Supplement to "Beyond Low-Code Development: Marrying Requirements Models and Knowledge Representations": RSL-DL metamodel, implementation of semantic rules and extended case study excerpts

1 RSL-DL meta-model

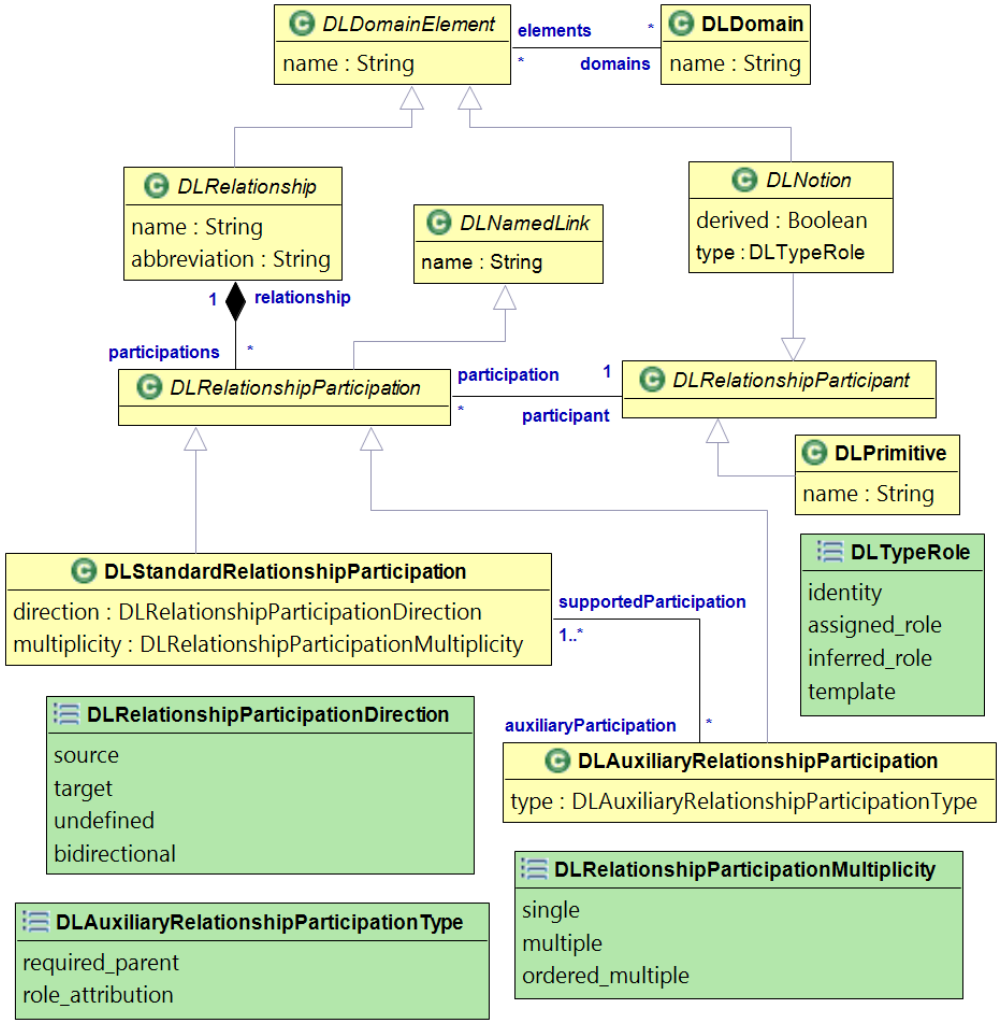

Figure 1: Meta-model of the core RSL-DL elements

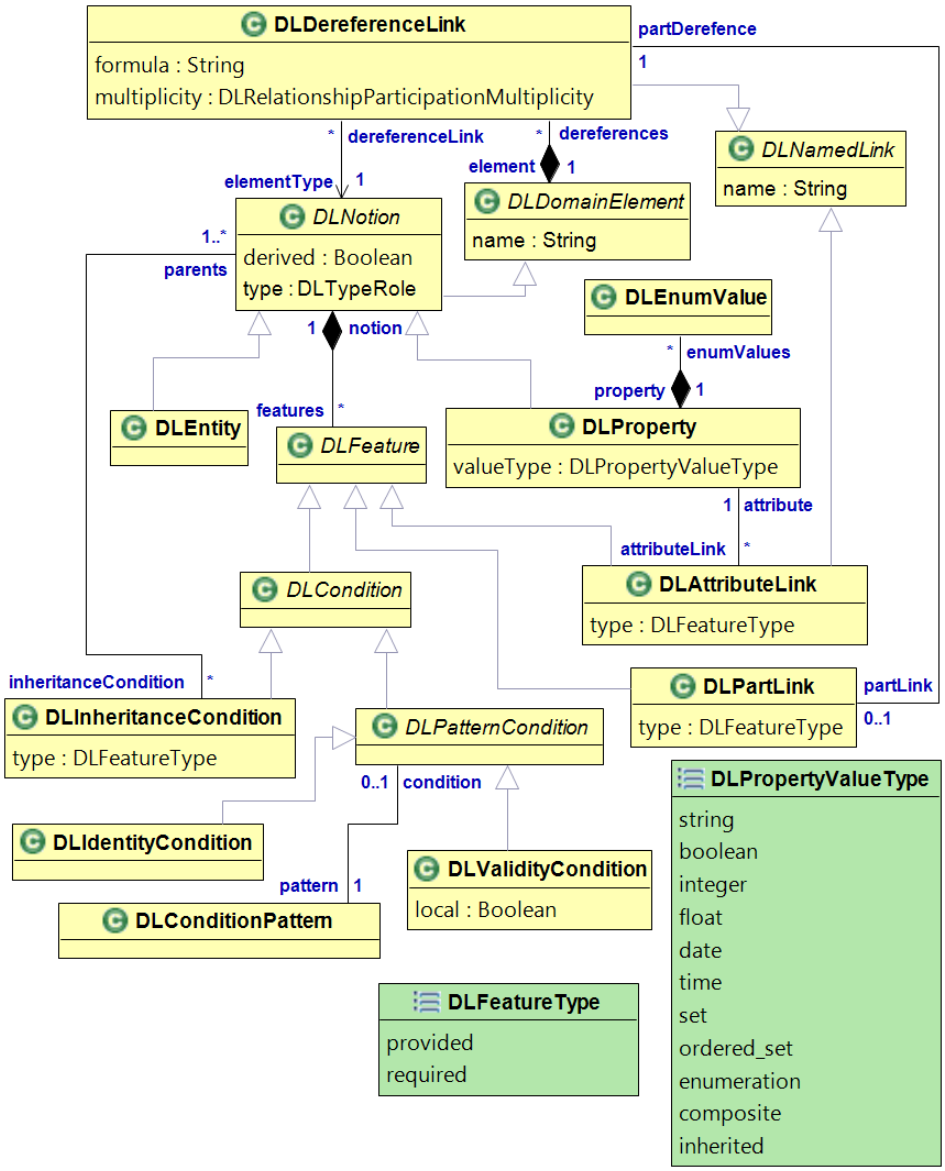

Figure 2: Meta-model details for notions

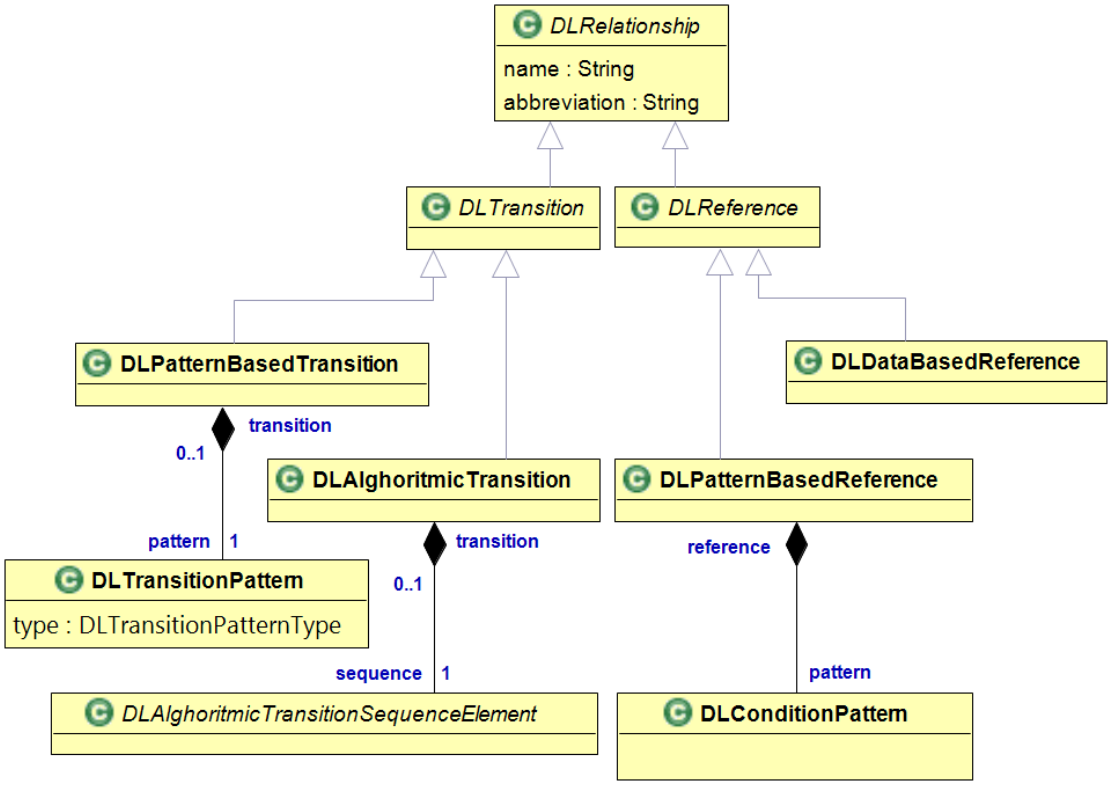

Figure 3: Meta-model details for relationships

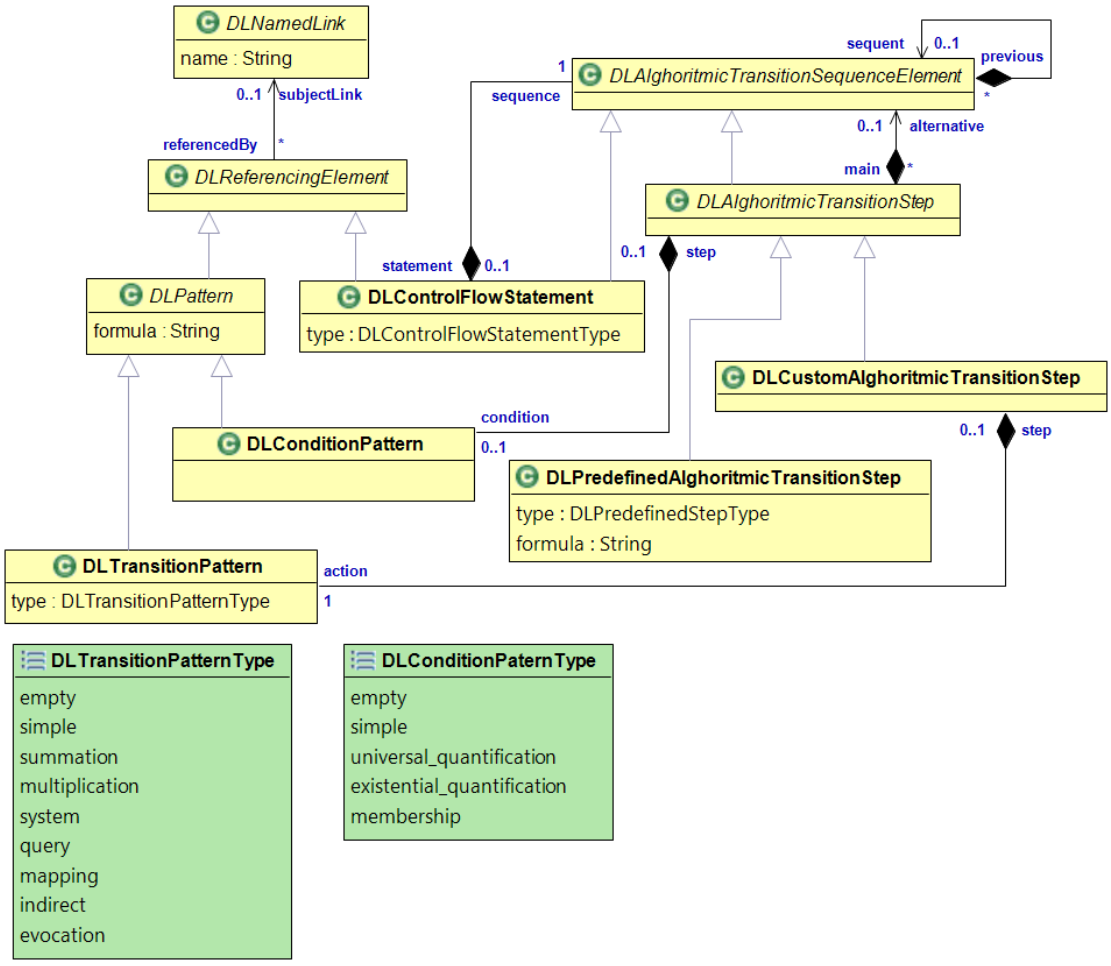

Figure 4: Meta-model details for patterns

```
<query> ::= <action> <subject> |
            <action> <subject> 'based on' <domain>
<action> ::= 'Get' | 'Transform to' | 'Create' |
             'Read' | 'Update' | 'Delete' | 'Validate'
<domain> ::= name of existing domain
<subject> ::= name of existing notion
```

Figure 5: Query syntax

2 Translational semantics rules

Rule 1. Classes and interfaces

a) An interface like "IMNotion" is created for every notion relevant for the generation of domain logic and does not reflect only a simple value. An additional "MNotion" class implementing the "IMNotion" interface is created if the notion is not of type "template". These elements extend the "IDLCLass" interface and the "DLClass" abstract class, respectively.

b) An "MSNotion" class that extends the "DLHelper" abstract class is created for notions of type "identity" or "assigned_role", or of type "inferred_role" with assigned identity conditions, or of type "template" from which inherit directly or indirectly notions of type "identity" or "assigned_role".

c) An enumeration type "MNotionValueEnum" is created for properties of data type "enumeration".

