Part I

FRAMING THE PROBLEM AND CURRENT STATE-OF-THE-ART

You can put some informational part preamble text here. Illo principalmente su nos. Non message *occidental* angloromanic da. Debitas effortio simplificate sia se, auxiliar summarios da que, se avantiate publicationes via. Pan in terra summarios, capital interlingua se que. Al via multo esser specimen, campo responder que da. Le usate medical addresses pro, europa origine sanctificate nos se.

Part II

DESIGN ITERATIONS AND CONSTRUCTING A THEORY

You can put some informational part preamble text here. Illo principalmente su nos. Non message *occidental* angloromanic da. Debitas effortio simplificate sia se, auxiliar summarios da que, se avantiate publicationes via. Pan in terra summarios, capital interlingua se que. Al via multo esser specimen, campo responder que da. Le usate medical addresses pro, europa origine sanctificate nos se.

Part III

GENERALISED MODELS, SOFTWARE ARCHITECTURE AND EVALUATION

In this part of the thesis, the more general concepts and techniques that can be applied to ubiquitous computing are described. These concepts and techniques were extracted from work done during the three design iterations.

9 ONTOLOGY ENGINEERING

Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.

— Antoine de Saint Exupéry

An ontology is a representation of knowledge (facts, things, etc.) in terms of concepts within a specific domain, as well as the relationships between them. Ontologies make it easier to publish and share knowledge. They are both machine-readable and human-understandable. The power of ontologies lies in their ability to create relationships among classes of objects, and to assign properties to those relationships that allows us to make inferences about them [55].

The word ontology is used in the literature to mean different things:

- a formal specification of concepts and relations in a domain, using axioms to specify the intended meaning
- an informal specification using UML class diagrams or entity-relationship models
- a vocabulary, or collection of named concepts agreed on by a group, defined in natural language

What these different usages of the word have in common is that an ontology is a *community contract* about the representation of a domain [50]. It also has to be maintained during its lifespan, and is created through clear conceptual modelling based on philosophical notions.

An example of clear conceptual modelling using roles is shown in Section 9.5.1

An Web Ontology Language (OWL) file can be used to represent an ontology or the individuals (instances) it describes, or both the ontology and its instances can be contained within the same file. For example, the concept `Man` could be defined as part of the ontology, and the individual `Gerrit` would be an instance of `Man`. The different types of restrictions that can be defined in OWL are shown in Table 6, together with the various syntaxes that can be used to represent these restrictions.

Even without using a reasoner to infer new facts, an ontology improves the usefulness of the data. Using unique identifiers to

RESTRICTION	DL	MANCHESTER	OWL
Existential	$\exists$	`some`	`owl:someValuesFrom`
Universal	$\forall$	`only`	`owl:allValuesFrom`
Value	$\ni$	`value`	`owl:hasValue`
Equivalence	$\equiv$	`equivalentTo`	`owl:equivalentProperty,` `owl:equivalentClass`
Cardinality	$=$	`exactly`	`owl:cardinality`
Minimum cardinality	$\geqslant$	`max`	`owl:minCardinality`
Maximum cardinality	$\leqslant$	`min`	`owl:maxCardinality`

Table 6: OWL restriction definitions using different syntaxes: Description Logic, Manchester OWL Syntax[31] and OWL syntax

represent concepts and relationships enables a computer to find and aggregate new information. For example, the relationship `knows` in the Friend-Of-A-Friend (FOAF) ontology can be used to find and aggregate relationships between two individuals, where asserting

```
:Jun :knows :Gerrit .
:Gerrit :knows :Bram .
```

we can infer that

```
:Jun :knows :Bram .
```

We distinguish between four layers of ontologies, that are used to present concepts ranging from the more general to the more specific: foundational ontologies, core ontologies, domain ontologies and application ontologies.

9.1 LAYERS OF ONTOLOGIES

9.1.1 *Foundational ontologies*

Foundational or upper ontologies are aimed at modelling very basic and general concepts, as to be highly reusable in different scenarios [91]. They are used to align concepts in other ontologies, and to ensure consistency and uniqueness of these concepts. Examples of foundational ontologies include Descriptive

Ontology for Linguistic and Cognitive Engineering (DOLCE), Basic Formal Ontology (BFO), OpenCyc and Suggested Upper Merged Ontology (SUMO). These ontologies can serve as reference ontologies when a new ontology is developed.

9.1.2 *Core ontologies*

Core ontologies are used to model knowledge about a specific field. A core ontology is based on a foundational ontology and should be modular and extensible [91]. A number of core ontologies exist for modelling things like events and multimedia objects. Core ontologies refine foundational ontologies by adding field-specific concepts and relations. The Event-Model-F ontology, for example, is used to model the causality, correlation and interpretation of events, and is based on DOLCE+DnS UltraLight (DUL). Core ontologies achieve modularity and extensibility by following a pattern-oriented approach. Event-Model-F uses the Descriptions and Situations (DnS) and Information Object patterns provided by DUL.

