**Semantic Engine Configuration Tutorial**

This documentation will walk a user through the creation of a new Semantic Engine configuration, including Transformation Instruction Set and Graph Specification files. To do this, we will start by describing a simple source dataset in its relational format, and its subsequent conversion into concise RDF triples. This conversion occurs outside of the scope of the Semantic Engine, and the concise RDF triples will be input to one or more of the Semantic Engine’s Update processes.

Let’s assume that we have a relational database, which we will call “A”, which stores only the most minimal of patient information: a single column of person identifiers. **Table 1** represents Database A.

**Database: A**

**Table: person**

|  |
| --- |
| person\_id |
| 1 |
| 2 |
| 3 |

**Table 1**: Sample Database A holding patient identifiers in column “person\_id”

Our goal is to transform this relational data into a representation using our semantically rich model defined in the Graph Specification file. The Semantic Engine Language (“Acorn”) allows us to apply only the subset of our model which is relevant for a given data source.

**Step 1:** **Convert the relational data in Database A into concise RDF triples**

Some external software can be used to create concise RDF triples based on the original RDF data. Consider the triples shown in **Figure 1**, which represent the shallow semantics of the relational source “**Database A**”. A software tool such as Carnival or Karma are suitable for RDF-ifying relational data, as well as many others. This concise RDF dataset should be inserted into an RDF repository compatible with the RDF4J library before the Semantic Engine Update processes are executed.

inputSchema:homoSapiens\_1 rdf:type inputSchema:homoSapiens .

inputSchema:homoSapiens\_1 inputSchema:identifier 1 .

inputSchema:homoSapiens\_2 rdf:type inputSchema:homoSapiens .

inputSchema:homoSapiens\_2 inputSchema:identifier 2 .

inputSchema:homoSapiens\_3 rdf:type inputSchema:homoSapiens .

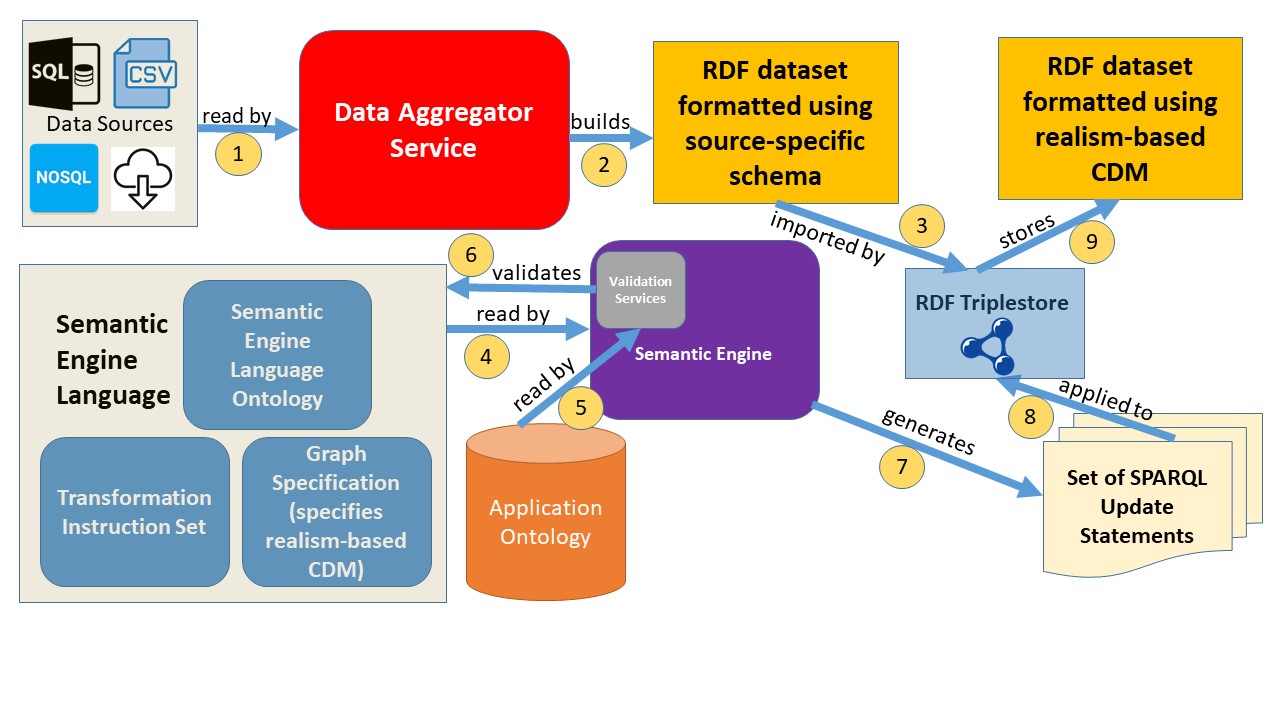
inputSchema:homoSapiens\_3 inputSchema:identifier 3 .

