JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche
Fakultät

Incremental Consistency Checking with SPARQL

BACHELORARBEIT
(Projektpraktikum)

zur Erlangung des akademischen Grades

Bachelor of Science

im Bachelorstudium

INFORMATIK

Eingereicht von:
Patrick Themessl-Huber, 0855303

Angefertigt am:
Institute for Systems Engineering and Automation

Beurteilung:
Univ.-Prof. Dr. Alexander Egyed, M. Sc.

Linz, Mai 2013

Incremental Consistency Checking with SPARQL

Patrick Themessl-Huber

May 18, 2013

Incremental consistency checking is a method of increasing the performance of the evaluation of UML consistency rules at design-time. It works by determining and then keeping a list of model elements in memory for every consistency rule, the so called *change impact scope*, and re-evaluating the rule only when the user performs a modification of one of those model elements in the list. This method has been shown to work if the consistency rules are evaluated by an interpreter that is under our control so that we can observe their evaluation. In this paper we assume the software model to be stored in RDF and the consistency rules to be expressed in SPARQL. We develop a method to determine the change impact scope by transforming the queries in a way that the scope is contained in the result set of the modified query. We show that it is possible to port incremental consistency checking to systems where the evaluation of consistency rules is *black-boxed*, which renders the previous approach of tracing the execution of an (OCL) interpreter inapplicable. Since (legacy) consistency rules are typically expressed in OCL, in the second part of this paper I am introducing my approach to determine a set of model elements that are required to calculate the result of an OCL rule executed against a software model stored in RDF in order to allow a pre-fetching of those elements in situations where the execution of single SPARQL queries comes with a constant, high time penalty.

Inkrementelles consistency checking ist eine Methode, um die Auswertung von UML Konsistenzregeln zur Designzeit zu beschleunigen. Für jede Instanz einer Konsistenzregel wird eine Liste von Modellelementen bestimmt und dann im Speicher gehalten, der s. g. change impact scope, wobei gilt, dass bei jeder Änderung eines der enthaltenen Modellelemente die zugehörige Konsitenzregel-Instanz neu ausgewertet werden muss. Es wurde gezeigt, dass diese Methode funktioniert, wann immer der Interpreter der Konsistenzregeln unter der eigenen Kontrolle ist und somit die Auswertung der Regeln beobachtet werden kann. In dieser Arbeit nehmen wir an, dass das Software-Modell in RDF gespeichert wird und die Konsistenzregeln in SPARQL ausgedrückt werden und wir entwickeln eine Methode, um den change impact scope dieser Regeln zu bestimmen, indem die Regeln so transformiert werden, dass der change impact scope im Ergebnis der Auswertung enthalten ist. Wir zeigen somit, dass es möglich ist, inkrementelles consistency checking in Szenarien einzusetzen, wo die Auswertung der Konsistenzregeln in einer *black box* abläuft. Da die meisten existierenden Konsistenzregeln in OCL ausgedrück sind, beeschreiben wir im zweiten Teil dieser Arbeit eine Methode, um zu einer gegebenen Konsistenzregel in OCL eine Übermenge der zur Auswertung erforderlichen Modellelemente mittels SPARQL aus einem RDF triple store im Voraus abzufragen, um die Auswertung in Situationen zu beschleunigen, wo das Ausführen einer einzelnen SPARQL-Abfrage mit einem hohen, konstanten Zeitoverhead verbunden ist.

Contents

1. Problem

Most software modelling utilities use proprietary or specialized technologies for the task of persisting software models. For various reasons, efforts have been made to move the persistence layer towards open standards, such as the *Resource Description Framework* (RDF). The main reasons for doing so are to be more generic and enable the use of less specialized algorithms that can easily be adapted to work on different model types, the use of different constraint languages, and of course also the general advantages of RDF (see http://www.w3.org/RDF/advantages.html).

Using conventional technologies for the evaluation of consistency rules against software models stored in RDF would cause many small queries to be executed against the RDF triple store, where each one of them comes with a certain time overhead, cumulatively slowing down the consistency checking process. For that reason it is desirable to evaluate a consistency rule with as few as possible queries to the RDF triple store.

This thesis covers two aspects of this transition to RDF:

In the first part, we assume a consistency rule to be expressed in SPARQL and then develop a method to apply *incremental* consistency checking of that rule to a software model stored in RDF, following the principles as described by Alexander Egyed [Egy11], while avoiding a large number of queries for the reason mentioned in the previous paragraph. Generally speaking, this approach constitutes a solution to situations where the evaluation of a consistency rule happens in a black-boxed system, like an RDF triple store, where incremental consistency checking done the usual way does not work because the observation of the inner workings of the evaluating system is not possible.

In the second part, we assume a consistency rule to be expressed in OCL, one of the currently more typical languages for this purpose, and ask ourselves how we can evaluate this rule against a software model stored in RDF with as few SPARQL queries as possible.

2. Introduction

Industrial software models may become very large in size and can contain thousands of model elements. For obvious reasons it can become difficult to avoid design-time inconsistencies in such models without the assistance of an automatic consistency checking mechanism. Consistency rules expressed in languages like OCL are the foundation for such mechanisms, but their execution can be very resource consuming, so an automatic

re-evaluation of all the consistency rules after *every* change in the software model would cause the responsiveness of modelling utilities to decrease dramatically, or even make them completely unusable depending on the size of the model and the number of consistency rules. Since the execution time of evaluating a large number of rules against a large software model can easily reach several hours, [Egy11] even a user-triggered evaluation may not be sufficient in terms of usability, because failing to regularly perform such a lengthy (re-)evaluation (which additionally causes an interruption of the designer's workflow) allows for many inconsistencies to happen between two evaluations.

The purpose of incremental consistency checking is to improve the performance of an evaluation of the consistency rules so that this task can be done in the background without interrupting the workflow and providing near-instant feedback if a certain consistency rule is violated during the design process. Several methods of achieving this exist, some requiring the designer to specially annotate the consistency rules, while others perform the task completely automatic. One of the latter ones [Egy11] do so by observing the evaluation of every single consistency rule and remembering which model elements are accessed in order to infer the resulting truth value of the rule. For every rule, this list of model elements is kept in memory and *only* when one of them undergoes a modification the related rule will be re-evaluated. The author states that this technique is feasible for many constraint languages; one requirement being that the evaluation of a certain constraint must be observable in some way. [Egy11] In this paper we are showing that the approach also functions if the evaluation of the rules is *not* observable, by taking advantage of the knowledge about the semantics of the constraint language and modifying the constraints such that they themselves contain the list of accessed model elements in their result which is then returned. This allows for the evaluation to take place in a black box, like it is usually the case in declarative query languages combined with database systems.

3. Automatic Incremental Consistency Checking

In this section I am giving a short introduction to what automatic incremental consistency checking is and demonstrate it by an example.

OCL consistency rules typically consist of two parts: A *context element* and an OCL expression. The context element can be either a class or an interface, or any UML element of a meta model. It does, however, not represent a particular instance of that element, but covers all its instances. For example, defining a consistency rule with

context element *Class* means that the rule is evaluated against every Class in the model and the evaluations of the rule against the different Classes happen independent from each other. In the second part of the consistency rule, which is the OCL expression, a keyword called *self* is available which describes the current instance of the context element, or in our particular example a certain class in the model. To evaluate one consistency rule, the first step is to determine all the instances of the context element in the model. We then generate the cross-product of our consistency rule with all those context element instances and call the resulting tuples consistency rule instances: consistency rules in combination with a model element. We assume that every OCL expression of a consistency rule must evaluate to either *true* or *false* and furthermore assume that a consistency rule is satisfied iff. all its consistency rule instances evaluate to *true*.

In automatic incremental consistency checking, we start by generating this list of consistency rule instances. This is necessary because the evaluation of two distinct consistency rule instances may very likely require information about distinct sets of model elements. Initially we evaluate each one of them and while doing so, we store all the model elements that were accessed during the evaluation and that therefore influence the result of the evaluation. We call this list the *change impact scope* of the consistency rule and we maintain all the change impact scopes in memory. During the following software design process, whenever the user modifies one of the elements that are in one of the change impact scopes, we know that we need to re-evaluate the corresponding consistency rule instance as soon as possible and also create a new change impact scope while we do so because the modifications in the model may have caused the change impact scope to change.

There are two important requirements to a change impact scope. The first one obviously is that it has to be complete, because only a complete change impact scope per definition guarantees that a consistency rule is re-evaluated after every possible modification in the software model that causes the rule's truth value to change. The second requirement is that the change impact scope has to be as small as possible, ideally minimal (which means the change impact scope contains *only* model elements of which a modification causes the result of the corresponding consistency rule to change). [Egy11]

Let us consider the example described by the class diagram in figure 1 and the sequence diagram in figure 2.

In the class diagram we can see that the example consists of two classes: *Switch* and *Light*, with the latter one containing the operations *turn-on* and *deactivate*. The

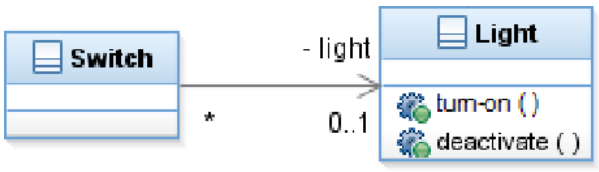

Figure 1: "Light Switch" Class Diagram

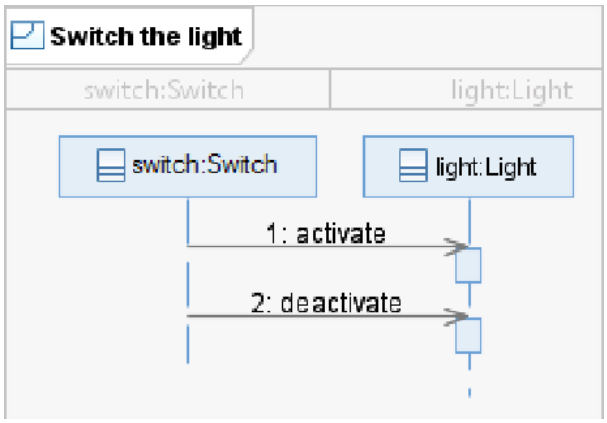

Figure 2: "Light Switch" Sequence Diagram

sequence diagram states that in the scenario *Switch the light*, the *Switch* instance first sends a message called *activate* and then a message called *deactivate* to the receiver of type *Light*.

Let us now declare a consistency rule which states that whenever a message is sent in the model, the receiver of that message must define an operation with a name equal to that of the message. We can immediately observe that this rule is not satisfied in our example, since the *Light* class does not define an operation called activate, while the first message sent in the sequence diagram would require it. Such a consistency rule could look like the one demonstrated in listing 1.

```
1 Context: Message
2 Description: Message action must be defined as an
3              operation in reciever's class
4 OCL: self.receiveEvent.oclAsType(InteractionFragment).
5      covered->forAll(represents.type.oclAsType(Class).
6      ownedOperation->exists(name=self.name))
```

Listing 1: OCL Consistency Rule Example

In incremental consistency checking, we start by determining the context instances.

In our example, those context instances would be the two messages occurring in the sequence diagram: *activate* and *deactivate*. In a second step, we initially need to evaluate the rule against both context instances. While we do so, we need to remember the model elements being accessed, which constitute our change impact scope. For the first message, *activate*, those are: the message *activate*, its receive event, all the lifelines in the diagram that are covered by that receive event, their types which represent the receiving classes and all the operations owned by those classes. To demonstrate that the change impact scope may very well vary from context instance to context instance, consider that only due to the fact that the class *Light* does *not* contain an operation called *activate*, for the evaluation of the consistency rule we need to iterate over *all* the owned methods until we find that *activate* is not among them, so the change impact scope contains all the operations of that class. Assuming we were looking for an operation called *turn-on*, the OCL interpreter might very well use short-circuit evaluation and stop the iteration of the operations immediately after *turn-on* was found, since this is enough knowledge to infer that the rule is satisfied. In that scenario, depending on the iteration order, the operation *deactivate* might be excluded from the change impact scope.

We have now gathered a change impact scope for every consistency rule instance and whenever the user changes one of the model elements that occur in one of the change impact scopes, we simply re-evaluate the corresponding rule instance and gather its new change impact scope. In contrast, using a framework that does not implement incremental consistency checking requires us to re-evaluate *all* the consistency rule instances in the model after every small change somewhere in the model, which is by no means efficient.

4. Outline

In the first part of this paper I am describing how to modify a consistency rule expressed in SPARQL so that the result of the new query contains the change impact scope of the consistency rule. In the second part of the paper I describe a method that, given a consistency rule expressed in OCL, determines a SPARQL query which returns a superset of the model elements required to evaluate the OCL expression.

Part I.
RDF and the Change Impact Scope

5. Introduction

Industrial software design models sometimes consist of a very large quantity of model elements which makes it necessary for software modelling utilities such as the IBM Rational Software Modeller to incorporate technologies that assist the designer in the task of keeping the models in a consistent state. Languages exist which serve the purpose of expressing consistency rules that are automatically evaluated against a given software model. For obvious reasons instant feedback is desirable but considering the potentially huge amount of model elements, the straightforward approach of re-evaluating all the consistency rules after every single change somewhere in a software model is too inefficient, so engineers have invested the effort to create a framework that determines for each rule on which events it has to be re-evaluated.

One approach, as described by Alexander Egyed [Egy11], is *incremental consistency checking*: Initially, that is, when the designer first writes or modifies a consistency rule, we observe the evaluation of that rule and keep track of all the model elements that are accessed during this evaluation and which therefore potentially influence the rule's resulting truth value. For each rule, we then assemble those model elements in a list, called the *change impact scope*. Only if and when the designer modifies a field of one of the model elements that is in the change impact scope of a particular rule, we need to re-evaluate the rule. Egyed demonstrates that even though the change impact scope obtained by this approach is not necessarily minimal, we can observe a substantial speedup when applying it to large industrial software models. It is common practice to define a *context* for every consistency rule, that is a type of model element and acts as the perspective out of which we evaluate the rule. So for every rule we employ *a set of* change impact scopes, one for every *consistency rule instance* that consists of a rule and the actual model element of the type as described by the rule's context.