The Core Ontology Multimedia (COMM) ontology is used represent multimedia objects such as images, video and audio, and is also based on DUL. An audio recording could be modelled as `AudioData`, while a text description could be modelled as `TextData`. However, `AudioData` (a subconcept of DUL `InformationObject`) represents the information that is contained in the audio recording, not the digital audio stream itself [91]. The location of the audio file is represented with a concept that denotes the Uniform Resource Identifier (URI).

9.1.3 *Domain ontologies*

Domain ontologies represent reusable knowledge in a specific domain and are usually handcrafted. The Gene ontology, for example, describes gene products in terms of their biological processes, cellular components, and molecular functions in a species-independent manner [55].

Ontologies are particularly well-suited to domains such as biomedical research, where there is an abundance of available data with non-hierarchical relationships.

9.1.4 *Application ontologies*

An application ontology is created for a specific application, so they are not considered to be reusable. However, the tools or processes used to create the ontology may be reusable. The

Cell Cycle ontology[1], for example, is specific to modelling the cell cycle process, a rather specialised domain.

9.2 OUR APPROACH

In the following sections we will describe our approach to modelling ontologies, as well as ontology design patterns that we have identified. First we introduce the features of OWL that we used. OWL can be used to define classes and relationships, as well as restrictions. A restriction is used to define a formal description of a class that restricts class membership [3].

An example of an OWL restriction is shown in Section 9.3.5.

In some cases we need more expressiveness than what is allowed by OWL. Rule languages go beyond what can be expressed by OWL, or can be easier to understand [49]. We made use of Semantic Web Rule Language (SWRL) in the first design iteration, and in some of the examples in this chapter. Later, we discovered some limitations of SWRL, like not being able to construct new individuals. We also experienced some performance issues when using SWRL.

SWRL performance issues are described in more detail in Section 4.5.

This necessitated the switch to another way of defining rules, called SPARQL Inferencing Notation (SPIN). SPIN allows us to specify rules in SPARQL Protocol and RDF Query Language (SPARQL). These SPARQL rules are contained within the ontology itself. The TopSPIN reasoning engine, implemented in our version of the Semantic Information Broker (SIB), supports both OWL 2 and SPIN.

9.3 REASONING WITH OWL

In order to make the data generated by the smart environment more useful, we need a consistent way of understanding the combination of data from multiple sources. Reasoning or inferencing provides a robust solution to understanding the meaning of novel combinations of terms [49]. A reasoner may be used for truth maintenance, belief revision, information consistency and information creation in an information space [78].

As of October 2009 the OWL 2 Web Ontology Language is the W3C recommendation for creating ontologies. Most semantic reasoners have some kind of support for OWL as well as support for a rule language like SWRL:

1 http://www.CellCycleOntology.org

- Pellet (Java): Supports OWL 2 and SWRL (DL-safe rules), has a command-line option with `explain` command.
- Fact++ (C++): Supports OWL DL, does not fully support OWL 2.
- HermiT (Java): Supports OWL 2 and SWRL (DL-safe rules without built-ins), uses hypertableau calculus to perform reasoning, comes pre-installed with Protégé editor, has a command-line option.
- TopSPIN (Java): Supports OWL 2 RL/RDF Rules defined as SPIN rules, comes pre-installed with TopBraid Composer.

Let us now look at a number of services provided by reasoners.

Subsumption testing

One of the services provided by a reasoner is to test whether or not one class is a subclass of another class, also known as subsumption testing. The descriptions of the classes are used to determine if a superclass/subclass relationship exists between them. It also infers disjointness and equivalence of classes. By performing such tests on the classes in an ontology it is possible for a reasoner to compute the inferred ontology class hierarchy. The reasoner can also determine class membership for individuals based on their properties, i.e. class membership does not always have to be asserted. It is also possible to infer new property relations with other individuals.

Subsumption refers to the reflexive, transitive and antisymmetric relationship between classes, that states that a class A subsumes a class B if and only if the set of instances of class A includes the set of instances of class B [83]. The same principle holds for OWL properties.

Preuveneers and Berbers [83] evaluated the Pellet ontology reasoner on a smart phone for semantic matching, but it was considered unsuitable due to performance requirements. They developed an encoding scheme to provide a compact representation of subsumption relationships. It is based on the idea that subsumption of classes in an ontology is somewhat related to multiple inheritance in an object-oriented programming language, which means that inheritance-encoding algorithms can be used for subtype testing. However, the algorithm cannot test for satisfiability - whether instances of a specific class can actually exist.

Being able to use a reasoner to automatically compute the class hierarchy is one of the major benefits of building an ontology using OWL. When constructing large ontologies the use of a reasoner to compute subclass-superclass relationships between classes becomes almost vital. Without a reasoner it is very difficult to keep large ontologies in a maintainable and logically correct state.

With ontologies it is possible for a class to have many superclasses, also called multiple inheritance. Usually it is easier to construct the class hierarchy as a simple tree, and leave computing and maintaining multiple inheritance to the reasoner. Classes in the asserted hierarchy therefore have no more than one superclass. This helps to keep the ontology in a maintainable and modular state and minimises human errors that are inherent in maintaining a multiple inheritance hierarchy. For example, in our ontology we have

```
AlarmSetEvent rdfs:subClassOf SetEvent .
TimeSetEvent rdfs:subClassOf SystemEvent .
```

where `TimeSetEvent` is asserted to belong to one superclass `TimeSetEvent`, but could also be inferred by the reasoner to belong to `SetEvent` if that is preferred.

