Deliverable number:	D3.2
Deliverable title:	

API for data and knowledge exchange

Delivery date:	2010/05/31
Submission date:	2010/05/31
Leading beneficiary:	University of Amsterdam (UVA)
Status:	Version 15 (final)
Dissemination level:	PU (public)
Authors:	Jochem Liem, Wouter Beek, Floris Linnebank, Bert Bredeweg

Project number:	231526
Project acronym:	DynaLearn
Project title:	DynaLearn -7 Engaging and informed tools for learning conceptual system knowledge
Starting date:	February 1st, 2009
Duration:	36 Months
Call identifier:	FP7-ICT-2007-3
Funding scheme:	Collaborative project (STREP)

Abstract

This document describes the representations for knowledge exchange between the different components of the DynaLearn software. The basis of these representations is the DynaLearn ontology, which formalizes the basic vocabulary as it is used within the DynaLearn software. Models, simulations, and other knowledge assets exchanged between the DynaLearn components are represented based on this ontology. However, for particular tasks this approach would be suboptimal. For these tasks a tailored representation has been created. Although the focus of this document is on knowledge structures, it also describes the communication protocols used to communicate between the different components.

Internal review

- Jorge Gracia del Río (UPM), Ontology Engineering Group, Universidad Politécnica de Madrid, Spain.
- Paulo Salles (FUB), Institute of Biological Sciences, University of Brasília, Brazil.

Acknowledgements

The authors would like to thank Oscar Corcho, Jorge Gracia del Río, Esther Lozano and Michal Trna for their insightful comments on the OWL representation of QR models and the extracted vocabulary and for their work on the semantic web services. We also thank Michael Wißner and René Bühling for implementing the communication between the CM and VC components and learning to use Semantic Web technologies to be able to work with the knowledge representations. Finally, we would like to thank Jorge Gracia del Río (UPM) and Paulo Salles (FUB) for reviewing this document and providing insightful comments and corrections.

Document History

Version	Modification(s)	Date	Author(s)
01	Outline	2010-04-12	Liem, Bredeweg
02	DynaLearn Knowledge Representation	2010-04-16	Liem, Beek, Bredeweg
03	Web Ontology Language (OWL)	2010-04-19	Liem
04	An Ontological Perspective on QR	2010-04-21	Liem
05	The DynaLearn QR Ontology	2010-04-27	Liem
06	Representing QR models based on the DynaLearn QR Ontology	2010-04-29	Liem
07	Representing Support Knowledge	2010-05-05	Beek
08	Summarizing Simulation Results	2010-05-12	Liem
09	Extracting the Domain Vocabulary for Grounding	2010-05-13	Liem
10	Virtual Character Interaction	2010-05-18	Linnebank
11	Semantic Technology Web Services	2010-05-18	Liem
12	Introduction + Rewriting	2010-05-20	Liem
13	Discussion and Conclusion + Rewriting	2010-05-25	Liem
14	Processed comments by Bert Bredeweg	2010-05-30	Bredeweg, Liem
15	Internal review	2010-06-07	Gracia del Río (UPM)
16	Internal review	2010-06-08	Salles (FUB)

Contents

Chapter 1

Introduction

The DynaLearn Interactive Learning Environment (ILE) consists of three interacting components [10]. The Conceptual Modelling (CM) component allows learners to develop their conceptual ideas by articulating them in qualitative reasoning (QR) models. Furthermore, the CM component allows learners to simulate their models, and see the consequences of their ideas. The Virtual Character (VC) component is responsible for interacting with the learner via on-screen characters, planning and executing dialogues leading to a learning goal, and keeping track of a learner's progress via a learner model. The Semantic Technology (ST) component allows the storage and retrieval of QR models, grounds terminology on external resources (i.e. links terms in models to terms in structured knowledge sources such as DBPedia and WordNet), generates feedback based on a norm model, and recommendations based on a set of models.

This document describes how the knowledge is exchanged between the different components within DynaLearn. It focusses particularly on the representation of the QR models, simulations, vocabulary (extracted from QR models), and the support knowledge required to work with DynaLearn. We argue that if these representations are well-defined they can provide the basis of derived representations. As such, the other DynaLearn knowledge assets (such as questions or different kinds of feedback) will be described in less detail. In addition, deliverables reporting on the tasks that generate a particular knowledge asset will also discuss their formalisations.

1.1 Overview of Knowledge Exchange

In order to keep the knowledge flowing through the DynaLearn ILE components synchronized, we have opted to use a uniform representation throughout the components. Furthermore, by making use of a common vocabulary, it becomes relatively easy to create new representations of different knowledge assets and adapt them when particular use-cases require them.

Figure 1.1 shows the main knowledge assets as they are currently communicated. The semantic repository should function as a model library in which models can be stored and retrieved. The developed QR model representation is used to be able to index the models and perform intelligent searches. The Grounding, Ontology-Based Feedback (OBF) and Recommendation tasks also take a QR model as input. However, both the OBF and the Recommendations tasks also require an alternative representation which focusses on the the domain vocabulary used in a model.

The VC component bases the structure of its *Learner Model* on both the QR model and a summary of the simulation results. The learner model, which is a cognitive model about the students current knowledge, keeps track of a learner's current knowledge about both particular simulations of scenarios, as well as the complete model. The learner's knowledge about simulations feeds back into the learner model of the complete QR model. The support knowledge requires its own representation, that is closer to natural language in order to create a meaningful dialogue. The same is true for the questions generated on the basis of a simulation.

1.2 Document Outline

This document is structured as follows. Firstly the DynaLearn knowledge representation is presented in Section 2. Section 3 describes the Web Ontology Language (OWL), the language we have chosen as our knowledge exchange format. Since the knowledge representation of QR models and ontological structures are different, an ontological perspective is given on QR models in Section 4. The structure that results from the ontological perspective leads to the establishment of a DynaLearn ontology (Section 5), which captures the common vocabulary used to talk about QR models and simulations. The representation of QR models (Section 6) and (summaries of) simulation results (Section 7) are a natural extension of the DynaLearn ontology.

The particular representation chosen to represent QR models and their simulations are insufficient to optimally perform the Ontology-Based Feedback (OBF) and Recommendations tasks. This is due to the fact that state of the art Ontology Matching technologies (used by the OBF and recommendation tasks) require knowledge to be structured in a certain way. As such, we have chosen to create an additional alternative representation that extracts and represents the domain vocabulary from a QR model for the specific purpose of optimizing ontology matching results (Section 8). For support knowledge a similar argument holds. The support knowledge has to be structured in a particular way in order to be able to easily create natural language dialogues. As such the *Support Knowledge* also has its own structure (Section 9). Sections 10 and 11 describe the particular interaction with the Virtual Character component and the Semantic Technology component. Finally, Section 12 concludes this document.

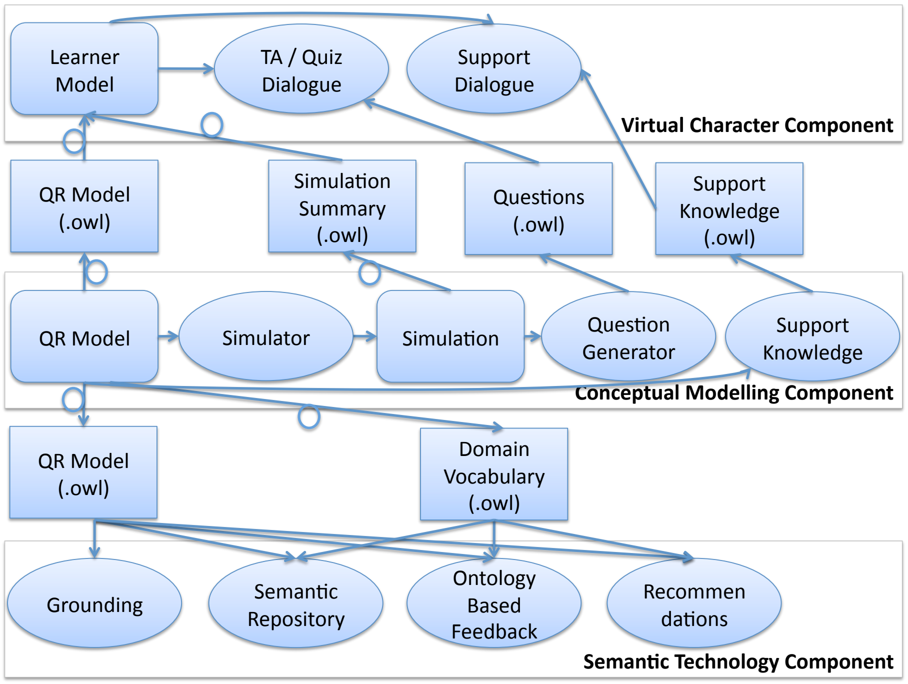

Figure 1.1: The knowledge exchange between the CM, VC and ST component. For the legend see Figure 1.2

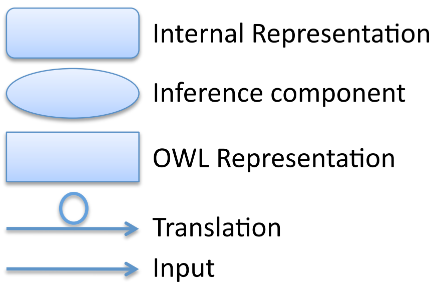

Figure 1.2: Legend for Figure 1.1.

Chapter 2

The DynaLearn Knowledge Representation

The DynaLearn ILE provides six Learning Spaces (LSs) of increasing complexity (in terms of the model ingredients that can be used to construct knowledge). Each LS is a self-contained interactive workspace that can be used to learn specific details about system behaviour. The model ingredients that are available in each LS are chosen from the full set of primitives in such a way that they emphasize a particular type of knowledge, but also allow for automated reasoning on behalf of the underlying software. The LSs and the model ingredients introduced in each LS are shown in Table 2.1. The interfaces that can be used to inspect the simulation results on each LS are shown in Table 2.2.

The goal of the LSs is to ensure that from a (qualitative) System Dynamics perspective [16], learners are able to create sensible representations of the phenomena they perceive when observing the behaviour of a real-world system. Moreover, what learners express using the software, will have consequences for what the software can infer. Hence, learners can be confronted with the logical consequences of their expressions, which may or may not match the behaviour of the observed system (or the learner's expectations thereof). Particularly in the case of mismatch there is ample room for interactive learning. Progression between LSs happens by augmenting the current level with the smallest subset of possible modelling ingredients, again ensuring that the next level is self-contained.

2.1 Learning Space 1: Concept map

The first LS in DynaLearn is meant to allow the definition of the key concepts and relationships in a domain of discourse. As such, the focus of this LS is mostly ontological. Learners using this LS are encouraged to structure their thoughts.

A concept map (sometimes referred to as an entity-relation graph) is a graphical representation that consists of two primitives: nodes and arcs. Nodes reflect important concepts, while arcs show the relationships between those concepts [28]. An example of a concept map is shown in Figure 2.1.

Nr	Learning space	Introduced ingredients
1	Concept map	Entities
		Configurations
2	Causal model	Attributes
		Quantities
		Value assignments
		Derivatives
		Causal relationships (+ and -)
3	Causal model with state graph	Quantity spaces
		Correspondences
4	Causal differentiation	Causal relationships:
		Influences
		Proportionalities
		Inequalities
		Operators: (+ and -)
		Agents
		Assumptions
5	Conditional knowledge	Conditional expressions
6	Generic and reusable	Scenarios
		Model fragments
		Entity, agent, and assumption hierarchies

Table 2.1: Overview of modelling ingredients per Learning Space in the DynaLearn software. Each LS augments the representation with a new set of model ingredients.

Nr	Learning space	Introduced simulation views
1	Concept map	
2	Causal model	Simulation in expression
3	Causal model with state graph	State graph
		Quantity values
		Value history
		Transition history
4	Causal differentiation	Equation history
5	Conditional knowledge	Entities, Configuration, and Attributes
		Dependencies view
		Model fragments view
6	Generic and reusable	

Table 2.2: Overview of simulation views per LS in the DynaLearn software.

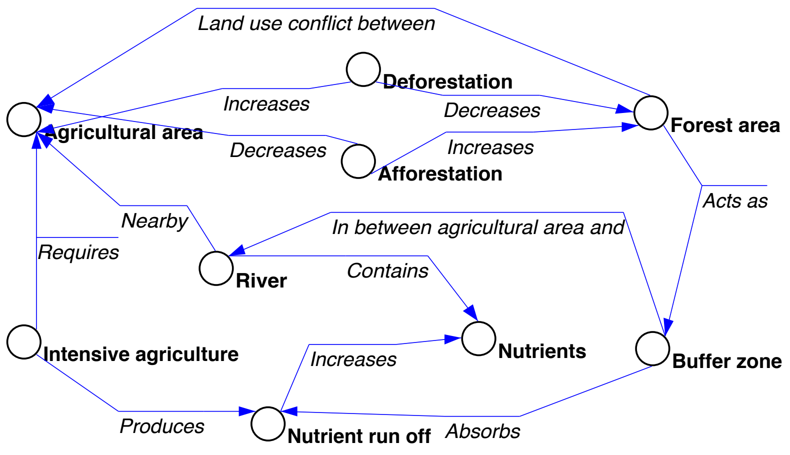

Figure 2.1: Learning Space 1: Land use conflict concept map.

Concept maps do not allow for computer-based reasoning (simulation). However, having this LS is useful from an educational point of view, as it is the basis from which more complex knowledge representations emerge. The DynaLearn ILE does not attempt to replicate the advanced features of other concept map tools (that for instance allow nested nodes, colouring and adding media) [12].

2.2 Learning Space 2: Causal model

The second LS allows learners to create a causal model (Figure 2.2). As the name suggest, the main focus of this level is the representation of *causal knowledge*. There is a causal relationship that is directed and can be either positive or negative. These relationships can be placed between quantities. For example: $Amount \xrightarrow{+} Height$ represents that changes in the height of a contained liquid is positively affected by changes in the amount of liquid.

An important ontological distinction that is introduced at this level is that between *structural and behavioural knowledge*. The structural aspects of the system are modelled using entities, configurations and attributes. For example: $Container \xrightarrow{\text{Contains}} Liquid$. The behavioural aspects of the system consist of quantities that are connected to entities ($Liquid : [Amount, Height, Pressure]$), their derivatives ($\delta Amount \in \{\blacktriangle, \varnothing, \blacktriangledown\}$), derivative value assignments ($\delta Amount_v = \blacktriangle$), and the two causal relationships mentioned above. The learner only defines the names of the quantities, while the possible derivative values are predefined.

The causal model LS is the first LS that introduces semantics to the representation. Whereas a concept map only gives a syntactic specification of the structure of a system, LS2 adds meaning to the model representation, resulting in a simulation of the modelled system's future behaviour. In this LS, a single state simulation is performed that is visualised in the model representation that the learner is manipulating (Table 2.2).

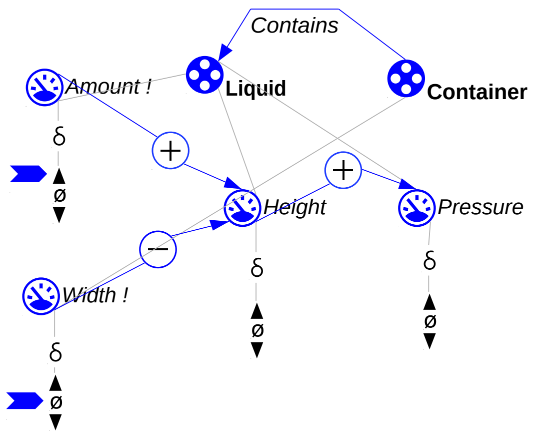

Figure 2.2: Learning Space 2: Expression of contained liquid model.

The assigned values (as e.g. shown in Figure 2.2) become grey, while derivative value assignments that are inferred by the simulator are added in blue. The reasoning engine computes the derivatives of the quantities based on the derivative value assignments and the causal relationships between the quantities.

There are several possible outcomes when calculting the derivative of a quantity. Consider the model in Figure 2.2. Consider that $\delta Amount_v = \blacktriangle$, but $\delta Width$ is unknown. In the simulation, the values for $\delta Height$ and $\delta Pressure$ will also be unknown. But if $\delta Width_v = \varnothing$, then the values for $\delta Height$ and $\delta Pressure$ would be uniquely derivable (i.e. both would be $\blacktriangle$). If $\delta Width_v = \blacktriangle$, then the competing causal relations would cause ambiguity, and value assignments would be placed on all possible derivative values of $Height$ and $Pressure$. Finally, if the derived values are inconsistent with the assigned values (e.g. $\delta Amount_v = \blacktriangle, \delta Width_v = \blacktriangledown$, and $Height_v = \blacktriangledown$), the expression is considered inconsistent, and a question mark is visualised on top of the model in order to send a signal the learner.

The representational approach taken for LS2 relates to Betty's brain [26]. One difference is that Betty's brain allows causal relationships with different strengths. In DynaLearn this kind of knowledge can be represented using inequality statements, but these are available at LS4 and beyond (Causal differentiation, see below). Another difference is the explicit inclusion of structural knowledge in DynaLearn at LS2.

2.3 Learning Space 3: Causal Model with State Graph

The representation of the causal model level is augmented with the notion of values of a quantity (notice that LS2 focussed on the direction of changes only, represented by the derivatives). Quantities can be assigned a quantity space that indicates which values a quantity can have (e.g. $Pressure_v \in \{zero, low, average, high, max\}$). Furthermore, value assignments can be put on these values ($Pressure_v = low$).

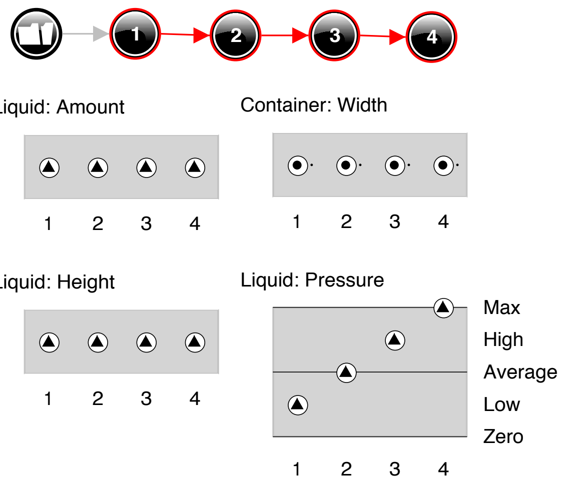

Figure 2.3: Learning Space 3: State graph and value history.

The introduction of current values for quantities has the consequence that the simulation results cannot be represented inside the expression (which is a visualisation of a single state of behaviour). The reason is that it cannot adequately show how the values of quantities change in time. As such, the simulation results are represented as a state graph (Figure 2.3), in which each state identifies a qualitatively unique situation and each transition a possible change from one situation to another (Table 2.2). A sequence of states from the begin state to the end state is called a behaviour path, and the visualisation of a sequence of values of a quantity in these states is called a value history (Figure 2.3).

Another notion that is introduced in this LS is co-occurrence. Consider that $Height$ and $Pressure$ should have the same values at the same time, and both have the quantity space $\{zero, low, average, high, max\}$. To represent this notion, correspondences are introduced. There are quantity correspondences (e.g. $Height \xrightarrow{Q} Pressure$), and value correspondences (e.g. $Height(zero) \xrightarrow{V} Pressure(zero)$), which can both be either directed or undirected. The value correspondence indicates that if $Height_v = zero$ then $Pressure_v = zero$. If the value correspondence is bidirectional, the reverse inference is also possible. Quantity correspondences can be considered a set of value correspondences between each consecutive pair of the values of both quantities. There are also inverse quantity space correspondences ($Q_1 \overset{Q\hookleftarrow}{\longleftrightarrow} Q_2$) that indicate that the first value in the quantity space of Q_1 corresponds to the last value in the quantity space of Q_2, the second to the one before last, and so on.

In LS2 and LS3, given $Q_1 \xrightarrow{+} Q_2 \xleftarrow{-} Q_3$, it is not possible to indicate which causal effect is stronger (i.e. indicating $\delta Q_1 > \delta Q_3$). The reason is the design choice of not having introduced inequalities and operators at these LSs.

2.4 Learning Space 4: Causal Differentiation

The causal differentiation LS (Figure 2.4) focusses particularly on the notion of *processes*. To accommodate this, the causal relationships (+ and -) are refined into influences and proportionalities [14]. Quantities that represent processes have influence relationships to other quantities in the system (e.g. $Birth\ rate \overset{I+}{\rightarrow} Size$). The structure of the system exemplifies a particular situation in which the process is applicable.

In order to be able to control the results of simulations, inequalities ($Birth\ rate_v > Death\ rate_v$) and operators (+ and -) are introduced (Figure 2.4). The operators are particularly useful when there are multiple conflicting causal relationships (e.g. $Birth\ rate_v + Immigration_v > Death\ rate_v + Emigration_v$, which results in $\delta Size = \blacktriangle$). How inequalities change over time is visualised in the equation history.

Since inequality statement such as the one above show a particular behaviour of the system (while removing other behaviour), the notion of assumptions becomes important. Assumption labels can be used to indicate that particular statements are true for purposes of the simulation, but are not true in general.

When modelling, learners are encouraged to focus on a constrained set of phenomena in the world. For purposes of modelling this choice constitutes the system, while all other effect on the modelled phenomena are considered to be outside the system. For this purpose, this LS introduces the notion of an agent, which models an influence from outside the system.

At LS4, it becomes possible to model the effects from outside the chosen system. For this purpose, this LS introduced the notion of agents. For example, in a model about the euthrophication of lakes due to agriculture runoff, the processes that result in the runoff can be considered less relevant to the message that the model tries to convey. Since these processes are also not affected by the eutrophication, the learner can choose not to model them explicitly. Instead, they can be considered constant influences from outside the system. This can be modelled by introducing an agent agriculture and a quantity runoff. The behaviour of the runoff can be provided via an *exogenenous quantity behaviour* [7, 6], such as increasing, decreasing, stable, sinusoidal, generate all values. Such exogeneous behaviour makes the quantity of the agent exhibit specific behaviour, without the need to explicitly model this behaviour.

