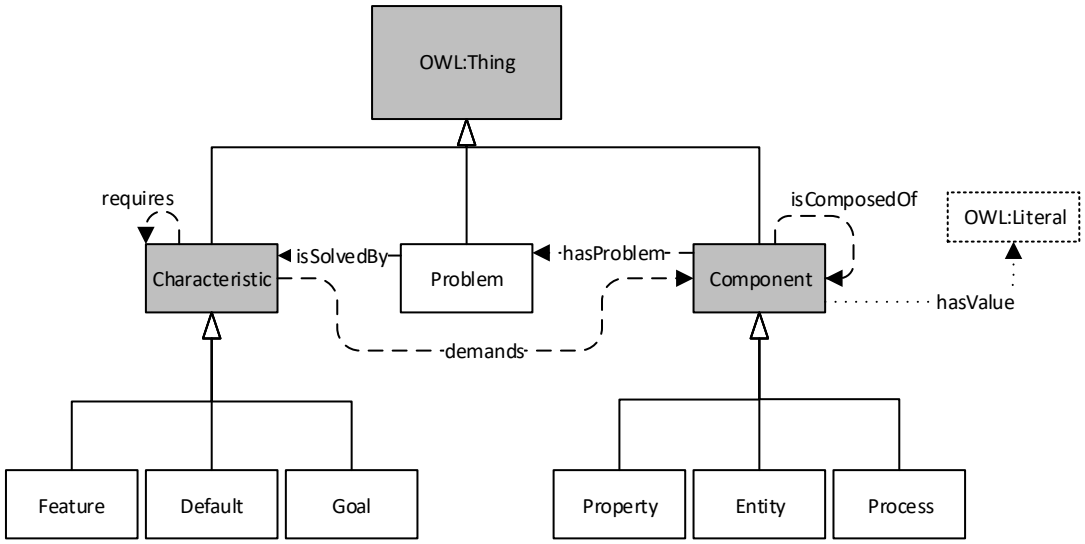
Contents

1 Glossary

Individual – an instance of an OWL class
Referenced Individual – the individual is explicitly referenced in a SeML script.
Instantiated Class – an individual of that class was created
owl:Class – defines a group of individuals that belong together because they share some properties
Subclass – A class may be derived from an existing base class. This idea of inheritance implements the is-a relationship.
owl:Thing – is the built-in most general class. It is the class of all individuals and is a superclass of all OWL classes.
owl:Nothing – is the built-in most specific class. It is the class that has no instances and is a subclass of all OWL classes.
owl:Property – Properties can be used to state relationships between individuals or from individuals to data values. Examples of properties include hasChild, hasRelative, hasSibling, and hasAge. The first three can be used to relate an instance of a class Person to another instance of the class Person (and are thus occurrences of ObjectProperty), and the last (hasAge) can be used to relate an instance of the class Person to an instance of the datatype Integer (and is thus an occurrence of DatatypeProperty).
OP – Object Property
DP – Data Property
SeML – Semantically-enriched Modelling Language
SeMLI – Semantically-enriched Modelling Language Infrastructure

2 Classes

2.1 Upper Ontology



Class	Purpose
Characteristic	Model Characteristic
Feature	Model feature
Default	Model default Characteristic
Goal	Model goal
Component	Model description block
Property	Constants or Variables (e.g. basic types, composite types)
Entity	Structure (e.g. Program, File, Module, Package, Class)
Process	Continuous Behavior (e.g. Function, Method, Sequential instructions)
Problem	Problem raised by a Component

A class that contains no individual and is instantiable, i.e., does not contain annotations stating otherwise, is automatically instantiated by the DSL, which creates a single individual with the same name as the class (but starting with lower case).

2.1.1 Characteristic

A Characteristic represents features (attributes) and goals (desired results) of the model. These create an abstract layer above all Component classes.

Characteristic classes might be used to:

- Require or reject the instantiation of other Characteristics;
- Impose existential restrictions on Component individuals;
- Influence solution proposals, eliminating options that would generate errors by excess;
- Trigger SWRL rules, when instanced.

Characteristic classes can be instanced in two ways: either they are explicitly referenced in the DSL or they are required by other Characteristic classes. The Default Characteristic is special because it is used, even when not referenced, although it is not instanced in the ontology.

The DSL reports to the user which Characteristic classes are in use, every time the model is validated. If there is a non-solved requirement, it will display an error.

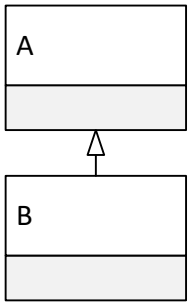
2.2 Required Individuals

Individuals must be static or required to be referenced in the DSL. Otherwise, an error will occur. If the individual is static, it will be instantiated when the model is created, being its references automatically allowed. Those individuals might be used on the left part of a relation. Individuals on the right part of a relation become allowed to be referenced in the left part of subsequent relations.