9.3.1 *Consistency checking*

A reasoner performs consistency checking to check whether all axioms and assertions are consistent. Based on the description of a class the reasoner can check whether or not it is possible for the class to have any instances. A class is deemed to be inconsistent if it cannot possibly have any instances.

9.3.2 *Necessary versus necessary and sufficient*

RDF Schema (RDFS) is a subset of OWL [3].

A *necessary* condition will allow a class to be inferred as a subclass (`rdfs:subClassOf`), compared to a *necessary and sufficient* condition, which will make a class equivalent to another class (`owl:equivalentClass`). The second condition usually requires an intersection of classes to be defined using the `and` keyword.

9.3.3 *Inverse properties*

If one defines a new inverse property of an existing property with a specified domain and range, the inverse domain and range will be inferred for new individuals with this property. As an example:

Note that in Protégé this inverse domain and range might not show up for the property itself, but that it will be inferred for new individuals.

SmartObject $\equiv$ isSmartObject $\exists$ `Self`

This can also be represented in OWL as:

```
:SmartObject
    a owl:Class ;
    owl:equivalentClass
    [   a owl:Restriction ;
        owl:hasSelf "true"^^xsd:boolean  ;
        owl:onProperty :isSmartObject
    ] .
```

Any individual that is related to itself via the `isSmartObject` property will be identified as an instance of `SmartObject`, and any individual asserted as an instance of SmartObject will be related to itself via that property [52].

9.3.4 *Property chains*

A new feature introduced in OWL 2 is property chains, which allows for the specification of the propagation of a property along some path of interconnected properties [53]. Examples of property chains are shown in Section 4.4.2 and Section 7.4.

9.3.5 *Using cardinality restrictions*

When modelling cardinality in OWL 2, one might expect to be able to infer that an individual is a member of a class based on a cardinality restriction, for example

```
TwoButtonDevice rdfs:SubClassOf Device hasButton exactly 2 Button
```

Unfortunately, due to the Open World Assumption (OWA), it cannot be known whether an individual might have additional properties of that type. The only way to identify an individual is using minimum cardinality. However, this approach can be problematic if the concept is underspecified [53].

In OWL 2, it is possible to define a Qualified Cardinality Restriction (QCR), which means the cardinality restriction can be applied to a specific class [49].

This means that it is possible to define that a smart object has only one current state:

```
SmartObject
rdfs:subClassOf
          [
            rdf:type owl:Restriction;
            owl:qualifiedCardinality 1;
            owl:onProperty hasCurrentState;
            owl:onClass State
          ];
```

If we then assert a certain smart object to have two current states, e.g.

```
phone1 hasCurrentState playing .
phone1 hasCurrentState stopped .
```

Individuals are distinct if it is asserted that they are different from one another.

it will violate the QCR if `playing` and `stopped` are distinct. In earlier versions of OWL, it was not possible to define a specific class for a cardinality restriction.

9.4 REASONING WITH SPIN

SPIN[2] is a W3C Member Submission created and maintained by TopQuadrant, who is also responsible for the TopBraid Composer ontology editor. With SPIN, rules are expressed in SPARQL, the W3C recommended Resource Description Framework (RDF) query language, which allows for the creation of new individuals using CONSTRUCT queries. Let us now look at some features of SPIN.

9.4.1 *Integrity constraints*

SPIN allows us to specify integrity constraints, e.g. that

```
:event1 :generatedBy :device1
```

should exist. Domain and range are not integrity constraints, but allow us to infer for example the class type of new individuals, e.g. if

2 http://www.spinrdf.org

```
:generatedBy rdfs:range :SmartObject
```

then asserting

```
:event1 :generatedBy :device1
```

would infer

```
:device1 rdf:type :SmartObject
```

9.4.2 *SPARQL Rules*

SPIN allows for fine-grained control of how rules are executed. For example, it is possible to have a rule fire only once, by setting the SPIN property `spin:rulePropertyMaxIterationCount` to 1, in cases where new inferences could cause the rule engine to iterate infinitely. It is also possible to specify the order in which rules are executed using `spin:nextRuleProperty`.

9.4.3 *Built-in SPARQL Functions*

Built-in functions with `fn:` (XPath/Xquery) or `afn:` (ARQ Functions) prefix are also available as part of ARQ, the Jena query engine. The `spif:` prefix denotes the SPIN Standard Functions Library.

SPIN has a number of built-in functions[3] that provides additional functionality not available in OWL 2. These built-in functions can be very helpful when creating your own SPIN rules, functions or magic properties. They can be used to retrieve substrings (`fn:substring`), perform modulo arithmetic (`spif:mod`), or generate random numbers (`spif:random`).

An example of where they are used in our ontology is the `afn:now()` function in the `currentDateTime` magic property:

```
SELECT ?datetime
WHERE{BIND(afn:now() AS ?datetime) .
}
```

Magic properties are described in Section 9.4.5.

