Universität Paderborn

Fakultät für Elektrotechnik,
Mathematik und Informatik

Studienarbeit

A Bidirectional Transformation between EMF Models and Typed Graphs

Thomas Rheker

13.8.2008

vorgelegt bei
Prof. Dr. Gregor Engels

Zweitgutachter
Prof. Dr. Franz Josef Rammig

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbst angefertigt und keine anderen als die angegebenen und bei Zitaten kenntlich gemachten Quellen und Hilfsmittel benutzt habe.

Thomas Rheker

Abstract

Dynamic Meta Modeling (DMM) allows to expand meta modeling to the semantics of a model. By transforming Eclipse Modeling Framework (EMF) models to typed graphs, the results of this thesis allow to use the Groove toolset to compute graph transition systems for the EMF models. Using Groove and a corresponding ruleset, we can then check the transition system for interesting states. This thesis also presents the reverse transformation from Groove to EMF which could be used to transform these interesting states back and inspect them inside the Eclipse platform.

Contents

1 Introduction **2**

2 Fundamentals **3**
2.1 Dynamic Meta Modeling 3
2.2 Eclipse Modeling Framework 4
2.3 Groove 9

3 Transforming EMF to Groove **10**
3.1 Duplicate Class Names 16
3.2 Enumerations 18
3.3 Ordered References 19

4 Transforming Groove to EMF **21**
4.1 Resolving Duplicate Class Names in Groove Graphs 26
4.2 Setting the Values of EEnum Attributes 26
4.3 Resolving Ordered References in Groove 26

5 Testing the Transformations **27**
5.1 Testing the Transformation of Attributes 28
5.2 Testing the Transformation of References 28
5.3 Testing Duplicate Class Names 30
5.4 Other Tests 31

6 Conclusion and Outlook **31**

A List of Acronyms **33**

B List of Listings **34**

C List of Figures **35**

1 Introduction

In software engineering today, the approach of Model Driven Architecture is becoming ever more prominent. The need to have a way of software engineers and customers to communicate has led to the specification of Visual Modeling Languages, with Unified Modeling Language (UML) being the de facto industry standard. Visual Modeling Languages allow to model a software in a way that is well understood by both the software developers and their customers.

A problem of UML is that it has a well defined syntax, defined by a meta model, but the semantics are provided only as a textual description. This approach has the danger of leading to incosistencies or even contradictions, because the description is too large to be checked manually and a textual representation can not be checked automatically by a software system.

For this problem, Engels, Hausmann et al. [7] have introduced the method Dynamic Meta Modeling (DMM) which allows to define the semantics of a meta model in a formalized way. In DMM, the semantics of a meta model is represented by a set of graph transformation rules typed over the meta model, called the dynamic meta model. The syntax is represented by a model called the static meta model. These rules can be applied to a model which is an instance of the meta model resulting in a graph transformation system.

Meta models can be represented according to the Meta Object Facility (MOF) standard [3]. Ecore, a part of the Eclipse Modeling Framework (EMF) implements the Essential MOF standard and is used in this work to define meta models. Using EMF, we can define model instances of these meta models which can be used as static meta models in DMM.

Röhs [11] presented a graphical editor for DMM rules based on Graphical Modeling Framework (GMF). This can be used to produce rulesets for DMM graphs typed over EMF models, which constitute the dynamic part of the DMM meta model.

When transformed to Groove rulesets, the dynamic model can be used by the GRaphs for Object Oriented VErification (Groove) [8, 9, 10] toolset to automatically compute the resulting transition system of a graph and a set of transition rules. Groove can also be used to identify interesing states that meet conditions defined in Groove rules, but it cannot read EMF models, so a transformation from EMF models to Groove state graphs is necessary

This bachelor thesis presents such an implementation to transform EMF models to Groove state graphs. Additionally, the Groove state graphs can be transformed back to EMF for further inspection. This is done by implementing an Eclipse plugin that consists of two transformation methods transforming from EMF to Groove and back.

Figure 1 gives an overview: An EMF model is transformed to a Groove start state that is then used together with a ruleset to compute a transition system. Groove can identify states that meet defined conditions, which can then be transformed back to EMF to inspect them. The transformations from EMF to Groove and back are presented in this thesis.