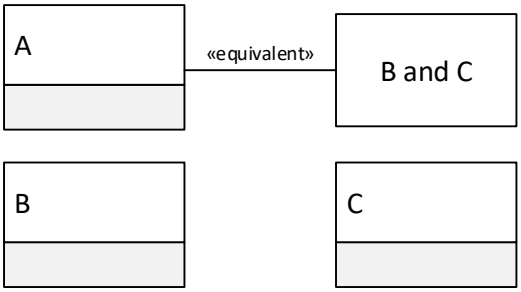
2.3 Theory

2.3.1 Transformation from UML Notation to OWL Notation

A class of an individual might be inferred from an equivalent class or from an SWRL rule.



Class: A
Class: B
SubClass Of: A



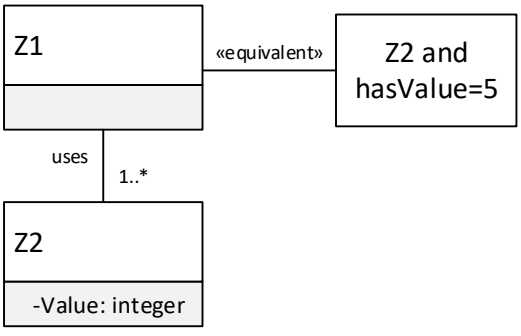
Class: A
Equivalent To: B and C
Class: B
Class: C

2.4 Examples

2.4.1 Example I

This example shows how to create classes equivalent to anonymous classes. It could be equally accomplished with an SWRL rule.

Z1 is equivalent to the anonymous class “Z2 and hasValue=5”. When z2 is referenced in the DSL, it must be assigned an integer. If that integer is equal to 5, z2 will inherit the restrictions of the Z1 class. This means that z2 must use some individual of Z2. However, if the integer is different from 5, z2 can’t use any individual, because the restrictions is not required.



Class: Z1
Equivalent To: Z2 and hasValue some {5}
SubClass Of: uses some Z2
Class: Z2
SubClass Of: hasValue some xsd:integer

3 Annotations

3.1 Implementation

3.1.1 Theory

3.1.1.1 Ontology perspective

The implementation stage is characterized by a strong relation between the DSL and user-defined external tools. Special annotations can be used to map ontological elements to implementation artefacts. Each annotation represents a different behavior as can be seen in the following table, and can be asserted multiple times on the same entity.

Annotation Property	Trigger Condition	Arguments	Assertion (Individual)	Assertion (Property)
ImplInd	instanced individual	individual*	tool,method [,priority[,annotProp]]	-
ImplOP	instanced individual and OP	individual1*, individual2*	tool, method, property[,priority [,annotProp]]**	tool,method[,priority [,annotProp]]
ImplDP	instanced individual and DP	individual*, literal	tool, method, property[,priority [,annotProp]]	tool,method[,priority [,annotProp]]

*source of annotations (ImplArg or user-defined)
**assertion performed on first individual (relation's domain)

The annotation property is defined in the upper ontology. In the implementation stage, the DSL will look for trigger conditions to call a specific method of the provided tool. The command order is defined by the priority parameter, which represents a positive integer. A higher value means lower priority. If not specified, the parameter's value is 10.

By default, the arguments sent to the tool are of type *ImplArg*, unless the user states otherwise when asserting the annotation. These arguments are taken from predefined ontological elements and sent to the tool's method. Each behavior is asserted in a certain element and contains a string with the tool name, method name and other parameters which depend on the chosen behavior.

3.1.1.2 Tool Perspective

Every public method must be static and return a valid type. These return types can be consulted in the following table, along with the respective behavior after invocation.

Return Type	Return Value	Behavior after invocation
null	-	-
boolean	false	Display generic error message
	true	-
String	string	Display returned string
	null	-

If an exception is thrown, the SeMLI will display the corresponding error detail message.

There are optional methods which are called at key moments during the implementation stage. These are documented in the next table. The return types are not fixed and follow the same rules that were stated in the last table.

Method signature	Definition
setSrc(String)	setSrc is called at the beginning of the implementation. The only argument is the absolute path of the source folder, inside the current project's folder.
apply()	apply is called at the end of the implementation.

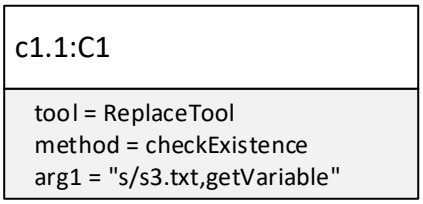
Every other public method must have one argument only, of type *ArrayList<List<String>>.* The list is composed of several string lists, each representing an ontological element. Each element may have several annotations of type *ImplArg* or user-defined equivalents.

3.1.2 Examples

3.1.2.1 Example I

This example shows how to perform a simple mapping from one individual of the model to the matching implementation artefact.

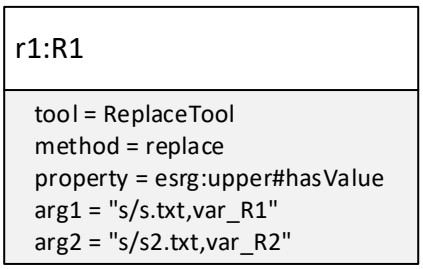
When the user runs the implementation stage, if c1.1 was referenced, the method "CheckExistence" of the tool "ReplaceTool" will be called with one argument, "s/s3.txt,getVariable". The mentioned tool will split the comma separated values of the provided argument to obtain a relative path of the source file and the string which must be found at least once.