Some built-in functions, like `spif:buildUniqueIRI` (used to create new URIs), are only available as part of the extended TopBraid SPIN API[4], and cannot be used with the free open-source edition[5]. That said, it is possible to build your own `buildURI` function using `fn:concat` as we did in the second design iteration:

```
BIND (IRI(fn:concat("example.com#mediaPath_", afn:localname(?this), "_to_",
afn:localname(?x3))) AS ?mp) .
```

3 The reference documentation for the built-in functions can be accessed in TopBraid Composer from `Help` → `Help Contents` → `TopBraid Composer` → `Reference` → `SPARQL Functions Reference`
4 Available under a commercial license from TopQuadrant
5 http://topbraid.org/spin/api/

9.4.4 Custom functions

If you use the `.spin.rdf` extension to store the ontology file, custom functions will be loaded into TopBraid Composer on startup.

It is possible to create your own custom functions in SPIN. These functions are written in SPARQL and stored in the ontology. An example of a custom function we built[6] is `getMaxDateRsc`, which is used to retrieve the last interaction event that was generated by a specific smart object:

```
SELECT ?lastEvent
WHERE{
    ?lastEvent events:generatedBy ?arg1 .
    ?lastEvent events:inXSDDateTime ?last .
}
ORDER BY DESC (?last)
LIMIT 1
```

This was then combined with a SPIN rule to create an object for the `hasLastEvent` property:

```
CONSTRUCT{
    ?this events:hasLastEvent ?lastEvent .
}
WHERE{BIND (events:getMaxDateRsc(?this) AS ?lastEvent) .
}
```

The SPIN rule is required as magic properties cannot be used in local restrictions on their own.

When loading an ontology with SPIN functions into Jena, the functions should be registered using

```
SPINModuleRegistry.get().registerAll()
```

An extension of SPIN, called SPINx, allows for the definition of more elaborate custom functions using JavaScript. Unfortunately it cannot access the triple graph at execution time, but it does operate on arguments. Jena allows similar functionality to SPIN and SPINx functions using a `FunctionFactory`, which allows you to define and register your own functions in Java.

9.4.5 Magic properties

Magic properties, also called property functions, may be used in SPIN to dynamically compute values, even if there are no corresponding triples in the model. For example, we created the magic property `currentDateTime` with the SPIN body

The inferencing engine does not always infer superclasses for SPARQL queries, which could cause problems for magic properties.[7]

6 With help from Scott Henninger and Holger Knublauch from TopQuadrant

```
SELECT ?x
WHERE{BIND (afn:now() AS ?x) .
}
```

When we now create a query for something like

```
:phone1 :currentDateTime ?date
```

the current date/time is returned as an object. This allows us to write Knowledge Processor (KP) queries at triple-level, without having to send a SPARQL query from the KP to the SIB. Magic properties are more flexible than SPIN functions and can return multiple values.

9.5 ONTOLOGY DESIGN PATTERNS

In software engineering, design patterns are generalised solutions to problems that commonly occur in a specific software context. An example of such a pattern is the observer pattern, in which a software object maintains a list of observers which are notified of state changes. The observer pattern is one of the original patterns described in the seminal book on design patterns by the Gang of Four (GoF) [40]. The blackboard pattern, used in our software architecture, is a generalised version of the observer pattern that allows multiple readers and writers.

The blackboard pattern was first mentioned in Section 1.2.5.

A similar approach to design patterns has been applied to ontologies [42, 51, 29]. Dodds and Davis [29] used the following pattern template to document an ontology design pattern in their book “Linked Data Patterns”:

- Question - A question indicating the problem the pattern is designed to solve
- Context - Description of of the goal and context of the pattern
- Solution - Description of the pattern
- Example(s) - Real-world implementations that make use of this pattern
- Discussion - Analysis of the pattern and where it can be used
- Related - List of comparable patterns

They formalised a number of linked data patterns into a pattern catalogue, and we will now use the same pattern template to describe ontology design patterns that can be applied in the context of smart environments. In this section we first look at three examples of existing ontology design patterns, before we focus on new patterns that were identified during the course of the work described in this thesis.

One of the example patterns, DnS, is an ontology design pattern provided by the Ontology Design Patterns (ODP) initiative[9]. They maintain an entire online library of ontology design patterns, to be used as building blocks for creating new ontologies.

ODP distinguishes between a number of different pattern types, including:

- Content patterns, e.g. the Role pattern that defines `Student` as a role instead of a subclass of `Human`
- Logical patterns, like the n-ary relation or Situation pattern
- Reengineering patterns, e.g. converting microformats to RDF
- Alignment patterns, e.g. aligning FOAF with the VCard format
- Anti-patterns, e.g. modelling `City` as a subclass of `Country`

The first example pattern below, called the Role pattern, is required reading for understanding the DnS pattern.

9.5.1 *The Role pattern*

How can we represent the roles of devices and agents in an ontology?

Context

An example of clear conceptual modelling is that a `Student` is not a subclass of `Human`, but a *role*.

Solution

Roles can be modelled as classes, individuals or properties.

