scenario revolves around *Users*, *Presentations*, *Sessions* and *Devices*. Other entities are specializations of the latter or in close relation to them. For instance, different types of physical locations are considered such as a *SessionRoom*, a *DiscussionArea* or the *SpeakerArea* within a session room. *ProximityBeacons* and *Microphones* are types of devices present in the physical environment. Users are given a *ConferenceProfile* that holds information about contact details, affiliation and research interests.

[15] http://www.w3.org/TR/sparql11-query/#neg-minus

6.2. *Context Modeling*

After the three steps presented above, we have the required elements to provide a scenario model using the CONSERT ontology.

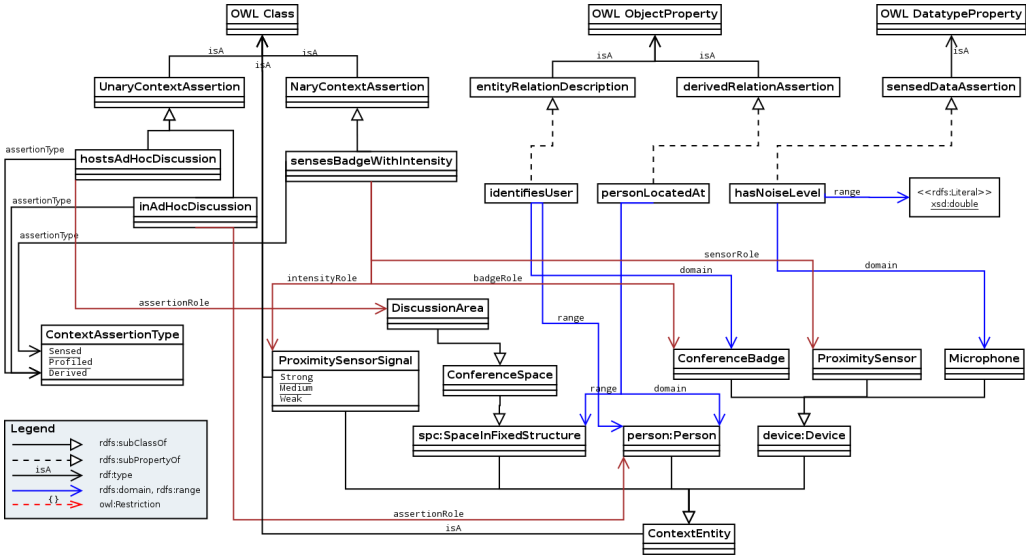

Figure 8: Excerpt from the smart conference context model based on the CONSERT ontology

Domain Model for Smart Conference Scenario

The diagram in Figure 8 shows an excerpt of the model. We begin by observing the information instances modeled as UnaryContextAssertions and NaryContextAssertions, showcasing the ability to consider and represent *ContextAssertions* of arbitrary arity. The unary *ContextAssertion* inAdHocDiscussion describes a situation applying to users (instances of the *ContextEntity* person:Person) that are currently involved in a conversation at an ad-hoc discussion area.

The sensesBadgeWithIntensity *ContextAssertion* is an example of using arbitrary arity statements to concisely capture relevant context information (i.e., the intensity with which a conference badge is sensed by a proximity beacon). Modeling sensesBadgeWithIntensity as a three parameter statement eliminates the need to artificially introduce binary *ContextAssertions* (which are not reused elsewhere in the model) the combination of which expresses the same meaning.

The sensesBadgeWithIntensity *ContextAssertion* has three assertionRole subproperties (sensorRole, badgeRole and intensityRole - marked in red in the figure) which relate this class to *ContextEntities* that enact the roles (ProximitySensor, ConferenceBadge and ProximitySensorSignal).

Apart from the unary and n-ary *ContextAssertions* the diagram shows an example of a derivedRelationAssertion property (personLocatedAt), a *sensedDataAssertion* property (hasNoiseLevel) which relates a *ContextEntity* (Microphone) to a literal value of type xsd:double and an instance of a *entityRelationDescription* (identifiesUser) which expresses the static information that a ConferenceBadge is indicative of a particular person:Person.

Constraints in the Smart Conference Scenario

After the initial context domain model is established, the application designer can enrich its runtime behavior by setting up context information integrity constraints and derivation rules. We complement here the example of ContextUniquenessConstraint shown in Section 5.2. Let us consider an instance of *ContextValueConstraint* by studying the ValueConstraintTemplate in Example 6.1.

Example 6.1 (Value Constraint for currentPresentationForSession in SPARQL)**.**

```
CONSTRUCT {
  _:b0 a ctx:ValueConstraintViolation .
  _:b0 ctx:onContextAssertion conf:currentPresentationForSession .
  _:b0 ctx:hasConflictingAssertion ?g1 .
  _:b0 ctx:hasConflictAnnotationValue ?validity1 .
}
WHERE {
 GRAPH ?g1 {
   ?this conf:currentPresentationForSession ?Session .
 } .
```

```
 GRAPH <conf:currentPresentationForSessionStore> {
   ?g1 ctx:hasValidity ?valAnn . ?valAnn ctx:hasValue ?validity1 .
 } .
 FILTER cfn:validityIntervalExceeds(?validity1, 2400) .
}
```

It expresses the restriction that a presentation cannot be marked as the current one for the session it is part of for longer than 40 minutes. A violation of this constraint can warn the session chair that the active presentation time is exceeding a threshold duration set by the Program Chairs. Though the internal logic of this restriction is much simpler than the one presented in Section 5.2, notice again the ability to relate to annotation values of an application domain statement in an attempt to maintain knowledge base integrity.

