Faculdade de Ciências e Tecnologia

Universidade Nova de Lisboa

Manual

DSLTrans

Authors:
Cláudio Gomes
Bruno Barroca

Supervisor:
Prof. Dr. Vasco Amaral

December 4, 2014

Contents

Versions

August 2011 Initial version.

December 2014 Updated installation instructions to the Eclipse Luna distribution.

1 Introduction

Model transformation is the process of converting one or more input models to one or more output models [11] where each model conforms to a metamodel (a set of well formedness rules that specify the abstract syntax of models and the interrelationships between model elements, i. e., the set of all possible conformant models [3]) specified using a metamodeling language that in this case is *Ecore*.

Ecore is a subset of the *Unified Modelling Language* (*UML*) and the main language for the creation of metamodels in the *Eclipse Modelling Framework* (*EMF*) [14].

EMF provides a modelling framework and code generation facility that lets us define models, from which then we can generate other models or code for building tools and other applications [14] [1].

Figure 1 shows the model transformation process. We can see that every model involved has to conform to some metamodel. At the highest level, the metametamodel conforms to itself meaning that its structure can be expressed with the same terms it describes. The rules that make up the transformation process are also expressed in a model that is interpreted by some engine that takes some input models and generates some output models.

[1]For more information about the *Ecore* language and *EMF* refer to `http://www.eclipse.org/modeling/emf/?project=emf`

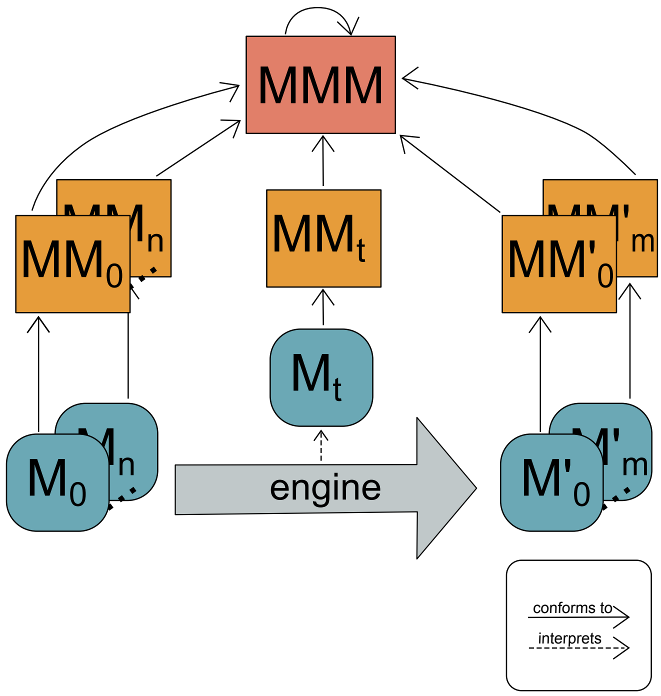


Figure 1: Model transformation process and required elements.

1.1 What is *DSLTrans*?

DSLTrans is a language specifically designed to support the definition of *Ecore*-based model transformations [6]. It is particularly useful when building a new language (for instance, a language to describe graphical user interfaces) whose semantics are not known and it is necessary to express them in terms of an existing well known language (a Java application[2]).

The process of assigning meaning to a new language trough transformations involves coming up with a set of mappings between the terms of the source language to terms in the target language [6]. In *DSLTrans* those mappings are expressed in the form of rules where the first part of each rule has a pattern describing some arrangement of terms in the source language and the second part has the terms to be created in case the first part exists in some input model.

Figure 2 shows the model transformation process according to the technologies and tools used throughout this manual. As you can see, *DSLTrans* is a metamodel used to express transformations that are interpreted by the *DSLTranslator* engine to translate models.

[2]Java code can be modelled using an *Ecore*-based metamodel thus making it possible to treat Java applications as models

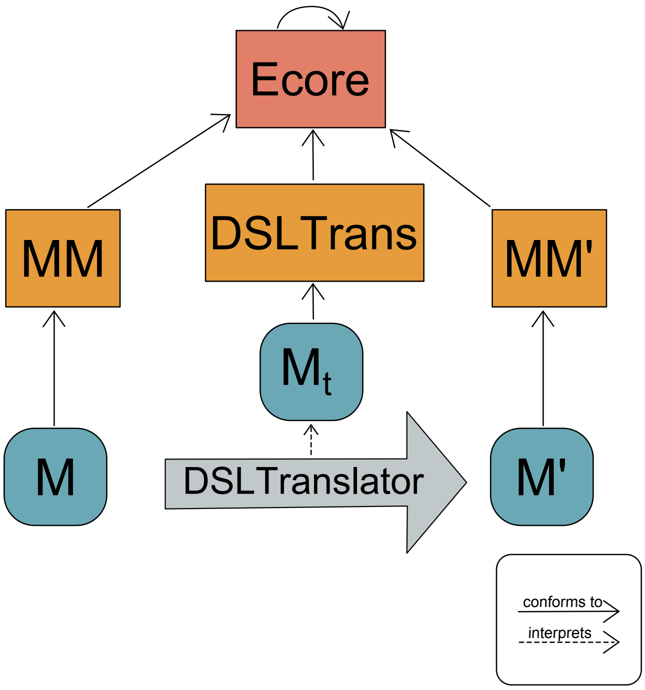


Figure 2: Model transformation process and required elements used throughout this manual.

1.1.1 A Metaphor

In order to better understand all these concepts, lets focus on a simple example where we have a small language (a.k.a. a metamodel) used to define an individual's genealogy tree and we will come up with transformations that present information from some genealogy tree (a.k.a. model) in different views.

Figure 3 shows an example of a genealogy tree. We can see that John and Mary married and had one son: Thomas who in turn married Sarah… and so on.

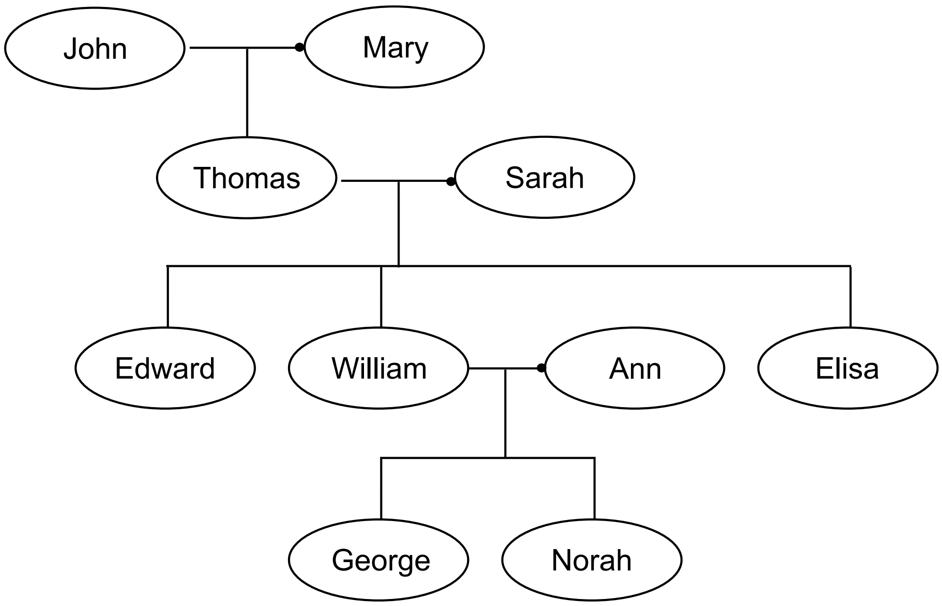


Figure 3: Genealogy tree example.

How can we find out, given a tree of any size, who are the couples? More specifically we want as a result of the transformation a set of couples like the one shown in figure 4.

The transformation rules have to be based on patterns as described earlier so something like figure 5 should be fine. The transformation is only made of a single rule, that says the following: *Every person that is married to other person in the source model becomes the same person associated with the same other person in the target (or output) model.* Notice that the connections in the source model are different than those in the target model.

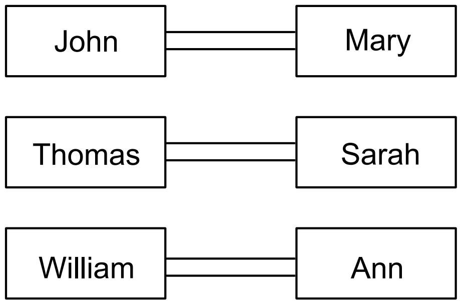


Figure 4: Couples set example.

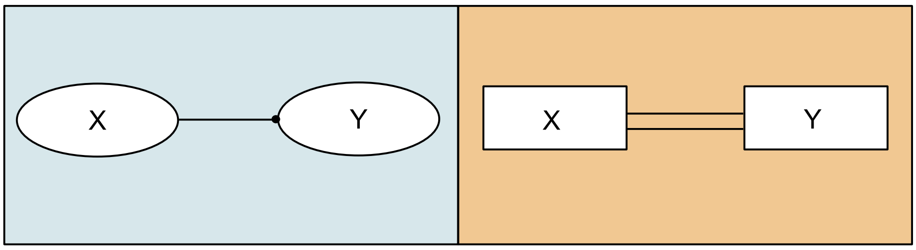


Figure 5: Genealogy to Couples transformation.

Another way of looking at the transformation is by defining two rules: one establishing the fact that *every married couple in the source metamodel becomes two individuals in the target* and afterwards *add* the relation between those two individuals, forming a couple. Figure 6 illustrates this approach. Notice that the dashed links between the source and target model elements mean that in the bottom rule we don't want to create new elements in the target model: we only want to create a connection between them.

Although partitioning the transformation and referring to elements that already exist in the output model[3] may seem too much work and counter-intuitive; for complex transformations it is a more natural way since we start by looking only to each element individually and expressing its meaning in the target language through simple rules, then we explore more and more complex patterns saying what those mean.

[3]because these elements where generated by some previous rule

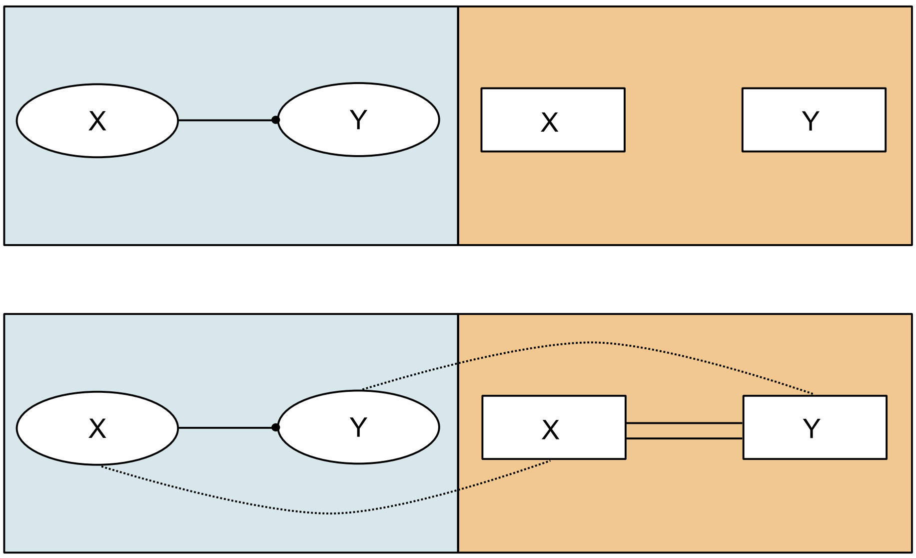


Figure 6: Genealogy to Couples transformation - Partitioned.

1.2 DSLTrans and Other Transformation Languages

There are several papers about transformation languages. Bellow is a brief description of some of them so you can have an idea about their general features with respect to *DSLTrans*.

1.2.1 QVT

Query / View / Transformation is a standard defined by the *Object Management Group*; its implementation is *SmartQVT*: a tool that compiles model transformations specified in QVT to produce Java code used to perform these transformations [8].

1.2.2 ATL

Atlas Transformation Language is a transformation language inspired by the *QVT* requisites that uses the *OCL* formalism. It provides declarative constructs as we used in the previous example and, for the most difficult transformations, it allows the user to define imperative rules, which increases its flexibility and complexity [9].

1.2.3 ATC

Atomic Transformation Code is a low level model transformation language with the main purpose of being the target for other transformation languages' specifications allowing for the execution of a diversity of model transformation languages via translation to *ATC* [13].

1.2.4 ETC

Epsilon Transformation Language is an hybrid[4], rule-based model-to-model transformation language that has the flexibility of allowing for query, navigation and modification of multiple target and source models [1].

1.2.5 MOLA

MOdel transformation LAnguage, similarly to *DSLTrans*, is a graphical transformation language that strives to produce readable transformations by combining traditional structured programming in a graphical form with simple pattern rules [10].

[4] An hybrid transformation language is one that provides both imperative and declarative constructs, as *ATL*, *ETC*, and others.

1.2.6 RubyTL

Ruby Transformation Language is a domain specific hybrid transformation language embedded in Ruby that provides an extension mechanism based on plugins [4].

1.2.7 UMLX

UMLX is a graphical transformation language that extends *UML* through the use of transformation diagrams that are no more than class diagrams with additional relations to specify mappings between input and output models [15].

1.2.8 GReAT

The Graph REwriting And Transformation is a rule-based graphical language that, as DSLTrans, sees models as labelled graphs and uses graph transformation formalisms to translate them [2].

1.2.9 DSLTrans

With respect to the described transformation languages, *DSLTrans* is a rule-based graphical transformation language[5] that uses declarative constructs and graph formalisms to prescribe transformations. A distinctive characteristic is that all the transformations specified in *DSLTrans* are guaranteed to terminate[6]. This means the language is not Turing complete and for very complex transformations it might not be the ideal transformation language.

[5]The latest version of DSLTrans supports also textual syntax for transformation specification

[6]DSLTrans transformations are guaranteed to terminate because the language doesn't support loop constructs and no rule can be applied forever. This means the language is not Turing complete but there are techniques that allows us to build complex and still readable transformations as you will see.

1.3 Assumptions

1.3.1 User

About the reader of this manual and hopefully user of *DSLTrans* we make the following assumptions:

Modelling Jargon The reader should be familiar with the modelling terms like model, metamodel, metametamodel, model transformation, language, etc.... [3] and [11] give readable overviews of most of the terms used throughout this manual.

Eclipse Modelling Framework and Ecore The user knows how to use the EMF main metamodeling language to created metamodels and respective instances. For more information refer to the project's home (`http://www.eclipse.org/modeling/emf/`) or read the EMF book in [14]. There is also a good tutorial on how to create metamodels here: `http://tinyurl.com/3oo8woz`

Advanced System User We assume that the user knows how to change environment variables and install software in its system.

1.3.2 Environment

In order to succeed in learning DSLTrans and following the examples presented in this manual it is highly recommended that you have the following tools:

- Eclipse Modeling Tools, Luna Service Release 1. You should be able to download it in the eclipse home page `http://www.eclipse.org`. After starting eclipse and going to *Help, About Eclipse*, you should see the version info shown in figure 7.

- SWI-Prolog version 5.10.2 by Jan Wielemaker. The *DSLTranslator* engine uses the Prolog language internally to process the transformations. You should be able to download SWIProlog from the project's home page `http://www.swi-prolog.org/`. Figure 8 shows Prolog's version info.

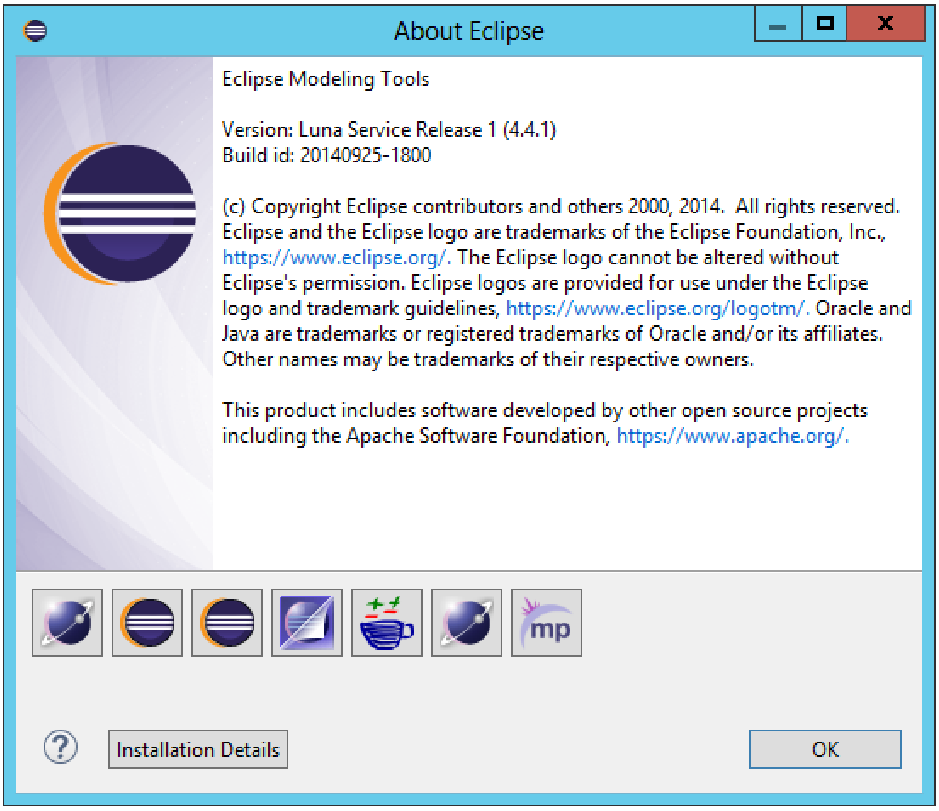


Figure 7: Recomended eclipse version.