The representation in LS4 relates to VMODEL [15]. One difference is that VMODEL works with single state simulations, while the DynaLearn LS4 allows for multiple state simulations. Because of that, Dynalearn LS4 also facilitates multiple views to inspect the simulation results.

2.5 Learning Space 5: Conditional Knowledge

The conditional knowledge LS focusses particularly on the activation conditions of processes. Consequently, choosing good landmark values in quantity spaces is an important task to solve. For example, the quantity space for the height of a bathtub should have a maximum value, as at this point the overflow process occurs.

Learners can create an expression using the same vocabulary as in LS4. The knowledge in the expression (except the value assignments) always applies. However, at this

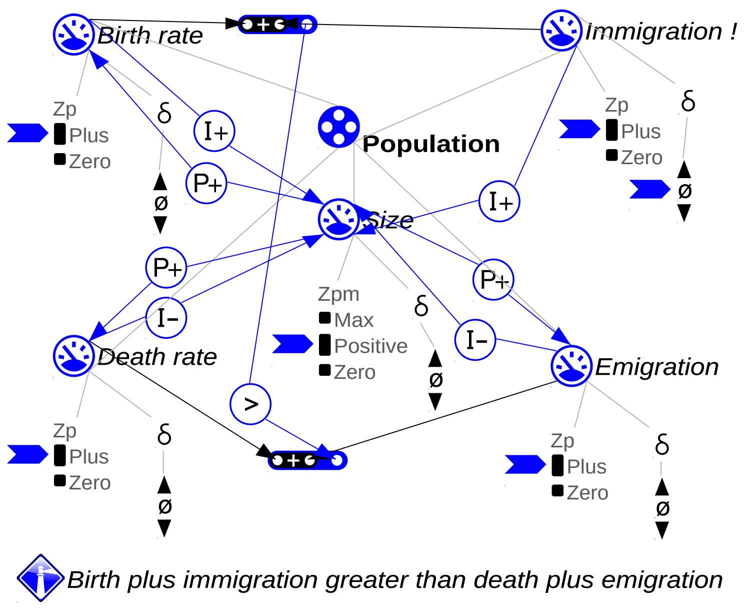

Figure 2.4: Learning space 4: Expression of population model.

level multiple conditional expressions can be defined. Conditional expressions consist of the expression and a set of conditional and consequential model ingredients (Figure 2.5). If the conditions are true, the consequences also apply. For example, in a model of a pan with water on a stove, the cooking process only becomes active if the temperature of the water is greater or equal than the cooking point. In the conditional expression of the boiling process (Figure 2.5), the inequality $Temperature \geq Cooking\ point$ is the condition that makes $Boiling\ rate_v = Positive$ (the consequence).

The conditional expression introduced in LS5, make modelling assumption more natural. Where in LS4 all assumptions and related model ingredients are inside a single expression, in LS5 each assumption (modelled as a condition) can be combined with a set of consequential model ingredients in a conditional expression. Assumptions can be added to the expression to run simulations.

The structural and behavioural relationships can dynamically change on LS5, as conditional expressions can introduce new entities and configurations, or new causal relationships in certain states. To investigate these newly introduced model ingredients the *entities, configurations and attributes* and the *dependencies* views are made available (Table 2.2). The former shows the structure of the model, while the latter shows the behavioural relationships in a particular state. The *Model fragment* view shows which of the conditional expressions have become active.

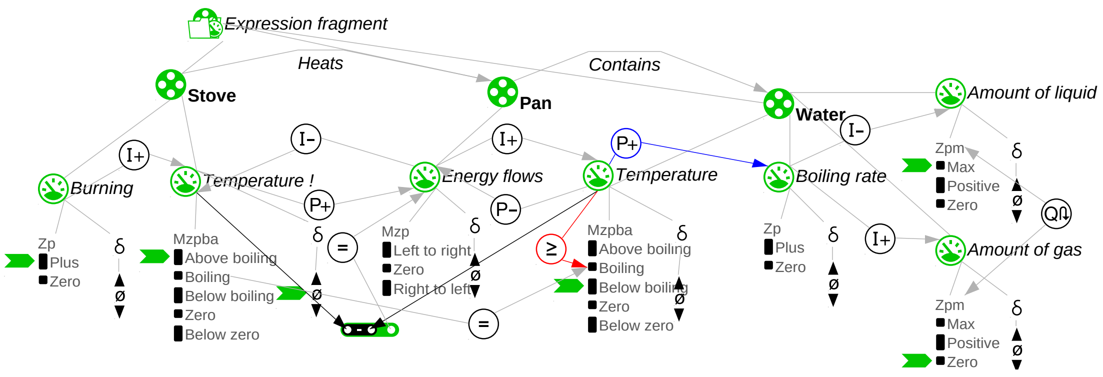

Figure 2.5: Learning space 5: Conditional expression of boiling process. The conditional expression introduces $Temperature \overset{P+}{\rightarrow} Boiling\ rate$, if $Temperature_v \geq Boiling$. The other content is inherited from the expression.

2.6 Learning Space 6: Generic and Reusable

The main focus of the generic and reusable LS is on generic knowledge and first principles. In contrast with the earlier LSs, where it is more natural to talk about specific instances of situations were certain processes are active, in LS6 the knowledge is represented in a generic way in Model Fragments (MFs). These can be considered formalisations of the generic knowledge that applies in multiple situations. Model fragments can be considered rules indicating that if certain model ingredients are present (conditions), certain other model ingredients must also apply (consequences). They can be represented as: $conditions \Rightarrow consequences$. Not every ingredient can be used as both a condition and a consequence. For example, causal relationships should always be consequences of structure (entities, configurations). The full listing of which ingredients can be used as conditions and which as consequences is shown in Table 2.3.

Three types of model fragments are distinguished: (1) process fragments are used to model processes, (2) agent fragments are used to represent the effects of agents, and (3) static fragments are used for the static structure of the system. Model fragments are organised in a is-a hierarchy, which causes model ingredients to be inherited to child model fragments. Furthermore, this LS allows model fragments to be reused in other model fragments.

Next to model fragments, different *scenarios* can be modelled. These represent specific situations that the system can be found in (with specific initial values). As such, behavioural relationships cannot be modelled in scenarios (only in model fragments), as these belong to the general knowledge. Note that the possible ingredients in a scenario correspond to the possible conditional model ingredients in a model fragment (Table 2.3). DynaLearn can run simulations of models based on a particular scenario. As in the previous LSs, the result of a simulation is a state graph.

At this level the notion of model ingredient definitions becomes important. Entity definitions are organised in an is-a hiearchy (e.g. a Lion is an animal). By specifying such ontological knowledge, it becomes possible to develop generally applicable

Possible conditions	**Impossible conditions**
Entities	Correspondences
Configurations	Proportionalities
Agents	Influences
Attributes	
Quantities	
Inequalities	
Minus/Plus	
Assumptions	
Model Fragments	
Possible consequences	**Impossible consequences**
Entities (except in static MF)	Agents
Configurations (except in static MF)	Assumptions
Attributes	Model Fragments
Quantities	
Inequalities	
Minus/Plus	
Correspondences	
Proportionalities	
Influences	

Table 2.3: Possible condition and consequence roles of QR ingredients.

model fragments. The more specific definitions lower in the hierarchy can be used to model particular cases in scenarios. Simulations then show how the generic knowledge applies to that particular case.

Examples of models in this LS can be found in a special issue of *Ecological Informatics* [9].

2.7 DynaLearn reasoning using the Garp3 engine

The formal context and starting point for developing the six LS is Garp3 [6, 8]. In terms of knowledge representation Garp3 is equivalent to LS6.

The simulations performed by the reasoning engine in Garp3 generate a state graph. The reasoning works as follows (Figure 2.6). The input for the simulation is a scenario and a library of model fragments. The scenario, which describes an initial situation in the system, is considered to be a partial state description. The *classification* task searches for model fragments for which the conditions hold. The consequences of those model fragments are introduced to the partial state description to generate an augmented state description. This representation contains all the structure (entities, configuration, etc.) and behavioural aspects of the system (quantities, causal dependencies, and correspondences), i.e. it incorporates all the consequences of the active model fragments.

The derivatives of the quantities in the augmented state description are still un-

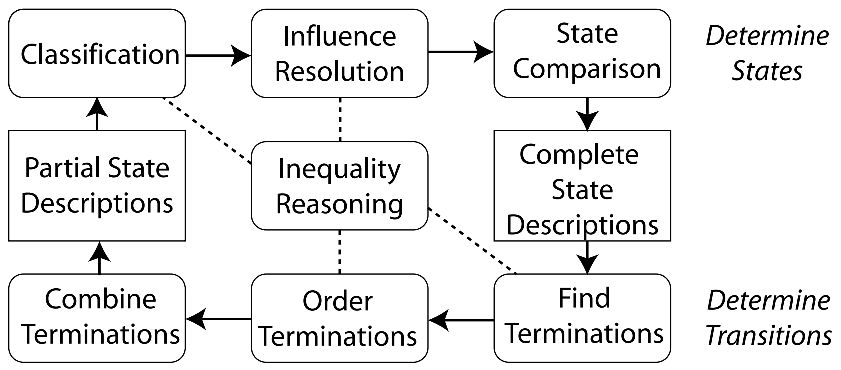

Figure 2.6: The workings of the reasoning engine. The dashed lines indicate that Classification, Influence Resolution, Find Terminations and Order Terminations all use the Inequality Reasoning component.

known. In the *influence resolution* step, a qualitative calculus is used to derive the derivatives of the quantities based on the causal relationships. This results in a potential *complete state description*. The *state comparison* step checks whether an equivalent state description already exists, in which case the two states are merged in the state graph. The result is a unique set of complete state descriptions.

Knowing the derivatives in the completed state description allows prediction of new states of behaviour. The *find terminations* step determines how current values of quantities can change (e.g. move to the next value). *Order terminations* determines which changes have precedence over others. Finally, the *combine terminations* step combines the possible changes to produce a new set of partial state descriptions. To generate the complete state graph the reasoning loop executed until no new states are found.

The reasoning engine uses its own internal representation, which is generated on the basis of the model representation. In order to use the Garp3 reasoner for the DynaLearn LS2-5, the functionality that generates this internal representation was adapted [27].

Chapter 3

Web Ontology Language (OWL)

OWL is a knowledge representation language based on description logics (DLs) and being developed as part of the semantic web initiative. This initiative proposes that making (web) content more machine-processable will simplify and improve search, improve interoperability among hetrogeneous systems, allow for service discovery and composition, allow automatic questions answering based on knowledge on the web, and improve information extraction [4, 21, 38]. Since its inception, OWL has become the de facto standard for developing ontologies [22]. OWL has a lot of desirable features: a large user community, multiple available reasoning engines, multiple programming language libraries to read, manipulate and save OWL files, and tools that allow non-experts to create OWL ontologies.

The main OWL primitives are classes, properties and instances[1]. Table 3.1 shows how these primitives can be used in DL notation. Instances represent occurrences of things, for example a particular animal. Properties are used to represent relationships between instances. For example, a particular animal living in a particular forest. Property definitions are organised in a hierarchy, meaning that if a property holds between two instances, the super-properties will also hold between those instances. Classes represent concepts, and can be seen as sets of instances. For example, the class Animal can be considered to be the set of all animals. Classes are also organised as hierarchies, meaning that being an instance of a class means that it is also an instance of its super-classes. OWL allows two ways to define classes: *extensional* and *intensional*. Intensional definitions define classes based on the properties their instances should fulfil, while extensional definitions indicate that a class consists of exactly a particular set of instances.

Intensional definitions in OWL are formalised as restrictions, which are sets of conditions. These restrictions can either represent *necessary* (N), or *necessary and sufficient* (N+S) conditions. Necessary conditions indicate that a class must fulfil certain conditions. Such conditions can be read as implication rules and use the subclass operator (e.g. $AntEater \sqsubseteq \forall preysOn.(Ant \sqcup Termite)$). In contrast, N+S conditions indicate that having certain conditions is enough to belong to that class (in addition

[1]Note that we use the term instance for what are called individuals in OWL.

to the requirement that instances of the class must fulfil certain conditions). As such, N+S conditions indicate equivalence, i.e. that two classes cover exactly the same sets of instances (e.g. $Predator \equiv \exists preysOn.LivingBeing$). N+S conditions allow instances to be classified as being instances of a particular classes, and classes as being subclasses of particular classes. Given the definitions in the examples (and assuming that ants or termites are formalised a being living things), the ant eater class is classified as being a subclass of the predator class.

In addition to classification, OWL reasoners can infer the inconsistency of classes and instances. A class is inconsistent when it cannot have instances, while an instance is inconsistent if it cannot exist. A class defined by the intersection between herbivores and carnivores is empty and is therefore inconsistent ($HerbivoreAndCarnivore \sqsubseteq \bot$). An ontology containing inconsistent classes or individuals is said to be inconsistent. Inconsistency informs the knowledge modeller that the formalised knowledge contradicts itself and should therefore be refined.

To discover inconsistencies in ontologies, it is important that modellers indicate the disjointness of classes, i.e. specify that the intersection between two classes is empty ($A \sqcap B \sqsubseteq \bot$). This means that there can be no instance which is an instance of both classes. This is required since the existence of two classes (without a disjointness statement) has 4 possible interpretations: either the classes have some overlap, no overlap, the first is a subclass of the second, or the second is a subclass of the first. Indicating disjointness (and subclass) relationships between classes contributes to OWL reasoners discovering inconsistencies.

Finally, a point about linking multiple ontologies. All OWL primitives (and definitions created using these primitives) are resources defined by a Uniform Resource Identifier (URI). It is possible to refer to definitions in other files by referring to such URIs. For example, to indicate that a certain resource is a class, it is required to refer to the owl:Class concept (e.g. $Population \sqsubseteq owl{:}Class$). This is possible since OWL is defined in terms of itself [2], making it possible to refer to OWL definitions via an URI. The owl: prefix indicates a namespace, which is a shorthand for the part of the URI.

$C_1 \sqsubseteq C_2$	C_1 is a subclass of C_2.
$R_1 \sqsubseteq R_2$	R_1 is a subproperty of R_2.
$C_1 \equiv C_2$	C_1 is equivalent to C_2.
$\neg C$	The complement of C.
$C_1 \sqcup C_2$	The union between classes C_1 and C_2.
$C_1 \sqcap C_2$	The intersection between classes C_1 and C_2.
$C_1 \sqcap C_2 = \bot$	C_1 and C_2 are disjoint.
$\forall R.C$	All fillers of relation R should be instances of C.
$\exists R.C$	At least one filler of relation R is an instance of C.
$\geq 3R$	There are at least three fillers for relation R.
$\geq 3R.C$	At least three fillers of type C for relation R
$\leq 1R$	Maximum 1 filler for relation R.
$o \in C_1$	Instance of type C_1.
$R_1 \equiv R_2^-$	R_2 is the inverse role of R_1.
$\langle o_1, o_2 \rangle \in R$	Instance o_1 is related to o_2 through an property R.
$\langle o, v_1 \rangle \in U$	Instance o is related through property U to value v_1.
$o_1 = o_2$	o_1 and o_2 are the same instances.
$o_1 \neq o_2$	o_1 and o_2 are different instances.
$C \equiv \{o_1, o_2, o_3\}$	Class C consists of exactly instances o_1, o_2, and $o3$.

Table 3.1: OWL Syntax

Chapter 4

An Ontological Perspective on QR

The word ontology is used in a number of different ways [35]. In this article, the emphasis is on the knowledge representation form in which concepts and relationships about a domain of discourse are formalised. Ontologies have a number of applications such as: natural language processing and text mining [11], linking large collections of (cultural heritage) resources to provide better indexing and search [32], establishing controlled vocabularies to facilitate knowledge management within communities [36], and capturing expert knowledge [20]. Particularly, the controlled vocabulary and capturing of expert knowledge applications are similar to the goals of the QR community.

By applying an ontological perspective on QR we aim to come to a structure in which QR models and simulations can be represented. On the other hand, the ontology field is challenged on its ability to accommodate the representations used in QR models. QR models are well-known for their advanced and detailed representations [13] and one concern is whether the available ontology languages are rich enough to encompass these. Furthermore, we investigate whether the reasoning methods used in the ontology field can be used to solve some of the QR reasoning tasks, including a form of classification. At the same time it also highlights additional reasoning competence that is required by complex knowledge systems (such as in the QR field) that currently are not well covered by approaches such as ontology reasoning. The latter may be informative for setting the ontology research agenda.

An earlier effort to support the interchange and reuse of QR models is the creation of the Compositional Modelling Language (CML) [5]. CML is a unified representation language for the different QR model paradigms, and is expressed in the Knowledge Interchange Format [18]. Formalising DynaLearn models in CML would support the interchange of models between the different DynaLearn components, by furthering the goal of establishing a of a common vocabulary for QR models and simulations. However, the representation would not allow easy processing of the QR knowledge structures, nor would reasoning with such structures be facilitated, since there are no KIF interpreters readily available. For these reasons, we have chosen not to formalise

DynaLearn knowledge assets in CML.

Instead of CML, we have chosen the Web Ontology Language (OWL) [2] (Section 3) as our knowledge representation language. OWL was developed as part of the semantic web initiative [4] to make content more computer accessible. OWL is a knowledge representation language based on description logics [1], and is usually written in an XML format. OWL has become the de-facto standard for the sharing of knowledge on the web in the form of ontologies [22]. By formalising the QR models in OWL, knowledge structures can be easily exchanged between components, interpreted using existing libraries, and be reasoned with using reasoning engines for OWL. It becomes possible to open the QR models in several well-established applications such as ontology editors (Protégé [25], SWOOP [24], Triple20 [40]), checked for consistency and applied classification reasoning on using reasoners (FACT++ [37], Pellet [34] and RacerPro (Racer Systems GmbH & Co. KG, http://www.racer-systems.com), and manipulated through using API's (the OWL API [3], and the SWI-Prolog Semantic Web Library [41]).

An additional benefit is that validity (to the OWL formalism) and consistency of the models (logical consistency) can be checked using OWL validators and OWL reasoners. Describing QR models in an established standard should also make it easier to develop import functionality in other QR tools.

To determine how the QR models can be formalised as ontologies, an ontological perspective on QR is taken. Previous research distinguishes different types of ontologies based on the type of ontological commitments they make [39]. For example, the ontological commitments of a knowledge representation language consist of the concepts that can be used as ingredients in the language. However, a domain model created by a knowledge engineer using such a language defines new concepts based the concepts in the knowledge representation language. These are are a different kind of ontological commitments, since knowledge representation language tries to define a domain-independent perspective free language, while the domain model defines a domain-specific vocabulary. We frame the QR knowledge representation on these different types of ontologies (Figure 4.1).

Since we chose OWL as our representation language for QR models, OWL becomes our representational ontology, and defines the concepts that we can use for our formalisation. The model ingredients provided in the QR vocabulary (Section 5) are formalised as an OWL ontology, called the *DynaLearn Ontology*. These model ingredients are *generic building blocks* such as the concept entity, causal relations, and inequalities. The DynaLearn QR ontology is a generic ontology that extends the ontological commitments made by OWL (i.e. defines new concepts based on the concepts provided by representational ontology). This generic ontology is domain-independent.

When modellers create QR models, they extend the QR vocabulary by defining domain specific model ingredients, called *domain building blocks*, such as entities, configurations, and quantities. Creating such a domain specific vocabulary can be seen as refining some of the generic building blocks in the generic ontology to define a domain ontology. In that sense, the QR models can be defined in OWL by extending the ontological commitments of the DynaLearn QR ontology for a specific domain. This ontology is called a *QR model ontology*.

The generic building blocks in the DynaLearn QR ontology, and the domain building blocks in the QR model ontology can be used to define the aggregates (model fragments and scenarios) that represent specific situations and processes. These aggregates are part of the domain ontology.

Ideally, the application specific information in a QR model (e.g. for visualisation) is stored in an application ontology separate from the domain ontology. However, from a user perspective it is more convenient to work with a single model file. Therefore, some application specific information has to be stored in the QR model ontology.

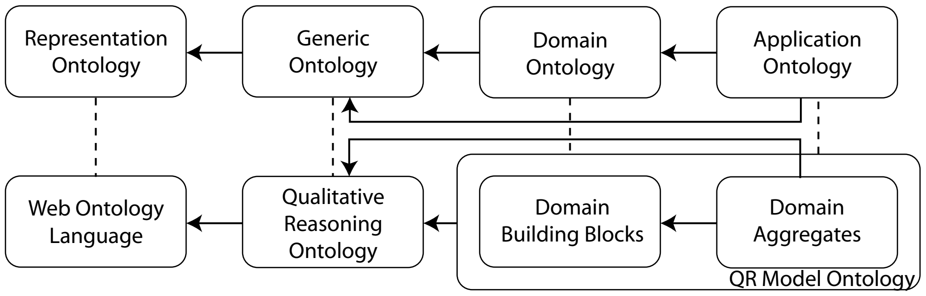

Figure 4.1: Correspondences between the QR ontologies and ontology types based on the type of ontological commitments made.

4.1 Design and Validation

We followed a step-wise approach. First, we manually created the *DynaLearn QR ontology* in which all model ingredient types in the QR vocabulary (i.e. the terminology used when talking about QR models) are defined. The details for this were derived from the DynaLearn workbench, Garp3, the Garp3 documentation, articles about QR, and discussions with experts. By running an OWL reasoner the consistency of the ontology was tested. Inconsistent concepts were refined until a consistent ontology was established that covered the entire QR vocabulary. This vocabulary was then used as the basis for the formalisation of QR models in OWL.

The main idea of creating the DynaLearn QR ontology is to restrict how model ontologies can be formalised. This is done by specifying necessary restrictions for each of the model ingredients. Consequently, QR models in OWL that are invalid from a QR point of view are inferred to be inconsistent. This helps to determine the structure of the OWL model format, and makes it possible to validate the model ontologies.