Derivation Rules in the Smart Conference Scenario

From the identification of *ContextAssertions* that are derived from other information, the designer defines the proper *Context Derivation Rules* as defined in CONSERT. Let's consider the following two examples: personLocatedAt *Context Derivation Rule* and hostsAdHocDiscussion *Context Derivation Rule*. The body of the rules contain examples of all the possible expressions introduced in Section 4.3. They are formulated using the syntax presented in this section, since it is more intelligible and less verbose than the SPARQL form[16].

```
personLocatedAt(P, Loc):{λ_src, λ_t, λ_valid, λ_acc}:
  isA(PS, proximityBeacon) ∧ deviceLocatedAt(PS, Loc) ∧
  isA(B, badge) ∧ identifiesUser(B, P) ∧
  sensesBadgeWithIntensity(PS, B, strong):{λ_senseTs, λ_senseAcc} ∧
  datetimeDelay(now(), -2, λ_earlier) ∧  λ_senseTs ⩾ λ_earlier ∧ λ_senseAcc ⩾ 0.8  ∧
  assignCert(λ_acc, λ_senseAcc) ∧
  assignSrc(λ_src, currentAgent) ∧
  assignTimestamp(λ_t, now()) ∧
  datetimeDelay(now(), +2, λ_later) ∧
  assignValid(λ_valid, makeValidityInterval(now(), λ_later))
```

Figure 9: Person Located At Context Derivation Rule

The first rule, shown in Figure 9, presents the way in which the value of the personLocatedAt *ContextAssertion* is derived. The rule firstly determines the location of a *ProximityBeacon* device. It then looks at the sensesBadgeWithIntensity *ContextAssertion* to determine the strength with which the beacon senses a given ConferenceBadge. If the badge identifies the person whose location we are trying to determine (by means of the identifiesUser *EntityDescription*) then the domain knowledge conditions of the rule are satisfied. The rule body contains additional requirements posed on the annotation information of the comprised *ContextAssertions*. Specifically, the sensesBadgeWithIntensity *ContextAssertion* must have been recently updated (less than 2 seconds ago) and the certainty of the affirmation must exceed 80%. If all these conditions are met the rule will fire. Notice how in the case of this rule, the *ContextAnnotation* values of the derived *ContextAssertion* are explicitly manipulated using annotation assignment functions (the assign* calls). As explained formally in Definition 4.12, these functions can determine derived *ContextAnnotations* by explicitly setting the corresponding value or using the annotation domain specific $\oplus$ and $\otimes$ operators (we see an example of this in the next rule). The currentAgent literal used to express the source of the derived *ContextAssertion* is a URI that identifies the service currently running the inference rule.

The second rule we consider is the one presented in Figure 10. The rule helps to derive the *ContextAssertion* that one of the dedicated discussion areas of our conference venue is actually hosting a discussion. The rule states that if more than two people are detected with a high enough average confidence as being located in the proximity of the discussion area for a duration of more than 5 minutes and the noise level picked up by the microphone in the vicinity of the discussion area at a time during the past 5 minutes is higher than 60 dB (the normal loudness of a human conversation) then the rule will fire. For this *Derivation Rule* notice how the annotation expression function for the certainty *ContextAnnotation* is making use of the specific $\otimes_{cert}$ operator to combine certainty annotation information from the personLocatedAt and hasNoiseLevel *ContextAssertions*.

6.3. Expressiveness Analysis

This section has presented an exemplification of the way in which our context model can be used to conceive the context domain of an application scenario. Along the way we have shown the advantages and support for expressiveness that the model

[16]Note however that by using the transformation mappings shown in Section 5.2, the rules depicted in their formal syntax can be converted to their corresponding SPARQL representation.

```
hostsAdhocDiscussion(DA):{λ_src, λ_t, λ_valid, λ_acc}:
  isA(Mic, microphone) ∧ deviceLocatedAt(Mic, DA) ∧
  datetimeDelay(now(), -300, λ_earlier) ∧ makeInterval(λ_earlier,now(),λ_interv) ∧
  hasNoiseLevel(Mic, NL):{λ_micTs, λ_accMic} ∧
  ∧ NL⩾60 ∧ λ_accMic ⩾ 0.75 ∧
  λ_micTs ⩾ λ_earlier ∧ λ_time < now() ∧
  aggregate([count(P),avg(λ_accP)],
    personLocatedAt(P,DA):{λ_validP, λ_accP}
    ∧includes(λ_validP, λ_interv),[Ct,avgAccP]
  ) ∧
  Ct⩾2 ∧ avgAccP ⩾0.75 ∧
  assignCert(λ_acc, avgAccP ⊗_cert λ_accMic) ∧
  assignSrc(λ_src, currentAgent) ∧
  assignTimestamp(λ_t, now()) ∧
  datetimeDelay(now(), +60, λ_later) ∧
  assignValid(λ_valid, makeValidityInterval(now(), λ_later))
```

Figure 10: Ad-hoc Discussion Context Derivation Rule

offers in different situations.

Most notably, we have seen how integrity constraints can be specified taking into account both domain and annotation information and how the CONSERT constraint vocabulary allows to accurately express the integrity conflict.

With regard to the *Derivation Rules* one can easily observe the capability of the rule syntax to place conditions over both domain and annotation information. This allows reasoning over sequences of events and situations that have a duration in time (as is the second rule example). Moreover, the rule in Figure 10 shows an example of an aggregation expression that allows our model to pose conditions on the number of *ContextAssertions* that respect a requirement, or to use the average value of a numeric *ContextAnnotation* information pertaining to the set of *ContextAssertions* that satisfy the aggregation condition (as again is the case for the rule in the second example). All of these elements are indicators of the expressiveness that our proposed *Context Derivation Rules* can support.

7. CONSERT Reasoning Engine

We now explore the design and implementation of a context representation and reasoning system (the CONSERT engine) whose functionality builds on the concepts and their implementation as described in the previous sections. We present the engine architecture and detail its execution cycle in what follows.

7.1. Architecture

Figure 11 presents an architectural overview of the CONSERT engine. Its most important building blocks and internal data structures can be observed on the right side of the figure.

The engine defines three indexes (ContextAssertionIndex, ContextAnnotationIndex, ContextConstraintIndex) which create an internal representation of the context model built using the ontology modules described in Section 5. Using the Apache Jena and SPIN APIs, the indexes create wrappers over the modeled constructs, allowing easy access to required information at runtime (e.g. URI of the named graph holding the annotation statements for instances of a given *ContextAssertion* type, records of the hasJoinOp, hasMeetOp and hasContinuityFunction properties for each *ContextAnnotation* type).

The Derivation Rule Dictionary provides a mapping from every *ContextAssertion* to the list of *Derivation Rules* in the body of which it appears. The *Derivation Rules* are stored as SPIN defined wrappers over the SPARQL CONSTRUCT query implementing the rule. This dictionary is used during inference checks to determine if the update request for a given *ContextAssertion* can trigger the execution of a deduction process.