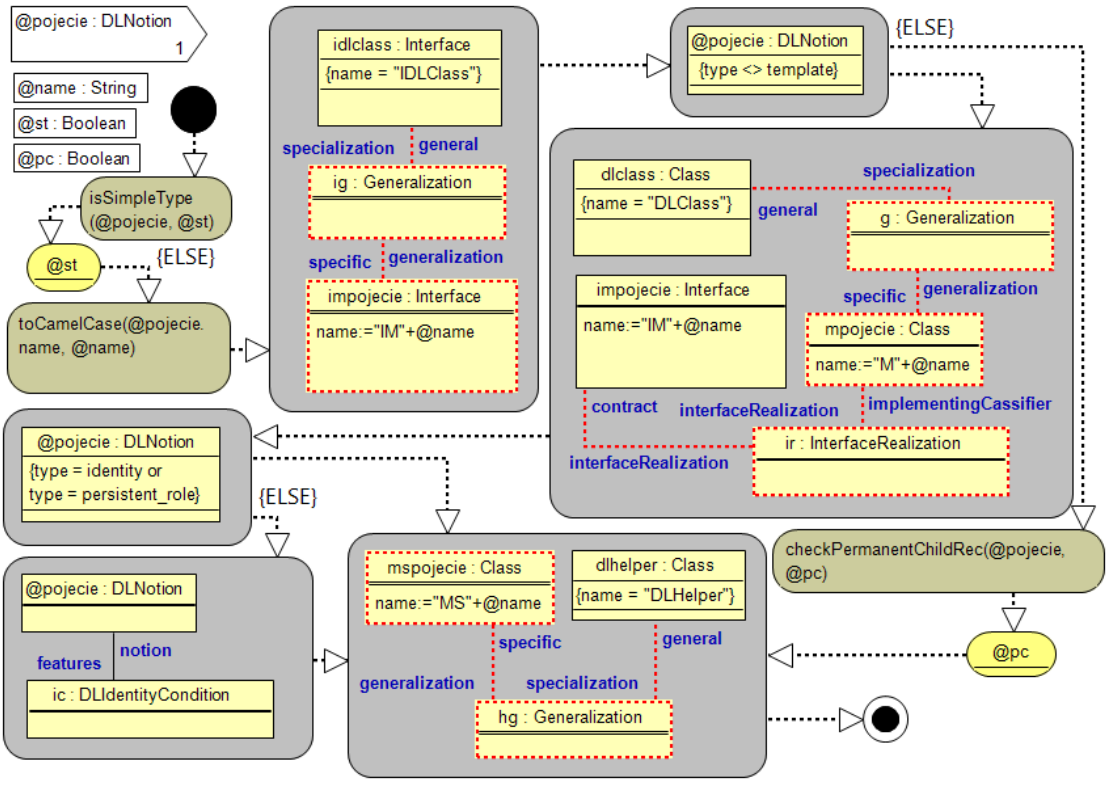

Figure 6: Formalization of transformation rule 1

Rule 2. Inheritance

If a given notion has an inheritance condition that refers to a single other notion, then the interface created for the first notion extends the interface for the second one.

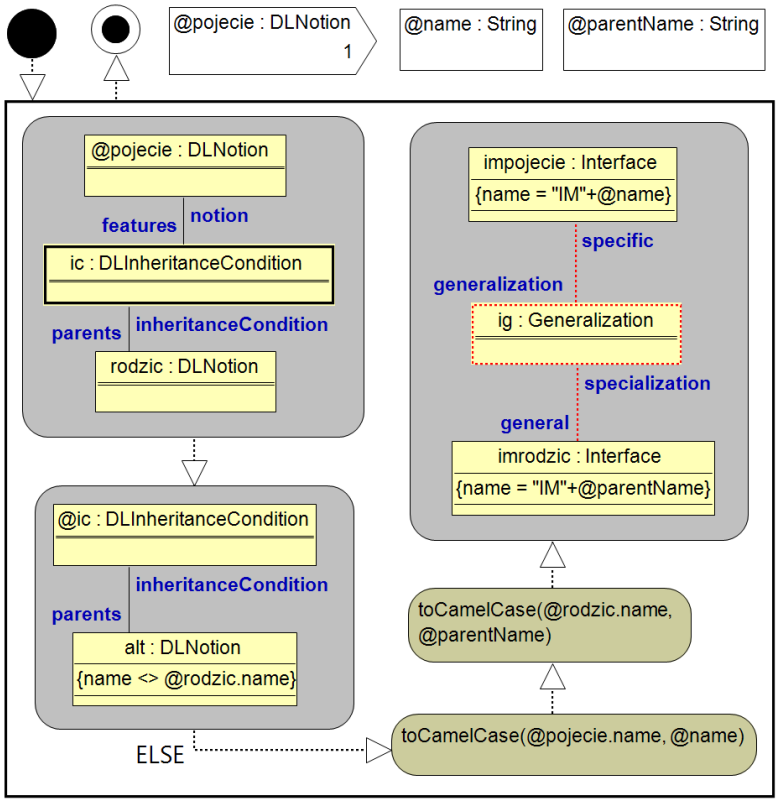

Figure 7: Formalization of transformation rule 2

Rule 3. Attribute-based fields

An additional "value" attribute, typed according to the value of the "data-Type" parameter, is created in every class created from a property. Moreover, "get" and "set" operations for the "value" attribute are declared in the interfaces created from properties. Method contents for these operations are placed in the class mentioned above. For properties of type "template" that do not have respective classes created, only the interface operations are created.

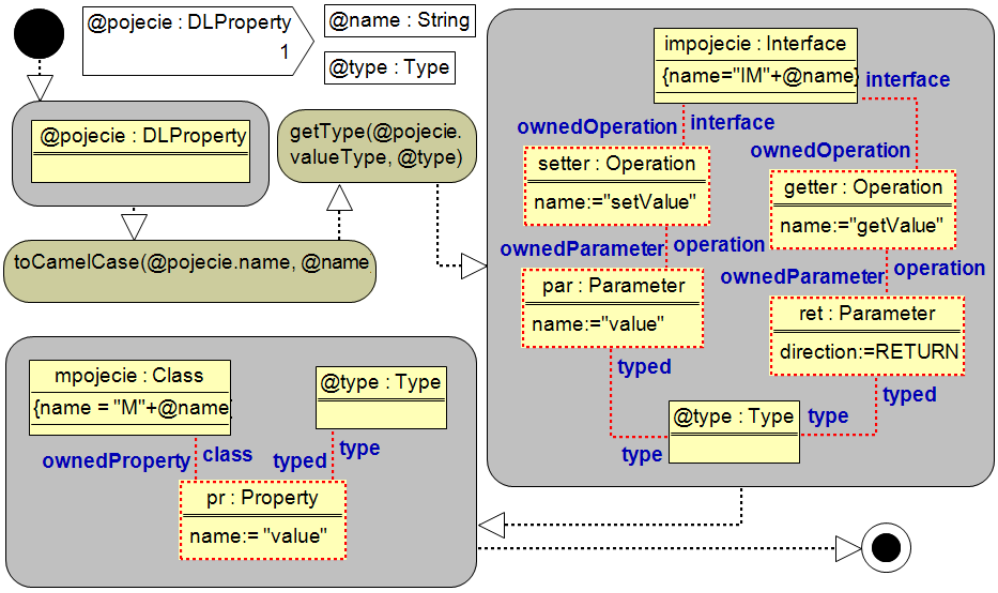

Figure 8: Formalization of transformation rule 3

Rule 4. Reference-based fields

a) For every non-template notion referenced by a "provided" attribute link contained in another notion, an attribute related to the first notion is created in the class created from the second notion. Appropriate access operations are declared in the related interface and their methods – in the above class.

b) For template notions as above, only interface operations are created. If the attribute link is "derived" also methods are created but without contents.

c) The above rules also pertain to any "identity" notion that inherits directly or indirectly from such notion or to any notion that inherits directly or indirectly from such notion of "template" type. In this case, attribute links are treated as if they were contained within the inheriting notion.

d) Getters, setters, and attribute declarations are typed with relevant "IMNotion" interfaces. The exception to this is associated with simple value properties. In this case, getter, setter, and attribute types depend on the value types of these properties.

e) For every data-based reference associated with a notion, appropriate attributes and access methods are created in the respective generated interface and class. They are typed depending on the number of elements participating in the reference. For binary references, the types are based on the other notion type. For n-ary (n¿2) references, the types are based on the dedicated class created from the reference (see Rule 9).

f) For template notions as in e) above, only interface operations are created.

g) For references as in e) above and with "multiple" or "ordered_multiple" participations, sets or lists of appropriate types are used respectively.