**Figure 1:**Source-specific triples mapped from Database A as input to Semantic Engine. “rdf:type” is the instantiation relation, which asserts that the subject (on the left) is an instance of the object (on the right).

At this point, the Semantic Engine has access to the data as concise triples. However, we will need to define mappings so that the software can understand data from this input source. That’s where the Semantic Engine Language comes into play.

**Step 2: Interpret concise RDF triples using Semantic Engine.**

The Semantic Engine Language configuration for a data source involves a collection of four files: an *application ontology*, a *Graph Specification*, a *Transformation Instruction Set*, and the *Semantic Engine Language ontology*. The application ontology, Graph Specification, and Semantic Engine Language ontology files are data source-agnostic, while the Transformation Instruction Set file is data source-specific. **Figure 2** shows a high-level overview of the components of the Semantic Engine and how they interact with various external components and data. In the following subsections, we will look at these components in detail in the context of defining the transformation for Database A.



**Figure 2***:* High-level overview of the components of the Semantic Engine and their interactions with various data sources to build a pipeline

**Components of a Semantic Engine Configuration**

1. **Application Ontology.** An application ontology is an ontology designed for a specific use, but which may import classes from more general ontologies. It may also provide additional native classes and predicates covering the cases in which no term exists in other ontologies to describe a relevant concept. The Semantic Engine application ontology file defines all of the classes and predicates which may be used in the Graph Specification. The application ontology is the main point of reference for validating any implementation of the target model. The first step in creating additions to the Graph Specification would be deciding which terms are relevant to add to the application ontology. See the [TURBO ontology](https://raw.githubusercontent.com/PennTURBO/Turbo-Ontology/master/ontologies/turbo_merged.owl) as an example.
2. **Graph Specification.** The Graph Specification file is where the target model is fully specified. It references classes and predicates from the application ontology. Unlike the application ontology, it defines all the ways that instances of particular classes may be connected with instances of other classes, classes themselves, or literal values. By making explicit the relationships by which two entities may be connected, it provides a representation of the target model independent of any data source or set of transformation instructions. Another way to understand the role of this component is that it provides a set of constraints on the application ontology.

The Graph Specification is composed using the syntax of the Semantic Engine Language. The Semantic Engine Language represents relationships as instances of RDF objects called Connection Recipes. Each Connection Recipe is specified with a Uniform Resource Identifier (URI). **Figure 3** is a subset of a possible Graph Specification file, showing only the Connection Recipes that relate to patient identifier data. The classes come from the PennTURBO Ontology[3](https://paperpile.com/c/1v2FvH/yhYc)3, the application ontology our group has developed. Classes in this figure are annotated with comments including the human-readable class label.

GraphSpecification.ttl

@prefix : <https://github.com/PennTURBO/Drivetrain/> .

@prefix obo: <http://purl.obolibrary.org/obo/> .

@prefix turbo: <http://transformunify.org/ontologies/> .