To implement the actual context knowledge base, we have chosen Apache Jena's TDB[17] as the supporting quad storage system. The knowledge base holds the *ContextAssertions* and their accompanying *ContextAnnotations* under the named graph form presented in Section 5. One advantage of TDB is that it offers an in-memory store variant with support for transactions, a feature we currently use to keep consistent views of the existing context information when handling the update and inference requests.

The CONSERT engine has three separate thread pools together with corresponding task queues: one for handling *ContextAssertion* update requests, one for processing inferences and one for answering queries.

Furthermore, the engine is designed to work as a service component, meaning that it exposes and requires interfaces that enable

[17] http://jena.apache.org/documentation/tdb/

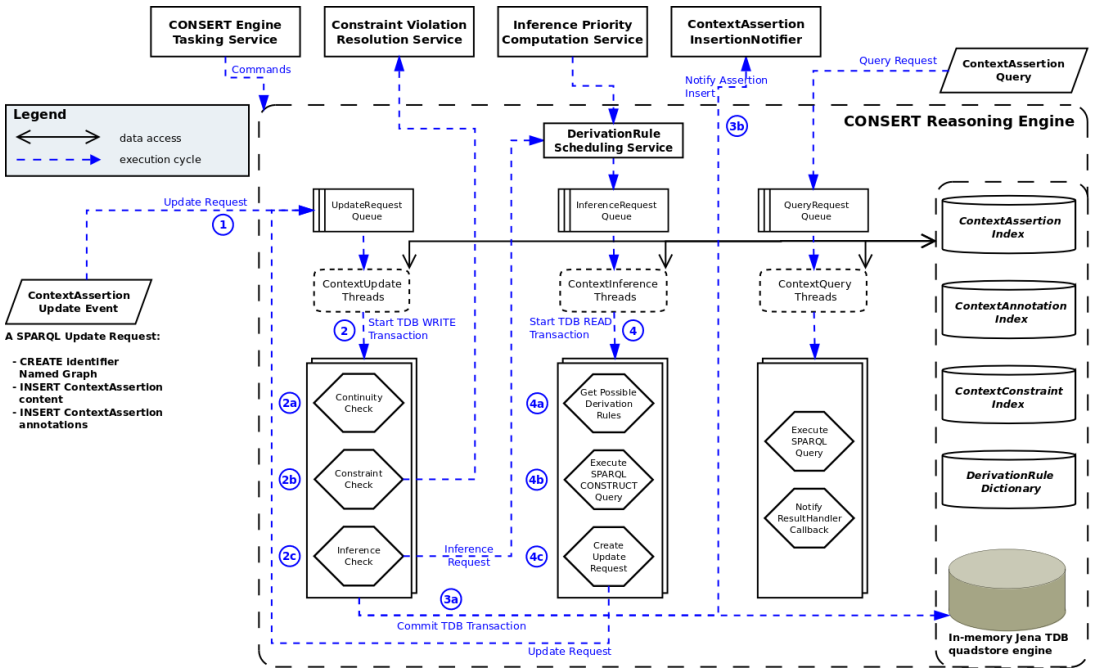

Figure 11: The CONSERT engine architecture and main activity cycle

it to work with additional application-specific services. In Figure 11 we observe a list of four such services depicted above the CONSERT engine container delimiter.

The CONSERT engine interacts with the *Constraint Violation Resolution Service* when either a uniqueness or an integrity constraint violation is detected during a *ContextAssertion* update. The service is supplied by the application developer and it effectively implements a policy for deciding which of the two conflicting assertion instances must be kept. The statements generated using the constraint vocabulary of the CONSERT ontology described in Section 5.1 provide the service with all the information required to retrieve both the content and the annotations of conflicting *ContextAssertion* instances, so as to make an informed decision. The CONSERT engine supplies two default implementations of the service which may be used to discriminate based on the timestamp (PreferNewest) or certainty(PreferAccurate) annotations.

The *Inference Priority Computation Service* is used by the *Derivation Rule* scheduler of the CONSERT engine when new inference requests get enqueued. The engine provides a default first-come first-served implementation of this service, but application developers may provide their own, which can assign, for example, a priority for each type of *Derivation Rule*. The priority value can be based on the inference usage and success statistics collected periodically by the CONSERT Engine. A full detail of the statistics collection process, however, falls outside the scope of this paper.

The *ContextAssertion Insertion Notifier* is a service notified by the CONSERT engine whenever a *ContextAssertion* update is successfully committed to the TDB store. When integrated into a larger context management platform, the *InsertionNotifier* can, for example, inform a query handler of a new update. The query handler could in turn submit the query associated with external client subscriptions triggered by the new insertion back to the CONSERT engine.

Finally, as the engine is meant to run within a broader context management solution, the *CONSERT Engine Tasking Service* is the means by which such a solution can control the different aspects of the CONSERT engine runtime execution. The service allows an external management platform *(i)* to request that a *Derivation Rule* be enabled or disabled, *(ii)* to trigger an ontology reasoning process or *(iii)* to clear the in-memory storage of *ContextAssertion* instances that exceed a certain time-to-live threshold. Options *(i)* and *(ii)* regulate the inference capabilities of the CONSERT engine. In particular, the *Derivation Rule* enable/disable switch controls the dynamic event processing side of the inference process. The ontology reasoning trigger, on the other hand, is targeted towards inference over the quasi-static background knowledge consisting of *ContextEntity* and *EntityDescription* definitions. Remember from Section 5.2 that *ContextEntities* and *EntityDescriptions* are logically contained within their own named graph, the entityStore. However, the entityStore is not totally static, in that a *ContextAssertion* update can insert new *ContextEntity* or *EntityDescription* instances within it. A simple example of such a case is one in which a new sensor starts sending assertion updates. The first time it submits an update, it can also supply statements that describe it as a sensor (e.g. its MAC address, its static location in a room). Since the information in the entityStore is used during *Derivation Rule* processing, ontology inferences (e.g. subclass, subproperty, inverse property) are important to ensure that all possible statements for *ContextEntities* and *EntityDescriptions* are covered. However, performing an ontology reasoning effort, whenever the *entityStore*

is updated incurs a significant overhead (as shown in Section 8.2). Therefore, the *CONSERT Engine Tasking Service* externalizes the logic of requesting ontological reasoning to the broader context management platform. Such a platform can define the timing for periodic invocations of the ontology reasoning process, so that the period is correlated with the perceived update frequency of each type of *ContextAssertion*.