Observing the execution of a consistency rule in order to gain a particular change impact scope is not always an option, especially when we use a black boxed and interchangeable database to store the software model and a descriptive language to express the consistency rules. In this paper we are developing a method to gain a change impact scope, assuming that the software model is stored in an RDF triple store and the

consistency rules are expressed as SPARQL queries.

6. Introduction to SPARQL

7. SPARQL Consistency Rules

SPARQL is a declarative, graph-based query language that is executed directly against an RDF triple store, which means that unlike when following the approach of employing a customized interpreter of a language for expressing consistency rules, we can not place our logic to determine the change impact scope in the interpreter which is assumed to be out of our control. Instead, the only thing we can work with is the SPARQL query itself: we have to transform it in a way that the resulting query contains some extra information from that we can derive the change impact scope.

To understand how we store a software model as RDF triples, let us take a look at the example of a class diagram in figure 1 and a subset of the corresponding RDF representation in figure 3. *URIs* are represented as blue rectangles, *string literals* as green rectangles, *predicates* as directed red arrows, and instances of the Bag-*concept* as triangles. We can see that in RDF, classes are represented by stub nodes consisting only of an URI. All the additional information about those classes is also represented by URI stubs. The association is represented by a set of RDF triples at the top of the diagram: One URI for the association itself, two bag instances, one for each end of the association that both contain a property which resolves to the classes "Light" and "Switch".

Since consistency rules are executed against a certain context, which is a type of model element[1], one rule consists of several SPARQL queries: One query to fetch all the context instances - all RDF nodes that should act as the context of the rule - and then one query to evaluate the consistency rule against one of those context instances. We will call one of this rules executed against a context instance a *consistency rule instance*, in accordance with [Egy11]: **CRI**=<ConsistencyRule, RDFNode>, where RDFNode is a result of the context instance query.

We will now look at it in the context of a concrete example. Let us assume a very simple consistency rule: "Every class must have an operation named *deactivate*". For this example, we are interested in a different subset (figure 4) of the RDF graph of the class diagram in figure 1. The context of this consistency rule is *class*, so as a first

[1] Theoretically we can use any set of RDF nodes as the context of a consistency rule, but typically this will be a model element.

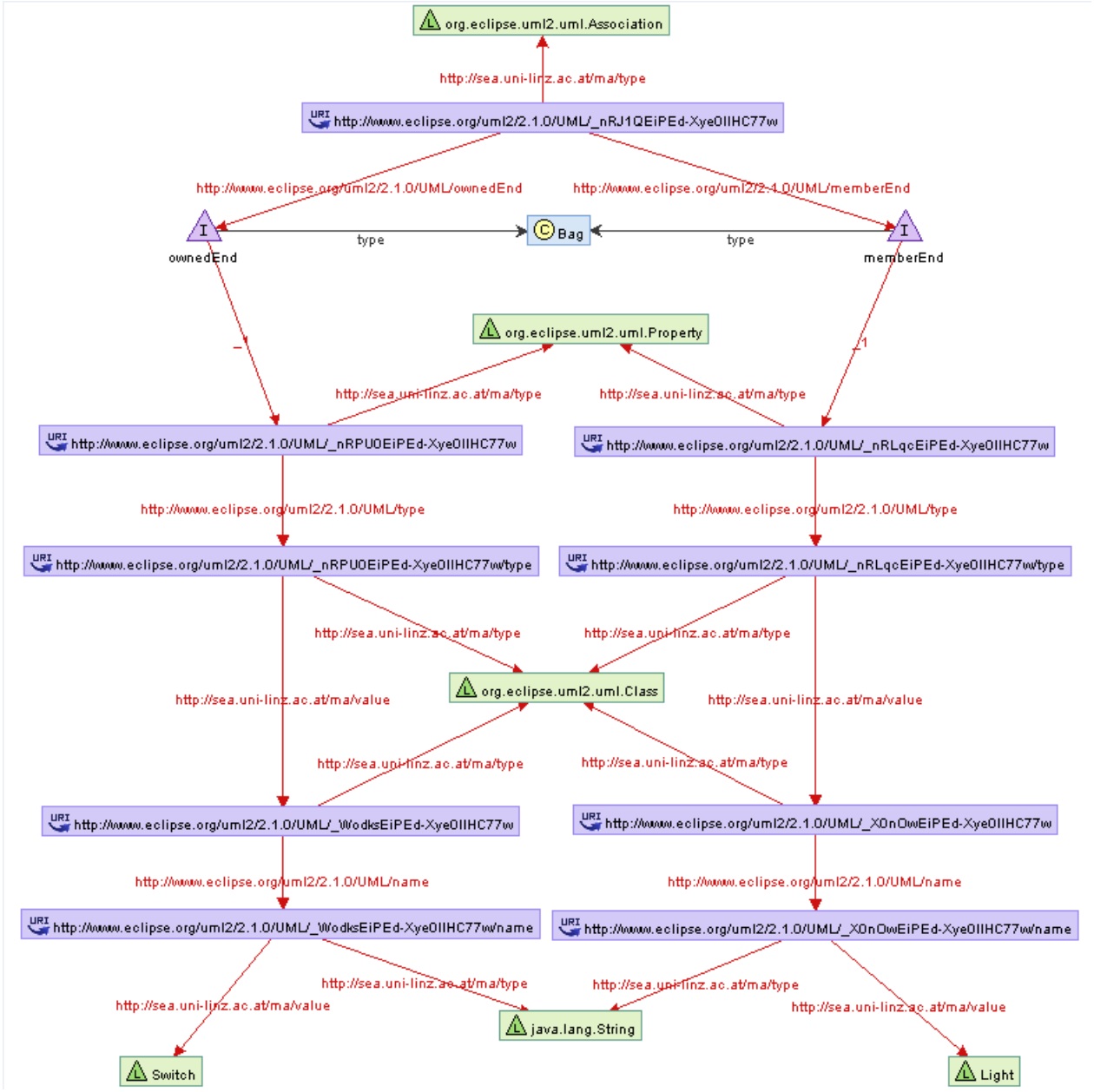

Figure 3: Subset of "Light Switch" RDF Representation

step we need to execute a SPARQL query that returns a list of all classes represented by their URI stubs (listing 2). In this example we do this by querying for all elements of type "org.eclipse.uml2.uml.Class" that are owned by a package, although we are by no means required to construct the query in this form; we allow everything SPARQL has to offer with the only limitation that the query must be of the SELECT-type and it must return exactly one column (i.e. there must be one variable in the SELECT-clause). Executing this query against our simple model (figure 1) returns two URIs (table 1), one representing each of the two classes in our model and therefore our context instances. We then proceed to create consistency rule instances for every context instance URI the initial query has returned. In our example those consistency rule instances are two queries for an operation called "deactivate", one of the *Switch* class (listing 3) and one of the *Light* class (listing 4)[2]. Listing 3 returns an empty result since the *Switch* class does not contain a *deactivate* operation, while listing 4 returns one result row with variable bindings representing the path in graph (figure 4) starting at the *Light* IRI down to operation name *deactivate*. It is necessary to define a set of more or less complex tests which determine the resulting truth value of the rule instance based on its result set to allow for greater flexibility, but for the sake of simplicity in this paper we assume that a rule evaluates to true exactly if there is at least one result, and false otherwise. Therefore we could rewrite queries 3 and 4 into the ASK form of a SPARQL query, which follows exactly this semantics.

```
1  PREFIX rdf:
2        <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3  PREFIX ma:
4        <http://sea.uni-linz.ac.at/ma/>
5  PREFIX uml:
6        <http://www.eclipse.org/uml2/2.1.0/UML/>
7  SELECT ?x
8  WHERE {
9    ?x ma:type "org.eclipse.uml2.uml.Class" .
10   ?x uml:owner ?y .
11   ?y ma:type "org.eclipse.uml2.uml.Package"
```

[2]In some of the queries in this paper we make use of SPARQL's namespace prefix definition, for instance to shorten *http://www.eclipse.org/uml2/2.1.0/UML/_WodksEiPEd-Xye0lIHC77w* to *uml:_WodksEiPEd-Xye0lIHC77w*. They are both the same thing.

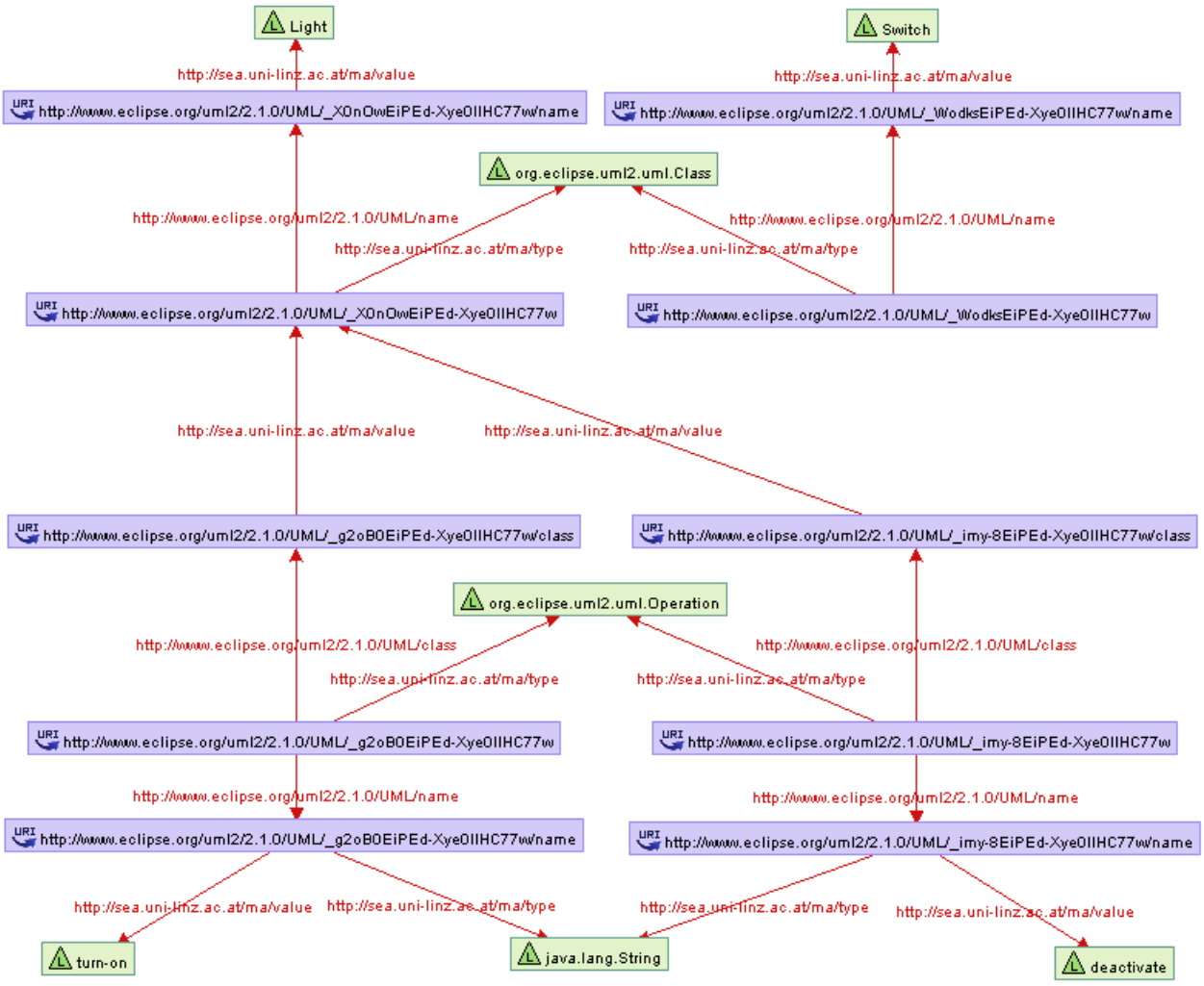

Figure 4: Another subset of "Light Switch" RDF Representation

```
12 }
```

Listing 2: SPARQL Context Query

x
$http://www.eclipse.org/uml2/2.1.0/UML/\_WodksEiPEd-Xye0lIHC77w$
$http://www.eclipse.org/uml2/2.1.0/UML/\_X0nOwEiPEd-Xye0lIHC77w$

Table 1: Result of SPARQL Query in Listing 2

```
1  PREFIX rdf:
2       <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3  PREFIX ma:
4       <http://sea.uni-linz.ac.at/ma/>
5  PREFIX uml:
6       <http://www.eclipse.org/uml2/2.1.0/UML/>
7  SELECT *
8  WHERE {
9    ?deactivate_op_class ma:value
10        uml:_WodksEiPEd-Xye0lIHC77w .
11   ?deactivate_op uml:class ?deactivate_op_class .
12   ?deactivate_op uml:name ?deactivate_op_name .
13   ?deactivate_op_name ma:value "deactivate"
14 }
```

Listing 3: SPARQL Consistency Rule Instance 1 Query

```
1  PREFIX rdf:
2       <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3  PREFIX ma:
4       <http://sea.uni-linz.ac.at/ma/>
5  PREFIX uml:
6       <http://www.eclipse.org/uml2/2.1.0/UML/>
7  SELECT *
8  WHERE {
9    ?deactivate_op_class ma:value
10        uml:_X0nOwEiPEd-Xye0lIHC77w .
```

```
11    ?deactivate_op uml:class ?deactivate_op_class .
12    ?deactivate_op uml:name ?deactivate_op_name .
13    ?deactivate_op_name ma:value "deactivate"
14  }
```

Listing 4: SPARQL Consistency Rule Instance 2 Query

8. Determining a Change Impact Scope

Given a consistency rule instance expressed in SPARQL, our goal is to determine the change impact scope of this rule by transforming the query in a way we can extract the URIs of our RDF graph from the new result set which represent features of our software model we need to monitor in order to determine when to re-execute the rule due to a possibly different resulting truth value caused by user changes in the model. In order to get there, we first look at the structure of a SPARQL consistency rule, starting with our simple example from section 7. To get a better insight, we visualize one of our consistency rule instances ("Class *uml:_WodksEiPEd-Xye0lIHC77w* must have an operation named *deactivate*"; SPARQL: see listings 3) as a query graph (figure 5). This is a graphical representation of the graph pattern our query is looking for in the software model. The variable bindings of a result row of this query represent possible paths in the RDF model from the context instance *uml:_WodksEiPEd-Xye0lIHC77w* to the string literal *deactivate*.