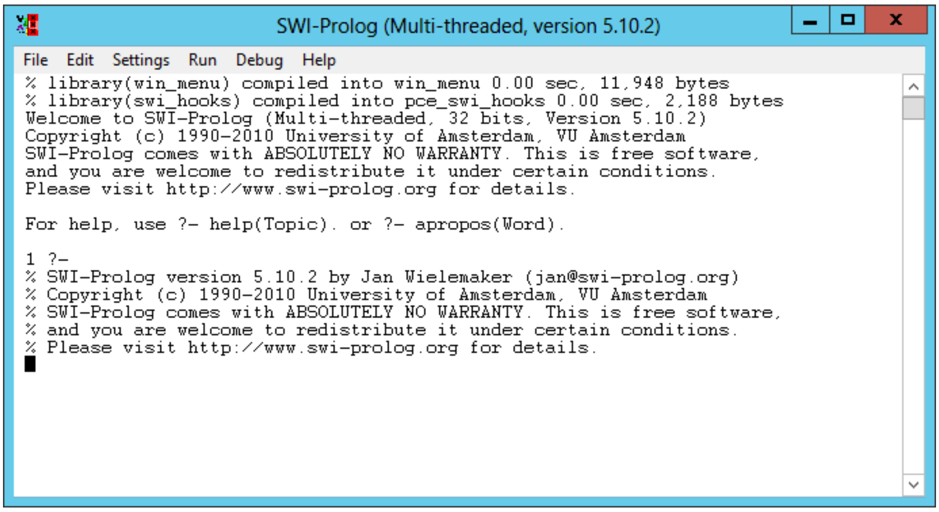


Figure 8: Recomended prolog version.

1.4 About this Manual

1.4.1 Objectives

After reading this manual the user should be able to:

- Create and read *DSLTrans* transformations either from scratch or by using prototyping techniques.
- Understand how higher order transformations work and even better: create them.
- Understand the advantages and limitations of *DSLTrans* and when to use it instead of other transformation languages like ATL [9], *SmartQVT* [8] or others presented in page 14.

1.4.2 Structure

If you are new to *DSLTrans*, you should read this manual from the beginning to at least section 4. The first sections are the ones that will help you understand the main concepts behind the *DSLTrans* and the remaining can be used as a reference.

This manual is divided in five sections:

Installation Where you will learn how to get the needed tools up and running so that you can follow the rest of this manual's examples and tutorials.

Quick Start This section presents a hands on approach to *DSLTrans* with a tutorial on how to create the transformation described in section 1.1.1 in two different ways. While it teaches you how to use *DSLTrans*, it explains the main concepts and procedures involved in the creation of a transformation.

Language Definition Where each element of a transformation is described and some examples in both graphical and textual syntax are given to help you understand how it can be used; possible restrictions and good practices are present. This section can be used as a reference as it contains the description of every element in the language.

Advanced Topics Once you know how to use *DSLTrans* well enough you will want to avoid some repetitive tasks when building most of the transformations. Since *DSLTrans* is a metamodeled language it is possible to use it to transform transformations. This section focusses in teaching you how to use (and build) higher order transformations and to build complex transformation to simulate the stepping of abstract machines and even how to create transformations that are specific to some domain.

FAQ and Common Installation Problems Frequently asked questions and common installation errors are described here.

ïż£

2 Installation

2.1 Windows

In order to successfully follow the examples in this manual and use *DSLTrans* you have to follow (carefully) these steps:

Download and install SWI-Prolog version 5.10.2. You will find it inside the *DSLTrans-Release* folder in our repository: `https://github.com/githubbrunob/DSLTransGIT`

Download and extract Eclipse Modeling Tools, Luna Service Release 1.

Set the environment variables shown in the figures 9, 10 and 11 below.

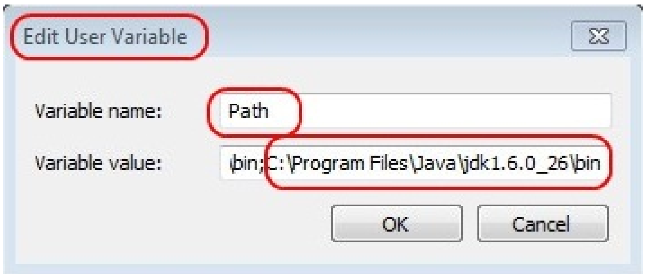


Figure 9: Path to add to the *user Path* variable.

Note that, figure 9 you have to replace `C:\Program Files\Java\jdk1.6.0_26\bin` for your system's Java bin directory. Also, beware that the environment variable you have to edit is the *user* Path variable.

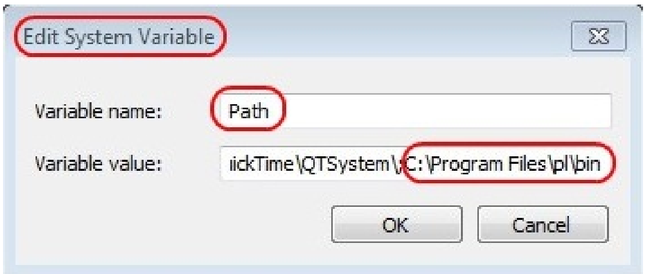


Figure 10: Path to add to the *system Path* variable.

In figure 10, you have to replace `C:\Program Files\pl\bin` for your prolog installation bin directory too. This time the variable to edit is the *System* Path variable.

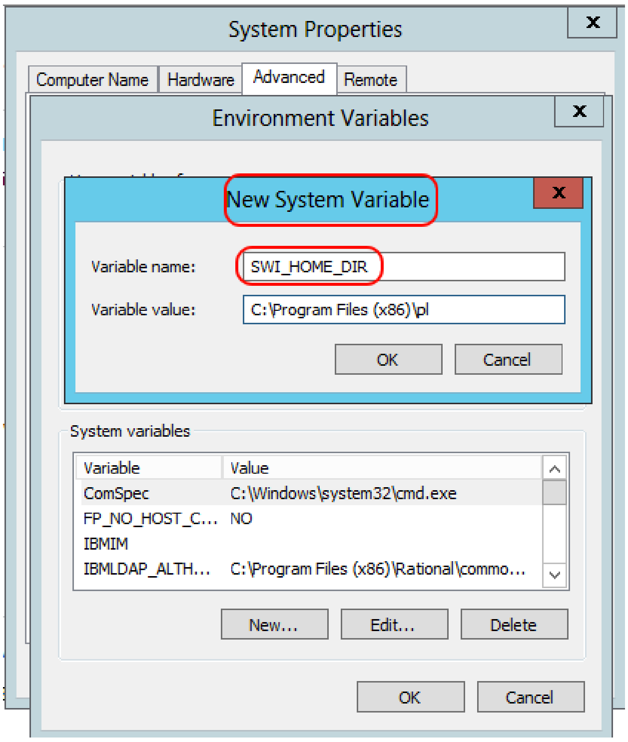


Figure 11: Path to add to the *system SWI_HOME_DIR* variable.

In figure 10, you have to replace `C:\Program Files (x86)\pl` for your prolog installation directory.

Now you have to copy the `jpl.jar` file, in the `C:\Program Files\pl\lib` directory, and paste it in Java's lib directory: `C:\Program Files\Java\jdk1.6.0_26\lib`.

Finally, you need to install the DSLTrans editor and transformer features from the Dsltrans update site. For this you need to download and extract the DSLTransUpdateSite from the *DSLTrans-Release* folder in our repository and install the features to eclipse using that update site.

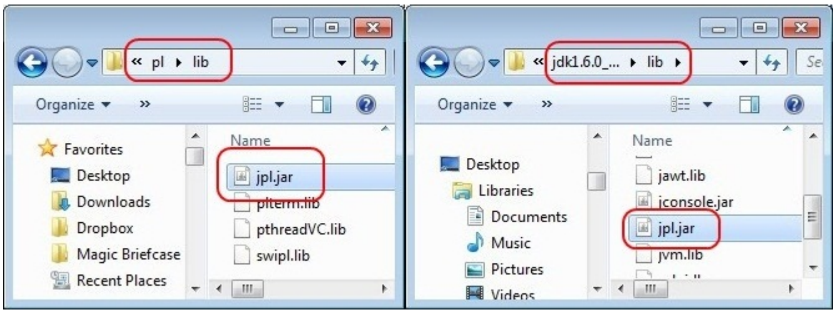


Figure 12: Prolog's jpl.jar copied to Java's lib directory.

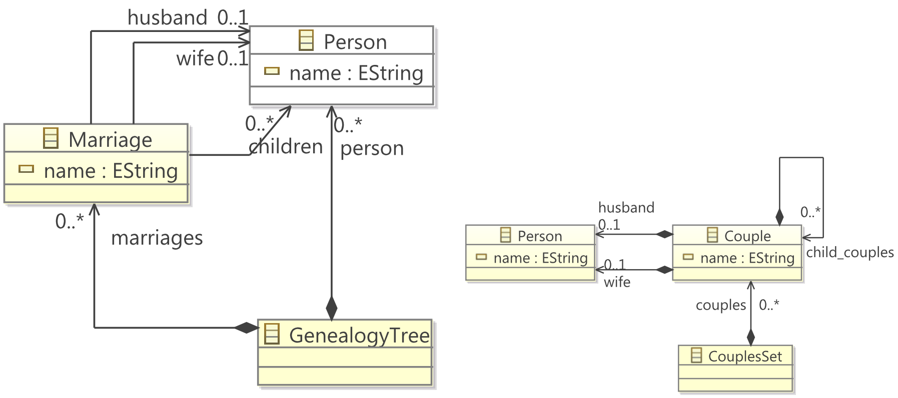
ïz£

3 Quick Start

In this section you will create the two transformations described in page 11 using *DSLTrans* graphical syntax.

3.1 Metamodels

First step is to build the required metamodels: *GenealogyTree* and *Couples*. Both were built using *Ecore Diagram Editor* and are shown in figures 13a and 13b.



(a) GenealogyTree. (b) Couples.

Figure 13: Metamodels.

Notice that in the *Couples* metamodel the notion of parenting relationship between *Couples* is kept. For a given couple to be directly connected to other couple it means that the first one gave birth to one of the elements of the second couple.

3.2 Example Model

To test the transformation you will need an example model. Open the *GenealogyTree.ecore* file and click on *Create Dynamic Instance...*. Name the new file as *GenealogyTree.xmi* (see figure 14). It is important that you name it like that to avoid confusion later in this tutorial.

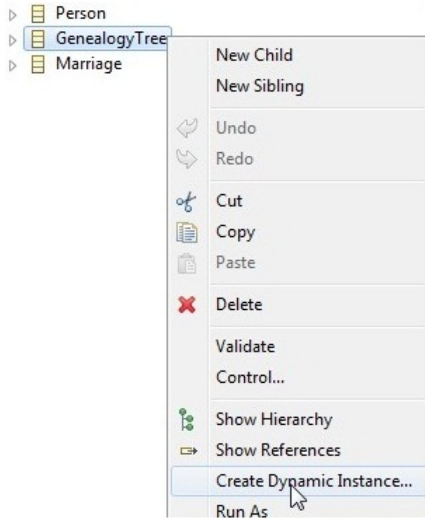


Figure 14: Dynamic Instance Creation.

Then open the created file (*GenealogyTree.xmi*) and create a model based on figure 3 in page 11. Your model should look like the one shown in figure 15. Don't forget to fill the *Children*, *Husband* and *Wife* properties (in eclipse's properties view) for each *Marriage*, where appropriate.

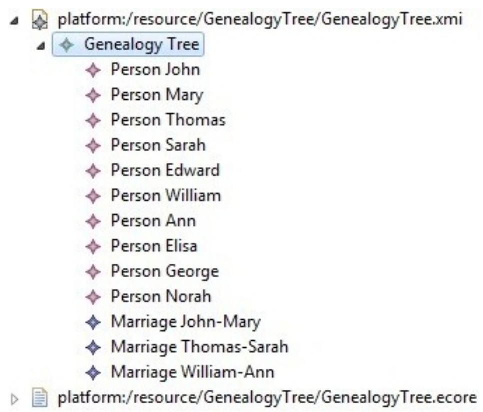


Figure 15: *GenealogyTree* Example Model.

3.3 Planning the Transformation

In page 11 we had two ways of expressing the transformation to generate *Couples* models: a simple one and an extended, more partitioned, version.

Informally, if we ignore the parenting relationship between couples, we can say that a couple is a *Couple* element, together with the respective *husband* and *wife Persons*. So we only have to match the *Persons* and *Marriage* elements in the *GenealogyTree* metamodel.

It is advised that before you start building the transformation, you write down the basic rules (or steps if you prefer) that make it up. For this case study ignore the parenting relationship between *Couples* and use the following rules:

1. Every *Marriage*, *husband* and *wife Persons* in the *GenealogyTree* becomes a *Couple*, *husband* and *wife Persons* in the *Couples* model;

2. Every *Couple* generated has to be connected with the *CouplesSet* (root) element.

3.4 Understanding DSLTrans Overall Semantics

A *DSLTrans* transformation is composed of multiple *layers*, each with several *rules*. A *rule* has a match side - where a pattern is matched against some input model - and an apply side - where a pattern is created in the output model. It is applied while there are elements in the input model that satisfy the match pattern. In a *layer*, all the *rules* are executed in a non-deterministic fashion while *layers* are executed sequentially following the *previous source* association.

3.5 Creating a Blank Transformation

Now that you have an idea of the rules needed and how a transformation is processed, you can start creating a blank transformation.

First open the *New File Wizard* and select *DSLtrans Diagram* inside the *Examples* category (see figure 16).

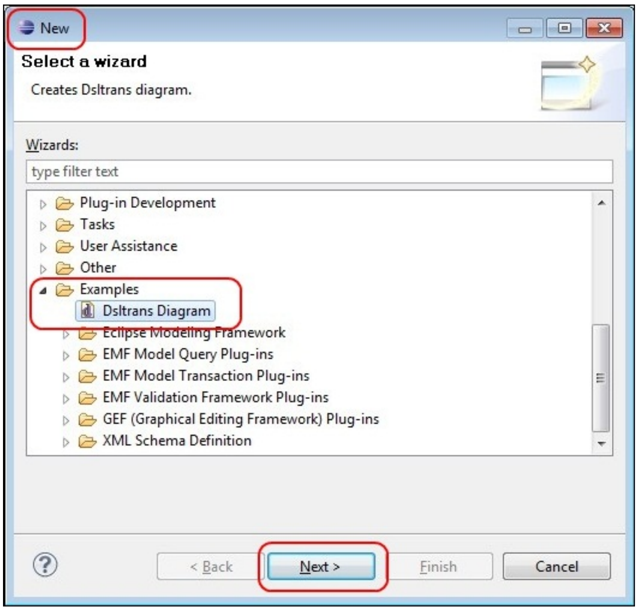


Figure 16: New File Wizard.

DSLTrans transformations are nothing but models conforming to the *DSLTrans* metamodel. According to *EMF*, the abstract definition of models is expressed in the *XML Metadata Interchange* (*XMI*) [7]. Since you are using the graphical syntax to build the transformation, the new *DSLtrans Diagram* wizard will create two files:

NewTransformation.dsltrans This file contains the transformation model in *XMI* format. See figure 17.

NewTransformation.dsltrans_diagram This file contains the additional information needed to create and position the transformation elements in a diagram. See figure 18.