Option *(iii)* (in-memory storage clearing) is a way to free up memory from the statements of *ContextAssertions* which have outlived their possible usage during runtime inference processing. The external management solution can for instance define a time-to-live threshold for each *ContextAssertion* type and request that assertion instances with a timestamp annotation that is older than the specified threshold be moved from memory to a persistent storage for possible offline processing.

7.2. Execution Cycle

The CONSERT engine is initialized based on a set of OWL files that define the context model of an application. To make things easier for a developer, he or she can specify a file for each module of the CONSERT ontology (i.e. core, annotation, constraint), as well as for the custom RDF Datatypes (e.g. intervalListType), SPARQL filter functions (e.g. makeValidityInterval, validityIntervalsInclude) and SPIN-encoded *Derivation Rules* required for reasoning. The engine uses parses the files to build the auxiliary data structures we discussed above. Once they are initialized the engine can start operating.

In Figure 11 we observe that the first step of the execution cycle is an arriving *ContextAssertion* update request. It is a `SPARQL UPDATE` query containing three parts: a `CREATE GRAPH` statement that leads to the creation of a new named graph which will identify the new *ContextAssertion* and two `INSERT` statements, one that puts the contents of the new *ContextAssertion* in the named graph created previously and one that inserts the *ContextAnnotations* of the assertion in the corresponding store graph (as described in Section 5.2). The update request is put into the waiting queue of the insertion handling thread pool. Once it is picked up, the handler thread proceeds to Step 2 of the execution process and creates a `TDB WRITE` transaction in which it will perform a sequence of three verifications:

Step 2a) The first one is called the *continuity check*, where the system checks if the content of the new *ContextAssertion* matches any of the already existing ones. If a content-based match is found, the check proceeds by looking at each *StructuredAnnotations* attached to the two continuity-merge candidate assertions. As hinted towards in earlier sections, the procedure accesses the *ContextAnnotationIndex* to retrieve the value of hasContinuityFunction property for the given *StructuredAnnotation*. The continuity function examines if the two *ContextAssertion* instances can be merged from the viewpoint of the annotation. For example, in the case of the certainty annotation, only assertions for which their certainty annotations are close to one another (within 0.1) may be allowed to merge. That is, a situation described by a *ContextAssertion* with a high certainty value is different from one where the same situation is asserted with a much smaller degree of confidence.

If after checking every continuity function, the two *ContextAssertion* instances are allowed to merge, the engine will access the corresponding $\oplus$ operator of each *StructuredAnnotation* to update the annotation values of the existing *ContextAssertion*. (an action which represents a use case of combining annotation information for statements that have the same content, as explained in Section 4.1). The values for the *BasicAnnotations* are simply taken from the newly inserted *ContextAssertion* and attached to the resulting merger.

Step 2b) The class of the *ContextAssertion* to be inserted is checked against the ConstraintIndex to determine if it has any attached integrity, uniqueness or value constraints. If found, the SPIN query wrappers are used to execute the constraints. While assertions violating value constraints are simply rejected, for any uniqueness or integrity violations, the *Constraint Violation Resolution Service* is accessed as explained earlier above.

Step 2c) The final check is made against the Derivation Rule Dictionary which the system uses to determine if the class of the updated *ContextAssertion* appears in any *Context Derivation Rules*. On success, the insertion handler thread will enqueue an inference request triggered by the current *ContextAssertion*.

After all verifications are completed, Step 3a of the execution cycle commits the active transaction to the in-memory instance of the TDB quadstore, while Step 3b sends a notification to the *ContextAssertion Insertion Notifier* such that it may inform any registered listeners.

When an inference request is received, the corresponding handler threads starts a `TDB READ` transaction (Step 4) in which it executes three actions as follows:

Step 4a) The handler thread uses the Derivation Rule Dictionary to retrieve the wrapper list of all rules in which the *ContextAssertion* in the inference request plays a role.

Step 4b) For each *Derivation Rule* in turn it executes the wrapped `SPARQL CONSTRUCT` query. If the rule could be successfully applied the thread proceeds to the last step in the inference process.

Step 4c) The thread takes the statements constructed by the inference query and transforms them into the three-fold `SPARQL UPDATE` statements reported earlier for the case of a *ContextAssertion* update. The newly inferred *ContextAssertion* is thus enqueued in the insertion waiting queue, which completes the deduction functionality cycle.

The design specifications listed above present the sequence of steps that the CONSERT representation and reasoning engine undertakes while handling updates of *ContextAssertions* that it may receive from external applications or sensing services. These steps help to build and maintain the contextual situation knowledge about current states and activities in the environment that other services can then exploit through querying. The CONSERT engine handles queries in an asynchronous manner and requires that clients submit their queries together with a *ResultHandler* callback (cf. Figure 11). What is important to notice is that the steps are built around the idea of making use of all the supporting benefits introduced by the CONSERT ontology discussed in the previous section: from using the structured operators of *ContextAnnotations* in the continuity check, to applying *Context Constraints* and collecting the possible constraint violation according to the properties of the ConstraintViolation class defined in the CONSERT ontology and down to using the SPIN API to index and execute *Context Derivation Rules* that infer derived *ContextAssertions*.

8. CONSERT Reasoning Engine: Performance Results

After having implemented the CONSERT representation and reasoning engine using the system architecture described earlier, we performed a series of tests in order to assert the technical validity of the approach and establish the limits of the current design choices. In the following we are going to detail the testing configuration that we set up and then provide an analysis and comments about possible improvements based on the obtained results.

8.1. Testing setup

In performing the test, our interest was to verify that the proposed representation and reasoning engine could be successfully employed as part of a real-time context provisioning mechanism. In other words, we wanted to perform a load test on our proposed system.

