INTERNATIONAL STANDARD

ISO/IEC 5087-1

First edition
2023-08

# Information technology — City data model —

## Part 1:
## Foundation level concepts

Reference number
ISO/IEC 5087-1:2023(E)

## Contents

Page

## Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives or www.iec.ch/members_experts/refdocs).

ISO and IEC draw attention to the possibility that the implementation of this document may involve the use of (a) patent(s). ISO and IEC take no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, ISO and IEC had not received notice of (a) patent(s) which may be required to implement this document. However, implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at www.iso.org/patents and https://patents.iec.ch. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*.

A list of all parts in the ISO/IEC 5087 series can be found on the ISO and IEC websites.

Any feedback or questions on this document should be directed to the user’s national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

# Introduction

The intended audience for this document includes municipal information systems departments, municipal software designers and developers, and organizations that design and develop software for municipalities.

Cities today face a challenge of how to integrate data from multiple, unrelated sources where the semantics of the data are imprecise, ambiguous and overlapping. This is especially true in a world where more and more data are being openly published by various organizations. A morass of data is increasingly becoming available to support city planning and operations activities. In order to be used effectively, it is necessary for the data to be unambiguously understood so that it can be correctly combined, avoiding data silos. Early successes in data “mash-ups” relied upon an independence assumption, where unrelated data sources were linked based solely on geospatial location, or a unique identifier for a person or organization. More sophisticated analytics projects that require the combination of datasets with overlapping semantics entail a significantly greater effort to transform data into something useable. It has become increasingly clear that integrating separate datasets for this sort of analysis requires an attention to the semantics of the underlying attributes and their values.

A common data model enables city software applications to share information, plan, coordinate and execute city tasks, and support decision making within and across city services, by providing a precise, unambiguous representation of information and knowledge commonly shared across city services. This requires a clear understanding of the terms used in defining the data, as well as how they relate to one another. This requirement goes beyond syntactic integration (e.g. common data types and protocols), it requires semantic integration: a consistent, shared understanding of the meaning of information.

To motivate the need for a standard city data model, consider the evolution of cities. Cities deliver physical and social services that have traditionally operated as silos. If during the process of becoming smarter, transportation, social services, utilities, etc. were to develop their own data models, the result would be smarter silos. To create truly smart cities, data needs to be shared across these silos. This can only be accomplished through the use of a common data model. For example, “Household” is a category of data that is commonly used by city services. Members of Households are the source of transportation, housing, education and recreation demand. This category represents who occupies a home, their age, their occupations, where they work, their abilities, etc. Though each city service can potentially gather and/or use different aspects of a Household, much of the data needs to be shared with each other.

Supporting this interoperability among city datasets is particularly challenging due to the diversity of the domain, the heterogeneity of its data sources, and data privacy concerns and regulations. The purpose of this document is to support the precise and unambiguous specification of city data using the technology of ontologies[1],[2] as implemented in the Semantic Web[3]. By doing so it will:

— enable the computer representation of precise definitions, thereby reducing the ambiguity of interpretation;

— remove the independence assumption, thereby allowing the world of Big Data, open source software, mobile apps, etc., to be applied for more sophisticated analysis;

— achieve semantic interoperability, namely the ability to access, understand, merge and use data available from datasets spread across the Semantic Web;

— enable the publishing of city data using Semantic Web and ontology standards, and

— enable the automated detection of city data inconsistency, and the root causes of variations.

With a clear semantics for the terminology, it is possible to perform consistency analysis, and thereby validate the correct use of the document.

Figure 1 identifies the three levels of the ISO/IEC 5087 series. The lowest level, defined in ISO/IEC 5087-1 (this document) provides the classes, properties and logical computational definitions for representing the concepts that are foundational to representing any data. The middle level, defined in ISO/IEC 5087-2:—[1], will provide the classes, properties and logical computational definitions for representing concepts common to all cities and their services but not specific to any service. The top level provides the classes, properties and logical computational definitions for representing service domain specific concepts that are used by other services across the city. For example, ISO/IEC TS 5087-3:—[2], will define the transportation concepts. In the future, additional parts will be added to the ISO/IEC 5087 series covering further services such as education, water, sanitation, energy, etc.

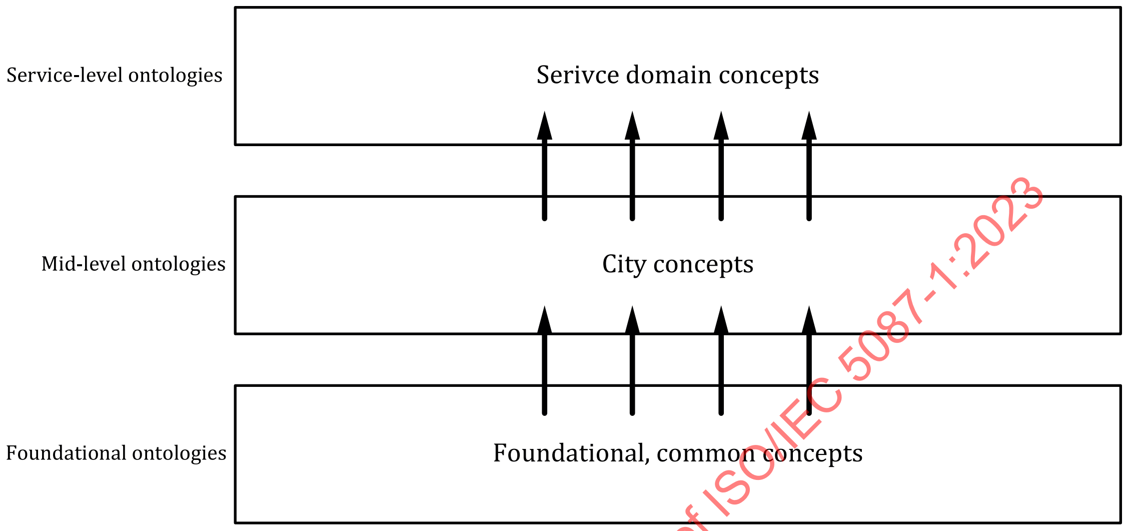


**Figure 1 — Stratification of city data model**

Figure 2 depicts example concepts for the three levels.

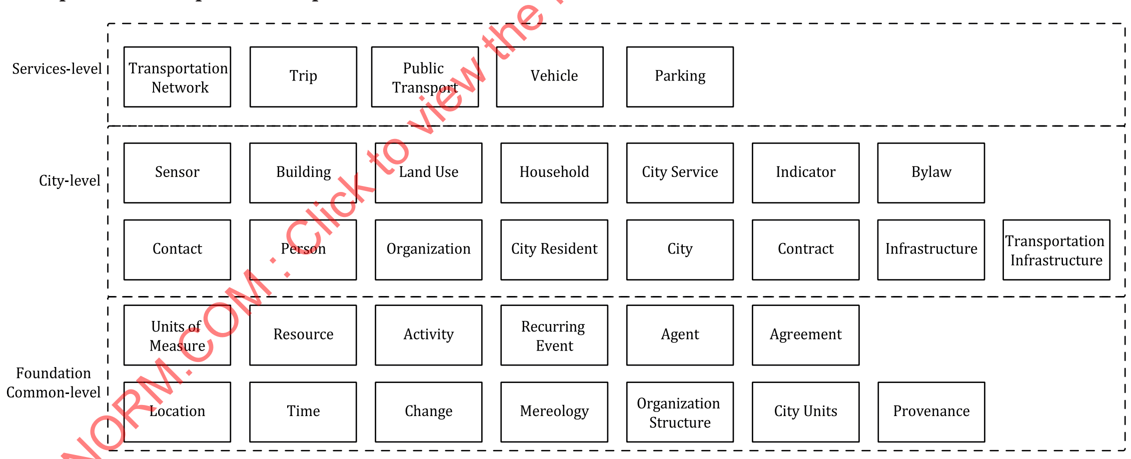


**Figure 2 — Example concepts for each level**

It is important to distinguish between the ISO/IEC 5087 series and the related, but distinct effort of ISO/IEC 30145-2. As specified in its Scope, ISO/IEC 30145-2:2020 "*specifies a generic knowledge management framework for a smart city, focusing on creating, capturing, sharing, using and managing smart city knowledge. It also gives the key practices which are required to be implemented to safeguard the use of knowledge, such as interoperability of heterogeneous data and governance of multi-sources services within a smart city.*" Figure 3 depicts the smart city knowledge management framework as described in ISO/IEC 30145-2. The smart city domain knowledge model includes a (cross-domain) core concept model and several domain knowledge models. This document defines the foundation level of the core concept model. ISO/IEC 5087-2 is intended to address some of the core concept model and cuts across

1) Under preparation. Stage at the time of publication: ISO/IEC DIS 5087-2:2023.

2) Under preparation.

the domain knowledge models. There is a possibility that subsequent parts of the ISO/IEC 5087 series (not yet defined) will define knowledge models for the services of citizen livelihood, urban management and smart transportation illustrated in the Figure 3.

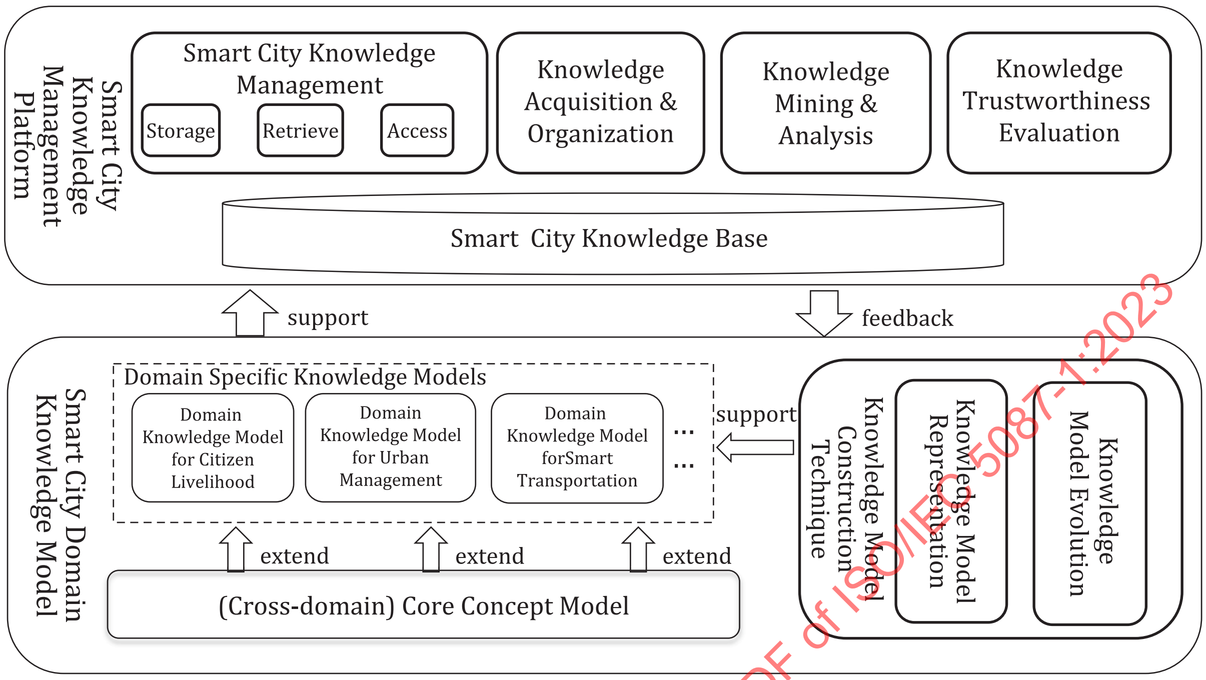

Smart City Knowledge Management Platform
Smart City Knowledge Management
Storage
Retrieve
Access
Knowledge Acquisition & Organization
Knowledge Mining & Analysis
Knowledge Trustworthiness Evaluation
Smart City Knowledge Base
support
feedback
Smart City Domain Knowledge Model
Domain Specific Knowledge Models
Domain Knowledge Model for Citizen Livelihood
Domain Knowledge Model for Urban Management
Domain Knowledge Model forSmart Transportation
...
support
extend
extend
extend
(Cross-domain) Core Concept Model
Knowledge Model Construction Technique
Knowledge Model Representation
Knowledge Model Evolution

**Figure 3 — The framework of smart city knowledge management from ISO/IEC 30145-2:2020**

There are other existing standards that overlap conceptually with some of the terms presented in this document. The relationship between ISO/IEC 5087-1 and existing documents that address similar or related concepts is identified in Annex A.

# Information technology — City data model —

# Part 1: Foundation level concepts

## 1 Scope

This document is part of the ISO/IEC 5087 series, which specifies a common data model for cities. This document specifies the foundation level concepts.

## 2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 21972, *Information technology — Upper level ontology for smart city indicators*

OGC GeoSPARQL, A Geographic Query Language for RDF Data, OGC 11-052r4, Open Geospatial Consortium, 10 September 2012. https://www.ogc.org/standards/geosparql

The Ontology in OWL, W3C Candidate Recommendation 26 March 2020, https://www.w3.org/TR/owl-time/

PROV-O. The PROV Ontology, W3C Recommendation 30 April 2013, https://www.w3.org/TR/prov-o/

The Organization Ontology, W3C Recommendation 16 January 2016, https://www.w3.org/TR/vocaborg/

## 3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

ISO and IEC maintain terminology databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at https://www.electropedia.org/

**3.1**
**cardinality**
number of elements in a set

[SOURCE: ISO/TS 21526:2019, 3.11]

**3.2**
**description logic**
**DL**
family of formal knowledge representation languages that are more expressive than propositional logic but less expressive than first-order logic

[SOURCE: ISO/IEC 21972:2020, 3.2]

**3.3**
**manchester syntax**
compact, human readable syntax for expressing Description Logic descriptions

[SOURCE: https://www.w3.org/TR/owl2-manchester-syntax/ (Copyright © 2012. World Wide Web Consortium. https://www.w3.org/Consortium/Legal/2023/doc-license).]

**3.4**
**measure**
value of the measurement (via the numerical_value property) which is linked to both Quantity and Unit_of_measure

[SOURCE: ISO/IEC 21972:2020, 3.4]

**3.5**
**namespace**
collection of names, identified by a URI reference, that are used in XML documents as element names and attribute names

Note 1 to entry: Names may also be identified by an IRI reference.

[SOURCE: ISO/IEC 21972:2020, 3.5, modified — Note 1 to entry has been added.]

**3.6**
**ontology**
formal representation of phenomena of a universe of discourse with an underlying vocabulary including definitions and axioms that make the intended meaning explicit and describe phenomena and their interrelationships

[SOURCE: ISO 19101-1:2014, 4.1.26]

**3.7**
**ontology web language**
ontology language for the Semantic Web with formally defined meaning

Note 1 to entry: OWL 2 ontologies provide classes, properties, individuals, and data values and are stored as Semantic Web documents.

[SOURCE: https://www.w3.org/TR/owl2-overview/ (Copyright © 2012. World Wide Web Consortium. https://www.w3.org/Consortium/Legal/2023/doc-license).]

**3.8**
**quantity**
property of a phenomenon, body, or substance, where the property has a magnitude that can be expressed by means of a number and a reference

Note 1 to entry: Quantities can appear as base quantities or derived quantities.

EXAMPLE 1 Length, mass, electric current (ISQ base quantities).

EXAMPLE 2 Plane angle, force, power (derived quantities).

[SOURCE: ISO 80000-1:2009, 3.1, modified — NOTEs 1 to 6 have been removed; new Note 1 to entry and two EXAMPLEs have been added.]

**3.9**
**Semantic Web**
W3C's vision of the Web of linked data

Note 1 to entry: Semantic Web technologies enable people to create data stores on the Web, build vocabularies, and write rules for handling data. The goal is to make data on the Web machine-readable and more precise.

[SOURCE: https://www.w3.org/standards/semanticweb/(Copyright © 2015. World Wide Web Consortium. https://www.w3.org/Consortium/Legal/2023/doc-license).]