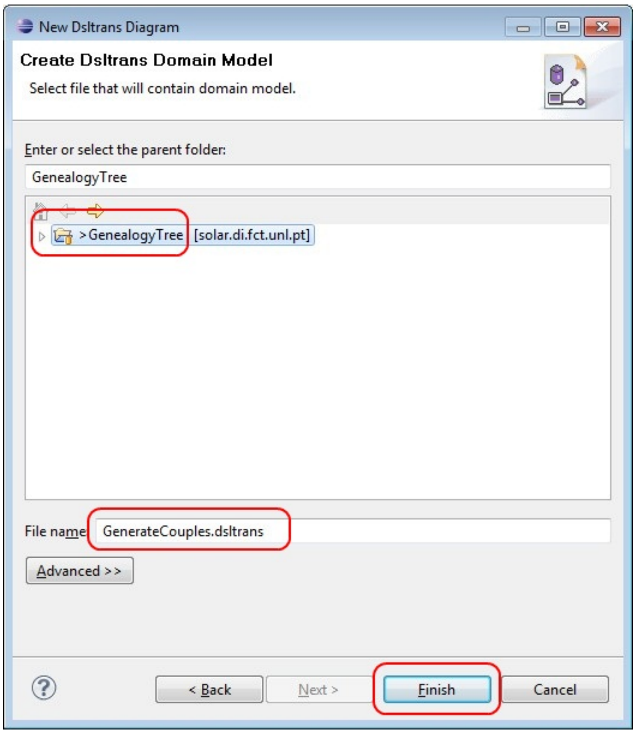


Figure 17: Setting model name.

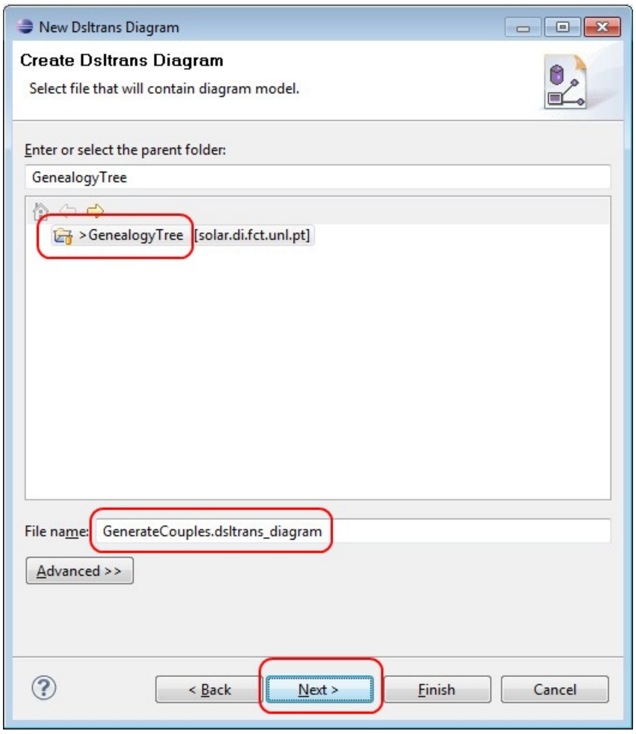


Figure 18: Setting diagram name.

3.6 DSLtrans Diagram Editor

After creating and opening a new *DSLTrans* file, you will see a window like the one shown in figure 19.

While building a transformation you will frequently use:

- The *Properties* window to set package names and other properties of the transformation elements;
- The *Palette* is used to add new elements to the transformation (e.g., *rules*, match classes, etc. . .) and connections among them;
- The *Outline* view to get an idea of the overall structure of the transformation and navigate easily trough complex diagrams;

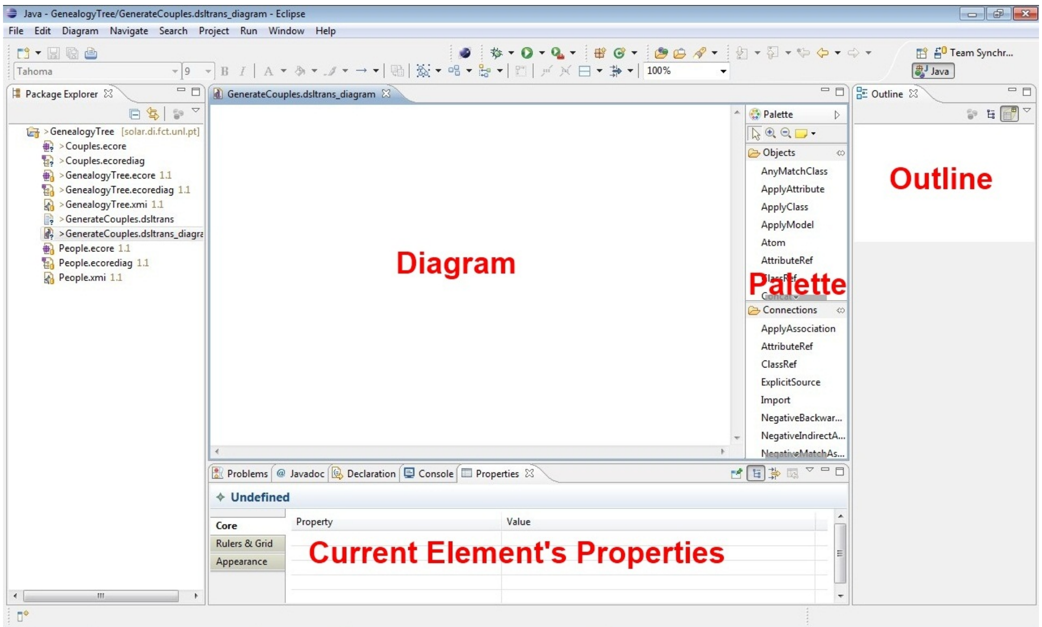
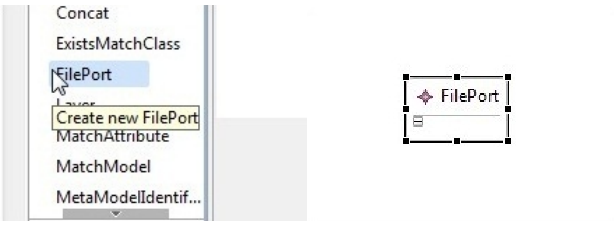


Figure 19: DSLTrans Diagram Editor Window.

3.7 Defining the Transformation

A transformation can have multiple input and output models but for this example you only need one input and one output. To set the input for the transformation you will add a new *FilePort* by clicking it in the objects section of the *Palette* and clicking again in the diagram. After this you should see something like figure 20.



(a) FilePort Palette. (b) FilePort Diagram.

Figure 20: Adding a new element - FilePort.

Then you should set the *FilePort*'s properties in the *Properties* window as in figure 21. Notice that the *Name* property can be anything. Just write something meaningful for the sake of readability.

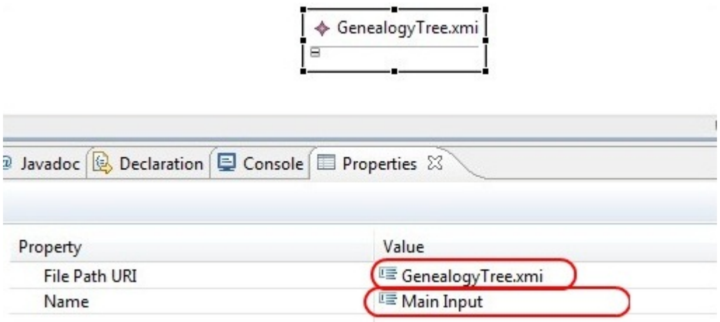


Figure 21: FilePort properties.

For every input model of a *DSLTrans* transformation, the metamodel it conforms to must be stated. To do that, you add a *MetaModelIdentifier* inside the *FilePort* previously created. Then set the properties as in figure 22.

Beware that *Meta Model Name* as to be always in the format `package.Package`. The `package` value is the metamodel root package (see figure 23). By default, this is the name of the metamodel in lower case letters.

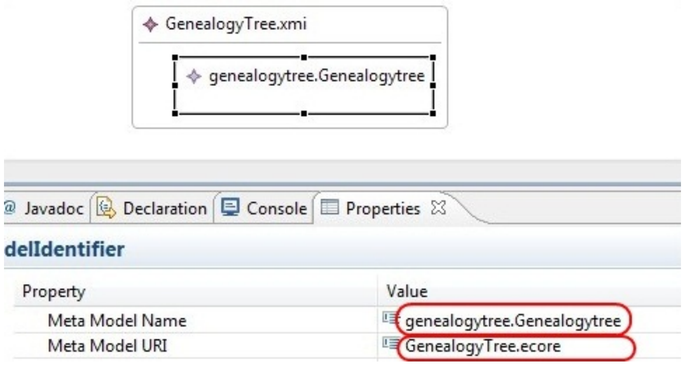


Figure 22: MetaModelIdentifier properties.

Now that the input is well known and identified, you can proceed to add a first *Layer* by the same procedure as described earlier to add elements to the transformation. As for its properties it is only recommended that you set the *Name* and *Description* to something meaningful.

The next step is to connect the input *FilePort* to the first *Layer* using a *PreviousSource* association. To add this connection you have to first click in the *PreviousSource* in the *Palette*, then click in the *Layer* and drag the connection to the *FilePort*. The result is in figure 24. Don't be misled by the fact that the arrows points downwards while you dragged it upwards. It shows the flow of the information but its name is *PreviousSource*.

Each *Layer* has an output. You can set that output to a file (by setting the *Output File Path URI* property) if you want to see the outcome of the transformation at a specific *Layer*. This is great for debugging purposes. You should set the *Output File Path URI* property to *Couples.xmi*. Even if there is no external output set, the outcome of a layer is always validated against it's metamodel. Because of it you have to create a *MetaModelIdentifier* pointing to the output metamodel for each layer. Figure 25 shows the properties of this *MetaModelIdentifier*.

As for the rules in this *layer*, we only need one: to create the root element of the output model. It has to be like this because in the next layers new elements will be generated and they need to be "attached" to the root element as described in the *Couples* metamodel. Don't worry if you can't understand everything yet, keep going, you're almost there!

Now insert a *Rule* in the recently created *Layer* and set its description to something that describes the main purpose of the *Rule* (e.g., root element). After that, insert a *MatchModel* and an *ApplyModel* in the top and bottom parts of the *Rule*, respectively (see figure 26).

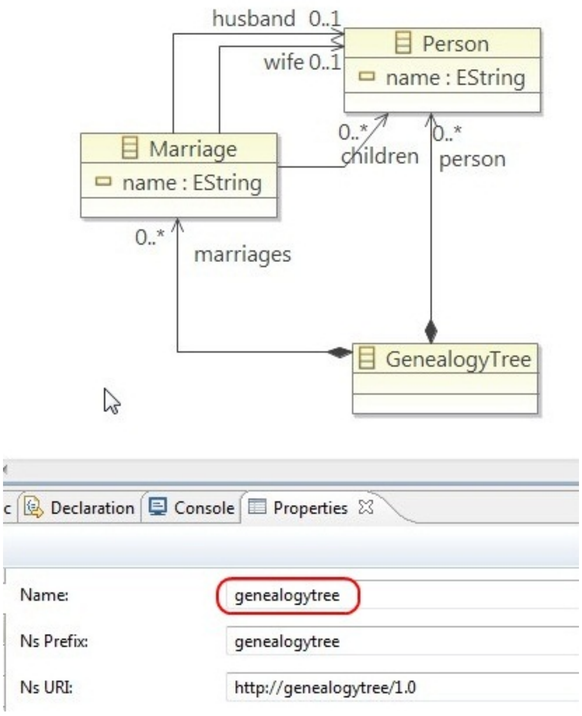


Figure 23: GenealogyTree metamodel package name.

According to figure 13 the root element of the *GenealogyTree* metamodel is the *GenealogyTree* element and the root of the *Couples* metamodel is the *CouplesSet* element.

The purpose of this rule is to say that for every *GenealogyTree* element in the input model, you want to generate a *CouplesSet* element in the output model. Figure 27 shows the *AnyMatchClass* and the *ApplyClass* elements created inside the respective containner models created previously. As for the properties of each inserted element, set them according to figure 28.

Now would be a good time to test the transformation. The transformation has one *FilePort* that points to a *GenealogyTree.xmi* file where the input model is (you created it in section 3.2); and one *Layer* whose output is a file named *Couples.xmi*, where the output model will be.

To run the transformation, just right-click in *TransformationName.dsltrans*, *DSLTranslator* and *Transform*, as in figure 29. You should see some debugging output in the console view. If any error occurs, refer to section 6 in page 91.

If everything went well you should see a new file named *Couples.xmi* on

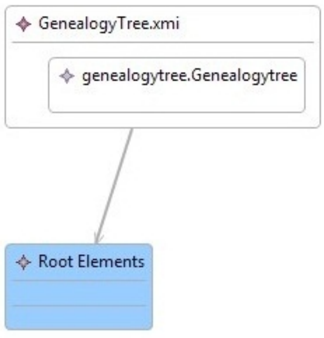


Figure 24: Transformation after adding the PreviousSource Association.

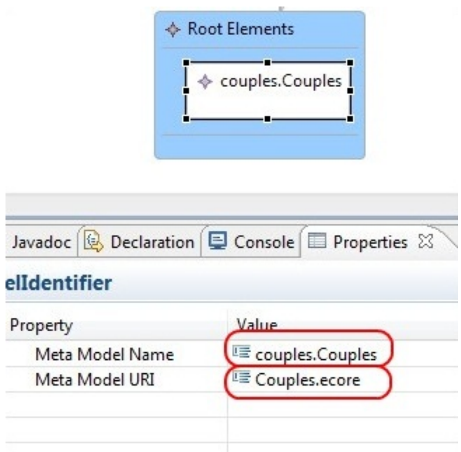


Figure 25: *MetaModelIdentifier* properties.

your project. Open it and you will see that the model only contains the root element (see figure 30). This makes sense since the transformation only matches *GenealogyTree* elements to produce *CouplesSet* elements and there is only one *GenealogyTree* element in the input model.

It's time to add a second *Layer* to the transformation. Don't forget to identify the output metamodel using the *MetaModelIdentifier* as previously. The properties of the *Layer* and *MetaModelIdentifier* are the same except theres is a new *Previous Source* association between the second *Layer* and the first one, as figure 31 illustrates.

The second *Layer* will have rules that match *Marriages* in order to generate *Couples*. The skeleton of the *Rule* to add is quite simple (see figure 32). Beware that you need to add *Match* and *Apply* models to each side of

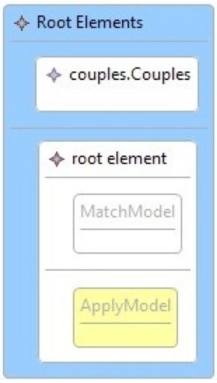


Figure 26: Root Elements rule with empty match and apply models.

the *Rule* before adding the *AnyMatchClasses* and *ApplyClasses*.

On the top of the *Rule* it is necessary to match a *Marriage* and the two associated *Persons*, so go ahead and set the appropriate properties for three of the four *AnyMatchClasses* in the *MatchModel* (see figure 33). Since a *Couple* and two *Persons* will be generated by this *Rule*, set the properties of the *ApplyCasses* according to figure 33. It is very important that you set the *PackageName* property of each *Rule* element. In this case we set the *PackageName* of *AnyMatchClasses* to `genealogytree` and the *ApplyClasses* with `couples` (as in the previous *Layer*).

The generated elements *Couple* and *Person's* need to be associated with each other and with the root element *CouplesSet* according to the *Couples* metamodel. To create associations between apply elements you have to insert *Apply Associations*, click and drag. Insert the needed associations between the generated elements according to figure 34. Notice that the direction of the *ApplyAssociations* and their names correspond to the associations declared in the metamodel. This is very important since *DSLTrans* will check the generated model correctness and won't allow models that do not conform to the metamodel.

The generated *Couples* need to be related to the root element *CouplesSet*. Set the properties of the last *ApplyClass* and insert the association as shown in figure 35.

If you start asking why do we want to generate more *CouplesSet* elements, then you are in the right track! In this rule you want to add new elements (*Couple* and *Persons*) but connect them to the previously generated *CouplesSet* element. How do you say in *DSLTrans* that you don't want to generate a new *CouplesSet* element but instead want to use the previously generated one? The answer is to add a *PositiveBackwardRestriction* between the gen-

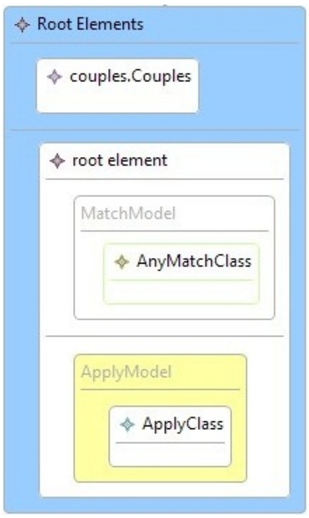
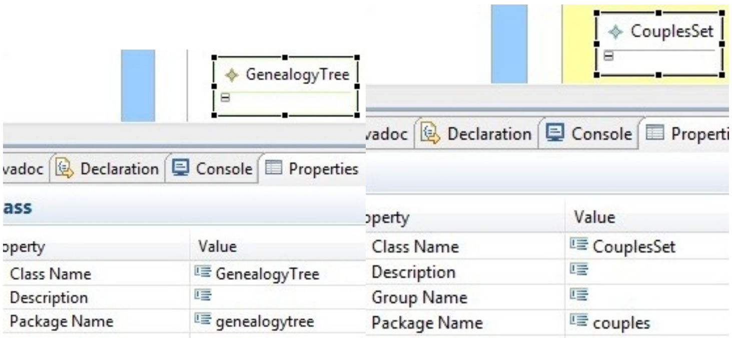


Figure 27: AnyMatchClass and ApplyClass elements.

erated element and one of the elements that generated it. In this case the generator element is the *GenealogyTree* and the generated one is the *CouplesSet*. Insert a *PositiveBackwardRestriction* between the *GenealogyTree* and the *CouplesSet* and set the required properties so that the *Rule* looks like the one in figure 36. Don't forget to set the appropriate *Package Names*, it's one of the most common errors (see section 6).