First we will give a further introduction to DMM, EMF and Groove in Section 2. Then the transformation from EMF to Groove is presented in Section 3

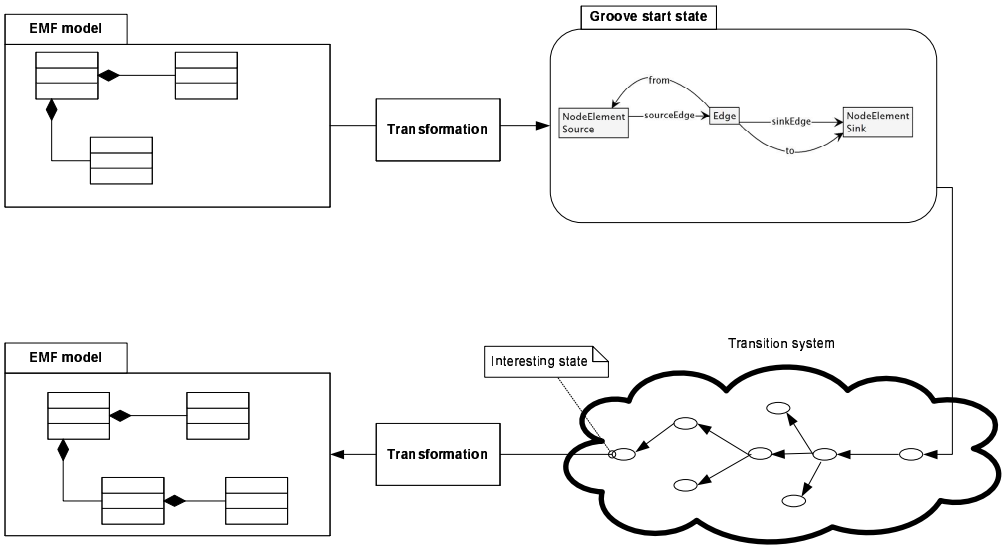

Figure 1: Overview over a bidirectional transformation

and the reverse transformation in Section 4. In Section 5 we will show that the bidirectional transformation works with a number of test cases. Section 6 concludes the paper and gives an outlook for future work.

2 Fundamentals

In this section we will look at the necessary technology for the transformations. Section 2.1 will give an overview over Dynamic Meta Modeling (DMM). Section 2.2 will deal with the Eclipse Modeling Framework (EMF) technology, showing how the ecore meta model is structured and how it helps us to generically access EMF models. Section 2.3 shows Groove toolset stores models as graphs.

2.1 Dynamic Meta Modeling

DMM is a method to model the semantics of a model in its meta model additionally to the syntax. Normally, a meta model consists of a graphical representation for the syntax, which is on the one hand easy to comprehend and on the other hand avoids misunderstandings. The semantics are often represented by textual, informal annotations. These can fill hundreds of pages, as in the case of UML. Since there is no way to automatically check these ambiguous texts for errors, inconsistencies and double meanings are prone to be overlooked. Another way to model the semantics is by using a mathematical model. This is formally defined, so we can mathematically check if it behaves we want and if there are inconsistencies, but it is not easy to understand for people without a technical background.

DMM, on the other hand, combines the good sides of both by using a graphical representation that is also a formal model for both the syntax and the semantics of the model. This not only unifies the representation of syntax and semantics, but also allows tools like Groove to check the graph for errors, if we give it the right input format.

Let us look at flow networks [4] as an example. Flow networks are a formal model, but they can also be represented graphically, as we will do in this thesis. An example meta model for flow networks is shown in Figure 2. There is a container class containing all others, which is called FlowNetwork. The nodes in the network are represented by the abstract NodeElement class, which defines a name and a height for use in algorithms like the Push Relabel Algorithm [6]. In the Push Relabel Algorithm, the nodes are lifted to a height h and can only push flow along edges that have a lower height. A node is overflowing if there is more flow entering it from incoming edges than leaving it through outgoing edges. As long as there are overflowing nodes, the Push Relabel Algorithm will either lift an overflowing node to a higher height than it's neighbors or, if possible, push as much flow along the outgoing edges as possible. When there are no more overflowing nodes, a maximum flow has been computed [6]. There are three classes inheriting from NodeElement, Source, Sink and Node. Source and Sink represent the source and sink in the flow network. They are special nodes, because they have only outgoing or incoming edges respectively and they can not be overflowing. The outgoing edges of the source are referenced by the sourceEdge association and the incoming of the sink by the sinkEdge association. The Node class specifies the additional boolean attribute overflowing and has both outgoing and incoming edges modeled by the associations outEdge and inEdge. Edges have a capacity, a flow and can be satisfied. An edge is satisfied if the flow is equal to the capacity. There can never be more flow flowing over the edge than the capacity allows. The directed edges also have associations "to" and "from" with the nodes they connect.

2.2 Eclipse Modeling Framework

The Eclipse Modeling Framework is a powerful framework developed for Eclipse. Ecore models are the standard model format of EMF and can be generated from Java code, XML data or UML diagrams. Java code generation from an ecore model is supported as well. In this thesis, we will concentrate on the modeling capacities. This subsection gives a short overview of the modeling concepts behind EMF. More detailed descriptions can be found in chapter 5 of [5].

Ecore models are defined by a meta model, which is also defined in Ecore. Figure 3 shows a simplified model of the Ecore meta model [5].

As we see, Ecore models mainly consist of four different types of objects [5]:

- EClass models classes. They are identified by their name attribute and can contain a number of references and attributes. Inheritance is modeled by the eSuperTypes reference, where a number of other EClasses can be referenced. In our example from Figure 2 Edge or Node are EClasses.