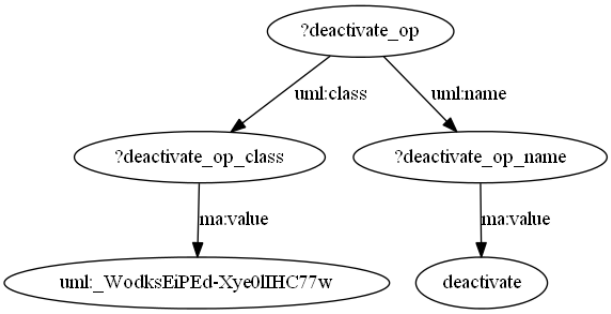

Figure 5: Query Graph of Listing 3

To gain the change impact scope of a rule instance, we are interested in determining a complete (but not necessarily minimal) list of URIs with the property that whenever one of those URIs appears in a notification about a user change in the software model, the result set of said rule instance query could potentially change.

While theoretically a list of *all* URI nodes in the software model would satisfy this requirement, we are looking for a smaller, ideally a minimal list. We need to understand that every edge with its adjacent nodes in figure 5 represents an RDF triple: The node at the shaft of the arrow is our subject, the node at the arrowhead is our object, and the edge label is our predicate. Furthermore, the result set of a rule instance query can only change if an RDF triple is inserted (or deleted) that has a predicate which matches one of the edges in the graph, as this is a requirement for a new path to emerge (or an existing path to disappear). Intuitively one could come up with the very simple approach to determine a change impact scope by doing something like gathering *all* the triples in the RDF store with predicates matching *any one* edge in the graph. While such an approach would theoretically work, i.e. it would deliver a valid change impact scope, it would not be very efficient since the scope would be relatively large. To come up with a smaller, still valid scope, let us take a look at SPARQL's *OPTIONAL* keyword, as described in the official SPARQL W3C recommendation [PS08]:

```
Optional parts of the graph pattern may be specified syntactically
with the OPTIONAL keyword applied to a graph pattern:

pattern OPTIONAL { pattern }

The syntactic form:

{ OPTIONAL { pattern } }

is equivalent to:

{ { } OPTIONAL { pattern } }

[...]

Graph patterns are defined recursively. A graph pattern may have
zero or more optional graph patterns, and any part of a query pattern
may have an optional part. In this example, there are two optional
graph patterns.

[...]
```

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name ?mbox ?hpage
WHERE  { ?x foaf:name  ?name .
         OPTIONAL { ?x foaf:mbox ?mbox } .
         OPTIONAL { ?x foaf:homepage ?hpage }
       }
```

If we now look at our SPARQL query graph in figure 5 as a tree[3], ignoring the directedness, and assume the context instance *uml:_WodksEiPEd-Xye0lIHC77w* to be the root of this tree, we can easily gain another valid change impact scope by traversing this tree and creating a new SPARQL query of the same structure, but which includes an *OPTIONAL* keyword whenever we enter a new level of the tree, and placing a *UNION* keyword between two or more triples on the same level that all lead to an *OPTIONAL* part of the query[4]. In terms of effect on the result set of the query this means that we do not only gain entire paths in the model from the context instance to *deactivate*, but in addition to that we gain all paths starting at the context instance and ending *somewhere on the way* to *deactivate*. We then gather our change impact scope by including all RDF nodes that are in the result set of this new query. We demonstrate the algorithm in pseudo-code listing 5. Performing it on our example in figure 5 generates a new SPARQL query as seen in listing 6.

```
1  query = new graph pattern
2  call traverse(context instance, query)
3
4  function traverse(node, graph pattern):
5    for each child c of node:
6      p = new optional pattern
7      graph pattern.attach(edge that lead to c, p)
8      p.attach(c)
9      traverse(c, p)
10   if node.children.count > 1 then
11         <put UNION between every pair of children>
```

[3]We will cover the special case of cycles in the query graph later.

[4]The *UNION* is required to put the *OPTINAL* branches at par, because without them, a match of the first *OPTIONAL* branch would limit the result of the second etc.

```
12    end if
```

Listing 5: Pseudo-Code of SPARQL Query Generation

```
1  PREFIX rdf:
2       <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3  PREFIX ma:
4       <http://sea.uni-linz.ac.at/ma/>
5  PREFIX uml:
6       <http://www.eclipse.org/uml2/2.1.0/UML/>
7  SELECT *
8  WHERE {
9    OPTIONAL { ?deactivate_op_class ma:value
10             uml:_WodksEiPEd-Xye0lIHC77w .
11   OPTIONAL { ?deactivate_op uml:class
12                  ?deactivate_op_class .
13   OPTIONAL { ?deactivate_op uml:name
14                  ?deactivate_op_name .
15   OPTIONAL { ?deactivate_op_name ma:value
16                  "deactivate"
17           } } } }
18 }
```

Listing 6: SPARQL Change Impact Scope Query for Figure 5

Let us now take a look at why this query produces a valid change impact scope for rules similar to the one in listing 3. We assume to be notified about changes in the software model through change notifications, each consisting of an operation $\in$ {INSERT, DELETE} and an operand which is an RDF triple (subject, predicate, object). As stated before, a re-evaluation of the consistency rule is necessary whenever a new path P between the context instance and a leaf of our query tree emerges (or an existing one disappears) in our software model. We re-evaluate a rule whenever we receive a change notification with an operand with a subject or an object that is in our change impact scope.

Let us first cover the *INSERT* case: For the new path P to emerge, it is trivially necessary that at least one new RDF triple is inserted that represents an edge of our tree. Let us assume that in the first place a partial path of P already exists, starting at the context instance node and spanning over an arbitrary number x of edges towards

a leaf (including the case of $x = 0$). For P to emerge, it is now necessary that at some point we receive an INSERT change notification with an operand that matches the $(x + 1)$th edge on the path towards our leaf, because without this RDF triple our path P can never be completed. Since this change notification obviously includes as its subject or object the node that comes after the xth transition on our path, which is in our change impact scope, we will re-evaluate the rule.

The *DELETE* case is almost trivial: Our change impact scope already contains all the paths from the context instance to the leafs of our query tree that form the result of the consistency rule. Once we receive a *DELETE* notification of an RDF triple that is part of one of those paths, both subject and object of this triple are in our change impact scope and therefore we re-evaluate the rule.

9. Special Case: Cycles

In this section we are dealing with the special case of the query graph having cycles, i. e. when we can not look at it as a tree as we did previously. To demonstrate this situation, let us take a look at the simple state chart in figure 6. Our example consists of two states, "On" and "Off"', and a transition from each to the other. We can find a relevant subset of the model's RDF representation in figure 7. At the bottom of the diagram we can see a region, on top of that the two states with their respective names, at the top of the diagram the two transitions with their names and at the center the source and target of both transitions. Let us now formulate another rather simple consistency rule: *Every region must have an "On" state of which a transition called "deactivate" must lead to another state which in turn must have another transition called "activate" that leads back to the original state.* After performing a query to gather all the context instances of the rule with the context element being *Region*, one of our rule instances could look like listing 7. Executing this rule instance query against our model in figure 7 will return one result row which we interpret as the rule evaluating to *true*. If we now proceed to draw our query graph (figure 8), we can see that the graph is not acyclic, which means we can not interpret it as a tree as we did in the previous example. In order to gain our change impact scope query, we need to resolve the cycle in the graph, which we can achieve by removing any one of the triple patterns that form it. Let us for instance remove *?activatetarget ma:value ?onstate* and we gain a new query graph as illustrated in figure 9.

```
1 PREFIX rdf:
```

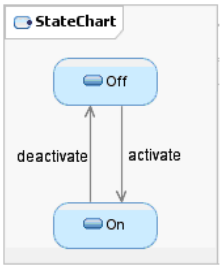

Figure 6: "Light Switch" State Chart

```
2       <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3  PREFIX ma:
4       <http://sea.uni-linz.ac.at/ma/>
5  PREFIX uml:
6       <http://www.eclipse.org/uml2/2.1.0/UML/>
7  SELECT *
8  WHERE {
9    ?onstate uml:owner
10         uml:_BGMEIEiREd-Xye0lIHC77w/owner .
11   ?onstate uml:name ?onstatename .
12   ?onstatename ma:value "On" .
13   ?deactivatesource ma:value ?onstate .
14   ?deactivate uml:source ?deactivatesource .
15   ?deactivate uml:target ?deactivatetarget .
16   ?deactivatetarget ma:value ?offstate .
17   ?activatesource ma:value ?offstate .
18   ?activate uml:source ?activatesource .
19   ?activate uml:target ?activatetarget .
20   ?activatetarget ma:value ?onstate
21 }
```

Listing 7: SPARQL Rule Instance

We can now interpret the query graph as a tree, again ignoring the directedness of the edges, starting at the context instance *uml:_ WodksEiPEd-Xye0lIHC77w* as our root node and can then apply our algorithm from pseudo-code listing 5 to transform the query. The resulting query will include all paths of the RDF graph that match any sub-tree of our query in figure 8 that shares the same root node *uml:_ WodksEiPEd-Xye0lIHC77w*.

Figure 7: Subset of "Light Switch" RDF Representation

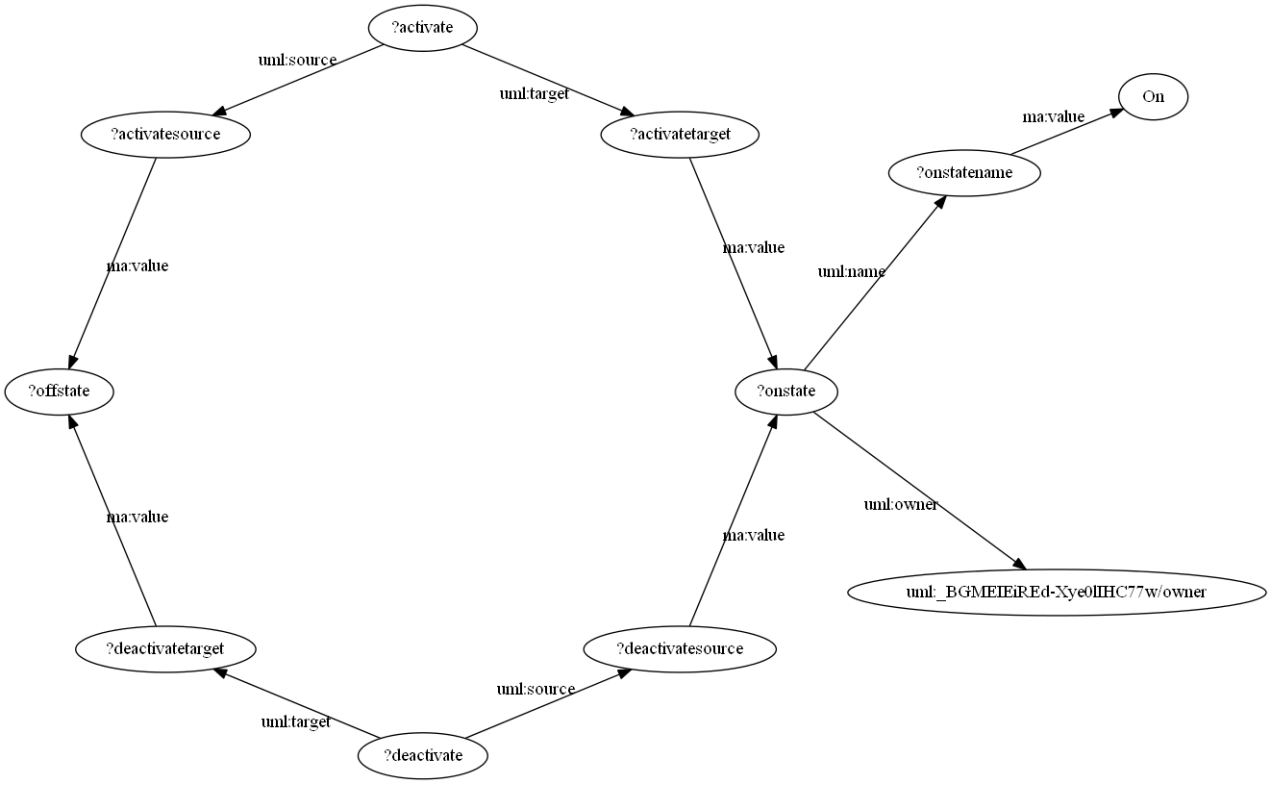

Figure 8: Query Graph of Listing 7

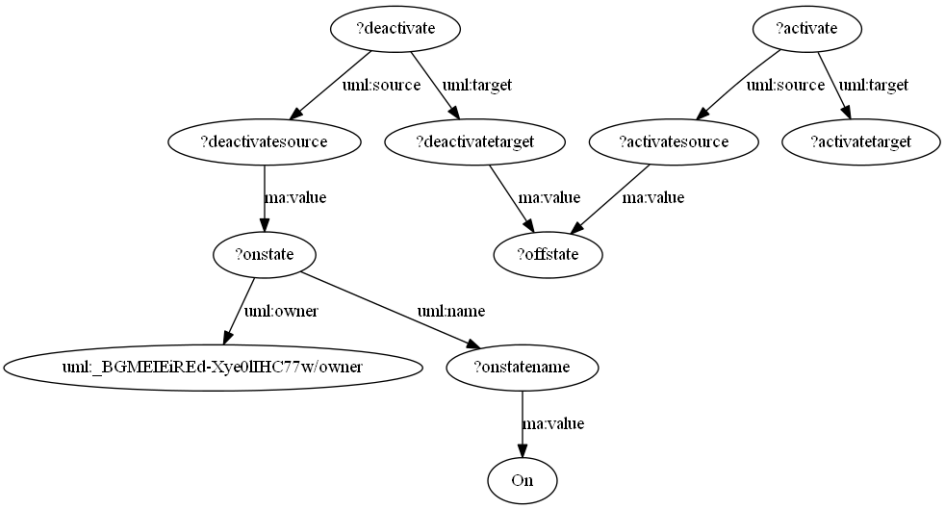

Figure 9: Modified Query Graph of Listing 7

To demonstrate that this is query generates a valid change impact scope, let us first assume that our state chart example does not have a transition called *deactivate* yet. The change impact scope returned by the transformed SPARQL query would then contain the nodes of the graph in figure 10. For the *INSERT* case, if we now add the *deactivate* transition to the model, we will have to insert many RDF triples, and the object of one of them must be *uml:_BGMEIEiREd-Xye0IIHC77w* because this is where the *deactivate* transition "attaches" to the *On* state, which is the transition's source state. As soon as this happens, the surrounding framework delivers an *INSERT* change notification containing the *On* state's stub node *uml:_BGMEIEiREd-Xye0IIHC77w* which is in our change impact scope, so the rule will be re-evaluated. The order in which all the *INSERT* notifications that together form the insertion of our *deactivate* transition arrive does not matter, because whatever model changes have happened before, at some point the transition has to "attach" to the *On* state's stub in our change impact scope which is the first chance (actually the exact point in time) when the result of the rule instance query changes.

Again, the *DELETE* case is trivial and follows the same logic as in the previous example.