**3.10**
**unit_of_measure**
definite magnitude of a quantity, defined and adopted by convention and/or by law

[SOURCE: ISO/IEC 21972:2020, 3.9]

## 4 Abbreviated terms and namespaces

| | |
|---|---|
| DL | description logic |
| OWL | ontology web language |
| RDF | resource description framework |
| RDFS | resource description framework schema |
| IRI | international resource identifier |

The following namespace prefixes are used in this document:

— activity: https://standards.iso.org/iso-iec/5087/-1/ed-1/en/ontology/Activity/

— agent: https://standards.iso.org/iso-iec/5087/-1/ed-1/en/ontology/Agent/

— agreement: https://standards.iso.org/iso-iec/5087/-1/ed-1/en/ontology/Agreement/

— change: https://standards.iso.org/iso-iec/5087/-1/ed-1/en/ontology/Change/

— cityunits: https://standards.iso.org/iso-iec/5087/-1/ed-1/en/ontology/CityUnits/

— genprop: https://standards.iso.org/iso-iec/5087/-1/ed-1/en/ontology/GenericProperties/

— geo: http://www.opengis.net/ont/geosparql#

— i72: http://ontology.eil.utoronto.ca/5087/2/iso21972/

— loc: https://standards.iso.org/iso-iec/5087/-1/ed-1/en/ontology/SpatialLoc/

— mereology: https://standards.iso.org/iso-iec/5087/-1/ed-1/en/ontology/Mereology/

— org: http://www.w3c.org/ns/org#

— org_s: https://standards.iso.org/iso-iec/5087/-1/ed-1/en/ontology/OrganizationStructure/

— owl: http://www.w3.org/2002/07/owl#

— partwhole: https://standards.iso.org/iso-iec/5087/-1/ed-1/en/ontology/Mereology/

— prov: http://www.w3.org/ns/prov-o#

— 5087prov: https://standards.iso.org/iso-iec/5087/-1/ed-1/en/ontology/Prov/

— rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#

— rdfs: http://www.w3.org/2000/01/rdf-schema#

— recurringevent: https://standards.iso.org/iso-iec/5087/-1/ed-1/en/ontology/RecurringEvent/

— resource: https://standards.iso.org/iso-iec/5087/-1/ed-1/en/ontology/Resource/

— time: http://www.w3.org/2006/time#

— xsd: http://www.w3.org/2001/XMLSchema#

The formalization of the classes in this document is specified using the following table format, which is a simplification of description logic (DL) where the first column identifies the class name, the second column its properties (a class is defined as the subclass of all of its properties) and the third column each property's range restriction. It shall be read as: The <Class> is a subClassOf the conjunction of the associated <property>s with their <value>s. Range restrictions are specified using the Manchester syntax. For example, Table 1 specifies that Agent is a subclass of the intersection of genprop:hasName exactly 1 xsd:string and resource:resourceOf only resource:TerminalResourceState and performs only activity:Activity.

**Table 1 — Example class formalization**

| Class | Property | Value restriction |
|---|---|---|
| Agent | genprop:hasName | exactly 1 xsd:string |
| | resource:resourceOf | only resource:TerminalResourceState |
| | performs | only activity:Activity |

The following value restrictions are used in this document:

— "min n": Specifies that the property has to have a minimum n values.

— "max n": Specifies that the property has to have a maximum n values.

— "exactly n": Specifies that the property has to have exactly n values.

— "only": Specifies that the values of the property can only be an instance/type of the class specified, e.g., a string, integer or another class such as Organization.

CamelCase is used for specifying classes, properties and instances. For example, "legalName" instead of "legal_name". The first letter of a class name is capitalized. The first letter of a property and instance name are not capitalized. An instance of a class shall satisfy the class's definition. The instance's properties and values shall satisfy the value restrictions of the class it is an instance of.

The formalization of the properties in this document is done similarly, using the following table format that allows for the identification of properties and their sub-properties, inverse properties, or other characteristics. It is to be read as: The <property> is <characteristic> of <value>, or simply the <property> is <characteristic> if no value is applicable. For example, in Table 2 hasPrivilege is a sub-property of the agentInvolvedIn property. Characteristics are specified using the Manchester syntax.

**Table 2 — Example property formalization**

| Property | Characteristic | Value (if applicable) |
|---|---|---|
| hasPrivilege | rdfs:subPropertyOf | agentInvolvedIn |
| | Irreflexive | |

In the case of DL definitions of classes where the simplified table representation is insufficient, the DL specification will be supplied.

The patterns defined in this document have also been implemented in OWL and made available online. The location of these encodings is identified in Annex D.

## 5 General

### 5.1 Unique identifiers

All classes, properties and instances of classes have a unique identifier that conforms to Linked Data/ Semantic Web standards. The unique identifier is an IRI. When using ISO/IEC 5087-1 (this document) in an application, a class is identified by the IRI for the pattern of which it is a member, followed by the class name. In the Agent example in Clause 4, the Agent class's unique identifier would be:

https://standards.iso.org/iso-iec/5087/-1/ed-1/en/ontology/Agent/Agent

Breaking the IRI down:

— “5087” identifies the series number

— “-1” identifies the part number

— “ed-1” indicates that the class is defined in edition 1 of the standard

— “en” indicates that the class is defined in a pattern implemented in English

— The first “Agent” identifies the Agent pattern

— The second “Agent” identifies the Agent class within the Agent pattern

The IRI can be shortened using the prefix’s defined in Clause 4:

agent:Agent

where agent: is the prefix for the Agent pattern.

Properties are identified in the same manner. The IRIs of individuals created by an application of ISO/IEC 5087-1 would have IRIs unique to the application.

### 5.2 Reference to existing patterns

The practice of reusing and referencing existing standard vocabularies is an important practice in the context of the Semantic Web. It is important that existing standard patterns are included as normative references where appropriate, rather than duplicating and inserting the appropriate content as a pattern within this document. The use of shared vocabularies directly enables interoperability and shareability between implementations and avoids the additional work of attempting to map to these standards after the fact. Where possible, existing standardized ontologies have been included as normative references to specify the patterns below. In cases where an extension is required, this is done in such a way as to preserve the content of the normative reference in order to support interoperability and make the relationship transparent.

## 6 Foundational ontologies

### 6.1 General

Beyond the domain-specific subjects that are clearly identified in consideration of the requirements, there are fundamental concepts that are necessary to formulate an accurate definition of the domain. These concepts are defined in a series of foundational ontologies, so-named because they provide a reusable foundation for the development of other ontologies for city services. The clear definition and uncoupling of the foundational concepts make the fundamental commitments of the city data model clear and accessible to potential adopters. It also ensures interoperability and consistency in the representation of key concepts such as time and location. The city data model defines eleven foundational patterns to capture these concepts. A pattern is a set of concepts that are related by topic and inter-connected by properties, thereby forming a graph. A foundational pattern is a pattern composed of a set of foundational concepts. These are described in the following subclauses.

### 6.2 Generic properties

#### 6.2.1 General

Most of the properties are identified and defined relative to a certain Class and in the context of a particular pattern in the following subclauses. However, there are certain exceptions where generic properties can be recognized as applicable to a wide range of classes with no common theme amongst them. Such properties are defined separately as generic properties. This allows for the reference to these properties independent of any particular pattern. These generic properties are imported by all of the patterns defined in the ISO/IEC 5087 series.

### 6.2.2 Key Properties

The following generic properties have been identified:

— **hasName**: identifies the name of a certain object;

— **hasDescription**: specifies a description of a certain object;

— **hasIdentifier**: specifies an identifier for a certain object.

## 6.3 Mereology pattern

### 6.3.1 General

Notions of parthood are ubiquitous. While sometimes conflated, there are clear distinctions which can be made between different types of parthood. The Mereology pattern focuses on identifying these differences and making them explicit. The distinction between types of parthood may be best explained with the use of examples. An item may be *contained in* a car, but that does not make it a *component of* a car. For example, there may be a need to describe passengers or cargo being *contained in* a vehicle, but this relation needs to be distinguished from the parts and components that make up a vehicle. Similarly, the front of a car is intuitively a part of the car, but not a component of the car. While components of a vehicle may be defined, different city zone systems (wards, postal codes) are not components, but proper parts of larger areas.

### 6.3.2 Key classes and properties

They key properties are formalized in Table 3. The Mereology pattern identifies the following different types of parthood: proper-part-of and component-of. A more detailed analysis, presented in Reference [10] reveals clear, ontological distinctions between these relations (as well as a containment relation) that may formalized clearly with a set of first-order logic axioms. The different properties may be described as follows:

— **partOf**: specifies a part-whole relationship between objects

— **properPartOf**: specifies a part-whole relationship between objects where an object cannot be part of itself

— **componentOf**: specifies a part-whole relationship between objects where the part is defined based on actual boundaries. The parts are often also defined according to distinct functions. For example, a trunk is a componentOf a car.

— **immediateComponentOf**: specifies a componentOf relationship where the if x immediateComponentOf y, then there does not exist a z where x immediateComponentOf z immediateComponentOf y.

The aforementioned analysis (presented in Reference [10]) also identifies the expressive limitations of OWL, which prevent a complete representation of this semantics, and discussed the various possible approximations. It is important to consider what should be captured, and what distinctions should be made in the introduction of properties, in contrast with what is actually expressible in the logic. Since the required semantics cannot be completely captured in OWL, some trade-off(s) is required for any partial specification, (e.g. OWL only allows the specification of transitivity for simple object properties).

The difficulty with such an approximation is that the resulting theory defines a semantics for something else entirely. Inherently, some semantics are omitted, which can potentially not be required for one application but can potentially be important for another. For example, if transitivity is a key aspect of some required reasoning, then perhaps a parthood relation would be defined as transitive, and some omissions would be made with respect to the formalization of other restrictions (e.g. cardinality) that should be applied to the parthood relation. Certainly, the use of approximations will be required in some cases, for example in order to support some desired reasoning problems. However, precisely which axiomatization is most suitable will vary between different usage scenarios. The Mereology pattern therefore omits a detailed, partial axiomatization in favour of an under-axiomatized specification of the key relations, in order to avoid prescribing one trade-off over another. This leaves the commitment open-ended and variable to suit individual applications' needs.

This ontology defines the general properties such that the commonality between domain-specific part-of relations may be captured, and more detailed semantics may be defined in extensions of the properties. This creates a means of indicating the intended semantics of a relation by identifying the type of parthood that it is intended to capture, while allowing for the specification of different partial approximations of the semantics (and possibly also specializations of this semantics), as required. For example, a notion of parthood arises in the description of a building and the units it is divided into. In this case, this relationship can be identified as a sort of hasComponent relation; a new property hasBuildingUnit can be identified then as a subPropertyOf hasComponent. The most suitable approximation of the component-of relation can then be defined for the hasBuildingUnit relation. The approximation chosen for one type of parthood relation does not constrain the choice of approximation for another.

Figure 4 illustrates the use of the properties defined in the Mereology pattern to serve as generic parthood properties. In this example, the hasComponent property is made more specific with the hasBuildingUnit subObjectProperty defined between Building and BuildingUnit classes.

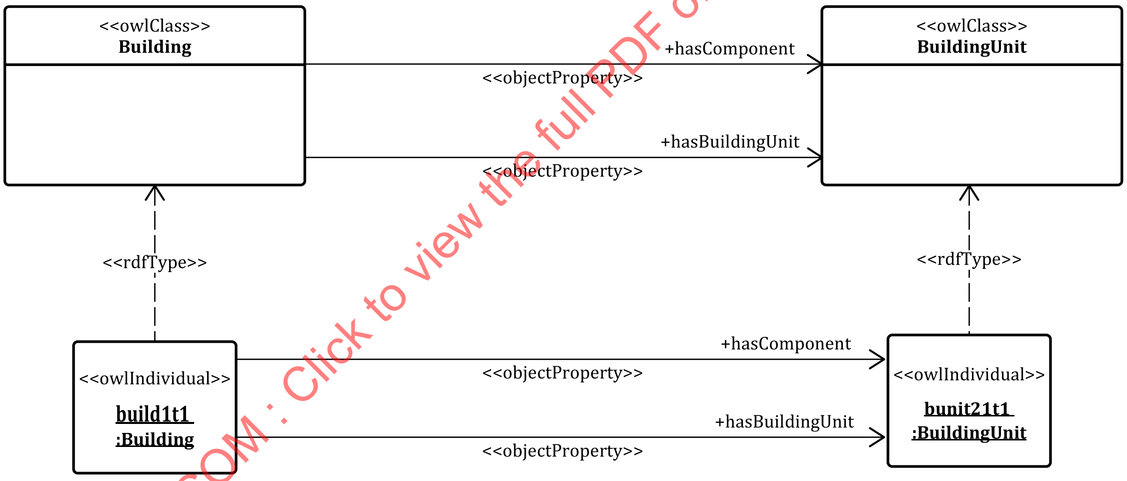


Figure 4 — Example use of the Mereology pattern

### 6.3.3 Formalization

Table 3 — Key properties in the Mereology pattern

| Property | Characteristic | Value restriction (if applicable) |
|---|---|---|
| partOf | | |
| properPartOf | inverseOf | hasProperPart |
| hasProperPart | inverseOf | properPartOf |
| componentOf | rdfs:subPropertyOf | properPartOf |
| | inverseOf | hasComponent |
| hasComponent | rdfs:subPropertyOf | hasProperPart |
| | inverseOf | componentOf |
| immediateComponentOf | rdfs:subPropertyOf | componentOf |

## 6.4 City Units Pattern

### 6.4.1 General

Units of measure are an important concept due to the observational nature of city data collection. It is particularly important to capture the relationship between some quantity and the unit of measure it is described with. This allows for a representation in which the same individual quantity may be associated with several values, according to different units of measurement.

The representation of quantities and measures information shall conform to the ontology specified in ISO/IEC 21972. ISO/IEC 21972 is a standard that defines classes and properties required for a foundational representation of indicators, of which units are an integral part. It is included in its entirety with the prefix ‘i72’.

### 6.4.2 Key classes and properties

The City Units pattern provides a structured vocabulary to describe, among other things, the different values (measures) that are associated to given quantities. This allows for the provision of greater detail regarding specific measurements that are defined in the ontology.

This pattern extends ISO/IEC 21972 with the classes and properties outlined in Table 4 and Table 5, respectively, to include the wider scope of quantities required for city data, such as those pertaining to physical descriptions.

Figure 5 illustrates the use of the City Units Pattern to capture numerical values and their associated units. This example shows the representation of a vehicle's speed. The speed is a single object (“veh123s1t1”) that can be associated with multiple different values (e.g. 62 or 100) depending on the associated unit.

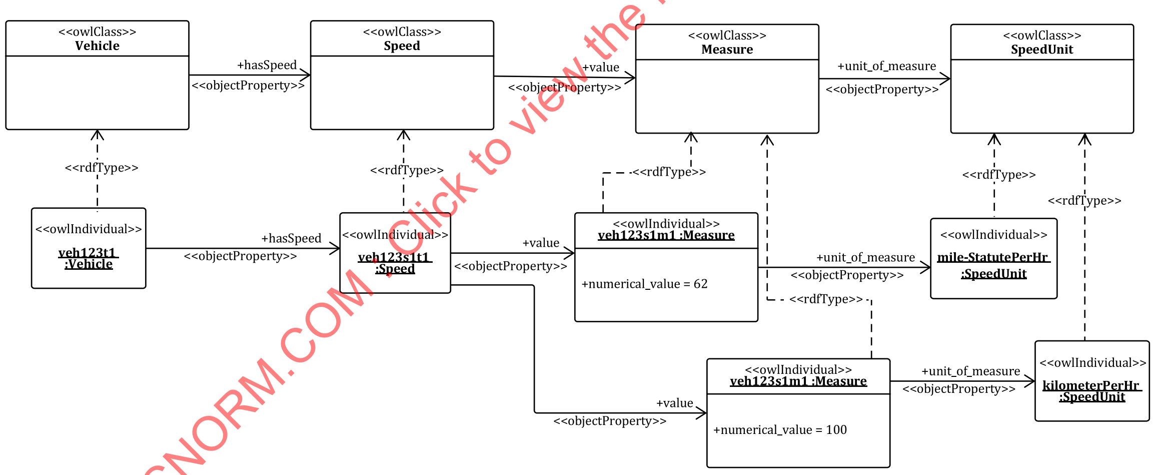

<<owlClass>> Vehicle
+hasSpeed <<objectProperty>>
<<owlClass>> Speed
+value <<objectProperty>>
<<owlClass>> Measure
+unit_of_measure <<objectProperty>>
<<owlClass>> SpeedUnit
<<rdfType>>
<<rdfType>>
<<rdfType>>
<<rdfType>>
<<rdfType>>
<<owlIndividual>> veh123t1 :Vehicle
+hasSpeed <<objectProperty>>
<<owlIndividual>> veh123s1t1 :Speed
+value <<objectProperty>>
<<owlIndividual>> veh123s1m1 :Measure
+numerical_value = 62
+unit_of_measure <<objectProperty>>
<<owlIndividual>> mile-StatutePerHr :SpeedUnit
<<rdfType>>
+value <<objectProperty>>
<<owlIndividual>> veh123s1m1 :Measure
+numerical_value = 100
+unit_of_measure <<objectProperty>>
<<owlIndividual>> kilometerPerHr :SpeedUnit

**Figure 5 — Example use of the City Units pattern**

Quantities, units, and/or measures that are defined using domain-specific concepts (e.g. vehicles, lanes) are defined by reusing and extending the City Units pattern in the relevant ontologies, such that the necessary concepts may be captured and the foundational ontology is not complicated with domain-specific concepts.