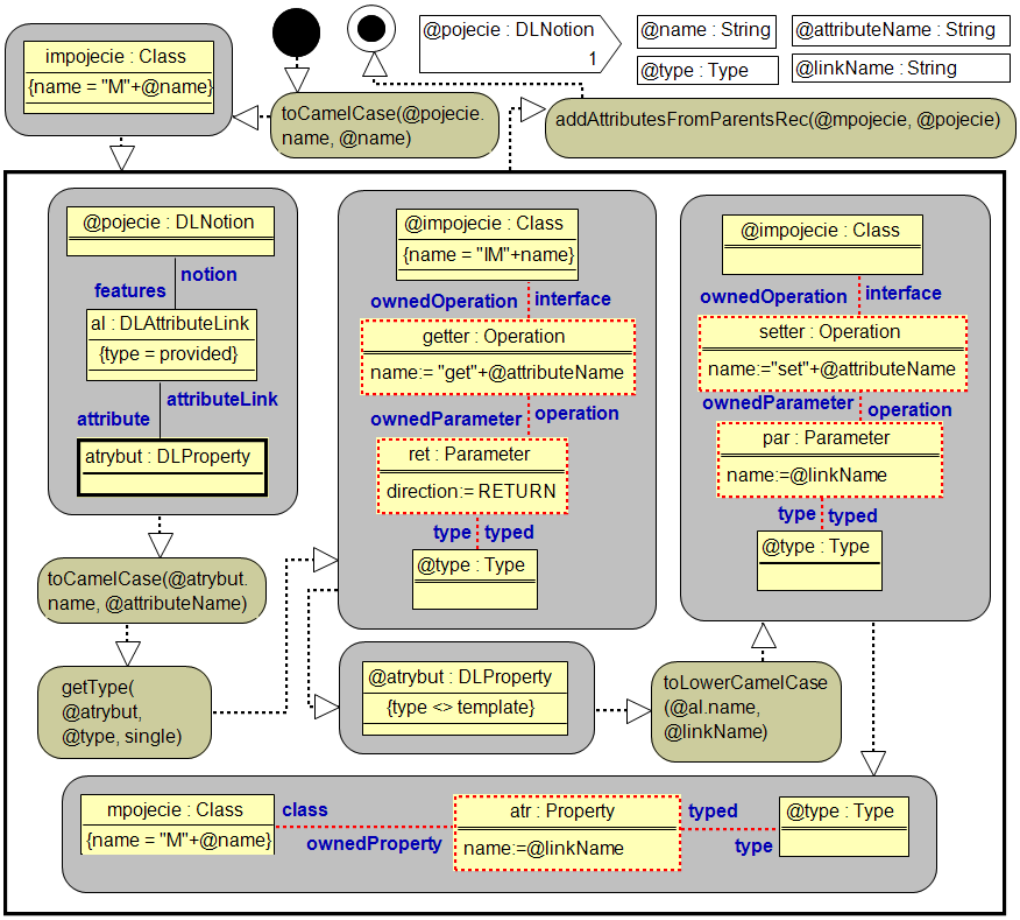

Figure 9: Formalization of transformation rule 4 – part 1

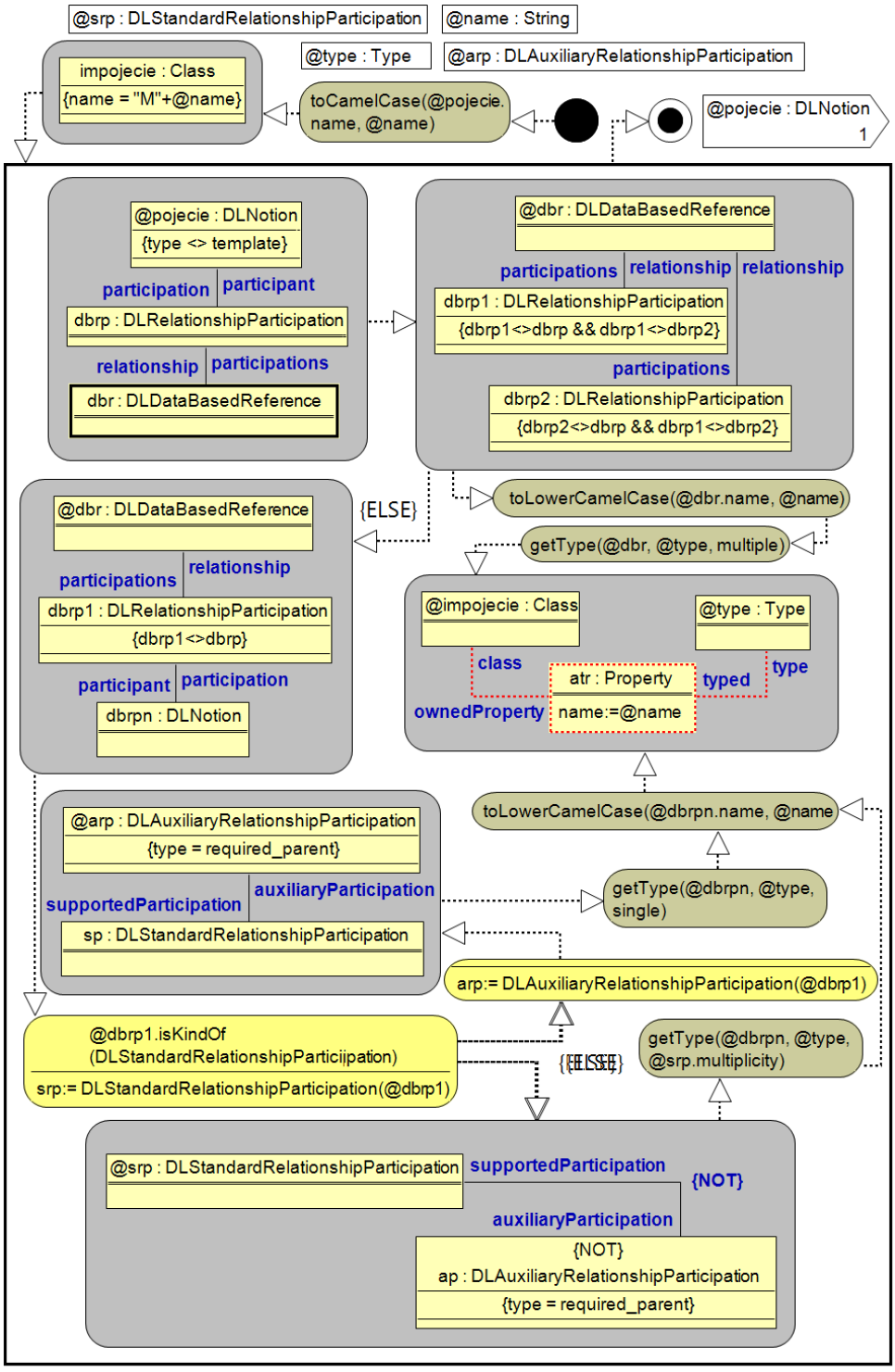

Figure 10: Formalization of transformation rule 4 – part 2

Rule 5. Constructors

a) For classes derived from "identity" type notions, constructors are created with parameters depending on the respective attributes (including those inherited). For property notions, this also includes parameters based on the value kinds represented by these notions.

b) For "assigned_role" and "inferred_role" notions, at least two constructors are created. The first one accepts the interface derived from the base notion (representatives of this notion can have the role assigned to). The second one also accepts parameters based on the attributes assigned to the respective notion (and inherited from "template" notions). For property notions, additional parameters are created that are based on the represented value kind. When a given notion can be assigned to representatives of more than one notions, additional constructor variants are created.

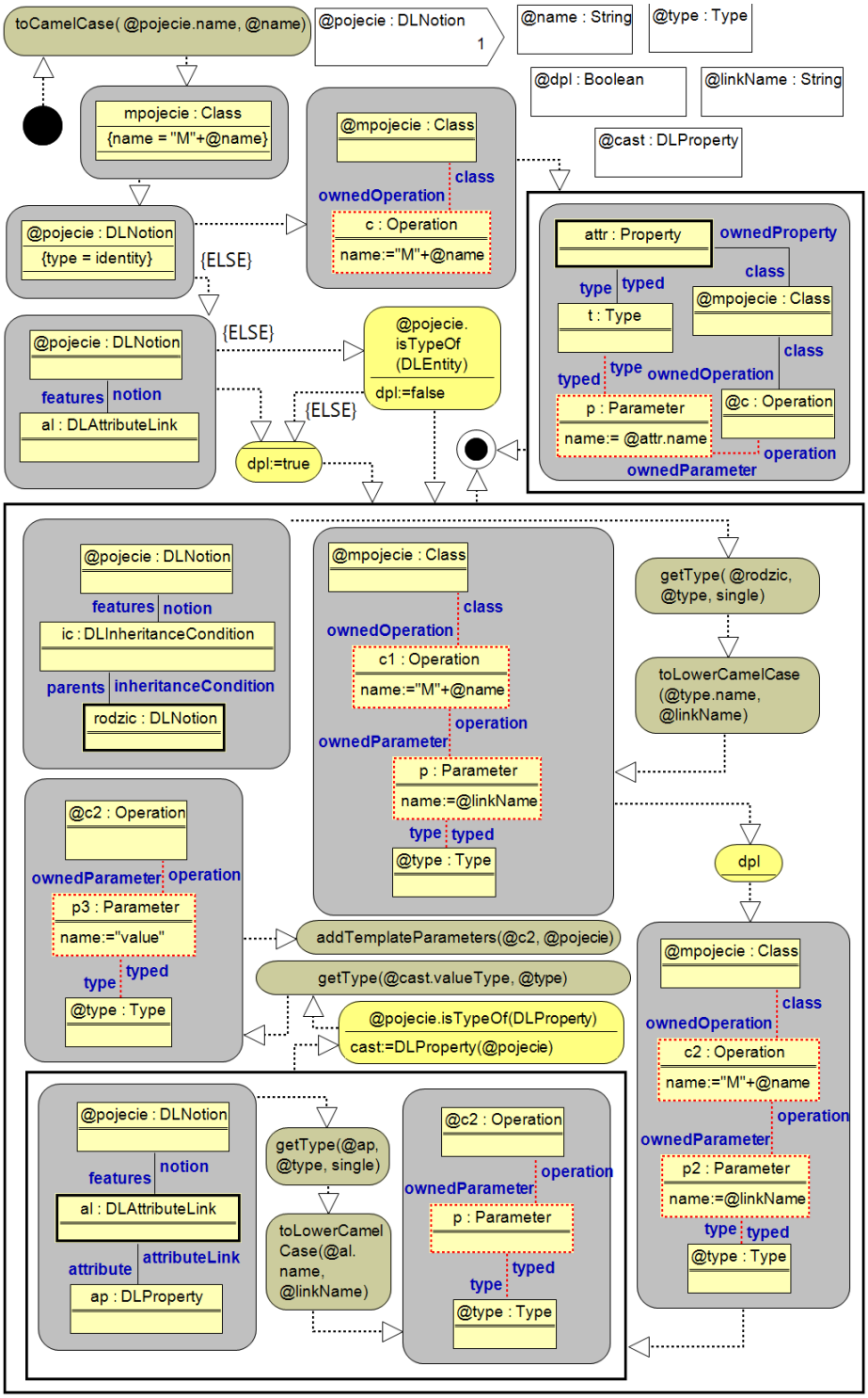

Figure 11: Formalization of transformation rule 5

Rule 6. Validation

For non-template notions with validity conditions defined directly or through inheritance, a "validate()" operation is created in the derived interface. An analogous method is created in the implementation class. The method contains a conjunction of all the validation rules included in the validity condition for the given notion, in the inherited notions, and their attributes.

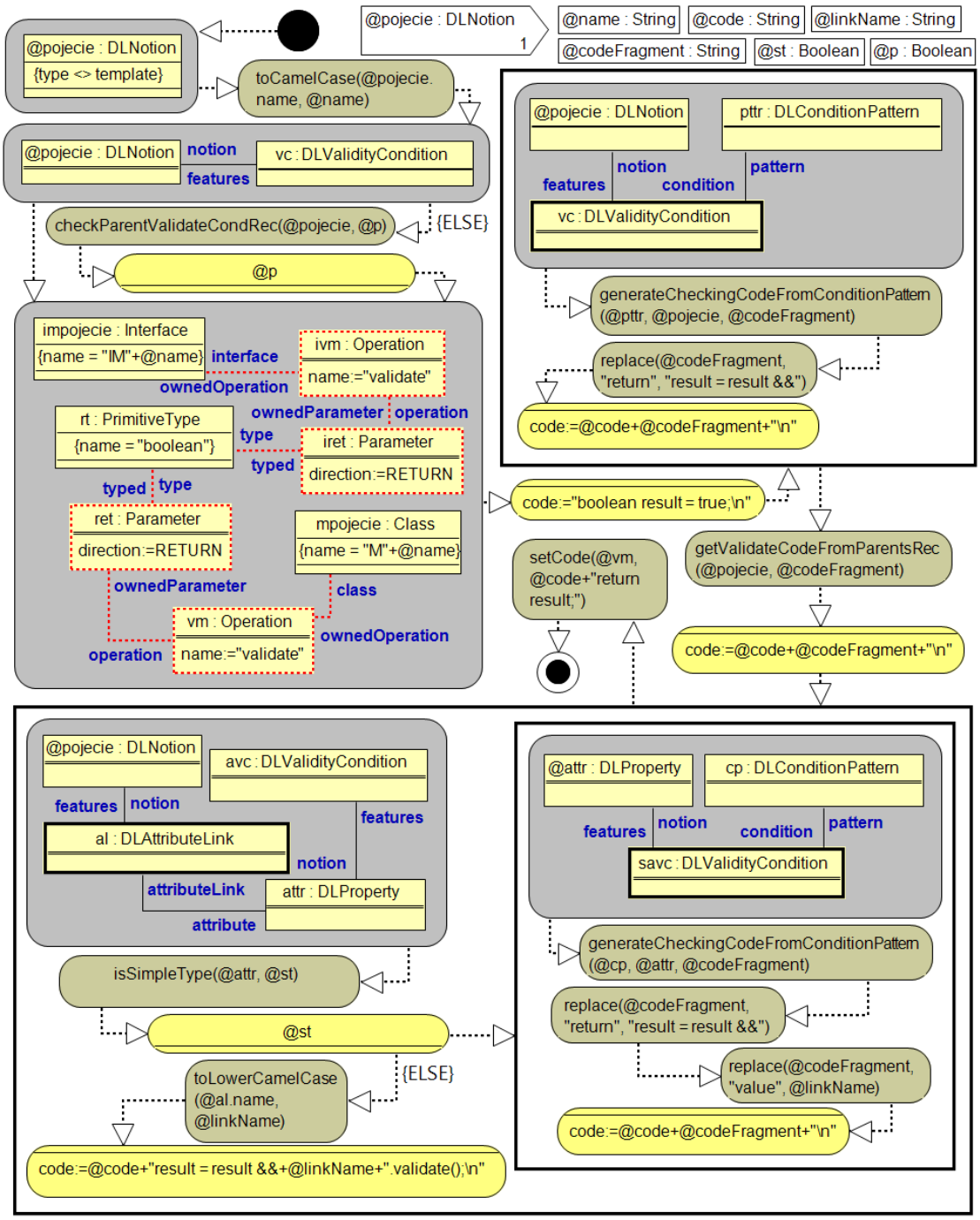

Figure 12: Formalization of transformation rule 6

Rule 7. Eligibility checks

For non-template notions with identity conditions defined directly or through inheritance, static "checkNotion()" methods are created for every constructor (with the same parameters). These methods are placed in the derived "MSNotion" class. The method contains a conjunction of all the rules included in the identity condition for the given notion, in the inherited notions, and their attributes.