```
1  PREFIX rdf:
2       <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3  PREFIX ma:
4       <http://sea.uni-linz.ac.at/ma/>
5  PREFIX uml:
6       <http://www.eclipse.org/uml2/2.1.0/UML/>
7  SELECT *
8  WHERE {
9    ?onstate uml:owner <uml:_BGMEIEiREd-Xye0lIHC77w/owner> .
10   { OPTIONAL {
11     ?onstate uml:name ?onstatename .
12   OPTIONAL {
13     ?onstatename ma:value "On"
14   } } }
15   UNION
16   {
17   OPTIONAL {
18     ?deactivatesource ma:value ?onstate .
```

```
19      OPTIONAL { ?deactivate uml:source
20                      ?deactivatesource .
21      OPTIONAL { ?deactivate uml:target
22                      ?deactivatetarget .
23      OPTIONAL { ?deactivatetarget ma:value
24                      ?offstate .
25      OPTIONAL { ?activatesource ma:value
26                      ?offstate .
27      OPTIONAL { ?activate uml:source
28                      ?activatesource .
29      OPTIONAL { ?activate uml:target
30                      ?activatetarget
31              } } } } } } } }
32  }
```

Listing 8: Change Impact Scope Query of SPARQL Rule Instance

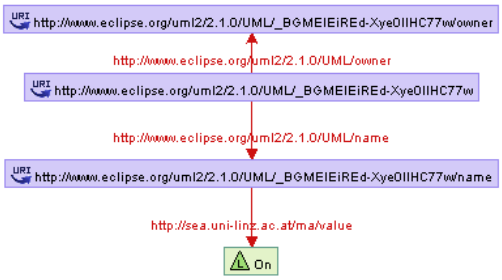

Figure 10: Nodes in Change Impact Scope of Example in Listing 8

Following the same logic, we extend our algorithm to support SPARQL queries with variables in the predicate of a graph pattern triple.

10. Implementation

We implemented a system that demonstrates the functionality of the described approach, covering almost the entire language specification of SPARQL [PS08]. In this section we will discuss the implementation of our approach as outlined by the system diagram in figure 11 and then explain its interface to the environment. Again, our goal is to gain a valid change impact scope, i. e. a set of RDF nodes that we need to observe with

the property that whenever one of them appears as the subject or object in a change notification, we need to re-evaluate the rule.

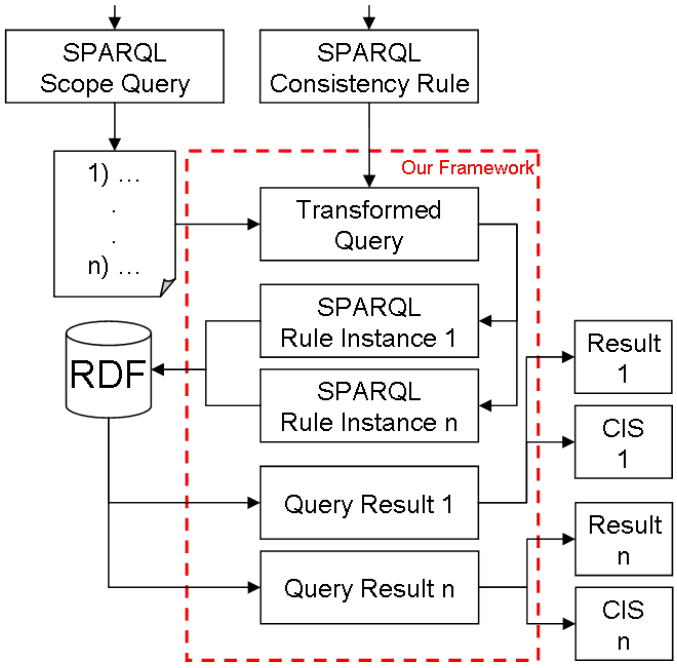

Figure 11: System Diagram

10.1. Transforming SPARQL

The first step in our transformation process is to lex and parse a textual representation of a consistency rule expressed in SPARQL (a $LL(1)$ grammar) and turn it into an abstract syntax tree, which we will then use to perform our transformation before using a code generator to eventually create the textual representation of the transformed SPARQL query. In this paper we are working with the *SPARQL Query Language for RDF* W3C recommendation dated 15 January 2008 [PS08]. We take advantage of an already existing SPARQL grammar for the ANTLR parser generator[5] to create a lexer and parser for the SPARQL language. The *sparkle-g* [TMP12] project contains such a tree grammar and we use it to generate a parser that creates an *abstract syntax tree* of the rule query. In a first pre-processing step we unify some of the various SPARQL notations of basic graph patterns in order to simplify the following steps. We proceed with building the

[5] *ANother Tool for Language Recognition*, is a language tool that provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in a variety of target languages. [Par12]

query tree as described earlier in this paper and then creating a new abstract syntax tree representing the transformed query, which we *UNION* with the original query in order to preserve its result in addition to the change impact scope. The following sections describe those steps in detail by working with another simple example of a consistency rule.

10.1.1. Example

To demonstrate the implementation of our approach we will use the following simple consistency rule: "Every class must have an operation called *turn-on* and an operation called *deactivate*", with *Class* obviously being the context of this rule. For flexibility reasons we allow the context instances to be arbitrary RDF nodes the user provides in the form of a list and we refer to it in the rule instances with a specially annotated variable. In this particular case the user could gain such a list of classes (the context) by executing a SPARQL query similar to the one in listing 2 with resulting context instances as shown in table 1 (each binding of x represents one class' stub RDF node). In a next step, we express the consistency rule itself in SPARQL, for instance as in listing 9. Since this is the generic SPARQL representation of the consistency rule we do not include a concrete context instance in the query, but rather refer to it by using a variable called *?context*. Later we will tell the system as a parameter of an interface call that this is the variable that is to be substituted with the context instances, one at a time.

```
1  PREFIX rdf:
2       <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3  PREFIX ma:
4       <http://sea.uni-linz.ac.at/ma/>
5  PREFIX uml:
6       <http://www.eclipse.org/uml2/2.1.0/UML/>
7  SELECT *
8  WHERE {
9    ?turnon_op_class ma:value ?context .
10   ?turnon_op uml:class ?turnon_op_class ;
11   uml:name ?turnon_op_name .
12   ?turnon_op_name ma:value "turn-on" .
13   ?deactivate_op_class ma:value ?context .
```

```
14    ?deactivate_op uml:class ?deactivate_op_class ;
15    uml:name ?deactivate_op_name .
16    ?deactivate_op_name ma:value "deactivate"
17  }
```

Listing 9: SPARQL Consistency Rule Query

10.1.2. Lexing and Parsing

We use the *sparkle-g* [TMP12] ANTLR tree grammar to generate a Java tree parser for SPARQL. When feeding our example rule from listing 9 to the parser we gain the abstract syntax tree in figure 12.

10.1.3. Pre-Processing

Since SPARQL allows multiple different notations to address identical sets of triple patterns, we do some pre-processing to unify the abstract syntax tree in order to simplify the actual transformation process. SPARQL offers three types of triple pattern notation: *subject* - *predicate* - *object*, *subject* with a following *predicate* - *object* list and *subject* - *predicate* with a following *object* list. Of course those different notations result in different abstract syntax trees and since they are nothing but syntactic sugar for one and the same thing, we prevent several special cases in the following implementation by assuring that only the first kind of pattern, *subject* - *predicate*- *object*, is present in the input queries. The following citation [PS08] demonstrates the syntax of the latter two pattern types by example:

```
4.2.1 Predicate-Object Lists

Triple patterns with a common subject can be written so that the
subject is only written once and is used for more than one triple
pattern by employing the ";" notation.

    ?x  foaf:name  ?name ;
        foaf:mbox  ?mbox .

This is the same as writing the triple patterns:
```

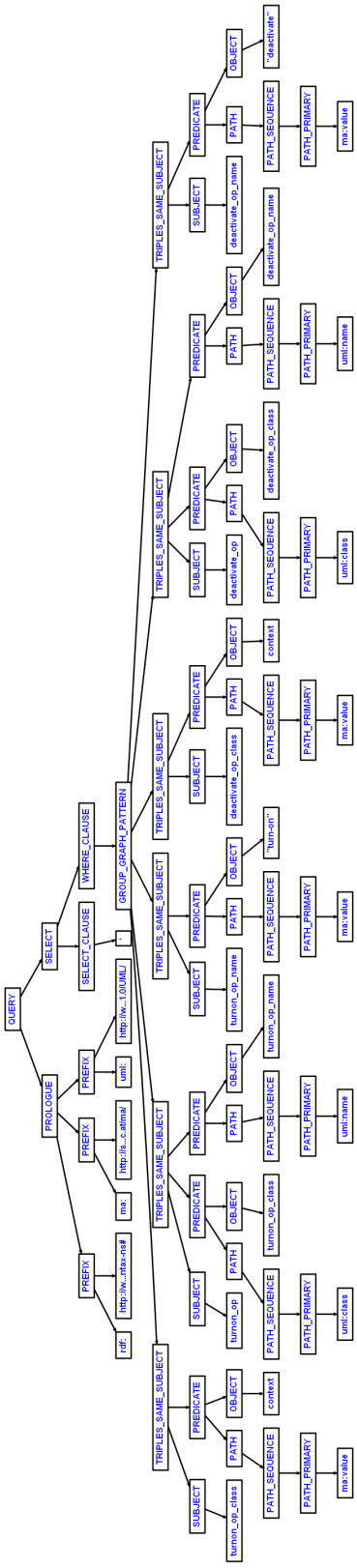

Figure 12: Abstract Syntax Tree of Query in Listing 9

```
    ?x  foaf:name  ?name .
    ?x  foaf:mbox  ?mbox .
```

4.2.2 Object Lists

If triple patterns share both subject and predicate, the objects may be separated by ",".

```
    ?x foaf:nick  "Alice" , "Alice_" .
```

is the same as writing the triple patterns:

```
   ?x  foaf:nick  "Alice" .
   ?x  foaf:nick  "Alice_" .
```

Object lists can be combined with predicate-object lists:

```
   ?x  foaf:name ?name ; foaf:nick  "Alice" , "Alice_" .
```

is equivalent to:

```
   ?x  foaf:name  ?name .
   ?x  foaf:nick  "Alice" .
   ?x  foaf:nick  "Alice_" .
```

In our pre-processing step we eliminate potential *predicate-object lists* and *object lists* in the query and replace them with the regular *subject - predicate - object* triple pattern notation. Applying this step to our abstract syntax tree from the example leads to a new abstract syntax tree as illustrated in figure 13.

10.1.4. Transformation

Starting with our unified abstract syntax tree, we extract the triple patterns and use them to build a query DAG of the rule, where each variable or literal is represented by a vertex and each predicate by an edge. The directedness of the edges denotes which vertex is the subject (shaft of the arrow) and which is the object (head of the arrow).

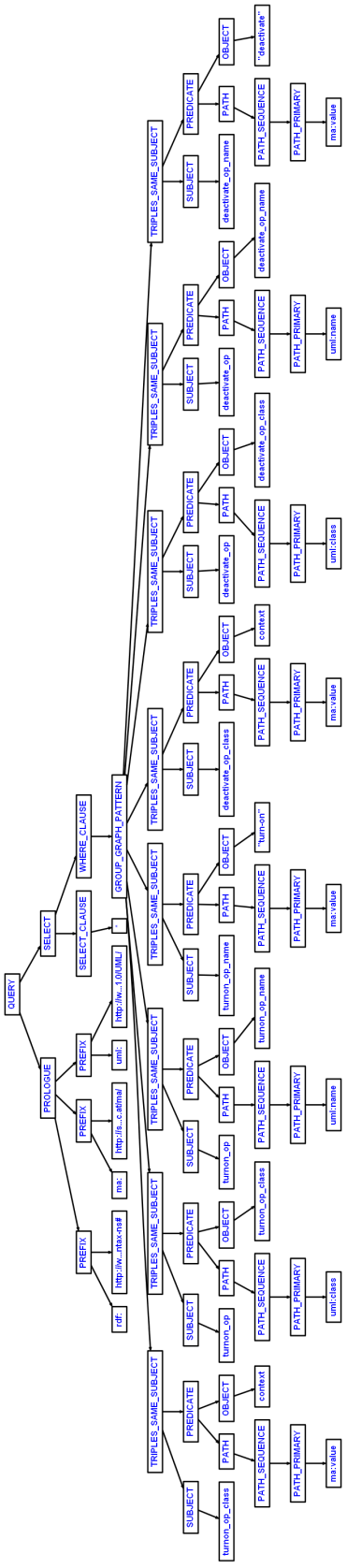

Figure 13: Unified Abstract Syntax Tree of Query in Listing 9

We do so by starting at the annotated context variable (*?context*) and then adding edges leading to or from new vertices, one for every triple pattern in the query that references the current vertex as its subject or object. This leads us to the query graph in figure 14. As specified earlier in this paper, our goal is to gain a graph which becomes a tree as soon as we ignore the directedness of the edges and consider the context variable to be the root, and we already demonstrated the correctness of this approach. In order to get there, we stop the graph building process as soon as we reach a vertex that has already been processed (compare figure 8 vs. figure 9).

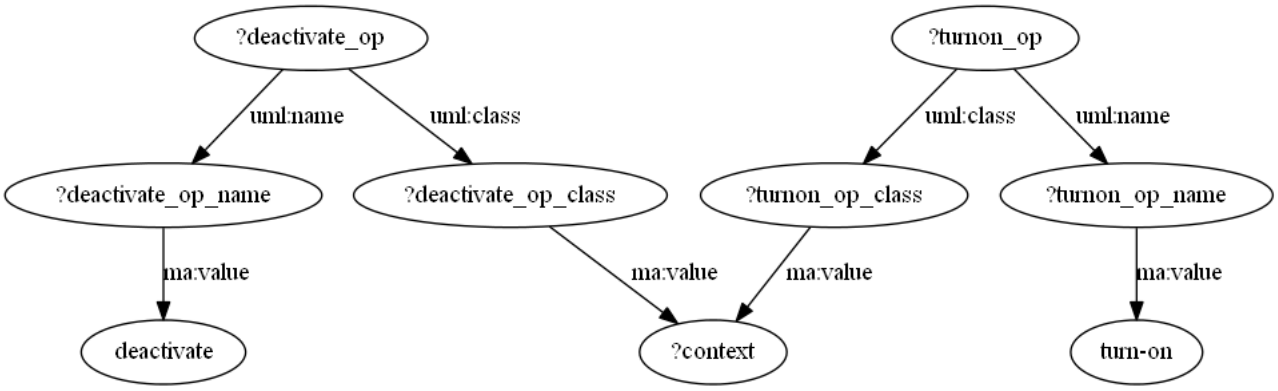

Figure 14: Query Graph of Query in Listing 9

In a next step, we apply our algorithm from pseudo-code listing 5 to actually transform the query to return a change impact scope. We start at the root of our query "tree" and traverse it, opening a new "level" of *OPTIONALs* whenever we step into the next depth level of the query tree and closing it when the current sub-tree has been processed. We use this algorithm to generate a fragment of an abstract syntax tree that queries for the change impact scope of the original SPARQL query. In a next step, we replace all the SPARQL variable names in that fragment such that they don't conflict with the variable names used in the original query and then we *merge* both by introducing a new top-level SPARQL *UNION* node with the original query pattern and our newly generated change impact scope query pattern as its children. In case the *SELECT* clause does not contain an asterisk, we add all the newly introduced variables to it. The result of those processing steps is the abstract syntax tree split into figures 15 and 16, which, after passing it to our code generator, yields the SPARQL query in listing 10. Before executing the query, we turn the query into a consistency rule *instance* by binding the *?context* variable (and the corresponding variable in the change impact scope part of the query) to one concrete instance of the rule's context element.

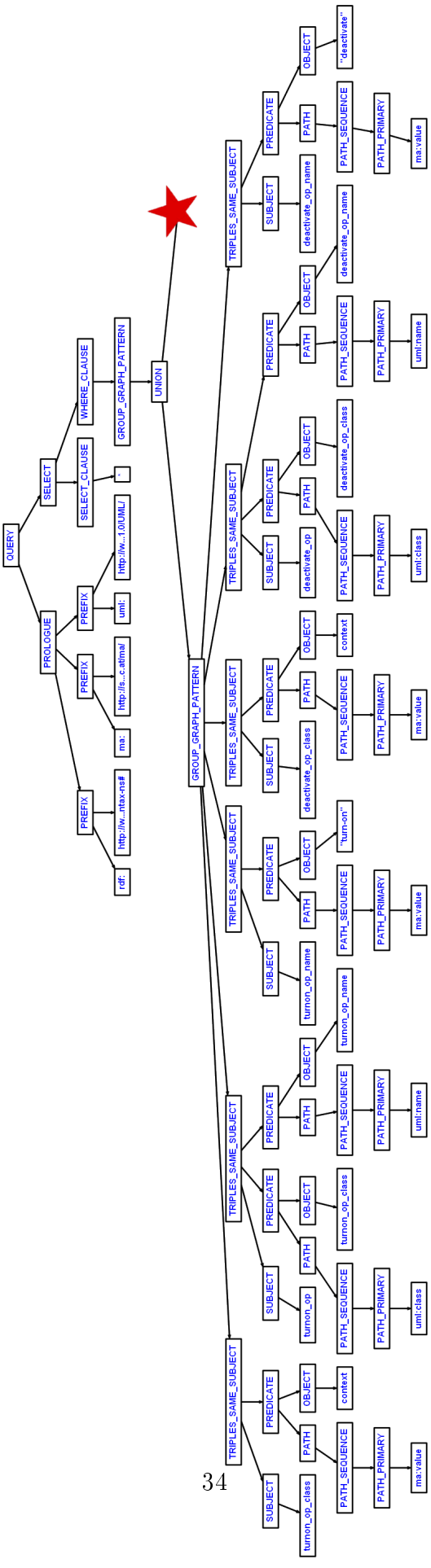

Figure 15: Resulting Abstract Syntax Tree of Query in Listing 9 (Part 1/2)

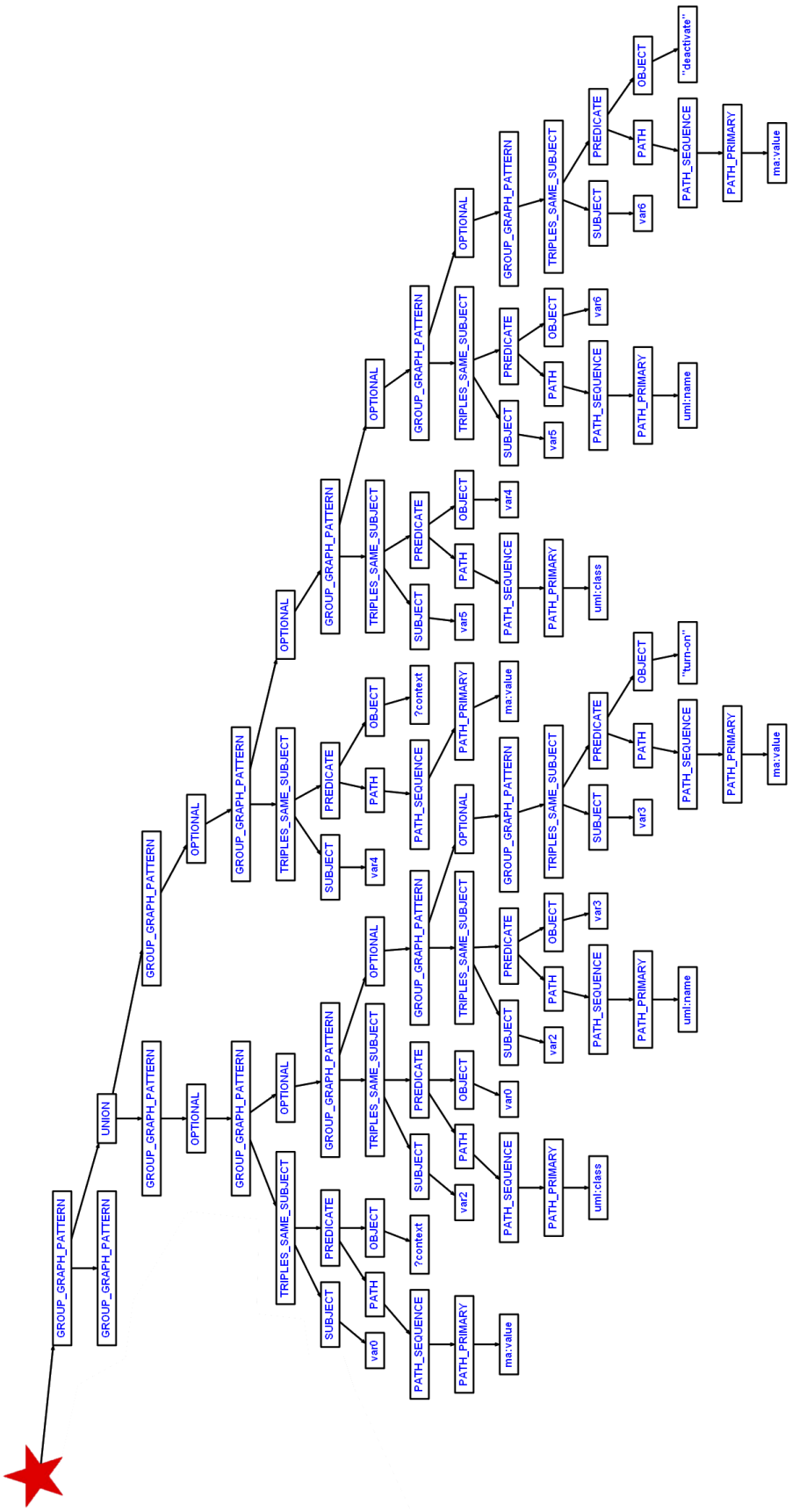

Figure 16: Resulting Abstract Syntax Tree of Query in Listing 9 (Part 2/2)

```
1  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2  PREFIX ma: <http://sea.uni-linz.ac.at/ma/>
3  PREFIX uml: <http://www.eclipse.org/uml2/2.1.0/UML/>
4  SELECT *
5  WHERE {
6  { ?turnon_op_class ma:value ?context .
7    ?turnon_op uml:class ?turnon_op_class ;
8    uml:name ?turnon_op_name .
9    ?turnon_op_name ma:value "turn-on" .
10   ?deactivate_op_class ma:value ?context .
11   ?deactivate_op  uml:class ?deactivate_op_class ;
12   uml:name ?deactivate_op_name .
13   ?deactivate_op_name ma:value "deactivate" .
14 } UNION {
15 { OPTIONAL { ?var0 ma:value ?context .
16   OPTIONAL { ?var2 uml:class ?var0 .
17   OPTIONAL { ?var2 uml:name ?var3 .
18   OPTIONAL { ?var3 ma:value "turn-on" .
19 } } } } }
20 UNION
21 { OPTIONAL { ?var4 ma:value ?context .
22   OPTIONAL { ?var5 uml:class ?var4 .
23   OPTIONAL { ?var5 uml:name ?var6 .
24   OPTIONAL { ?var6 ma:value "deactivate" .
25 } } } } } } }
```

Listing 10: Resulting SPARQL Query of Our Example

Pseudo-code listing 11 gives a coarse overview of the core transformation algorithm.

```
1  call unify_triples_same_subject()
2
3  /* variables keep record of the owning triple */
4  variables = { }
5
6  for every triple t in the original query:
7    if not variables.contains(t.subject) then
```

```
8       variables.add(t.subject)
9     end if
10    variables[t.subject].attach(t)
11    if not variables.contains(t.object) then
12      variables.add(t.object)
13    end if
14    variables[t.object].attach(t)
15
16  root = variables[context]
17  call traverse(root, { }, new group_graph_pattern)
18
19
20  function traverse(node, covered, graph_pattern):
21    for each child c of node:
22      p = new optional_pattern(c.triple)
23      if (node.children.indexOf(c) < node.children.size - 1) then
24        <put UNION node between all children and current node>
25      end if
26      graph_pattern.attach(p)
27      if !covered.contains(c) then
28        call traverse(c, covered.union(c), p)
29      endif
```

Listing 11: Pseudo-Code of SPARQL Transformation

10.1.5. Post-Processing

The result set of the transformed query consists of the result set of the original query, where any of the columns representing the original query is not *NULL*, and then a set of rows where the columns of the original query remain *NULL* and the columns of our additional variables contain information about the change impact scope of our consistency rule instance. In order to gather our change impact scope, we iterate over the latter part of the result set and create a list of unique IRIs which serves as the change impact scope as described earlier in this paper.

Part II.
Towards an OCL to SPARQL Translation

11. Introduction

Independent of the chosen technology of persistently storing an UML model we need to provide the ability to evaluate OCL consistency rules against the model, since OCL is part of the UML specification. For many reasons it is desirable to evaluate those rules as low as possible in a tiered architecture of a software modelling utility, one of them being that following this approach it becomes superfluous to (partially) load and map the model into main memory data structures multiple times before we can evaluate OCL expressions using traditional methods, which is advantageous in systems with dynamic loading of model fragments. An ideal solution to this task would be a complete translation from OCL to a query language that is native to the persistence technology used for storing the software model, such as SPARQL in the case of RDF. In our concrete case, the question arises whether or not the expressive power of SPARQL is enough to cover all the concepts of OCL. Existing work such as [AG08] suggests that SPARQL has an expressiveness equivalent to that of relational algebra, while other papers like [HWDD] try to tackle the translation from OCL to SQL (see part IV of this paper). These facts together let us conclude that in theory it *might* be possible to define a mapping from OCL consistency rules to SPARQL consistency rules. Another advantage of the evaluation of consistency rules in the persistence layer in situations where, for example, the system is distributed over multiple physical machines and the execution of single SPARQL queries causes a high overhead in execution time, due to network communication or other factors. In those situations, even the possibility of fetching *in one single SPARQL query* all the model information that an interpreter requires to evaluate a given consistency rule constitutes a notable advantage. In this part of the paper we describe an approach that does exactly this and due to the potential reusability the work might serve as the groundwork for a complete translation from OCL to SPARQL. Like in the first part of the paper, we assume a software design model to be stored in an RDF triple store and furthermore we assume that the execution of

(SPARQL) queries against that model to exhibit a high constant time penalty due to various overheads such as network communication and a general system-inherent delay. In situations where we want to evaluate consistency rules expressed in OCL against such a model, using an interpreter to fetch tiny portions of the software model on demand in order to evaluate an OCL rule is highly inefficient due to multiple delays caused by the constant execution overhead. Our goal here is to develop a method that, starting at a given OCL consistency rule, will generate one single SPARQL query that fetches all the necessary information about a software model that is sufficient for the OCL interpreter to evaluate the consistency rule without performing additional RDF queries. The approach is generic enough to suit different RDF representations of software models.

12. Introduction to OCL

The *Object Constraint Language* (OCL) is an addition to UML that serves as a formal language to specify constraints on UML software models. It is a declarative language that is side-effect free, which means that the execution of OCL expressions can never cause any changes in the software model it is evaluated against. OCL is an important addition to UML in that it allows specifying precise semantics on top of the (graphical) software models. [Dem09] The three most commonly used OCL constraints are invariants, pre-conditions and post-conditions. Invariants are constraints that have to be satisfied at all times, while pre-conditions and post-conditions have to be satisfied before or after the execution of some program code. Our OCL constraints consist of two parts: The context definition and the expression that is evaluated against all context element instances and that can refer to the context element instance by using the *self* keyword. An example of an OCL constraint (invariant) that states that the *participants* attribute of a class called *Meeting* must be at least of size 2 can be found in listing 12.