### 6.4.3 Formalization

**Table 4 — Key classes in the City Units pattern**

| Class | Property | Value restriction |
|---|---|---|
| Area | rdfs:subClassOf | i72:Quantity |
| | i72:unit_of_measure | only AreaUnit |
| AreaUnit | rdfs:subClassOf | i72:Unit_of_measure |
| Length | rdfs:subClassOf | i72:Quantity |
| | i72:unit_of_measure | only LengthUnit |
| LengthUnit | rdfs:subClassOf | i72:Unit_of_measure |
| Speed | rdfs:subClassOf | i72:Quantity |
| | i72:unit_of_measure | only SpeedUnit |
| SpeedUnit | rdfs:subClassOf | i72:Unit_of_measure |
| ValueOfMoney | rdfs:subClassOf | i72:Quantity |
| | rdfs:subClassOf | i72:AmountOfMoney |
| | i72:value | only MonetaryValue |
| MonetaryValue | rdfs:subClassOf | i72:Measure |
| | hasRelativeYear | exactly 1 xsd:gYear |
| | i72:unit_of_measure | only i72:Amount_of_money_unit |
| Cardinality_unit_per_time | rdfs:subClassOf | i72:UnitDivision |
| | i72:hasNumerator | only i72:Cardinality_unit |
| | i72:hasDenominator | only TimeUnit |
| TimeUnit | rdfs:subClassOf | i72:Unit_of_measure |
| average | rdf:type | Function |
| Duration | rdfs:subClassOf | i72:Quantity |
| | i72:value | only i72:Measure and (i72:hasUnit only TimeUnit) |
| | rdfs:subClassOf | time:TemporalDuration |
| Ratio | rdfs:subClassOf | i72:Quantity |
| i72:RatioIndicator | rdfs:subClassOf | Ratio |
| Volume | rdfs:subClassOf | i72:Quantity |
| | i72:value | only i72:Measure and (i72:hasUnit only VolumeUnit) |

**Table 5 — Key properties in the City Units pattern**

| Property | Characteristic | Value (if applicable) |
|---|---|---|
| hasAggregateFunction | rdfs:domain | i72:Quantity |
| | rdfs:range | i72:function |
| aggregateOf | rdfs:domain | i72:Quantity |
| aggregateOver | rdfs:domain | i72:Quantity |

## 6.5 Time Pattern

### 6.5.1 General

To define an ontology pattern for time, it is necessary to identify the objects of interest, i.e., which things will be described. In general, three approaches to a representation of time can be identified: point-based, interval-based, and mixed. In a point-based representation, the objects of interest are timepoints. The passing of time is described as an ordering over time points, and periods of time can be represented as a series of timepoints. In an interval-based representation the objects of interest are time intervals, whereas the mixed representation includes both timepoints and time intervals. Key to all of these representations is that there is an ordering that holds over these time objects. It is important to be able to describe whether one time object is before another; in the case of time intervals, it is also important to be able to describe other relationships, such as whether one interval is contained in or overlaps with another. The representation of time information shall conform to the ontology specified in the W3C Candidate Recommendation “Time Ontology in OWL”. It specifies a mixed representation that includes time points and intervals. It is included in its entirety with the prefix ‘time’.

## 6.6 Change pattern

### 6.6.1 General

Many of the concepts identified in the urban system ontologies are subject to change. For example, a Vehicle will have one location at one time, and another location at a later time; it can have only one passenger at one time, and four passengers at a later time. Similarly, many attributes of Persons, Households and even Transportation Networks are subject to change.

Change over time plays a role in many domains and thus the representation of change is by no means a new research topic. To this end, several approaches for capturing change in OWL have been proposed[11],[12]. Despite these solutions, Semantic Web practitioners lack clear and precise methods for how to apply these approaches to capture change at a domain level, whether reusing a temporal ontology or developing an ontology from scratch. The Change pattern serves as a clear guide to support a consistent approach to representing change over time.

### 6.6.2 Key classes and properties

The key classes and properties are formalized in Table 6 and Table 7, respectively. An approach to representing changing properties, or "fluents", that leverages the 4-dimensionalist perspective was proposed in Reference [11]. A similar approach is adopted in this document, based on the design pattern presented by Reference [13], requiring the representation of objects that are subject to change as subclasses of the Manifestation class. Manifestations may be interpreted as “snapshots” of an object at some point in time. This enables the representation of changing values of properties of an object, without losing information about its past values. The properties of the class can then be identified as properties that are (and are not) subject to change, in order to distinguish between the static and dynamic aspects of a particular entity.

The Manifestation class is defined using the following properties:

— **existsAt**: identifies the TemporalEntity that reflects the time Instant or Interval during which the snapshot of the object holds (is valid).

— **hasNextManifestation**: identifies the immediate successor Manifestation (i.e. the subsequent snapshot of the object).

— **hasPreviousManifestation**: identifies the immediate prior Manifestation (i.e. the prior snapshot of the object).

— **hasFirstManifestation**: identifies the First Manifestation that is related to a particular Manifestation (i.e., for some snapshot of an object, it identifies the first such snapshot of the object).

The class FirstManifestation denotes the first recorded manifestation of an individual. No prior manifestations (in time) exist for the individual. It is a subclassOf Manifestation and contains the following additional properties:

— **precedesManifestation**: identifies all subsequent Manifestations that follow a FirstManifestation. In other words, the FirstManifestation is related to each and every subsequent manifestation via the precedesManifestation.

— **hasLatestManifestation**: identifies the most recent Manifestation that is related to a particular FirstManifestation (i.e., for some snapshot of an object, it identifies the most recent snapshot of the object).

Manifestations sharing the same FirstManifestation, and often connected in a series (e.g. via the hasNextManifestation property), form a group of descriptions about a particular entity. Some applications may require the explicit representation this individual. In other words, there is a need to represent a single, distinct individual while also capturing its changes over time. The hasManifestation property is defined to support this requirement: a constant object class can be related to its counterpart subclass of Manifestation via the hasManifestation property. In an instantiation, a single instance of some object class could be associated to multiple instances of Manifestations, defining how the object changes over time. Note that this may also be achieved by representing the object as constant and reifying its variant properties as subclasses of Manifestation. Different representation choices can be selected as appropriate for different applications.

— **hasManifestation:** identifies a Manifestation that captures the instantiation of variant attributes of an object at some time (i.e., a temporal snapshot of some individual). The individual is related to all manifestations in a series via the hasManifestation property.

A key aspect of the representation of change is the task of distinguishing between invariant and variant properties. Invariant properties are those that are not expected to (or should not) change in the context of the application(s), whereas variant properties are expected to possibly change in the context of the application(s). This assessment will likely vary between applications, such that some properties may be considered invariant for some applications, but variant for others. In a broad sense, nearly all properties (except those associated with an object's identity) may be considered variant given a suitable perspective. While it is true that historical properties (e.g. a person's date of birth) are much less likely to change, even in this case it is plausible that some applications include the possibility of revision of prior knowledge. Similarly, the perception of a particular class as subject to change or not will vary between applications. Use of the Change pattern is an implementation decision that is left to the discretion of users of the model presented here. In other words, no classes in the ISO/IEC 5087 standard series are defined as subclasses of Manifestation. This provides the flexibility of representing only what is necessary (e.g. where it is important to track or enforce constraints on changing information) as a Manifestation, while maintaining some degree of interoperability with implementations that do not adopt the same decisions. The alternative, to require that a specific set of concepts (or all, or none) be defined as subclasses of Manifestation is impractical and would decrease the utility of the ISO/IEC 5087 standard series.

A generic approach can be adopted to extend the Change pattern to capture change as it occurs in specific classes throughout the city domain. This approach is a variation on the original logical design pattern outlined in detail in Reference [13] and can be summarized with the following sequence of steps, where the underlined terms indicate placeholders for domain-specific terms to be specified when implementing the design pattern. For each class, C, that is subject to change:

1) Distinguish between the *variant and invariant* properties of C i.e., identify which properties' values may change over time and which must remain the same for any instance of C, for as long as it exists.

2) Define the concept, C, as a subclass of Manifestation.

   **Axiom:** C subClassOf Manifestation

3) Annotate the class C by identifying any invariant properties using the hasInvariantProperty annotation to identify the property with its IRI.

   C hasInvariantProperty invariantProperty

The following example illustrates the approach in detail. Figure 6 illustrates a simple representation of objects that does not account for changes over time. The veh_123 object is an instance of Vehicle that represents an individual vehicle. It has a model, model_x, an owner, alice, and a location, represented by the object loc_1. No changes to veh_12 are expressible with this representation. For example, any change to the associated location (via the hasLocation property) would be captured with a rewrite of the statement about veh_123. In this approach it is not possible to state that the value was loc_1 and is now loc_2. This representation supports a single snapshot of objects in its domain.

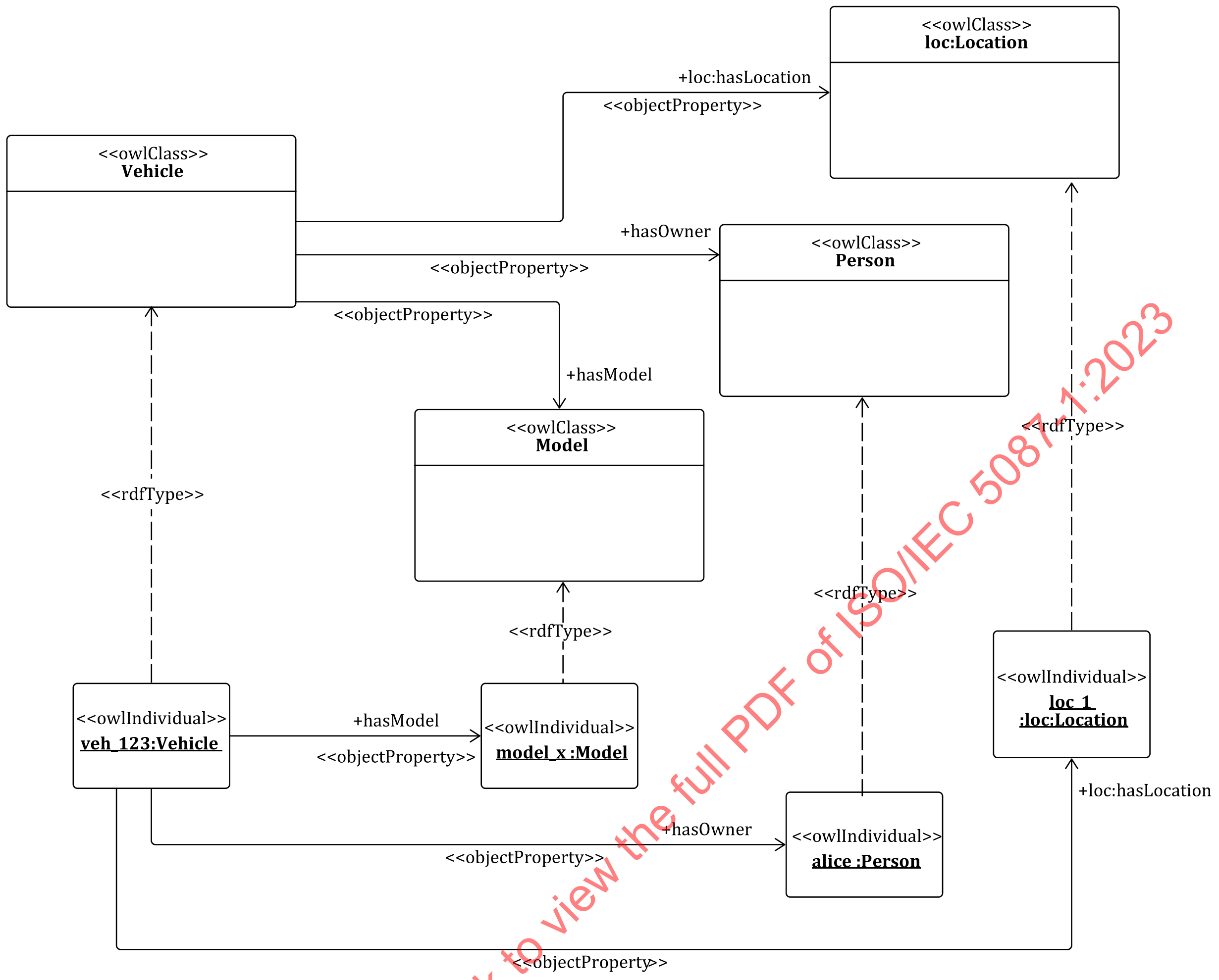
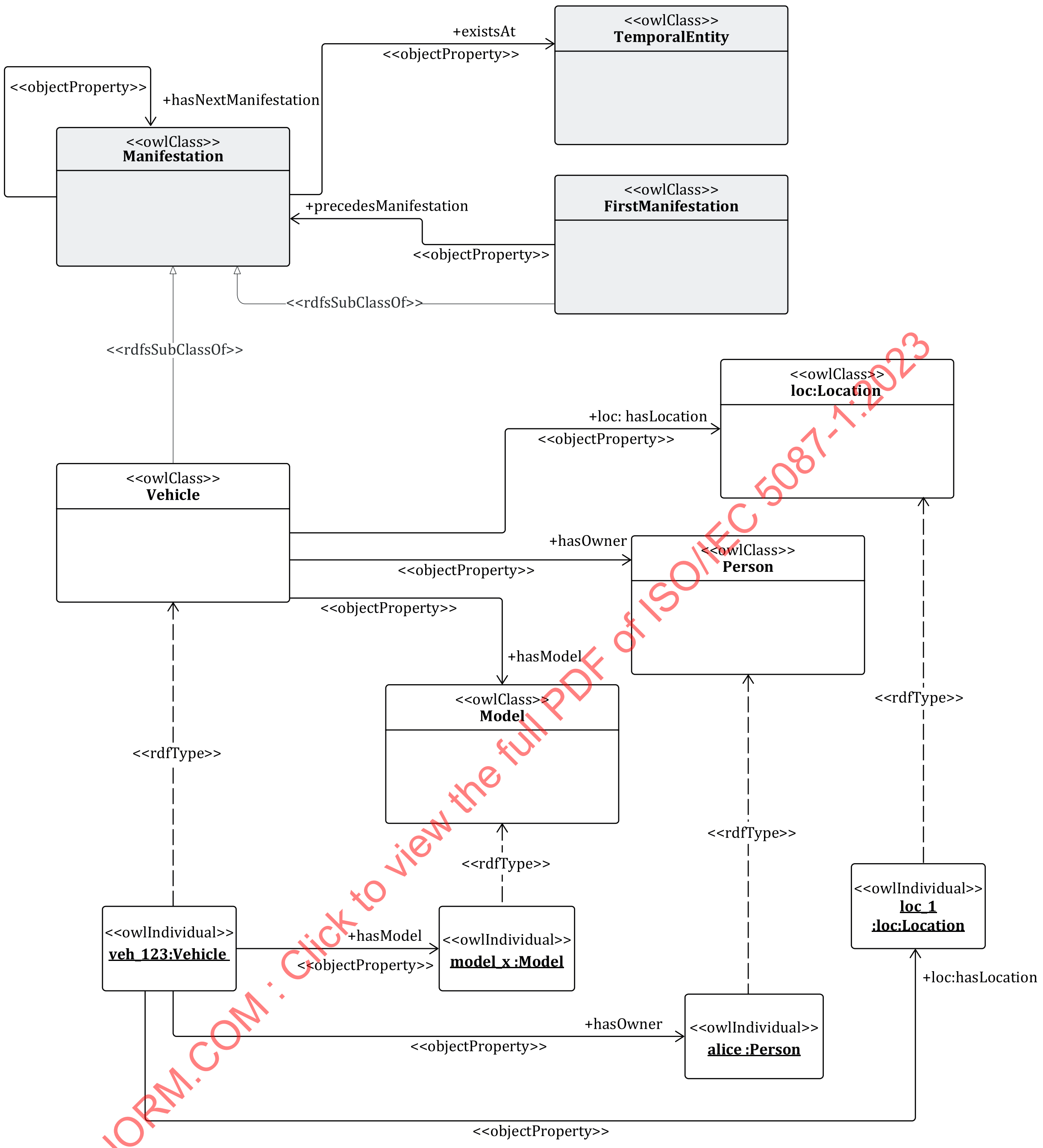


**Figure 6 — An atemporal representation**

The default (atemporal) approach exemplified by the Vehicle class may be adapted to a temporal approach (one that represents objects potentially changing over time) by introducing a relationship with the classes Manifestation, FirstManifestation, and TemporalEntity) as follows:

**Step 1:** Introduce classes Manifestation and FirstManifestation. The class that will have a temporal representation (possibly changing properties) should be defined as a subclass of Manifestation.