9 http://ontologydesignpatterns.org/wiki/Submissions:DescriptionAndSituation

Example(s)

Roles can be modelled as classes:

```
Object rdf:type Role
```

or as individuals:

```
Jim rdf:type Person .
SongWriter rdf:type Role .
Jim hasRole SongWriter .
```

or even as properties:

```
Table legs Books
```

where books are being used in the role of table legs.

Discussion

A commonly occurring issue when modelling ontologies is to whether model the concept as a property or a class. Consider the role *student*, where `Mark` can be seen as either an individual of the `Student` class, or have a relationship via a `student` property with his university. Classes have stronger ontological commitment[10] than properties, but using properties are often more convenient for practical use [52]. OWL 2 punning allows an entity to be treated as both a property and a class without comprising ontological commitment.

Related

- The Role pattern is described in detail in Hoekstra's PhD thesis [51]
- The Time Indexed Person Role Pattern [42]

9.5.2 *Descriptions and Situations (DnS) pattern*

How do we model non-physical objects like plans, schedules and context in an ontology?

10 See Section 11.2.1

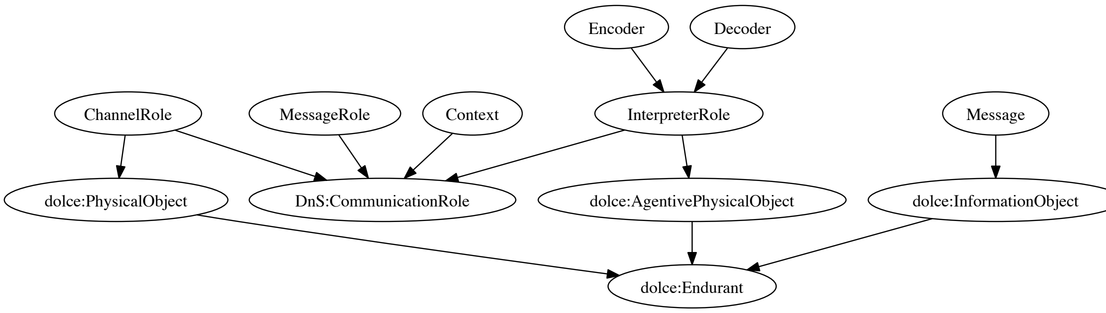

Figure 50: Example of modelling communication theory using DnS and DOLCE

Context

While modelling physical objects using an ontology is relatively straightforward, it becomes non-trivial when modelling *non-physical objects* [41] such as plans, schedules, social constructs, etc. Existing theoretical frameworks like Belief-Desire- Intention (BDI) theory and situation calculus are not at the level of concepts or relations, which we need to be able to model non-physical objects as a set of statements. The DnS pattern grew out of the work done on the DOLCE ontology to solve this problem.

During a summer school attended by the author, Aldo Gangemi (co-creator of DOLCE) mentioned that he considers DOLCE to be a collection of ontology design patterns.

Solution

The DnS design pattern provides an ontological formalisation of context [91]. It achieves this by using *roles* to classify entities into a specific context. The pattern defines a *situation* that satisfies a *description*. The `describes` object property is used between a `Description` and an object, while the `satisfies` object property relates a `Situation` with a `Description`.

Example(s)

As an example, consider communication theory [95] as modelled with DnS in Figure 50, where there is an encoder, a message, a context[11], a code and channel. In DnS, the encoder and decoder are modelled as agentive physical objects in DOLCE, while the channel is a non-agentive physical object. Messages are considered information objects.

11 What the message is about, not the circumstances surrounding the communication

Discussion

With DnS one can also reify events and objects and describe the n-ary relation that exists between multiple events and objects.

Related

- The DUL ontology [42]

9.5.3 *Defining n-ary relations*

How do we represent relations among more than two individuals?

Context

In OWL, a property is a binary relation between two individuals. However, some relationships are not binary and involve more than two resources, for example when modelling events.

Solution

We can use n-ary relations [76] to model relationships between more than two resources. A class is created to represent the relationship, with an instances of the class used to represent the relationship between the various resources.

Example(s)

`event-43495d51-29e3-11b2-807e-ac78eefc1f83` is an example of an `Event` instance that represents the n-ary relation between the device `phone1` and the various event resources:

```
:phone1 :generatesEvent :event-43495d51-29e3-11b2-807e-ac78eefc1f83.

:event-43495d51-29e3-11b2-807e-ac78eefc1f83
        rdf:type :IncreaseLevelEvent ;
        :inXSDDateTime "2012-01-17T11:23:06.887+01:00"^^xsd:dateTime ;
        :dataValue 255 ;
        :duration "PT3S"^^xsd:duration .
```

Discussion

This pattern is commonly used to represent complex relationships. This is quite a powerful pattern, as it can also be used to define the temporal order of sequences [76].

Related

- Qualified Relation pattern [29]

9.5.4 *Naming interaction events*

How should the URI of an interaction event be structured so that the name forms a natural hierarchy?

Context

Interaction events tend to form natural groups, such as events related to a specific device class. Reflecting these groups in the name of the interaction event itself makes it easier for developers to understand existing and/or inferred groupings, and to classify new events into an existing hierarchical event structure.