```
1 context Meeting
2 inv: self.participants->size()>=2
```

Listing 12: Simple OCL Example

13. Determining the Accessed Model Fragment

Let us look at the example OCL consistency rule in listing 13. In general, most of the model elements accessed during the evaluation of an OCL expression are repre-

sented by dereference terms consisting of a chain of names with optional additions like, for example, formal parameters enclosed by parenthesis. One of those dereference terms in the example is *self.namespace.oclAsType(Package).packagedElement*. In order to fetch the information required by the OCL interpreter to evaluate the rule, we need to fetch the resulting elements of *all* those dereference terms in the OCL expression, each representing a set of model accesses, and then let the interpreter perform the rule's semantics on top of those elements. In addition to that we need to keep track of the variables used in the expression, which are nothing but abbreviations for dereference terms (or other variables that eventually resolve to them). For instance, *children* in the example refers to a subset of the elements returned by *self.namespace.oclAsType(Package).packagedElement*, where the logic to determine this subset is based on additional model accesses on the set of elements returned by the navigation term. So in order to feed all the information of this concrete example necessary for the interpreter to interpret the *LET* expression, we need to remember the results of *self.namespace.oclAsType(Package).packagedElement* (the first dereference term), *self.namespace.oclAsType(Package).packagedElement.oclIsTypeOf(Class)* (the left operand of the *AND* operation in the select specification), and *self.namespace.oclAsType(Package).packa* (the information necessary to compute the right operand of the *AND* operation in the select specification).

```
1  Context: Class
2  Description: parent class should not have an attribute
3               referring to a child class
4  OCL:
5  let children:Set(NamedElement) =
6  self.namespace.oclAsType(Package).packagedElement->
7  select(pe:PackageableElement|pe.oclIsTypeOf(Class) and
8  pe.oclAsType(Class).allParents()->includes(self)) in
9  self.ownedAttribute->forAll(p:Property|
10 p.type.oclIsTypeOf(Class) implies children->
11 excludes(p.type.oclAsType(Class)))
```

Listing 13: Example OCL Consistency Rule

14. RDF Mapping and Type Resolving

Since there is no globally valid mapping from software design models to RDF triples, in order to determine what portions of the RDF model we need to fetch to enable the interpreter to process a consistency rule, we extract the semantics of the dereference terms to a system-specific module that provides type and mapping information.

We do so by introducing an interface with an option to call to the surrounding system which will accept as parameters the type of the current model element combined with the subsequent operation or property name in the navigation term and return the type of the resulting elements combined with two SPARQL fragments that together represent this one OCL navigation step: One fragment for the actual value of the new model element and one for its stub. (Due to the simple structure of RDF triples we typically find it necessary to facilitate some kind of *stub* node for every model element, which most likely is an URI representing it, and we need to remember it to perform additional navigation steps on the element.)

Let us now look at it in the form of an example, considering the navigation step *self* $\rightarrow$ *namespace*. We already know the type of the initial model element, which in this case is the context type *Class*. (Another way of gaining the initial type would be a previous application of this method.) We will now ask the surrounding system through an interface call as described above the following: If we navigate from a model element of type *Class* to its property *namespace*...

1. ...what is the type name of the property *namespace*?
2. ...what is the SPARQL fragment to navigate from the RDF stub node of the *Class* instance to the RDF stub node of the property *namespace*?
3. ...what is the SPARQL fragment to navigate from the RDF stub node of the *Class* instance to the RDF value node of the property *namespace*?

The surrounding system will then return the following answers (compare to the model fragment in figure 17):

1. *Namespace*
2. ?in uml:namespace ?out
3. ?in uml:namespace ?x . ?x ma:value ?out

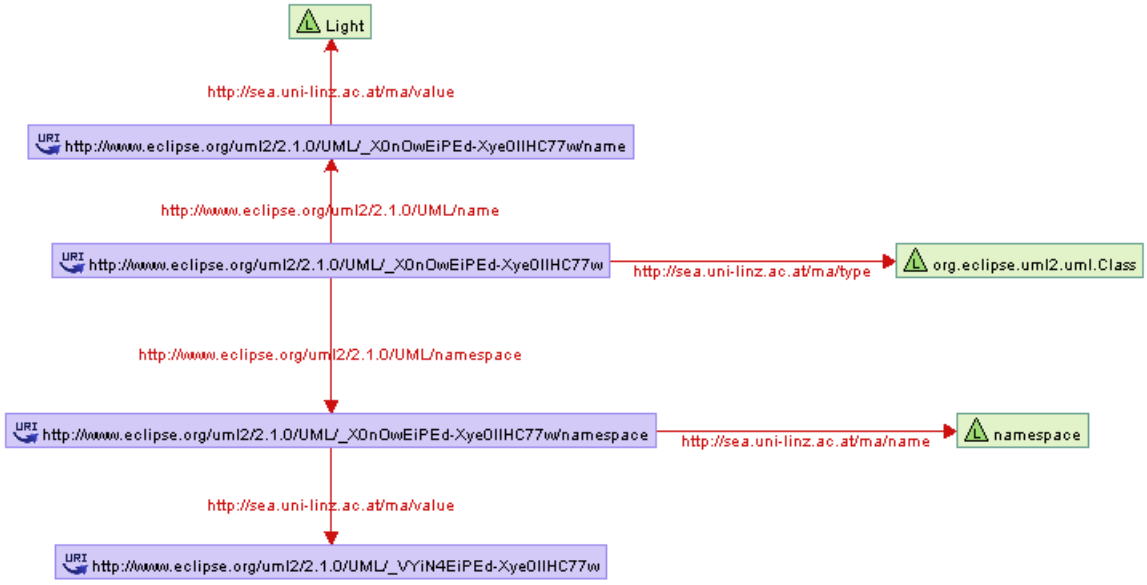

Figure 17: RDF Fragment of Example Model

Please note that this method allows the user to also fetch an arbitrary set of additional RDF nodes that may later be required by the interpreter simply by querying for them in one of the two SPARQL fragments returned by the interface call.

We can now repeat this whole process and thereby resolve the entire navigation term *self.namespace.oclAsType(Package).packagedElement*, one by one name (see figure 18).

Initial Type	Name	New Type	Stub SPARQL	Value SPARQL
Class	namespace	Namespace	?in uml:namespace ?out	?in uml:namespace ?x . ?x ma:value ?out
Namespace	oclAsType(Package)	Package	?in ma:value ?out	?in ma:value ?out
Package	packagedElement	PackageableElement	?in packagedElement ?x . ?x ?y ?out	?in packagedElement ?x . ?x ?y ?out

Figure 18: Resolving an Entire Navigation Term

15. Implementation

15.1. OCL Parser

The starting point for the implementation of this approach will be a parser generated by the *jacc* compiler compiler from an *EBNF* specification of OCL. *jacc* is short for *just*

another compiler compiler and it creates bottom-up/shift-reduce parsers targeting the Java language and its syntax is *closely syntactically compatible with Johnson's classic yacc parser generator for C.* [Jon04] It takes an input file specifying things such as, among others, a java package name, import statements, a class name, the names of the methods linking the parser to the scanner, token names and the operator precedence. For every EBNF production rule we can specify a semantic annotation in the form of a plain java statement enclosed in curly brackets. The java code specified in those annotations stays completely unchecked during the parser generation and the only changes *jacc* performs are literal replacements of $$ with the return value of the current production rule and of all $n with the return value of the n'th parameter on the right side of the production rule. In the first line of the following example, $1 will become whatever (Object) it is that *PrimitiveLiteralExp* returns and *LiteralExp* itself will return the same Object, due to the assignment of $1 to $$.