3.1.2.2 Example II

This example shows how to replace contents of source files using annotations asserted on properties.

The method *replace* will receive annotations from 2 ontological elements: r1 and the literal to which it is linked via the *hasValue* DP. The string "var_R1" will be replaced by the literal's value every time it is found in the file "s.txt". The analogous behavior can be observed with the string "var_R2" and the file "s2.txt".

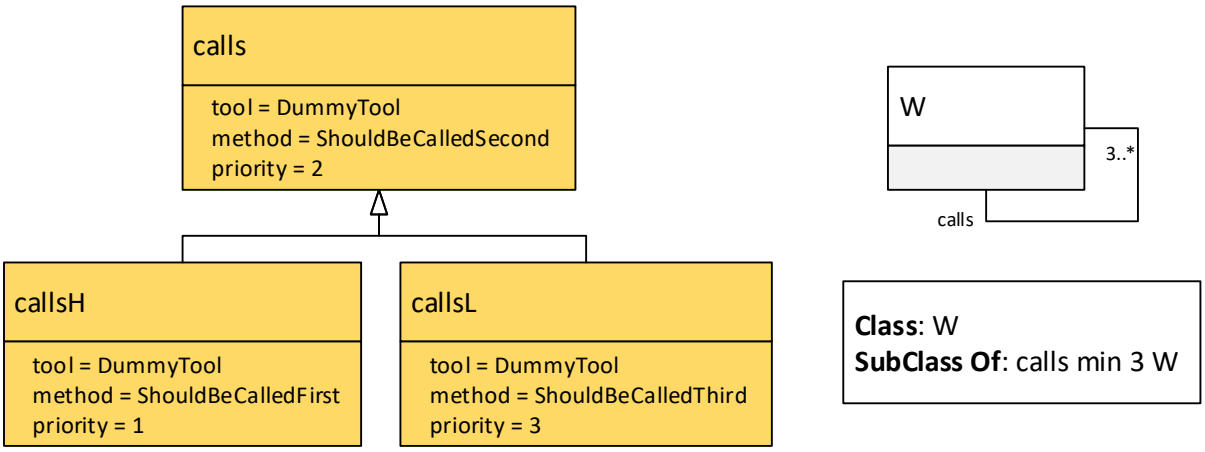


3.1.2.3 Example III

This example shows a more complex implementation procedure involving precedence, property triggers and property inheritance.

A class W individuals must call 3 other individuals of the same class. For this relation, the Object Property might be “calls” or any of its sub properties. Each Object Property has its own implementation data. In this example, since “calls” also has implementation data, if “callsH” is instantiated, both tools will be invoked for the same relation.

Every OP is invoking its tool with different priorities. This will dictate the precedence of invocations when the SeMLI is in the implementation stage.



3.2 Modifiers

3.2.1 Theory

There are several modifiers which can be set via OWL annotations.

Annotation Property	Target Element	Meaning
NonInstantiable	OWL Class	Class won’t generate individuals
DefaultValue	OWL Individual	Default value for the *hasValue* DP
ArraySize	OWL Individual	Number of individuals -1 to be generated
StaticIndividual	OWL Individual	Individual is a model entry point

The *ArraySize* annotation indicates the number of times the targeted individual (TI) will be replicated. The naming convention for the replicated individuals is based on the TI’s name, which will act as the first element of the array. Therefore, the last number found on its name will be the incrementing index and the first part will be the base name for all elements.

If the TI’s name contains only digits or it doesn’t end with a digit, the starting index will be considered 0 and the elements’ base name will be equal to the TI’s name.

4 Object Properties

4.1 Overview

The property must be required in order to be allowed in the DSL. Therefore, if the user does not create a restriction on any of the properties classes or declares the property as static, it cannot be referenced in the DSL.

4.2 Required Object Properties

Relations with object properties must be required, to be allowed in the DSL. Relations can be required by applying special restrictions on the classes of the individual which is on the left part of a relation. These restrictions must have a requirement behavior.

OP	Purpose	Domain	Range
demands	DSL Integration	Characteristic	Component
requires	DSL Integration	Characteristic	
solves	DSL Integration	Characteristic	Problem
uses	Normalize	Component	
isUsedBy	Normalize	Component	
isComposedOf	Normalize	Component	
isPartOf	Normalize	Component	

4.3 Component Relations

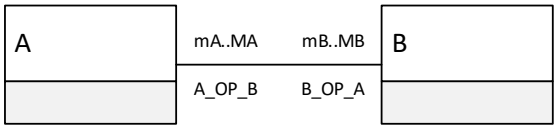
4.3.1 Theory

The user may create its own Object Properties in its own ontology without having to extend any existent property. However, to normalize knowledge across all domains, some Object Properties were created in the Upper Ontology: “uses”, “isComposedOf” and the respective inverse properties.

Restriction Type	Behavior
some	Requirement
only	Restriction
value	Requirement
Self	Requirement
min	Requirement
max	Restriction
exactly	Requirement and Restriction

A `Description` might contain unions (or logical operator) and intersections (and logical operator). Logical negation is only allowed if nested inside a `primary`.