We started by writing a simple scenario generating program that is able to create definitions of *ContextEntities*, *ContextAssertions* and *Context Derivation Rules*. The automated generation is controlled by the following parameters:

- number of *ContextEntity* classes and number of instances of each class
- total number of *ContextAssertion* classes of each arity (unary, binary or n-ary)
- number of derived *ContextAssertion* classes out of the total number of class types pertaining to each arity
- number of instances that would be generated during the test run for each type of *ContextAssertion*

The automatically generated context model is then used by a test script to generate a sequence of *ContextAssertion* update requests. The parameters which control the runtime dynamics are: the temporal validity duration of a *ContextAssertion* instance (in *ms*) and the instance pushrate (number of *ContextAssertion* instances generated per validity interval) for each arity class. The context model generation parameters influence the size and variety of the simulated domain, and they were used to study the memory footprint of the system, specifically in order to determine if an increasing number of named graphs (as would be expected by a larger number of *ContextAssertion* classes and instances) poses a concern. The parameters controlling the test script influence the responsiveness of the system and we wanted to determine under what kind of load it would still be able to perform at a level acceptable for real-time usage (e.g. a short enough time span passes from the moment a *ContextAssertion* that triggers an inference is created to the point where the inferred assertion becomes visible).

Note that for this automated test, we focused on checking the performance of the *event processing* capabilities of the CONSERT engine and did not trigger any ontology reasoning in between *ContextAssertion* updates. We discuss the influence that even simple RDFS reasoning can have on update processing in a second experiment. For the purpose of the automated test the *Context Derivation Rules* were kept at a low complexity in order to allow automatic generation and to ensure that a sufficient amount of rules fire during runtime.

The output of the scenario generation program is an OWL file that contains the ontology definitions of the simulated context domain and the accompanying context derivation rules. The file is used to initialize the CONSERT engine as described in

Section 7. After initialization, the test script is used to perform actual generation of *ContextAssertion* instances and collect runtime measures. We detail the type of collected information and its analysis in the following section.

8.2. Results and Analysis

During the run of a test we collected information regarding the trace left by each created *ContextAssertion* within the system. We looked at insertion delay time (Step 1 in Figure 11: time spent from entering the update request queue until start of processing), insertion processing time (Steps 2a – 2c: how long it takes to apply all the verifications detailed in Section 7), inference delay (Step 4) and inference processing time (in the case the newly created *ContextAssertion* triggers an inference - Steps 4a – 4c) and overall deduction time (the amount it takes from the moment an inference triggering *ContextAssertion* enters the system until the deduced one is also observed). For each of these parameters we also computed minimum, average and maximum values. To compute memory consumption we used the jProfiler[18] Java profiling framework to observe live memory usage. At the end of a test run, after performing garbage collection, we specifically recorded the number of instances and total size of several key data structures that would be directly influenced by the number of created context assertions. We also looked at the total memory size of the Java heap space, after garbage collection.

Runtime Processing Times

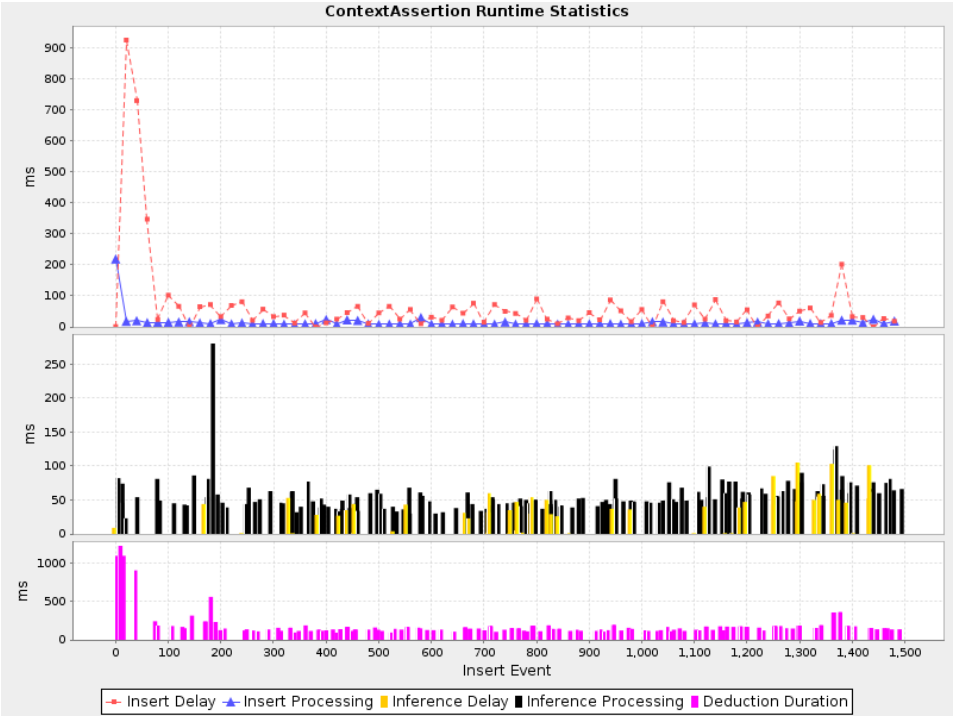

Figure 12: Runtime results for test run with 10 *ContextAssertion* classes of each arity type and 50 instances for each class. The validity duration of a *ContextAssertion* instance is set at 1000 ms and the pushrate has a value of 20 instances generated per validity interval (i.e., 20 events per second).

In Figure 12 we observe the runtime history of a test where the pushrate has been set to 20 *ContextAssertion* requests per second. The upper chart shows the insertion delay and insertion processing metrics, the middle one draws overlaid bars showing the inference delay and inference processing values, whereas the lower chart combines the two and shows a bar chart of the deduction duration information for those *ContextAssertion* instances that triggered the inference of a new one.

Table 1: Minimum, average and maximum values for the collected runtime parameters of the 20 events per second test run (in ms)

Insertion Delay			Insertion Processing			Inference Delay			Inference Processing			Deduction Duration		
min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max
0	71	952	7	15	272	0	14	141	23	59	255	85	206	1258

The spike at the beginning of the insertion delay plot is attributed to the “warm-up” of the thread pool handling the update, requests as well as, more decisively, that of the in-memory TDB quadstore engine where the inserted *ContextAssertions* are stored. The average value of the insertion delay metric is of only 71 ms, as can be seen in Table 1. The variation in the delay

[18] http://www.ej-technologies.com/products/jprofiler/overview.html

time is in sync with the peeks of the inference plot and it is due to the addition of the deduced *ContextAssertions* to the list of ones that have to be inserted as part of the next "batch" of events. Given the low complexity of the employed *Context Derivation Rules*, Table 1 shows that the inference duration time is fairly low and since the number of derived *ContextAssertions* is much lower than that of the created ones, there is no build up in the request queue of the inference thread pool and so the inference processing time dominates the inference delay time (the time that an inference request spends in the InferenceRequest Queue before actual execution).