```
LiteralExp : PrimitiveLiteralExp    { $$ = $1; }
           | CollectionLiteralExp { $$ = $1;
                System.out.println("Hello world!"); }
           | TupleLiteralExp       { $$ = $1; }
                      ;
```

We use this parser and the semantic annotation to generate a list of all model accesses that are occurring in any given OCL expression. In order to stay independent of a particular mapping from a software model to an RDF model, we introduce a framework specific abstract resolver class that provides a SPARQL fragment representing every dereference in an OCL expression.

15.2. System Description

As stated before, the starting point of our implementation is the OCL parser. During its operation, whenever it encounters a name token that marks the beginning of a dereference expression as described before in this paper, it will create a new *ModelFragment* instance that represents this dereference expression. It will try to look up the first name of that expression in its variable scope in order to determine its type and its SPARQL expressions, against which the subsequent names will be resolved later. If the first name of such an expression does not occur in the variable list, the system will raise an error since it encountered an unknown name. Every subsequent name in the current

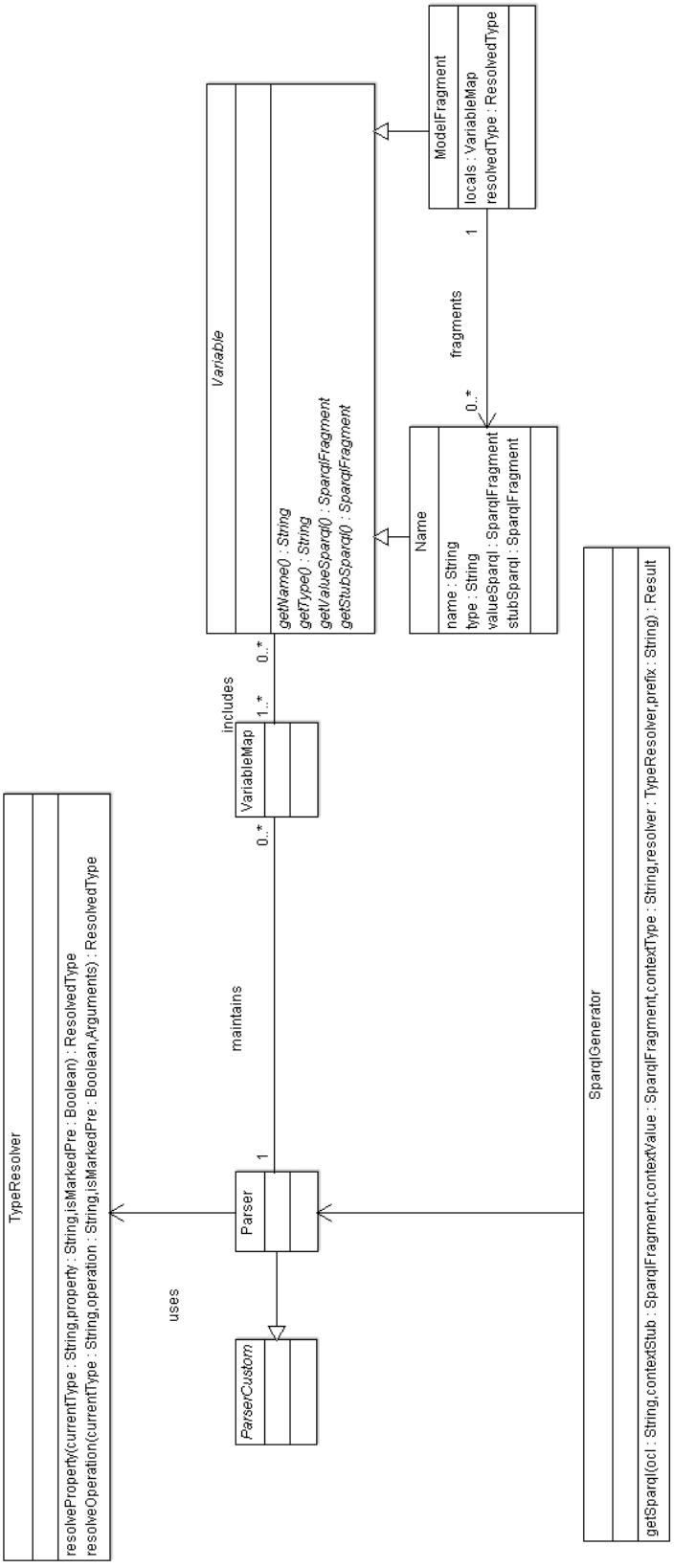

Figure 19: Class Diagram

expression will be added to the *fragments* list of the *ModelFragment*. This process is repeated until the parser completed parsing the OCL expression, while we remember every generated *ModelFragment* instance.

In the next step our goal is to derive the SPARQL fragments that query an RDF triple store for all the model elements that will be accessed during the evaluation of our OCL expression. To do so, we iterate over all our *ModelFragment* instances and call their respective *getSparql()* method. This method will in turn iterate over all its fragments (i.e. names), starting by looking up both stub and value SPARQL fragments that yield the model element represented by the first name in the fragment list. Those fragments include an out-anchor SPARQL variable that we textually replace with a new, query-wide unique variable name which also serves as the anchor point for the SPARQL fragment of the subsequent name in the fragment list. As described before, the system uses our *TypeResolver* to determine the type and SPARQL fragments of the next name in the fragment list. All but the first names in the list have both in and out anchor variables that are used to link the names' SPARQL fragments to eventually form a SPARQL query pattern representing the current *ModelFragment*. We keep track of all the *ModelFragments* and the newly introduced variables in their SPARQL representation to allow a mapping of the SPARQL results to the dereference expressions.

Once all the *ModelFragments* are converted to SPARQL fragments, we simply add them together and prepend the SPARQL prefix that we receive as an input parameter and that also contains the namespace definitions, and a *SELECT* clause with all the relevant variables used in any of the SPARQL fragments.

15.3. Variable Stack

Some OCL constructs, like *LET* expressions or iterators on collection types introduce variables that are not global and have a limited scope in the expression. In order to handle those variables and their scopes correctly in our approach, we introduce a variable stack. Each element on this stack is a variable map representing the scope of the most recently introduced variables. Whenever the parser reaches a point in the OCL expression that introduces a new variable, we perform a *push()* operation on the variable stack and thereby add a new variable map that holds all the variables in the current scope. When the parser reaches the point where the scope of the variable ends, it will perform a *pop()* operation and dispose of the top variable map and all the variables in it that have reached their end of life. Let us take another look at the example in listing

13 and compare it to the variable stack diagram in figure 20. At the very beginning we introduce the very first scope in the variable map stack which only holds one variable: *self* (t=1). Our system receives the binding of this variable as an input parameter. Then, the *pe:PackageableElement* part in the first *select* operation introduces another variable called *pe* (t=2) with a scope that spans over the body of the *select* operation. Then, the *LET* expression introduces a variable called *children* (t=3) which is valid only for the part of the *LET* expression after the *in* keyword (which in this case is until the end of the OCL expression). The last variable is introduced by the *p:Property* part of the *forAll* iterator (t=4) and it is also only valid inside the body of the *forAll* operation.

Every *ModelFragment* as described in section 15.2 holds a flat list of variables that are valid at the place in the OCL expression where the corresponding dereference expression occurred. When we instantiate a new *ModelFragment*, we copy all the variables from all levels of the variable stack to its internal variable list, while variables of an inner scope have precedence over variables of an outer scope and will overwrite them.

Part III. Evaluation

In order to evaluate our SPARQL transformation, we use the examples illustrated earlier in this paper, feed the corresponding SPARQL queries into our system and then analyze whether the produced *change impact scopes* are correct. Since the effort to manually verify those change impact scopes in the full-blown RDF representation of a real-world UML model would be unreasonably high, considering the potentially huge size of the resulting change impact scopes, and the infrastructure to perform higher-level tests in the context of a software modelling utility is not available to me, we will use our own rudimentary UML-RDF representation to validate our system. Since the algorithm in principle operates on any kind of software model represented in RDF, this approach does not violate any of our constraints.

Let us start with a rather basic example, which is the following consistency rule: "Every class must have an operation called *deactivate*." As our model, we pick the RDF graph as shown in figure 21. In SPARQL, we can represent this consistency rule as shown in listing 14. In order to apply this consistency rule, we need to replace the context variable with a context instance, in this case first *ex:Class1* and then *ex:Class2*.

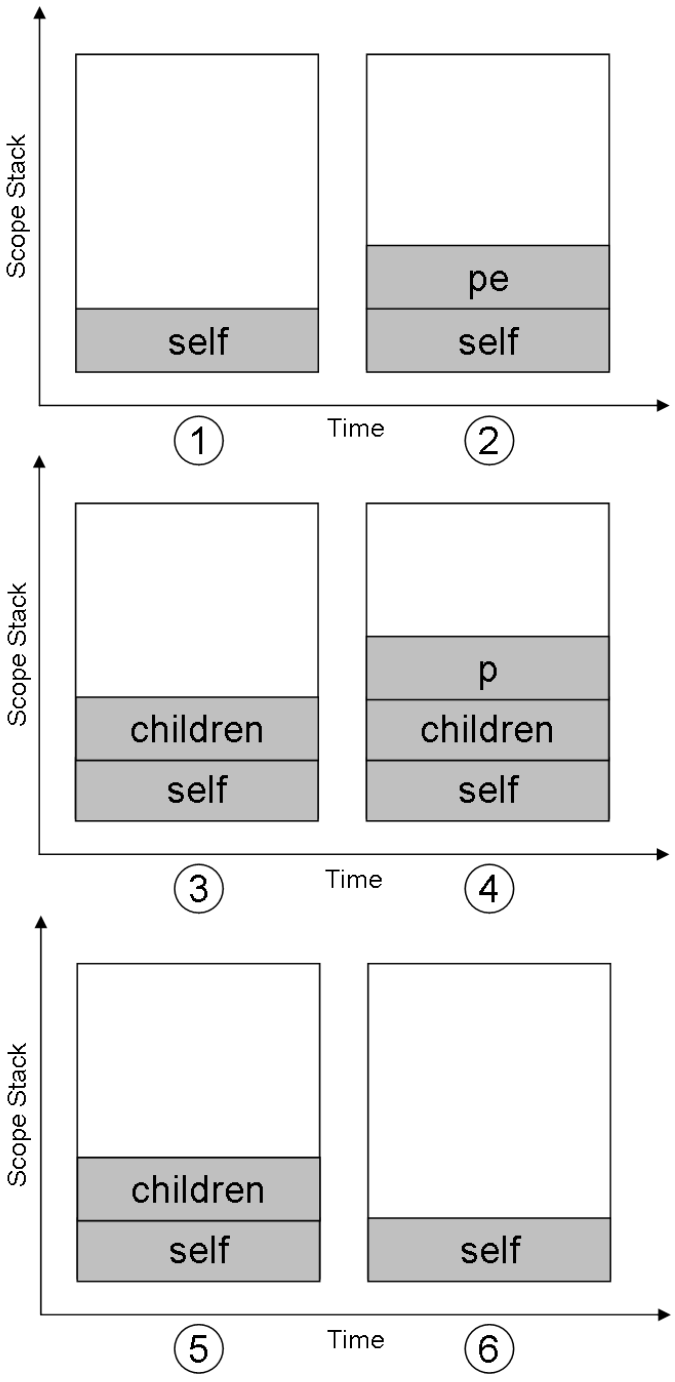

Figure 20: Variable Stack Example

If we use *ex:Class1* as the context instance, the expected change impact scope is as follows: *Class1*, *op1*, and *op3*. Executing our system with these parameters produces exactly this change impact scope. If we use *ex:Class2* as our context instance, we expect our change impact scope to consist of *Class2* and *op2*, which is exactly what our system returns.

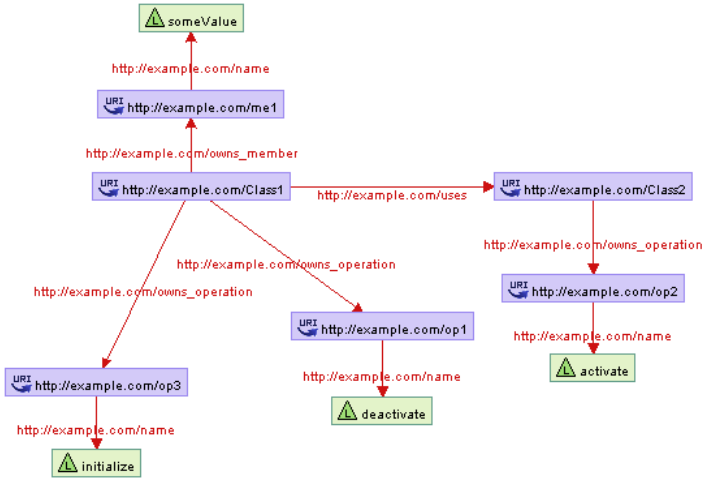

Figure 21: RDF Representation of Basic Example

```
1 PREFIX ex: <http://example.com/>
2 SELECT *
3 WHERE {
4   ?context ex:owns_operation ?op .
5   ?op ex:name "deactivate"
6 }
```

Listing 14: SPARQL Consistency Rule of Basic Example

Let us now look at another example based on the state diagram in figure 22. Our diagram consists of two states, one called "On" and one called "Off". There are two transition, one called "activate" and one called "deactivate". We now assume the following consistency rule: "Every region must have an "On" state of which a transition called "deactivate" must lead to another state which in turn must have another transition called "activate" that leads back to the original state." Again, in SPARQL, we can express this consistency rules as shown in listing 15. Our expected change impact scope in this case is the following: *ex:Tr1*, *Tr2*, *St1*, *St2*, *Re1* which is correctly computed

by our system. During and after the development, we have successfully evaluated the system using a number of more or less complex examples as illustrated here, which is also the case for the second mechanism described in this paper.

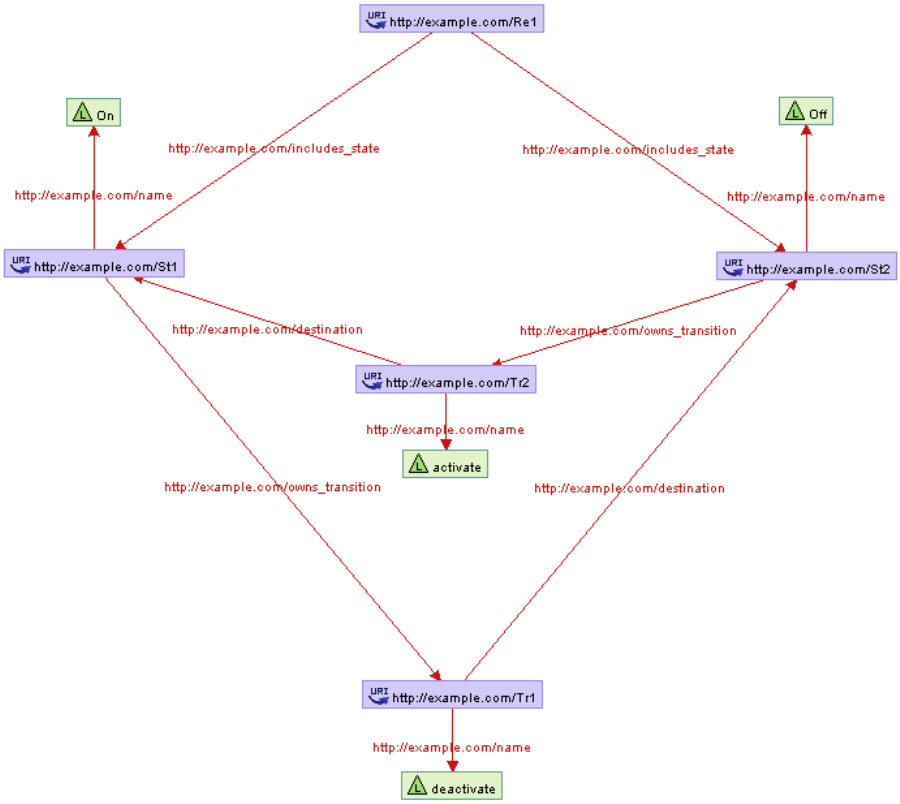

Figure 22: RDF Representation of Second Example

```
1  PREFIX ex: <http://example.com/>
2  SELECT *
3  WHERE {
4    ?context ex:includes_state ?on_state .
5    ?on_state ex:name "On" .
6    ?on_state ex:owns_transition ?deactivate .
7    ?deactivate ex:name "deactivate" .
8    ?deactivate ex:destination ?off_state .
9    ?off_state ex:name "Off" .
10   ?off_state ex:owns_transition ?activate .
11   ?activate ex:name "activate" .
12   ?activate ex:destination ?on_state
```