The apply pattern of the *Rule* is complete and the match elements only need associations between them. To express that the two *Person's* are in the same *Marriage*, you have *PositiveMatchAssociations*. They work much in the same way as the *ApplyAssociations* in the apply pattern, except they are inserted among match elements. Insert the required associations so that your match pattern looks like the one in figure 37.

Now the rule (and the transformation) is almost complete. However, an important detail is missing and without it the transformation won't work: when using *PositiveBackwardRestrictions* you are matching previously generated and generator elements. *DSLTranslator* internally keeps track of these elements but only if you say so, or else executing a large transformation would consume a lot of memory. In order to tell *DSLTranslator* to save traceability links between generated and generator elements you have to place an *ApplyAttribute* in the generated elements in the moment they are first created and then use the same *ApplyAttribute* to refer to them in later *Layers*. It's like using variables inside a transformation. Go back to the first layer in the transformation and assign an *ApplyAttribute* to the *CouplesSet* element and



(a) GenealogyTree AnyMatchClass properties.

(b) CouplesSet ApplyClass element properties.

Figure 28: Match and Apply Pattern properties.

place an *Atom* inside it with the value *Root Element* (see figure 38). Note that you have to leave the *Attribute Name* property of the *ApplyAttribute* empty.

Now in the second *Layer*, add an *ApplyAttribute* with the same *Atom* value in the *CouplesSet* element, as in figure 39.

Executing the transformation now should produce a result similar to the one in figure 40.

All the generated elements' attributes are missing. Apart from the *ApplyAttribute* (with no name) that you set for the root element, you didn't create any attribute for other elements.

You need to copy the name attribute from each element in the input model to the output model. To do that, insert an *ApplyAttribute*, name it according to figure 41 , insert *AttributeRef* (Objects) inside each *ApplyAttribute*, place *MatchAttributes* inside the relevant elements in the match pattern and insert *AttributeRefs* (Connections) between the *ApplyAttributes* and the corresponding *MatchAttributes*. The resulting *Rule* is in figure 41.

You just told *DSLTranslator* to copy the *name* attributes from the elements matched in the input model and paste them in the applied elements.

Finally, you can run the transformation against any model (expressed in *GenealogyTree.xmi*) and get the set of couples (in the *Couples.xmi* resulting file). The result for our case study is shown in figure 42.

With this transformation you are able to obtain a model of the existing couples in a genealogical tree. But wouldn't it be great if you were able to see the relations between couples. Who is the oldest couple? And the youngest? In the next session you will learn to build a transformation for that.

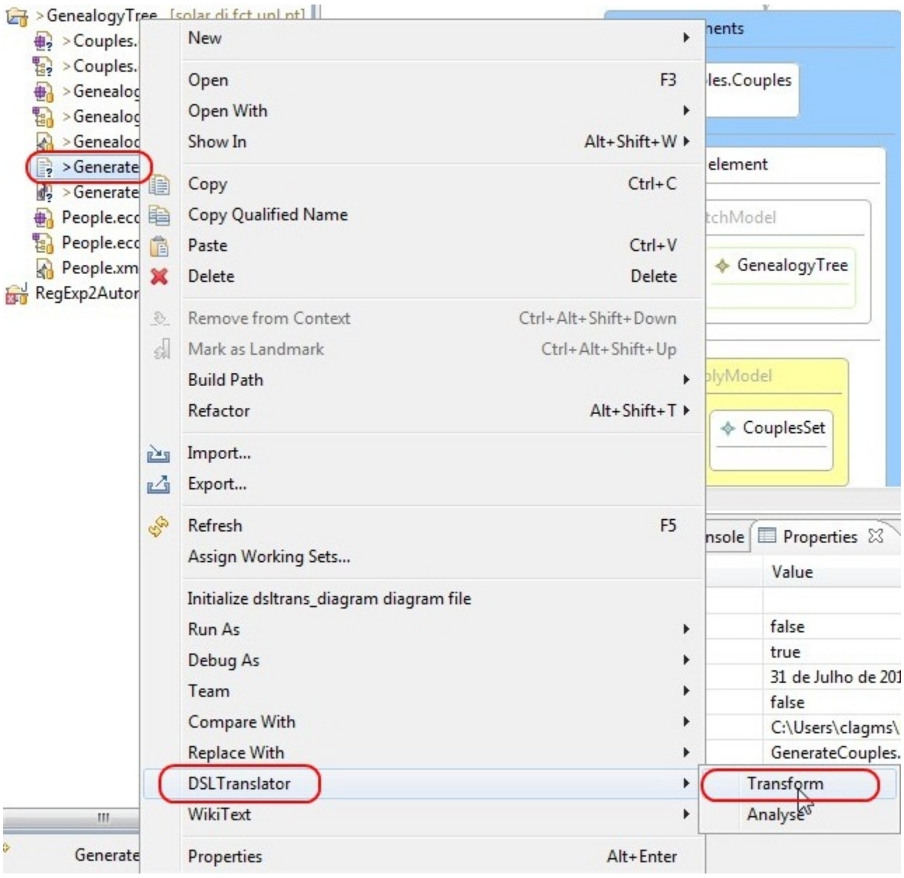


Figure 29: Executing a transformation.

Figure 30: Couples resulting model.

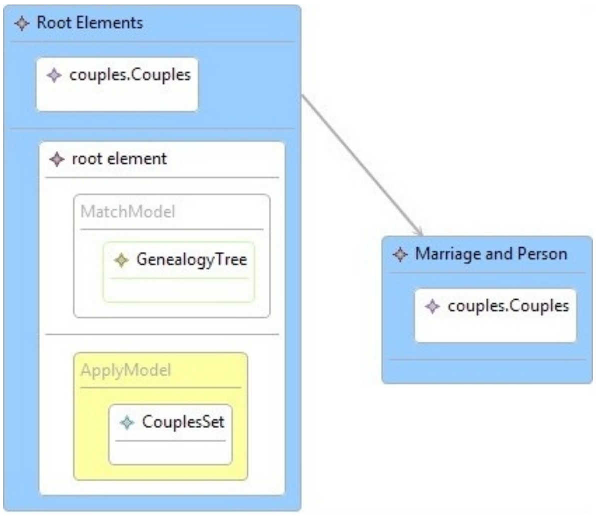


Figure 31: New *Layer* with *Previous Source* association.

Figure 32: Second Rule Skeleton.

Figure 33: Second Rule with class names.

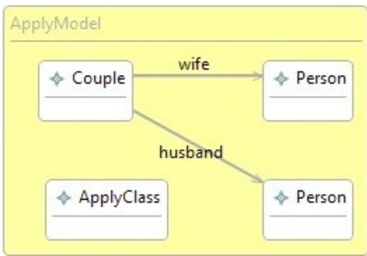


Figure 34: Generated *Couple* associations.

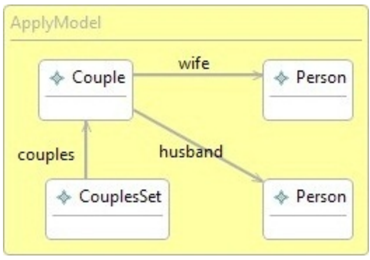


Figure 35: Generated *Couple* associations with *CouplesSet*.

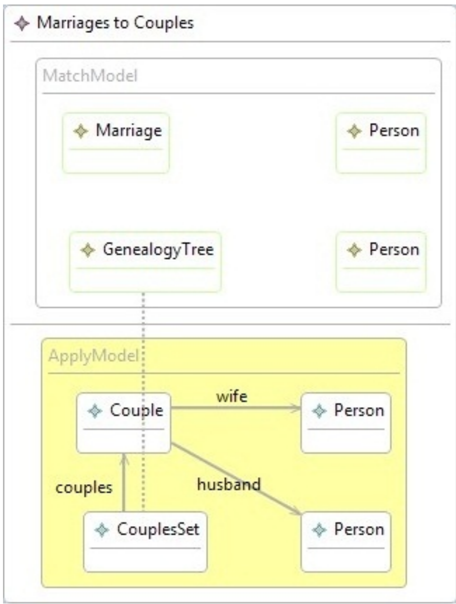


Figure 36: Rule with a *PositiveBackwardRestriction*.

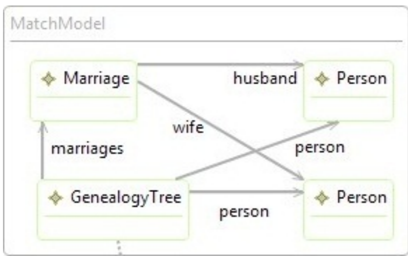


Figure 37: Match pattern with *PositiveMatchAssociations*.

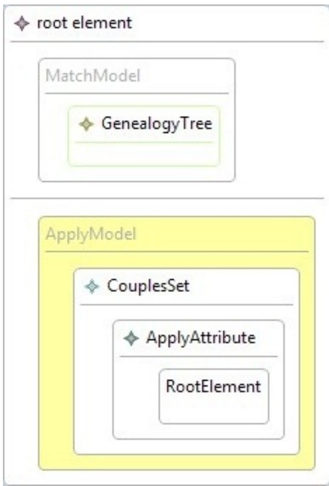


Figure 38: Root elements rule with *ApplyAttribute*.

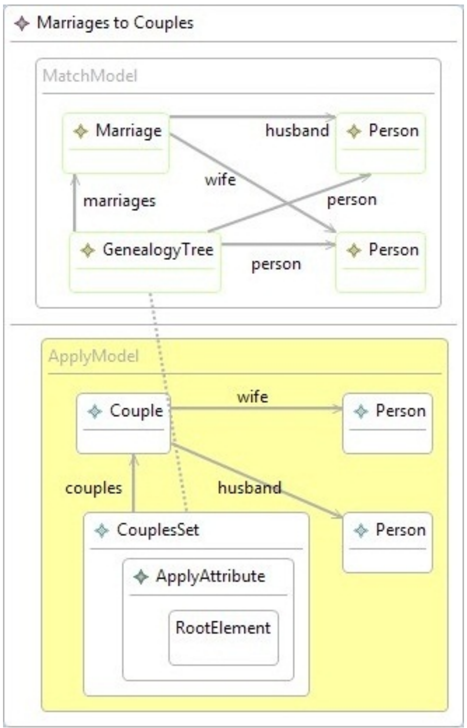


Figure 39: *CouplesSet* element with *ApplyAttribute*.

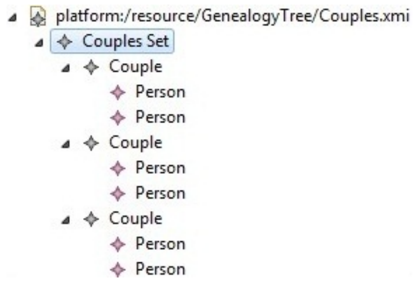


Figure 40: *Couples* result model with missing attributes.

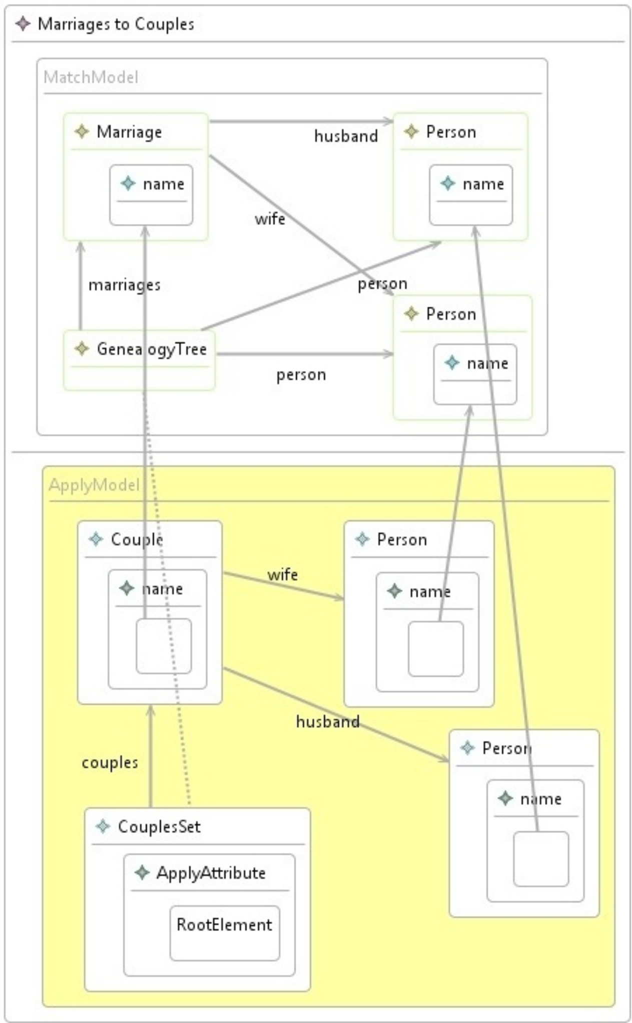


Figure 41: Rule with *MatchAttributes* and *AttributeRefs*.

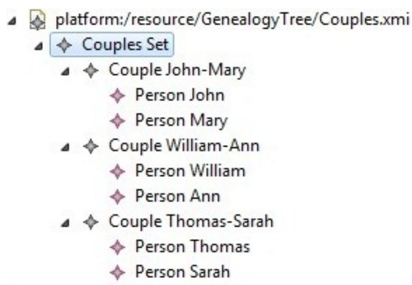


Figure 42: Final result.

3.8 Transformation Partitioning

In the previous section you learned how to build a simple transformation to generate a flat list of couples model out of a genealogical tree model. Now you will learn to do more than that: you will generate a hierarchical set of couples based on their age, i.e., the oldest couple will the the parent of all the other couples, and so on.

In order to build this transformation you will follow a slightly different approach: you will first transform each individual element, then you will look at groups of elements and create relations in the output model, thus connecting all the "loose" elements. Figure 43 gives an example of the processing stages of a transformation following this approach: first individual elements are considered, then it looks to bigger and bigger sets of elements; at the end, all the elements that where generated but are not "attached" to something are discarded.

Since you already have the required metamodels and an example model from previous section, all we have to do is to create a new *DSLTrans* transformation, add a *FilePort*, a *Rule*, the *MetaModelIdentifiers* needed to get a transformation like the one shown in figure 44.

Has described earlier, in this approach you first identify what each element in the input model *means* in the output model.

- Each *Person* in the *GenealogyTree* is a *Person* in *Couples*;
- Each *Marriage* can be seen as a *Couple*;
- The *GenealogyTree* element is the *CouplesSet* element.

With these three mappings you should create three simple rules in the first layer (see figure 45). Don't forget to set the *Package Name* properties for each *AnyMatchClass* and *ApplyClasses*.

Figure 46 shows the result of executing the layer you've just built. Notice that, internally, *DSLTranslator* keeps trace of the generated elements and generator elements. We call that *traceability links* (in the figure they are represented as dashed lines between generated and generator elements). This feature makes it possible to later match those elements and complete the transformation.

Now it is necessary to match the possible relations between elements in the input model and translate that to association (and sometimes new elements) in the output model. What does the relation of *husband* between a *Marriage* and a *Person* in the *GenealogyTree* mean? Insert a new *Layer* and all the elements needed to get it like the one shown in figure 47.