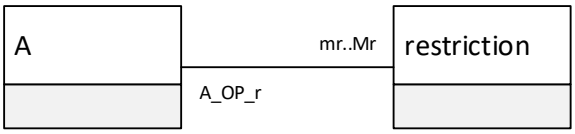
4.3.1.1 Transformation from UML Notation to OWL Notation



Class: A
SubClass Of: `restriction`
Class: B
SubClass Of: `restriction`

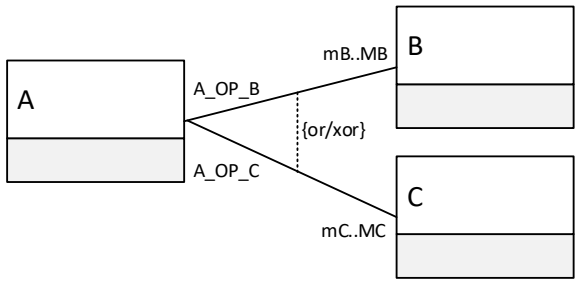
mA – minimum instances of A related with one instance of B
MA – maximum instances of A related with one instance of B
mB – minimum instances of B related with one instance of A
MB – maximum instances of B related with one instance of A
A_OP_B – object property pointing from A to B
B_OP_A – object property pointing from B to A
Multiplicity might also be represented as a number if the minimum and maximum values are equal

If A is connected to an anonymous class:



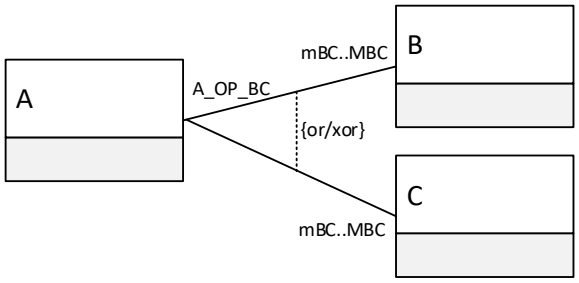
Class: A
SubClass Of: `restriction`

OR/XOR Relations:



Class: A
SubClass Of: `description` (or)
(xor requires particular analysis)
Class: B
Class: C

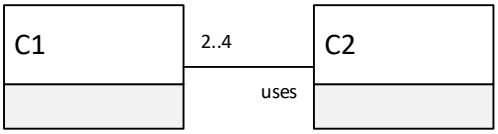
If (A_OP_B=A_OP_C) and (mb..MB=mc..MC):



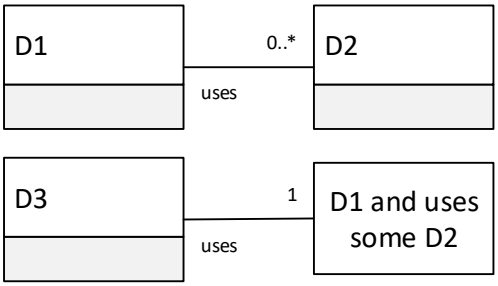
Class: A
SubClass Of: `description` (or)
(xor requires particular analysis)
Class: B
Class: C

4.3.2 Examples

4.3.2.1 Example I



Class: C1
Class: C2
SubClass Of: (uses min 2 C1) and (uses max 4 C1)



Class: D1
SubClass Of: uses min 0 D2
Class: D2
Class: D3
SubClass Of: uses exactly 1 (D1 and uses some D2)

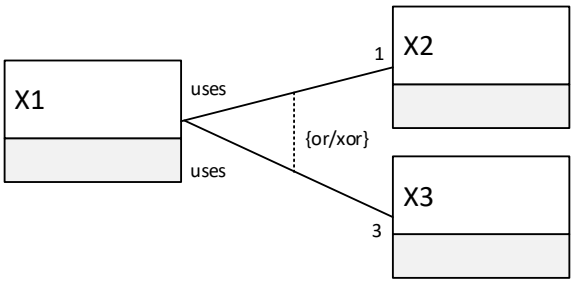
4.3.2.2 Example II

This example shows a simple variability using different multiplicities.

Using the OR logical operator, X1 requires X2 or X3. If both are referenced, there is no problem. While using the XOR logical operator, X1 requires exclusively X2 or X3. To accomplish this, a maximum number of related individuals is specified. Each case may have different solutions.

If the multiplicity of both relations was the same, the superclass expression that limits exclusiveness could be replaced with “uses exactly 2 (X2 or X3)”.

OR/XOR Relations:



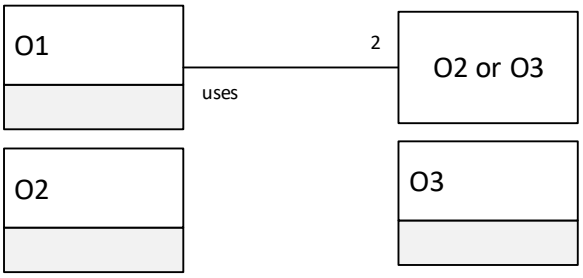
Class: X1
SubClass Of:
- (uses exactly 1 X2) or (uses exactly 3 X3)
- uses max 3 (X2 or X3) (if xor)

Class: X2
Class: X3

4.3.2.3 Example III

If the alternatives are expressed inside a `primary` symbol, the operand classes may be used interchangeably.