:homoSapiensIdentifierDenotesHomoSapiens a :InstanceToInstanceRecipe ;

:subject obo:IAO\_0000578 ; # centrally-registered identifier

:predicate obo:IAO\_0000219 ; # denotes

:object obo:NCBITaxon\_9606 ; # homo sapiens

:cardinality :1-1 ;

:mustExecuteIf :objectExists ;

.

:symbolPartOfCrid a :InstanceToInstanceRecipe ;

:subject obo:IAO\_0000028 ; # symbol

:predicate obo:BFO\_0000050 ; # part of

:object obo:IAO\_0000578 ; # centrally-registered identifier

:cardinality :1-1 ;

:mustExecuteIf :objectExists ;

.

:symbolHasRepresentation a :InstanceToLiteralRecipe ;

:subject obo:IAO\_0000028 ; # symbol

:predicate turbo:TURBO\_0010094 ; # has representation

:object :symbolLiteralValue ;

:cardinality :1-1 ;

.

:symbolLiteralValue a :IntegerLiteralResourceList .

**Figure 3:**A subset of the Graph Specification showing only Connection Recipes specifically relevant to the data in Database A

In order to specify that the existence in reality of a person is denoted by an identifier in our database, we instantiate an instance of a homo sapiens only when there is an instance of a patient identifier. Therefore, it would not be appropriate to have an instance of Homo sapiens without an associated instance of a patient identifier. This concept is represented by the following triple in the Graph Specification:

:homoSapiensIdentifierDenotesHomoSapiens :mustExecuteIf :objectExists .

This statement references the Connection Recipe **:homoSapiensIdentifierDenotesHomoSapiens**, and tells us that if the object of that recipe (an instance of class homo sapiens) exists, the subject of that recipe (an instance of class centrally-registered identifier, or CRID) must also exist, and be attached to the object by the predicate of that recipe, “denotes”. The ability to define rules for instantiating instances of classes based on the presence of specific incoming data fields was a key requirement for implementing the target model. If at any point an instance of class Homo sapiens existed without an associated instance of a CRID, it would be reported by the validation service, which will be discussed further along in this section.

Connection Recipes can represent general patterns that can be re-used within the model. For example, recipes **:symbolPartOfCrid** and **:symbolHasRepresentation** are not specific to CRIDs or symbols associated with people and can be used to help instantiate any type of CRID. Since the same Recipe can be referenced multiple times, it would be trivial to implement, say, CRIDs and symbols denoting a prescription or a diagnosis by referring to the same **:symbolPartOfCrid** and **:symbolHasRepresentation** recipes. This re-usability encourages consistency in pattern development and helps reduce errors introduced by duplicating code.

1. **Transformation Instruction Set.** While the Graph Specification provides a set of constraints on the application ontology but has no knowledge of incoming data sources, the Transformation Instruction Set maps concise source-specific triples (like in **Figure 1**) to a subset of the target model as defined in the Graph Specification.

A new Transformation Instruction Set must be created for each new data source that is incorporated. Like the Graph Specification, Transformation Instruction Set files are composed with the syntax of the Semantic Engine Language.

The process of making a new Transformation Instruction Set based on source-specific RDF triples from a previously unincorporated data source first involves surveying the Graph Specification, in order to get a sense of how various relationships in the target model fit together. Once a programmer has determined the subset of the Graph Specification that is relevant to their data source, they can begin writing new Connection Recipes in their Transformation Instruction Set that represent the source-specific RDF, and map them to pre-existing Recipes in the Graph Specification.

**Figure 4** shows what a Transformation Instruction Set for Database A could look like. It introduces a new type of RDF object, an Update Specification, which defines when and how a Connection Recipe will be executed.

A valid Transformation Instruction Set must contain at least one instance of an Update Specification that defines input and output Connection Recipes. For incoming data sets that contain many fields, we build several Update Specifications, with each one transforming a specific subsection of the data. Since a single Connection Recipe could be the output of one Update Specification and the input to another, order matters. Therefore, it is possible to chain Update Specifications together within the Transformation Instruction Set.