In this case (Figure 7), Vehicle is defined as a subclass of Manifestation. This means that any instances of Vehicle are now interpreted as snapshots of an object in time. Several different instances of Vehicle can refer to the same object at different points in time. For example, each instance Vehicle may have different values of a data property, such as an odometer reading or object property such as hasLocation, but may still maintain the same object relationships values, such as owner. Each instance is fully interconnected with other data being represented beyond the classes defined in the pattern. In addition, it is not necessary that the value of an object property be a Manifestation. For example, the value of hasLocation, loc_1, can be a simple instance of Location where change management is not applied. Note that Manifestations that share the same FirstManifestation form a group of Manifestations that describe the same object. So, in the example given, there can (and likely will) be multiple such collections of instances of Vehicle, each describing the changes of a different object.



<<owlClass>> TemporalEntity
+existsAt <<objectProperty>>
<<objectProperty>> +hasNextManifestation
<<owlClass>> Manifestation
<<owlClass>> FirstManifestation
+precedesManifestation <<objectProperty>>
<<rdfsSubClassOf>>
<<rdfsSubClassOf>>
<<owlClass>> loc:Location
+loc: hasLocation <<objectProperty>>
<<owlClass>> Vehicle
+hasOwner <<objectProperty>>
<<owlClass>> Person
<<objectProperty>> +hasModel
<<owlClass>> Model
<<rdfType>>
<<rdfType>>
<<rdfType>>
<<rdfType>>
<<owlIndividual>> loc_1 :loc:Location
+loc:hasLocation
<<owlIndividual>> veh_123:Vehicle
+hasModel <<objectProperty>>
<<owlIndividual>> model_x :Model
+hasOwner <<objectProperty>>
<<owlIndividual>> alice :Person
<<objectProperty>>

**Figure 7 — Introducing the Change pattern**

Next it is necessary to distinguish between invariant and variant properties. A consequence of Step 1 is that all of the property constraints that were associated with Vehicle are now interpreted as "time variant". In other words, the restrictions only apply at instants/intervals in time (to individual snapshots), so the values of these properties may change over time. In this example, the model of a Vehicle is invariant whereas the other two properties are not. It does not matter that the location will likely vary with much greater frequency than the owner of the vehicle (which might not vary at all). Both are interpreted as invariant properties, and any snapshot of the vehicle will allow for changes to either. The property hasModel will be annotated in the Vehicle class as an invariant property.

**Step 2:** distinguish between invariant and variant properties for the class that will have a temporal representation. At minimum, this shall be done by annotating the invariant property with the annotation hasInvariantProperty with a value of hasModel.

Existing instances of Vehicle are interpreted as representing the state of a Vehicle at a given Instant or Interval in time. Therefore, as shown in Figure 8, an additional assertion should be made as to the TemporalEntity at which each Vehicle individual exists.

Now that Vehicle is a subclass of Manifestation, each instance of Vehicle inherits the property existsAt. A temporal value, either an Instant or Interval, can be specified for which the individual is a temporal snapshot.

**Step 3:** assert an applicable TemporalEntity for which the instance is valid, for the existsAt property. Implicitly, all existing instances of Vehicle are now the first manifestation in a possible series of changing snapshots.

At this point all instances of Vehicle represent separate vehicles, and are the first manifestation of the instance. To reflect this, each of the instances are made to have the type FirstManifestation to reflect that they are the first temporal snapshot of what are to be many snapshots for each instance.

**Step 4:** assert all instances to be of type FirstManifestation (Figure 8).

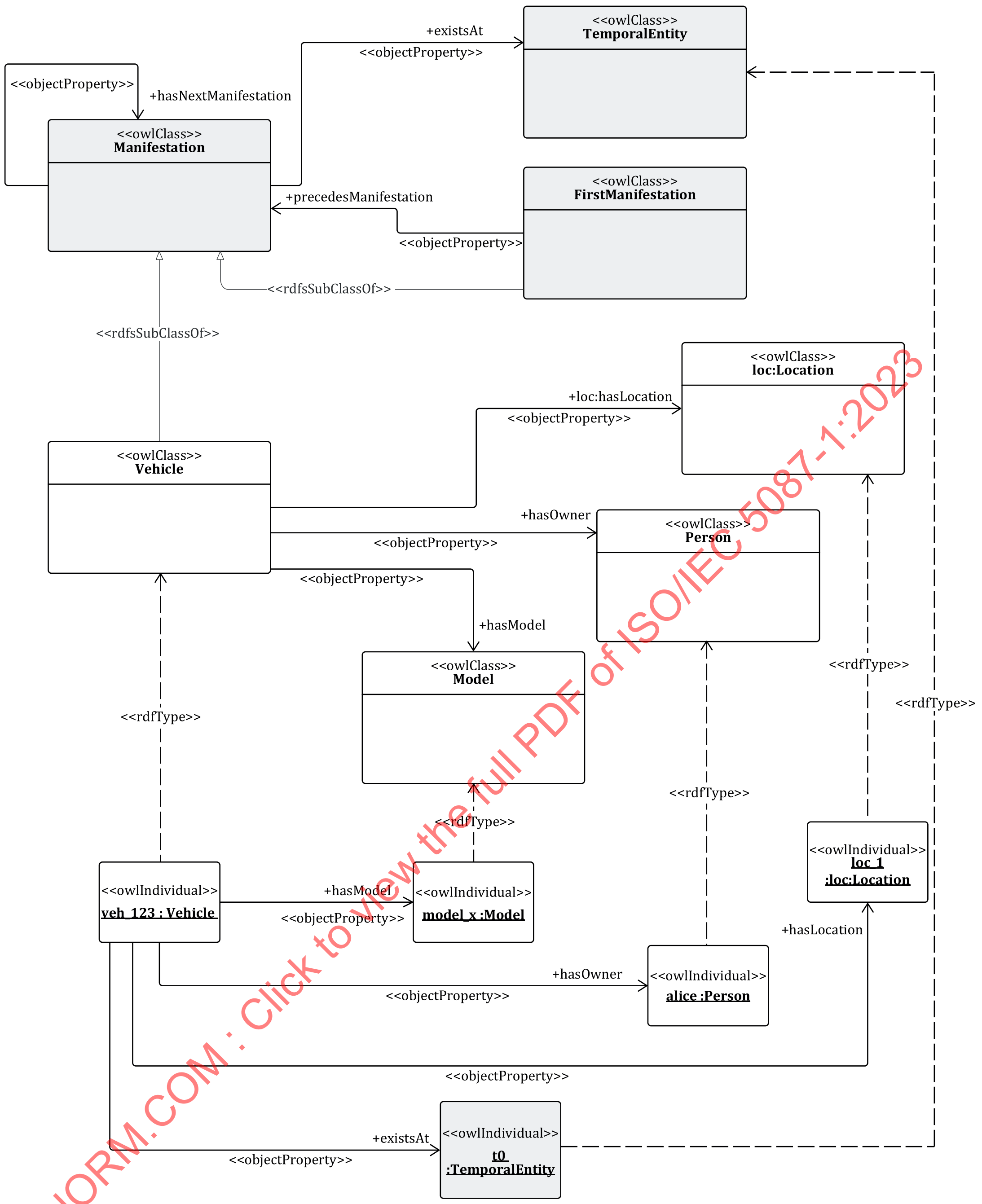


**Figure 8 — Adapting existing instances**

With these revisions complete, subsequent instances may be asserted to describe changes that occur to a particular object over time. Each new snapshot of an instance, i.e. manifestation, will also be associated with a TemporalEntity. In addition, the first instance (manifestation) will be associated with all future instances via the precedesManifestation property. All instances will be related to any subsequent instance with the hasNextManifestation property. Examples of the assertion of subsequent instances are illustrated in Figure 9.

**Step 5:** The initial, FirstManifestation instance will be related to all subsequent manifestations of an object via the precedesManifestation property.

**Step 6:** Each manifestation will be related to its following manifestation (i.e. the instance representing the next change to the object) via the hasNextManifestation property.

**Step 7:** Each manifestation will be related to the first manifestation via the hasFirstManifestation property

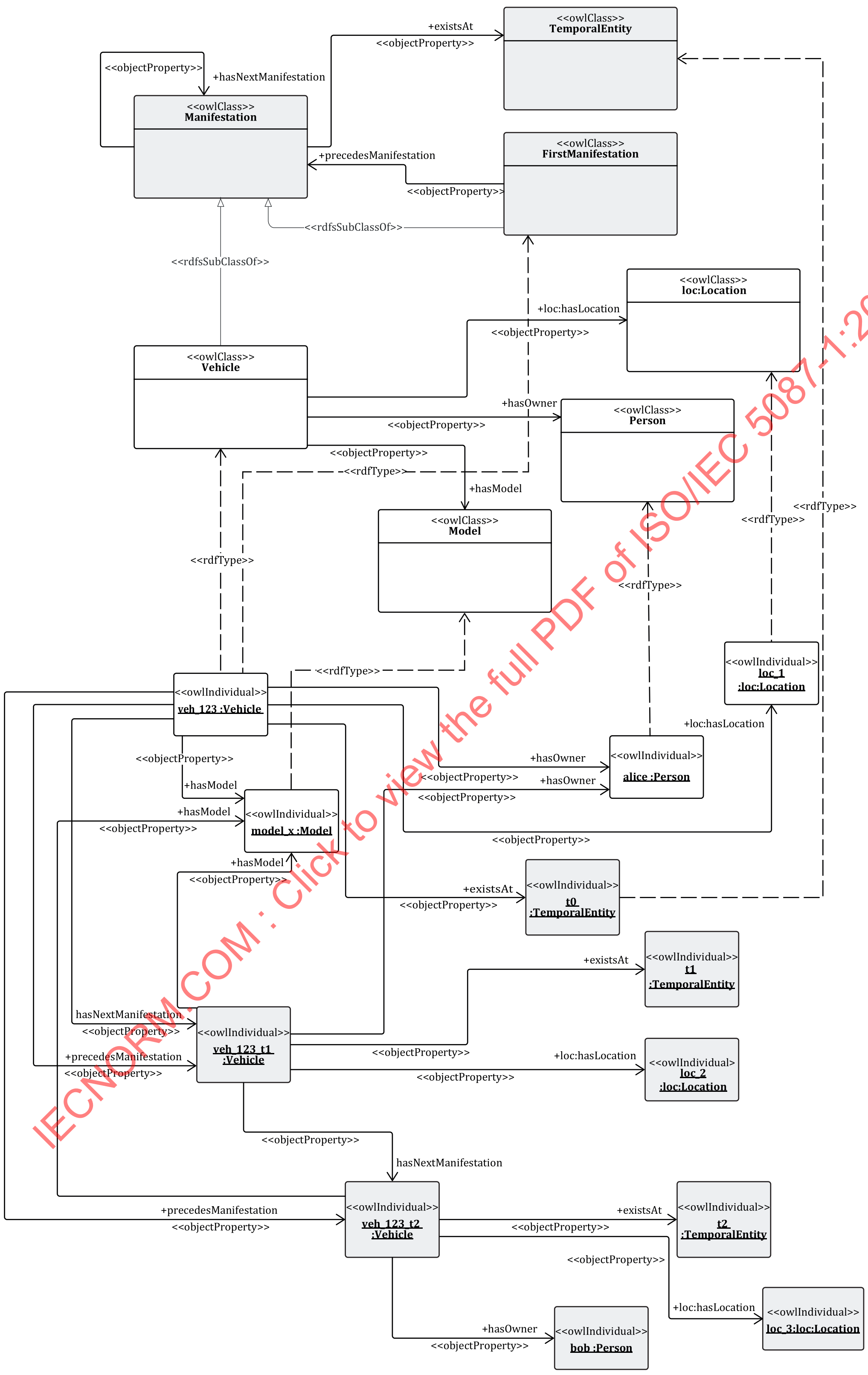


**Figure 9 — Asserting new instances**

It may be desirable to support inferences regarding the inheritance of invariant properties. In such cases, rules languages (formalisms outside the OWL standard) may be specified. Alternative approaches to formalizing this semantics for implementations are described in Annex A.

### 6.6.3 Formalization

**Table 6 — Key classes in the Change pattern**

| Class | Property | Value restriction |
|---|---|---|
| Manifestation | existsAt | exactly 1 time:TemporalEntity |
| | hasNextManifestation | max 1 Manifestation |
| | hasPreviousManifestation | max 1 Manifestation |
| | hasFirstManifestation | exactly 1 Manifestation |
| FirstManifestation | rdfs:subClassOf | Manifestation |
| | precedesManifestation | only Manifestation and not (FirstManifes-tation) |
| | hasLatestManifestation | exactly 1 Manifestation |
| | Inverse(hasManifestation) | exactly 1 owl:Thing |

**Table 7 — Key properties in the Change pattern**

| Property | Characteristic | Value restriction (if applicable) |
|---|---|---|
| existsAt | rdfs:range | time:TemporalEntity |
| hasNextManifestation | rdfs:domain | Manifestation |
| | rdfs:range | Manifestation |
| hasPreviousManifestation | inverseOf | hasNextManifestation |
| precedesManifestation | rdfs:domain | FirstManifestation |
| | rdfs:range | Manifestation and not (FirstManifestation) |
| hasFirstManifestation | inverseOf | precedesManifestation |
| hasLatestManifestation | rdfs:domain | FirstManifestation |
| | rdfs:range | Manifestation |
| hasManifestation | rdfs:range | Manifestation |

## 6.7 Location pattern

### 6.7.1 General

The ontology for representing location information shall conform to the vocabulary specified in OGC 11-052r4 (GeoSPARQL). To capture generic spatial objects requires concepts of location as well as concepts of geometry in order to describe shapes that are more complex than a single point in space. In addition, there is a need to be able to describe the spatial relationship between various objects (e.g. containment, overlap). The GeoSPARQL Ontology is used in the Location pattern to achieve this. It is included in its entirety with the prefix 'geo'.

The GeoSPARQL Ontology has been selected to form the basis of the representation of location information because it is the defacto standard for geospatial information on the Semantic Web. Along with the formalization of ontology concepts the GeoSPARQL standard also specifies requirements for the implementation of query functions. Conformant Semantic Web data stores provide the ability to employ these functions to enable geospatial queries over information that is represented using the vocabulary defined in the GeoSPARQL Ontology. It is therefore of practical importance to any users of the ISO/IEC 5087 standard series that the terminology used to describe locations utilize the GeoSPARQL labels.