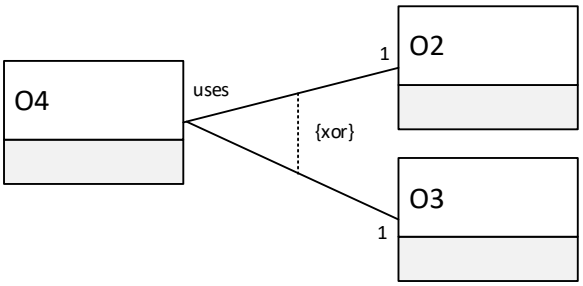
O1 uses 2 individuals of O2 or O3 but there is no distinction between both classes. This could also be accomplished by creating an anonymous class, equal to the union of O2 and O3.



Class: O1
SubClass Of: uses exactly 2 (O2 or O3)
Class: O2
Class: O3

4.3.2.4 Example IV

The previous example can also be expressed in this notation if the multiplicity of both relations is 1. O4 uses exclusively O2 or O3.



Class: O4
SubClass Of: uses exactly 1 (O2 or O3)
Class: O2
Class: O3

4.4 Characteristic Relations

4.4.1 Theory

4.4.1.1 Existential Restrictions

Any Characteristic can have existential restrictions. If a Characteristic requires the exclusive existence of one or another Characteristic, this mean that the user can only reference on of them.

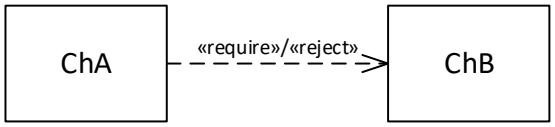
The Characteristic class **Default** is used, even if it is not explicitly referenced. It is asserted in the Upper Ontology.

The following table shows the allowed restrictions which can be made using the Object Property “requires”.

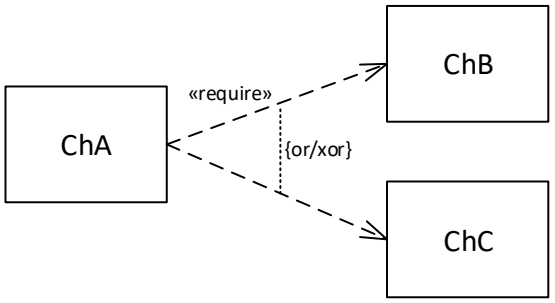
Restriction Type	Cardinality	`primary`	Meaning
exactly	1	`classIRI`	Characteristic requires Characteristic

	0	`classIRI`	Characteristic rejects Characteristic
	1	`classIRI` (or `classIRI`)	Characteristic requires ∃! Characteristic (the others are rejected)
some	-	`classIRI` (or `classIRI`)	Characteristic requires ∃ Characteristic

4.4.1.2 Transformation from UML Notation to OWL Notation



Class: ChA (Characteristic)
SubClass Of: requires exactly 0/1 ChB
Class: ChB (Characteristic)



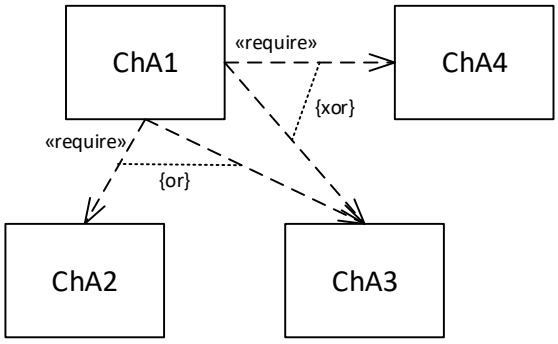
Class: ChA (Characteristic)
SubClass Of: requires (exactly 1)/some (ChB or ChC)
Class: ChB (Characteristic)
Class: ChC (Characteristic)

4.4.2 Examples

4.4.2.1 Example I

This example shows the interaction between Characteristic classes when using requirement relations with logical operators.

ChA1 requires (ChA2 or ChA3) and requires exclusively (ChA3 or ChA4). This example assumes that ChA1 is referenced. If the user references ChA2, there is still a variability to solve. If the user references ChA3, ChA2 can be referenced but ChA4 cannot, because of the exclusive requirement. If the user references ChA4, ChA3 cannot be referenced and ChA2 will automatically be used.



Class: ChA1 (Goal)
SubClass Of:
- requires some (ChA2 or ChA3)
- requires exactly 1 (ChA3 or ChA4)

Class: ChA2 (Feature)
Class: ChA3 (Feature)
Class: ChA4 (Feature)

4.5 Characteristic to Component Relations

4.5.1 Theory

Characteristics create an abstract layer above all Component classes. The restrictions imposed by Characteristic classes are useful to validate the model but also influence solution proposals, eliminating options that would generate errors by excess.