DatabaseA\_TransformationInstructionSet.ttl

@prefix : <https://github.com/PennTURBO/Drivetrain/> .

@prefix obo: <http://purl.obolibrary.org/obo/> .

@prefix turbo: <http://transformunify.org/ontologies/> .

@prefix inputSchema: <http://databaseA.org/RdfSchema/> .

:expandPatientIdentifiers a :UpdateSpecification ;

:inputNamedGraph :inputTriples ;

:outputNamedGraph :expandedTriples ;

:hasOutput :homoSapiensIdentifierDenotesHomoSapiens ;

:hasOutput :symbolPartOfCrid ;

:hasOutput :symbolHasRepresentation ;

:hasRequiredInput :databaseAPersonToIdentifier ;

.

:databaseAPersonToIdentifier a :InstanceToLiteralRecipe ;

:subject inputSchema:homoSapiens ;

:predicate inputSchema:identifier ;

:object :symbolLiteralValue ;

:cardinality :1-1 ;

.

**Figure 4:** A Transformation Instruction Set for Database A

1. **Semantic Engine Language Ontology.** The Semantic Engine Language ontology file defines all of the classes and predicates available in the Semantic Engine Language. Following OBO Foundry guidelines, all classes defined in the TURBO application ontology are represented by a partially numeric URI. Our goal was to make using the Semantic Engine Language an intuitive experience for programmers and ontologists defining their own transformations, so the URIs of its classes and predicates are text-based and suggest the term’s function. Unless a user is developing new features for the Semantic Engine Language, the provided [Semantic Engine Language Ontology file](https://github.com/PennTURBO/semantic-engine/blob/master/drivetrain/ontologies/acornOntology.ttl) should be sufficient.

**3.5 Running the Semantic Engine**

**Step 3: Run the Semantic Engine using aforementioned configuration files.** The Semantic Engine is written in Scala and can be run either as a precompiled .jar file or from the console of SBT, the build tool for Scala. Upon running the application, the Semantic Engine Language configuration files will be loaded into an RDF repository. Validation checks specified in the application code will ensure that the input data and Semantic Engine Language components are valid and consistent with each other. If these checks pass, the generated SPARQL updates will be executed in sequence against the input data using the RDF4J library[3](https://paperpile.com/c/1v2FvH/92Zi)5. The application will generate SPARQL update queries based on the contents of the Transformation Instruction Set.

**Figure 5** shows the SPARQL update that will be generated by running the Semantic Engine using the Graph Specification defined in **Figure 3** and the Transformation Instruction Set defined in **Figure 4**. SPARQL updates generated by the Semantic Engine have 3 distinct sections: INSERT, WHERE, and BIND.

1. The WHERE clause matches and binds variables against the source-specific RDF triples seen in **Figure 1**, using the pattern defined in **Figure 4**.
2. The BIND clause handles the creation of URIs for newly created instances. We create these URIs with the output of an SHA256 function which takes three parameters:

* a string representation of the type of the instance to be created, to ensure that new elements of different types will not have the same URI
* a string key specified by the user which acts as a “salt”, ensuring that the created URIs will be unique to this instantiation, unless the same key is used again
* a string representation of the URI identifying the instance with which the new instance should share a cardinality relationship (i.e. for each input of **inputSchema:homoSapiens**, we should create one new URI identifying an instance of type **obo:NCBITaxon\_9606**, the OBO Foundry term for homo sapiens)

1. The INSERT clause assigns the bound variables to the pattern derived from the relevant subset of the Graph Specification in **Figure 3**.