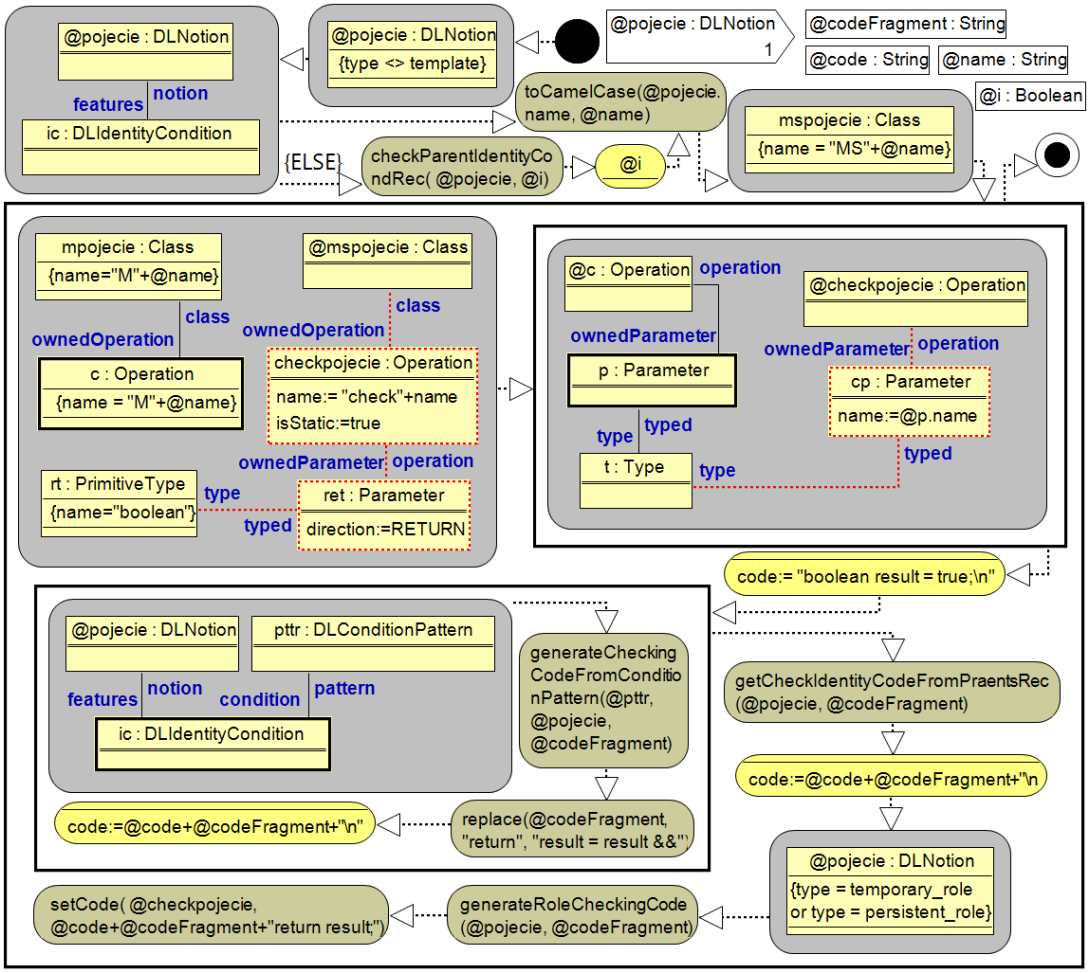

Figure 13: Formalization of transformation rule 7

Rule 8. RUD operations

For every notion, an "update()" and a "delete()" operation is created in the respective interface. For every non-template notion, analogous methods are created in the implementing class. Additionally, a "getNotion()" method in the "MSNotion" class is created for every non-template notion or template notion from which inherit "identity" or "assigned_role" notions.

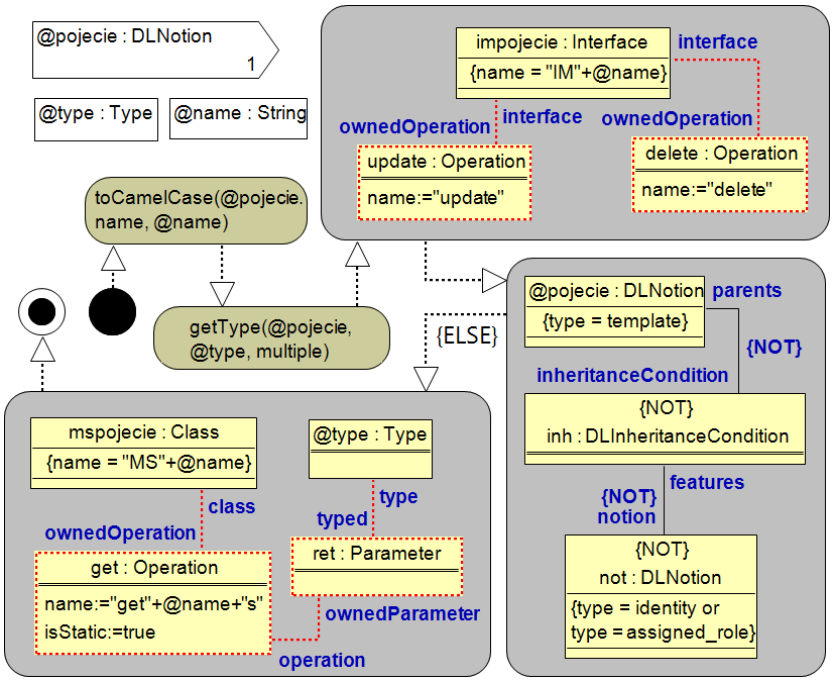

Figure 14: Formalization of transformation rule 8

Rule 9. Relation-based retrieval

a) A class like "MSRelation" is created for every relation relevant for the generation of domain logic. Static methods based on relation participations are created in this class. One of the participations is used to create the return type for a given method, and the others – to create the method's parameters. These methods are created only for 1) standard participations with the direction "target" or "undefined" not associated with a "role_attribution" auxiliary participation, and 2) "role_attribution" auxiliary participations.

b) For the participations used for generating parameters associated with a "required_parent" auxiliary participation, the appropriate method's parameters are substituted by a single parameter derived from this auxiliary participation.

c) Returned values are of primitive types or typed as "IMNotion" interfaces. For "multiple" or "ordered_multiple" participations, sets or lists of appropriate types are used respectively.

d) An "MRelation" class is created for every non-binary data-based reference. It contains attributes based on all the participations of the given reference that are not associated with a "required_parent" auxiliary participation, or based on "required_parent" auxiliary participations. Appropriate constructors and access methods are also created.

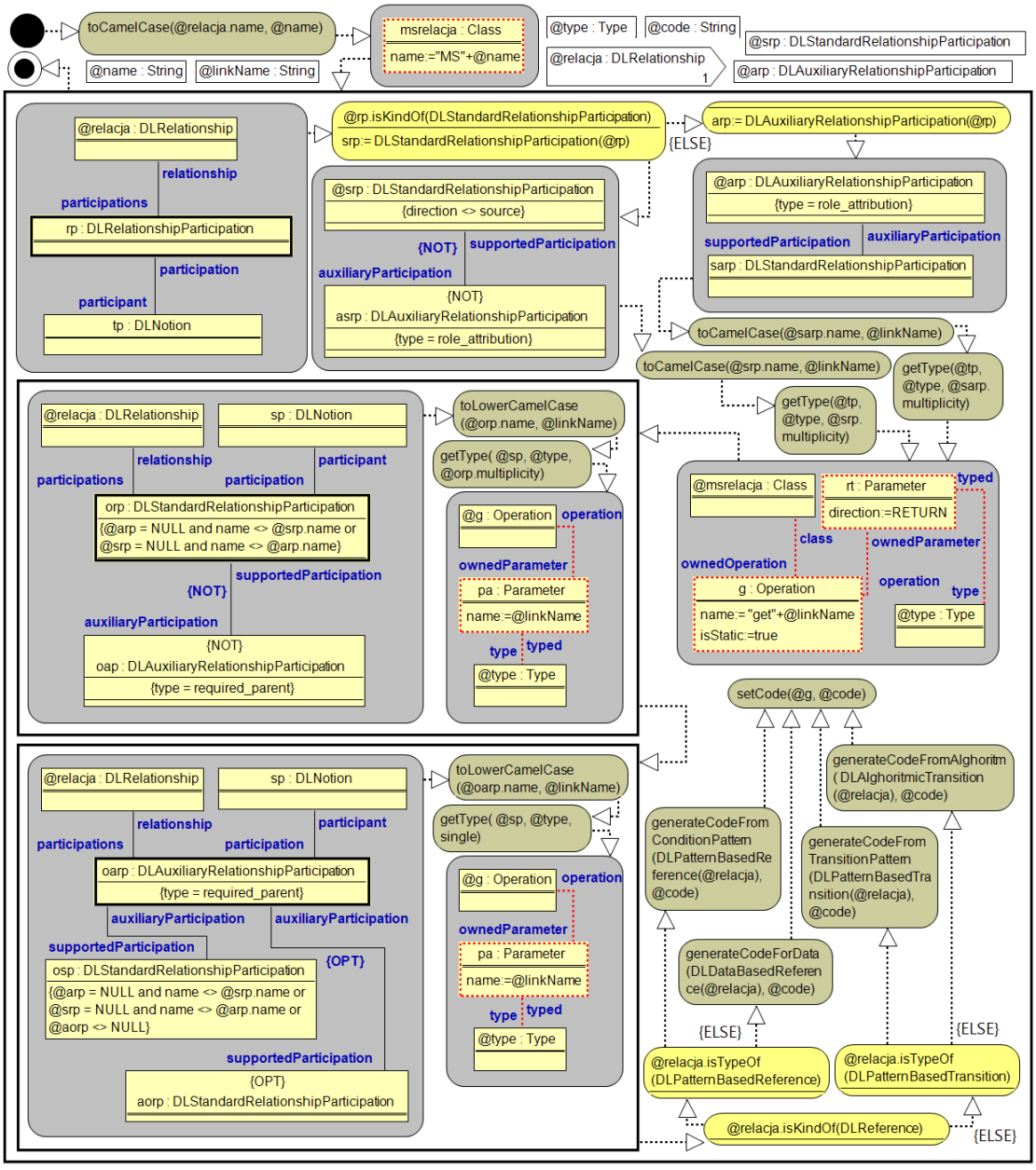

Figure 15: Formalization of transformation rule 9 – part 1

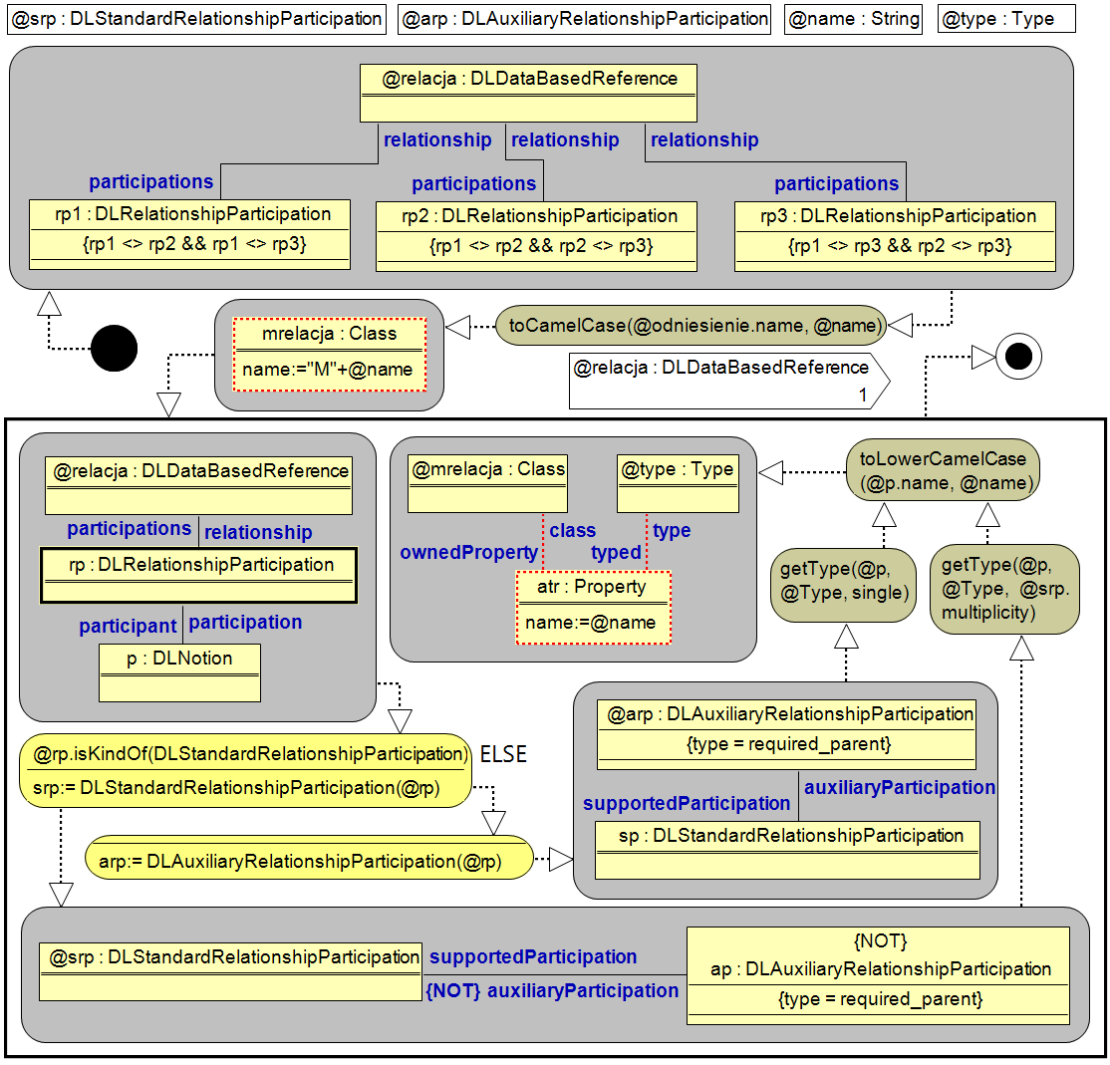

Figure 16: Formalization of transformation rule 9 – part 2

Rule 10. Relation checking

For every reference, a "checkRelation()" method is created in the respective derived class. It accepts parameters representing all the participating notions and returns a boolean value. For the participations that lead to these notions, associated with a "required_parent" auxiliary participation, the method's parameters are substituted by a single parameter derived from this auxiliary participation. For a "multiple" or "ordered_multiple" participation indicated as a subject link of a condition pattern, sets or lists of appropriate types are used, respectively.