The second step was to manually define a set of simple QR models in OWL. The concepts in these domain ontologies refer to concepts defined in the QR vocabulary. These model ontologies were validated by searching for inconsistencies using OWL reasoners. Inconsistent ontologies were either to blame on the definitions in the generic ontology (which were then refined), or on definitions in the hand-made model ontologies.

Thirdly, the OWL export code was implemented. To validate the correctness of the resulting OWL code (and the vocabulary) increasingly complex models were exported to OWL and checked for inconsistencies using OWL reasoners (Section 4.2). Inconsistencies were resolved by refining both the vocabulary and the OWL export code.

Finally, the OWL import code was implemented. This made it possible to further validate the QR ontology defined in OWL, and the accompanying export and import code, by comparing the original QR model (and its simulation results) with the exported-and-imported version of that model. These two should be identical.

4.2 Implementation

Several OWL editors and reasoners have been used to test the validity of the QR ontology and the QR models represented in OWL: Protégé 3 and 4 [25], SWOOP [24] and Triple20 [40]. The following reasoners have been used to test for consistency and to do classification: FACT++ [37], Pellet [34] and RacerPro (Racer Systems GmbH & Co. KG[1]).

The OWL import and export functionality was implemented using the SWI-Prolog Semantic Web Library [41]. Notice that in the OWL file format only the model itself is represented, as the simulations can be recreated using the software. However, functionality was added to export summaries of simulations (Section 7).

[1]http://www.racer-systems.com

Chapter 5

The DynaLearn QR Ontology

This chapter discusses the formalisation of the QR vocabulary used in the DynaLearn ILE. It provides the basis of the formalisation of QR models in OWL (Section 6). It presents both the generic ontology of QR model ingredients, which introduces the necessary vocabulary in terms of OWL, and the domain ontologies (Section 4), which can be used to capture domain specific QR models. The basis of the DynaLearn QR ontology is LS6, since it uses the most complete QR vocabulary. The formalisations of the other learning spaces is derived from the representation of LS6 models (Section 6). For clarity, the vocabulary used in the following chapter is as follows:

QR vocabulary The terminology that is used to talk about QR models. That is, all the possible model ingredients, the categorizations of these model ingredients, the different views on simulation results, such as the state graph, states, transitions, value history etc. Note that domain specific terminology is not part of the QR vocabulary.

(DynaLearn) QR Ontology The formalisation of the QR vocabulary in OWL.

QR Models in OWL The formalisation of the QR models in OWL based on the DynaLearn QR Ontology. These QR models use domain specific terms, but refer to the QR Ontology to indicate of which type specific terms are.

5.1 A Hierarchy of Qualitative Model Ingredients

5.1.1 The Hierarchy

The formalisation of the QR domain starts with the ordering of the vocabulary in a class hierarchy. Figure 5.1 shows the taxonomy of the QR model ingredients. The taxonomy shows both the QR concepts and relations, as the latter are reified (treated as classes, see section 5.1.2). The top node of is called *QualitativeModelIngredient*, as every class is a possible ingredient of a qualitative model. The model ingredients are divided into the sets *BuildingBlock* and *Aggregate*. The former describes separate model ingredients, while the latter describes collections of related model ingredients. The aggregate concepts *ModelFragment* and *Scenario* are described in section 5.6. As mentioned in chapter 2, qualitative models describe both the structural and the behavioural aspects of systems. Therefore, the building blocks are subdivided into the sets *Structural* (section 5.3), *Behavioural* (section 5.5), and *AssumptionType* (section 5.4). Assumptions are considered to be separate, as they do not describe inherent aspects of the system.

One of the most difficult problems when developing an ontology is finding proper names for the defined concepts. Whenever a vocabulary concept is described in this report, its position in the hierarchy will be discussed along with the naming (if it differs from the standard QR vocabulary). The formalisation of the hierarchy in OWL, however, is straight forward. The classes are defined by giving them an id, label, comment, and specifying their superclasses. As OWL assumes that the sets which the classes model overlap, the siblings on each level have been specified as being disjoint. In the rest of this chapter, the definitions of the concepts are extended by adding restrictions.

5.1.2 Reification of Relations

All relations in the QR vocabulary have been reificated. This was necessary, as some QR relation ingredients are tertiary in nature. Furthermore, in order to render models in a graphical model editor, the position on the screen of each model ingredient must be stored. Treating the relations as classes (reification) allows instances of those relations to be connected to multiple objects, and have data type properties (for the position information).

The reification of relations in OWL is a familiar problem and can be solved by using existing reification patterns [29]. There are two ways to reify a relation depending on the objects that take part in the relation. Generally, there is at least one independent object participating in a relation. An independent object exists without any other objects, while a dependent object, such as a property value, needs another object to exist. For the first pattern (Figure 5.2) to apply, there must be only one independent object, which is the owner of the relation. If there is more than one independent object taking part in the relation, and there is no clear distinguishable owner, the second pattern is suggested to be used (Figure 5.3).

The single independent object in the first pattern (Figure 5.2) must be the owner of the relation, as the dependent objects would not exist without it. Relations with only one independent object usually model properties of classes. Properties should have only one dependent object which is the value. For this reason, in this report relations from a reified relation to the dependent objects are defined as having cardinality 1. The standard pattern mentioned before uses *functional* properties instead of a cardinality restriction. The difference is that a cardinality restriction restricts the number of relations to 1, while a *functional* property indicates that all values of the property are the same individual. So, a cardinality imposes a restriction, while making the relation *functional* allows the reasoner to make an inference. For the QR domain the restriction is more appropriate, as it decreases the number of possible models with the same meaning. Furthermore, it is also possible for a reified relation to have an optional dependent object as a value. This restriction can be formalised by using *maxCardinality*, instead of *cardinality=1*. The second existing reification pattern applies when the relation has multiple independent objects as arguments, instead of just one. In this case, there is no

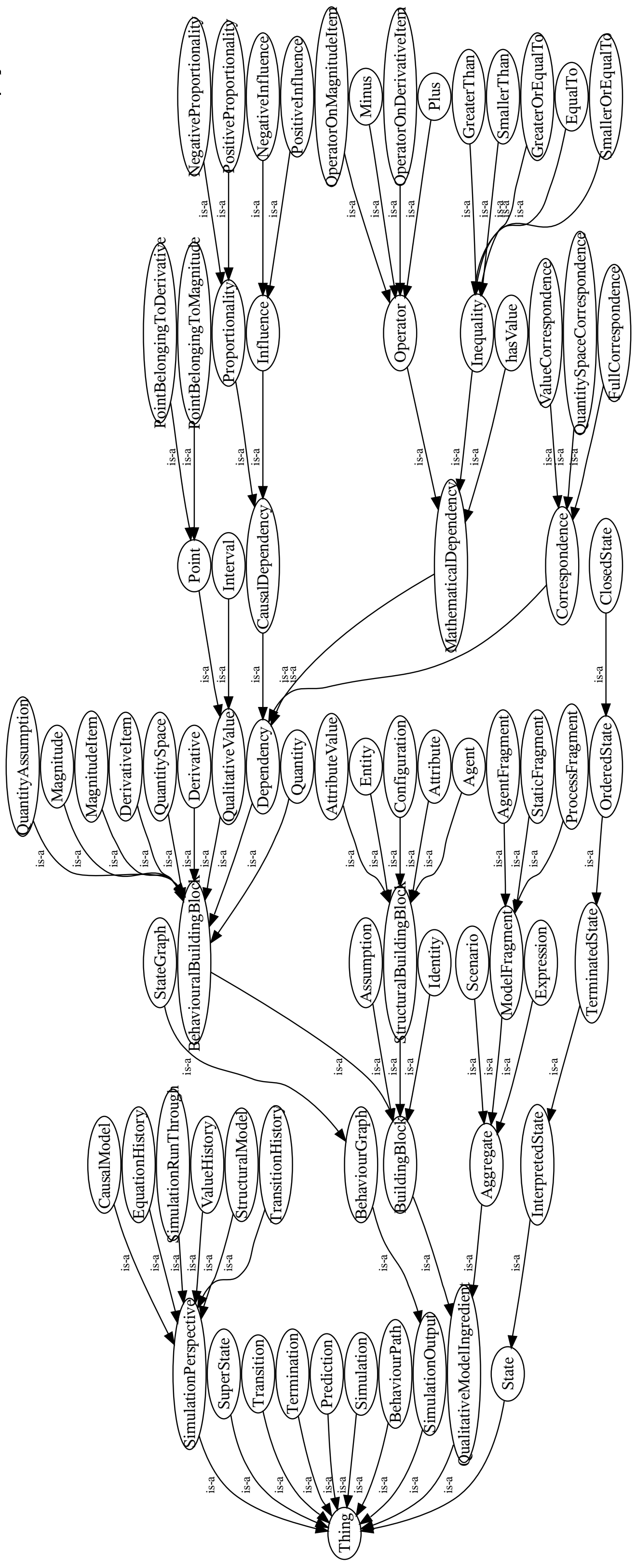

Figure 5.1: The QR ingredient taxonomy defining the QR vocabulary.

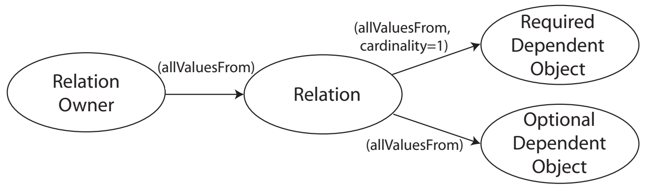
Relation Owner
(allValuesFrom)
Relation
(allValuesFrom, cardinality=1)
Required Dependent Object
(allValuesFrom)
Optional Dependent Object

Figure 5.2: Relation Reification Pattern 1: One independent object which is the owner of the relation.

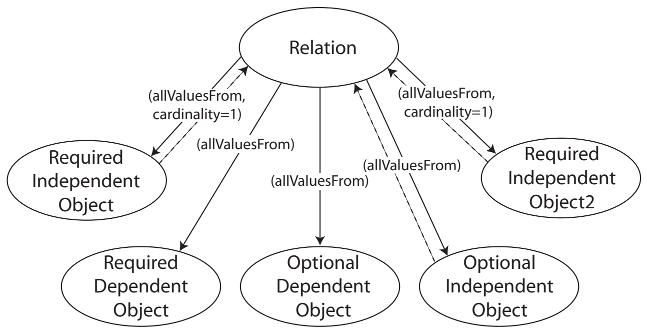
Relation
(allValuesFrom, cardinality=1)
(allValuesFrom)
(allValuesFrom)
(allValuesFrom)
(allValuesFrom, cardinality=1)
Required Independent Object
Required Independent Object2
Required Dependent Object
Optional Dependent Object
Optional Independent Object

Figure 5.3: Relation Reification Pattern 2: Multiple independent objects with no clear owner. Note that the restrictions apply to the relations from the reificated relation.

clear owner of the relationship (see Figure 5.3). For this reason, relations are modelled from the reified relation to the arguments. The inverse relations of the independent object should be defined, as the relations these objects take part in should be deducible. This second pattern is not applied in the QR ontologies as restriction semantics are not defined in the relation, but the class which owns the relation (see section 5.1.3). As this requires the ownership relations to point to the reificated relation, the first pattern is more appropriate.

5.1.3 Property Restrictions

The defined QR hierarchy specifies that classes inherit the properties defined in their superclasses. In this case the properties are the restrictions which specify how model ingredients may be related using object properties. This section describes how such restrictions have to be formalised. For brevity, the restrictions used in this chapter are formalised in the logical notation shown in Table 3.1.

A common mistake made when modelling properties in OWL is using domain and range definitions of properties as restrictions [31]. These fields specify that the domain and range of the property *are* of a specific type. This means that if a relation *playsInstrument* with domain *Person* and range *MusicalInstrument* is used to model a robot playing piano, the statement is not inconsistent. It merely allows the reasoner to infer that the robot is a person. OWL users are advised not to specify these fields, as the resulting ontologies are very hard to debug [31].

The correct way to restrict the use of relations is by specifying the restrictions in the class owning the relation. Examples of the specification of such restrictions have been shown in chapter 3. These restrictions become more complex when relations have been reificated, as it becomes possible to model the restrictions directly in the relation. This should not be done, as it makes it impossible to reuse the relation. If the *playsInstrument* relation from the previous paragraph would be reificated, and the domain and range restricted, the statement that the robot plays piano would become inconsistent.

$$playsInstrument \sqsubseteq \forall\ hasPlayer\ Human$$
$$playsInstrument \sqsubseteq \forall\ hasInstrument\ Instrument$$

In order to be able to reuse reificated patterns a new pattern was developed. Instead of modelling the restrictions in reificated relations, they should be defined in the classes using the relation. This is possible as OWL allows the creation of new anonymous class definitions within restrictions. In the robot pianist example, the person class would be restricted to having a relation with a reificated *playsInstrument* relation, which has a relation with an instrument. This definition allows reuse, as now the robot class can be formalised using the same restrictions. It is even possible to allow different instruments for the robot by replacing the Instrument class in the restriction.

$$Human \sqsubseteq \forall\ hasPlaysInstrumentRelation(playsInstrument\ \sqcap$$
$$(\forall\ hasInstrument\ Instrument))$$
$$Robot \sqsubseteq \forall\ hasPlaysInstrumentRelation(playsInstrument\ \sqcap$$
$$(\forall\ hasInstrument\ Instrument))$$

5.2 Qualitative Model Ingredient

The qualitative model ingredients used in model fragments and scenarios have to be visualised. Therefore, the concept has the two data type properties *has_xposition_on_screen* and *has_xposition_on_screen*. As not every model ingredient has a specific position which needs to be stored (because the position is indicated by other ingredients), filling these values is not always necessary. Therefore, instead of cardinality restrictions, maximum cardinality restrictions are used.

$$Qualitative\ Model\ Ingredient \sqsubseteq has\_xposition\_on\_screen\ \leq\ 1$$
$$Qualitative\ Model\ Ingredient \sqsubseteq has\_yposition\_on\_screen\ \leq\ 1$$

5.3 Structural

This section describes structural ingredients set of the qualitative model ingredients, which are used to describe the structure of a system. This class encompasses the concepts (1) *entities*, (2) *agents*, (3) *configurations*, (4) *attributes* and (5) *attribute values*.

5.3.1 Entities, Agents, and their Relations

Entities describe the objects which exist in a system. Agents are very similar in the kind of relations they can take part in, but are used to model 'outside forces' on the system. They are both structural ingredients, and are alike in the kinds of relations they can take part in. It is possible for entities and agents to have attributes (section 5.3.3) and quantities (section 5.5.1) as is shown in Figure 5.4. These necessary restrictions are shown below. Entities and agents can have configuration relations with other entities and agents, which are described in section 5.3.2. The formalisation below applies to the Agent concept as well as the Entity class.

$$Entity \sqsubseteq \forall\ hasAttribute\ Attribute$$
$$Entity \sqsubseteq \forall\ hasQuantity\ Quantity$$

Developers of qualitative models can define their own entities and agents in a subtype hierarchy. These entities and agent definitions are stored in the *domain ontology*. The formalisation of these hierarchies in OWL is similar to the formalisation of the QR vocabulary hierarchy (section 5.1.1). Again, the classes are defined with their respective superclasses, and disjointness axioms have to be added to indicate that individuals of classes cannot be members of the sibling classes. The top nodes of these hierarchies are the general *entity* and *agent* concepts defined in the *generic ontology*. Therefore, every model imports the generic ontology, and refers to these concepts though the correct namespace.

5.3.2 Configurations

Configurations are used to describe the structural relations between entities and agents. A configuration is a relation between two independent objects, but it is a directed relation. This means that the owner of the relation can be clearly identified. This means that both the first (a clear owner of the relation can be identified) and the second relation reification pattern (participants in the relation are independent) could apply. As mentioned in section 5.1.2, the second reification patterns is not used. Therefore the configuration is modelled as a mixture of the two patterns, as is shown in Figure 5.4. The difference with the first pattern is that the inverse relations have been defined, so the configurations which entities and agents take part in can be derived.

Configurations have exactly one owner and one target, therefore the *hasConfigurationTarget* and *hasConfigurationOwner* relations are defined as having cardinality 1.

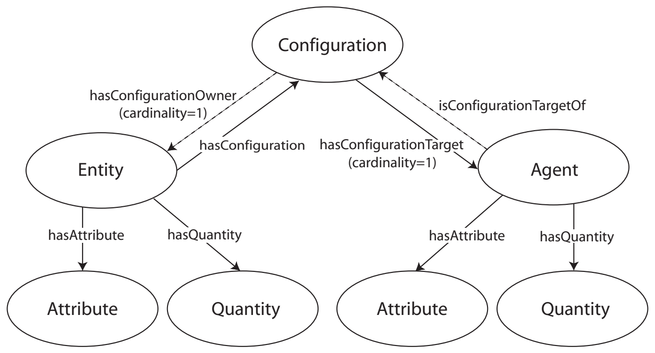

Figure 5.4: The possible relations of *entities* and *agents*. Note that both the entity may be changed to agent, and the agent into entity.

By defining inverse relations the owner belonging to the configuration, and the configuration which targets an entity or agent can be derived.

$$Configuration \sqsubseteq hasConfigurationOwner\ =\ 1$$
$$Configuration \sqsubseteq hasConfigurationTarget\ =\ 1$$

As mentioned in section 5.1.3, the restrictions which apply to relations are defined in the class which utilises them. In this case the semantics of the configuration relation are defined in the agent and entity classes. This allows new model ingredients to be defined which also use the configuration relation. Entities and agents can participate in configuration relations with other entities and agents. The formalisation for entities below also applies to the Agent class.

$$Entity \sqsubseteq \forall\ hasConfiguration\ (Configuration\ \sqcap$$
$$(\forall\ hasConfigurationTarget\ (Entity\ \sqcup\ Agent)))$$

Developers of qualitative models can define their own configurations. In contrast with entities and agent, configurations are not arranged in a taxonomy. As a result every configuration is a subclass of the concept *configuration* defined in the generic ontology.

5.3.3 Attributes

Attributes describe the features of entities and agents that do not change gradually. Models can define their own attributes, which consist of an attribute name, and its possible values. Attributes are considered to be properties of entities and agents, and should have exactly one attribute value. As OWL does not make a distinction between relations between objects and properties of objects[1], a pattern has to be used

[1]Not to be confused with ObjectProperties in OWL

to formalise properties in OWL. An existing pattern to model property values uses an enumeration of individuals [30].

Property Values as an enumeration of individuals In the enumeration pattern the values of a property are considered to be a set of individuals, as shown in Figure 5.5. As a result, the values are unique in the ontology, and objects with the property all refer to a value from the same set of individuals. It is necessary to explicitly state that the values are different using the *owl:differentFrom* statement, as OWL does not make the Unique Name Assumption (two individuals with different names are not necessarily different objects).

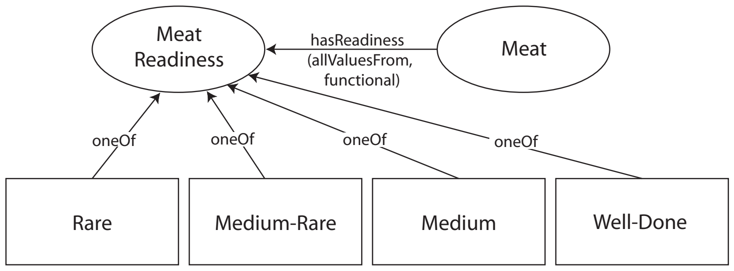

Figure 5.5: Values as an enumeration of individuals.

Back to the formalisation of attributes. The attribute value is defined to be an enumeration of individuals (see Figure 5.6). This allows the same value set to be used for each instance of an attribute. This formalisation works for attribute values, since these instances do not have relationships themselves. As will be shown in Section 5.5.2, this pattern does not work for qualitative values, since they do have relationships with other instances (e.g. inequalities).

Attribute relations are a typical example of reification pattern 1. The attribute itself is an independent object, while the attribute values are dependent on the existence of the attribute. Since an attribute always has a relation with an attribute value, the semantics are stored in the attribute class instead of in the classes having attributes.

$$Attribute \sqsubseteq \forall\ hasAttributeValue\ AttributeValue$$
$$Attribute \sqsubseteq \forall\ hasAttributeValue\ =\ 1$$

The attributes defined in the domain ontology have some further restrictions. The attribute is prohibited to have any other attribute value, other than an instance of the associated attribute value class. The attribute value class is defined as the enumeration of the possible values.

$$OpenOrClosed \sqsubseteq \forall\ hasAttributeValue\ OpenOrClosedValue$$
$$OpenOrClosedValue \equiv \{Open, Closed\}$$

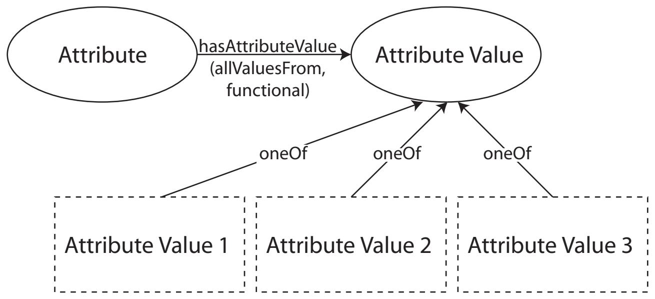

Figure 5.6: The attribute components and its relations.

5.4 Assumption Types

As it is likely that new types of assumptions will be introduced in the QR vocabulary, the *AssumptionType* concept is defined. For now, the only class which is part of this set is the generic *assumption*. Assumptions are neither structural nor behavioural ingredients, as they describe knowledge about the model, and not about the system.

Assumptions are used as conditions in model fragments to assert that something is true. They are usually used in combination with inequalities to reduce the possible behaviour of a system. As assumptions have no further relations with other QR ingredients, no restrictions have to be imposed in the ontology. Model builders can define their own assumptions in the same way they can define entities and agents. The formalisation of this subsumption hierarchy in OWL is equivalent to those of the entities and agents.