**Figure 5:** The SPARQL update generated by the Semantic Engine’s processing of Update Specification :expandPatientIdentifiersfrom Figure 5

prefix obo: <http://purl.obolibrary.org/obo/>

prefix turbo: <http://transformunify.org/ontologies/>

prefix inputSchema: <http://databaseA.org/RdfSchema/>

prefix : <http://UserDefinedPrefix.org/>

# The INSERT clause represents the target pattern

INSERT {

GRAPH :expandedTriples {

?IAO\_0000578 obo:IAO\_0000219 ?NCBITaxon\_9606 . # key denotes homo sapiens

?IAO\_0000578 rdf:type obo:IAO\_0000578 . # key type declaration

?NCBITaxon\_9606 rdf:type obo:NCBITaxon\_9606 . # homo sapiens type declaration

?IAO\_0000028 obo:BFO\_0000050 ?IAO\_0000578 . # symbol part of key

?IAO\_0000028 rdf:type obo:IAO\_0000028 . # symbol type declaration

?IAO\_0000028 turbo:TURBO\_0010094 ?symbolLiteralValue . # symbol has representation symbol value

}

}

# The WHERE clause represents the input pattern

WHERE {

GRAPH :inputTriples {

?homoSapiens inputSchema:identifier ?symbolLiteralValue . # input homo sapiens has identifier

?homoSapiens rdf:type inputSchema:homoSapiens . # input homo sapiens type declaration

}

# The BIND clause handles the creation of new URI’s for instances in the target pattern, based on instances in the

# input pattern

BIND(uri(concat("http://UserDefinedPrefix.org/",SHA256(CONCAT("?IAO\_0000578","8af9e702e52844f99005716ed520c40e", str(?homoSapiens))))) AS ?IAO\_0000578)

BIND(uri(concat("http://UserDefinedPrefix.org/",SHA256(CONCAT("?NCBITaxon\_9606","8af9e702e52844f99005716ed520c40e", str(?homoSapiens))))) AS ?NCBITaxon\_9606)

BIND(uri(concat("http://UserDefinedPrefix.org/",SHA256(CONCAT("?IAO\_0000028","8af9e702e52844f99005716ed520c40e", str(?homoSapiens))))) AS ?IAO\_0000028)

}

**3.6 Diving Further into Connection Recipes**

**3.6.1 Connection Recipe Overview**

A Connection Recipe must include a subject, predicate, object, and cardinality declaration, and optionally specifies the conditions under which it must be executed. The predicate referenced by a Connection Recipe must be present in the application ontology. The subject may be present in the application ontology or it may reference a URI from an external vocabulary. The object can be present in the application ontology, reference an external URI, or represent a literal value.

Below are the six types of Connection Recipes, two of which are highlighted for their use in our minimal example.

1. *Instance-to-instance recipes* define a triple where the subject is an instance of a class and the object is also an instance of a class (see **:homoSapiensIdentifierDenotesHomoSapiens** in **Figure 3**)
2. *Instance-to-term recipes* define a triple where the subject is an instance of a class and the object is a class itself. This is used in the case that an instance has some direct relationship with a term in the associated application ontology, or to a term from an outside terminology.
3. *Term-to-instance recipes* are similar to Instance-to-term recipes, but the subject is the class and the object is the instance.
4. *Instance-to-literal recipes* define a triple where the subject is an instance of a class and the object is a literal value. The type of literal value may be further specified as instances of one of five literal types: string, Boolean, integer, double, or date. (see **:symbolHasRepresentation** in **Figure 3**)
5. *Term-to-literal recipes* define a triple where the subject is a class and the object is a literal value. We have used this type of recipe thus far mainly for retrieving classes from our application ontology that are linked to OMOP concept codes that are present in our input dataset.
6. *Term-to-term recipes* define a triple where the subject and the object are both classes. We have used this type of recipe thus far mainly for specifying subclass relationships.

The cardinality fieldof a Connection Recipe, specified by predicate **:cardinality,** defines how many instances of the subject may connect to an instance of the object via the stated predicate, and vice versa. The conditions that can be specified are **:1-1**, **:many-1**, and **1-many**.