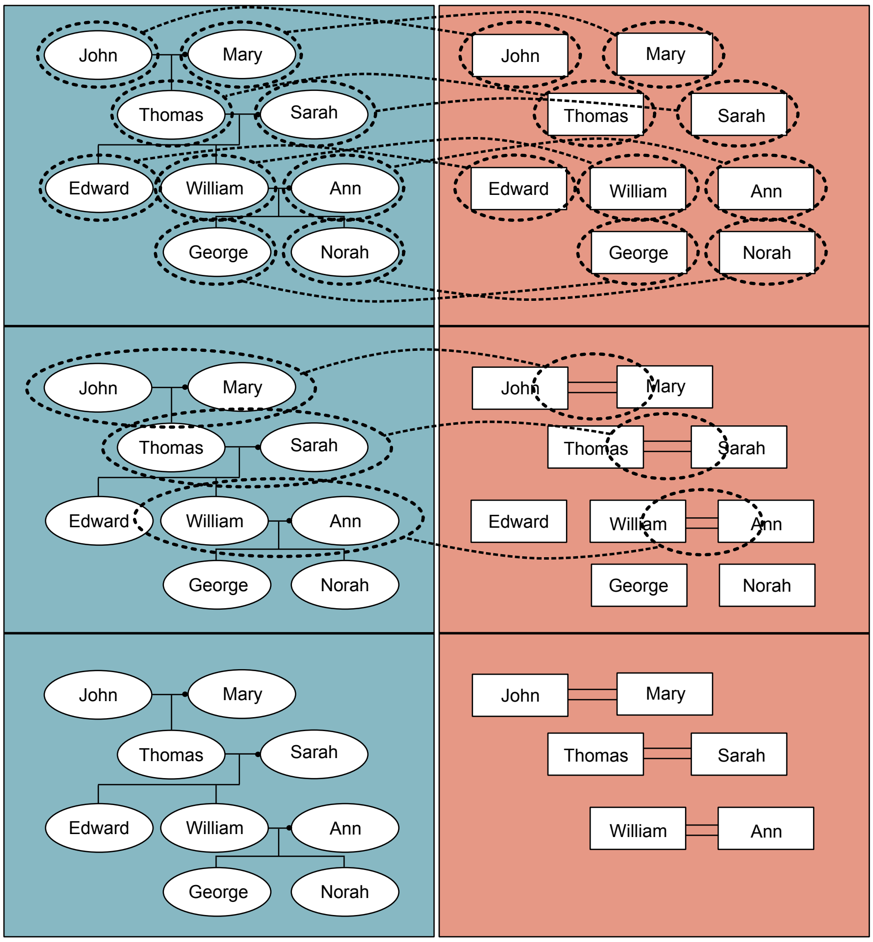


Figure 43: Transformation partitioning approach.

The first layer generates a set of loose elements in the output model, the second one connects people with couples as figure 48 illustrates. The only thing missing is to connect the couples in a hierarchical fashion so go ahead and build a third *Layer* connected to the second one and with the proper *MetaModelIdentifier* but without any rule.

How do you know that a couple is a *child* or a *parent*? If *John* is married to *Mary* and one (or more) of their children is married to someone else then *John's Marriage* is a parent of its children's *Marriages*.

The two rules shown in figure 49 express this concept. Notice that the two cases have to be considered since there are two ways of being in a marriage (husband or wife) in the *GenealogyTree* model.

After executing the transformation (with the rules shown in figure 49

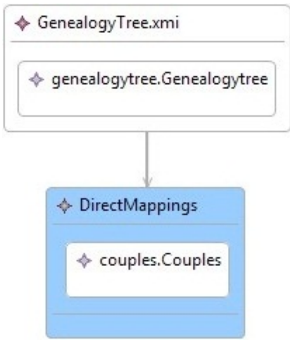


Figure 44: Basic transformation skeleton.

added) you will have as a result something like figure 50.

What about the oldest couple? According to the rules defined previously the oldest couple (in this case *John* and *Mary*) is not contained anywhere in the model. If you don't make a rule for this couple none of the other younger couples will be visible in the output model. Figure 51 shows the rule you need to add to the third layer in order to match the oldest couple and connect it to the *CouplesSet* element.

The rule in figure 51 matches a couple whose individuals (*husband* and *wife*) aren't children of anyone else. Notice the way to express a nonexistent class (and association) in *DSLTrans*.

After the execution of this last rule all the relevant elements are connected to the output model and hence, are displayed in the final result. The elements that are generated during the transformation (for instance, *George*, *Edward* and *Norah*) and are not (in)directly contained in the output model root element (in this case, the *CouplesSet* element) do not appear in the final result as shown in figures 52 and 53.

In the next sections you will be able to learn more about each element of the *DSLTrans* language individually. It is up to you to combine the elements in order to create almost any rule you need in a readable and elegant manner.

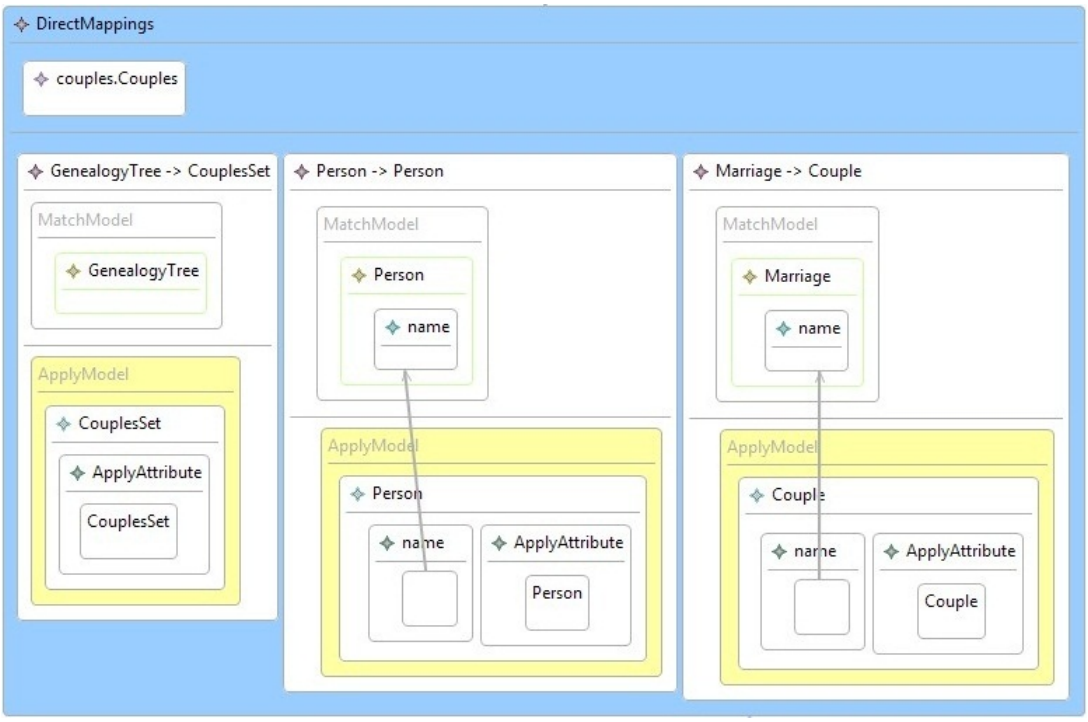


Figure 45: First layer direct mappings.

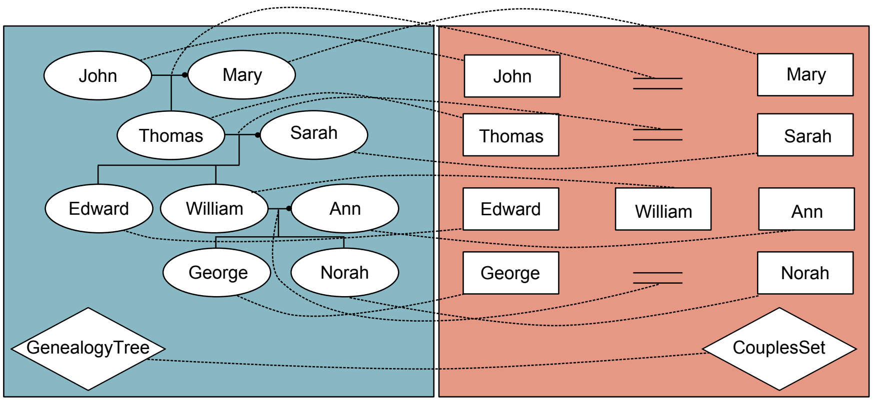


Figure 46: Resulting models after executing the mappings layer.

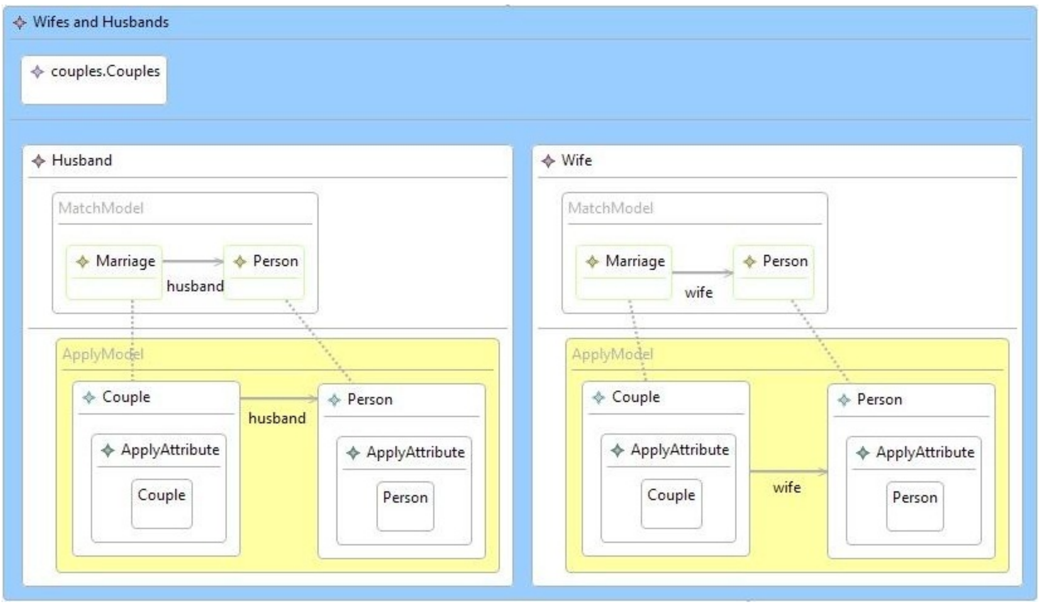


Figure 47: Husband and Wife relations layer.

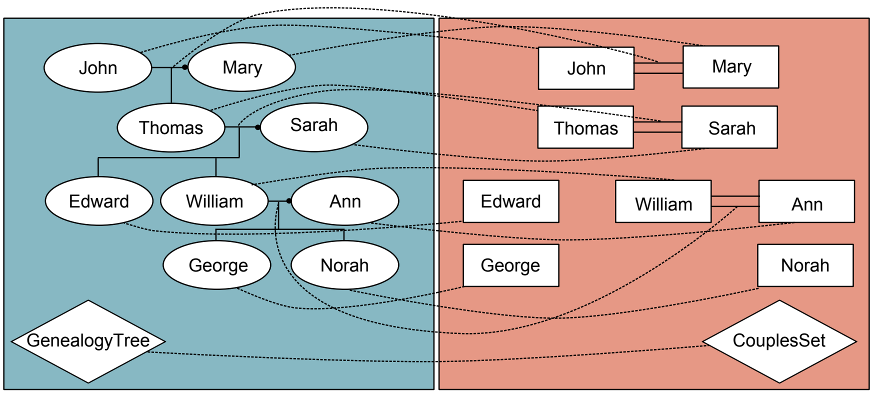


Figure 48: Resulting models after executing the second layer.

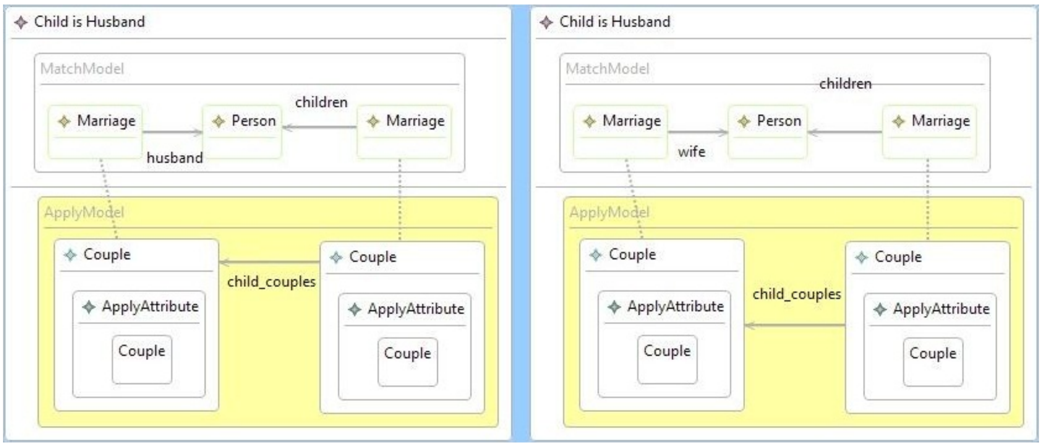


Figure 49: Couples hierarchy rules.

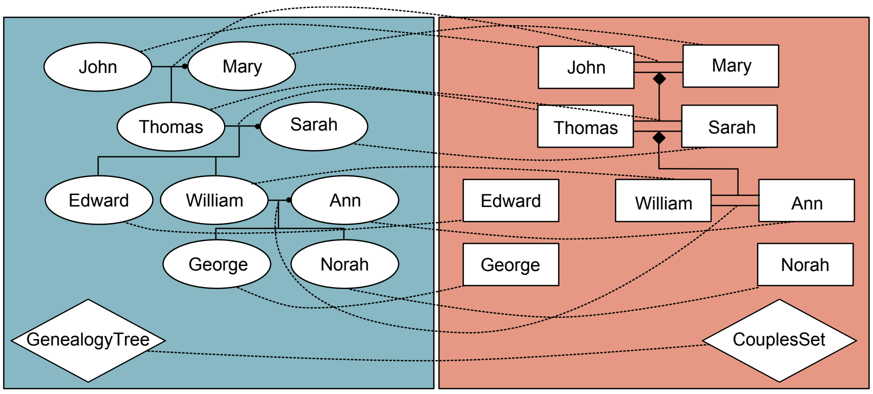


Figure 50: Resulting models after executing the third layer's two rules.

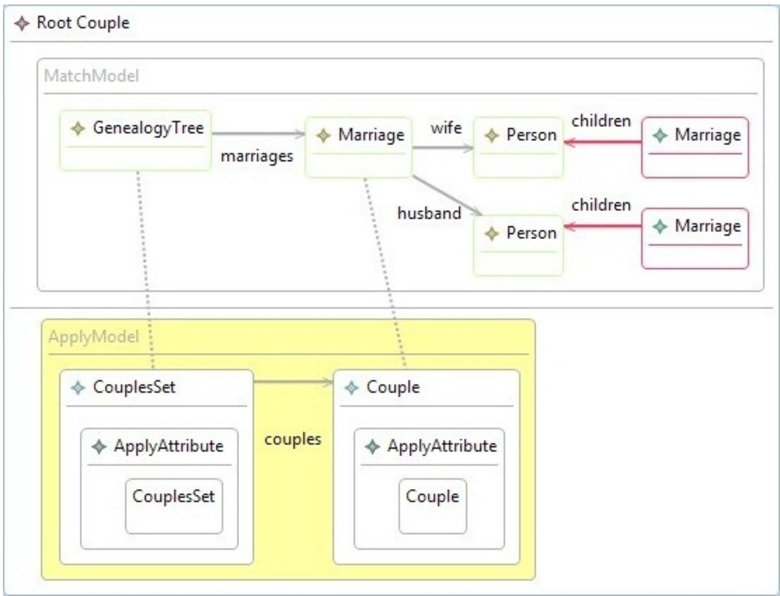


Figure 51: Root couple rule.

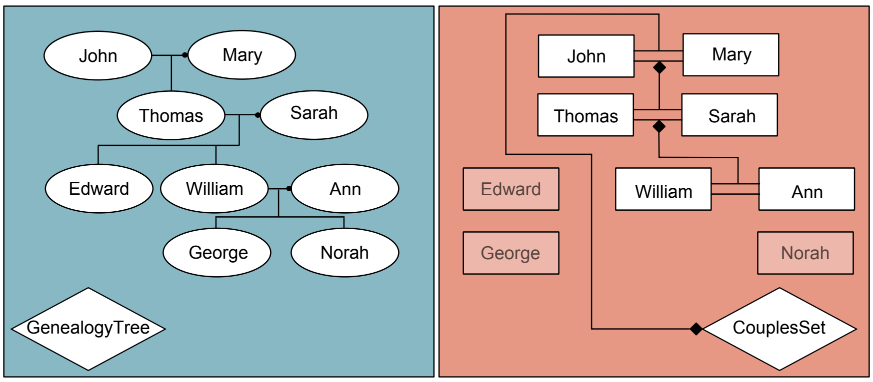


Figure 52: Resulting models after executing the transformation.

Figure 53: Resulting *XMI* file.