For this test configuration the average deduction duration is set around 206 ms. Considering that one second can be seen as a decent response time for a realtime recognition of a situation, it means that the current load of 20 *ContextAssertion* insertion events per second can actually leave room to spare (in case the derivation rules are more complex and require more time for evaluation).

Deduction Time Analysis

In general, the deduction duration plot helps us to determine two important aspects concerning the runtime dynamics of a system. First, depending on the value that is considered acceptable for the average deduction duration, we can set the corresponding maximum number of *ContextAssertion* update requests that can be handled per second. This implicitly translates into an upper threshold on the frequency with which different physical, logical or virtual sensors would provide updates to the data they perceive. Keeping a log of the deduction duration data could help a future version of the system determine how to automatically negotiate such sensor update rates. Second, the value of the deduction duration is also an indicator of the minimum temporal validity that a detected situation must have in order to be usefully utilized. That is, if the actual situation is shorter than the time it takes for the system to have it recognized and available for query or decision making, the effort to infer it will not have brought any added value.

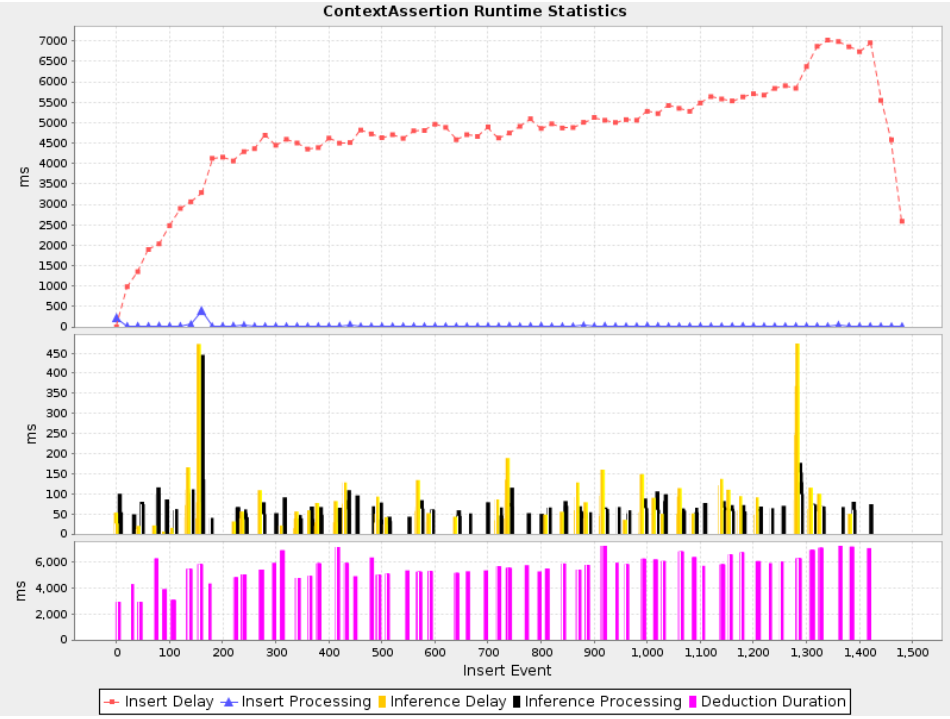

Figure 13: Same configuration as for the test case presented in Figure 12, but with a pushrate set at 60 instances generated per validity interval (i.e., 60 requests per second).

We can see the effect of increasing the pushrate load in Figure 13 and Table 2. For both the insertion and inference of *ContextAssertions* we observe a dramatic increase and dominance of the delay component (the time spent in the request queues). Still, the insertion and inference processing metrics remain almost the same.

Table 2: Minimum, average and maximum values for the collected runtime parameters of the 60 events per second test run (in ms)

Insertion Delay			Insertion Processing			Inference Delay			Inference Processing			Deduction Duration		
min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max
1	4478	7035	8	17	347	0	46	306	31	65	268	2897	5466	7842

This line of experimentation has led us to observe that the true limitation factor in the current instantiation of the CONSERT

engine is given by Jena TDBs ability to handle multiple concurrent READ and WRITE transactions. Since every new *ContextAssertion* update request requires the creation of a WRITE transaction, a high amount of update events per second creates a contention with regard to synchronization after all the verification steps of the insertion processing cycle have been carried out. Under current test configurations we determined that the push rate value ensuring a less than one second deduction duration resides somewhere along the 30 update requests per seconds mark. This still represents a reasonable value for potential real life scenarios.

Runtime Memory Consumption

In terms of memory consumption, our main consideration was the creation of a large amount of named graphs (since every *ContextAssertion* has its own identifier graph). From an insertion and inference lookup point of view, the above analysis seems to indicate that this is not an issue, since Jena TDBs quad indexing scheme is able to efficiently handle the work. To investigate the evolution of the memory consumed by the in-memory TDB quadstore used by the CONSERT engine, we used a Java profiling framework and looked at classes that are in direct relation with the number of generated and updated *ContextAssertions*: com.hp.hpl.jena.graph.Node_URI and com.hp.hpl.jena.graph.Node_Literal. The first class is directly related to the number of named graphs created as each named graph is identified by an URI which becomes an instance of the class. The number of instances of the second class is largely influenced by the annotations of a *ContextAssertion* since it will refer to *ContextAnnotations* such as validity interval, certainty or timestamp which have datatype representations. Apart from the classes mentioned above, we also measured the total heap size at the end of a test run, after having performed garbage collection.