Solution

We use the notation

```
[DeviceClass][Action]Event
```

to define the interaction event.

Example(s)

Consider a simple light switch with two states, `Up` and `Down`. We can define two interaction events, `switchDownEvent` and `switchUpEvent`, which can then later be grouped by either device class or by action.

Discussion

If the naming convention of a URI follows a common pattern, they become easier to remember and easier to work with. They can even be constructed automatically. It makes the URI human-readable and improves the relation between the name and the event it describes.

Related

- Hierarchical URIs [29]
- Patterned URIs [29]

9.5.5 *Using local reflexivity in property chains*

How can we specify classes as part of an OWL 2 property chain?

Context

Sometimes it is necessary to restrict property chains to specific classes. We need to be able to specify these classes as part of the property chain.

Solution

The `self` keyword[12] is used to indicate local reflexivity (also called a self restriction) in OWL 2 and can be used to transform classes to properties when creating property chains.

Example(s)

We can apply local reflexivity to the class `Student`, for example

$\text{Student} \equiv \text{isStudent}\ \texttt{some self}$

If the individual Mark has a `isStudent` relation with itself, it will be inferred that Mark is a `Student`. Also, if Mark is asserted as a `Student`, then the `isStudent` property will be inferred. This can then be combined with property chains where necessary, e.g.

$\text{hasRole} \circ \text{isStudent} \sqsubseteq \text{student}$

Discussion

In his PhD thesis on ontology design patterns, Hoekstra [51] uses this pattern extensively to model actions, beliefs, intentions and social constructs. For example,

$$\begin{aligned} \text{Intention} &\equiv \text{isIntention}\ \texttt{some self} \\ &\sqsubseteq \text{PropositionalAttitude} \\ \text{holds} \circ \text{isIntention} \circ \text{towards} &\sqsubseteq \text{intends} \end{aligned}$$

12 Manchester syntax, used when editing ontologies in Protégé and other ontology editors. See Table 6.

Related

- DnS pattern

9.5.6 *Semantic matching with property chains*

How can we perform semantic matching of functionalities between devices using property chains?

Context

Property chains are useful for semantic matching, but with basic property chains the inverse is inferred as well, which is not always desired. Property chains cannot be made irreflexive, as only *simple* properties can be irreflexive in order to guarantee decidability [8]. Defining domain and range to as constraints just makes the ontology inconsistent. Thus, when using property chains, the properties involved need to be symmetric, as in

hasFunctionality $\circ$ isFunctionalityOf

Solution

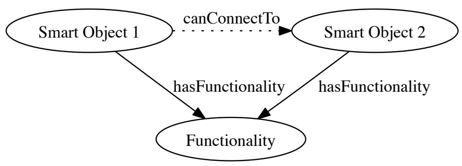

Figure 51: Two individuals related to the same object

When we have two individuals with the same object, but different predicates (see Figure 51), and we want to infer a new property, this is intuitively represented in SWRL:

`hasFunctionality(?s1,?f), hasFunctionality(?s2, ?f)` $\Rightarrow$ `canConnectTo(?s1`

However, this cannot be represented in the same fashion using a property chain, as

hasFunctionality $\circ$ hasFunctionality $\sqsubseteq$ canConnectTo is not equivalent. This is however, easily solved by introducing an inverse property `isFunctionalityOf`, and the property chain becomes

$$\text{hasFunctionality} \circ \text{isFunctionalityOf} \sqsubseteq \text{canConnectTo}$$

Modelling the above using the Relation Partition Algebra (RPA) of Feijs [36], where
$\text{hasFunctionality} \circ \text{hasFunctionality}^{-1} \subseteq \text{canConnectTo}$

shows the property chain can also be represented using RPA, apart from the inverse relation, which is denoted by $R^{-1} = \{(x,y)|(y,x) \in R\}$.

Example(s)

First we define two smart objects and their corresponding functionalities:

```
:Music a :Functionality .

:phone1 a :SmartObject .
:phone1 :functionalitySource :Music .

:speaker1 a :SmartObject .
:speaker1 :functionalitySink :Music .
```

Using the property chain

$$\text{functionalitySource} \circ \text{isFunctionalityofSink} \sqsubseteq \text{canConnectTo}$$

where `isFunctionalityofSink` is the inverse property of `functionalitySink`, we can infer that

```
:phone1 :canConnectTo :speaker1 .
```

Discussion

There are two caveats when using property chains to perform semantic matching. First, OWL 2 property chains cannot be built with datatype properties, only object properties, i.e. use

```
:device1 :hasFunctionality :Audio
```

instead of

```
:device1 :hasFunctionality "audio"
```

This means we cannot infer

```
:device1 :hasRFIDTag "ABCD123F"
```

and we have to use a rule language like SWRL or SPIN.

SWRL was used for semantic matching in the second design iteration in Section 4.4.2. SPIN was used in the third design iteration, with the implementation described in more detail in Chapter 7.

The second caveat is that property chains cannot be used for cardinality restrictions. We have only tested this with the Pellet reasoner, and it is possible that other reasoners could allow for this to happen.