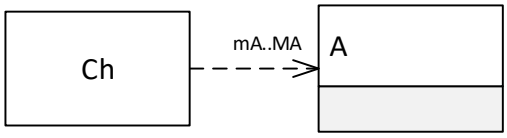
4.5.1.1 Restrictions

The following table shows which restriction can be made using the Object Property “demands”.

Restriction Type	Meaning	Behavior
some	Model contains some primary	Requirement
only	Model contains only primary	Restriction
value	Model contains individual	Requirement
Self	Not Supported	-
min	Model contains min n primary	Requirement
max	Model contains max n primary	Restriction
exactly	Model contains exactly n primary	Requirement and Restriction

A `Descriptions` might contain unions (or logical operator) and intersections (and logical operator). Logical negation is only allowed if nested inside a `primary`.

4.5.1.2 Transformation from UML Notation to OWL Notation



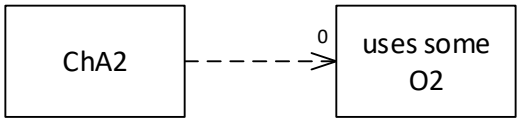
Class: A (Component)
Class: Ch (Characteristic)
SubClass Of: restriction

4.5.2 Examples

4.5.2.1 Example I

This example illustrates the effects of Characteristic imposed restrictions on the model's validation and solution proposals.

ChA2 requires the existence of zero individuals which belong to the anonymous class "uses some O2". If ChA2 is used and o1 is referenced the solution will contain only one proposal.



Class: Ch (Characteristic)
SubClass Of: demands max 0 (uses some O2)

4.6 Problem related relations

4.6.1 Theory

Each Problem class describes an obstacle of one or more Component classes. Each Problem might be solved by one or more Characteristics. If a Component is instantiated and it is linked to a Problem, at least one of the instantiated Characteristics must solve it. Otherwise, an error will be thrown.

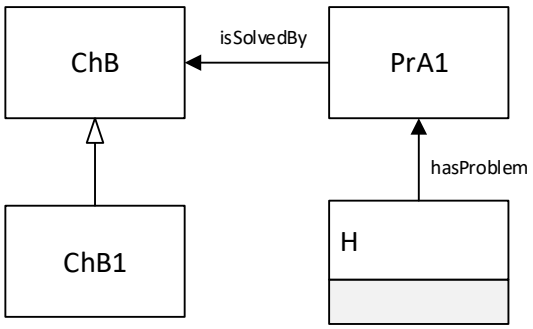
Problems might be connected to Characteristics via the "isSolvedBy some `classIRI`" `restriction`. Components might be connected to Problems via the "hasProblem some `classIRI`" `restriction`.

4.6.2 Examples

4.6.2.1 Example I

This example shows how to solve a Problem created by a Component using a Characteristic.

Component H has a Problem, PrA1. That Problem is solved if ChB is instantiated. Since ChB1 is-a ChB, if ChB1 is instantiated, then, the problem is also solved.



Class: ChB (Feature)
Class: ChB1 (Feature)
SubClass Of: ChB
Class: PrA1 (Problem)
SubClass Of: isSolvedBy some ChB
Class: H
SubClass Of: hasProblem some PrA1

5 Data Properties

5.1 Overview

OP	Purpose
hasValue	DSL Integration
hasReport	DSL Support (not instantiable)
hasError	DSL Integration
hasWarning	DSL Integration
hasInfo	DSL Integration

5.2 Required Data Properties

As with Object Properties, Data Properties must be required to be allowed in the DSL. Since the DSL only supports assignments with the “hasValue” Data Property, it is also the only one which needs to be required. This can be accomplished using any of the two allowed restrictions regarding “hasValue”.

5.3 Reports

5.3.1 Theory

Reports are intuitive and descriptive messages for the DSL User, for when a certain incongruence occurs or a specific combination of components raises a warning or a special information for the user.

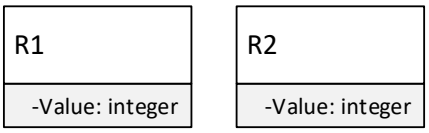
These are static or dynamically created using SWRL rules. There are 3 data properties *hasError*, *hasWarning* and *hasInfo* which can be used to generate reports. Any datatype is allowed.

5.3.2 Examples

5.3.2.1 Example I

This example shows how to create a report, namely, an information report, using SWRL rules.

For each individual of R1 and R2, the SWRL rule will create an information report that sums the values assigned to both individuals through the “hasValue” Data Property. The resulting report will be attached to the evaluated individual of R1 using the “hasInfo” Data Property.



If r1 ∈ R1, r2 ∈ R2, r1 hasValue v1, r2 hasValue v2 **then** r1 hasInfo “r1 + r2 = ?”