4 Language Definition

4.1 A Typical Transformation

Most *DSLTrans* transformations have a common subset of elements. Figure 54 shows some of those. Usually there is one *FilePort* that points to some input model *XMI* file and contains a *MetaModelIdentifier* that references the metamodel of the input model so *DSLTrans* can validate the input. Then there are multiple *Layers*, connected using a *PreviousSource* association. Each *Layer* can have an output model and must have a *MetaModelIdentifier* and various *Rules*. Every *Rule* has a *MatchModel* and an *ApplyModel*, each with the match and the apply pattern respectively.

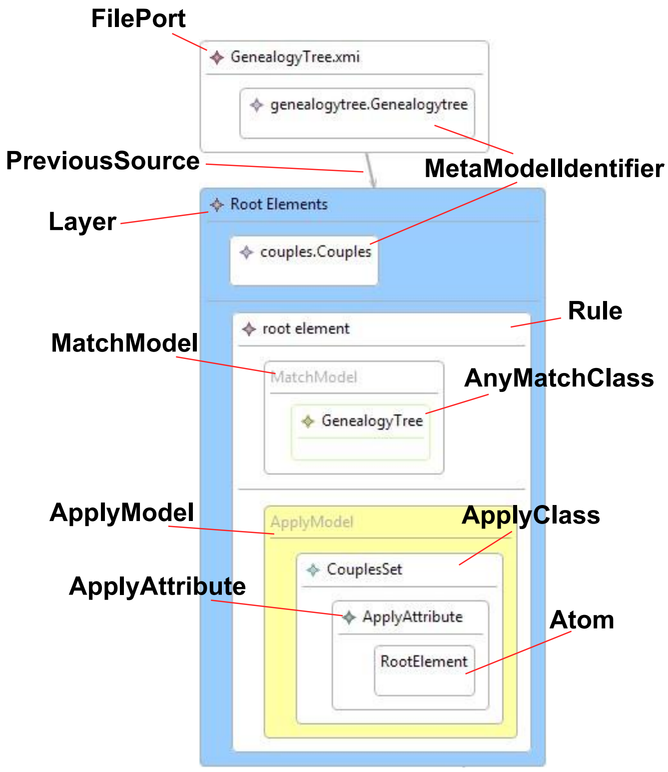


Figure 54: Example transformation structure.

4.2 Language Constructs

Bellow is the description of each *DSLTrans* element along with its representation in both visual and textual concrete syntaxes.

4.2.1 Objects

AnyMatchClass The *AnyMatchClass* is used in a *MatchModel* to capture *all* the elements in the input model. When used within a more complex pattern the set of matched elements can be reduced. Figure 55 shows an example where all the *Marriage* elements are being captured and in figure 56 only those whose attribute *name* has the value *Thomas-Sarah* are matched.

Property	Description
Class Name	Type or Class of the element to be matched.
Description	A meaningful description should be used for documentation purposes.
Package Name	This is a very important property that should always be correctly set. You can find the correct value by looking to the corresponding metamodel's root package as shown in figure 57.

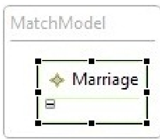


Figure 55: AnyMatchClass example.

ApplyAttribute *ApplyAttributes* are inserted inside *ApplyClasses* either to specify an attribute value or to capture a previously generated element with some attribute value (if used in an *ApplyClass* connected with a *PositiveBackwardRestriction*). Figure 69 shows one *ApplyAttribute* with no name specified. This is usually used to tell *DSLtranslator* to keep traces in memory so that the generated element (in this case, a *Couple*) can be later referenced.

Property	Description
Attribute Name	Name of the attribute to be applied.
Description	A meaningful description should be used for documentation purposes.

Figure 56: AnyMatchClass example combined with MatchAttribute and Atom.

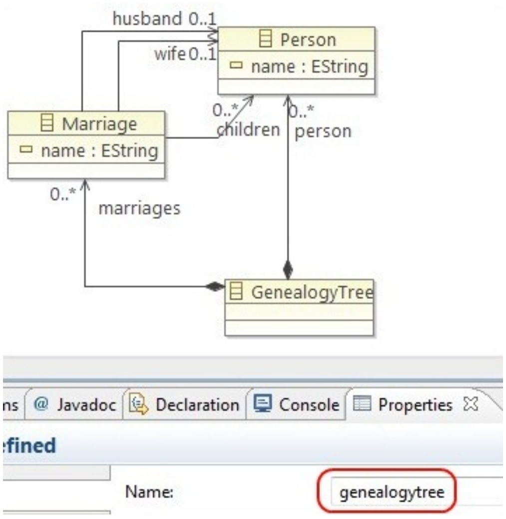


Figure 57: Root package name property.

ApplyClass The *ApplyClass* is used to created new elements in the apply patterns or match previously generated elements (if used with a *PositiveBackwardRestriction*).

Figure 69 shows an *ApplyClass* named *Couple*.

Property	Description
Class Name	Type or Class of the element to be applied.
Description	A meaningful description should be used for documentation purposes.
Group Name	This property helps you to organize your *ApplyClasses* by groups if you want.
Package Name	This is a very important property that should always be correctly set. You can find the correct value by looking to the corresponding metamodel's root package as shown in figure 57.

ApplyModel An *ApplyModel* is just a container for the pattern to apply in case a match is found. Figure 65 shows one.

4.2.2 Atom

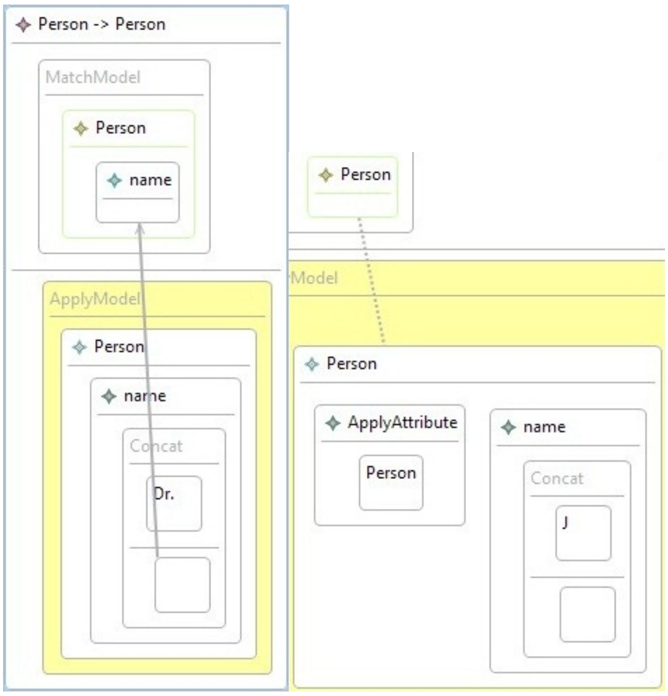
Atoms are usually used inside *MatchAttributes* and *ApplyAttributes* to express arbitrary values. Figure 61b shows an *Atom* inside a *MatchAttribute* and figure 61a an *Atom* combined with an *ApplyAttribute*.

Property	Description
Value	Value that the *Atom* represents.

AttributeRef The *AttributeRef* element is used to copy some attribute value from a matched element to an applied one. Figure 58a shows an *AttributeRef* inside the second part of a *Concat* together with the *AttributeRef (Connection)* that points to the attribute being copied.

Concat The *Concat* concatenates *Atoms*, *AttributeRefs* and *WildCards* inside *ApplyAttributes* allowing for a flexible value manipulation. Figure 58a shows a *Concat* element combined with an *Atom* and an *AttributeRef* to give the "Dr." title to every *Person*. In figure 58b shows a complex apply pattern that captures all *Person* elements that generated new *Person* whose name starts with "J". It combines a *Concat* with an *Atom* and a *WildCard*.

ExistsMatchClass As the *AnyMatchClass* element, the *ExistsMatchClass* is also used to create match patterns but it only cares about finding one element, not all of them. It can be combined with *MatchAttributes* to further refine the element to be matched. Figure 59b shows an example of a pattern with an *ExistMatchClass*. Beware that when combining an *ExistMatchClass*



(a) Concat with AttributeRef example. (b) Concat and wildcard example.

Figure 58

and an *AnyMatchClass*, the *AnyMatchClass* will always prevail over the *ExistMatchClass* no matter what the direction of the association between them. For instance, in pattern 59b the intention is to capture *every Marriage* with a *wife*, not *only one Person* that is a *wife* in every *Marriage*.

Property	**Description**
Class Name	Type or Class of the element to be matched.
Description	A meaningful description should be used for documentation purposes.
Package Name	This is a very important property that should always be correctly set. You can find the correct value by looking to the corresponding metamodel's root package as shown in figure 57.

FilePort The *FilePort* element represents an input model. A transformation can have multiple *FilePorts* if it uses multiple input models. An input

(a) Two AnyMatch-Classes

(b) Equivalent pattern using an AnyMatchClass and an ExistMatchClass.

Figure 59

model always has to conform to a metamodel, that is why the *FilePort* always contains a *MetaModelIdentifier* element to tell *DSLTranslator* which metamodel the input model conforms to. Figure 60 shows an example of a *FilePort* and its *MetaModelIdentifier*.

Property	Description
File Path URI	A meaningful name for the current input.

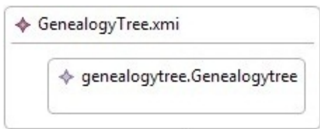
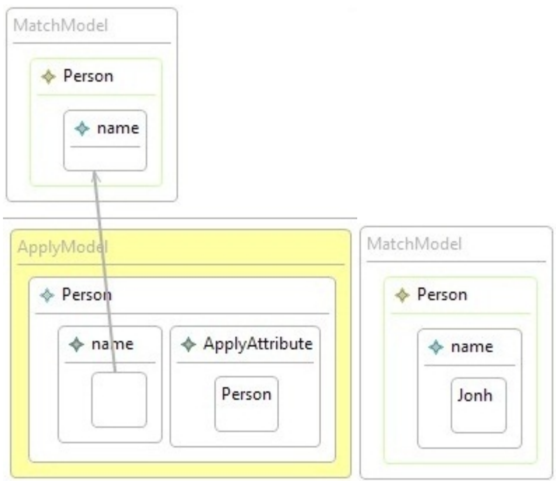


Figure 60: FilePort example.

Layer *Layers* establish an order to the transformation execution. A transformation can have several sequential *Layers* or even parallel ones if its purpose is to produce more than one output model. Each *Layer* has a *PreviousSource* association that connects it to another *Layer* or a *FilePort*. Figure 63 shows an example of a *Layer*.

Property	Description
Description	Here you write a brief description on what the *Layer* is supposed to do.
Group Name	This property helps you to organize your *Layers* by groups if you want.
Name	A symbolic name for the *Layer*.
Output File Path URI	The relative or absolute path for the resulting model of the current *Layer*.
Previous Source	This property is automatically filled if you insert the *PreviousSource* connection but if you prefer you can set it manually by writing the name of the previous *Layer* or *FilePort* here.

MatchAttribute The *MatchAttribute* element is used when it is necessary to capture some element's attribute value or to match an element with a specific attribute value. Figure 61b shows the *MatchAttribute* element being used to say that only the *Person* elements whose name is John are matched and figure 61a illustrates a way to copy an attribute value between match and apply elements by combining the *MatchAttribute* with *ApplyAttribute* and *AttributeRef*.



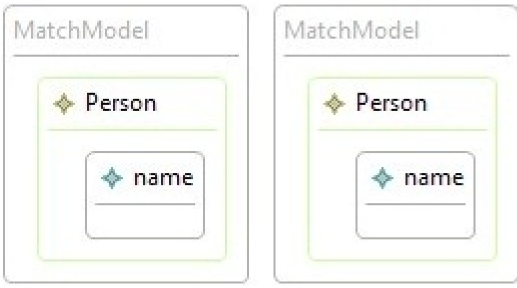
(a) MatchAttribute with an AttributeRef example

(b) MatchAttribute with an Atom example.

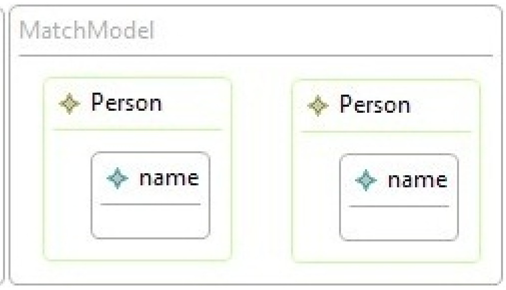
Figure 61

Property	Description
Attribute Name	Name of the attribute to be matched.
Description	A meaningful description should be used for documentation purposes.

MatchModel The *MatchModel* contains a *Rule's* match pattern (see figure 65). There can be multiple *MatchModels* in the same *Rule* although it is rare: one can actually override the *PreviousSource* connection of a *MatchModel* by setting it's *ExplicitSource* property or by creating an *ExplicitSource* connection to some *FilePort*. Figures 62a and 62b illustrate two equal patterns but the left one is split across two *MatchModels*. This does not seem very useful and it isn't but when combined with the *ExplicitSource* property lets you parametrize transformations (see section 5.1).



(a) Two separated MatchModels



(b) Equivalent pattern in a single MatchModel.

Figure 62: MatchModel examples.

Property	Description
Explicit Source	Name of a *FilePort* to get an input model from.

MetaModelIdentifier The *MetaModelIdentifier* element is used inside *FilePorts* and *Layers* to refer to the relevant metamodels. Wherever there is an input or output model, the *MetaModelIdentifier* has to be there. Figure 60 shows a *MetaModelIdentifier* inside a *FilePort* and figure 63 illustrates it in a *Layer* because a *Layer* can generate an output model.

Property	Description
Meta Model Name	Specifies the name of the metamodel. This name usually takes the form of the `root_package_name.Root_package_name` as you have seen in section 3.
Meta Model URI	The relative or absolute path to the metamodel.

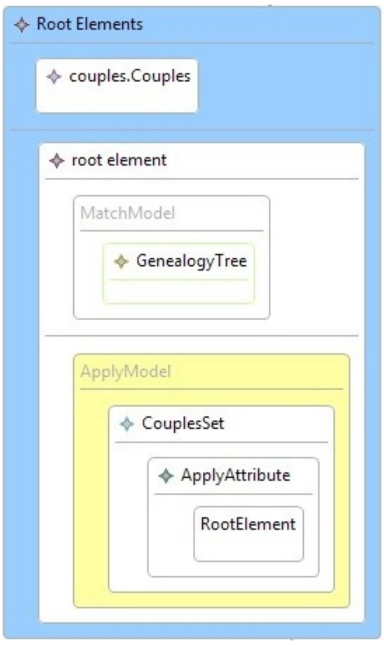


Figure 63: Layer with MetaModelIdentifier example.

NegativeMatchClass The *NegativeMatchClass* is mostly used in combination with a *NegativeMatchAssotiation* to express that you don't want an element to exist in some pattern. In figure 64 the pattern captures all *Person* objects that are not children, i. e., it will match all root elements.

Property	Description
Class Name	Type or Class of the element to be matched.
Description	A meaningful description should be used for documentation purposes.
Package Name	This is a very important property that should always be correctly set. You can find the correct value by looking to the corresponding metamodel's root package as shown in figure 57.

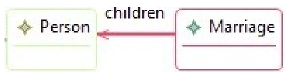


Figure 64: NegativeMatchClass example.

Rule *Rules* are inserted inside each *Layer* and they contain a match side and an apply side. Figure 65 shows an example rule already filled with some elements. A *Rule* always needs to contain at least a *Match* and an *Apply* models.

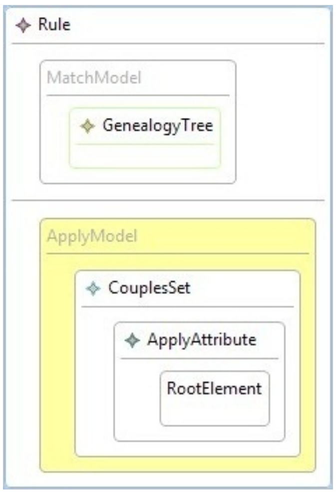


Figure 65: Example rule containing a MatchModel, ApplyModel, AnyMatchClass and ApplyClass along with attributes.

Property	Description
Description	Use this property to describe the purpose of the rule if you want.

Wildcard *Wildcards* are used most frequently inside *ApplyAttributes*, combined with *Atoms* and *Concats* to restrict the number of matched elements that where previously generated[7]. Figure 58b shows an example of a pattern that will only be applied to *Person* elements that where previously generated and whose name starts with a "J". A *Wildcard* represents any value.

4.2.3 Connections

ApplyAssociation *ApplyAssociations* always generate relations between *ApplyClasses* in the output model. Notice that they cannot be used to capture previously generated elements as *ApplyAttributes* can do. Figure 70 shows an *ApplyAssociation* between a *CouplesSet* and a *Couple*.