### 6.7.2 Key classes and properties

The key classes and properties extended from GeoSPARQL are formalized in Table 8 and Table 9, respectively. Specifically, the class Location is added, and and the properties hasLocation and associatedLocation are introduced. These terms represent a minor extension of the GeoSPARQL ontology.

— **Location**: a specialization (subclass) of geo:Geometry that refers to a geospatial representation of the space occupied by some object.

In addition, the pattern introduces the associatedLocation property to support the reference of locations by other classes:

— **hasLocation**: specifies the Location of an object.

— **associatedLocation**: captures the association of a non-spatial object with a Location. For example, an activity is not a spatial object but may be associated with some location in which it (mostly) takes place.

The Mereology pattern is imported in order to specify the relationship between the mereotopological properties defined in GeoSPARQL for Spatial Objects and the more general part-whole relations that can also apply to other types of entities; namely, to relate geo:'tangential proper part' and geo:non -tangential proper part properties with the partwhole:properPartOf property. These relationships are illustrated in Table 9.

The diagram in Figure 10 illustrates the use of the Location pattern to represent a vehicle's Location. Note that the Location of a particular Vehicle individual can then be associated with other Locations (e.g. via a containment relation). Note that the Location object may be a point or other more complex shape and may be associated with more than one data property based on different serializations (such as well-known text, shown here).

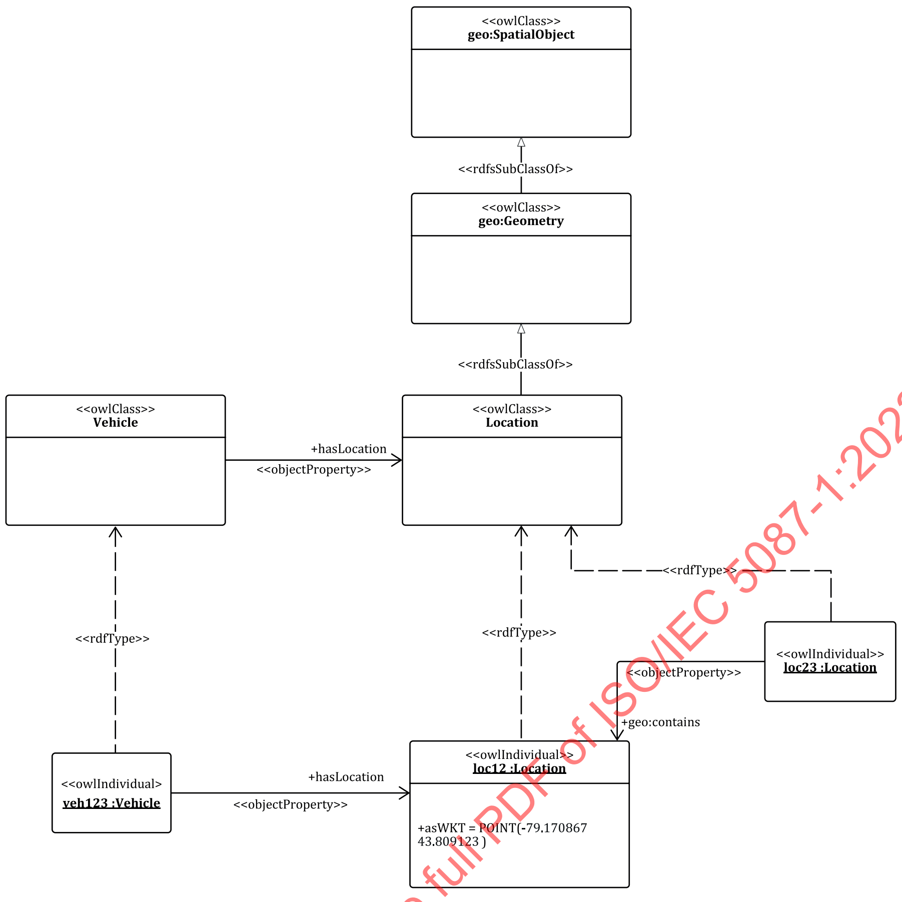


Figure 10 — Example use of the Location pattern

### 6.7.3 Formalization

Table 8 — Key classes in the Location pattern

| Class | Property | Value restriction |
|---|---|---|
| Location | rdfs:subClassOf | geo:Geometry |

Table 9 introduces two new properties and extends a subset of properties in GeoSPARQL so that they are integrated with the Mereology pattern.

Table 9 — Additional properties introduced in the Location pattern

| Property | Characteristic | Value (if applicable) |
|---|---|---|
| hasLocation | rdfs:range | Location |
| associatedLocation | rdfs:range | Location |
| tangentialProperPart | rdfs:subPropertyOf | geo:'tangential proper part' |
| nonTangentialProperPart | rdfs:subPropertyOf | geo:'non-tangential proper part' |
| tangentialProperPart | rdfs:subPropertyOf | partwhole:properPartOf |
| nonTangentialProperPart | rdfs:subPropertyOf | partwhole:properPartOf |
| hasGeometry | rdfs:Range | geo:Geometry |

## 6.8 Activity pattern

### 6.8.1 General

The concept of activities arises in many cases in city data: there are trip activities that contribute to the demand on a transportation system, and the routine activities that motivate these trips; there are educational and recreational activities offered by various city services. In the most general sense, activities are things that happen; events that occur (scheduled or not) or actions that are performed by some agent. Activities may be described by the time and location of their occurrence, their preconditions and effects, as well through the identification of any objects that are somehow involved.

There are many OWL ontologies that in some way address the concept of activities. However, most are lacking with respect to the basic representation requirements. The Activity pattern adopts the Activity Specification design pattern that was presented in Reference [14] as a solution to address these limitations. The representation of activity specifications is based on the activity clusters introduced by Fox, Sathi, and colleagues[15],[16].

A precursor to the TOronto Virtual Enterprise[17] and Process Specification Language[18] activity ontologies, an activity cluster provides a basic structure for representing activity specifications. Illustrated in Figure 11, it consists of an activity connected to an enabling and caused state, each of which may be a state tree that defines complex states via decomposition into conjunctions and disjunctions of states.

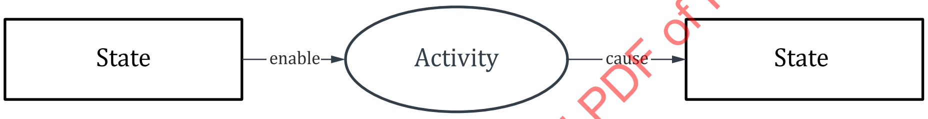


**Figure 11 — A generic activity cluster**

In this approach an activity is interpreted as a class of occurrences, in contrast with other approaches where activities are separate entities that are related to occurrences via an "occurrence of" relation. This decision was motivated by several pragmatic factors: in many cases it is sufficient to capture information regarding individual activities (i.e. occurrences or events). These activities may be categorized via different subclasses of "Activity", but there is no need to associate them with a single activity type entity, unless it is wished to characterize the activity type itself. The capability for this more complex formalization is supported, if necessary, by the Recurring Event Pattern (see 6.9). Dividing these representations into two separate ontologies allows users of the ISO/IEC 5087 series the discretion to only include what is needed. In addition, much of the semantics that relate activity types and occurrences is not expressible in OWL. The Activity Ontology works within the limitations of OWL to capture the concepts of activities, their composition, preconditions and effects, and ordering.

### 6.8.2 Key classes and properties

The key classes and properties are described below and are formalized in Table 11 and Table 12, respectively.

— **Activity**: An Activity describes something that occurs in the domain. It has the following set of core properties:

**hasSubactivity**: identifies a more granular Activity into which the Activity may be decomposed.

**hasStatus:** identifies an ActivityStatus. This specifies the status of the Activity at some point or interval in time. For example, the Activity may be "scheduled, "executing" or "completed".

**hasPrecondition:** identifies a State that must be realized in order for the Activity to occur.

**hasEffect:** identifies a State that is realized once the Activity has occurred.

**occursAt:** identifies a time Interval over which the Activity occurs.

**loc:associatedLocation:** identifies a Location where the Activity occurs.

**scheduledFor:** identifies the time Interval that an Activity was scheduled to be performed/occur at.

**occursBefore:** identifies an Activity that the Activity occurred before.

An Activity may also be described with the following, supplemental properties:

**enabledByState:** identifies a State that in some (indirect) way enabled the Activity to occur. An Activity is enabled by a State if the State is a precondition for the Activity or if the State is a precondition of some subactivity of the Activity. The enabledByState property is a generalization (super-property) of the hasPrecondition property.

**causesState:** identifies a State that in some (indirect) way was caused by the occurrence of the Activity. An Activity is caused by a State if the State is an effect of the Activity or if the State is an effect of some subactivity of the Activity. The causes property is a generalization (super-property) of the hasEffect property.

**occursDirectlyBefore:** identifies an Activity that occurred immediately prior to the Activity. The occursDirectlyBefore property is a sub-property of the occursBefore property.

**beginOf:** identifies the time Instant at which the Activity occurs.

**endOf**: identifies the time Instant at which the Activity ends.

— State: A State describes some situation in the world which may or may not be satisfied (true) at a given point in time. States have the following core set of properties:

**hasStatus**: identifies the StateStatus at some point or interval in time. For example, the State may be "unsatisfied, "satisfied" or "completed".

**achievedAt**: identifies the time Interval or Instant during which the State was satisfied.

**scheduledFor**: identifies a time Interval during which the State is scheduled to be realized.

**effectOf**: identifies an Activity that the State was realized by.

**preconditionOf**: identifies an Activity that requires the State to be realized in order to occur.

A State may also be described with the following, supplemental properties:

**enablesActivity:** identifies an Activity that in some (indirect) way was enabled by the State. The enables property is the inverse of the enabledBy property and is a generalization (super-property) of the preconditionOf property.

**causedByActivity:** identifies an Activity that in some (indirect) way caused the State to be realized. The causedBy property is the inverse of the causes property and is a generalization (super-property) of the effectOf property.

— TerminalState: A State may be either non-terminal or terminal. A TerminalState has no sub-states.

— ManifestationState: A specialization of TerminalState, the ManifestationState specifies a Manifestation class that an individual must satisfy in order for the ManifestationState to be true.

**satisfiedBy**: specifies the Manifestation that is to be satisfied (i.e., realized).

— NonTerminalState: a NonTerminalState has child States, which may be TerminalStates, or further define some other complex state types. A State cannot be both non-terminal and terminal. NonTerminalStates have the following core properties:

**hasDecomp**: identifies two or more States into which the State may be immediately decomposed, i.e. its direct children.

NonTerminalState has the following supplemental properties:

**hasSubState**: identifies a State into which the complex state decomposes, at any level (i.e. its child state, grandchild state, etc.). The hasSubState property is transitive and a super-property of the hasDecomp property.

— ConjunctiveState**:** a ConjunctiveState is a type of NonTerminalState that is defined by the conjunction of its child States.

— DisjunctiveState**:** a DisjunctiveState is a type of NonTerminalState that is defined by the disjunction of its child States. A State cannot be both conjunctive and disjunctive. Conjunctive and disjunctive States, which do have substates, are achieved at some time if their decomposition of States is achieved.

It is not possible to completely define the semantics of an ordering over occurrences in OWL. However, the start and end times of an activity may be used to describe the occursBefore property using object property chaining as formalized in Table 10.

**Table 10 — Formalization of occursBefore**

| **Property** | **Characteristic** | **Value (if applicable)** |
|---|---|---|
| endOf o before o inverse(beginOf) | rdfs:subPropertyOf | occursBefore |
| occursBefore | TransitiveObjectProperty | |
| occursAt o intervalMeets o inverse(occursAt) | rdfs:subPropertyOf | occursDirectlyBefore |

Figure 12 illustrates the use of the Activity pattern to describe a class of objects that should be identified as “Drive to Work” activities, where the effect of such an Activity is a ConjunctiveState. The ConjunctiveState is decomposed into two TerminalStates, meaning that it should be interpreted as a State where both substates (“DriverAtWork” and “CarAtWork”) are true.

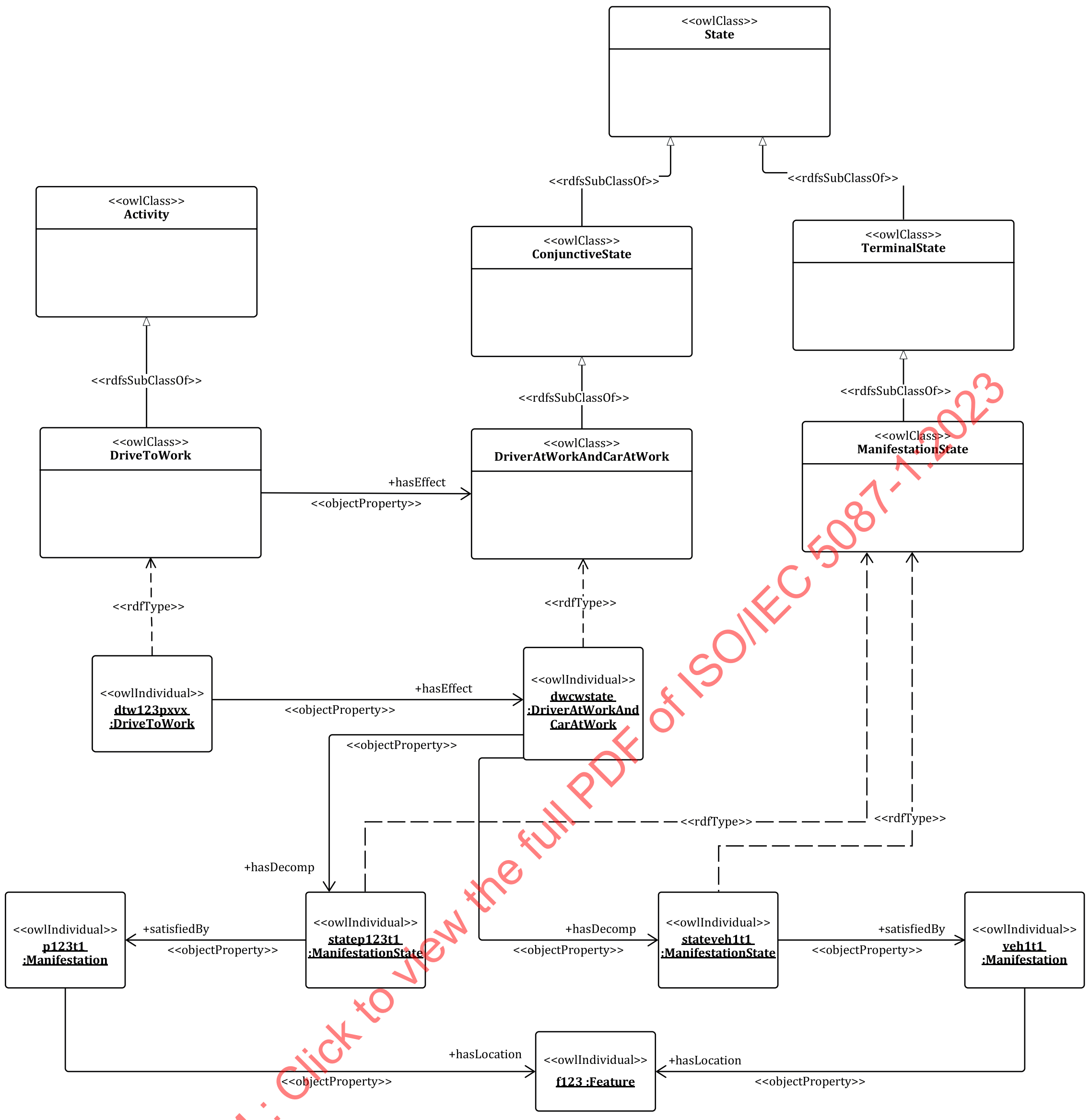


**Figure 12 — Example use of the Activity pattern**

### 6.8.3 Formalization

Table 11 — Key classes in the Activity pattern

| Class | Property | Value restriction |
|---|---|---|
| Activity | hasSubactivity | only Activity |
| | hasStatus | exactly 1 ActivityStatus |
| | hasPrecondition | only State |
| | enabledByState | only State |
| | hasEffect | only State |
| | causesState | only State |
| | scheduledFor | exactly 1 time:Interval |
| | occursAt | some time:Interval |
| | beginOf | some time:Instant |
| | endOf | some time:Instant |
| | loc:associatedLocation | only loc:Location |
| | occursBefore | only Activity |
| | occursDirectlyBefore | only Activity |
| State | preconditionOf | only Activity |
| | enablesActivity | only Activity |
| | effectOf | only Activity |
| | causedByActivity | only Activity |
| | scheduledFor | max 1 time:Interval |
| | achievedAt | only time:TemporalEntity |
| | hasStatus | exactly 1 StateStatus |
| TerminalState | rdfs:subClassOf | State |
| | disjointWith | NonTerminalState |
| | hasDecomp | exactly 0 State |
| ManifestationState | rdfs:subClassOf | TerminalState |
| | satisfiedBy | only change:Manifestation |
| NonTerminalState | rdfs:subClassOf | State |
| | disjointWtih | TerminalState |
| | hasDecomp | only State and min 2 State |
| | hasSubstate | only State |
| ConjunctiveState | rdfs:subClassOf | NonTerminalState |
| | disjointWith | DisjunctiveState |
| DisjunctiveState | rdfs:subClassOf | NonTerminalState |
| | disjointWith | ConjunctiveState |
| ActivityStatus | rdfs:subclassOf | owl:Thing |
| StateStatus | rdfs:subclassOf | owl:Thing |