5.5 Behavioural

Behavioural ingredients describe the model ingredients used to describe the behaviour of a system. They include: (1) *magnitudes*, (2) *derivatives*, (3) *quantities*, (4) *quantity spaces*, (5) *qualitative values*, and (6) *dependencies*. Qualitative values can be further divided into *points* and *intervals*.

Dependencies are the possible relations between the behavioural ingredients of a system. They are used to model the processes causing change, constrain what changes can occur, and how these changes occur. The dependency ingredients include: (1) the *causal dependencies: proportionality* and *influence*, (2) *correspondences*, and (3) the *mathematical operators: plus/minus* and all *(in)equalities*.

5.5.1 Quantities, Magnitudes, Derivatives and Quantity Spaces

Quantities describe the changeable features of entities and agents. These behavioural ingredients consist of exactly one magnitude and exactly one derivative (see Figure 5.7). Both the magnitude and the derivative have exactly one quantity space.

The magnitude is actually the zero derivative of the quantity, while the derivative is the first derivative. Forbus uses the term *amount* to refer to the zero derivative, and

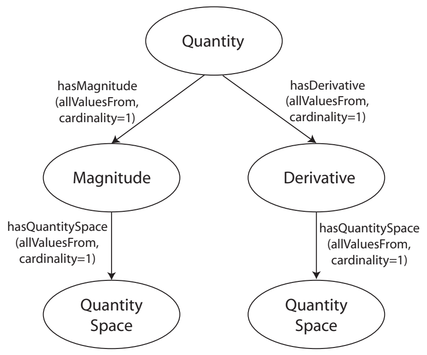
Quantity
hasMagnitude (allValuesFrom, cardinality=1)
hasDerivative (allValuesFrom, cardinality=1)
Magnitude
Derivative
hasQuantitySpace (allValuesFrom, cardinality=1)
hasQuantitySpace (allValuesFrom, cardinality=1)
Quantity Space
Quantity Space

Figure 5.7: The quantity has one magnitude and one derivative, which both have quantity spaces.

magnitude to refer to the value of the magnitude or derivative [14]. The term amount is not used in our approach, because it it has multiple meanings. Next to Forbus' use, it could also be used to refer to the quantity "amount". Another alternative is the term value, but that word has the problem that the qualitative values within a quantity space are also referred to as values. Magnitude does not have these problems, and is an appropriate name in this context. Quantities can have proportionality and influence relations (section 5.5.3). Magnitudes and Derivatives can participate in (in)equality relations (section 5.5.5) and Plus/Minus relations (section 5.5.6).

$$
\begin{aligned}
&Quantity \sqsubseteq \forall\, hasMagnitude\; Magnitude\\
&Quantity \sqsubseteq hasMagnitude\; = \;1\\
&Quantity \sqsubseteq \forall\, hasDerivative\; Derivative\\
&Quantity \sqsubseteq hasDerivate\; = \;1\\
\\
&Magnitude \sqsubseteq \forall\, hasQuantitySpace\; QuantitySpace\\
&Magnitude \sqsubseteq hasQuantitySpace\; = \;1\\
\\
&Derivative \sqsubseteq \forall\, hasQuantitySpace\; QuantitySpace\\
&Derivative \sqsubseteq hasQuantitySpace\; = \;1
\end{aligned}
$$

Qualitative model builders can define their own quantities. A quantity can have a number of allowed quantity spaces (section 5.5.2). This information has to be stored in the specific quantity concept in the domain ontology. The formalisation in OWL is done using a restriction. The quantity has a magnitude, which has a quantity space relation with one of the quantity spaces in a union.

$$\begin{aligned}&Flow \sqsubseteq \exists\, hasMagnitude(Magnitude\ \sqcap\\&\quad(\forall\, hasQuantitySpace(Minimumnegativezeropositivemaximum\ \sqcup\\&\quad Negativezeropositive\ \sqcup\ Zeropositivemaximum)))\end{aligned}$$

5.5.2 Quantity Spaces and Qualitative Values

The quantity space is a behavioural ingredient which defines the possible qualitative values a quantity can have. A quantity space consists of at least one qualitative value. These values are also behavioural ingredients, and can be either a point or an interval. The quantity space describes a total order, meaning that a magnitude or derivative can only change to a value directly above or below its current value. Quantity spaces can participate in a correspondence relations (section 5.5.4).

$$\begin{aligned}&QuantitySpace \sqsubseteq \forall\, hasQualitativeValue\ QualitativeValue\\&hasQualitativeValue\ \geq\ 1\end{aligned}$$

As each qualitative value in a model fragment can participate in (in)equality relations, it is impossible to formalise them as an enumeration of individuals. It is more graceful to model each value which can participate in a relation as a separate individual. Therefore, “the property values as a set of individuals” pattern cannot be used. Another existing pattern which is more appropriate formalises each value as a class [30].

Property Values as classes Values of properties can be thought of as a set of subclasses forming a parent class (see Figure 5.8). This class binds all the possible value classes of the property using the *owl:unionOf* construct. It is necessary to explicitly state that the subclasses are disjoint, as otherwise a property could have two values at the same time (because they are the same, i.e. the value is an instance of both subclasses). In contrast with the ’property values as individuals’ pattern, values are not unique, but a new instance of the value is created for each property instance. A problem with this pattern is that the property values do not have an explicit order, which is needed for quantity spaces. An existing pattern which might solve this problem is to model the values as a sequence in a list.

Property Values as a Sequence in a List This pattern is described in the n-ary relations document of the Semantic Web Working Group [29]. If the possible values of a property have a strict sequence, the previous patterns are not expressive enough, as they do not enforce an ordering. A possible solution to this problem is to model the values as a list. Unfortunately, using *rdf:List* in an OWL ontology causes it to become OWL Full. A solution is to model a list in OWL (see Figure 5.9), which is then used to store values.

A list consists of a number of arguments, each pointing to the next item in the list. Each argument item has as *has_content* relation with the an object. The list pattern can

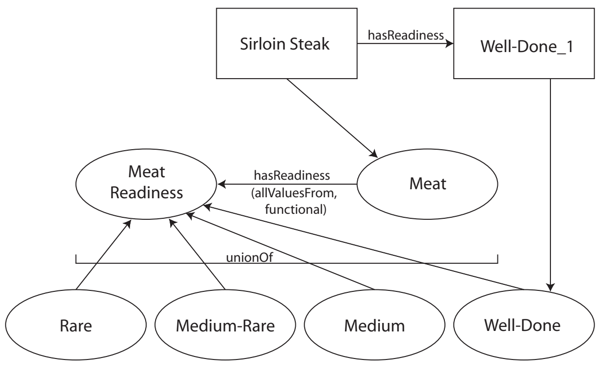

Figure 5.8: Property values as the instances of classes which form a union.

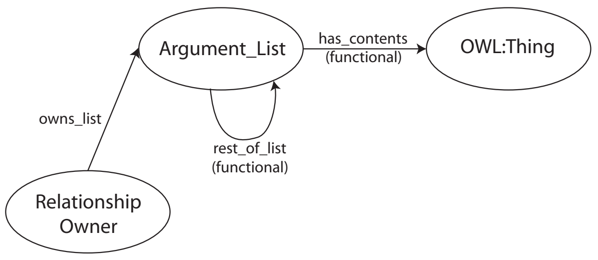

Figure 5.9: The definition of a list in OWL.

be used in combination with both the "property values as an enumeration of individuals", and the "property values as a set of classes" patterns. To formalise a quantity space the latter pattern must be used. The values are ordered in a list to model their sequence, as is shown in Figure 5.10.

However, there are two problems with the list pattern. Firstly, a list has no ontological meaning, as it is a data structure. A list with the cities Amsterdam, Brussel, and Paris has little meaning. They could indicate a travel route, cities with the around the same amount of inhabitants, or something else entirely. Secondly, it is awkward to determine the owner of a list from one of the possible values. One has to "reason through" all the values. In order to solve these problems a new pattern had to be developed; the ontological sequence.

Ontological Sequence In order to formalise a quantity space the ordering of the values has to be made explicit. The quantity space is connected with its points and intervals using *containsQualitativeValue* relations, as is shown in Figure 5.11. The ordering is established by using inequalities (section 5.5.5). Each consecutive value in the order must have another type than the previous one. For that reason, the (in)equality restrictions are formalised in the intervals. This still allows connecting two points, but that is

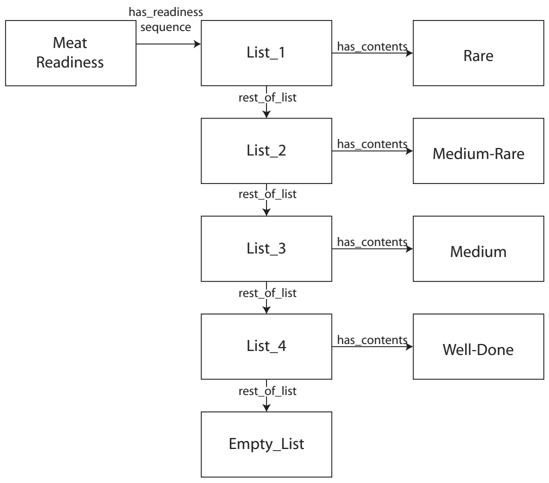

Figure 5.10: The application of a list in OWL.

necessary for the other (in)equality relations (section 5.5.5). The point *Zero* is universal among quantity spaces, and is therefore defined in the generic ontology. Qualitative model builders can specify their own quantity spaces. These have the form shown in Figure 5.11.

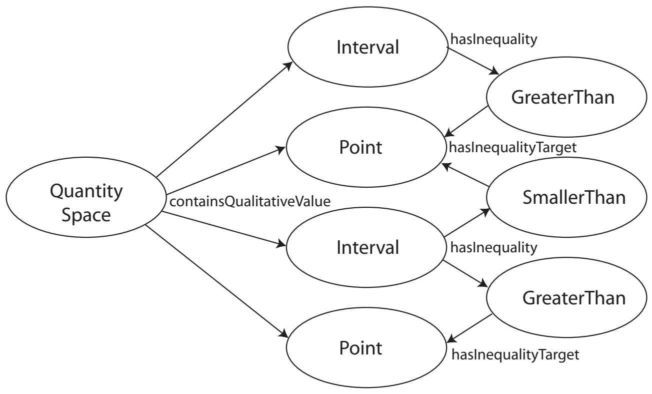

Figure 5.11: The formalisation of a quantity space and its values using inequalities.

$$Quantity \sqsubseteq \forall\, containsQualitativeValue\ QualitativeValue$$
$$Quantity \sqsubseteq containsQualitativeValue\ \geq\ 1$$

$$NegativeZeroPositive \sqsubseteq \exists\, containsQualitativeValue(Positive \sqcap$$
$$(\exists\, hasInequality(GreaterThan\ \sqcap\ (hasInequalityTarget\ \ni\ Zero))))$$
$$NegativeZeroPositive \sqsubseteq \exists\, containsQualitativeValue(Negative\ \sqcap$$
$$(\exists\, hasInequality(SmallerThan\ \sqcap\ (hasInequalityTarget\ \ni\ Zero))))$$
$$NegativeZeroPositive \sqsubseteq containsQualitativeValue\ \ni\ Zero$$

5.5.3 Proportionalities and Influences

Proportionalities and Influences are relations between quantities which indicate what quantities in a model change. These relations can be either positive or negative. A positive proportionality indicates that the derivative of the target quantity is positive if the derivative of the origin quantity is positive, and is negative if the derivative of the origin quantity is negative. For a negative proportionality this is just the opposite. A positive influence indicates that the target quantity derivative is positive if the magnitude of the origin quantity is greater than zero, and negative if it less than zero (if it is the only influence on the target quantity). For the negative influence this is just the other way around.

The semantics of the causal dependencies are modelled in the owner class, in this case the quantity class. Quantities can have influences and proportionalities as causal dependency relations, and they must have exactly one quantity as a target (see Figure 5.12).

$$Quantity \sqsubseteq \forall\, hasCausalDependency($$
$$(Influence\ \sqcup\ Proportionality) \sqcap$$
$$(\forall\, hasCausalDependencyTarget\ Quantity)\ \sqcap$$
$$(hasCausalDependencyTarget\ =\ 1))$$

5.5.4 Correspondences

Correspondences specify that values occur simultaneously. They normally occur between qualitative values of different quantity spaces, but it is also possible to create a correspondence relation between quantity spaces themselves. Those relations are an abbreviation for value correspondence relations between each of the quantity values of the different quantity spaces. Directed and undirected correspondences are distinguished. The former states that the target value may be derived from the origin value, while the latter allows the derivation of the value in both directions.

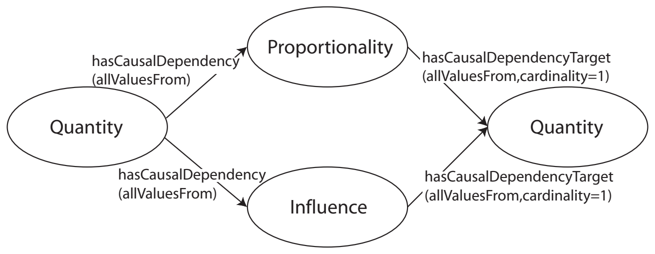
Proportionality
hasCausalDependency (allValuesFrom)
hasCausalDependencyTarget (allValuesFrom,cardinality=1)
Quantity
Quantity
hasCausalDependency (allValuesFrom)
hasCausalDependencyTarget (allValuesFrom,cardinality=1)
Influence

Figure 5.12: The use of proportionality and influences relations. Note that the restrictions applied to the hasCausalDependencyTarget relation are modelled in the Quantity class which owns the relation, and not in the influence or proportionality class.

The semantics of quantity space correspondences is unclear when the quantity spaces have a different amount of values, as it is unknown between which values value-correspondences would exist. The only way to formalise this in OWL is to create classes for each quantity space size, and restrict correspondences to members with the same number of qualitative values. The correct formalisation would take an infinite number of classes, therefore this restriction is not implemented. It would be beneficial if it would be possible to express in OWL that a domain and range should have the same cardinality. Another problem is that correspondences, just as configurations, should not be reflexive. It is a known problem that this restriction is inexpressible in OWL. For value correspondences a restriction is needed that restricts value correspondences to members of different quantity spaces. As OWL restrictions can only state that a range has to be of a specific type, this is impossible to implement. This leaves the basic restrictions that correspondences should be between members of the same class, and should have exactly one target, as is shown in Figure 5.13.

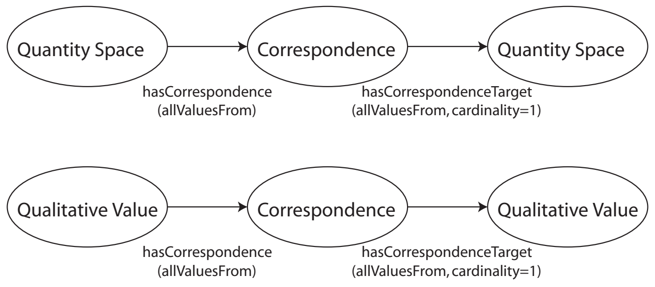
Quantity Space
Correspondence
Quantity Space
hasCorrespondence (allValuesFrom)
hasCorrespondenceTarget (allValuesFrom, cardinality=1)
Qualitative Value
Correspondence
Qualitative Value
hasCorrespondence (allValuesFrom)
hasCorrespondenceTarget (allValuesFrom, cardinality=1)

Figure 5.13: The only valid correspondence relations are between two quantity spaces, or two qualitative values.

$$
\begin{aligned}
&QualitativeValue \sqsubseteq \forall\ hasCorrespondence(ValueCorrespondence \sqcap \\
&\quad (\forall\ hasCorrespondenceTarget\ QualitativeValue)\ \sqcap \\
&\quad (hasCausalCorrespondenceTarget\ =\ 1))
\end{aligned}
$$

$$
\begin{aligned}
&QuantitySpace \sqsubseteq \forall\ hasCorrespondence(QuantitySpaceCorrespondence \sqcap \\
&\quad (\forall\ hasCorrespondenceTarget\ QuantitySpace)\ \sqcap \\
&\quad (hasCausalCorrespondenceTarget\ =\ 1))
\end{aligned}
$$

5.5.5 Inequalities

Inequalities are dependencies which can be used by a lot of ingredients. They indicate the difference between, or equality of values. The same reificated (in)equality relations *SmallerThan*, *SmallerOrEqualTo*,*EqualTo*,*GreaterOrEqualTo* and *GreaterThan* are reused for each possible domain/range combination. This reuse is possible as the restrictions are formalised in the classes owning the relations, as was discussed in section 5.1.3.

The first (in)equality relation which is described are the inequalities between points. These relations are only valid if both points belong to either a magnitude or a derivative. As it is undesirable to have to distinguish between these type of points in a qualitative model, a pattern is developed to formalise these specific restrictions.

Restrictions for Classes Meeting Specific Conditions There are cases when the relations of an individual have to be restricted depending on the relations that individual has. For example, a workspace meant for a desktop computer should also contain a monitor, but a workspace meant for a laptop does not. Because of the option between a laptop or computer the necessity of having a monitor cannot be modelled as a condition in the workspace class.

This problem can be solved by defining a new class *workspaceWithDesktop* in which necessary and sufficient, and necessary conditions are combined. Recall that necessary conditions are restrictions a consistent individual has to adhere to. Being an individual (or subclass) of such a class implies that its conditions apply ($class \Rightarrow conditions$). Necessary and sufficient conditions should be read as an equivalence relation. Fulfilling the conditions means the individual is an instance of the class, and being an individual of the class means the conditions apply ($class \Longleftrightarrow conditions$).

The class *WorkspaceWithDesktop* (see Figure 5.14) has "being of the class *Workspace*" and "having a desktop as a computer" as necessary and sufficient conditions, meaning that individuals fulfilling these conditions are classified as being a *WorkspaceWith-Desktop*. Individuals of this class have to fulfil the necessary conditions, in this case containing a monitor. As shown combining necessary and sufficient, and necessary conditions allows modelling additional restrictions to individuals adhering to certain conditional relations.

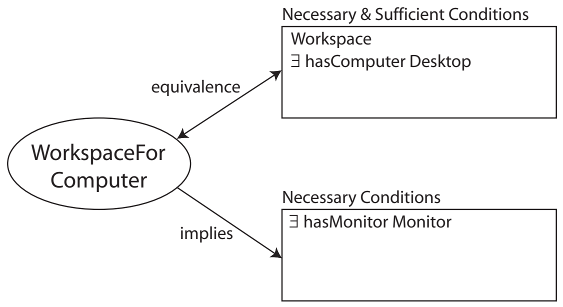

Figure 5.14: Combining necessary and sufficient, and necessary conditions.

Inequalities from Points Belonging to Magnitudes or Derivatives Points belonging to magnitudes (or derivatives) can be classified as such by combining necessary, and necessary and sufficient conditions. Points belonging to magnitudes (or derivatives) can have (in)equality relations with other points belonging to magnitudes (or derivatives). Using the presented pattern necessary and sufficient conditions are stated to classify points, after which the necessary restrictions on the (in)equality apply. The range of the (in)equality should not be a value in the same quantity space, but this is, as mentioned before, not expressible in OWL. The restrictions can be seen in Figures 5.15 and 5.16.

$$\begin{aligned}
&PointBelongingToMagnitude \equiv \exists\, belongsToQuantitySpace \\
&\quad (QuantitySpace \sqcap (\exists\, isQuantitySpaceOf\ Magnitude)) \\
&PointBelongingToMagnitude \sqsubseteq \\
&\quad \forall\, hasInequality\ PointBelongingToMagnitude
\end{aligned}$$

$$\begin{aligned}
&PointBelongingToDerivative \equiv \exists\, belongsToQuantitySpace \\
&\quad (QuantitySpace \sqcap (\exists\, isQuantitySpaceOf\ Derivative)) \\
&PointBelongingToDerivative \sqsubseteq \\
&\quad \forall\, hasInequality\ PointBelongingToDerivative
\end{aligned}$$

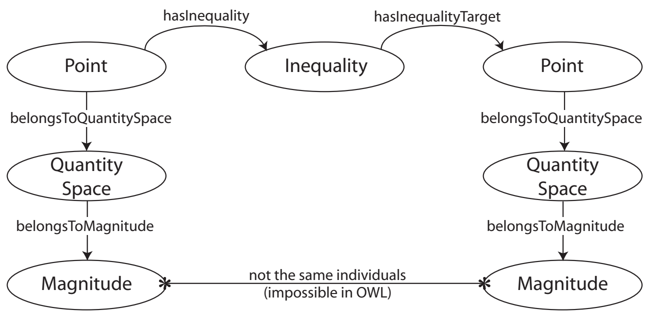

Figure 5.15: Points belonging to magnitudes can only have (in)equality relations with points belonging to magnitudes.

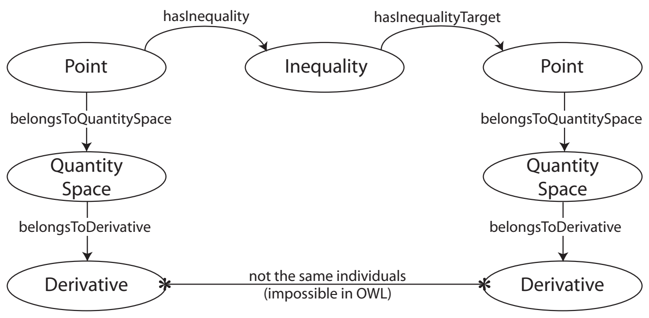

Figure 5.16: Points belonging to derivatives can only have (in)equality relations with points belonging to derivatives.