Property	Description
Association Name	The name of the association. This depends on the input metamodel.

AttributeRef The *AttributeRef* connection points to a *MathAttribute* to copy its value (see figure 58a).

ExplicitSource The *ExplicitSource* allows the user to connect a *MatchModel* directly to a *FilePort* and match a pattern against an input model. Figure 66 shows an example of an *ExplicitSource* connection (it's the thin line between the *MatchModel* and the *FilePort*).

Import Using the *Import* element, the user is capable of copying an entire tree of elements from an input model to an output model by matching the tree's root element. *DSLTranslator* will copy the element along with its attributes and descendants, keeping all the relations that belong to the tree. Beware that if any of the imported elements has an association referring other element that does not belong to the tree (i. e., is not a descendant of the root matched element), that connection will cease to exist. Naturally, all imported elements must conform to the same metamodel as the output model. For instance, if you already had a *CouplesHierarchy* model (like the one shown in figure 67) and you wanted to create a new model (as the one in figure 68) that extends it with information from a *GenealogyTree* model , you could use an *Import* to copy the *Frank-Basie* couple along with its child couples and then attach the imported tree to the output model like figures

[7]This means the *ApplyClass* has to be connected to some match class with a *Positive-BackwardRestriction*.

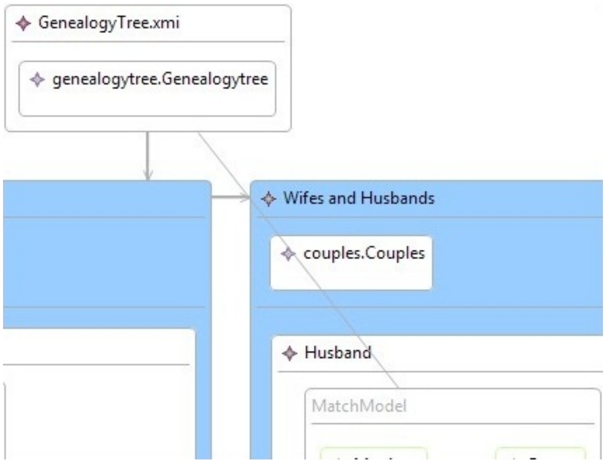


Figure 66: ExplicitSource example.

69 and 70 illustrate. The *MatchModels* that capture the imported elements are directly connected to *FilePort* that points to the existing *Couples* model shown in figure 67.

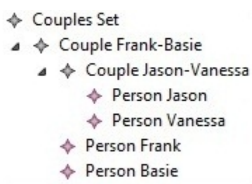


Figure 67: Couples Hierarchy model.

NegativeIndirectAssociation The *NegativeIndirectAssociation* is used to create patterns where a containment association of any depth between two connected elements cannot exist. It usually is combined with a *NegativeMatchClass* to capture elements that are not contained in other elements by any depth. Figure 71 shows a pattern with the same meaning of the one shown in figure 72 but disregarding the containment association's name.

Property	Description
Association Name	This can be any name you want.

NegativeMatchAssociation As the *PositiveMatchAssociation*, the *NegativeMatchAssociation* is used to connect match classes allowing for more

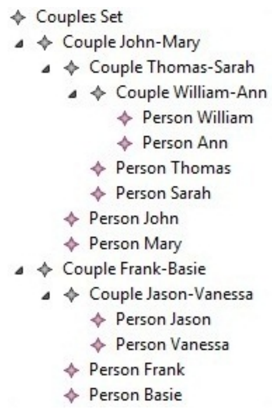


Figure 68: Couples Hierarchy extended model.

complex match patterns. Unlike the *PositiveMatchAssociation*, it expresses that an association must not exist between two elements and it is often combined with a *NegativeMatchClass* to say that an element cannot exist in some pattern. Figure 72 shows an example where *Persons* (and respective *Marriages*) that have no parents are being matched.

Property	Description
Association Name	The name of the association. This depends on the input metamodel.

PositiveBackwardRestriction The *PositiveBackwardRestriction* association is used to generate associations between output model elements generated in previous layers. It connects match elements to apply elements in order to match the generated and generator elements. For instance, in figure 73 a new relation named *husband* is being created between any *Couple* and *Person* that were previously created and whose creators (*Marriage* and *Person*) are related by the *husband* association.

PositiveIndirectAssociation The *PositiveIndirectAssociation* is used to abstract long containment relationships[8] between two elements. For instance, in the *GenealogyTree* metamodel, the association *marriages* between a *GenealogyTree* and a *Marriage* is a 1-level containment and the pattern shown in figure 74a matches the two elements. But what if the metamodel allowed *Marriages* inside *Marriages* by adding an containment association between

[8]Long containment relationships mean that there can be several elements in between.

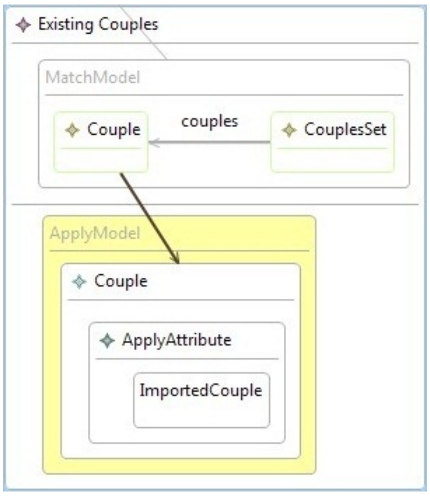


Figure 69: Import existing couples tree rule.

Marriage elements? The resulting models could have long lists of *Marriages* inside *Marriages*, all connected with containment relationships. In that scenario, the pattern shown in figure 74b would match all *Marriages* inside the *John-Mary Marriage*. Notice that all containment relations (of any depth) will be matched, even if they haven't got the same name.

Property	Description
Association Name	This can be any name you want.

PositiveMatchAssociation It is often necessary to create match patterns with more than one element. The *PositiveMatchAssociation* is a possible connection between two match classes that expresses that a relation has to exist between those elements in the input model. In figure 75 a *Person* that is a wife and a child simultaneously is being matched; on the other hand, the rule will not be applied to a *Person* that is not a child.

Property	Description
Association Name	The name of the association. This depends on the input metamodel.

PreviousSource *PreviousSource* is an association that connects a *Layer* to another *Layer* or *FilePort*. It controls the flow of the transformation. Figure 76 shows two *PreviousSource* connections.

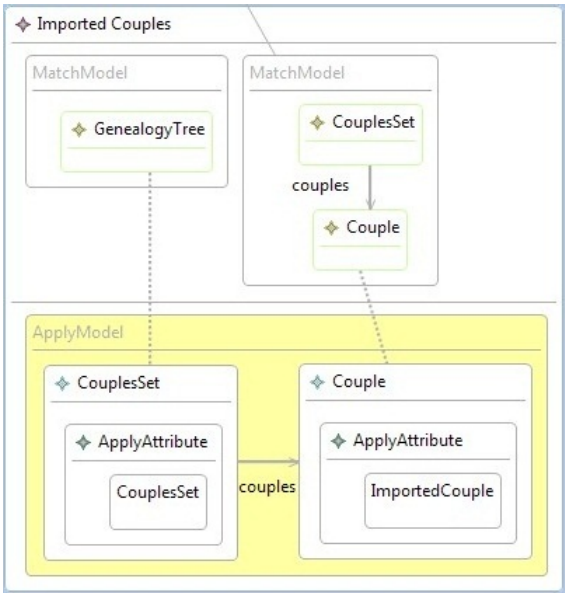


Figure 70: Connect imported couples tree rule.

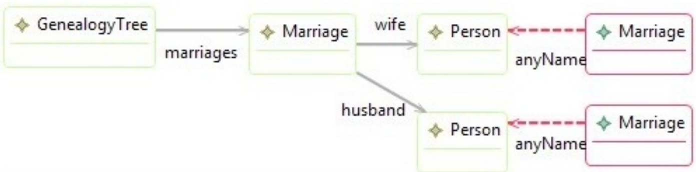


Figure 71: Negative Indirect Associations together with Negative Match Classes.

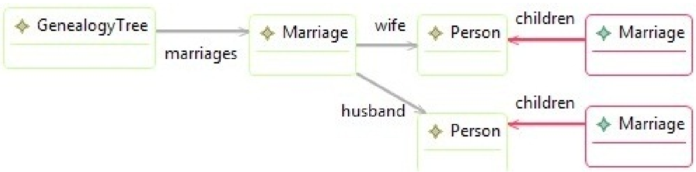


Figure 72: Negative Match Associations together with Negative Match Classes.

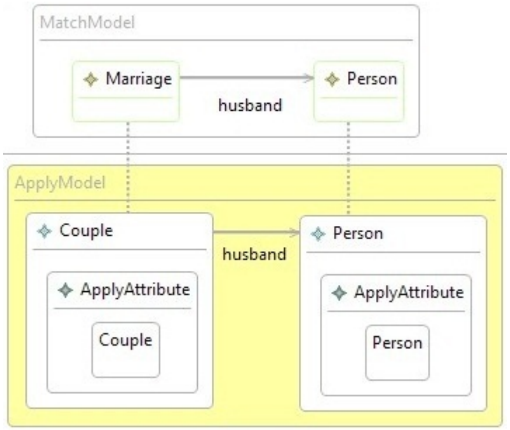
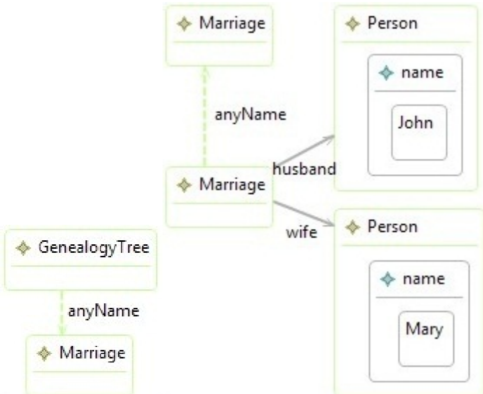


Figure 73: Positive Backward Restriction example.



(a) Positive Indirect Association between GenealogyTree and Marriage.

(b) Positive Indirect Association between Marriages.

Figure 74

Figure 75: PositiveMatchAssociation examples.

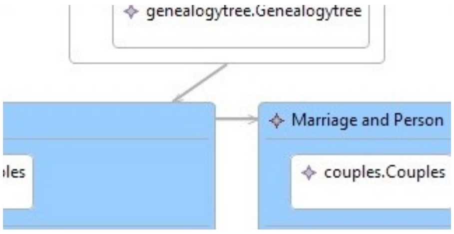


Figure 76: PreviousSource connections between Layers and FilePort.

5 Advanced Topics

Models can be used to describe the dynamic aspects of a system and transformations can be built in order to simulate the changes in that system along it's lifetime. Thus, model transformations can be viewed as a kind of declarative programming where a set of rules define computations as changes in the information present in the system model [5].

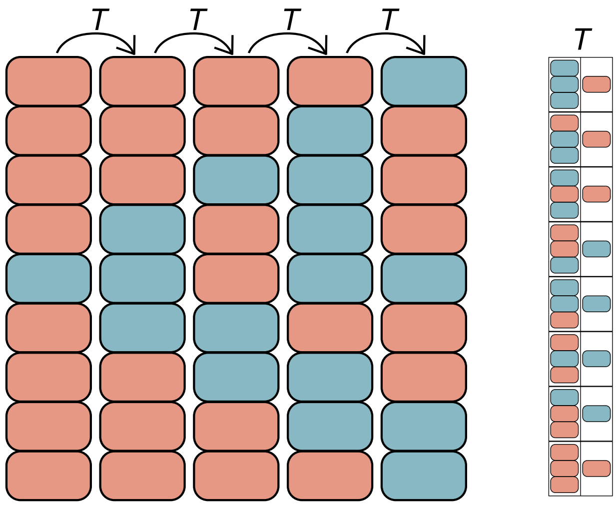


Figure 77: Executing a system by applying the same set of rules T several times.

In figure 77 an example of the changes occurred in a system by applying the same set of transformations is shown. In this particular case, the system is represented as the tape of a cellular automaton[9] and it's behaviour is defined by the set of rules T. The next color of a matched cell is defined according to its own color and its neighbour's. Figure 77 shows the state of the cellular automaton's tape across four transformations. Curiously, with the set of rules T, if you look to several more transformation applications (with a bigger tape than the one shown in the figure) you will see that no pattern arrises in the automaton's behaviour [16].

As you have seen in section 1, *DSLTrans* transformations are no more than models conforming to the *DSLTrans* metamodel. If *DSLTrans* allows one to

[9]In the context of this manual, a cellular automaton is a abstract device with an infinite tape divided in cells that can have two colors. Its behaviour is defined by means of transformation rules involving a cell and its nearest neighbours.

create model transformations, then why can't one build a transformation that handles transformations? In fact, it is perfectly possible and opens a wide range of possibilities as you will see in this section.

5.1 Finite Deterministic Automata Execution

In this section an example of how a transformation can be used to simulate the behaviour of an abstract mathematical system.

A representation of a Finite Deterministic Automaton (FDA) is shown in figure 78 and consists of a reader that crosses a tape in one way, in this case, from left to right, reading a symbol at a time and, depending on the symbols read, it will accept (or not) the sequence present in the tape. Figure 79 shows an automaton that has accepted the sequence read from the tape.

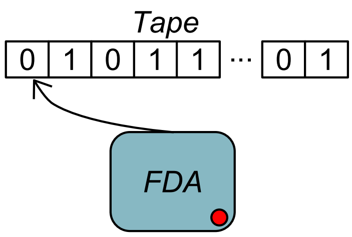


Figure 78: A representation of an automaton system with it's tape and current pointed cell. The red light indicates that the automaton has accepted the sequence read yet.

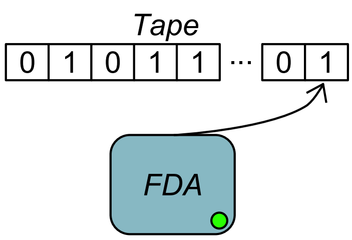


Figure 79: A representation of a FDA that has accepted the sequence read from the tape.

The acceptance criteria of an automaton can be defined by a labelled graph (hence, a model) with multiple states, an initial state and final/acceptance states. Figure 80 shows a specification of an automaton that accepts only sequences with an even number of 1's.

From a transformation point of view, each step of an automaton execution depends on the current state, the current symbol read from the tape and the

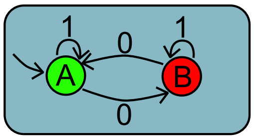


Figure 80: The behaviour of an automata based on a labelled graph. There are two states, A and B, and four transitions, each occurring depending on the current state and the current read symbol (0 or 1).

transitions available at the current state. The result is a new automaton with the same states and transition from the previous one but with a new current state (that is the target of the executed transition) and a different current symbol. For more information about automata refer to [12].

In order to show how *DSLTrans* can be used to execute any possible FDA models are needed to represent the state of the automaton across its states. We can either model the tape and the automaton together, or have two separate models: one to represent the tape and the other the automaton. In this example we will opt to follow the second approach. In a latter section (5.2) we use a single model.

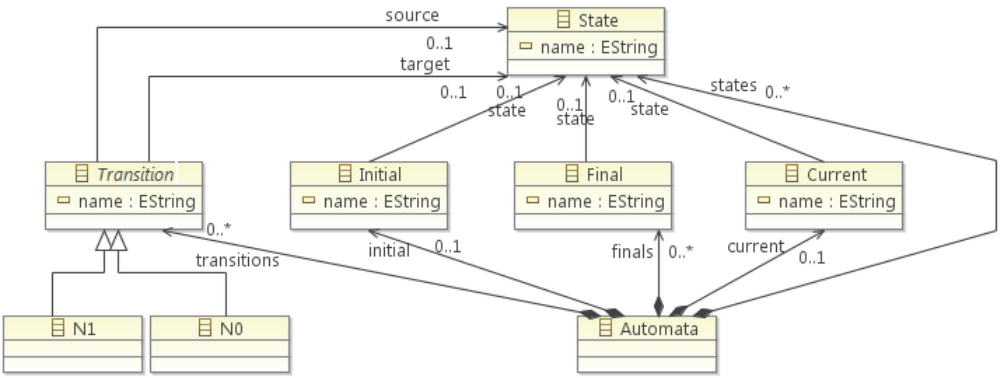


Figure 81: Automata Metamodel.

The metamodel for the automaton is shown in figure 81. It has various states, transitions with labels that are 1 or 0. Pointers are needed to represent to the initial, current and final (acceptance) states. These pointer could be attributes of a state but the last are more difficult to change in a transformation but it is perfectly doable.