**Table 12 — Key properties in the Activity pattern**

| Property | Characteristic | Value (if applicable) |
|---|---|---|
| hasPrecondition | rdfs:subPropertyOf | enabledBy |
| hasEffect | rdfs:subPropertyOf | causedByActivity |
| enablesActivity | inverseProperty | enabledByState |
| preconditionOf | rdfs:subPropertyOf | enablesActivity |
| causedByActivity | inverseProperty | causesState |
| effectOf | rdfs:subPropertyOf | causedByActivity |
| occursDirectlyBefore | rdfs:subPropertyOf | occursBefore |
| occursBefore | TransitiveObjectProperty | — |
| hasSubState | TransitiveObjectProperty | — |
| hasDecomp | rdfs:subPropertyOf | hasSubState |

## 6.9 Recurring Event pattern

### 6.9.1 General

A specification of recurring events, in particular those that are defined according to calendar dates (e.g. every Monday, every March), is required in order to capture information regarding hours of operation, road restrictions, restrictions on parking policies, and so on. A recurring event is a means of describing scenarios where some activity is scheduled to recur at some regular interval. It is important to note that recurring events such as scheduled transit trips and operating hours represent planned or usual occurrences. But exceptions to the planned recurrence can exist. For example, while a business can be open at some recurring intervals, it is possible that in the case of some exceptional circumstances (for example, a power failure) they are not open during the predefined days and times.

### 6.9.2 Key classes and properties

The key classes are formalized in Table 13. The Recurring Event pattern adopts the following representation of recurring events: daily, weekly, and monthly recurring events (and their related properties) are defined. However, the pattern may be extended with similar definitions of other types of recurring events, as required.

Note that despite the close relationship, a recurring event is distinct from an activity. An instance of a *recurring event* corresponds to a class of activities (e.g. all of the occurrences of a Tuesday, all of the occurrences of the weekly waste pickup), but it is not itself an activity. The intuition is that the occurrences of a recurring event are all the same type of activity. What defines a recurring event is a combination of the activity type (e.g. a transit trip from point A to point B or the provision of a service) and the frequency at which it recurs.

The pattern captures the associated activity type with the *hasOccurrence* property that relates recurring events to activities. Classes of recurring events may be captured by identifying their associated classes of activities, while individual recurring events may be associated with one or more instances of an activity.

The Recurring Event pattern uses the Activity pattern, as the concept of an activity is central to the notion of a recurring event: the activities are the recurrences. It is important to note that while the concept of Activity defined in the Activity pattern is necessary for the definition of a RecurringEvent, it is *not* the case that the concept of RecurringEvent is required for the definition of an Activity. This allows for a simpler representation of events in cases where the notion of recurrence is not required.

Recurring events are also identified based on the regular interval at which they occur. This is captured using some combination of the startTime, endTime, dayOfWeek and onDateTimeDescription properties. Using these properties, the pattern supports definitions of specializations of the RecurringEvent class. In particular, subclasses for daily, weekly, monthly and yearly recurring events are defined; other classes of recurring events may be defined similarly, as required. In addition, the properties startState and endState are used to identify recurring events that occur due to certain circumstances, i.e. States, rather than at specific points in time. Fuzzy sorts of recurring events that recur over imprecise periods of time are not included in this pattern. However, the specified representation may be consistently extended to include fuzzy recurring events with the introduction of classes to define concepts such as fuzzy days or fuzzy times on which such an event would recur.

Exceptions to recurring events can also be defined. For example, a business that normally operates on Monday-Friday, except for public holidays. Exceptions may also be defined on specific dates (e.g. June 23, 2018), for example due to construction or on special calendar days (e.g. holidays) with the ExceptionDay class. These exceptions can be defined for recurring events with the recursExcept property. Conversely, so-called exceptions can involve an additional, unusual occurrence. This is captured with the recursAddition property.

A RecurringEvent is defined to have the following properties:

— **hasOccurrence:** identifies the Activities that take place at the time and location.

— **associatedLocation:** identifies the Location where the event recurs.

— **hasSubRecurringEvent:** identifies the sub-recurring events that comprise the RecurringEvent. As with an Activity, a RecurringEvent may be decomposed/decomposed into simpler/more complex RecurringEvents to support varying levels of granularity.

— **startTime:** specifies the start time of the RecurringEvent's activity using xsd:time format.

— **endTime:** specifies the end time of the RecurringEvent's activity using xsd:time format.

— **beginsRecurringTime:** specifies the start time of the RecurringEvent's activity with a time: TemporalEntity from the Time pattern.

— **endsRecurringTime:** specifies the end time of the RecurringEvent's activity with a time: TemporalEntity from the Time pattern.

— **hasDayOfWeek:** specifies the day of the week on which a Weekly RecurringEvent occurs with the time:DayOfWeek class from the Time pattern.

— **onDateTimeDescription** specifies the date-time description (as defined in the Time pattern) on which a recurring event recurs. For example, the day of the month on which a MonthlyRecurringEvent recurs, or the day and month on which a YearlyRecurringEvent recurs.

— **beginsRecurringState:** defines a State that is required to be true in order to initiate the RecurringEvent.

— **endsRecurringState:** defines a State that is required to be true in order terminate the RecurringEvent.

— **recursExcept:** defines the exceptions for the RecurringEvent, i.e. conditions when it does not occur. This can specify a time, day of the week, or specific dates.

— **recursAddition:** defines a condition when the RecurringEvent should be added to. This can specify a time, day of the week, or specific dates.

An ExceptionDay specifies a day or days that recursExcept and recursAddition use to specify unique days that do not recur on the same day each year, for example, holidays. Similar sorts of exceptions (e.g. for times or months) can be specified, but are not included in this pattern. It has the following properties:

— hasName: the name of the exception day, e.g. Labour Day.

— time:hasTime: specifies the year/month/day on which the exception occurs.

A DailyRecurringEvent occurs every day. It has a maximum of one associated time, the start time. Typically, a daily-recurring event will occur at the same time every day, but it is also possible that no commitment is made to a recurring start time for the event, in which case no start time is specified. A DailyEvent does not necessarily have a recurring end time (this would require a constant duration), therefore this is not part of the definition (although it is possible to specify).

A WeeklyRecurringEvent recurs regularly on the same day of the week, as specified by the dayOfWeek property.

A MonthlyRecurringEvent recurs regularly on the same day of each month, as specified by the onDateTimeDescription property. Note that there is often ambiguity regarding the semantics of a monthly-recurring event: in this formalization, a MonthlyRecurringEvent is any event that recurs regularly on the same day of each month; other interpretations sometimes consider events that recur on the same day of week, or first or last day, in which case the day of month will vary. Such a representation is not included in this pattern but could be captured in an extension.

A YearlyRecurringEvent recurs regularly on the same day of the same month, as specified by the onDateTimeDescription property. As with MonthlyRecurringEvent, there can be ambiguity regarding the semantics of a yearly-recurring event. However, this formalization captures only the notion of an event that recurs on the same day of the same month (e.g. a birthday).

Figure 13 and Figure 14 illustrate how the Recurring Event pattern could be used to define the concept of a scheduled bus trip. In this example, the scheduled bus trip “sched_bus_trip” recurs daily at 8:00 am so all occurrences of the event (i.e. activities “bustrip1_123”, “bustrip1_321”) ought to occur at 8:00 am on their respective dates. For reference, a more complex example is outlined in Annex C.

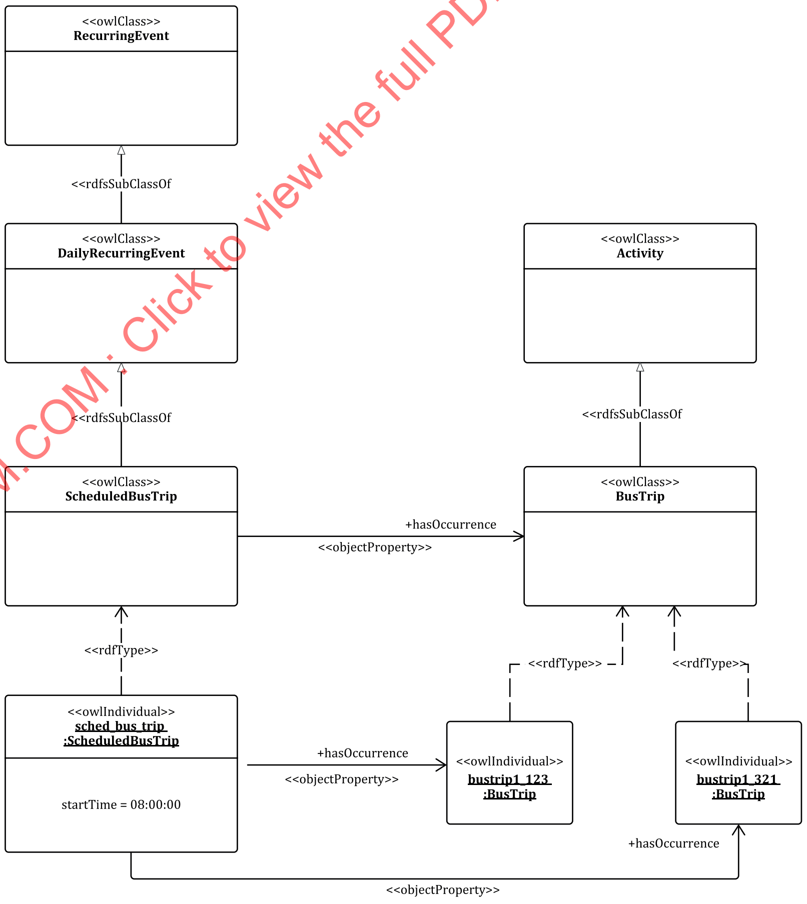


**Figure 13 — Example use of the Recurring Event pattern**

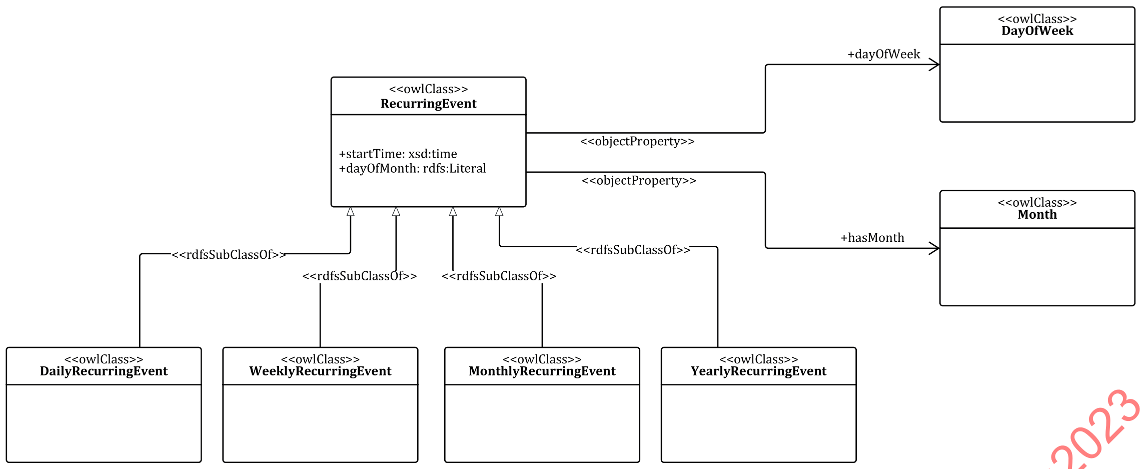


**Figure 14 — Basic structure of the Recurring Event pattern**

### 6.9.3 Formalization

**Table 13 — Key classes in the Recurring Event pattern**

| Class | Property | Value restriction |
|---|---|---|
| RecurringEvent | hasOccurrence | only activity:Activity |
| | associatedLocation | only loc:Location |
| | hasSubRecurringEvent | only RecurringEvent |
| | startTime | only xsd:time |
| | endTime | only xsd:time |
| | dayOfWeek | only DayOfWeek |
| | onDateTimeDescription | only time:DateTimeDescription |
| | beginsRecurringTime | only time:TemporalEntity |
| | endsRecurringTime | only time:TemporalEntity |
| | beginsRecurringState | only State |
| | endsRecurringState | only State |
| | recursExcept | only time:TemporalEntity or DayOf-Week |
| | recursAddition | only time:TemporalEntity or DayOf-Week |
| DailyRecurringEvent | rdfs:subClassOf | RecurringEvent |
| | startTime | max 1 xsd:time |
| WeeklyRecurringEvent | rdfs:subClassOf | RecurringEvent |
| | hasDayOfWeek | exactly 1 DayOfWeek |
| MonthlyRecurringEvent | rdfs:subClassOf | RecurringEvent |
| | onDateTimeDescription | Exactly 1 (time:DateTimeDescription and time:day exactly 1 rdfs:Literal and time:month exactly 0 rdfs:Literal and time:year exactly 0 rdfs:Literal) |
| YearlyRecurringEvent | rdfs:subClassOf | RecurringEvent |
| | onDateTimeDescription | exactly 1 (time:DateTimeDescription and time:day exactly 1 rdfs:Literal and time:month exactly 1 rdfs:Literal and time:year exactly 0 rdfs:Literal) |

**Table 13** *(continued)*

| Class | Property | Value restriction |
|---|---|---|
| ExceptionDay | genprop:hasName | some xsd:string |
| | onDateTimeDescription | some time:DateTimeDescription |

## 6.10 Resource pattern

### 6.10.1 General

Resources are an important aspect of activities. They often capture important preconditions and effects of activities. In the context of city data, resources such as vehicles, income and transit passes will impact travel behaviour. The representation of resources is also important for tasks related to asset management. For example, city infrastructure and its scheduled maintenance activities and failure rates are important factors for predicting the performance of various city services.

The Resource pattern adopts the view presented in the Toronto Virtual Enterprise model[19] that whether or not an object is considered a resource is dependent on the role that it plays for a particular activity. A Resource can be associated with an activity in different ways: it can be intended for an activity (this is captured with the Planned Allocation class), or it can be actually consumed, used, produced, or released by an activity (this is captured with the Terminal Resource State class and its specializations, which can be used to describe the enabling and caused states of an activity, i.e. the Activity cluster described in the Activity pattern).

An object may be classified as a different type of resource, dependent on its participation in an activity. The Resource pattern reuses the Activity pattern.

### 6.10.2 Key classes and properties

The Resource pattern defines the following classes, formalized in Table 14:

— Resource: A Resource is an object that plays, or is intended to play, a role in some Activity. Various types (subclasses) of Resource may be defined as required. A Resource class may be used by or consumed by some Activity class. The specification of the Resource and the quantity used or consumed in an activity is defined by the UseState and/or ConsumeState, which are part of a State definition that is linked to an Activity by an enablesActivity property. If some Resource is used by an Activity, then when the Activity occurs, the Resource needs to be (partially) not available. If a Resource is consumed by an Activity, then the Resource (partially) ceases to exist by the end of the occurrence. Resources have the following core properties:

**hasCapacity**: identifies the quantity that specifies how much of the Resource exists, e.g. its volume, if it is liquid; how much it can hold, if it is a container.