Table 3: Memory consumption and instance count for selected data structures during different test runs. Showing values for configurations with 30, 90 and 150 *ContextAssertion* class types and 50 instances per class

	30 x 50		90 x 50		150 x 50	
	instance count	mem. size	instance count	mem. size	instance count	mem. size
Node_URI	3825	60 KB	8227	131 KB	12478	199 KB
Node_Literal	4385	69 KB	8035	128 KB	11857	189 KB
Total Heap size	25.798 KB		40.676 KB		53.325 KB	

What we were most interested in seeing was how the memory consumption would scale with increasing number of *ContextAssertion* class types and instances. The results of 3 tests can be followed in Table 3. What is readily observable, and an important point, is that memory usage increase between the 3 test runs is sublinear. We consider that the main motive for this result is the existence of the *continuity check*. Though scenario *ContextAssertion* instances are generated randomly, during test runs we observed a fair amount of successful continuity checks (meaning that the content of a new *ContextAssertion* is the same as that of a previously existing one). In such cases, no additional identifying named graph has to be created and only the *ContextAssertion*'s annotation data has to change, leading to a very small memory increase.
Indeed, a theoretical analysis of the potential *ContextAssertions* of a context domain leads us to see that, given the existence of the continuity check, the number of named graphs that identify *ContextAssertions* is bounded by the number of distinct values that the *ContextEntities* involved in the assertion can have. If this amount is either naturally low, or can be made so by considering aggregations or discretization of physical or virtual sensor data, then the scalability of the system in terms of memory consumption can be decently addressed.

Influence of ontology reasoning

We mentioned that the automated test was meant to assess the performance of the event processing capability of the CONSERT Engine. To analyze the influence of ontology reasoning over the background knowledge in the entityStore, we implemented an additional test script using a subset of the *ContextAssertions* that make up the context model of the Smart Conference scenario. Specifically, we simulated a conference room with several sensors that supplied the following *ContextAssertions*: sensesBadgeWithIntensity (1 sensor), hasNoiseLevel (7 sensors), sensesTemperature (4 sensors), sensesLuminosity (4 sensors) – the latter two were not reported in the scenario description in Section 3. We set each sensor to be in sync, sending updates every 10 seconds, and furthermore ensured that at each update three *Derivation Rules* would fire so as to infer the presence of the users badges, the presence of the user in the room and, by inclusion, their presence within the conference building. We simulated a room with 3 people and, in total, at every event cycle there are 18 updates coming in simultaneously from the sensors, plus the ones generated by the derivation rules. This creates conditions similar to the ones in the automated model generation test.
To see the influence of additional reasoning, we performed two test runs. In the first one, the sensors insert a description of themselves into the entityStore (as *ContextEntities* and *EntityDescriptions*) only in the beginning. In the second one, they change

these descriptions whenever they make an update. The entityStore is meanwhile bound to a Jena RDFS reasoner, triggered whenever the entityStore is updated. Table 4 shows the results of the two test runs in terms of the same measurement parameters

Table 4: RDFS reasoning influence in Smart Conference test: min., avg. and max. values for the collected runtime parameters (in ms)

Insertion Delay			Insertion Processing			Inference Delay			Inference Processing			Deduction Duration		
min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max
0	44	677	6	**21**	164	0	33	5254	2	17	82	35	127	935
min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max
0	1056	3756	6	**101**	667	0	34	5056	2	21	101	58	1221	4164

used for the automated test. The upper rows show the values for the case where the entityStore is touched only in the beginning and the lower row for the one where it changes at every *ContextAssertion* update. While we can see that the values corresponding to the execution of *Derivation Rules* (inference delay and processing) stay the same, the important difference is highlighted for the insertion processing measure. Even though only RDFS reasoning is performed, the average processing time is almost five times greater when performing reasoning at every update. Cumulating this difference for 18 updates obviously leads to increased insertion delay times, especially for assertions which get inserted after having been inferred. The reported max values can be considered outliers, since they are always recorded only at the beginning of the test runs, where the combination of TDB initialization and RDFS reasoning drives the waiting and processing times up. This test shows why the *CONSERT Engine Tasking Service* described in section 7.1 is crucial in regulating both event processing and ontology related inferences.

9. Discussion

Throughout this article we have shown the advantages that the proposed context modeling approach and its runtime usage can bring in different situations. However, the model and the current implementation of the CONSERT engine do present some limiting factors.

Current Limitations

A technical limitation observed in Section 8.2 was our usage of TDB transactions to allow consistent views of the context knowledge base when performing updates or inferences. This in turn translated into a limit on the number of *ContextAssertions* that can be updated per second, whilst still attaining decent real-time performance. Besides considering to use alternative quadstore platforms (e.g. Sesame[19]) apart from Jena TDB to see if they provide different transactional behavior, one possible solution is to reconsider the update mechanism. The information in the CONSERT knowledge base has a good logical separation, given that every *ContextAssertion* has its own named graph identifier and the annotations of a given *ContextAssertion* class all reside within the same named graph. Thus, when performing a new insertion of a *ContextAssertion* it is possible to determine what logical partitions of the quadstore are going to be affected by the verification steps applied by the CONSERT engine during the insertion process. We can take advantage of this fact in attempting to implement custom locks or transaction behavior that creates views only for the named graphs concerned by an update. In this way update requests for different *ContextAssertions* could take place concurrently, since their locks or transaction views would not overlap.

Another concern relates to the definition of the *Context Derivation Rules* using the SPARQL syntax. In the current form, the SPARQL query that implements a rule is quite verbose, given our logical named graph separation of *ContextAssertion* instances and the *stores* that record their annotations. A custom CONSERT-specific interpretation engine for the SPARQL queries could help introduce syntactic sugar that drives down the complexity of *Derivation Rule* writing.

Future Work

In future work, we first plan to address the limitations presented above. Whilst presenting related works, we showed that efforts already exist which incorporate SPARQL as part of systems that try to combine semantic and event-driven reasoning [19, 20]. Besides the improvements already sketched out, we plan to investigate if and how the execution cycle and functionality of the CONSERT engine can be augmented by integration with the above mentioned works.
Furthermore, we plan to use the CONSERT engine as part of a larger context management middleware solution, the implementation of which is already on the way. The core of this solution relies on using a multi-agent system to wrap every aspect of the

[19] http://www.openrdf.org/

context provisioning life cycle (acquisition, modeling, dissemination) into individually configurable components, with the aim of increasing deployment and runtime flexibility. We then intend to report on the implementation and performance of a real life test of this context management solution within the ambient intelligence laboratory of our university.