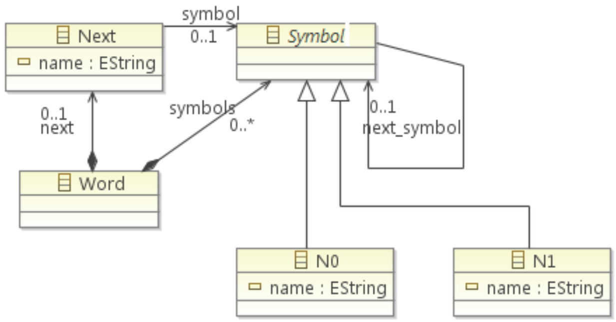


Figure 82: Word (a.k.a. Tape) Metamodel.

Figure 82 shows the metamodel used to define a word to be read by the automaton. Notice that to keep a relation of order between the symbols (0 or 1) a *next_symbol* relation is used. The *Next* pointer indicates the symbol to be read by the automaton in the next step.

In both metamodels, the pointers, states and transitions have names so the models are more readable, they have no influence in the automaton behaviour.

The transformation has two independent flows as can be seen in its outline in figure 83. That makes sense since there are two models that have to be changed and each flow applies those changes to each model. The tape will have it's *Next* pointer changed according to the automaton and the automaton will have it's *Current* state changed according to the tape, it's current state and the available transitions.

The complete transformation is in the files that come with this manual, please refer to them in the next paragraphs.

Since only the pointers of each model will change and the rest of the elements have to remain intact from the input to the output, the first layer of each flow has the mappings for the elements that remain the same, and copies their attributes.

The second layer of the left flow (the one that changes the word model) has four rules: two of them keep the consistency of the model (order of elements and their connection to the root element) and the other two change the *Next* pointer referring to the current automaton state and the available transitions.

In the right side of the transformation, the second layer has several rules but only two of them actually add any dynamic behaviour to the automaton since the other ones exist only to keep the consistency between input and output models. In those two rules, the *Current* pointer is set according to the current state of the automaton, it's transitions and the symbol read from the word model.

Figures 84 and 85 show the main rules that define the behaviour of this system. All the other rules and layers exist so that the output model remains the same as the input model (except for its pointers).

To see how the models change, figures 86 and 87 show the automata and the word models before and after two transformation executions.

5.1.1 Conclusions

Apart from the automaton behaviour simulation using *DSLTrans*, the main ideas to retain from this section are:

- How to execute two transformations simultaneously using the *PreviousSource* association and different *FilePorts*.
- How to use the *ExplicitSource* association of a *MatchModel* to control which rules are applied according to an external model. This technique is used in the rules shown in figures 85 and 84.
- How to take advantage of class hierarchies in models to reduce the set of rules needed when matching previously generated elements. An example of this technique is shown in figure 88 where the left side shows a concrete *Symbol* (*N1*) being generated and the right side shows a rule with a *BackwardLink* matching any *Symbol* generated previously.

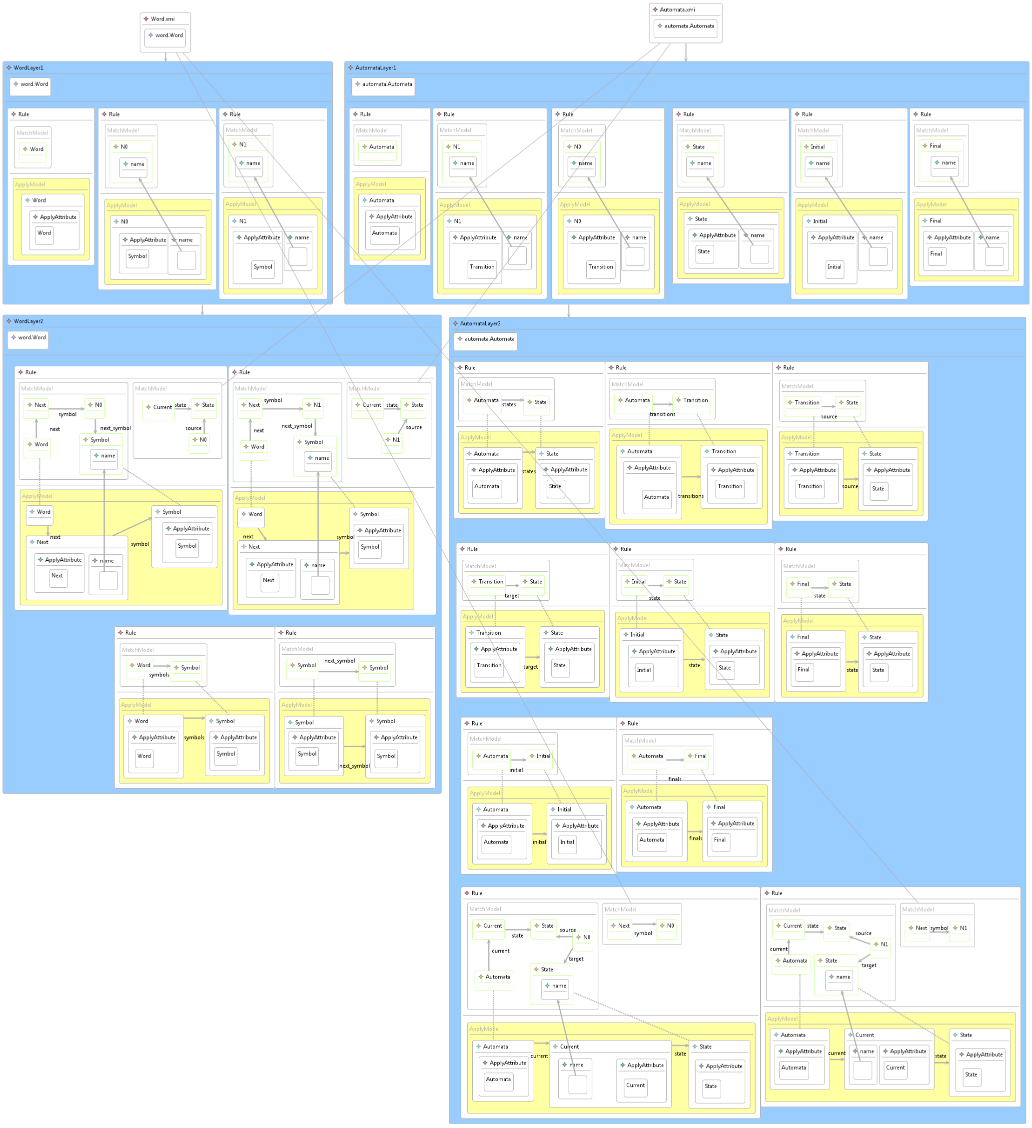


Figure 83: Automata execution transformation outline.

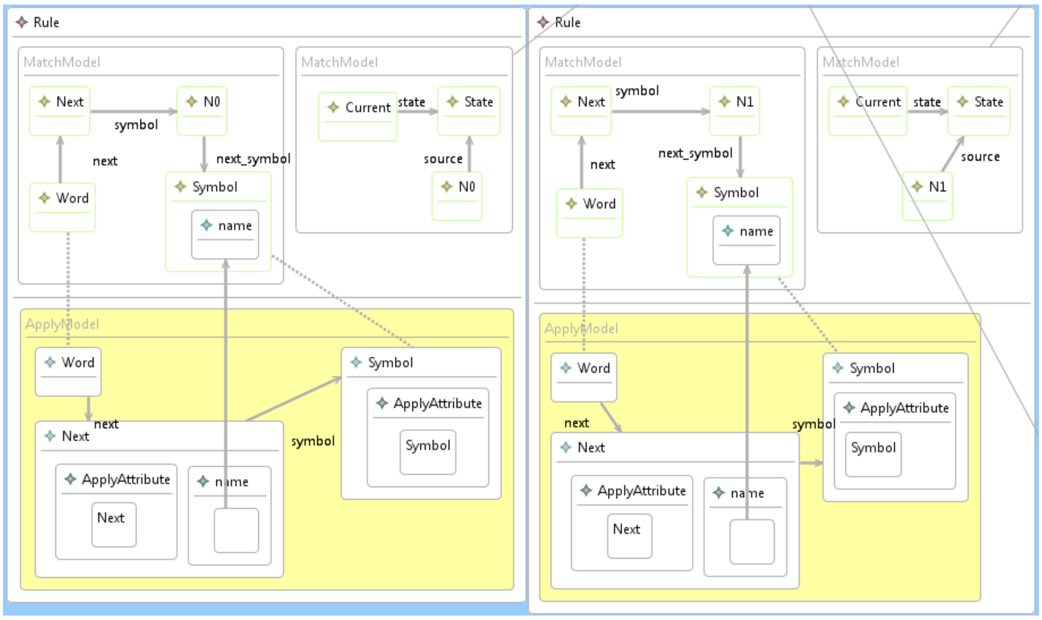


Figure 84: The two Word rules the define the *Next* pointer.

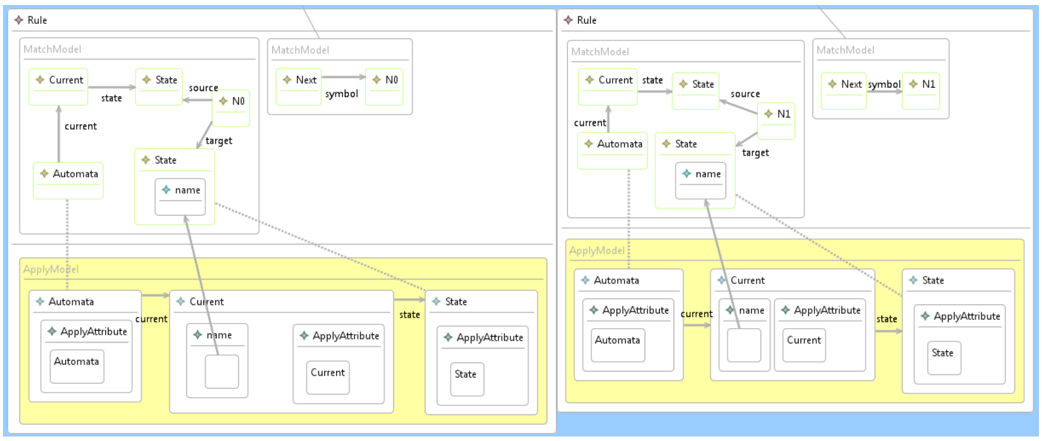
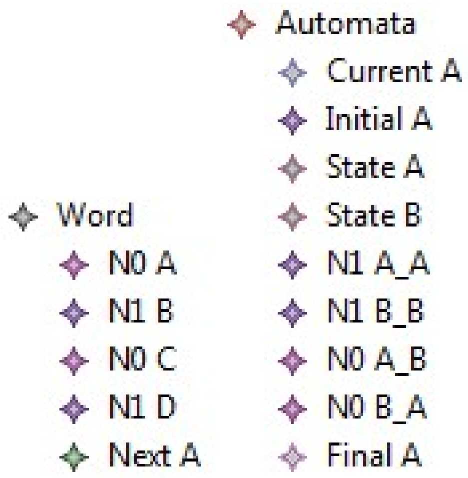
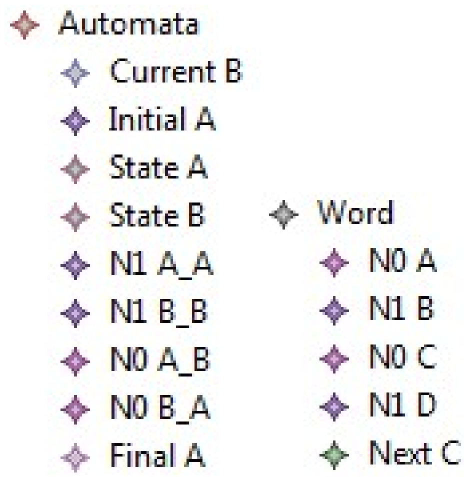


Figure 85: The two Automata rules the define the *Current* state pointer.



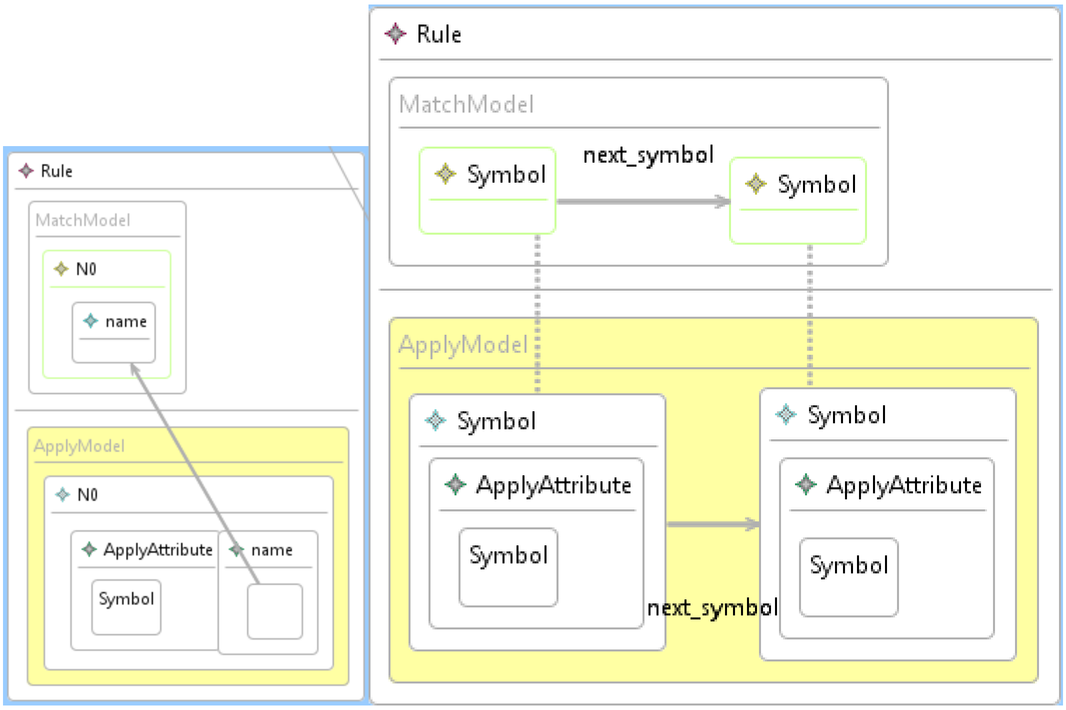
(a) (b)

Figure 86: Initial system state.



(a) (b)

Figure 87: Final system state.



(a) Rule in the first layer. (b) Rule in the second layer.

Figure 88: Using class hierarchy and *BackwardLinks* to reduce the set of rules needed.

5.2 Turing Machine Step Transformation

To be completed.

5.3 High Order Transformations

To be completed.

5.4 Prototyping Transformations

To be completed.

5.4.1 Identity Generation

To be completed.

5.4.2 Fixed Identity Generation

To be completed.

ïz£

6 FAQ and Common Installation Problems

6.1 SWI-Prolog: [FATAL Error: Could not find system resources]

This error occurs when you one (or more) of the following steps in the section 2:

- Install swi prolog.
- Add the bin directory of prolog to the *Path* variable.
- Set the *SWI_HOME_DIR* system variable.
- Copy the jpl.jar file to the appropriate destination.

References

[1] Epsilon transformation language. `http://www.eclipse.org/gmt/epsilon/doc/etl/`.

[2] Daniel Balasubramanian, Anantha Narayanan, Chris vanBuskirk, and Gabor Karsai. The graph rewriting and transformation language: Great.

[3] Jean BÃl'zivin. On the unification power of models.

[4] JesÃžs SÃąnchez Cuadrado, JesÃžs GarcÃ■a Molina, and Marcos Menarguez Tortosa. Rubytl : A practical, extensible transformation language.

[5] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation.* Springer, 2006.

[6] Roberto Felix, Bruno Barroca, Vasco Amaral, and Vasco Sousa. Dsltranslator: providing tool support for language's transformational semantics.

[7] Object Management Group. Xml metadata interchange. `http://www.omg.org/spec/XMI/`.

[8] I.Arrassen, A.Meziane, R.Sbai, and M.Erramdani. Qvt transformation by modeling - from uml model to md model.

[9] FrÃl'dÃl'ric Jouault, Freddy Allilaire, Jean BÃl'zivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(1-2):31 – 39, 2008. Special Issue on Second issue of experimental software and toolkits (EST).

[10] Audris Kalnins, Janis Barzdins, and Edgars Celms. Model transformation language mola.

[11] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development.

[12] Michael Sipser. *Introduction to the Theory of Computation.* Boston, 1997.

[13] Victor SÃąnchez. Atc user guide. `http://www.modelset.es/atc/atcdownload.html`.

[14] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. Emf eclipse modeling framework.

[15] Edward D. Willink. Umlx : A graphical transformation language for mda.

[16] Stephen Wolfram. *A New Kind of Science*. 2002.