SWRL Rule: R1(?r1) ^ R2(?r2) ^ hasValue(?r1, ?v1) ^ hasValue(?r2, ?v2) ^ swrlb:add(?v3, ?v1, ?v2) ^ swrlb:stringConcat(?s, "r1 + r2 = ", ?v3) -> hasInfo(?r1, ?s)

5.4 Value and user-defined data properties

5.4.1 Theory

5.4.1.1 Transformation from UML Notation to OWL Notation

A	**OWL 2 Manchester Syntax Transformation**
-AttributeName: TypeName[*] {constraint}	**Class**: A **SubClass Of**: `description`

User-defined Data Property
AttributeName: String
TypeName: [xsd:] RDF Datatype
(omitted if contained in constraint)
[*]: Multiplicity. Options:
- Omitted if the multiplicity is 1
- [<lower-bound> '..' <upper-bound>]

constraint: literal constraint

hasValue Data Property
AttributeName: Value
TypeName: [xsd:] RDF Datatype
(omitted if contained in constraint)
[*]: Multiplicity. Options:
- Omitted if the multiplicity is 1
- [0..1] if value is not mandatory

constraint: literal constraint

Class: A
SubClass Of: hasValue some/only dataPrimary

dataRange ::= dataConjunction 'or' dataConjunction { 'or' dataConjunction } | dataConjunction
dataConjunction ::= dataPrimary 'and' dataPrimary { 'and' dataPrimary } | dataPrimary
dataPrimary ::= ['not'] dataAtomic
dataAtomic ::= Datatype | '{' literalList '}' | datatypeRestriction | '(' dataRange ')'
datatypeRestriction ::= Datatype '[' facet restrictionValue { ',' facet restrictionValue } ']'
facet ::= 'length' | 'minLength' | 'maxLength' | 'pattern' | 'langRange' | '<=' | '<' | '>=' | '>'
restrictionValue ::= literal

a:A	**Individual**: a **Annotations**: - *hasDefaultValue* defaultValue - *isArray* arraySizeValue (xsd:integer)
default = defaultValue arraySize = arraySizeValue	

5.4.1.2 Notes

There is only one predefined Component data property, *hasValue*, which is defined in the upper ontology. The property is functional, which means that it can only have one value per individual. It can be used to validate the model and/or take part in code generation. The restrictions regarding this property must be of type *some* or *only*. Some logical operators are not allowed in the external part of the restricting expression: Negation and the *OR* logical operator.

These expressions are not allowed because they contain illegal uses of *or* and *not* keywords:

- (not (hasValue some xsd:float))
- (hasValue some xsd:integer) or (hasValue some xsd:float)
- (hasValue some xsd:integer) or (not (hasValue some xsd:float))

The same expressions can be rewritten to abide by the rules:

- hasValue only not xsd:float
- hasValue some xsd:integer or xsd:float
- hasValue only not xsd:float

While dealing with data properties, both *some* and *only* will act as requirements, so that a specific individual can be assigned a value in the DSL. The following table shows the effects of data property class restrictions:

Purpose	Restriction	Reasoner Effect	DSL Effect
The individual must be assigned some value	hasValue some dataPrimary	The dataPrimary will be tested if the relation is instantiated	The assignment is compulsory. If an individual is instantiated and it is not assigned, an error is thrown.
The individual can be assigned a value	hasValue only dataPrimary	The dataPrimary will be tested if the relation is instantiated	The assignment is allowed.

User-defined data properties are only evaluated by the reasoner and cannot be assigned in the DSL. Although these have a reasoner effect and can be used to validate the model, the DSL won't evaluate the corresponding class restrictions and the data properties may only be used for code generation if they are constant or rule-defined.

5.4.1.3 Datatype transformations

Input RDF Datatype (default)	DSL	Output RDF Datatype
xsd:boolean	BOOL	xsd:boolean
xsd:float	FLOAT	xsd:double
xsd:double		
xsd:integer	INT	xsd:integer
other	STRING	xsd:string

5.4.2 Examples

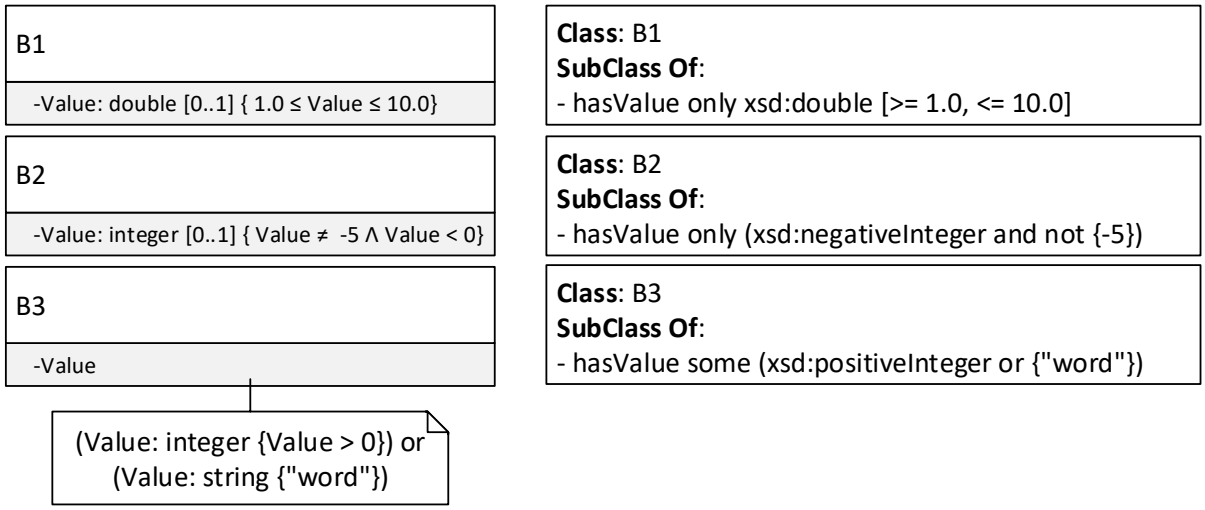
5.4.2.1 Example I

This example shows how to assert simple data property restrictions.