**hasAvailableCapacity**: identifies the quantity that specifies how much of the Resource is available for use.

**capacityInUse**: identifies the portion of capacity in use, by some Activity(s).

**hasAllocation**: specifies a planned allocation for the Resource. This indicates a State and time interval to which the Resource has been allocated.

**participatesIn**: identifies the Activity by which the Resource is being used, consumed, produced, or released.

**hasLocation**: identifies the Location (as defined the in the Spatial Location pattern) where the resource is located.

For additional detail, a Resource may be classified according to more specific resource types. A Resource may either be a DivisibleResource or a NonDivisibleResource, but not both.

— DivisibleResource: may be divided for use or consumption between multiple Activities.

— NonDivisibleResource: may only be used for a single Activity at once, even if it isn't fully utilized.

— Planned Allocation: the planned assignment of a Resource to an Activity via a State. It has the following properties:

**forResource**: specifies the Resource that is allocated.

**forState**: specifies the State to which the Resource is allocated.

**hasQuantity**: specifies the amount of the Resource allocated.

**hasTime**: specifies the time:TemporalEntity (i.e. time interval or instant) at which the Resource is allocated to the State.

— TerminalResourceState: The Resource pattern extends TerminalState (defined in the Activity pattern), specialized as TerminalResourceState with the following properties:

**hasQuantity**: specifies the quantity (as defined in the City Units pattern) of the Resource that participates in the State, if applicable.

**hasResource**: specifies a Resource that participates in the State. This identifies the condition of the object that plays a resource role (i.e. is used, consumed, etc.) when the state's status is active.

**has Allocation**: specifies one or more planned allocations that identify a Resource and the time for which it is allocated to the State.

Specializations of TerminalResourceStates are identified in order to more precisely distinguish the role of Resources as preconditions and effects:

— **ConsumeState**: identifies a Resource and quantity it consumes. The quantity is removed from the Resource.

— **ProduceState**: identifies a Resource and quantity it produces.

— **UseState**: identifies a Resource and quantity it uses (without consuming).

— **ReleaseState**: identifies a Resource and quantity it releases (after using).

Figure 15, Figure 16, Figure 17 and Figure 18 illustrate the Resource pattern employed to describe the resources associated with a pothole repair activity. The basic activity cluster, represented according to the Activity pattern, is shown with a single instantiation in Figure 15. The representation describes the FixPothole activity as being enabled by a state where resources are available and a pothole exists (ResourcesAvailPotholeExists). This state class is then more precisely specified as a conjunctive state that is decomposed into a state where resources are available (ResourcesAvail), and a state where the pothole exists (PotholeExists state). Similarly, the activity causes a state where the pothole is fixed, and the resources have been affected in some way (consumed or made available). The FixPothole activity class and associated states describes the generic requirements (enabling and caused states) of any pothole fixing activity, whereas the instances describe specific occurrences of a pothole fixing activity, including specific instances of enabling and caused states. For example, Figure 15 also depicts a particular instance of the FixPothole activity that is enabled by a specific state where resources are available, s11, and a specific state where a pothole exists, s12.

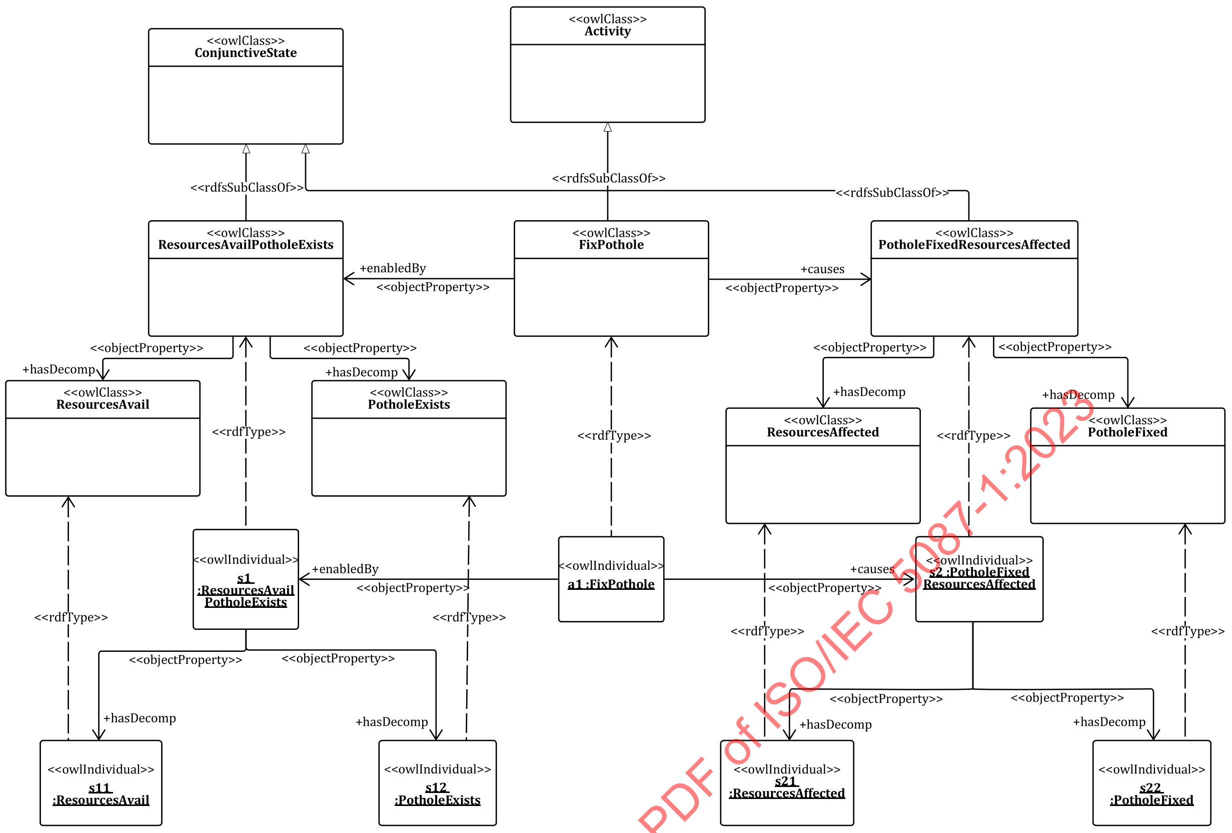
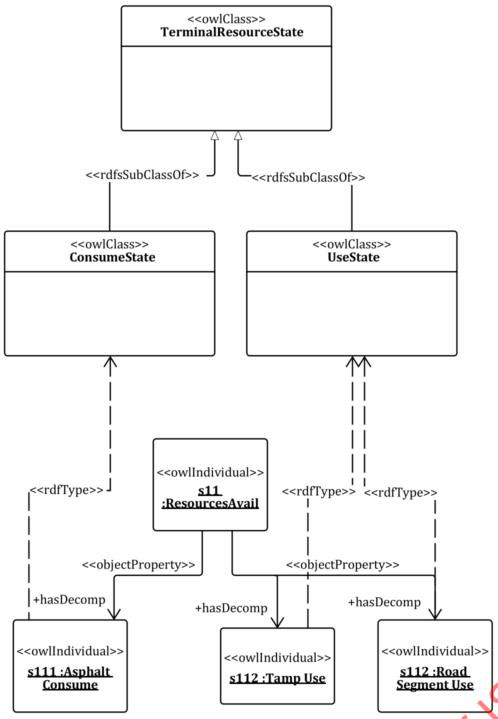


**Figure 15 — Example definition and instantiation of a basic activity cluster with resources**

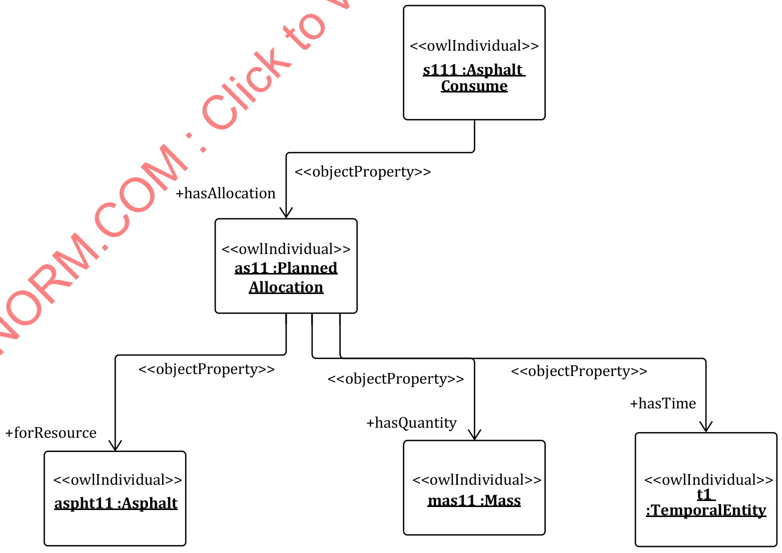
Focusing on the representation of required (enabling) resources in more detail, the ResourcesAvail state can be decomposed into three terminal states, corresponding to the required resources. The identified states, illustrated in Figure 16, are AsphaltConsume, a consume state indicating that the resource is to be consumed by the activity, TampUse, a use state indicating that the tamping tool is to be used by the activity, and Road Segment Use, a use state indicating that the road segment (with the pothole) is to be used by the activity.



<<owlClass>> TerminalResourceState
<<rdfsSubClassOf>>
<<rdfsSubClassOf>>
<<owlClass>> ConsumeState
<<owlClass>> UseState
<<rdfType>>
<<owlIndividual>> s11 :ResourcesAvail
<<rdfType>>
<<rdfType>>
<<objectProperty>>
<<objectProperty>>
+hasDecomp
+hasDecomp
+hasDecomp
<<owlIndividual>> s111 :Asphalt Consume
<<owlIndividual>> s112 :Tamp Use
<<owlIndividual>> s112 :Road Segment Use

**Figure 16 — Decomposition of individual resource states**

This level of detail can be sufficient in some cases. However, if there is a need to capture the planned allocation of resources, then the planned allocation class can be instantiated. Figure 17 illustrates an allocation defined for the Asphalt Consume state, s11. This indicates that a quantity (mas11, a measurement of mass) of a particular resource (aspht11, an instance of Asphalt) has been allocated to the state at time, t1.



<<owlIndividual>> s111 :Asphalt Consume
<<objectProperty>>
+hasAllocation
<<owlIndividual>> as11 :Planned Allocation
<<objectProperty>>
<<objectProperty>>
<<objectProperty>>
+forResource
+hasQuantity
+hasTime
<<owlIndividual>> aspht11 :Asphalt
<<owlIndividual>> mas11 :Mass
<<owlIndividual>> t1 :TemporalEntity

**Figure 17 — Example instantiation of PlannedAllocation**

If or when it is required, the actual resource use (or consumption, production, or release) can also be defined for individual states. The diagram in Figure 18 illustrates the specification that the resource that was allocated (i.e. planned for use) to the Asphalt Consume state was actually used, but the quantity and the time at which it was used were different than planned.

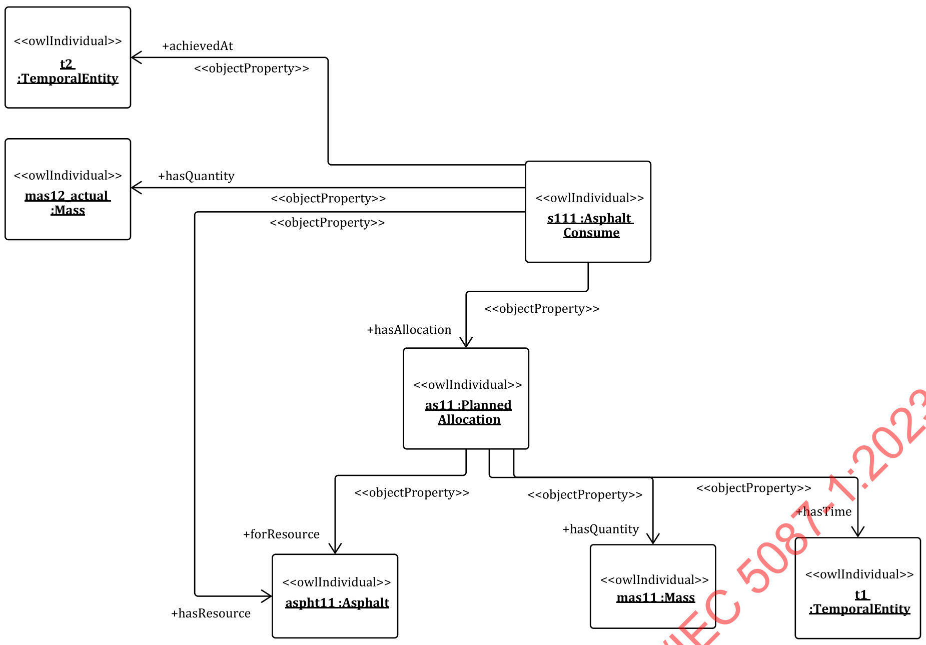


**Figure 18 — Example representation of actual resource use (and consumption)**

## 6.10.3 Formalization

**Table 14 — Key classes in the Resource pattern**

| Class | Property | Value restriction |
|---|---|---|
| Resource | hasLocation | only loc:Location |
| | hasCapacity (functional) | exactly 1 i72:Quantity |
| | capacityInUse (functional) | exactly 1 i72:Quantity |
| | hasAvailableCapacity (functional) | exactly 1 i72:Quantity |
| | participatesIn | only activity:Activity |
| | hasAllocation | only PlannedAllocation |
| DivisibleResource | rdfs:subClassOf | Resource |
| | disjointWith | NonDivisibleResource |
| NonDivisibleResource | rdfs:subClassOf | Resource |
| | disjointWith | DivisibleResource |
| PlannedAllocation | time:hasTime | exactly 1 time:TemporalEntity |
| | forResource | exactly 1 Resource |
| | hasQuantity | exactly 1 i72:Quantity |
| | forState | exactly 1 activity:State |
| TerminalResourceState | rdfs:subClassOf | activity:TerminalState |
| | hasQuantity | only i72:Quantity |
| | hasResource | only Resource |
| | hasAllocation | only PlannedAllocation |
| ConsumeState | rdfs:subClassOf | TerminalResourceState |
| ProduceState | rdfs:subClassOf | TerminalResourceState |

**Table 14** *(continued)*

| Class | Property | Value restriction |
|---|---|---|
| UseState | rdfs:subClassOf | TerminalResourceState |
| ReleaseState | rdfs:subClassOf | TerminalResourceState |

The properties in Table 15 are defined for use by other patterns.

**Table 15 — Additional properties in the Resource pattern**

| Property | Characteristic | Value restriction (if applicable) |
|---|---|---|
| hasAsset | | |
| hasResource | inverseProperty | resourceOf |
| has Allocation | inverseProperty | forState |

## 6.11 Agent pattern

### 6.11.1 General

An Agent is defined in the context of an Activity that it affects or is affected by, or their role within an organization. An Agent can refer to something like a person, organization, software or mechanical device.

### 6.11.2 Key classes and properties

The key class in this pattern is formalized in Table 16.

— Agent: An Agent affects, is affected by, or performs some Activity(s). Examples of an Agent include persons and organizations. An Agent has the following core properties:

**hasName**: an identifier for the Agent;

**resourceOf**: identifies what State the agent may be a resource of. This is specified using the Resource pattern, where the Agent would be considered to be a resource from the perspective of an Activity that uses it in some way. For example, a construction activity may use some persons(s) and those persons would be considered a Resource for that Activity.