Inequalities originating from Magnitudes and Derivatives Magnitudes (or derivatives) can have (in)equality relations with other magnitudes (or derivatives) (Figure 5.17 and 5.18). Again non-reflexivity is impossible to formalise. Magnitudes (or derivatives) can also have (in)equality relations with qualitative values from their own quantity space (Figure 5.19 and 5.20). As it is not possible to refer to the individual to which the restriction applies, this restriction is also not formalised. As a result, undesirable (in)equality relations can be formalised, while the ontology remains consistent. Firstly, reflexive inequalities can be described, which are not valid in the reasoning engine underlying DynaLearn. Secondly, inequalities from magnitudes (or derivatives) to qualitative values of other magnitudes (or derivatives) can be described, which are also not allowed by the qualitative simulator.

$$
\begin{aligned}
&Magnitude \sqsubseteq \forall\, hasInequality(Inequality\ \sqcap \\
&\quad (\forall\, hasInequalityTarget\ (Magnitude\ \sqcup\ Point))) \\
&Derivative \sqsubseteq \forall\, hasInequality(Inequality\ \sqcap \\
&\quad (\forall\, hasInequalityTarget\ (Derivative\ \sqcup\ Point)))
\end{aligned}
$$

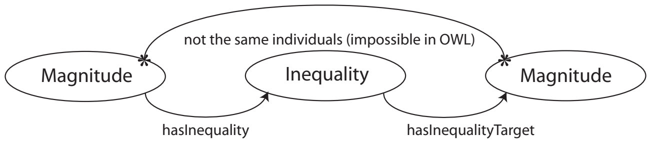

Figure 5.17: A valid (in)equality relations between two magnitudes.

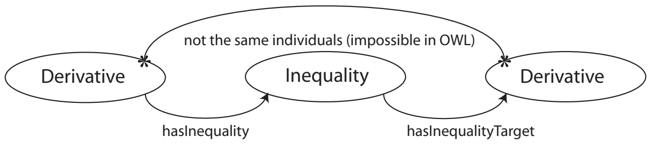

Figure 5.18: A valid (in)equality between two derivatives.

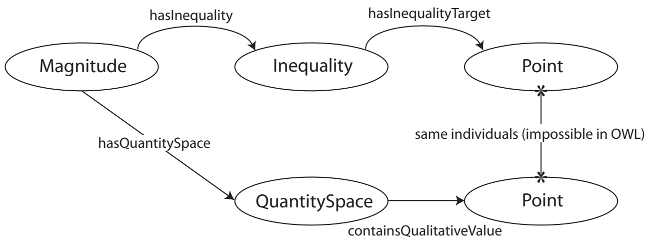

Figure 5.19: An (in)equality relations between a magnitude and a point.

Inequalities Used to Model the Total Order in Quantity Spaces The inequalities used to model the total order in quantity spaces were already discussed in section 5.5.2. Given that inequalities between points are valid, it is impossible to enforce the strict alternation between points and values in a quantity space. In order not to complicate the restrictions in points, the (in)equality relations used to specify the quantity space order are modelled in the interval class (see Figure 5.11).

$$
\begin{aligned}
&Interval \sqsubseteq \forall\, hasInequality(Inequality\ \sqcap\ (hasInequalityTarget\ \geq\ 1)\ \sqcap \\
&\quad (\forall\, hasInequalityTarget\ Point))
\end{aligned}
$$

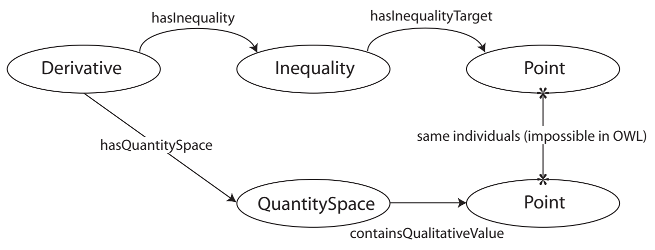

Figure 5.20: An (in)equality relations between a derivative and a point.

The hasValue Relation In order to facilitate easier modelling of (in)equality relations, user-interface introduces *hasValue* relations. These relations are visualised as arrows which point at values of quantity spaces, indicating that the magnitude or derivative has that specific value. Unlike inequalities, *hasValue* relations may point to intervals. The qualitative reasoner translates *hasValue* relations with intervals to 2 inequalities. One stating that the magnitude (or derivative) has a value smaller than the point above the interval, and one stating the value is greater than the point below the interval.

$$hasValue \sqsubseteq \forall\ hasValueTarget\ QualitativeValue$$

5.5.6 Operator Relations

The plus and minus (operator) relations are tertiary relations which add or substract two values, and indicate that a magnitude, derivative or point is equal to the result. Two different operator relations may be distinguished. In the first, only magnitudes and point belonging to magnitudes may participate in the relation (see Figure 5.21). In the second only derivatives may participate in the relation (see Figure 5.22). Both plus/minus relations should have exactly one left-handside and one right-handside. Any number of inequalities may be placed from the plus/minus relations to valid targets. The restrictions for the first plus/minus relation are formalised in both the *PointBelongingToMagnitude* class and the *Magnitude* class. The restrictions for the second is formalised in the derivative class.

PointBelongingToMagnitude Restrictions:

$$\begin{aligned}&\forall\, isLefthandSideOf(Operator \sqcap (hasLefthandSide\ =\ 1) \sqcap\\ &\quad(\forall\, hasRighthandSide\ (Magnitude \sqcup PointBelongingToMagnitude)) \sqcap\\ &\quad(hasRighthandSide\ =\ 1) \sqcap\\ &\quad(\forall\, hasInequality(Inequality \sqcap\\ &\quad(\forall\, hasInequalityTarget(Magnitude \sqcup PointBelongingToMagnitude)))))\end{aligned}$$

Magnitude Restrictions:

$$\begin{aligned}&\forall\, isLefthandSideOf(Operator \sqcap (hasLefthandSide\ =\ 1) \sqcap\\ &\quad(\forall\, hasRighthandSide(Magnitude \sqcup PointBelongingToMagnitude)) \sqcap\\ &\quad(hasRighthandSide\ =\ 1) \sqcap\\ &\quad(\forall\, hasInequality(Inequality \sqcap\\ &\quad(\forall\, hasInequalityTarget(Magnitude \sqcup PointBelongingToMagnitude)))))\end{aligned}$$

Derivative Restrictions:

$$\begin{aligned}&\forall\, isLefthandSideOf(Operator \sqcap (hasLefthandSide = 1) \sqcap\\ &\quad(\forall\, hasRighthandSide\ Derivative) \sqcap\\ &\quad(hasRighthandSide\ =\ 1) \sqcap\\ &\quad(\forall\, hasInequality(Inequality \sqcap\\ &\quad(\forall\, hasInequalityTarget\ Derivative))))\end{aligned}$$

5.6 Aggregate

Aggregates are model constituents consisting of multiple QR ingredients. *Model Fragments* and *Scenarios* are aggregate types. Model Fragments can be further subdivided into (1) *static fragments*, (2) *process fragments*, and (3) *agent fragments*.

5.6.1 Model Fragments

Model fragments consist of multiple model ingredients. As these ingredients are incorporated as either conditions or consequences, model fragments have the possible relations *hasCondition* and *hasConsequence*. As was already discussed in section 2 there are restrictions on what kind of ingredients may be used as conditions and consequences in model fragments. These restrictions were described in Table 2.3.

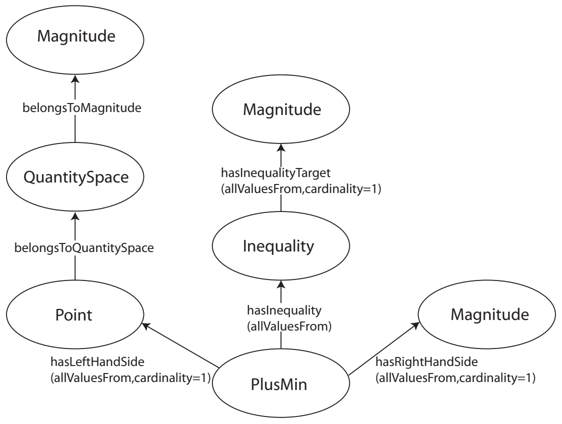

Figure 5.21: The possible Plus and Minus relations between magnitudes and points belonging to magnitudes. Note that every magnitude and the point belonging to the magnitude could be interchanged.

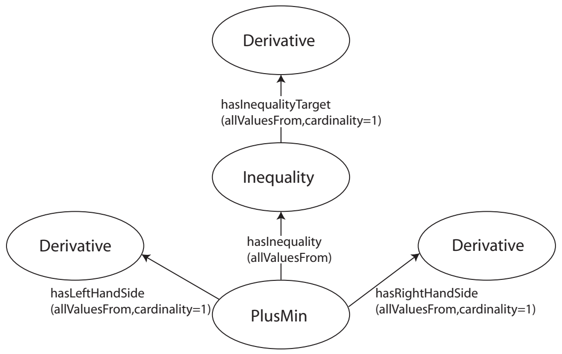

Figure 5.22: The possible Plus and Minus relations between derivatives.

$$\begin{aligned}&ModelFragment \sqsubseteq \forall\, hasCondition(Structural \sqcup Behavioural \sqcup \\ &\quad AssumptionType \sqcup Inequality \sqcup Operator \sqcup ModelFragment) \\ &ModelFragment \sqsubseteq \forall\, hasConsequence(Entity \sqcup Configuration \sqcup \\ &\quad Attribute \sqcup AttributeValue \sqcup Behavioural \sqcup Dependency)\end{aligned}$$

Static fragments are used to describe partial structures of systems. No influences causing change are formalised in static fragments, and thus no new entities, agents and configurations may be introduced (as consequences). Furthermore, no agents may be included as conditions.

$$\begin{aligned}StaticFragment \equiv\ &\neg(\exists\, hasCondition\ Agent) \sqcap \\ &\neg(\exists\, hasConsequence\ Configuration) \sqcap \\ &\neg(\exists\, hasConsequence\ Entity) \sqcap \\ &\neg(\exists\, hasConsequence\ Influence) \sqcap \\ &\neg(\exists\, hasConsequence\ Agent)\end{aligned}$$

Process fragments are used to describe the processes which take place in a system. The processes in process fragments may not be caused by agent, which are therefore prohibited in these fragments.

$$\begin{aligned}ProcessFragment \equiv\ &\neg(\exists\, hasCondition\ Agent) \sqcap \\ &\exists\, hasConsequence\ Influence\end{aligned}$$

Agent fragments describe situations in which an agent may interact with a system. Every model fragment which contains an agent should be an agent fragment.

$$AgentFragment \equiv \exists\, hasCondition\ Agent$$

5.6.2 Scenarios

Scenarios describe situations of the modelled system which becomes the start node of the state graph. All the ingredients which may be used as conditions in model fragments (Table 2.3), except Model Fragments, may be used in scenarios.

$$\begin{aligned}&Scenario \sqsubseteq \forall\, hasConsequence(Structural \sqcup Behavioural \sqcup \\ &\quad AssumptionType \sqcup Inequality \sqcup Operator)\end{aligned}$$

Chapter 6

Representing QR models based on the DynaLearn QR Ontology

The representation of QR models requires formalisations on different levels. Firstly, the model ingredient definitions need to be represented. Secondly, and more complex, is the representation of model fragments and scenarios. The representation of model ingredient definitions is relatively straight-forward given the structure of the DynaLearn QR Ontology (Section 5).

Representations of models are natural extensions to the DynaLearn QR ontology. Each model has its own unique namespace (URI on the web), but each model references concepts in the QR ontology. This assures that each model uses a common vocabulary and can be compared. For example, when representing an entity hierarchy, the entities refer to the Entity concept in the QR ontology ($qrm : Animal \sqsubseteq qr : Entity$, $qrm : Lion \sqsubseteq qrm : Animal$). Similarly the definitions of the other model ingredients are created by creating subclasses of the QR ingredient definitions in the QR ontology. Human readable names and remarks are represented using the RDFS label and comment properties ($qrm : Lion\ rdfs : label\ Lion@en$, $qrm : Lion\ rdfs : comment$ A Lion is a species in the genus Panthera$@en$). The xml:lang attribute of the label and comment properties are used to store multilingual models. Examples of the representation of more complex model ingredient definitions, i.e. attributes, quantities and quantity spaces are discussed in (Section 5).

6.1 Representing model fragments, scenarios and expressions

One benefit of using OWL is the possibility of using a reasoning engine to make inferences. The classification inference in DynaLearn reasoning engine (Section 2) is similar to the classification task that OWL reasoners perform. As such, we investigated the viability of using an OWL reasoner to perform the QR classification task.

6.1.1 Viability of using an OWL reasoner for the QR classification task

OWL reasoners can classify instances with certain properties to particular concepts (with necessary and sufficient conditions). The QR classification task finds model fragments whose conditions are satisfied by a particular scenario. As such, if (the conditional aspects of) model fragments could be represented as concepts in OWL, and scenarios as instances, the OWL reasoner could classify a scenario as being an instance of particular model fragments. Conceptually this can be viewed as that the particular situation described in the scenario is an instance of one or more general situations described by certain model fragments.

The consequences should not be part of the necessary and sufficient conditions, as these model ingredients should be introduced to a scenario when the conditions are met. As such, they would require a separate representation. It might be possible to introduce the consequences of a model fragment in a scenario using the Semantic Web Rule Language [23]. There would be two problems which would have to be solved. The first problem is that part of the classification task in QR context (see section 2) is the derivation of inequalities. From existing inequalities, new inequalities can be derived, which make it possible for a scenario to match on new model fragments. It is not possible to do such reasoning using an OWL reasoner. The second problem is that if no more model fragments can be found which match on the scenario, inequalities may be assumed by the QR reasoner. Again, this allows new model fragments to match on the scenario. This kind of reasoning is not supported by the OWL reasoner to solve.

However, a representation of model fragments that allows for classification by an OWL reasoner is not possible, since the language cannot distinguish between two objects of the same type within the restrictions of a class. For example, it is not possible to specify that that a container x is full and has a relation with container y, and container y has a relation with container z. As OWL does not have the expressiveness to refer to specific individuals in this way, modelling model fragments as classes is impossible. In OWL it is only possible to describe the restriction as: "There is a container which is full, which is connected to a container, which is connected to a container." Such a restriction has multiple interpretations, and is as such unusable. It would even be possible to create a scenario with only 2 containers which would fulfil the restrictions.

6.1.2 A pragmatic representation of model fragments

As a results of the lack of expressiveness to represent model fragments as classes in OWL, we have chosen for a formalisation that uses individuals. However, this introduces a new issue, which is that model fragments are organised in a subtype hierarchy and model fragments can be reused in other model fragments. The subclasses contain exactly the same model ingredients as its parent. However, in the subclass they are all conditional. Furthermore, the subclass introduces new model ingredients in the form of conditions and consequences. The only way to create subclasses in OWL is from classes. The incorporated model fragments are instances of the generic model fragment. This is another reason model fragments have to be formalised as classes, as it is impossible to create instances of instances in OWL. To summarise, the model

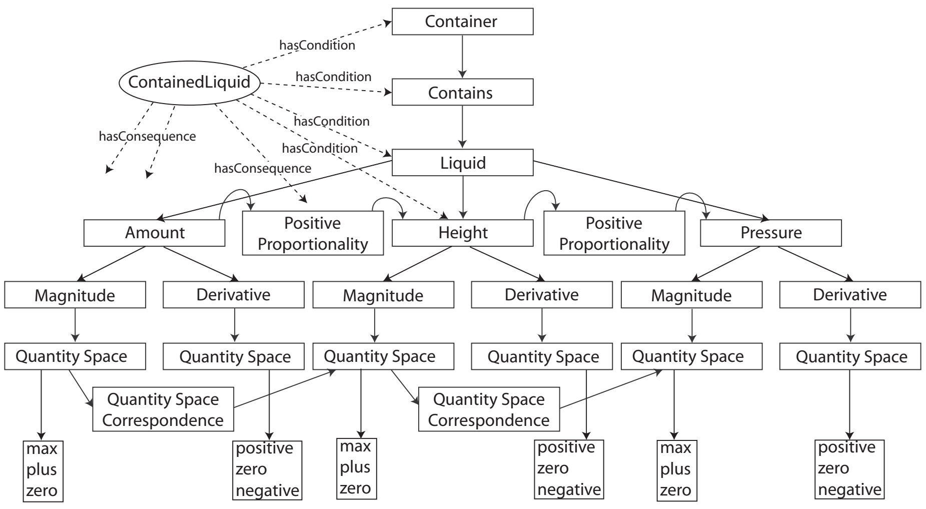

Figure 6.1: An example of the formalisation of a model fragment. Note that that most of the *hasCondition* and *hasConsequence* relations have not been drawn.

fragments have to be formalised as classes, but the conditions and consequences as individuals.

This problem can only be solved by treating the model fragment classes as individuals in OWL, thus making domain ontologies OWL Full. The model fragment definitions are classes, so a class hierarchy of model fragments can be created. These classes have *hasCondition* and *hasConsequence* relations with instances of the QR ingredients they incorporate. This is ontologically not the most desirable solution, as the conditions in model fragments do not correspond to the conditions in OWL. A further disadvantage is that the individuals which model the model fragments pose no restrictions. Therefore, it is impossible to use an OWL reasoner in any way to classify scenarios on model fragments, as individuals cannot be classified on individuals. Furthermore, when model fragments are reused in other model fragments, the reasoner does not indicate the domain ontology is inconsistent when not all of the conditions or consequences belonging to the incorporated model fragment are mentioned in the incorporating model fragment.

An example of a model fragment formalisation using the OWL Full solution is shown in Figure 6.1. Note that not all the *hasCondition* and *hasConsequence* relations have been drawn from the model fragment class to the incorporated individuals. Also the names of the relations have been left out, in order to create a readable figure. Scenarios and expressions have a similar representation. However, they are simpler due to the fact that they are not organised in a hierarchy, nor can model fragments be imported into them.

6.2 Representing Learning Spaces

Each of the LSs (Section 2) have a knowledge representation that is a subset of the knowledge representation of the most complex level *generic and reusable knowledge*. As such, the representation of this level in OWL, can also be used for the representation of the lower levels. One key difference are the concepts of *expressions* and *conditional expressions* in the lower LSs, in contrast with the scenarios and model fragments in LS6. As such, to deal with LSs the LS6 OWL import and export functionality has been adapted to deal with these concepts. The other model ingredients on the lower LSs that seem to differ from LS6 are actually based on the same knowledge representation. For example, the causal relationships on LS2-3 are actually proportionalities. The nodes and relationships on UL1 are actually entities and configurations. The lack of magnitude quantity spaces on UL2 is achieved by hiding a quantity space with a single interval value. As such, models on lower LS can be almost fully considered to be particular LS6 models. A such, the implementation has added an ontology annotation property that indicates which LS a particular model represents. Using this annotation the LS is set when importing the model, which, together with the particular changes mentioned above, is enough to successfully export and import QR models with different LSs from and to OWL.

Chapter 7

Summarizing Simulation Results

For particular use-cases in DynaLearn the simulation results are important. For example, the learner model maintained by the VC component is based on both the model representation and the particular simulation that is currently being run. As such there is a need for a representation of the simulation results.

There are several views that can be taken on QR simulations. For example, the state graph shows how, from the initial state described by the scenario, the system transitions into other distinct qualitative states. The value history shows the values and trends for certain quantities in different states. The dependency view shows the complete structural and behavioural structure in a particular state, which includes the entities, quantities, causal relationships and the particular values of quantities in that state.

The most complete representation would consist of the complete state graph and the complete knowledge in each state. However, although this representation can be generated based on a simulation and the QR formalisation presented in this report, it is not the most appropriate for the knowledge exchange between the DynaLearn components. The main reason is the size of the knowledge structure. Simulations can have many states (into the hundreds), as such having complete descriptions for each of them would results in a large knowledge structure (on average 10Mb). An additional concern is how this knowledge structure can be used in a meaningful way by the other DynaLearn components.

Due to these concerns we have developed a simulation summary, called a *super state* which consists of the union of all the states in the state graph. As such, this super state consists of all the model ingredients that are present in each of the states (except particular value assignments).

The representation we have developed is similar to the one used for model fragments, scenarios and expressions. The representation is shown in Figures 7.1 and 7.2 (in two images for clarity). Figure 7.1 shows some concepts from the DynaLearn QR Ontology. Within the simulation representation the required model ingredient types

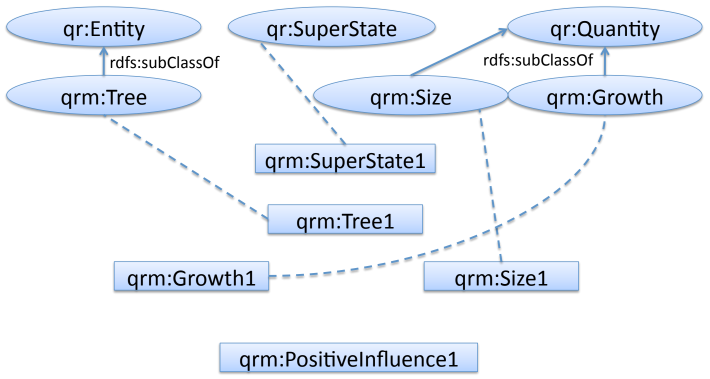

Figure 7.1: Part of the super state representation showing the subclass relationships, and the instance relationships.

from the QR model are added (e.g. $Tree \sqsubseteq qr : Entity$). The main node in the super state representation is the SuperState1 instance, which is an instance of the SuperState concept in the QR ontology. Figure 7.2 shows how the SuperState instance is related to the ingredients in the SuperState. Notice that this representation is equivalent to the representation of model fragments and scenarios.

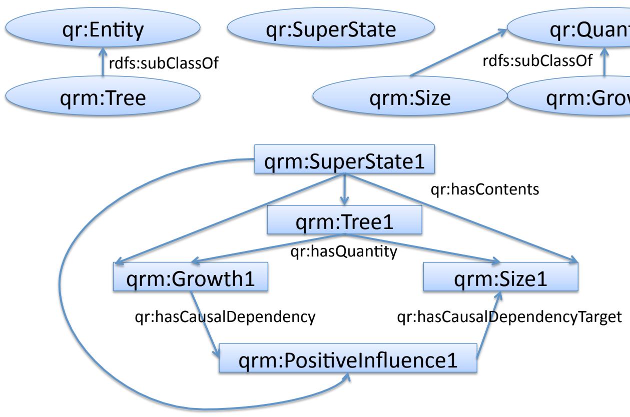

Figure 7.2: Part of the super state representation showing the relationships between within the super state.