The implementation requirement field, specified by predicate **:mustExecutetIf**, defines the conditions under which a given Connection Recipe must be implemented by a Transformation Instruction Set. There are several conditions that can be specified, including **:objectExists** which we saw in **Figure 3**, as well as **:subjectExists** and **:subjectOrObjectExists**.

**3.6.2 Representation in SPARQL**

The SPARQL representation of a Connection Recipe depends on the type of entities that the recipe represents. Subjects or objects of recipes representing instances of classes will be automatically converted into a unique SPARQL variable based on the URI of the class. Types of instances are automatically assigned. **Table 2** demonstrates how the predicate used to link an Update Specification to a Connection Recipe affects where that Recipe will be placed in the generated SPARQL clause. **Table 3** contains examples of each of the six types of Connection Recipes as well as their subsequent compilation into SPARQL string snippets by the Semantic Engine.

|  |  |
| --- | --- |
| **Relationship between Update Specification and Connection Recipe** | **SPARQL Section** |
| :hasRequiredInput | WHERE |
| :hasOptionalInput | WHERE (IN OPTIONAL BLOCK) |
| :hasOutput | INSERT |
| :removedBy | DELETE |

**Table 2**: The predicate that links Connection Recipes to an Update Specification determines where the Connection Recipe will be placed in the generated SPARQL update statement

|  |  |  |
| --- | --- | --- |
| **Connection Recipe Type** | **Example Representation in Semantic Engine Language Configuration** | **Example Representation in SPARQL** |
| *Instance-*  *to-*  *Instance* | :homoSapiensHasQualityBioSex a :InstanceToInstanceConnectionRecipe ;  :subject :homoSapiens ;  :predicate :hasQuality ;  :object :biologicalSex ;  :cardinality :1-1 ;  . | ?homoSapiens :hasQuality ?biologicalSex .  ?homoSapiens rdf:type :homoSapiens .  ?biologicalSex rdf:type :biologicalSex . |
| *Instance-to-Term* | :keyHasPartRegistryDenoter  a :InstanceToTermRecipe ;  :subject :key ;  :predicate :hasPart ;  :object :someRegistry ;  :cardinality :1-1 ;  . | ?key :hasPart :someRegistry .  ?key rdf:type :key . |
| *Term-to-*  *Instance* | :registryDenoterPartOfKey  a :TermToInstanceRecipe ;  :subject :someRegistry ;  :predicate :partOf ;  :object :key ;  :cardinality :1-1 ;  . | :someRegistry :partOf ?key .  ?key rdf:type :key . |
| *Instance-to-Literal* | :cridSymbolHasRepresentation  a :InstanceToLiteralRecipe ;  :subject :cridSymbol ;  :predicate :hasRepresentation ;  :object :tumor\_LiteralValue ;  :cardinality :1-1 ;  . | ?cridSymbol :hasRepresentation ?tumor\_LiteralValue .  ?cridSymbol rdf:type :cridSymbol . |
| *Term-to-*  *Literal* | :datumHasOmopConceptId  a :TermToLiteralRecipe ;  :subject :datum;  :predicate :hasOmopConceptId ;  :object :omop\_concept\_id ;  :cardinality :1-1 ;  . | :datum :hasOmopConceptId ?omop\_concept\_id . |
| *Term-to-*  *Term* | :genderIdentityDatumSubclassOfDatum  a :TermToTermRecipe ;  :subject :genderIdentityDatum ;  :predicate rdfs:subClassOf ;  :object :datum ;  :cardinality :1-1 ;  . | :genderIdentityDatum rdfs:subClassOf :datum . |

**Table 3:** Connection Recipes and their translation to SPARQL

**Validation Services**

The Semantic Engine software includes several layers of validation checks, broken up into modules called Validation Services. The overall objectives of the Validation Services are to ensure that all terms in the Semantic Engine Language are used as expected, ensure that created graph patterns are consistent both in the formal logical sense and with regards to the definition of the target model, and ensure that the incoming source-specific RDF data matches the patterns specified by the Transformation Instruction Set. We break down the existing services below, in the order that the Semantic Engine executes them.