10. Conclusions

Our goal in this work was to obtain a context modeling approach that is able to address the challenges of expressiveness in representation and reasoning by using technology standards of the semantic web, which promote interoperability and ease of usage in open application environments. The Smart Conference scenario implementation described in this paper highlights the flexibility in representation offered by our proposed context meta-model taking the form as the CONSERT Ontology. We have shown how arbitrary arity statements, specifications for structured annotations and integrity constraints ranging over both domain and meta-property information are all addressed with our model. We then introduced the CONSERT engine, a service component that, through its execution cycle, leverages the CONSERT Ontology to provide support for rule-based event processing and ontology reasoning, structured annotation manipulation and constraint detection. Further, it exposes and interacts with a variety of service interfaces that connect the engine to an external management platform allowing an application to control different aspects of the engine execution at runtime. Based on initial testing, we identified existing limits and means by which to increase engine throughput.
We consider that the presented work is promising and has the potential of being useful to support context modeling in ambient intelligence computing.

Acknowledgements

This work has been supported by the French Foreign Ministry through the *"Doctorat en co-tutelle"* scholarship program and by the Sectoral Operational Programme Human Resources Development 2007-2013 of the Romanian Ministry of European Funds through the Financial Agreement POSDRU/159/1.5/S/134398.

References

[1] Dey, A. K. "Understanding and using context." Personal and ubiquitous computing 5.1, 4–7. (2001).

[2] Strang, T., Linnhoff-Popien, C. "A context modeling survey." Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp 2004-The Sixth International Conference on Ubiquitous Computing, Nottingham/England. (2004).

[3] Bettini, C., Brdiczka, O., Henricksen, K., Indulska, J., Nicklas, D., Ranganathan, A., Riboni, D. "A survey of context modelling and reasoning techniques." Pervasive and Mobile Computing. 6, 161–180. (2010).

[4] Perera, C., Zaslavsky, A., Christen, P., Georgakopoulos, D., "Context Aware Computing for The Internet of Things: A Survey," Communications Surveys & Tutorials, IEEE, vol. 16, no. 1, 414–454, First Quarter. (2014)

[5] Strang, T., Linnhoff-Popien, C., Frank, K. "CoOL: A context ontology language to enable contextual interoperability." Distributed applications and interoperable systems. 236–247. (2003).

[6] Chen, L., Nugent, C. "Ontology-based activity recognition in intelligent pervasive environments." International Journal of Web Information Systems 5.4, 410–430. (2009).

[7] Chen, H., Finin, T., Joshi A. "The SOUPA ontology for pervasive computing." Ontologies for agents: Theory and experiences. Birkhuser Basel, 233–258. (2005).

[8] Henricksen, K. "A Framework for Context-Aware Pervasive Computing Applications." PhD Thesis, School of Information Technology and Electrical Engineering, University of Queensland. (2003).

[9] Gu, T., Wang, X., Pung, H. "An ontology-based context model in intelligent environments." Proceedings of Communication Networks and Distributed Systems Modeling and Simulation Conference. (2004).

[10] Krummenacher, R., Strang, T. "Ontology-based context modeling." Proceedings of Third Workshop on Context-Aware Proactive Systems (CAPS). (2007).

[11] Riboni, D., and Bettini, C. "OWL 2 modeling and reasoning with complex human activities." Pervasive and Mobile Computing 7.3, 379–395. (2011).

[12] Grosof, B., Horrocks, I., Volz, R., Decker, S., "Description logic programs: combining logic programs with description logics." Proceedings of the 12th International Conference on the World Wide Web, WWW-2003, 48–57. (2003).

[13] Bucur, O., Beaune, P., Boissier, O. "Representing context in an agent architecture for context-based decision making." Proceedings of the Workshop on Context Representation and Reasoning (CRR05), Paris, France. (2005).

[14] Fuchs, F., Hochstatter, I., Krause M. "A metamodel approach to context information." Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications Workshops. IEEE Computer Society. (2005).

[15] Broda, K., Clark, K., Miller, R., Russo, A. "SAGE: A Logical Agent-Based Environment Monitoring and Control System." Proceedings of the European Conference on Ambient Intelligence, 112–117, Springer-Verlag. (2009).

[16] Toninelli, A., Montanari, R., Kagal, L., Lassila, O. "A semantic context-aware access control framework for secure collaborations in pervasive computing environments." The Semantic Web-ISWC 2006, 473–486. Springer Berlin Heidelberg. (2006).

[17] Bikakis, A., Antoniou, "Defeasible Contextual Reasoning with Arguments in Ambient Intelligence." IEEE Transactions on Knowledge and Data Engineering, 22(11), 1492–1506. (2010).

[18] Meditskos, G., Dasiopoulou, S., Efstathiou, V., Kompatsiaris, I. "SP-ACT: A hybrid framework for complex activity recognition combining OWL and SPARQL rules." 2013 IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops), 25–30. (2013).

[19] Anicic, D., Fodor, P., Rudolph, S., Stojanovic, N. "EP-SPARQL: a unified language for event processing and stream reasoning." Proceedings of the 20th international conference on World Wide Web, 635–644. ACM. (2011).

[20] Teymourian, K., Rohde, M., Paschke, A. "Fusion of background knowledge and streams of events." Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems, 302–313. ACM. (2012).

[21] Pietschmann, S., Mitschick, A., Winkler, R., Meißner, K. "Croco: Ontology-based, cross-application context management." Third International Workshop on Semantic Media Adaptation and Personalization, SMAP'08. 88–93. IEEE. (2008).

[22] Zimmermann, A., Lopes, N., Polleres, A., Straccia, U. "A general framework for representing, reasoning and querying with annotated semantic web data." Web Semantics: Science, Services and Agents on the World Wide Web, 11, 72–95. (2012).

[23] Udrea, O., Recupero, D. R., Subrahmanian, V. S. "Annotated rdf." ACM Transactions on Computational Logic (TOCL) 11.2. (2010).

[24] Kifer, M., Subrahmanian, V. S. "Theory of generalized annotated logic programming and its applications." The journal of Logic Programming 12.4, 335–367. (1992).

[25] Carroll, J. J., Bizer, C., Hayes, P., Stickler, P. "Named graphs." Web Semantics: Science, Services and Agents on the World Wide Web 3.4, 247–267. (2005).