Chapter 8

Extracting the Domain Vocabulary for Ontology Matching

For storing and retrieving QR models, as well as the Ontology Based Feedback (OBF), recommendations and grounding tasks, the rich QR representation (Section 6) is essential (Figure 1.1). However, for the Ontology Matching techniques used by the OBF and recommendation tasks, an additional adapted representation is required.

One issue with the model representation when used for the OBF and recommendation tasks is the particular knowledge representation patterns that are used in order to have a complete representation of the model. For example, configurations and causal relationships are represented as classes and instances in order to be able to add remarks and screen positioning information to them (the reification pattern described in Section 5.1.2). Another issue concerns the peculiarities of the QR knowledge representation in contrast to knowledge representation as used typically in ontologies. Ontologies tend to describe concepts based on their possible relationships (e.g. by specifying restrictions on concepts in OWL). However, in DynaLearn, such knowledge about such properties is not explicitly represented for the model ingredient definitions (i.e. the general concepts). Instead, the way such ingredients are used in model fragments gives knowledge about their conceptual meaning (i.e. on the instance level).

Both these issues are problematic for the state of the art ontology matching techniques [33, 17] that are used in DynaLearn (in Work Package 4). These techniques assume that the ontologies being aligned have an ideal ontological structure. As such, to improve the performance of these techniques an adapted representation has been made for the domain vocabulary used in QR models that suits the alignment techniques better.

The domain vocabulary representation uses the same base QR ontology as the QR models, however, relationships such as configurations and causal dependencies are represented as properties instead of classes. As a consequence, this representation is not rich enough to represent a full model. As such, both representations are required for the OBF and recommendation tasks.

The adapted QR ontology has a hierarchy of different causal relationships as properties:

- hasCausalRelationShipWith
  - causes
    - influences
      - positivelyInfluences
      - negativelyInfluences
    - proportionalTo
      - positivelyProportionalTo
      - negativelyProportionalTo

Also, the configurations become a hierarchy of properties. However, where the causal relationships are predefined, the configurations in the hierarchy are defined by the learner.

- hasStructuralRelationship
  - structurallyConnectedTo
    - Connected to
    - Contains

Another key difference is that the properties of domain concepts are defined in the model ingredient definitions instead of in model fragments as done in the QR models. This allows ontology alignment techniques to use their semantics to improve ontology matching performance. However, since such properties are not defined at the model ingredient definitions level in DynaLearn, they have to be inferred from model fragments and scenarios. The resulting representation is as follows:

$$\begin{aligned}
&Container \sqsubseteq Entity\\
&Container \sqsubseteq \exists contains\ Liquid\\
&Container \sqsubseteq \exists contains\ Oil\\
&Container \sqsubseteq \exists connected\_to\ Pipe\\
&Flow \sqsubseteq Quantity\\
&Flow \sqsubseteq \exists negativelyInfluences\ Amount\\
&Flow \sqsubseteq \exists positivelyInfluences\ Amount
\end{aligned}$$

This adapted representation for model ingredient definitions allows ontology alignment techniques to make use of the properties of the domain vocabulary in their comparison with vocabulary in other ontologies. As such, we assume that the ontology matching results will improve in comparison to using the full model representation. Further investigation is required to verify that this assumption holds.

The algorithm to create the adapted representation works as follows. Firstly, loop through each of the model ingredient definitions in the model and create classes for each of them. Secondly, walk through each of the model ingredient instances in the model fragments, scenarios and expression. For each element, check each of the relationships with other model ingredient instances. For each relationship of an element, indicate at the model ingredient definition of this element (i.e. the concept level) that it has this property. In a way, the algorithm abstracts the definition of the model ingredient definitions based on instance data. One issue is that some of the properties might not be generally true for the concept. However, whether this will be problematic is something that has to be investigated within the context of Work Package 4.

Chapter 9

Representing Support Knowledge

In order to assist the learner in his modelling attempt, the DynaLearn ILE provides automated support. Automated support starts with the learner who wants to know something. The learner's request for knowledge is established via interaction with the DynaLearn UI. In response to the learner's request, Virtual Characters present the required knowledge in a communicatively effective way. The responses of the Virtual Characters are custom-tailored towards the learner's needs. Also, the learner is able to interact with the VC's dialogue in order to gain more in-depth knowledge. By assisting the learner in his knowledge-requests, the automated support feature realizes the *a-teacher-for-every-learner* paradigm. This section discusses the content and representation of the support knowledge that is transmitted between learner and VC.

9.1 Support features

Broadly speaking, there are three kinds of support knowledge that the learner may be interested in: model support, User Interface support, and simulation support. We shall now discuss these three forms of automated support.

Model support

Model support explains the characteristics of every ingredient in the model. Since the model is created by the learner based on the broad array of elements and operations that the DynaLearn ILE provides, model support will be different for each model and at any point during the creation of that model. Therefore, the knowledge for the model support feature must be generated dynamically, and must be presented to the learner in conformity with the complexity of both the use level and the status of the model. For every model ingredient on the screen, the learner can pose a "What is?"-question. A Virtual Character will consequently communicate the relevant model support knowledge.

Because of the structured nature of system dynamics models, most model ingredients are connected to other model ingredients. Often support knowledge regarding a certain ingredient X will mention of other model ingredients Y and Z. In such cases, an understanding of Y and Z may be required for a more thorough understanding of X. For instance, knowing what an entity $Column$ (representing a column filled with liquid) is, depends on understanding what its quantities $Volume$, $Height$, and $Pressure$ are. Alternatively, a learner may be interested in a logically connected chain of model ingredients. For instance, knowing that $Volume$ is positively proportional to $Height$, the learner may want to know more about $Height$ (which is positively proportional to $Pressure$).

Because of the inherent connectedness of model ingredients, there is not a clear end to the number of relations and related ingredients, that may be explained by the Virtual Character. Rather than choosing some predefined subset of the totality of available model support knowledge, the learner is given interaction handles that allow him to decide what knowledge he is interested in.

Task support

The second form of support is tailored towards the tasks that a learner can perform. The DynaLearn ILE is divided into different screen contents (or views) and dialogues. Each view and every dialogue addresses a specific functionality. In every such context the learner is able to ask a set of "How to?"-questions, namely those questions that pertain to the current view.

For instance, in the entity hierarchy view, showing the entity definitions of the model, the task support allows questions regarding operations that can be performed in that view: e.g. "How to add an entity?", "How to remove an entity?", "How to add remarks to an entity?" But there are also questions that are conceptually related, even though they belong to a different view. For instance the question "How to add an entity instance?" seems logical once we are concerned with entity definitions. However, instances of those definitions are added in a different view. The task support must communicate this (and similar) knowledge.

Simulation support

The third form of support concerns the simulation results. For the learner it is valuable to know why certain behaviours are generated and why specific values occur in the simulation of his model. The simulation support feature allows the learner to pose a "Why?"-question with respect to the simulated values that are displayed in the simulation results. E.g. when the relation results show an increase of in $Volume$ (i.e. a positive derivative for that quantity), then the user can select that value and as "Why does Volume increase?".

9.2 Requirements for representing support knowledge

In the OBF and the recommendation tasks, the whole model is considered and logical properties of the representation are important. The goal of the support functionality is more *lightweight*, either pertaining to a single model ingredient, a single UI view/dialogue, or a single simulation result. This is because the information that the Virtual Characters communicate to the learner should be concise. More elaborate information requests are accessible via additional hyperlinks in the Virtual Character's speech balloons, but each individual information request should be kept small. A surplus of information would distract the learner too much.

Support information should be made available immediately based upon the learner's interaction with the User Interface or the Virtual Character. Therefore, complex information extraction and/or logical derivation methods should be avoided as much as possible. Instead, the representation of the support knowledge should make complex processing superfluous by being particularly tailored towards the support effort.

We call this the *modularity* requirement. Additionally, a second requirement for the knowledge architecture of the support functionality is that it should be as **close to natural language** as possible. We shall now explain how both requirements are met.

9.2.1 Modularity

In order to address the lightweight nature of the support interactions between learner and Virtual Character, we want the representation to be inherently **modular**. A support interaction typically consists of a to-and-fro between learner requests and Virtual Character elucidations. Modular chunks of knowledge should be communicated by the Virtual Character to the learner, and the learner can click any of the follow-up hyperlinks that occur in the VC's text balloons in order for the VC to communicate yet another modular chunk of knowledge.

One disadvantage of an OWL representation is that the information is not represented in terms of model ingredients, but in terms of the axiomatic restrictions between those ingredients. The information pertaining to a single model ingredient is therefore typically distributed throughout the OWL representation of the entire model.

The modular chunks of support information are not only ideal for the to-and-fro request-and-follow-up interaction mode that characterizes support interactions. Also, these modular chunks of data can be combined without any further restrictions. E.g. it is possible to aggregate the information regarding a quantity *one level deep*. This would encompass the modular data chunk describing the quantity itself (including the relations that it has to other ingredients) *and* the data chunks for all of these related ingredients. Another example is the aggregate of all entity definition chunks. Putting these together automatically forms the entity definition hierarchy (including all the parent/child-relationships). We hope to make ample use of this **modular knowledge aggregation feature** when we implement further feedback functionalities that make use of specific subsets of knowledge.

Natural language

The second requirement is that the support knowledge representation stays as close to natural language as possible. The reason for this is that the knowledge should be communicated by a Virtual Character. Natural language is the most accessible and expressive way of doing this. Artificial languages are less natural and must first be learnt, whereas the support functionality should also help starters, i.e. learners without prior knowledge of the DynaLearn ILE. Schematic representations, although potentially powerful, would not be adequate for the support functionality, since its purpose is to explain aspects of the schematic representation of qualitative systems knowledge. Introducing yet another schematic representation would thus be counter-productive.

9.3 Implementation of support knowledge

In order to be modular, each ingredient is associated with its own RDF/XML structure. Any combination of these modules forms a valid RDF/XML file. This means that *every* combination of knowledge chunks the support features might generate are each of the following things: (1) a well-formed and valid XML document, (2) a well-formed RDF document, (3) input that can be directly used by Semantic Web processors, (4) input that can be directly parsed by the Virtual Character component of the DynaLearn ILE.

Each collection of support information chunks has the following RDF/XML structure wrapped around. `UNIQUE-MODEL-ID` is the name of the model (as given by the user) appended with the date and time at which the model was created.

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE rdf:RDF [
    <!ENTITY qr
       'http://www.science.uva.nl/~jliem/ontologies/QRvocabulary.owl#'>
    <!ENTITY xsd "http://www.w3.org/TR/2009/WD-xmlschema11-2-20091203/">
    <!ENTITY rdf 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
    <!ENTITY rdfs 'http://www.w3.org/2000/01/rdf-schema#'>
]>

<rdf:RDF
    xmlns:qr="&qr;"
    xmlns:rdf="&rdf;"
    xmlns:rdfs="&rdfs;"
    xml:base="http://www.dynalearn.eu/models/UNIQUE-MODEL-ID.owl"
>

CONTENT

</rdf:RDF>
```

In the above code wrapper, `CONTENT` can be any number of modular information chunks.

A modular chunk of support information has the following structure.

```
<rdf:Description rdf:about="SUBJECT-URI">
    PREDICTATE-OBJECT-STATEMENTS
</rdf:Description>
```

SUBJECT-URI is the unique web address for the subject ingredient that this information chunk is about. The subject ingredient is described by a number of PREDICATE-OBJECT-STATEMENTS. There are two types of predicate-object statements: data and object properties.

Data property

A *data property* predicates a data value of the subject ingredient. A data value is itself not a model ingredient, but is defined by XML Schema. The structure of a data property is as follows.

```
<PREDICATE-URI xml:lang="LANGUAGE-CODE"
   rdf:datatype="DATA-TYPE-URI">
      CONTENT
</PREDICATE-URI>
```

PREDICATE-URI is the URI of the data property. These are all part of the qr: namespace, which is specifically defined for the DynaLearn project. LANGUAGE-CODE indicates the language of the data content. The name can be set in the DynaLearn ILE. The language code only applies to some data values, most notably strings. DATA-TYPE-URI refers to the type of data value that is used. Data types all refer to the xsl: namespace for XML-Schema. CONTENT is the data value itself. The kinds of content that is allowed here depends on the specified data type in DATA-TYPE-URI.
An example of a data property instance is given.

```
<qr:hasName xml:lang="en" rdf:datatype="&xsd;#string">
   Container left
</qr:hasName>
```

Object property

An *object property* predicates an object ingredient of the subject ingredient. Since the object term is itself a model ingredient, the object term is itself defined in another modular information chunk. The structure of an object property is as follows.

```
<PREDICATE-URI rdf:resource="OBJECT-URI"/>
```

PREDICATE-URI is the URI of the object property. These are all part of the qr: namespace, which is specifically defined for the DynaLearn project. OBJECT-URI is the unique address of the object ingredient. An example of an complete information chunk for the request in is given.