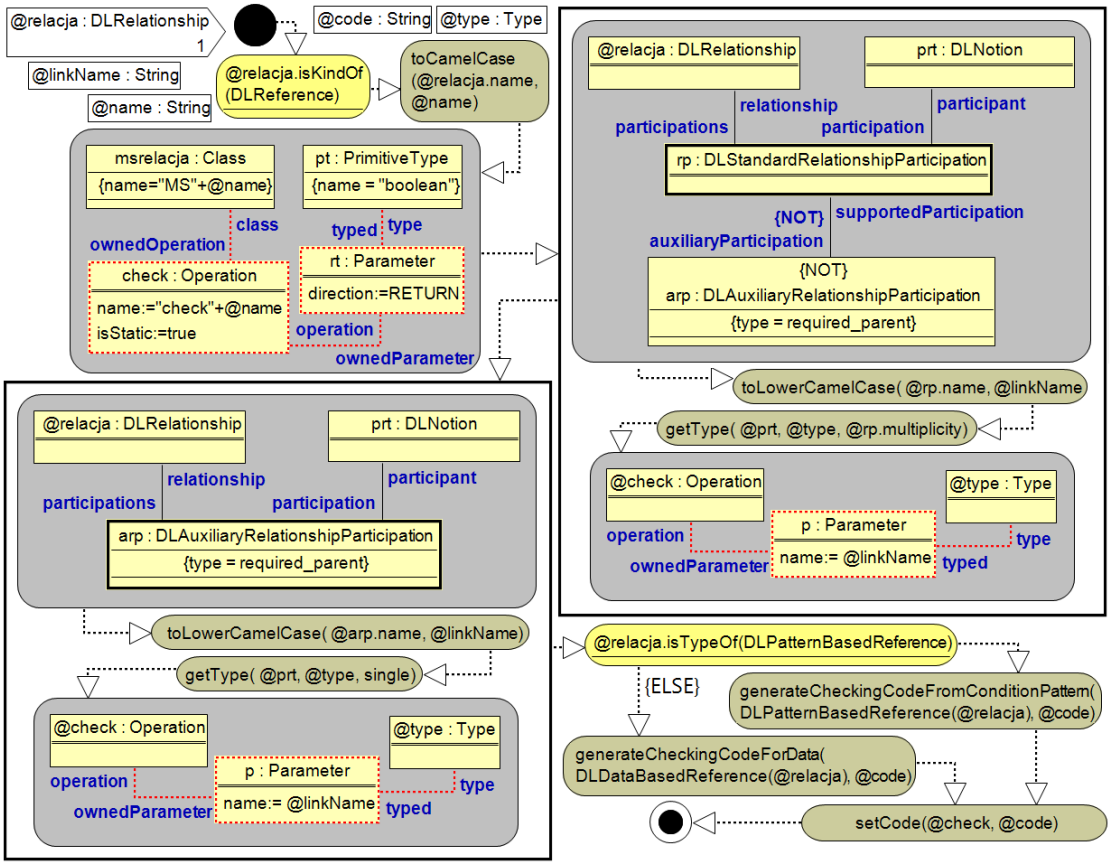

Figure 17: Formalization of transformation rule 10

Rule 11. Queries

a) For every "get" query that requires more than one method invocation, a static method is created. It is created in the class derived from the element being the subject of the query. This method accepts parameters required to solve the query according to the inference engine.

b) The contents of this method is generated as a sequence of method calls, getter calls and type casting operations. It is done according to the sequence of rules determined by the inference engine. Each of the call/cast results is assigned to a local variable and used in other calls/casts.

c) If any of the method invocations was created based on a "MULTIPLE_USE" inference rule, an additional method is created in the class derived from the element referenced by this rule. This method contains an iterator with a call/cast and is called in the appropriate place in the sequence defined in b).

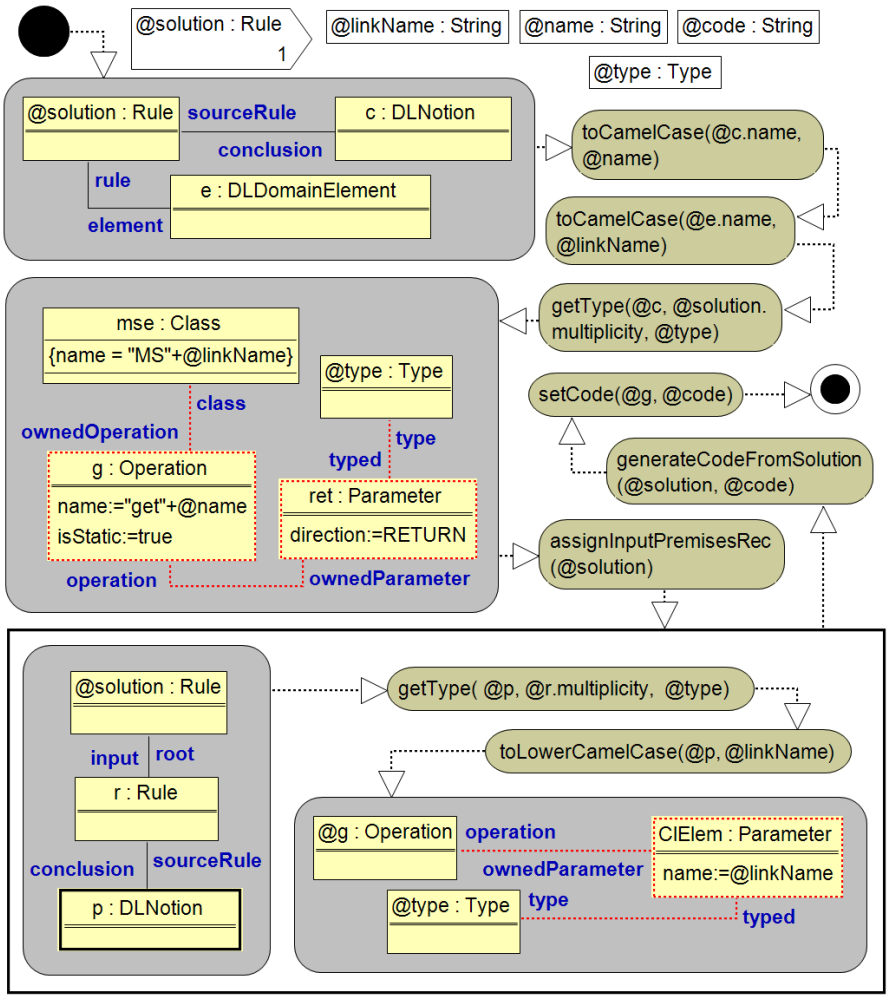

Figure 18: Formalization of transformation rule 11

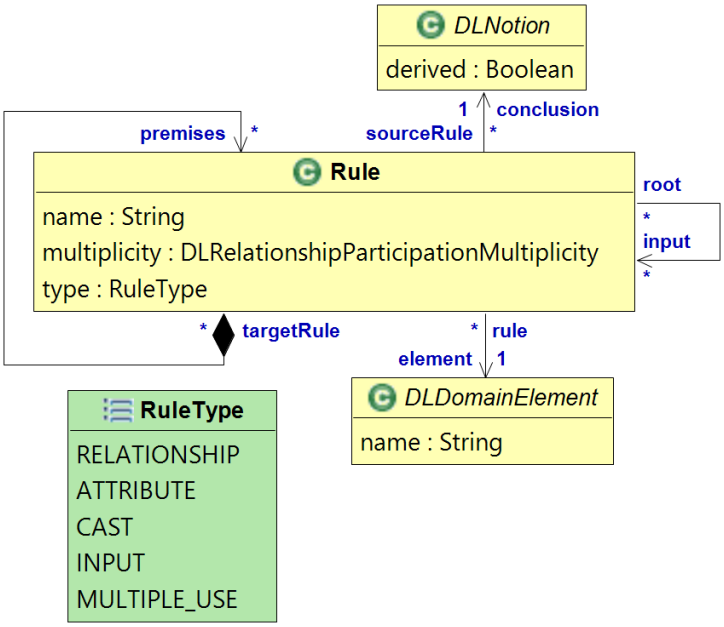

Figure 19: Inference rule meta-model

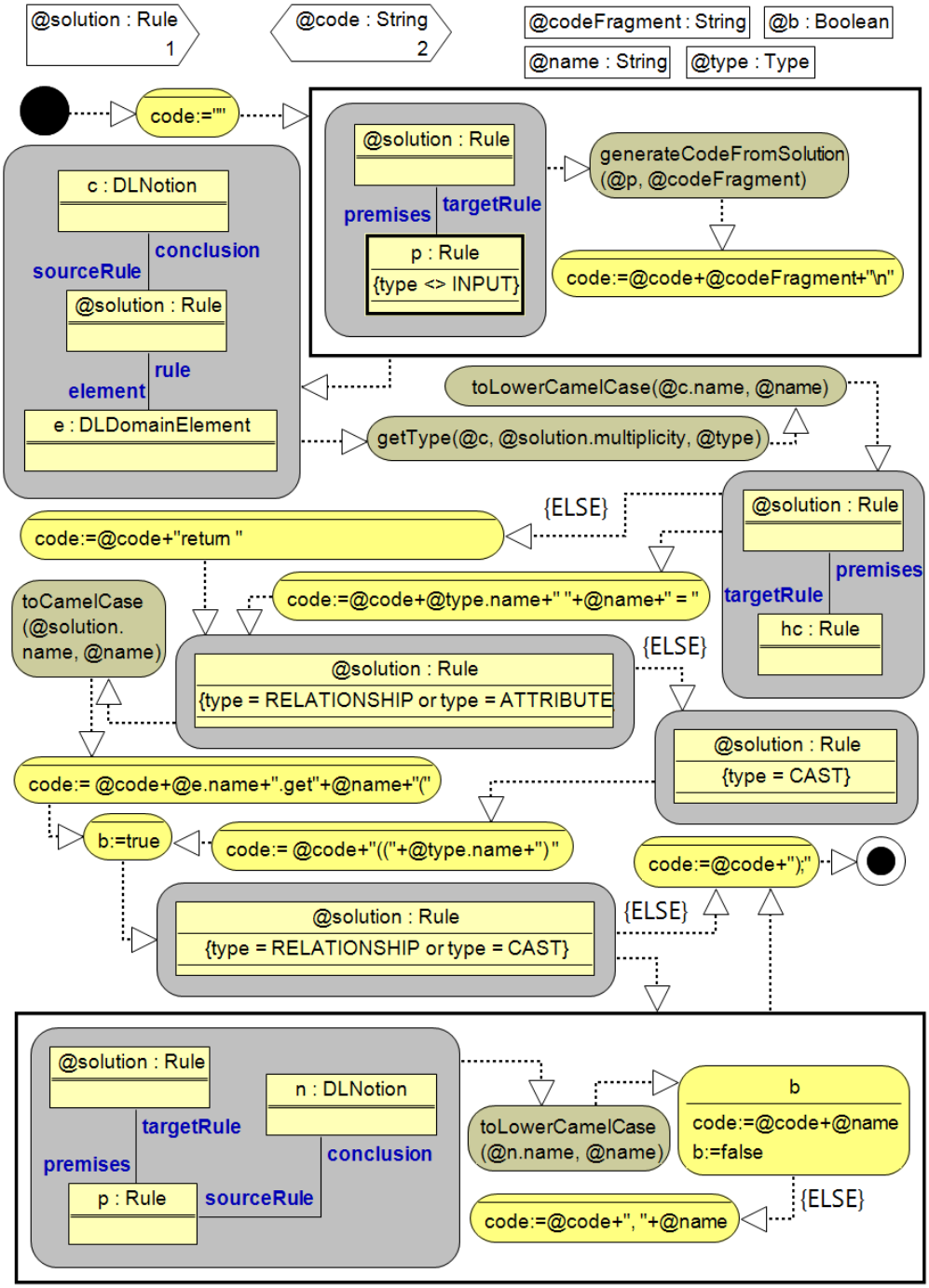

Figure 20: Algorithm for generating the method body in Rule 11

Rule 12. Derived attributes

For every getter created from a "derived" attribute link (see Rule 3), appropriate method body contents are generated, analogous to code generated with rule 11. The necessary "get" query used the relevant attribute as its subject and its containing notion – as the only premiss.