Related

- The Role pattern

9.5.7 *Inferring new individuals*

How can new individuals be created when an existing literal value changes?

Context

Ontology languages like OWL are used to classify existing individuals, not create new ones. In some cases we want to insert a new individual when a literal value changes or is inserted. When using only OWL and DL-safe rules (e.g. SWRL), no new individuals may be inserted, and the work-around is that individuals are pre-populated in the triple store. For example, if `OnEvent` and `OffEvent` are pre-populated, you can model that

```
:event1 :dataValue 1
```

should infer

```
:event1 :mappedTo :OnEvent
```

SWRL built-in atoms in rule heads [49] present another solution to this problem, but these built-in atoms cannot be handled by reasoners like Pellet, which only supports DL-safe rules.

Solution

A SPARQL CONSTRUCT query, defined as a SPIN rule, can be used to insert a new individual into the triple store.

Example(s)

A new individual, representing a media path, can be inferred using:

```
CONSTRUCT{
    ?mp a sc:MediaPath .
    ?x3 sc:hasMediaPath ?mp .
    ?mp bonding:mediaSourceSO ?x2 .
    ?mp bonding:mediaOriginator ?this .
}
WHERE{
    ?this sc:convertsMediaType ?x2 .
    ?x2 sc:convertsMediaType ?x3 .
    ?this sc:connectedTo ?x3 .
    BIND (IRI(fn:concat( "example.com/ontology#mediaPath_" , afn:localname(?this),
      "_to_" ,afn:localname(?x3))) AS ?mp) .
}
```

In the example, a new `mediaPath` individual is created if two smart objects are connected to each other and there is a `mediaSourceSO` (semantic transformer) that converts the media types between them. This could be a media player transmitting music as source, an ambient lighting object that accepts RGB colour values as sink, and a semantic transformer that converts audio streams into RGB lighting information. For more information about media paths and semantic transformers, see [72].

The `?this` variable indicates to SPIN how the definition should be applied to the members of a class, as the rule itself is defined as part of the class definition - thus defining the scope of the query. `fn:concat` and `afn:localname` are SPIN functions used to concatenate the name of the individual and retrieve the local names of the variables used respectively.

Discussion

When a new individual is inserted using a SPIN rule, care should be taken in how the name of the individual is generated. If we define the new individual as a blank node, the TopSPIN reasoning engine will not terminate, because a new blank node is defined with each iteration. The same issue arises if we assign a random value as the name. Using a fixed URI is a simpler solution, as shown in the example above.

Related

None.

9.5.8 *Removing inferred triples*

How do we remove inferred triples from the triple store when an asserted triple changes?

Context

Removing inferred triples when an asserted triple changes, or is deleted from the model, can be notoriously difficult. For irreflexive properties, it is possible to use constraint violations to detect them, and then remove them one by one. Unfortunately constraint violation checking is very slow, for example taking 834 ms when the inferencing itself takes only 313 ms[13]. Creating a SPIN rule to clean up irreflexive properties does not work, as the properties get inserted and removed after each iteration of the inference engine.

Solution

Two models are used in the triple store, one for the asserted model and one for the inferred model. The inferred model is cleared before each reasoning iteration.

Example(s)

Not applicable.

Discussion

According to TopQuadrant[14], removing inferred triples based on a triple that was deleted is a tricky use case, requiring a `BufferingGraph` that is not available in the open source SPIN Application Programming Interface (API).

Related

None.

9.5.9 *Inferring subclass relationships using properties*

Can we infer subclass relationships based on existing properties using OWL?

13 Based on a model size of 2304 inferred triples
14 `topbraid-users` mailing list discussion

Context

Suppose we wanted to use an object property called `mappedTo` to create a mapping between interaction events, for example

```
SwitchUpEvent mappedTo SwitchOnEvent
```

This prompts the question: Is it possible to create an OWL restriction that says

If Class A is related via Property B to Class C, then Class A is a subclass of Class C.

When modelled in SPARQL, it looks like this:

```
CONSTRUCT{
        ?A rdfs:subClassOf ?C .
}
WHERE{
        ?A :B ?C .
}
```

Solution

Evidently, this could be implemented as a SPIN rule, but we would prefer an OWL-only solution. It turns out that while it is not possible in OWL 2 DL, it is possible in OWL 2 RL/RDF Rules:

```
:B rdfs:subPropertyOf rdfs:subClassOf
```

Example(s)

To solve our original problem in the Context, we would define

```
mappedTo rdfs:subPropertyOf rdfs:subClassOf
SwitchUpEvent mappedTo SwitchOnEvent
```

which would then infer

```
SwitchUpEvent rdfs:subClassOf SwitchOnEvent
```

Discussion

This simple but powerful pattern is a good example of meta-modelling.

Related

None.

9.5.10 *Inferring connections between smart objects and semantic transformers*

When we use semantic transformers to control devices, how can we infer these connections between the smart objects and the semantic transformer?