```
<rdf:Description
   rdf:about="entityInstance#Container_right_000000068">
   <qr:hasName xml:lang="en" rdf:datatype="&xsd;#string">
      Container left
   </qr:hasName>
   <qr:hasRemarks xml:lang="en" rdf:datatype="&xsd;#string">
   </qr:hasRemarks>
   <qr:hasCategory xml:lang="en" rdf:datatype="&xsd;#string">
      entityInstance
   </qr:hasCategory>
   <qr:hasState xml:lang="en" rdf:datatype="&xsd;#string">
      condition
   </qr:hasState>
   <qr:inAggregate
      rdf:resource="modelFragment#Contained_liquid_000000023"/>
   <qr:hasRelationFrom
      rdf:resource="configurationInstance#Contains_000000048"/>
   <qr:hasRelationTo
      rdf:resource="configurationInstance#Tube_000000058"/>
</rdf:Description>
```

Natural language

In order to stay close to natural language, we want to represent the knowledge in a flat, i.e. **non-nested** manner. The standard OWL export provides deeply nested representation structures, even for relatively simple model ingredients. As seen in the previous section, the RDF/XML representation takes these considerations into account.

In addition to avoiding nesting as much as possible, the structure of the representation should itself be similar to natural language's grammatical structure. The general form of a sentence is $SubjectPredicateObject$, so that the $Object$ term is predicated of the $Subject$ term. As seen in the previous section, the RDF/XML statements have exactly the right structure.

9.4 Properties of support knowledge modules

In the next section we give an exhaustive overview of the knowledge that will be transmitted in the support interactions in the DynaLearn ILE. Every model ingredient has its own information pattern. We will first enumerate the properties that every model ingredient's information pattern has, and will then provide an overview of all the model ingredient's information patterns. A model ingredient information patters consists of the following components.

Name

Each model ingredient has a unique name. The name is one of the properties of a class, i.e. the `qr:hasCategory` property.

Description

Each model ingredient has exactly one description which describes the class' purpose in an informal way. The description is for informal purposes exclusively and is unrelated to the OWL representation.

Properties

Each model ingredient has a number of properties that describe it. Data and object properties (see Section 9.3 and 9.3) are both included in this list.

Subclasses list

Each model ingredient has a, possibly empty, list of child model ingredients. A child ingredient takes over all properties that are defined for the parent ingredient. E.g. 'entityInstance' incorporates the properties that are defined for 'Instance' and 'Ingredient', because 'entityInstance' is an 'Instance', and each 'Instance' is an 'Ingredient' as well.

9.5 Support knowledge modules overview

In this section we give an exhaustive overview of the knowledge that will be transmitted in the support interactions in the DynaLearn ILE.

9.5.1 Ingredient

Properties:

- `qr:hasCategory ('AgentDefinition' | 'AgentFragment' | 'AgentInstance' | 'AssumptionDefinition' | 'AssumptionInstance' | 'AttributeDefinition' | 'AttributeInstance' | 'Operator' | 'ConfigurationDefinition' | 'ConfigurationInstance' | 'CorrespondenceDefinition' | 'CorrespondenceInstance' | 'EntityDefinition' 'EntityInstance' | 'ExpressionFragment' | 'FullConfiguration' | 'ProcessFragment' | 'QuantityDefinition' | 'QuantityInstance' | 'QuantityRelation' | 'QuantitySpaceConfiguration' | 'QuantitySpaceDefinition' | 'QuantitySpaceInstance' | 'Scenario' | 'StaticFragment' | 'Identity' | 'Inequality' | 'ValueConfiguration' | 'ValueDefinition' | 'ValueInstance')`
- `qr:hasName string`

Subclasses:

- Aggregate 9.5.2
- Definition 9.5.8
- Instance 9.5.18

9.5.2 Aggregate

Description: An aggregate is a collection of instances 9.5.18. Aggregates are either entirely conditional, i.e. scenarios 9.5.7, or both conditional and consequential, i.e. model fragments 9.5.3.
Properties:

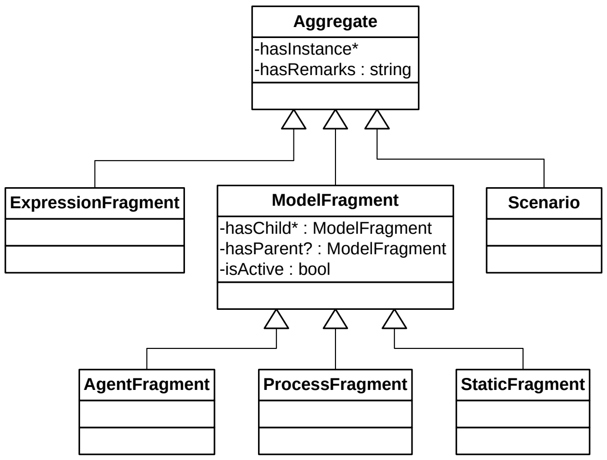

Figure 9.1: The knowledge that is transmitted for aggregate ingredients

- `hasCategory ('AgentFragment' | 'ExpressionFragment' | 'ProcessFragment' | 'Scenario' | 'StaticFragment')`
- `qr:hasInstance* Instance`
- `qr:hasRemarks string`

Subclasses:

- Model Fragment 9.5.3
- Scenario 9.5.7

9.5.3 Model Fragment

Description: A model fragment is a consequential aggregation 9.5.2 of instances 9.5.18.
Properties:

- `qr:hasCategory 'AgentFragment', 'ProcessFragment', 'StaticFragment'`
- `qr:hasChild ModelFragment`
- `qr:hasParent ModelFragment`
- `qr:isActive boolean`, indicates whether a model fragment is operational in the current simulation or not.

Subclasses:

- Agent fragment 9.5.4
- Process fragment 9.5.5
- Static fragment 9.5.6

9.5.4 Agent Fragment

Description: An agent fragment is a model fragment 9.5.3 that contains at least one agent instance 9.5.19 and that may contain one or more influences 9.5.32. They are used to describe the influences that agent instances (= exogenous entity instances) have on the system.
Properties:

- `qr:hasCategory 'AgentFragment'`

9.5.5 Process Fragment

Description: A process fragment is a model fragment 9.5.3 that contains at least one influence 9.5.32, but no agent instances 9.5.19. They are used to describe processes that take place within the system.
Properties:

- `qr:hasCategory 'ProcessFragment'`

9.5.6 Static Fragment

Description: A static fragment is a model fragment 9.5.3 that does not contain agent instances 9.5.19 nor influences 9.5.32. They describes parts of the structure of the system, together with the proportionalities that exist between the quantity instances 9.5.33.
Properties:

- `qr:hasCategory 'StaticFragment'`

9.5.7 Scenario

Description: A scenario is a conditional aggregation 9.5.2 of instances 9.5.18.
Properties:

- `qr:hasCategory 'Scenario'`

9.5.8 Definition

Properties:

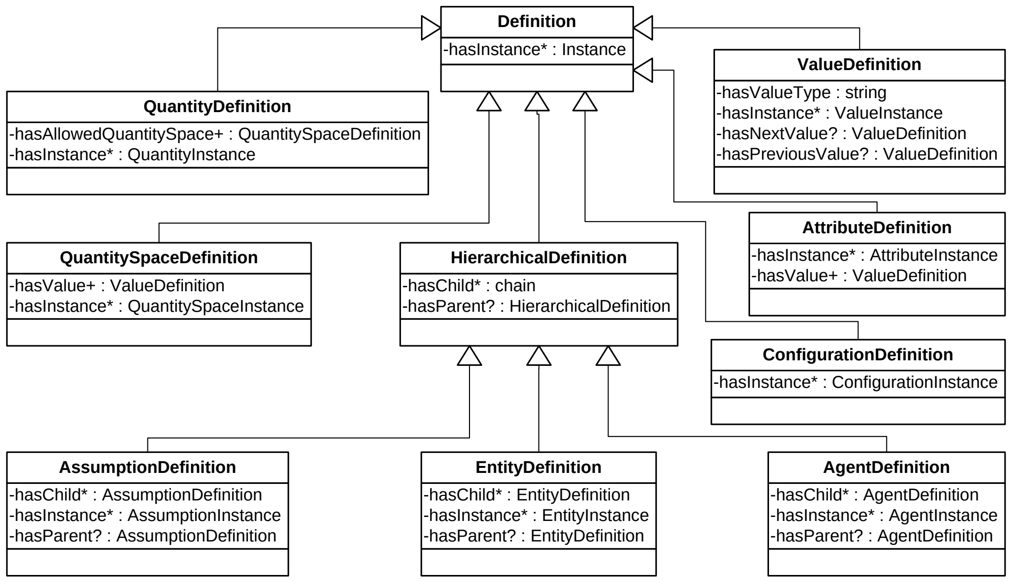
Definition
-hasInstance* : Instance
QuantityDefinition
-hasAllowedQuantitySpace+ : QuantitySpaceDefinition
-hasInstance* : QuantityInstance
ValueDefinition
-hasValueType : string
-hasInstance* : ValueInstance
-hasNextValue? : ValueDefinition
-hasPreviousValue? : ValueDefinition
AttributeDefinition
-hasInstance* : AttributeInstance
-hasValue+ : ValueDefinition
QuantitySpaceDefinition
-hasValue+ : ValueDefinition
-hasInstance* : QuantitySpaceInstance
HierarchicalDefinition
-hasChild* : chain
-hasParent? : HierarchicalDefinition
ConfigurationDefinition
-hasInstance* : ConfigurationInstance
AssumptionDefinition
-hasChild* : AssumptionDefinition
-hasInstance* : AssumptionInstance
-hasParent? : AssumptionDefinition
EntityDefinition
-hasChild* : EntityDefinition
-hasInstance* : EntityInstance
-hasParent? : EntityDefinition
AgentDefinition
-hasChild* : AgentDefinition
-hasInstance* : AgentInstance
-hasParent? : AgentDefinition

Figure 9.2: The knowledge that is transmitted for definition ingredients

- `qr:hasCategory ('AgentDefinition' | 'AssumptionDefinition' | 'AttributeDefinition' | 'ConfigurationDefinition' | 'EntityDefinition' | 'QuantityDefinition' | 'QuantitySpaceDefinition' | 'ValueDefinition')`
- `qr:hasInstance Instance`

Subclasses:

- Attribute Definition 9.5.9
- Configuration Definition 9.5.10
- Hierarchical Definition 9.5.11
- Quantity Definition 9.5.15
- Quantity Space Definition 9.5.16
- Value Definition 9.5.17

9.5.9 Attribute Definition

Description: Attributes are properties of entities 9.5.22 that remain static during simulation (i.e. that do not change). They have an associated set of possible attribute values 9.5.17.
Properties:

- `qr:hasCategory 'AttributeDefinition'`
- `qr:hasInstance* AttributeInstance`
- `qr:hasValue* ValueInstance`

9.5.10 Configuration Definition

Description: A configuration represents a relation between instances of entities 9.5.22 and agents 9.5.19.
Properties:

- `qr:hasCategory 'ConfigurationDefinition'`
- `qr:hasInstance* ConfigurationInstance`

9.5.11 Hierarchical Definition

Properties:

- `qr:hasCategory 'AgentDefinition' | 'AssumptionDefinition' | 'EntityDefinition'`
- `qr:hasChild* HierarchicalDefinition`
- `qr:hasInstance* (AgentInstance | AssumptionInstance | EntityInstance)`
- `qr:hasParent? HierarchicalDefinition`

Subclasses:

- Agent Definition 9.5.12
- Assumption Definition 9.5.13
- Entity Definition 9.5.14

9.5.12 Agent Definition

Description: An agent represents an entity outside of the modelled system. Agents can have quantities 9.5.33 influencing 9.5.32 the rest of the system. Influences that start at agents are called exogenous or external influences.
Synonyms:

- exogenous entity

Properties:

- `qr:hasCategory 'AgentDefinition'`
- `qr:hasChild* AgentDefinition`
- `qr:hasInstance AgentInstance`
- `qr:hasParent? AgentDefinition`

9.5.13 Assumption Definition

Properties:

- `qr:hasCategory 'AssumptionDefinition'`
- `qr:hasChild* AssumptionDefinition`
- `qr:hasInstance AssumptionInstance`
- `qr:hasParent? AssumptionDefinition`

9.5.14 Entity Definition

Description: An entity represents a physical object or an abstract concept that plays a role within the modelled system. Entities are themselves arranged in a subtype/supertype-hierarchy.
Properties:

- `qr:hasCategory 'EntityDefinition'`
- `qr:hasChild* EntityDefinition`
- `qr:hasInstance* EntityInstance`
- `qr:hasParent? EntityDefinition`

9.5.15 Quantity Definition

Description: A quantity represent a changeable feature of an entity 9.5.22 or agent 9.5.19. Each quantity has two associated quantity spaces 9.5.16: a definable one for the magnitude, and the default quantity space $\{Min, Zero, Plus\}$ for the derivative of the quantity.
Properties:

- `qr:hasAllowedQuantitySpace* QuantitySpaceDefinition`
- `qr:hasCategory 'QuantityDefinition'`
- `qr:hasInstance* QuantityInstance`

9.5.16 Quantity Space Definition

Description: A quantity space specifies a range of qualitative values 9.5.17 a quantity's 9.5.33 magnitude or derivative can have. The qualitative values in a quantity space form a total order. Each qualitative value is either a point or an interval, and within the quantity spaces these two types consecutively alternate.
Properties:

- `qr:hasCategory 'QuantitySpaceDefinition'`
- `qr:hasInstance* QuantitySpaceInstance`
- `qr:hasValue* ValueDefinition`

9.5.17 Value Definition

Description: A value definitions, either a points or intervals, form the quantity space definitions 9.5.16 of quantity definitions 9.5.15, as well as the derivative quantity space definition 9.5.16.
Properties:

- `qr:hasCategory 'ValueDefinition'`
- `qr:hasNextValue? ValueDefinition`
- `qr:hasPreviousValue? ValueDefinition`
- `qr:hasValueType ('interval' | 'point')`

9.5.18 Instance

Properties:

- `qr:hasCategory ('AgentInstance' | 'AssumptionInstance' | 'Attribute Instance' | 'EntityInstance' | 'Operator' | 'FullConfiguration' | 'QuantitySpaceConfiguration' | 'ValueConfiguration' | 'CorrespondenceInstance' | 'Identity' | 'Inequality' | 'Influence' | 'QuantityRelation' | 'ValueInstance')`
- `qr:hasState 'condition, 'consequence'`

Subclasses:

- Agent Instance 9.5.19
- Assumption Instance 9.5.20
- Attribute Instance 9.5.21
- Entity Instance 9.5.22
- Relation Instance 9.5.23
- Value Instance 9.5.35

9.5.19 Agent Instance

Description: An agent represents an entity outside of the modelled system. Agents can have quantities 9.5.33 influencing 9.5.32 the rest of the system. Influences that start at agents are called exogenous or external influences.

- `qr:hasCategory 'AgentInstance'`
- `qr:hasDefinition AgentDefinition`

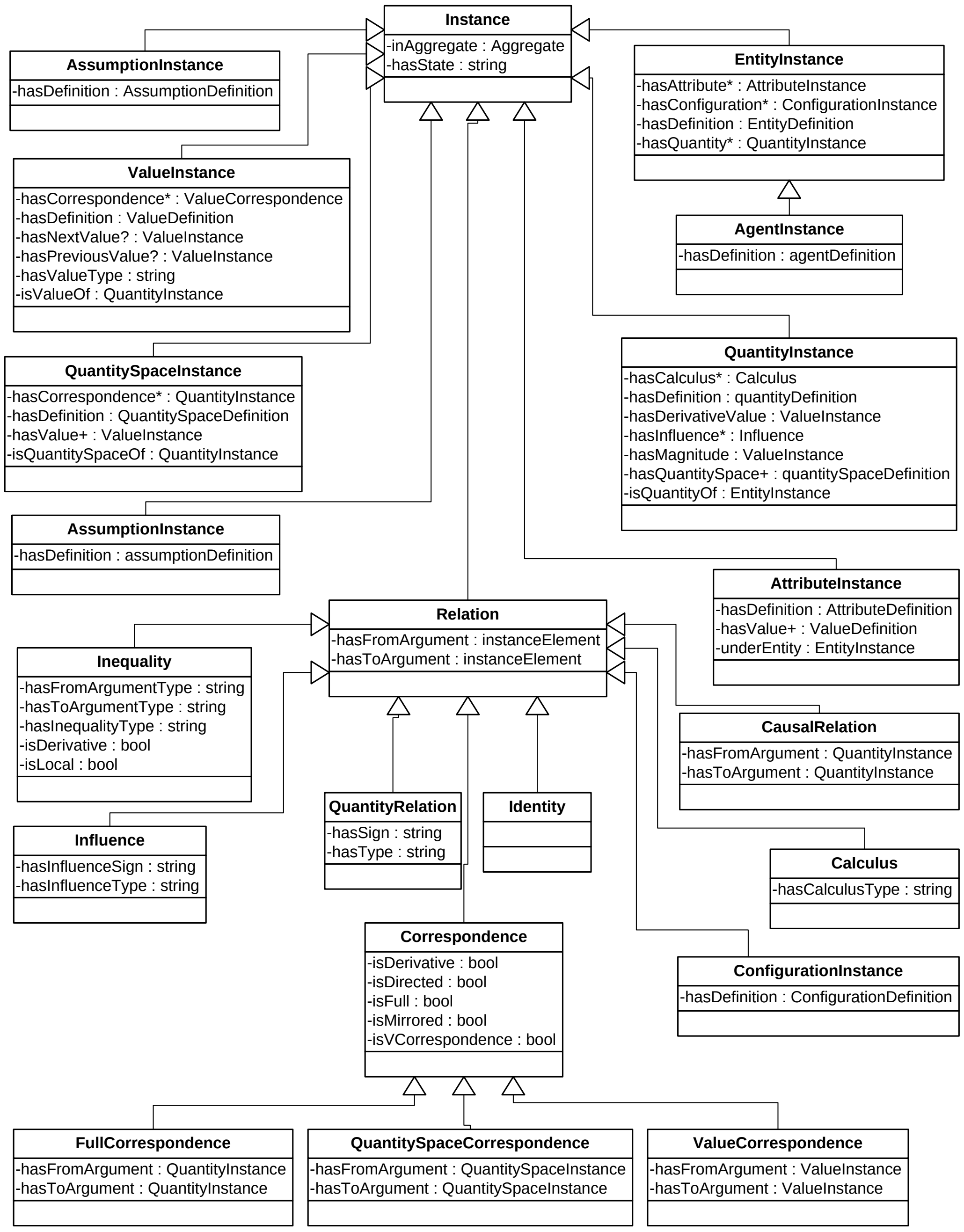
Instance
-inAggregate : Aggregate
-hasState : string
AssumptionInstance
-hasDefinition : AssumptionDefinition
EntityInstance
-hasAttribute* : AttributeInstance
-hasConfiguration* : ConfigurationInstance
-hasDefinition : EntityDefinition
-hasQuantity* : QuantityInstance
ValueInstance
-hasCorrespondence* : ValueCorrespondence
-hasDefinition : ValueDefinition
-hasNextValue? : ValueInstance
-hasPreviousValue? : ValueInstance
-hasValueType : string
-isValueOf : QuantityInstance
AgentInstance
-hasDefinition : agentDefinition
QuantitySpaceInstance
-hasCorrespondence* : QuantityInstance
-hasDefinition : QuantitySpaceDefinition
-hasValue+ : ValueInstance
-isQuantitySpaceOf : QuantityInstance
QuantityInstance
-hasCalculus* : Calculus
-hasDefinition : quantityDefinition
-hasDerivativeValue : ValueInstance
-hasInfluence* : Influence
-hasMagnitude : ValueInstance
-hasQuantitySpace+ : quantitySpaceDefinition
-isQuantityOf : EntityInstance
AssumptionInstance
-hasDefinition : assumptionDefinition
AttributeInstance
-hasDefinition : AttributeDefinition
-hasValue+ : ValueDefinition
-underEntity : EntityInstance
Relation
-hasFromArgument : instanceElement
-hasToArgument : instanceElement
Inequality
-hasFromArgumentType : string
-hasToArgumentType : string
-hasInequalityType : string
-isDerivative : bool
-isLocal : bool
CausalRelation
-hasFromArgument : QuantityInstance
-hasToArgument : QuantityInstance
QuantityRelation
-hasSign : string
-hasType : string
Identity
Influence
-hasInfluenceSign : string
-hasInfluenceType : string
Calculus
-hasCalculusType : string
Correspondence
-isDerivative : bool
-isDirected : bool
-isFull : bool
-isMirrored : bool
-isVCorrespondence : bool
ConfigurationInstance
-hasDefinition : ConfigurationDefinition
FullCorrespondence
-hasFromArgument : QuantityInstance
-hasToArgument : QuantityInstance
QuantitySpaceCorrespondence
-hasFromArgument : QuantitySpaceInstance
-hasToArgument : QuantitySpaceInstance
ValueCorrespondence
-hasFromArgument : ValueInstance
-hasToArgument : ValueInstance

Figure 9.3: The knowledge that is transmitted for instance ingredients

9.5.20 Assumption Instance

Description: Assumptions are labels that are used to indicate that certain conditions are assumed to be true. They are often used to constrain the possible behaviour of a model.
Properties:

- `qr:hasCategory 'AssumptionInstance'`
- `qr:hasDefinition AssumptionDefinition`

9.5.21 Attribute Instance

Properties:

- `qr:hasCategory 'AttributeInstance'`
- `qr:hasDefinition AttributeDefinition`
- `qr:hasValue+ ValueDefinition`

9.5.22 Entity Instance

Properties:

- `qr:hasAttribute* AttributeInstance`
- `qr:hasCategory 'EntityInstance'`
- `qr:hasConfiguration* ConfigurationInstance`
- `qr:hasDefinition EntityDefinition`
- `qr:hasQuantity* QuantityInstance`

9.5.23 Relation Instance

Properties:

-
  ```
  qr:hasCategory ('Operator' |
  'CorrespondenceInstance' | 'FullConfiguration' |
  'Identity' | 'Inequality' | 'Influence' |
  'QuantityRelation' | 'QuantitySpaceConfiguration' |
  'ValueConfiguration')
  ```
- `qr:hasState 'condition, 'consequence'`

Subclasses:

- Operator 9.5.24
- Configuration Instance 9.5.25

- Correspondence 9.5.26
- Identity 9.5.30
- Inequality 9.5.31

9.5.24 Operators

Description: Operators (or calculi) are used to calculate the sum (plus) or difference (minus) of two value items. Operator relations can also be the target or source of an inequality relation. There are nine different ways in which plus/minus relations can be used, depending on the type of the two arguments in the relation. Using operator relations, more complex expressions can be created than is possible with only inequalities.
Properties:

- `qr:hasCategory 'Operator'`
- `qr:hasOperatorType ('minus' | 'plus')`

9.5.25 Configuration Instance

Properties:

- `qr:hasCategory 'ConfigurationInstance'`
- `qr:hasDefinition ConfigurationDefinition`

9.5.26 Correspondence

Description: A correspondence represents that different qualitative values belonging to different quantity spaces are the current value at the same time. Correspondences can be either directed or undirected.
Properties:

- `qr:hasCategory ('FullConfiguration' | 'QuantitySpaceConfiguration' | 'ValueConfiguration')`
- `qr:isDerivative boolean`
- `qr:isDirected boolean`
- `qr:isFull boolean`
- `qr:isMirrored boolean`
- `qr:isVCorrespondence boolean`

Properties:

- Full Correspondence 9.5.27
- Quantity Space Correspondence 9.5.28
- Value Correspondence 9.5.29

9.5.27 Full Correspondence

Description: A *full correspondence* is both a quantity space correspondence 9.5.28 between the quantity spaces 9.5.34 of the magnitudes, as well as a quantity space correspondence between the quantity spaces of the derivatives.

- `qr:hasCategory 'FullCorrespondence'`
- `qr:hasFromArgument QuantityInstance`
- `qr:hasToArgument QuantityInstance`

9.5.28 Quantity Space Correspondence

Description: A *quantity space correspondence* is a correspondence between two quantity spaces 9.5.34, indicating that *each* of the values 9.5.35 of the quantity spaces correspond to each other.

- `qr:hasCategory 'QuantitySpaceCorrespondence'`
- `qr:hasFromArgument QuantitySpaceInstance`
- `qr:hasToArgument QuantitySpaceInstance`

9.5.29 Value Correspondence

Description: A value correspondence 9.5.29 is a relations between qualitative values 9.5.35 of quantity spaces 9.5.34 belonging to different quantities 9.5.33.

- `qr:hasCategory 'ValueCorrespondence'`
- `qr:hasFromArgument ValueInstance`
- `qr:hasToArgument ValueInstance`

9.5.30 Identity

Description: Identities are relations that are used to specify that two entities 9.5.22 in different imported model fragments 9.5.3 are the same. There are two possible applications for identities.
Properties:

- `qr:hasState 'condition'`
- `qr:hasCategory 'Identity'`
- `qr:hasFromArgument (AgentInstance | EntityInstance)`
- `qr:hasToArgument (AgentInstance | EntityInstance)`

9.5.31 Inequality

Description: An inequalities specifies an ordinal relation between two items. It expresses that the two items are different from (or equal to) one another. (Because inequalities specify an order between items, they are sometimes referred to as ordinal relations. There are eleven ways to use inequalities, depending on the type of the two items related by it.)
Properties:

- `qr:hasCategory 'Inequality'`
- `qr:hasFromArgumentType string`
- `qr:hasInequalityType ('SmallerThan', 'SmallerThanOrEqualTo', 'EqualTo', 'GreaterThanOrEqualTo', 'GreaterThat')`
- `qr:hasResult boolean`
- `qr:hasToArgumentType string`
- `qr:isDerivative boolean`
- `qr:isLocal boolean`

9.5.32 Influence

Definition Influences are directed relations between two quantities, and are either positive or negative. Influences are the cause of change within a model, and are therefore said to model processes. Depending on the magnitude of the source quantity and the type of influence, the derivative of the target quantity either increases or decreases.
Properties:

- `qr:hasCategory 'Influence'`
- `qr:hasInfluenceType ('influence' | 'proportionality')`
- `qr:hasInfluenceSign ('minus' | 'plus')`

9.5.33 Quantity Instance

Properties:

- `qr:hasOperator* Operator`
- `qr:hasCategory 'QuantityInstance'`
- `qr:hasDefinition QuantityDefinition`
- `qr:hasDerivativeValue ValueInstance`
- `qr:hasInfluence* Influence`

- `qr:hasMagnitude ValueInstance`
- `qr:hasQuantitySpace+ QuantitySpaceInstance`
- `qr:isQuantityOf (EntityInstance | Agentinstance)`

9.5.34 Quantity Space Instance

Properties:

- `qr:hasCategory 'quantitySpaceInstance'`
- `qr:hasCorrespondence* QuantitySpaceCorrespondence`
- `qr:hasDefinition QuantitySpaceDefinition`
- `qr:hasValue+ ValueInstance`
- `qr:isQuantitySpaceOf QuantityInstance`

9.5.35 Value Instance

Description: A value instance is either a point or interval which can become the current magnitude or current derivative of a quantity instance 9.5.33. Qualitative value instances are organised in quantity space instances 9.5.34.
Data properties:

- `qr:hasCategory 'ValueInstance'`
- `qr:hasCorrespondence* CorrespondenceInstance`
- `qr:hasNextValue? ValueInstance`
- `qr:hasPreviousValue? ValueInstance`
- `qr:hasValueType ('interval' | 'point')`
- `qr:isValueOf QuantityInstance`

Chapter 10

Virtual Character Interaction

This section discusses the function calls and responses between the Conceptual Modelling (CM) component and the Virtual Character (VC) component. As can be seen in Figure 1.1, the main functions are communicating model content, simulation content, questions, and support knowledge. Model and simulation content are discussed in Sections 6 and 7 respectively. Support knowledge is discussed in Section 9. This Section therefore deals with the questions, but also with the some low-level functionality too detailed to be shown in Figure 1.1.

10.1 General communication

This section discusses general communication between the (CM) component and the VC component. When required the CM component starts the VC components by calling their executables. The CM component then acts as a server to which the VC component connects. The knowledge exchange between the components is done via this persistent socket connection (Figure 10.1). Both the CM component and the VC component can send request to each other. However, the CM requests are only meant to start specific use-cases. Once a use case has started the VC has the initiative. Note that this corresponds with the user initiative. Initiative is with the CM component if the user manipulates the CM, initiative is with the VC component if the user interacts with the virtual characters. The data format used in the communication is XML, so that messages can be easily parsed. However, for larger knowledge structures OWL is incorporated into the XML messages.

The general approach is to have each request have an immediate response. In general, if no specific output has to be generated, the CM component responds to the VC with `<response>ok</response>`. When the VC component is has performed a specific task it responds to the CM with:

```
<notification>hamsterLabReady</notification>
```

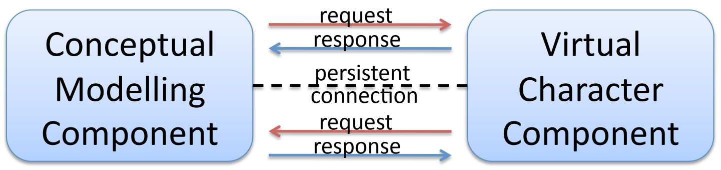

Figure 10.1: In the interaction between the CM and VC components, a persistent socket connection is established. Both the CM and the VC components can send requests to each other.

10.2 Communication of Questions

This section discusses the communications of questions, these are output of the question and answer generator subcomponent (as will be described in Deliverable 'D3.3 Question generation and answering'). Questions about model simulations are relevant in both the Teachable Agent and the quiz use case. Their relevant function calls are discussed respectively.

10.2.1 Teachable Agent Communication

The Teachable Agent interaction is started by a learner manipulation of the CM component. The CM sends the VC the following request `<request type="pickPet"/>`. The Teachable Agent selection screen appears on screen. After the learner has chosen a virtual character, the following options can be chosen **ask**, **explain**, **repeat** and **challenge**.

Selecting **Ask** generates a dialogue to ask a question about the model in progress: How will a quantity behave given a certain behaviour of another quantity? The VC sends the CM the request `<request>startTeachableAsk</request>`, and the ok response is sent back. A dialogue is then started in which the learner constructs the question. Upon completion of the dialogue, the question generator checks the simulation and returns a set of questions to the VC:

```
<assessmentQuestion id="1" type="whatHappensTo">
   <text>
      What happens to biomass if nutrients increases?
   </text>
   <answer type="solution">biomass increases.</answer>
   <relevantConcepts>
      <concept type="relation">
         owl_q_nutrients2, owl_q_biomass2
      </concept>
      <concept>
         owl_q_biomass2_Derivative
      </concept>
    </relevantConcepts>
```

```
</assessmentQuestion>
```

Selecting **Explain** generates an explanation of the answer given to an **Ask** question. This is especially relevant for longer causal paths. These are explained step by step. The request sent to the CM is `<request>startTeachableExplain</request>`. If no ask question has been asked yet the CM responds:

```
<response>
   <say>
      Please ask a question first.
   </say>
</response>
```

And if an explanation was already given the response is:

```
<response>
   <say>
      Please use the repeat button.
   </say>
</response>
```

If an Ask question was asked previously, the question generator consults the simulation and generates an explanation to send to the VC.