**performs**: identifies activities that the Agent performs.

### 6.11.3 Formalization

**Table 16 — Key classes in the agent pattern**

| Class | Property | Value restriction |
|---|---|---|
| Agent | rdfs:subClassOf | <http://xmlns.com/foaf/spec/#term_Agent> |
| | genprop:hasName | exactly 1 xsd:string |
| | resource:resourceOf | only resource:TerminalResourceState |
| | performs | only activity:Activity |

Note that the term Agent defined in the foaf vocabulary referenced above is introduced to support identification of the relationship (subclass of, a specialization) between the class Agent defined in this document and the foaf vocabulary term <http://xmlns.com/foaf/spec/#term_Agent>. The term from the foaf vocabulary is not defined with any formal definitions and does not add anything to the formal definition here, but enables mapping with other representations that use this term, such as the W3C Recommendation "The Organization Ontology" which is a normative reference used in the Organization Structure pattern.

## 6.12 Organization Structure pattern

### 6.12.1 General

The representation of organization structure information shall conform to the ontology specified in the W3C Recommendation “The Organization Ontology”. It defines a standard set of classes and properties to describe organizations. It is included in its entirety with the prefix ‘org’.

In the W3C Organization ontology, an Organization may act as an agent and is comprised of a collection of people, organized with some structure, and with some common goal. In this pattern, an Organization is defined as a subclass of org:Organization with extensions and specializations relevant to this document.

### 6.12.2 Key classes and properties

The key classes are formalized in Table 17. In this subclause, a subset of the W3C Organization Ontology is replicated and specialized. This extension is needed to identify a formal relationship between an Organization and an Agent.

— Organization: an Organization may act as an Agent and is comprised of a collection of people, organized with some structure, and with some common goal.

### 6.12.3 Formalization

**Table 17 — Key classes in the Organization Structure pattern**

| Class | Property | Value restriction |
|---|---|---|
| Organization | rdfs:subClassOf | agent:Agent |
| | rdfs:subClassOf | org:Organization |

## 6.13 Agreement pattern

### 6.13.1 General

An agreement exists between two or more agents. It is established at some point in time and it may be considered valid only in some Location and/or for some interval in time. An agreement may be defined at varying levels of detail, this is supported with the introduction of the ComplexAgreement and AtomicAgreement class. A complex agreement may be decomposed into sub-agreements, whereas an atomic agreement cannot. Similar to the approach taken for the representation of activities, a complex agreement may be decomposed into disjunctive or conjunctive sub-agreements. This allows for the representation of both types of agreement composition. At their simplest level, the AtomicAgreement describes a commitment to some activity; this is captured with the commitsToActivity property. Finally, agreements involve some specification of rights or commitments of the involved parties. This is represented as a relationship between the involved Agent and a particular activity. The precise nature of the relationship indicates the type of agreement. The possible relationships are defined according to the elements of the so-called primary rules[20] of the Hohfeldian analytical system,[21] (and their opposites): claim and privilege.

### 6.13.2 Key classes and properties

The key classes and properties are formalized in Table 18 and Table 19, respectively. The class Agreement has the following properties:

— **involvesAgent**: identifies the Agents that are party to the Agreement.

— **validIn**: identifies the Location where the Agreement is valid.

— **establishedOn**: specifies the Instant of time at which the Agreement was created.

— **validFor**: specifies the time Interval during which the Agreement is in force.

ComplexAgreement is a subclass of Agreement and has one additional property:

— **hasSubAgreement**: identifies two or more Agreements that comprise the Agreement.

Elements of the Hohfeldian analytical system are used to define the following sub-properties of the inverse of involvesAgent (agentInvolvedIn) in order to represent the nature of the agreement between two (or more) agents in greater detail:

— **hasClaim**: the hasClaim property indicates that an Agent is the beneficiary of an Activity fulfilled by another Agent in the Agreement (i.e. the Agent with the duty to fulfil the Activity), e.g. payment of wages, provision of services.

— **hasNoClaim**: the hasNoClaim property indicates that an Agent has no claim on (i.e. has no right to) the described Activity. For example, under certain circumstances an Agent can have no claim to a service provided by another Agent (e.g. a person under the legal drinking age has no claim to any services provided by a bar).

— **hasPrivilege**: the hasPrivilege property indicates that an Agent is not required to fulfil the described Activity. For example, if gratuities are left to a person's discretion then they have the right (privilege) not to include a tip in their payment.

— **hasDuty**: the hasDuty property indicates that an Agent is required to fulfil the described Activity. Contrary to the example above, if gratuities are mandatory, then the person is required (has a duty) to include the tip in their payment.

The relationship between these properties in a given Agreement can be summarized by the following opposites and correlatives, as originally identified by Hohfeld:[21]

— If agent A hasClaim, then A lacks a hasNoClaim

— If agent A hasPrivilege, then A lacks a hasDuty

— If agent A hasClaim, then some agent B hasDuty

— If agent A hasPrivilege, then some agent B hasNoClaim

AtomicAgreement is a subclass of Agreement. It has no decomposition and specifies the "essence" of an agreement. In particular, it identifies how Agents participating in the Agreement are involved:

— **forActivity**: identifies the Activity the AtomicAgreement is for.

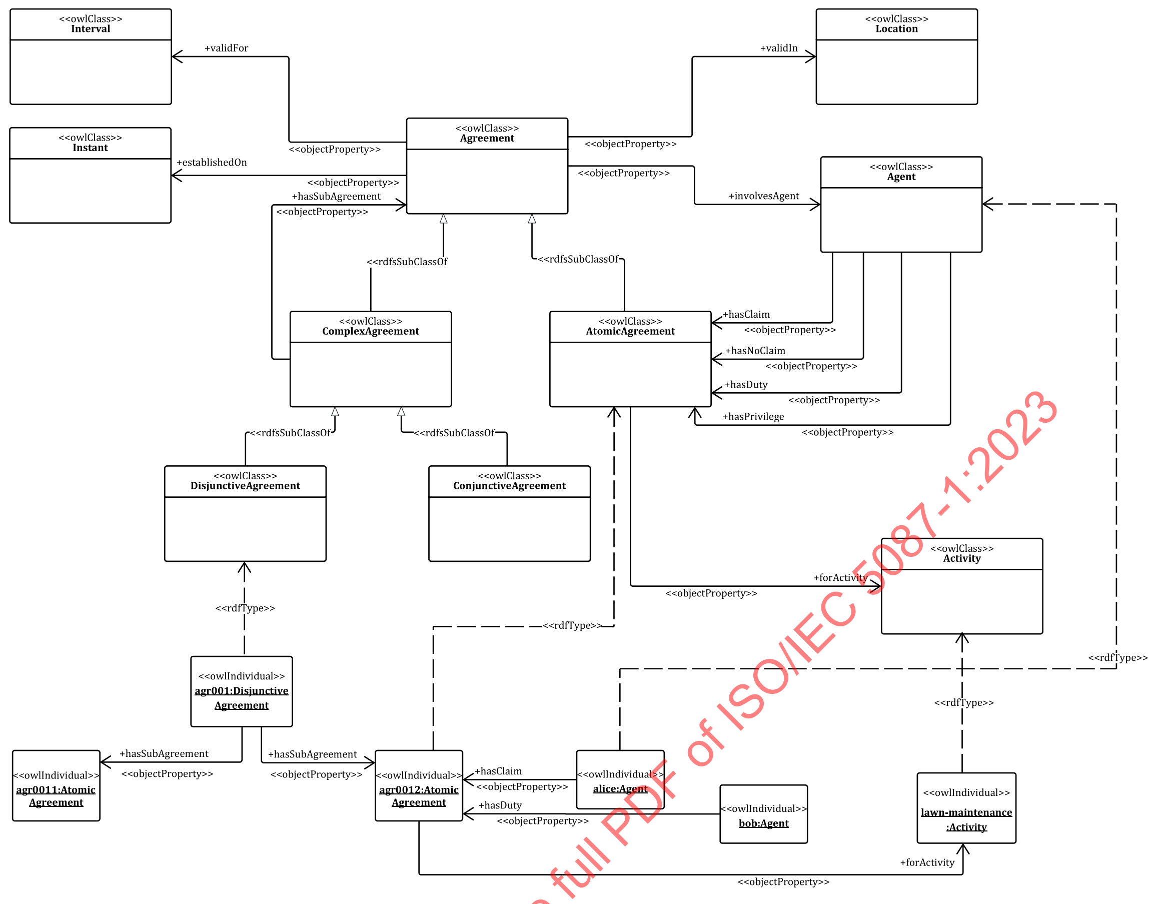
— inverse **hasClaim**: links the Agreement to any Agent that has a claim.

— inverse **hasNoClaim**: links the Agreement to any Agent that does not have a claim.

— inverse **hasDuty**: links the Agreement to any Agent that has a duty to perform the Activity.

— inverse **hasPrivilege**: links the Agreement to any Agent that has the privilege to perform the Activity.

Figure 19 illustrates the use of the Agreement pattern to represent agreements at different levels of detail. The example shown captures a complex DisjunctiveAgreement that can be decomposed into two simple Agreements. One option ("agr0012") describes "alice"'s right (claim) to have lawn maintenance be performed by "bob", (also read "bob"'s duty to perform lawn maintenance for "alice").



<<owlClass>> Interval
<<owlClass>> Location
<<owlClass>> Instant
<<owlClass>> Agreement
<<owlClass>> Agent
<<owlClass>> ComplexAgreement
<<owlClass>> AtomicAgreement
<<owlClass>> DisjunctiveAgreement
<<owlClass>> ConjunctiveAgreement
<<owlClass>> Activity
+validFor
+validIn
+establishedOn
+hasSubAgreement
+involvesAgent
+hasClaim
+hasNoClaim
+hasDuty
+hasPrivilege
+forActivity
<<objectProperty>>
<<rdfsSubClassOf>>
<<rdfType>>
<<owlIndividual>> agr001:Disjunctive Agreement
<<owlIndividual>> agr0011:Atomic Agreement
<<owlIndividual>> agr0012:Atomic Agreement
<<owlIndividual>> alice:Agent
<<owlIndividual>> bob:Agent
<<owlIndividual>> lawn-maintenance :Activity

Figure 19 — Example use of the Agreement pattern

### 6.13.3 Formalization

Table 18 — Key classes in the Agreement pattern

| Class | Property | Value restriction |
|---|---|---|
| Agreement | involvesAgent | min 2 Agent |
| | validIn | only loc:Location |
| | establishedOn | only time:Instant |
| | validFor | only time:Interval |
| ComplexAgreement | rdfs:subClassOf | Agreement |
| | hasSubAgreement | min 2 Agreement |
| AtomicAgreement | rdfs:subClassOf | Agreement |
| | hasSubAgreement | exactly 0 Agreement |
| | forActivity | min 1 Activity |
| | inverse hasDuty | only Agent |
| | inverse hasPrivilege | only Agent |
| | inverse hasClaim | only Agent |
| | inverse hasNoClaim | only Agent |
| | disjointWith | ComplexAgreement |

**Table 18** *(continued)*

| Class | Property | Value restriction |
|---|---|---|
| Agent | rdfs:subClassOf | agent:Agent |
| | hasClaim | only AtomicAgreement |
| | hasNoClaim | only AtomicAgreement |
| | hasDuty | only AtomicAgreement |
| | hasPrivilege | only AtomicAgreement |
| DisjunctiveAgreement | rdfs:subClassOf | ComplexAgreement |
| | disjointWith | ConjunctiveAgreement |
| ConjunctiveAgreement | rdfs:subClassOf | ComplexAgreement |
| | disjointWith | DisjunctiveAgreement |

**Table 19 — Key properties in the Agreement pattern**

| Property | Characteristic | Value (if applicable) |
|---|---|---|
| hasClaim | rdfs:subPropertyOf | agentInvolvedIn |
| hasNoClaim | rdfs:subPropertyOf | agentInvolvedIn |
| hasDuty | rdfs:subPropertyOf | agentInvolvedIn |
| hasPrivilege | rdfs:subPropertyOf | agentInvolvedIn |
| agentInvolvedIn | inverseOf | involvesAgent |

## 6.14 Provenance pattern

### 6.14.1 General

The concept of provenance is important for many applications in the smart city domain. It is often important to represent the origin of a particular record, document or other object. To do this in a consistent way requires a mechanism to represent the provenance of objects. The ontology for representing provenance shall conform to the ontology specified in the W3C Recommendation "PROV-O: The PROV Ontology". PROV-O is a W3C standard that defines an appropriate set of classes and properties for the representation of provenance information in any domain.

### 6.14.2 Key classes and properties

The key classes are formalized in Table 20. The core classes of Agent and Activity in PROV-O are extended with specialized classes (subclasses). These classes extend the original classes defined in PROV-O in order to formalize the relationship with the Agent and Activity patterns defined in this document. Without this extension, there would be nothing to connect the Agent and Activity classes defined in PROV-O with the Agent and Activity classes defined in the Agent and Activity patterns, respectively.

### 6.14.3 Formalization

**Table 20 — Key classes in the Provenance pattern**

| Class | Property | Value |
|---|---|---|
| Agent | rdfs:subClassOf | prov:Agent |
| | rdfs:subClassOf | agent:Agent |
| Activity | rdfs:subClassOf | prov:Activity |
| | rdfs:subClassOf | activity:Activity |

# Annex A
(informative)

# Implementation alternatives for additional change semantics

It can be desirable to support inferences regarding the inheritance of invariant properties. In such cases, a rule of the following form may be specified, property, where invariantProperty represents a placeholder for the property:

FirstManifestation(?x) & invariantProperty(?x,?y) & precedesManifestation(?x,?n) -> invariantProperty(?n,?y)

The SWRL rules language[22] is used above as an example. However, in practice, the chosen rules language will vary depending on the intended implementation. This asserts that all subsequent manifestations should inherit the invariant property from the first manifestation. Returning to the example, the class expressions asserted for Arc in the adapted representation now apply to each snapshot of an Arc. In other words, at any point in time an Arc should have exactly one TTI value, exactly one start Node and exactly one end Node. If startNode and endNode are identified as invariant properties, then they should be annotated as such in order to be interpreted correctly. As described above, a rule of the following form could also be asserted to support this interpretation:

FirstManifestation(?x) & startNode(?x,?y) & precedesManifestation(?x,?n) -> startNode(?n,?y)

FirstManifestation(?x) & endNode(?x,?y) & precedesManifestation(?x,?n) -> endNode(?n,?y)

Note that with this approach, any property that is identified as invariant is defined so universally. This can require the definition of class-specific properties in cases where a property is identified as variant for one class, but invariant for another (for example, the height of a person vs. the height of a table).

In order to maintain decidability, OWL2 restricts the use of some class expressions to simple properties. The use of object property chaining results in all invariant properties being non-simple. Therefore, certain class expressions (e.g. cardinality) for invariant properties will not be supported with OWL2 reasoners. For this reason, it is not possible to capture the semantics of invariant properties with property chains of the following form as it would preclude the use of any class restrictions using the invariant property:

inverse (precedesManifestation) o invariantProperty -> invariantProperty

Instead, the Change pattern proposes the use of a property annotation, and possibly supplemental rules to support the necessary inference in implementation.

An alternative approach would be to define a separate object property for all subsequent manifestations, e.g. "inheritsInvariantProperty". This could be defined with the object property chain, and the constraints could be specified for the original property. Returning to the Arc example described in 6.5, the resulting alternative representation would be as shown in Tables A.1 and A.2.

**Table A.1 — Formalization of Arc with separate inherited properties**

| Class | Property | Value |
|---|---|---|
| Arc | rdfs:subClassOf | change:Manifestation |
| | arcHasInheritedStartNode | exactly 1 Node |
| | arcHasInheritedEndNode | exactly 1 Node |
| | hasTTI | exactly 1 TTI |
| Arc and FirstManifestation | startNode | exactly 1 Node |

**Table A.1** *(continued)*

| Class | Property | Value |
|---|---|---|
| Arc and FirstManifestation | endNode | exactly 1 Node |

**Table A.2 — Formalization of separate inherited properties**

| Property | Characteristic | Value (if applicable) |
|---|---|---|
| arcHasInheritedStartNode | rdfs:subPropertyOf | inverse(precedesManifestation) o startNode |
| arcHasInheritedEndNode | rdfs:subPropertyOf | inverse(precedesManifestation) o endNode |

This approach requires the definition of an additional property and some additional assertions. However, it has the advantage of being fully expressible within OWL 2.