Rule 13. Condition patterns – checking

Translation of condition patterns into code depends on the contained formula types.

a) For "simple" patterns the computed formula result is returned with a "return" statement.

b) For "universal quantification" and "existential quantification" patterns an appropriate iteration code is generated. It iterates over a set of elements indicated by the subject link and checks the pattern formula (the "$" signs in the formula are substituted by the iteration variable). For the "universal quantification" pattern, the returned value is a conjunction, and for the "existential quantification" – an alternative of these formula checks.

c) For "membership" patterns, the generated code checks if the given object has a given role in one of the three ways. Assigned roles are checked using the "getDLRoles()" operation of the "IDLClass" interface. Inferred roles are checked using the "checkNotion()" method of the appropriate "MSNotion" class. In the remaining cases, the standard "instanceof" Java operation is used.

Rule 14. Condition patterns – filtering

The code that seeks for elements based on condition patterns contains appropriate iteration code. It iterates over a specified set of elements and invokes the respective condition checking method (see rule 13). If relevant participation to the sought element has the "single" multiplicity, the first found element is returned, and in other cases – a list of all found elements is returned.

Rule 15. Transition patterns

For "simple", "system", "summation" and "multiplication" transition patterns, the contained formulas are transformed into appropriate code.

a) For "simple" and "system" patterns, contained formulas need to be first solved for the sought variables. The result is returned with a "return" statement.

b) For "summation" and "multiplication" patterns, an appropriate iteration code is generated. It iterates over a set of elements indicated by the subject link and computes the pattern formula (the "$" signs in the formula are substituted by the iteration variable). For the "summation" pattern, the returned value is a sum, and for the "multiplication" – a product of computation results.

c) "Query" patterns are treated as "get" queries (see Rule 11).

d) For "indirect" patterns without a subject, the generated code creates a new object. The contained formulas are used to initialise the object's fields.

e) For "indirect" patterns with a subject, the contained formulas are used to update the object's fields derived from the subject.

f) For "invocation" patterns, the contained formula is copied directly into the generated code.

g) For "mapping" patterns, a set of "if" statements are generated. The first element of each mapping is used in the "if" condition as a "simple" condition pattern (see Rule 14a). The second element is used to generate the 'if" contents as a "simple" transition pattern (see a) above).

Rule 16. Auxiliary code generation

a) If a given rule's premiss is a primitive, the beginning of the code generated using this rule should be appended. This code obtains the value for the primitive and assigns to a variable.

b) If the given pattern-based reference does not indicate a specific set of elements to which it should be applied, the beginning of the code generated using this rule should be appended. This code should obtain all the elements of the appropriate type.

3 Case study

4 RSL model

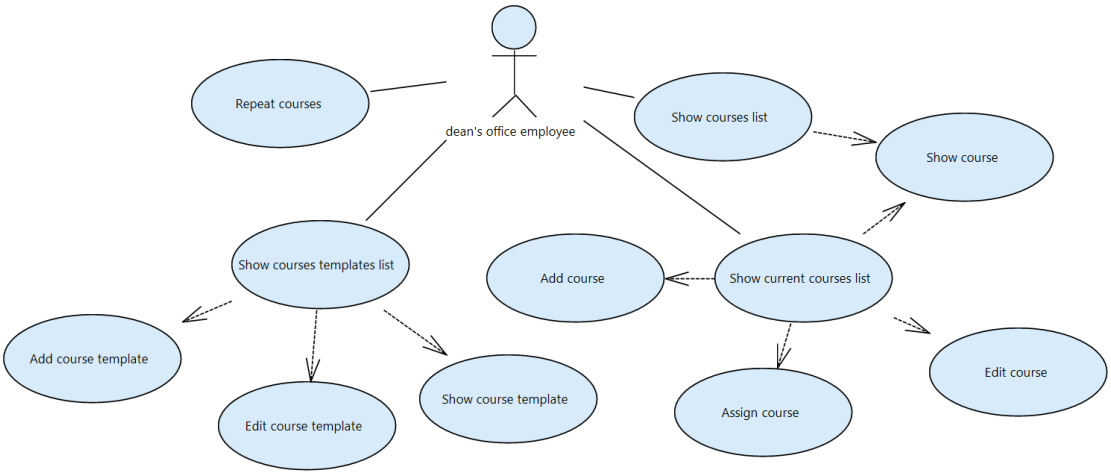

Figure 21: Course management related use case model

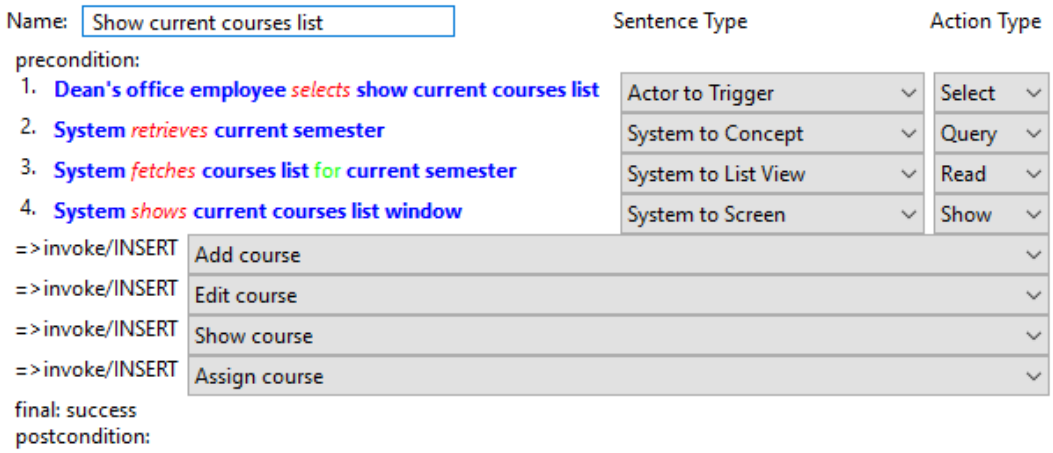

Figure 22: Scenario of 'Show current courses list' use case

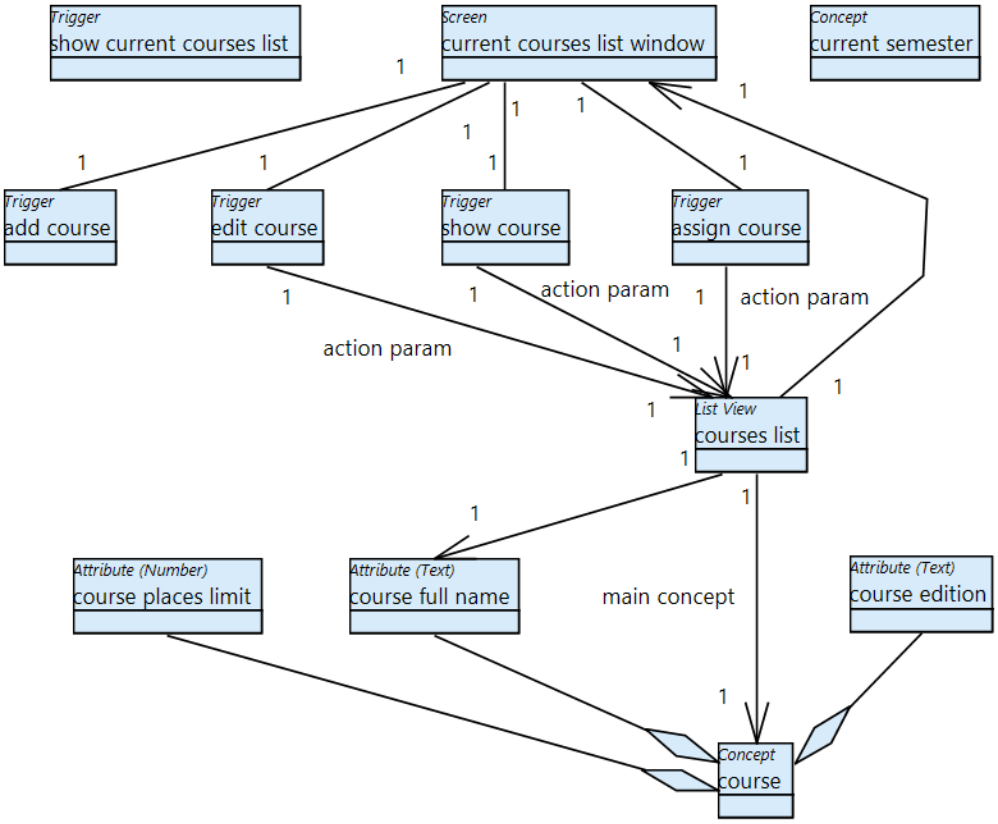

Figure 23: RSL domain model for 'Show current courses list' use case

Name: Repeat courses

precondition:

1. **Dean's office employee** *selects* **repeat courses**
2. **System** *retrieves* **previous semester**
3. **System** *fetches* **courses list** for **previous semester**
4. **System** *shows* **courses to repeat selection window**
5. **Dean's office employee** *selects* **set of courses** from **courses list**
6. **Dean's office employee** *selects* **create courses repetitions**
7. **System** *generates* **course repetitions list** for **set of courses**
8. **System** *saves* **course repetitions list**
9. **System** *shows* **course repetitions created message**

final: success

postcondition:

Sentence Type	Action Type
Actor to Trigger	Select
System to Concept	Query
System to List View	Read
System to Screen	Show
Actor to List View	n/a
Actor to Trigger	Select
System to List View	Query
System to List View	Create
System to Message	Show

Figure 24: Scenario of 'Repeat courses' use case

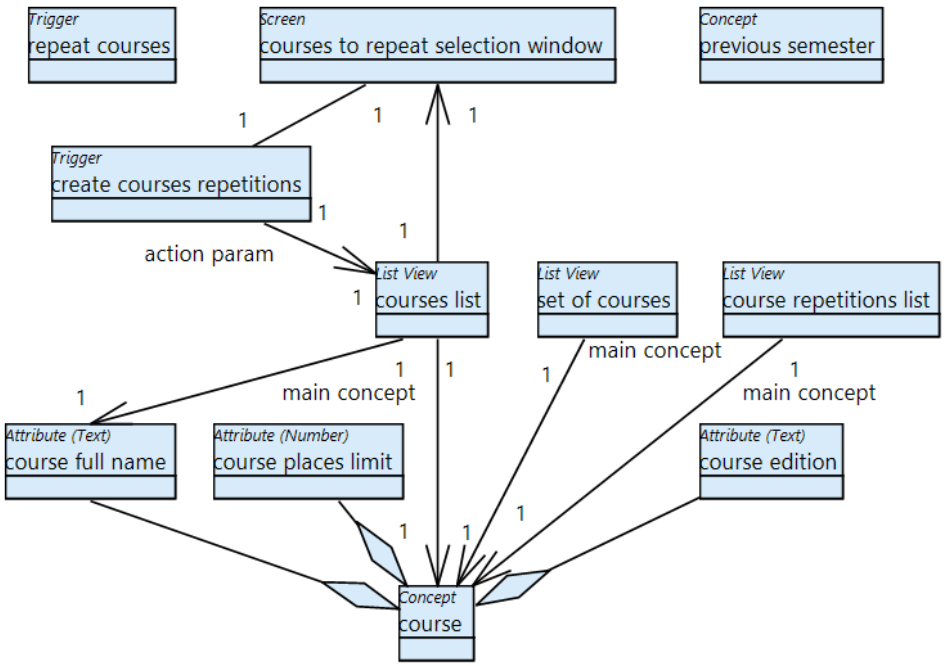

Figure 25: RSL domain model for 'Repeat courses' use case

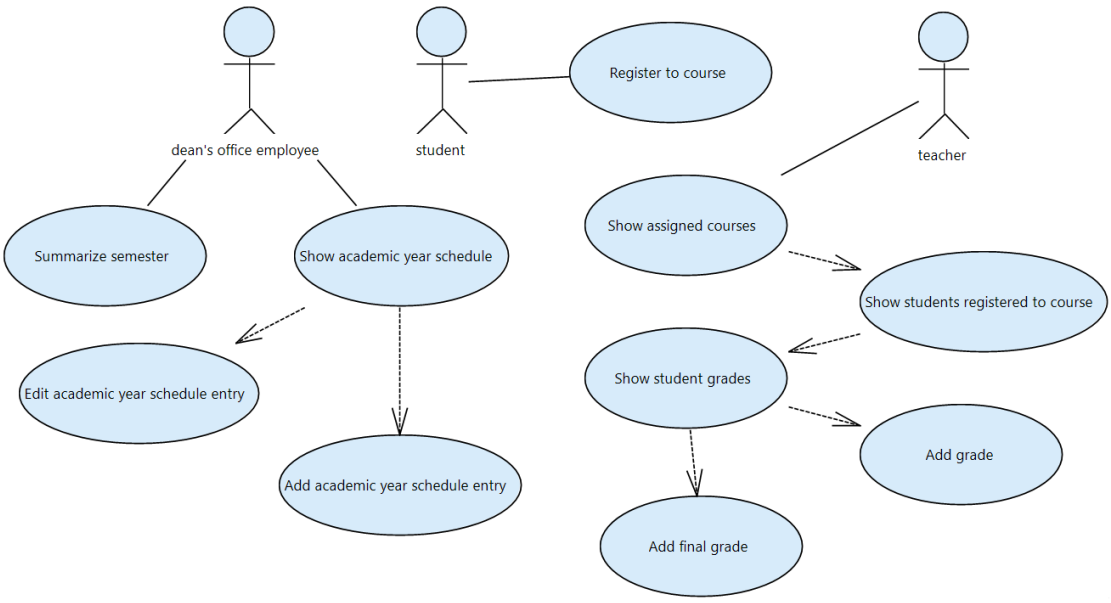

Figure 26: Course registration and grading related use case model

Name: Add final grade

precondition: <param>student</param><param>course</param>

	Sentence Type	Action Type
1. Teacher *selects* add final grade	Actor to Trigger	Select
2. System *fetches* partial grades list	System to List View	Query
3. System *computes* weighted average grade data	System to Simple View	Query
4. System *shows* add final grade form	System to Screen	Show
5. Teacher *enters* final grade data	Actor to Simple View	n/a
6. Teacher *selects* save final grade	Actor to Trigger	Select
7. System *saves* final grade data	System to Simple View	Create
8. System *shows* final grade added message	System to Message	Show
9. System *closes* add final grade form	System to Screen	Close

final: success

postcondition:

Figure 27: Scenario of 'Add final grade' use case

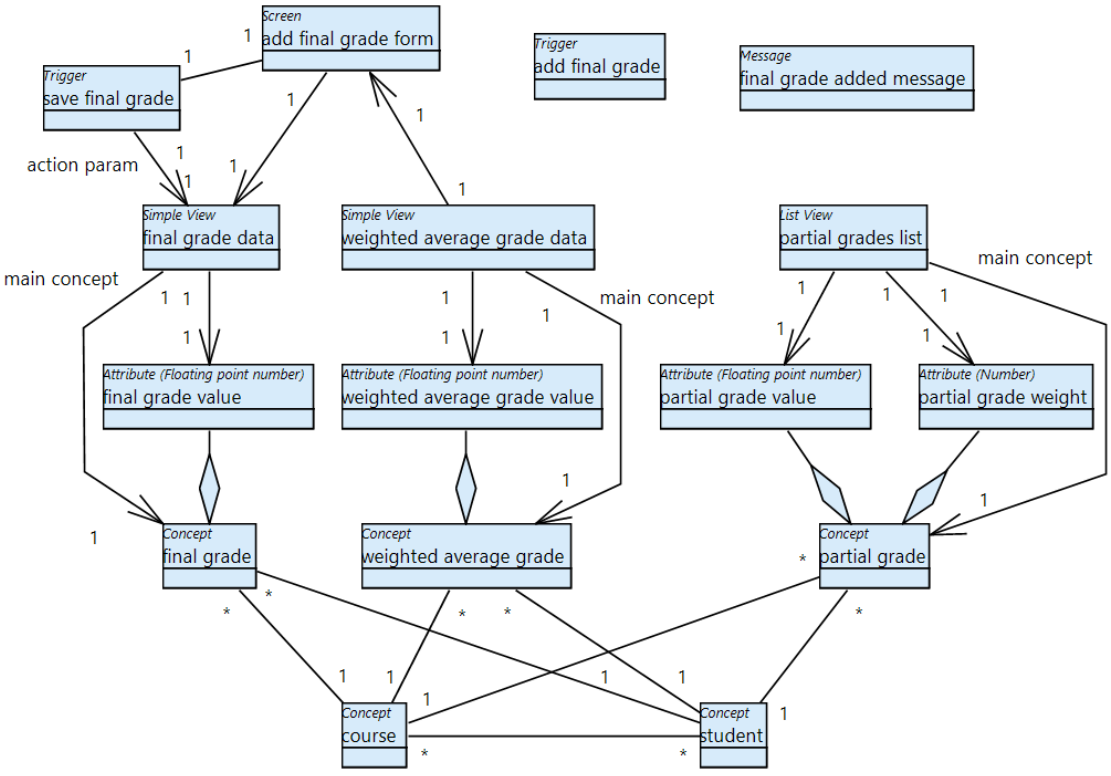

Figure 28: RSL domain mode for 'Add final grade' use case

Name: Summarize semester

precondition:

		Sentence Type	Action Type
1.	**Dean's office employee *selects* summarize semester**	Actor to Trigger	Select
2.	**System *fetches* students list**	System to List View	Read
3.	**System *prepares* semester summarization data**	System to List View	Query
4.	**System *shows* semester summarization window**	System to Screen	Show
5.	**Dean's office employee *selects* accept summarization**	Actor to Trigger	Select
6.	**System *closes* semester summarization window**	System to Screen	Close
7.	**System *realizes* semester summarization data**	System to List View	Update
8.	**System *shows* semester summarized message**	System to Message	Show

final: success

postcondition:

Figure 29: Scenario of 'Summarize semester' use case

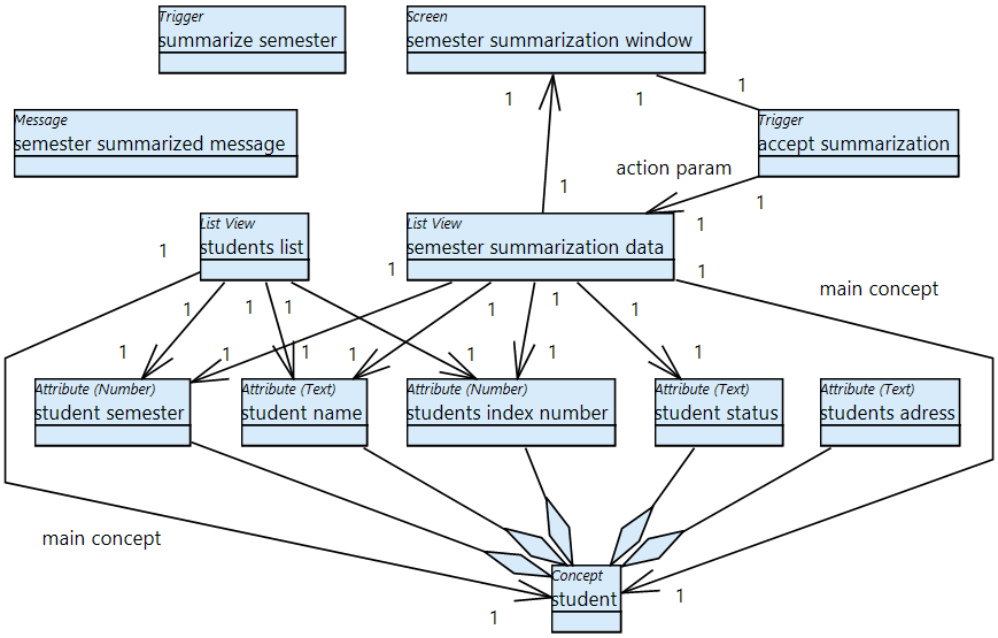

Figure 30: RSL domain model for 'Summarize semester' use case

5 RSL-DL model

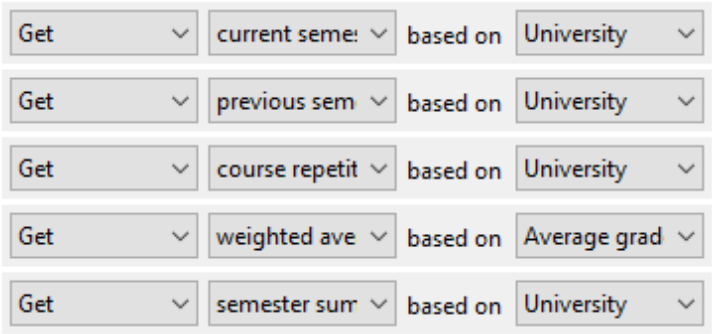

Figure 31: Queries

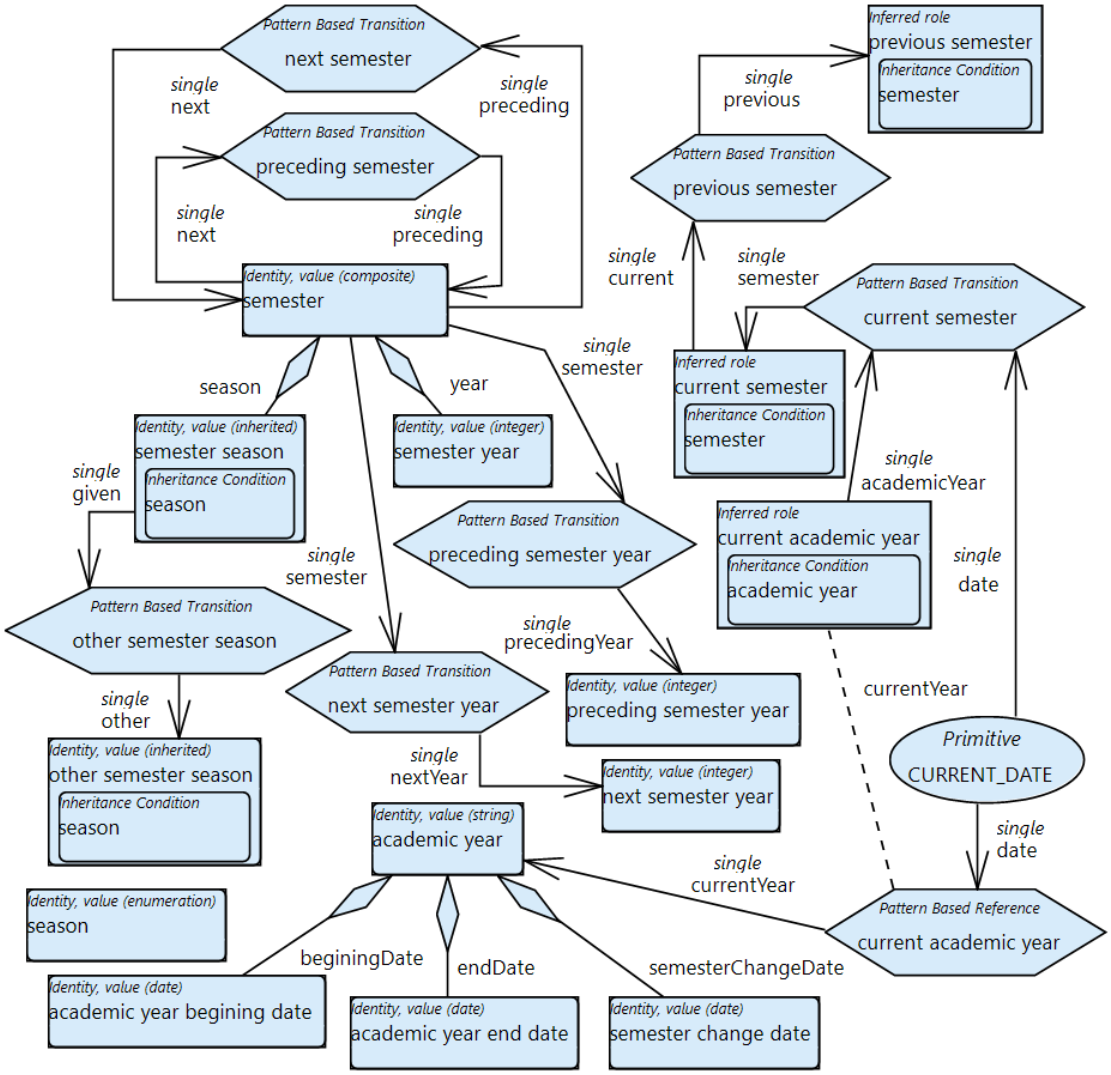

Figure 32: RDL-DL model containing knowledge about academic years and semesters

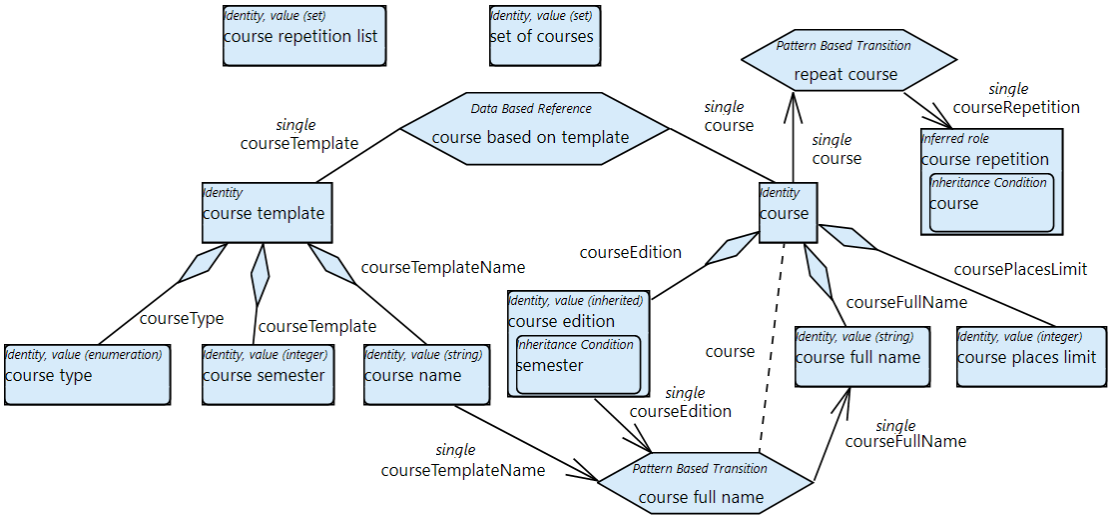

Figure 33: RDL-DL model containing knowledge about course editions

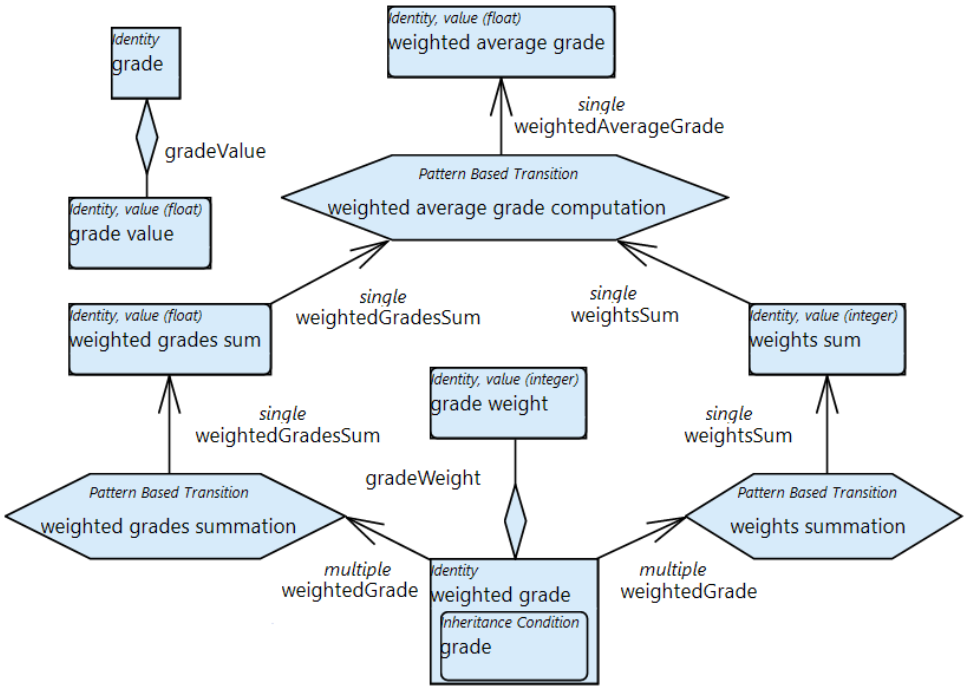

Figure 34: RDL-DL model containing knowledge about computing of weighted average grade

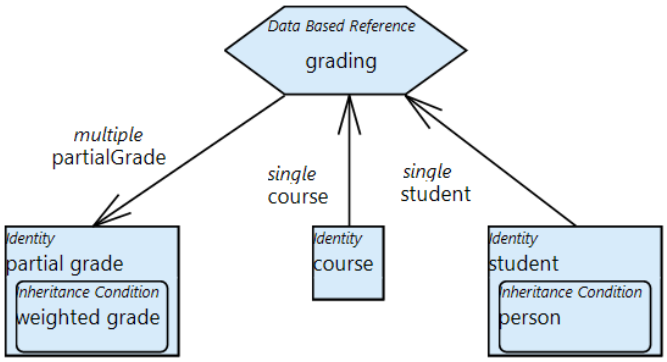

Figure 35: RDL-DL model containing knowledge about partial grades

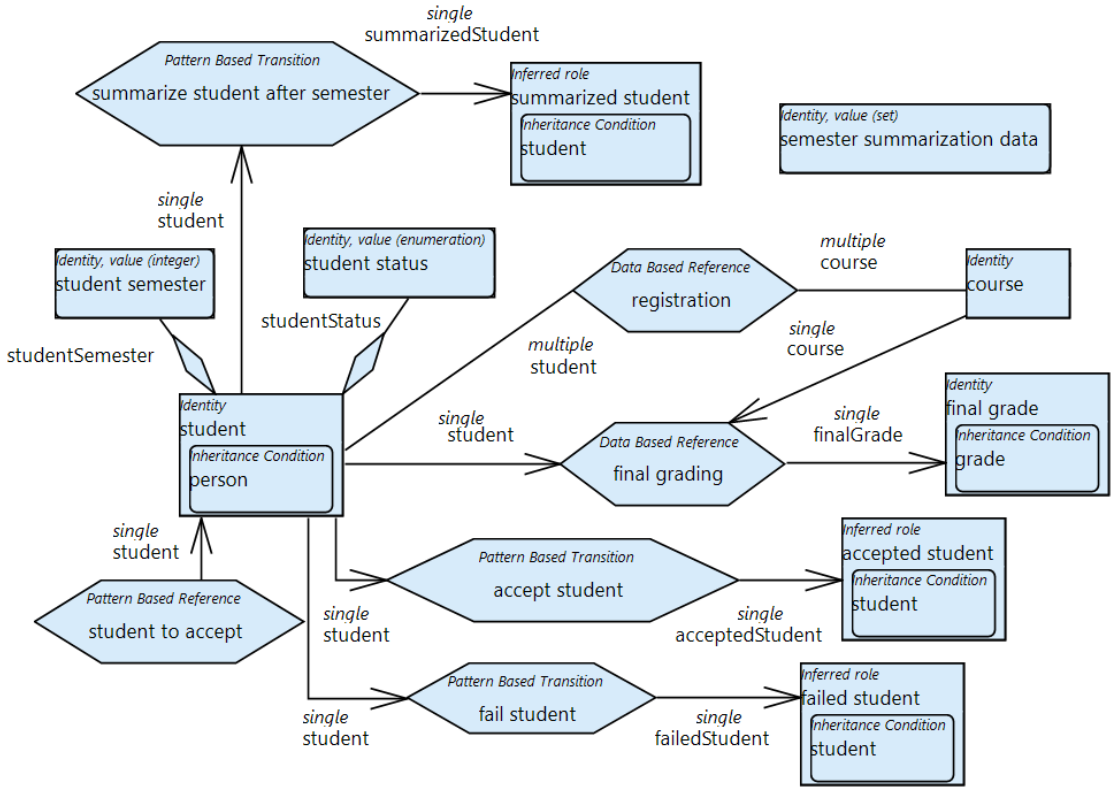

Figure 36: RDL-DL model containing knowledge about students registration for the next semester

6 Generated code

```
public class MSPreviousSemester {

    public static IMPreviousSemester getPrevious(IMCurrentSemester current){
        return MPreviousSemester(MSPrecedingSemester.getPreceding(current));
    }

    public static IMPreviousSemester getPreviousSemester(){
        IMCurrentAcademicYear currentAcademicYear = MSCurrentAcademicYear.getCurrentYear();
        IMCurrentSemester currentSemester = MSCurrentSemester.getSemester(currentAcademicYear);
        return getPrevious(currentSemester);
    }

}
```

Figure 37: 'MSPreviousSemester' class code

```
 public class MSPrecedingSemester {

     public static IMSemester getPreceding(IMSemester next) {
         IMSemester preceding = new MSemester();
         preceding.setSeason(MSOtherSemesterSeason.getOther(next.getSeason()));
         preceding.setYear(MSPrecedingSemesterYear.getPrecedingYear(next));
         return preeceding;
     }

 }
```

Figure 38: 'MSPrecedingSemester' class code

```
public class MSCurrentAcademicYear {

    public static boolean checkCurrentAcademicYear(IMAcademicYear academicYear){
        LocalDate date = LocalDate.now();
        return academicYear.getBeginingDate().isBefore(date) && academicYear.getEndDate().isAfter(date);
    }

    public static IMCurrentAcademicYear getCurrentYear()
        List<IMAcademicYear> academicYears = MSAcademicYear.getAcademicYears();
        for (IMAcademicYear $_iter:academicYears)
            if (checkCurrentAcademicYear($_iter))
                return new MCurrentAcademicYear($_iter);
    }

}
```

Figure 39: 'MSCurrentAcademicYear' class code

```
public class MSRepeatCourse {

    public static List<IMCourseRepetition> getCourseRepetitions(List<IMCourse> courses){
        List<IMCourseRepetition> result = new ArrayList<IMCourseRepetition>();
        for (IMCourse $_iter:courses)
            result.add(getCourseRepetition($_iter));
        return result;
    }

    public static IMCourseRepetition getCourseRepetition(IMCourse course){
        IMCourse courseRepetition = new MCourse();
        courseRepetition.setCourseEdition(MSNextSemester.getNext(course.getCourseEdition()));
        courseRepetition.setCoursePlacesLimit(course.getCoursePlacesLimit());
        courseRepetition.setCourseTemplate(course.getCourseTemplate());
        return new MCourseRepetition(courseRepetition);
    }

}
```

Figure 40: 'MSRepeatCourse' class code

```
public class MCourse extends DLClass implements IMCourse {

    int coursePlacesLimit;
    IMCourseEdition courseEdition;

    public MCourse(int coursePlacesLimit, IMCourseEdition courseEdition){
        this.coursePlacesLimit=coursePlacesLimit;
        this.courseEdition=courseEdition;
    }

    public int getCoursePlacesLimit(){
        return coursePlacesLimit;
    }

    public void setCoursePlacesLimit(int coursePlacesLimit){
        this.coursePlacesLimit=coursePlacesLimit;
    }

    public String getCourseFullName(){
        IMCourseTemplate courseTemplate = MSCourseBasedOnTemplate.getCourseTemplate(this);
        String courseName = courseTemplate.getCourseName();
        return MSCourseFullName.getCourseFullName(courseName, courseEdition);
    }

    (...)

}
```

Figure 41: 'MCourse' class code

```
public class MSWeightedAverageGradeComputation {

    public static double getWeightedAverageGrade(double weightedGradesSum, int weightsSum){
        return weightedGradesSum/weightsSum;
    }

    public static double getWeightedAverageGrade(List<IMPartialGrade> partialGrades){
        List<IMWeightedGrade> weightedGrades = new List<IMWeightedGrade>(partialGrades);
        double weightedGradesSum = MSWeightedGradesSummation.getWeightedGradesSum(weightedGrades);
        int weightsSum = MSWeightsSummation.getWeightsSum(weightedGrades);
        return getWeightedAverageGrade(weightedGradesSum, weightsSum);
    }
```

Figure 42: 'MSWeightedAverageGradeComputation' class code

```
public class MSWeightedGradesSummation {

    public static double getWeightedGradesSum(List<IMWeightedGrade> weightedGrades){
        double result = 0;
        for(IMWeightedGrade $_iter:weightedGrades)
            result+=$_iter.getGradeValue()*$_iter.getGradeWeight();
    }

}
```

Figure 43: 'MSWeightedGradesSummation' class code

```
public class MSSummarizeStudentAfterSemester {

    public static List<IMSummarizedStudent> getSummarizedStudents
        (List<IMStudent> students) {
        List<IMSummarizedStudent> result =
                new ArrayList<IMSummarizedStudent>();
        for (IMStudent $_iter:students)
            result.add(getSummarizedStudent($_iter));
        return result;
    }


    public static IMSummarizedStudent getSummarizedStudent
        (IMStudent student) {
        if (MSStudentToAccept.checkStudentToAccept(student))
            return new MSummarizedsStudent(MSAcceptStudent.
                    getAcceptedStudent(student));
        return new MSummarizedsStudent(MSFailStudent.
                getFailedStudent(student));
    }

}
```

Figure 44: 'MSSummarizeStudentAfterSemester' class code

```
public class MSStudentToAccept {

    public static boolean checkStudentToAccept(IMStudent student){
        for (IMFinalGrade $_iter:student.getFinalGrades())
            if (!$_iter.getGradeValue()>=3)
                return false;
        return true;
    }

    public static List<IMStudent> getStudents(){
        List<IMStudent> students = MSStudent.getStudents();
        List<IMStudent> result = new ArrayList<IMStudent>();
        for (IMCourse $_iter:students)
            if (checkStudentToAccept($_iter))
                result.add($_iter);
        return result;
    }

}
```

Figure 45: 'MSStudentToAccept' class code

```
public class MSAcceptStudent {

    public static IMAcceptedStudent getAcceptedStudent(IMStudent student){
        student.setStudentStatus(MStudenStatusValueEnum.ACTIVE);
        student.setStudentSemester(student.getStudentSemester()+1);
        return new MSummarizedStudent(student);
    }

}
```

Figure 46: 'MSAcceptStudent' class code