```
<response id="1" type="explanationWhatHappensTo">
   <say highlight="nutrients" gesture="up">
      If nutrients increases, and has a
      positive effect on carrying capacity...
   </say>
   <say highlight="carrying_capacity" gesture="up">
      Then carrying capacity increases.
      And if carrying capacity increases,
      and has a positive effect on biomass...
    </say>
    (...)
    <assessmentQuestion id="1" type="whatHappensTo"> ....
    </assessmentQuestion>
</response>
```

The TA will then go over this explanation and at each ingredient send a highlight request which signals the CM component to highlight the given ingredient.
`<highlight name="nutrients"/>`
The CM does this and responds with the screen coordinates so that the virtual character may point at the correct quantity.

```
<response>
   <quantityPosition quantity="nutrients" x="82" y="361" />
</response>
```

Selecting **repeat** reproduces the ask or explain behaviour. The VC sends the CM `<request>startTeachableRepeat</request>`. The response being the Ask or Explain question. If no ask or explain question was generated yet this is communicated the CM responds:

```
<response>
   <say>
      No question or explanation to repeat
      and challenges can be reviewed
      via the quizmaster.
   </say>
</response>
```

Selecting **challenge** will have the question generator compose a quiz made up of questions derived from an expert model associated with the assignment the student is working on. The answers to the questions are generated by the model made by the student. The VC sends the CM `<request>startTeachableChallenge</request>`. The response to the VC is:

```
<response type="teachableAgentChallenge">
   <quizSummary>
      <score correct="2" incorrect="3" percentage="40"/>
      <aboutModel>Stage1</aboutModel>
      <learnerID>dummyLearnerID</learnerID>
   </quizSummary>
   <questionlist>
      <question id="1" state="1" questiontype="14">
      <header></header>
      <text>
         What happens to cyanotoxins if carrying
 capacity decreases?
      </text>
      <answer type="solution">
         cyanotoxins decreases.
      </answer>
      <answer type="teachableAgent">
         Sorry, cyanotoxins is unknown to me.
      </answer>
      <relevantConcepts>
         <concept type="relation">
    owl_q_carrying_capacity, owl_q_cyanotoxins
         </concept>
      </relevantConcepts>
      </question>
      (...)
</questionlist>
</response>
```

The quizmaster will appear and present this quiz to the TA which will give the answers. The TA trained by the student hereby obtains a score indicating the quality of his knowledge and thereby the student's modelling effort.

10.2.2 Quiz communication

The quiz use case involves a multiple choice quiz concerning the currently loaded model to test the user understanding of this model. This use case is started from the VC which sends the CM: `<request type="startquiz"/>`. The VC component requires model content information (Section 7) for modelling learner knowledge. The request sent to the CM is `<request type="superstate"/>`. The CM responds with the XML representation of the OWL formalisation of the super state.

From the model of learner knowledge a focus is generated for the question generator containing the topics in the model that should be quizzed. An initial formalisation of the question request has been made in OWL (Figure 10.2). The question request consists of a single instance of the question request class, which has a number of relationships:

withConcept indicates which QR concept the question should be about, e.g. causal relationships, derivatives, or correspondences.

withSystemScope indicates which parts of system (in terms of entities) the question should be about.

withSubjectQuantity indicates which quantities the question should be about.

withNonSubjectQuantity indicates which quantities the question cannot be about.

The fillers of these relationships are all either concepts from the DynaLearn QR ontology or of concepts in the model or simulation representations. The questions are still returned in an XML representation (although concepts in the OWL representation are included). However, for the final version these questions will also be formalised in OWL.

The response to the question request is generated by the question generator which consults the model simulation and constructs a set of appropriate questions and multiple choice answers given the requested focus. The question list consists of questions, which in turn have question content, correct and incorrect answers and relevant concepts addressed by the question. The latter information is necessary to update the model of the user knowledge given his or her answer. The CM returns the following message to the VC:

```
<questionlist>
   <question id="1" state="1" questiontype="3">
   <header>Now look at state 1</header>
   <text>
      What will be the value of size in the
      next state?
   </text>
```

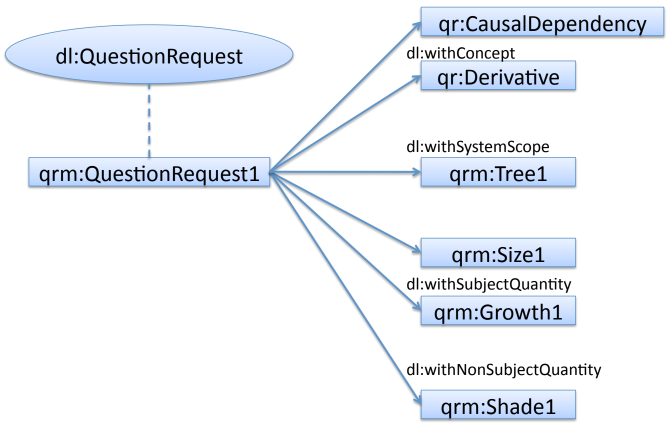

Figure 10.2: An example of the formalisation of a question request.

```
   <answer type="solution">
      size‘s value will rise from small to
      medium from this state to state 2
   </answer>
   <answer>
      Nothing will happen with size.
   </answer>
   <answer>
      size‘s value will rise from small to large
      from this state to state 2
   </answer>
   <relevantConcepts>
      <concept>
         owl_q_size1_Magnitude
      </concept>
   </relevantConcepts>
   </question>
   (...)
</questionlist>
```

Chapter 11

Semantic Technology Web Services

This section discusses the interaction between the Conceptual Modelling (CM) component and the Semantic Technology (ST) component. The ST component functions as a server on the web to which the CM component connects. Due to the difficulties of direct connections over the web, such as redirects, firewalls, and increased load on the server side, we have chosen not to use permanent connections between the CM and ST component. Instead we have opted for a synchronous request-response pattern. The CM component acts as a client that connects to the ST component, sends a request, and disconnects after it has either received a response, or the connection times out (Figure 11.1).

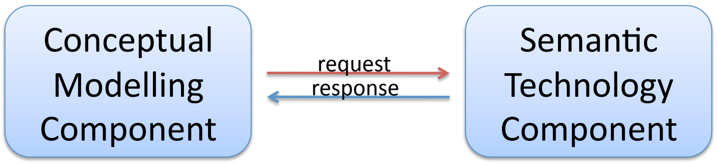

Figure 11.1: In the interaction between the CM and ST component, only the CM component is sending requests, to which the ST component responds. The protocol for knowledge exchange are SOAP Web Services.

The functionality of the ST component is externalised via a Web Services API. As such, the communication between the CM and ST is done via HTTP calls that exchange XML wrapped in SOAP envelopes (Figure 11.2) [19]. A SOAP Web Service client was implemented in SWI-Prolog to allow the CM component to interact with the ST component.

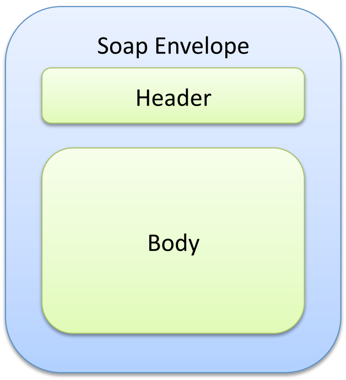

Figure 11.2: The SOAP Web Service protocol requires requests and responses to be wrapped in a SOAP envelope, which consists of a header and a body. The header consists of authentication information, while the body consists of the knowledge that is to be communicated.

11.1 Repository

The semantic repository allows the storage and retrieval or QR models. Furthermore, it also allows for search for specific models based on keywords, and a full listing of all the models in the repository. The results of a search and a full listing is a list of URIs that identify specific models. Storage, as can be seen in Figure 1.1, exchanges a QR model (described in Section 6). However, in order to utilize models for the Ontology-Based feedback and Recommendation tasks, the extracted vocabularies (Section 8) are also stored. More details on the specific calls will be detailed in deliverable 'D4.1 Semantic repository and ontology mapping'.

11.2 Grounding

The grounding functionality takes a concept in a QR model and links it to a concept an external resource such as DBPedia or WordNet. The grounding establishes a common vocabulary between QR models so that they can be compared. The grounding functionality makes use of the QR model representation (Section 6). The functionality returns a list of possible groundings to the CM components including URIs, labels, and descriptions. The specific calls will be detailed in deliverable 'D4.1 Semantic repository and ontology mapping'.

11.3 Ontology-Based Feedback and Recommendations

The Ontology-Based Feedback and Recommendation functionality take a QR model and an extracted domain vocabulary as input, and generate feedback or recommendations based on the models (and their vocabularies) in the Semantic Repository. As such, it used the model representation (Section 6) and the extracted vocabulary (Section 8). The specific functionality and the form of the output will be described in deliverables 'D4.2 Ontology based feedback on model quality' and 'D4.3 Model-based and memory-based Collaborative Filtering algorithms for complex knowledge models'. Also, for M18, the Ontology-Based Feedback will only be available via the CM component. From M18 onward, the feedback will be provided as input for the tutorial planner task (deliverable D5.4 'Integration of tutorial planner and animated agent'). The VC component will request the CM component for feedback and recommendations. The CM component will call the ST component to get this knowledge, format the knowledge so that it is easy to use for the VC component and send it to the VC component. The VC component will then create a dialogue based on a specific learning goal.

11.4 Authentication

The ST component contains knowledge that should not be accessible for learners at all times. For example, when creating a model about climate change, a teacher might choose to have access to existing models about climate change to be limited. Furthermore, in the Teachable Agent use-case in which a learner should construct a model that corresponds to an expert model, the learner should not be able to download this expert model from the repository. As such, there is a need for user authentication in the CM-ST interaction.

The user authentication has been implemented using SOAP Message Security 1.1. In every request to the server, a security element is added to the SOAP header. The username and password are stored in order to add them to each request. The canonical form of the SOAP header is:

```
<soapenv:Header
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
   <wsse:Security xmlns:wsse=
"http:// (...) oasis-200401-wss-wssecurity-secext-1.0.xsd"
   soapenv:mustUnderstand="1">
      <wsse:UsernameToken>
         <wsse:Username>
            username@example.com
         </wsse:Username>
         <wsse:Password Type=
"http:// (...) oasis-200401-wss-username-token-profile-1.0\#PasswordText">
            secretpassword
         </wsse:Password>
      </wsse:UsernameToken>
   </wsse:Security>
</soapenv:Header>
```

User account can be made and managed via the web[1]. The user administration allows for the assignment of roles. As such, teachers can be distinguished from students.

[1]http://elnath.dia.fi.upm.es/DLadmin/

Chapter 12

Discussion and Conclusion

The DynaLearn Interactive Learning Enviroment (ILE) consists of the Conceptual Modelling (CM), Virtual Character (VC) and Semantic Technology (ST) components. This document describes the formalisation of QR models, simulations and other key knowledge structures. These formalisations provide the basis for all the knowledge exchange between the three components.

The formalisation of QR has been estabished by applying an ontological perspective on QR models (Section 4). Based on this perspective the QR Ontology has been developed (Section 5). The QR models and simulations are a natural extension to this ontology (Sections 6 and 7). However, for the Ontology-Based Feedback and Recommendation tasks an additional alternative knowledge representation had to be developed (Section 8). Similarly, for the support knowledge that has to be communicated via a dialogue also an alternative representation is required (Section 9).

The communication between the CM and the VC components is established using socket communication. The connection between these components is persistent. Between the CM and ST components however, the connection is not persistent. The ST API is externalised via Semantic Web services that are called by the CM component.

There are a few open issues to be resolved in the remainder of the project. Firstly, the precise formalisation of each of the outputs of specific tasks has to be developed. These formalisation will be based on the QR ontology, or refer to specific URIs within its namespace, or URIs within specific models. Secondly, the Ontology-Based Feedback and Recommendation output has to be communicated with learners using the virtual characters. Currently, the CM component communicates with the ST for feedback. However, in the future the VC will send a request to the CM for feedback or recommendations. The CM wil in turn interact with the ST to retrieve the results, and forward them to the VC.

To summarize, the goal of establishing knowledge exchange between the different DynaLearn components has been successfully completed. The formalisation of QR models and simulations in the Web Ontology Language has proved to be a stable basis to derive representations for different tasks.

Bibliography

[1] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider. *The description logic handbook: theory, implementation, and applications*. Cambridge University Press, New York, NY, USA, 2nd edition, August 2007.

[2] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL web ontology language reference. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/owl-ref/ (visited August 2009).

[3] Sean Bechhofer, Phillip Lord, and Raphael Volz. Cooking the semantic web with the OWL API. In Dieter Fensel, Katia P. Sycara, and John Mylopoulos, editors, *Proceedings of the 2nd International Semantic Web Conference (ISWC 2003)*, volume 2870 of *Lecture Notes in Computer Science*, pages 659–675, Sanibel Island, Florida, USA, October 2003. Springer.

[4] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, May 2001.

[5] D. Bobrow, B. Falkenhainer, A. Farquhar, R. Fikes, K. Forbus, T. Gruber, Y. Iwasaki, , and B. Kuipers. A compositional modeling language. In Y. Iwasaki and A. Farquhar, editors, *Proceedings of the Tenth International Workshop for Qualitative Reasoning(QR-96)*, AAAI Technical Report WS-96-01, pages 12–21, Menlo Park, 1996. AAAI Press.

[6] B. Bredeweg, F. Linnebank, A. Bouwer, and J. Liem. Garp3 - workbench for qualitative modelling and simulation. *Ecological informatics*, 4(5-6):263–281, 2009.

[7] B. Bredeweg, P. Salles, and T. Nuttle. Using exogenous quantities in qualitative models about environmental sustainability. *AI Communications*, 20(1):49–58, 2007.

[8] Bert Bredeweg and Paulo Salles. *Handbook of Ecological Modelling and Informatics*, chapter Mediating conceptual knowledge using qualitative reasoning, pages 351–398. WIT Press, Southampton, UK, 2009.

[9] Bert Bredeweg and Paulo Salles. Qualitative models of ecological systems – editorial introduction. *Ecological Informatics*, 4(5-6):261 – 262, 2009. Special Issue: Qualitative models of ecological systems.

[10] B. Bredeweg (ed.), E. André, N. Bee, R. Bühling, J. M. Gómez-Pérez, M. Häring, J. Liem, F. Linnebank, N. Thanh Tu, M. Trna, and M. Wißner. Technical design and architecture. Technical Report Deliverable D2.1, STREP project FP7 no. 231526, DynaLearn, 2009.

[11] Paul Buitelaar, Philipp Cimiano, Anette Frank, Matthias Hartung, and Stefania Racioppa. Ontology-based information extraction and integration from heterogeneous data sources. *Int. J. Hum.-Comput. Stud.*, 66(11):759–788, 2008.

[12] Alberto J. Cañas, Greg Hill, Roger Carff, Niranjan Suri, James Lott, Gloria Gómez, Thomas C. Eskridge, Mario Arroyo, and Rodrigo Carvajal. Cmaptools: A knowledge modeling and sharing environment. In A. J. Cañas, J. D. Novak, and F. M. González, editors, *Concept maps: Theory, methodology, technology. Proceedings of the first international conference on concept mapping*, pages 125–133, Universidad Pública de Navarra, Pamplona, Spain, 2004.

[13] K. de Koning, B. Bredeweg, J. Breuker, and B. Wielinga. Model-based reasoning about learner behaviour. *Artificial Intelligence*, 117(2):173–229, 2000.

[14] K. D. Forbus. Qualitative process theory. *Artificial Intelligence*, 24(1-3):85–168, December 1984.

[15] K. D. Forbus, K. Carney, B. L. Sherin, and L. C. Ureel II. Vmodel - a visual qualitative modeling environment for middle-school students. *AI Magazine*, 26(3):63–72, 2005.

[16] Andrew Ford. *Modeling the Environment, Second Edition*. Island Press, second edition edition, November 2009.

[17] Avigdor Gal and Pavel Shvaiko. *Lecture Notes In Computer Science. Advances in Web Semantics I: Ontologies, Web Services and Applied Semantic Web*, chapter Advances in Ontology Matching, pages 176–198. Springer-Verlag, Berlin, Heidelberg, 2009.

[18] M. R. Genesereth and R. E. Fikes. Knowledge interchange format, version 3.0 reference manual. Technical Report Technical Report Logic-92-1, Stanford University Logic Group, 1992.

[19] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Lafon Yves. SOAP version 1.2 part 1: Messaging framework (second edition). W3C recommendation, W3C, April 2007.

[20] Pat Hayes, Thomas C. Eskridge, Raul Saavedra, Thomas Reichherzer, Mala Mehrotra, and Dmitri Bobrovnikoff. Collaborative knowledge capture in ontologies. In *K-CAP '05: Proceedings of the 3rd international conference on Knowledge capture*, pages 99–106, New York, NY, USA, 2005. ACM.

[21] Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph. OWL 2 web ontology language primer. W3C recommendation, World Wide Web Consortium, October 2009.

[22] Ian Horrocks. Semantic web: the story so far. In *W4A '07: Proceedings of the 2007 international cross-disciplinary conference on Web accessibility (W4A)*, pages 120–125, New York, NY, USA, 2007. ACM Press.

[23] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A semantic web rule language combining OWL and RuleML. W3C member submission, W3C, May 2004.

[24] Aditya Kalyanpur, Bijan Parsia, Evren Sirin, Bernardo Cuenca-Grau, and James Hendler. Swoop: A 'web' ontology editing browser. *Journal of Web Semantics*, 4(2):144–153, June 2006.

[25] H. Knublauch, R. Fergerson, N.F. Noy, and M.A. Musen. The Protege OWL plugin: An open development environment for semantic web applications. In S. A. McIlraith, D. Plexousakis, and F. van Harmelen, editors, *International Semantic Web Conference*, pages 229–243, Hiroshima, Japan, November 2004. Springer.

[26] Krittaya Leelawong and Gautam Biswas. Designing learning by teaching agents: The betty's brain system. *International Journal of Artificial Intelligence in Education*, 18:181–208, August 2008.

[27] Jochem Liem, Wouter Beek, and Bert Bredeweg. Multi use level workbench. Technical Report Deliverable D3.1, STREP project FP7 no. 231526, DynaLearn, 2010.

[28] J. D. Novak and D. B. Gowin. *Learning how to learning*. Cambridge University Press, Cambridge and New York, 1984.

[29] N. Noy and A. Rector. Defining n-ary relations on the semantic web. W3C working group note, April 2006. http://www.w3.org/TR/swbp-n-aryRelations/.

[30] A. Rector. Representing specified values in OWL: "value partitions" and "value sets". W3C working group note, May 2005. http://www.w3.org/TR/swbp-specified-values/.

[31] A. Rector, N. Drummond, M. Horridge, J. Rogers, H. Knublauch, R. Stevens, H. Wang, and C. Wroe. OWL pizzas: Practical experience of teaching OWL-DL: Common errors & common patterns. In E. Motta, N. Shadbolt, A. Stutt, and N. Gibbins, editors, *Proceedings of the European Conference on Knowledge Acquisition*, pages 63–81, 2004.

[32] Guus Schreiber, Alia Amin, Mark van Assem, Viktor de Boer, Lynda Hardman, Michiel Hildebrand, Laura Hollink, Zhisheng Huang, Janneke van Kersen, Marco de Niet, Borys Omelayenko, Jacco van Ossenbruggen, Ronny Siebes, Jos Taekema, Jan Wielemaker, and Bob Wielinga. MultimediaN E-Culture demonstrator. In Isabel Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel

Schwabe, Peter Mika, Mike Uschold, and Lora Aroyo, editors, *Proceedings of the Fifth International Semantic Web Conference (ISWC'06)*, number 4273 in Lecture Notes in Computer Science, pages 951–958, Athens, Georgia, USA, november 2006. Springer-Verlag.

[33] Pavel Shvaiko and Jérôme Euzenat. Ten challenges for ontology matching. In *OTM '08: Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part II on On the Move to Meaningful Internet Systems*, pages 1164–1182, Berlin, Heidelberg, 2008. Springer-Verlag.

[34] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics*, 5(2):51–53, 2007.

[35] Barry Smith and Christopher Welty. FOIS introduction: Ontology—towards a new synthesis. In *FOIS '01: Proceedings of the international conference on Formal Ontology in Information Systems*, pages 3–9, New York, NY, USA, 2001. ACM.

[36] The Gene Ontology Consortium. Gene ontology: tool for the unification of biology. *Nature Genetics*, 25:25–29, 2000.

[37] D. Tsarkov and I. Horrocks. FaCT++ description logic reasoner: System description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 292–297. Springer, 2006.

[38] Frank van Harmelen and Grigoris Antoniou. *A Semantic Web Primer*. The MIT Press, 2004.

[39] G. van Heijst, S. Falasconi, A. Abu-Hanna, G. Schreiber, and M. Stefanelli. A case study in ontology library contruction. *Artificial Intelligence in Medicine*, 7(3):227–255, June 1995.

[40] J. Wielemaker, G. Schreiber, and B. Wielinga. Using triples for implementation: the Triple20 ontology-manipulation tool. In Y. Gil, E. Motta, V. R. Benjamins, and M. A. Musen, editors, *International Semantic Web Conference*, pages 773–785, Berlin, Germany, November 2005. Springer Verlag. LNCS 3729.

[41] Jan Wielemaker, Zhisheng Huang, and Lourens van der Meij. SWI-Prolog and the web. *Theory and Practice of Logic Programming*, 8(3):363–392, 2008.