Context

In the sleep use-case, a semantic transformer was implemented in order to generate lighting values for the dimmable lamp to create the desired wakeup experience. During the implementation, several observations and decisions were made:

- Between smart objects and semantic transformers only `indirectlyConnectedTo` connections can exist, as the semantic transformers are virtual entities that cannot be directly connected to smart objects using the Connector object.
- When a `canIndirectlyConnectTo` relationship is inferred between smart object A and the semantic transformer B, and between B and smart object C, a `canConnectTo` relation between A and C should be inferred (transitive).
- When a connection is made between two smart objects that can be connected through a semantic transformer, the semantic transformer is connected to the smart objects with `indirectlyConnectedTo` relationships, and a `connectedTo` relationship between the smart objects is then automatically inferred.
- A semantic transformer thus acts as a bridge.
- A semantic transformer is *not* a smart object.

When using semantic transformers to control other smart objects, we could make use of the n-ary ontology design pattern, which was also applied to creating media paths in Section 4.4.2 on semantic matching:

- Subscribe to `controlSource` to see if it becomes a control source

- When it becomes a control source, subscribe to the events generated by the control originator

While this is feasible, it is complicated and we would like to use a simpler solution using `connectedTo` relationships. What we would like to infer is shown in Figure 52.

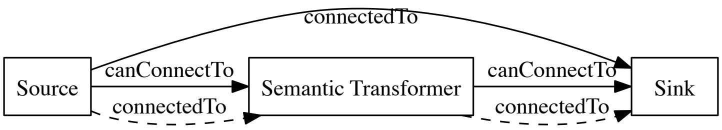

Figure 52: Inferring `connectedTo` relationships between sources/sinks and a semantic transformer

Solution

At first glance, it seems like this might be expressed using property chains and local reflexivity, as described in the ontology design pattern in Section 9.5.5.2. However, this is a special case which cannot be expressed in OWL. It can, however, easily be expressed as a SPIN rule as follows:

```
CONSTRUCT{
    ?source :connectedTo ?semanticTransformer .
    ?semanticTransformer :connectedTo ?sink .
}
WHERE{
    ?source :canConnectTo ?semanticTransformer .
    ?semanticTransformer :canConnectTo ?sink .
    ?source :connectedTo ?sink .
}
```

Example(s)

If the following triples are asserted:

```
:phone1 a :SmartObject .
:phone1 :functionalitySource :Alarm .

:lamp1 a :SmartObject .
:lamp1 :functionalitySink :AdjustLevel .
```

```
:wakeup1 a :SemanticTransformer .
:wakeup1 :functionalitySource :Alarm .
:wakeup1 :functionalitySink :AdjustLevel .

:phone1 :connectedTo :lamp1 .
```

Using the pattern defined in Section 9.5.6, we infer:

```
:phone1 :canConnectTo :wakeup1 .
:wakeup1 :canConnectTo :lamp1 .
```

Using this pattern, we infer the following `connectedTo` relationships:

```
:phone1 :connectedTo :wakeup1 .
:wakeup1 :connectedTo :lamp1 .
```

Discussion

RPA was first mentioned in Section 9.5.6.2.

In some cases, SPIN rules can be easier to compose and interpret than ontology restrictions and property chains. In this case, it cannot even be expressed in OWL, as OWL has no support for modelling property intersections. In RPA, on the other hand, relations are first class citizens, and Figure 52 can be composed using:

$$\text{connectedTo} \subseteq \text{canConnectTo} \cap \text{connectedTo} \circ \text{canConnectTo}^{-1}$$

Related

- N-ary pattern
- Semantic matching with property chains

9.6 DISCUSSION

When applying inference to the physical world, the level of ambiguity and uncertainty is quite high. A system might infer that you are in a room because your RFID badge is in a room. What if you forgot your badge in the office? The challenge is to figure figure out what functions in the smart home are possible with limited inference, which are possible only through inference, and which require an oracle [33]. Systems that rely on inference will be wrong some of the time, and users will need to have models to figure out how the system arrives at its conclusions, along with ways to override the system's behaviour.

Sabou [90] argues that smart objects will require more sophisticated reasoning mechanisms than what is currently used in the area of sensor networks, which primarily relies on subsumption matching. They expect that smart spaces will rely on rule engines rather than DL reasoners, and that the ambiguities and uncertainties in smart environments will require fuzzy or probabilistic methods.

Throughout the development of the ontology, we tried to avoid rule-based formalisms where possible, to see to what extent we can push the limits of OWL 2's expressive power. Hoekstra and Beuker [53] noted that to avoid problematic interactions between the two formalisms, it is undesirable to combine them. However, they also accepted that it is sometimes unavoidable, given the real problems that occur with elaborate concepts.

It is our experience that people commonly underestimate the differences between data modelling and ontology engineering. While some concepts in an ontology can be modelled using UML class diagrams or represented using Java objects, there are some fundamental differences. Data modelling does not allow for axiomatisation to specify the semantics of the information, nor is it much concerned with conceptual modelling based on philosophical notions.

However, much is already gained with using some simple ontology engineering techniques, such as unique identifiers or distinguishing between actors and roles. As James Hendler, one of the authors of the seminal article on the Semantic Web in Scientific American [12], once stated, "a little semantics goes a long way".

Part IV

APPENDIX