**3.7.1 Graph Specification Validation Service**

The application ontology file defines all of the classes and predicates which may be used in the target model, and the Graph Specification references a subsection of these terms. The Graph Specification Validation Service enforces that:

1. All subjects, predicates, and objects referenced by Connection Recipes in the Graph Specification are defined in the application ontology.
2. If domains and/or ranges for predicates are defined in the application ontology, all subjects referenced by Connection Recipes are within the domain, and all objects referenced by Connection Recipes are within the range.
3. All terms in the Graph Specification are defined either in the application ontology or in the Semantic Engine Language ontology.

**3.7.2 Transformation Instruction Set Validation Service**

The Transformation Instruction Set must be consistent with the Graph Specification’s stated requirements for when a given Connection Recipe must be executed, and all of its terms must be defined. The Transformation Instruction Set Validation Service enforces that:

1. If a Connection Recipe referenced by an Update Specification as an output refers to a subject or object which is referenced by another Connection Recipe in the Graph Specification, and that Connection Recipe must execute if the subject or object exists, then that Connection Recipe must be the output of at least one Update Specification in the Transformation Instruction Set.

For example, given the following, an error would be thrown:

- An Update Specification has as an output a Connection Recipe with object **:patientRole**

- The Graph Specification contains another Connection Recipe called **:encounterRealizesPatientRole** with subject **:encounter** and object **:patientRole**, which must execute if **:objectExists**

- No Update Specification has as an output **:encounterRealizesPatientRole**

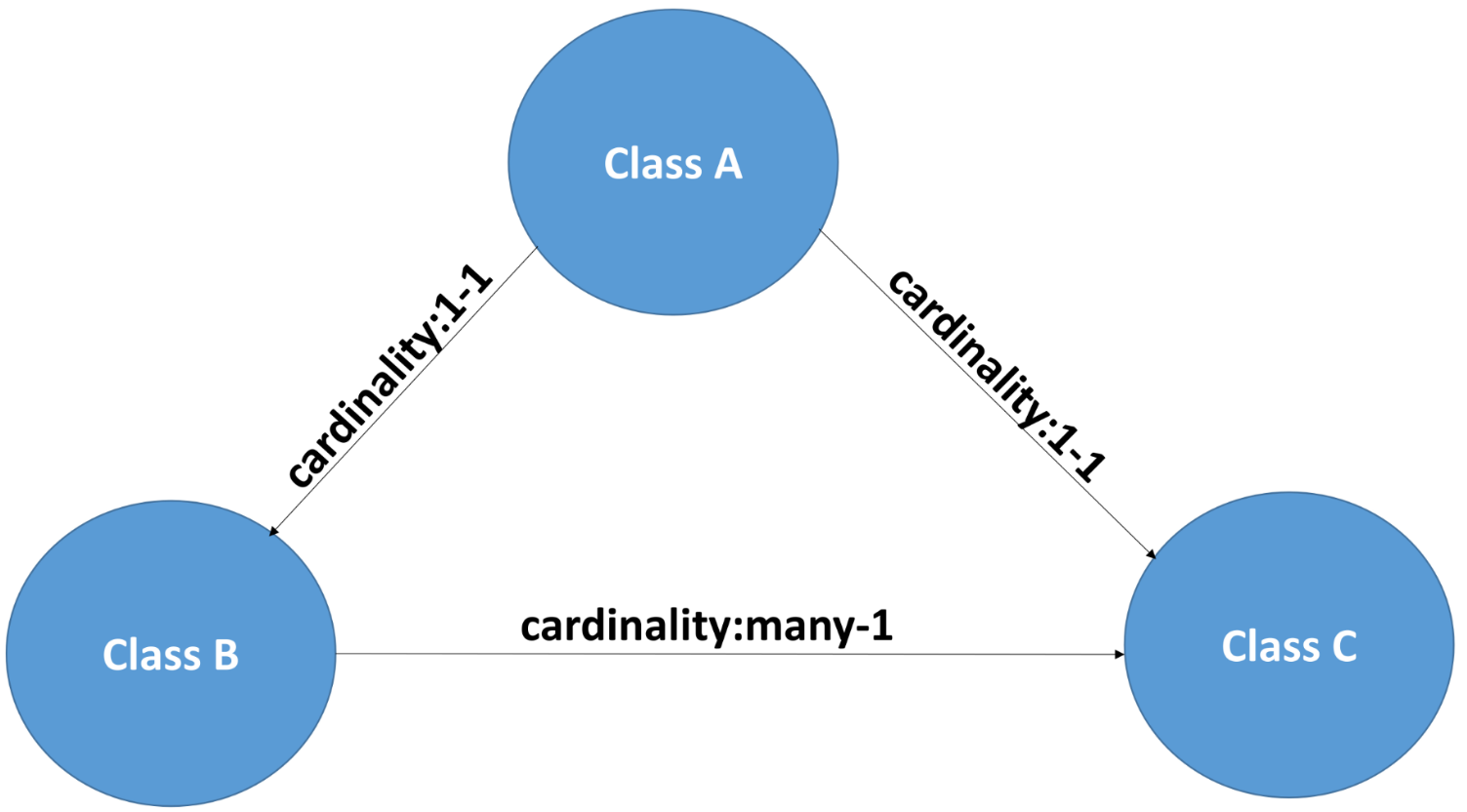
- This is an error because an instance of **:patientRole** is being created and its required relationship with an instance of **:encounter** is not being created.