```
13 }
```

Listing 15: SPARQL Consistency Rule of Second Example

Part IV.
Related Work

16. The Expressive Power of SPARQL

While in this paper we provide a method of determining the change impact scope of a consistency rule expressed in SPARQL and a method to pre-fetch part of the model that is required for an OCL interpreter to interpret an OCL consistency rule against a model stored in RDF, we did not discuss the question whether or not SPARQL is expressive enough to cover all the possible consistency rules that the OCL grammar can produce. A positive answer to that question would enable an interesting continuation of our work: the direct translation of OCL expressions to SPARQL.

In their paper, Angles et al. [AG08] claim that they were able to prove that SPARQL has the expressiveness of relational algebra. They define a set of transformations from SPARQL to *non-recursive safe Datalog with negation*, show that SPARQL is contained in the latter one, and then prove by applying the transformations that the results of queries before and after the transformation produce equivalent results (and everything vice-versa) and then state that since *non-recursive safe Datalog with negation* has the same expressive power as relational algebra, SPARQL has the same expressive power as relational algebra. This in turn implies that *if a transformation from OCL to SQL exists, it should in theory be possible to perform a direct transformation from OCL to SPARQL.*

17. A Framework for Generating Query Language Code from OCL Invariants

In their paper, [HWDD] Florian Heidenreich et al. claim they have developed a generic software framework to create query language code from OCL invariants in the context

of model-driven software development (MDSD). They state that most current MDSD approaches only focus on transforming structural descriptions of software systems, while neglecting semantical integrity rules. Their framework consists of three parts: The first one reads the UML/OCL model and creates an abstract syntax model of it, while the second one performs the transformation of the UML model to the target data schema. The third part maps OCL invariants to declarative query languages. For this transformation, they identified common patterns that occur in OCL constraints. The framework provides the possibility for the developer to specify the equivalent code fragments of the target relational algebra query language of his choice. They exhibit an example of a transformation from OCL to SQL, however the rules in the target system are enforced by creating a *VIEW* on one or multiple database tables and then applying view-based integrity checks. Little implementation details are given so it remains questionable as to whether or not their approach functions without database-specific features that are outside the scope of relational algebra.

18. On the Expressive Power of OCL

In another interesting paper, [MC99] Luis Mandel and María Victoria Cengarle investigate the expressive power of OCL. They show that the expressiveness of OCL and relational calculus are not identical, however they do so only by showing that some of the constructs of relational calculus are not expressible in OCL, which does not allow the converse argument that OCL is not contained in relational calculus. Another finding was that due to the missing capability of computing some recursive functions, OCL is not equivalent to a Turing machine.

19. Transformation Techniques for OCL Constraints

J. Cabot and E. Teniente [CT07] investigate different syntactic possibilities to define equivalent integrity constraints. They show alternative expressions for some constructs in OCL that yield identical semantics, for instance by replacing the context element of a constraint, replacing collection operators with others and applying Boolean algebra laws. Depending on the interpreter, some of those semantically identical integrity constraints are more efficient to evaluate than others. *In order to further increase the efficiency of incremental consistency checking, one could take those aspects into account and perform a transformation of OCL consistency rules before their evaluation.*

20. Automatically Detecting and Visualizing Errors in UML Diagrams

In [CCMS02], Campbell et al. build on top of their "formalization framework that attaches formal semantics to a subset of UML diagrams used to model embedded systems" and describe automated structural and behavioural analyses applicable to UML diagrams. They state that one of the reasons UML has become a *de facto* standard in software modelling and development is that there is no out-of-the-box formal semantics for UML and therefore it is customizable to a variety of domains. They have developed a general formalization framework that supports several target languages, for example *VHDL*, *Promela*, and the *SPIN* specification language. The tool chain first consists of *MINERVA*, which supports the graphical construction of UML diagrams and translates them into a textual representation, *HIL*. The second component is their framework, *Hydra*, which takes the textual representation of the UML diagrams and turns it into an appropriate formal specification in a particular target language. In their paper they specifically show how the SPIN model checker can then be used to automatically analyze UML diagrams. The result of that analysis are returned to *MINERVA*, where it is visually displayed to the user.

21. Using ViewPoints for Inconsistency Management

In [EN95], Steve Easterbrook et al. define *ViewPoints* as loosely coupled, locally managed, distributable objects which encapsulate partial knowledge about a system and its domain, specified in a particular, suitable representation scheme, and partial knowledge of the process of development. They partition a development task into several ViewPoints and then maintain consistency rules that describe the relationships between various ViewPoints. Any two ViewPoints do not need to be consistent with each other during the entire development process, but rather inconsistencies are allowed and every ViewPoint keeps a list of unresolved inconsistencies involving itself. When a change in the software model is made in one of the ViewPoints, only the consistency rules related to the modified ViewPoint are re-evaluated.

22. Detecting Model Inconsistency through Operation-Based Model Construction

Xavier Blanc et al. distinguish between structural and methodological consistency rules, where the earlier ones are constraints that are evaluated against a model in a certain state, at one point in time, while methodological consistency rules consider the sequence of actions that lead to the current model. They define classes of actions that, executed in a certain order, lead to a certain model. Those actions can either be the creation of a model element, the assignment of a value or reference, or the deletion of the model element. Instead of only working with consistency rules that are based on a model in its current state, they additionally allow us to impose constraints on the sequence of actions that lead to the model, and they express those constraints in predicate calculus such that existing logical inference engines can be used to evaluate the consistency rules.

23. xlinkit: A Consistency Checking and Smart Link Generation Service

Christian Nentwich et al. [NCEF02] describe a lightweight application service called *xlinkit* that checks the consistency of distributed web content. It is given a set of distributed XML resources and a set of potentially distributed rules, expressed in a defined language, that relate the content of those resources. In the context of consistency checking, their system returns a set of hyperlinks between inconsistent elements instead of Boolean values. Their focus does not lie on avoiding inconsistencies at any price, but rather providing diagnostic information in the case of inconsistencies, which they do by providing those links between conflicting elements. In order to avoid having to re-evaluate the entire universe of elements and rules after every change, they introduce a partitioning mechanism that allows defining document sets that are subsets of the entire modelled system. They applied their technology to UML models supplied by industrial partners.

References

[AG08] Renzo Angles and Claudio Gutierrez. The expressive power of sparql. In *Proceedings of the 7th International Conference on The Semantic Web*, ISWC

'08, pages 114–129, Berlin, Heidelberg, 2008. Springer-Verlag.

[CCMS02] Laura A. Campbell, Betty H. Cheng, William E. Mcumber, and R. E. K. Stirewalt. Automatically detecting and visualising errors in uml diagrams. *Requirements Engineering*, V7(4):264–287, December 2002.

[CT07] J. Cabot and E. Teniente. Transformation techniques for ocl constraints. *Sci. Comput. Program.*, 68(3):152–168, October 2007.

[Dem09] Dr. Birgit Demuth. Einführung in ocl, 2009.

[Egy11] Alexander Egyed. Automatically detecting and tracking inconsistencies in software design models. *IEEE Transactions on Software Engineering*, 37:188–204, march/april 2011.

[EN95] Steve Easterbrook and Bashar Nuseibeh. Using viewpoints for inconsistency management. *Software Engineering Journal*, 11:31–43, 1995.

[HWDD] Florian Heidenreich, Christian Wende, Birgit Demuth, and Technische Universität Dresden. A framework for generating query language code from ocl invariants.

[Jon04] Marc P. Jones. jacc: just another compiler compiler for java, 2004.

[MC99] Luis Mandel and María Victoria Cengarle. On the expressive power of ocl. In *Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume I - Volume I*, FM '99, pages 854–874, London, UK, UK, 1999. Springer-Verlag.

[NCEF02] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: a consistency checking and smart link generation service. *ACM Transactions on Internet Technology (TOIT)*, 2(2):151–185, 2002.

[Par12] Terence Parr. Antlr parser generator, October 2012.

[PS08] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. W3C recommendation, W3C, January 2008. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[TMP12] Simone Tripodi, Michele Mostarda, and Juergen Pfundt. sparkle-g, October 2012.

A. Example Output

Textual Description	**The class name should be 'ClassName'**
OCL Consistency Rule	`Context: Class` `self.name = 'ClassName'`

SPARQL Consistency Rule*/**	`PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>`
	`PREFIX ma: <http://sea.uni-linz.ac.at/ma/>`
	`PREFIX uml: <http://www.eclipse.org/uml2/2.1.0/UML/>`
	`SELECT ?context ?name`
	`WHERE {`
	`  ?context ma:type "org.eclipse.uml2.uml.Class" .`
	`  ?context uml:name ?x .`
	`  ?x ma:value ?name`
	`}`

SPARQL Change Impact Scope Query	`PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>`
	`PREFIX ma: <http://sea.uni-linz.ac.at/ma/>`
	`PREFIX uml: <http://www.eclipse.org/uml2/2.1.0/UML/>`
	`SELECT ?context ?name ?var2 ?var0 ?var1`
	`WHERE {`
	`{ ?context ma:type "org.eclipse.uml2.uml.Class" .`
	`  ?context uml:name ?x .`
	`  ?x ma:value ?name . }`
	`UNION`
	`{ { OPTIONAL { ?test ma:type "org.eclipse.uml2.uml.Class" . } }`
	`  UNION`
	`  { OPTIONAL { ?test uml:name ?var1 . OPTIONAL { ?var1 ma:value ?var2 . } } } }`
	`} }`

SPARQL Model Fragment Query***	`PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> PREFIX ma: <http://sea.uni-linz.ac.at/ma/> PREFIX uml: <http://www.eclipse.org/uml2/2.1.0/UML/>`
	`SELECT ?v19`
	`WHERE {`
	`{?v18 ma:type "org.eclipse.uml2.uml.Class" . ?v18 uml:name ?cn . ?cn ma:value "Switch" . ?v18 uml:name ?cn . ?cn ma:value ?v19}`
	`}`

Textual Description	**At most one association end may be an aggregation or composition**
OCL Consistency Rule	`Context: Association` `self.memberEnd->size()>0 implies self.memberEnd->select(p|p.aggregation<>AggregationKind::none)->size()<=1`

SPARQL Consistency Rule*/**	`PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>`
	`PREFIX ma: <http://sea.uni-linz.ac.at/ma/>`
	`PREFIX uml: <http://www.eclipse.org/uml2/2.1.0/UML/>`
	`SELECT ?context ?t`
	`WHERE {`
	`  ?context uml:memberEnd ?me .`
	`  ?me ?x ?e .`

*) For illustrative purposes; better SPARQL representations might exist.
**) Requires some logic on result set to evaluate.
***) Custom type resolver for illustrative purposes was used.

	?e uml:aggregation ?a . ?a ma:type "org.eclipse.uml2.uml.AggregationKind" . ?a ma:value ?t . }

SPARQL Change Impact Scope Query	PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
	PREFIX ma: <http://sea.uni-linz.ac.at/ma/>
	PREFIX uml: <http://www.eclipse.org/uml2/2.1.0/UML/>
	SELECT ?context ?t ?var5 ?var3 ?var4 ?var0 ?var1 ?var2
	WHERE {
	{ ?context uml:memberEnd ?me .
	?me ?x ?e .
	?e uml:aggregation ?a .
	?a ma:type "org.eclipse.uml2.uml.AggregationKind" .
	?a ma:value ?t .
	} UNION {
	OPTIONAL { ?test uml:memberEnd ?var1 .
	OPTIONAL { ?var1 ?var2 ?var3 .
	OPTIONAL { ?var3 uml:aggregation ?var4 .
	{ OPTIONAL { ?var4 ma:type "org.eclipse.uml2.uml.AggregationKind" . } }
	UNION
	{ OPTIONAL { ?var4 ma:value ?var5 . } }
	} } } } }

SPARQL Model Fragment Query***	PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> PREFIX ma: <http://sea.uni-linz.ac.at/ma/> PREFIX uml: <http://www.eclipse.org/uml2/2.1.0/UML/>
	SELECT ?v7 ?v11 ?v15
	WHERE {
	{?v6 ma:type "org.eclipse.uml2.uml.Association" . ?v6 uml:name ?cn . ?cn ma:value "AName" . ?v6 uml:memberEnd ?me .
	?me ?x ?v7}
	UNION
	{?v0 ma:type "org.eclipse.uml2.uml.Association" . ?v0 uml:name ?cn . ?cn ma:value "AName" . ?v0 uml:memberEnd ?me .
	?me ?x ?v14 . ?v14 uml:aggregation ?b . ?b ma:value ?v15}
	}

Textual Description	**Association ends must have a unique name within the Association**
OCL Consistency Rule	Context: Association OCL: self.memberEnd->forAll(p1,p2:Property|p1<>p2 implies p1.name<>p2.name)
SPARQL Consistency Rule*/**	PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> PREFIX ma: <http://sea.uni-linz.ac.at/ma/> PREFIX uml: <http://www.eclipse.org/uml2/2.1.0/UML/> SELECT ?context ?v WHERE { ?context uml:memberEnd ?me .

*) For illustrative purposes; better SPARQL representations might exist.
**) Requires some logic on result set to evaluate.
***) Custom type resolver for illustrative purposes was used.

	?me ?x ?e . ?e uml:name ?n . ?n ma:value ?v . }

SPARQL Change Impact Scope Query	PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
	PREFIX ma: <http://sea.uni-linz.ac.at/ma/>
	PREFIX uml: <http://www.eclipse.org/uml2/2.1.0/UML/>
	SELECT ?context ?v ?var5 ?var3 ?var4 ?var0 ?var1 ?var2
	WHERE {
	{ ?context uml:memberEnd ?me .
	?me ?x ?e .
	?e uml:name ?n .
	?n ma:value ?v .
	} UNION {
	OPTIONAL { ?test uml:memberEnd ?var1 .
	OPTIONAL { ?var1 ?var2 ?var3 .
	OPTIONAL { ?var3 uml:name ?var4 .
	OPTIONAL { ?var4 ma:value ?var5 .
	} } } } } }

SPARQL Model Fragment Query***	PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> PREFIX ma: <http://sea.uni-linz.ac.at/ma/> PREFIX uml: <http://www.eclipse.org/uml2/2.1.0/UML/>
	SELECT ?v27 ?v29 ?v31 ?v35 ?v39
	WHERE {
	{?v26 ma:type "org.eclipse.uml2.uml.Association" . ?v26 uml:name ?cn . ?cn ma:value "AName" . ?v26 uml:memberEnd ?me . ?me ?v40 ?v27}
	UNION
	{?v20 ma:type "org.eclipse.uml2.uml.Association" . ?v20 uml:name ?cn . ?cn ma:value "AName" . ?v20 uml:memberEnd ?me . ?me ?v41 ?v38 . ?v38 uml:name ?cn . ?cn ma:value ?v39}
	}

Textual Description	**A class must use unique attribute names**
OCL Consistency Rule	Context: Class OCL: self.ownedAttribute->forAll(p1,p2:Property|p1<>p2 implies p1.name<>p2.name)
SPARQL Consistency Rule*/**	PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> PREFIX ma: <http://sea.uni-linz.ac.at/ma/> PREFIX uml: <http://www.eclipse.org/uml2/2.1.0/UML/> SELECT ?context ?nv WHERE { ?context uml:ownedAttribute ?oa . ?oa ?x ?a . ?a uml:name ?n . ?n ma:value ?nv

*) For illustrative purposes; better SPARQL representations might exist.
**) Requires some logic on result set to evaluate.
***) Custom type resolver for illustrative purposes was used.

	}

SPARQL Change Impact Scope Query	PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
	PREFIX ma: <http://sea.uni-linz.ac.at/ma/>
	PREFIX uml: <http://www.eclipse.org/uml2/2.1.0/UML/>
	SELECT ?context ?nv ?var1 ?var3 ?var4 ?var0 ?var5 ?var2
	WHERE {
	{ ?context uml:ownedAttribute ?oa .
	?oa ?x ?a .
	?a uml:name ?n .
	?n ma:value ?nv .
	} UNION {
	OPTIONAL { ?test uml:ownedAttribute ?var1 .
	OPTIONAL { ?var1 ?var2 ?var3 .
	OPTIONAL { ?var3 uml:name ?var4 .
	OPTIONAL { ?var4 ma:value ?var5 .
	} } } } } }

SPARQL Model Fragment Query***	PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> PREFIX ma: <http://sea.uni-linz.ac.at/ma/> PREFIX uml: <http://www.eclipse.org/uml2/2.1.0/UML/>
	SELECT ?v11 ?v13 ?v15 ?v19 ?v23
	WHERE {
	{?v10 ma:type "org.eclipse.uml2.uml.Class" . ?v10 uml:name ?cn . ?cn ma:value "AName" . ?v10 uml:ownedAttribute ?me . ?me ?v20 ?b . ?b ma:value ?v11}
	UNION
	{?v0 ma:type "org.eclipse.uml2.uml.Class" . ?v0 uml:name ?cn . ?cn ma:value "AName" . ?v0 uml:ownedAttribute ?me . ?me ?x ?v21 . ?b ma:value ?v18 . ?v18 uml:name ?cn . ?cn ma:value ?v19}
	}

*) For illustrative purposes; better SPARQL representations might exist.
**) Requires some logic on result set to evaluate.
***) Custom type resolver for illustrative purposes was used.