The class B1 is composed of individuals which may be connected to a literal, hence the multiplicity [0..1]. That literal must be a value of type *xsd:double*, ranging from 1.0 to 10.0. The keyword "only" is used because the relation is not required.

The second class has a similar constraint except that it uses negation, which can be transformed to OWL 2 using the keyword "not". The constraint datatype is *xsd:integer* but, since the value must be negative, that can be represented in OWL 2 with the datatype *xsd:negativeInteger*.

The third class's individuals may be connected to a positive integer or a string equal to "word". The connection is not optional though. Since the datatype is not fixed, it must be represented as a constraint.

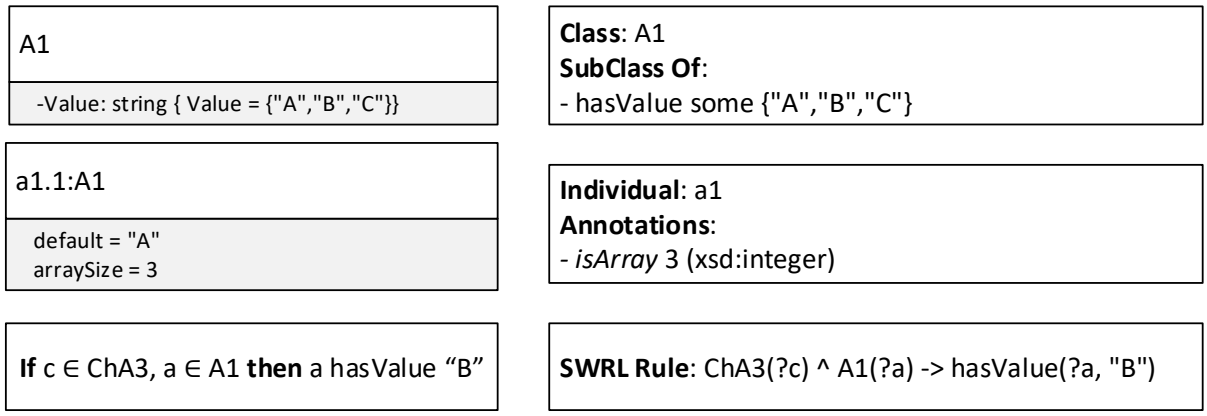


5.4.2.2 Example II

Arrays can also be used in conjunction with default values. If a certain individual represents an array and has a default value, each element will share that same value. This example also implements a rule that relates the use of a Characteristic with a value assignment.

A1 uses an enumeration constraint, containing string elements. Its only individual, a1.1 is an array. The SeMLI will search for the last number in the individual's name which will act as the initial index. The remaining elements will be named after the first, with incremental indexing.

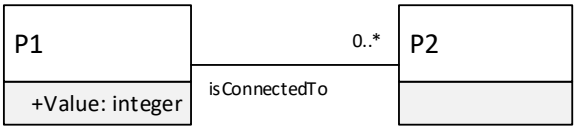
When ChA3 is used, all individuals of class A1 are assigned with a value, "B".



5.4.2.3 Example III

The purpose of this example is to integrate rules in the process of creating a data property, when the trigger is another relation. This establishes a condition to check whether the user should be given the option of manually creating the assignment.

Assume that a model is composed of two classes: P1 and P2. Every individual of P1 must be connected to a literal, of type *xsd:integer*, through the *hasValue* data property. If that individual is connected to an individual of P2, the literal's value is 30. Otherwise, the literal must be manually assigned, being 50 its default value.



Class: P1
SubClass Of:
- hasValue some xsd:integer
- isConnectedTo min 0 P2
Class: P2

If a ∈ P1, b ∈ P2, a isConnectedTo b
then a hasValue 30

P1(?a) ^ P2(?b) ^ isConnectedTo(?a, ?b) -> hasValue(?a, 30)

Individual: p1 (P1)
Annotations:
- hasDefaultValue: 50 [xsd:integer]
Individual: p2 (P2)

6 SWRL

Some inferences cannot be expressed in OWL. SWRL should be used in those cases.

6.1 Built-ins

6.1.1 Theory

There is another guide for this. It will be merged one day.

6.1.2 Examples

6.1.2.1 Example I

This example shows how to use a custom SWRL atom to interact with Reports.

If there are two individuals of the same subclass of S, then all individuals of S will have a warning report. The Java File can be obtained from GIT.

If s ∈ S, twoInst("esrg:basis#S") = true
then s hasWarning “There are two instances of the same subclass of S!”

twoInst("esrg:basis#S") ^ S(?s) -> hasWarning(?s, "There are two instances of the same subclass of S!")