1. All terms in the Transformation Instruction Set must have a type.
2. All terms in the Transformation Instruction Set that are not subjects, predicates, or objects referenced by a Connection Recipe must be defined in the Semantic Engine Language ontology.

**3.7.3 Conflict Detection Validation Service**

The Conflict Detection Validation Service operates over both the Transformation Instruction Set and Graph Specification. Its goal is to find single Connection Recipes or a set of Connection Recipes that would introduce an inconsistency, and alert the user before the transformation begins. The Conflict Detection Service enforces that:

1. A Connection Recipe’s stated type corresponds with the subject and/or object that it references. For example, if a recipe is an Instance-to-Instance recipe, and the recipe references an object that is a literal value, this would be flagged.
2. Multiple Connection Recipes do not define cardinalities that are mutually inconsistent. A simple example of an inconsistent cardinality loop is shown in **Figure 6**.

****

:ClassAtoClassB a :InstanceToInstanceRecipe ;

:subject :classA ;

:predicate :relatesTo ;

:object :classB ;

:cardinality :1-1 ;

.

:ClassBtoClassC a :InstanceToInstanceRecipe ;

:subject :classB ;

:predicate :relatesTo ;

:object :classC ;

:cardinality :many-1 ;

.

:ClassAtoClassC a :InstanceToInstanceRecipe ;

:subject :classA ;

:predicate :relatesTo ;

:object :classC ;

:cardinality :1-1 ;

.

**Figure 6:** Graphical display and triples of an example of inconsistent cardinality relationships between Connection Recipes in a Transformation Instruction Set

**3.7.4 Input Data Validation Service**

The Input Data Validation Service is the only Validation Service that operates over the source data that a user has loaded into a repository. It is important that this service run last, because the assumption is that by this point the Semantic Engine Language configuration files have been validated. The input Data Validation Service enforces that:

1. If a Connection Recipe referenced by an Update Specification as a required input must execute on some condition, that condition must be present in the instance data.

For example, given the following, an error would be thrown:

- An Update Specification requires as an input Connection Recipe with subject **inputData:medication and** object **inputData:medOrderName**, which must execute if **:subjectExists**

- An instance of class **inputData:medication** without an associated instance of **inputData:medOrderName** in the input data exists