- EAttributes are the attributes of an EClass. They are also identified

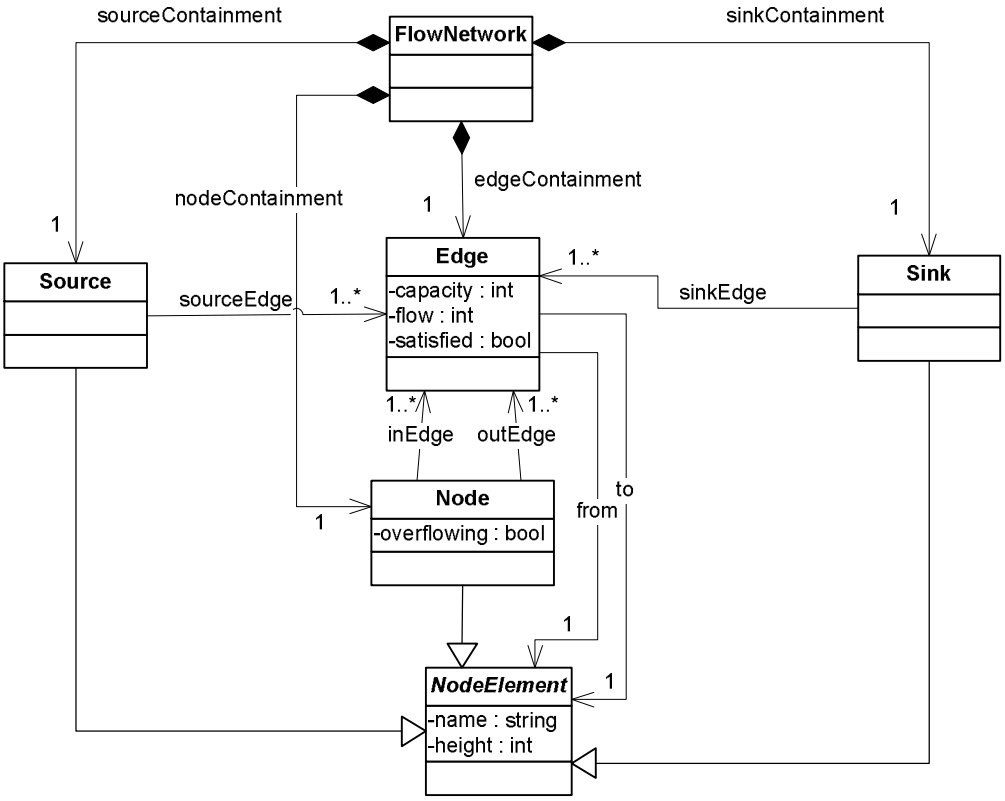

Figure 2: Example core model modeling a flow network

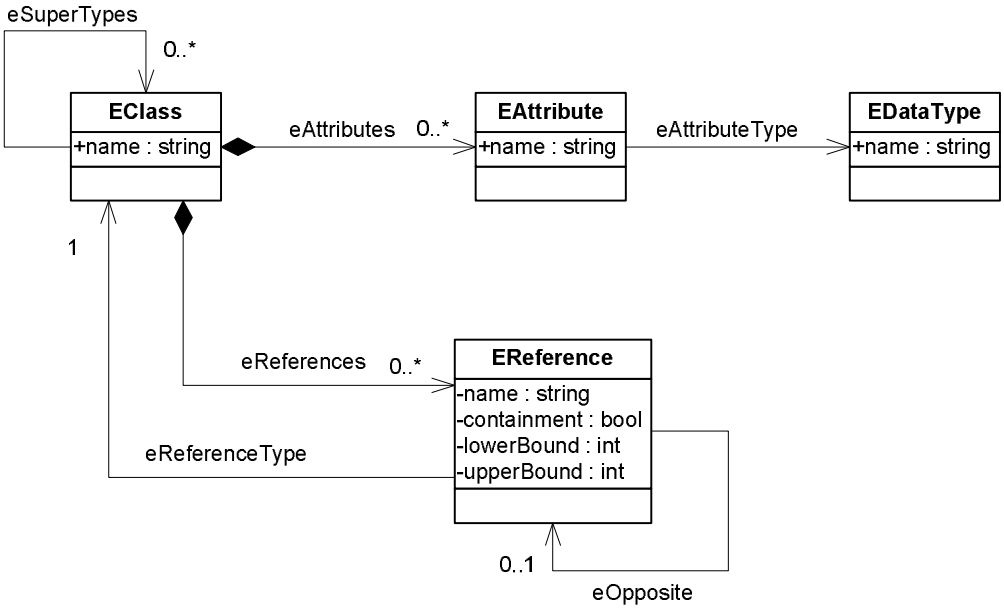

Figure 3: The simplified Ecore Kernel [5]

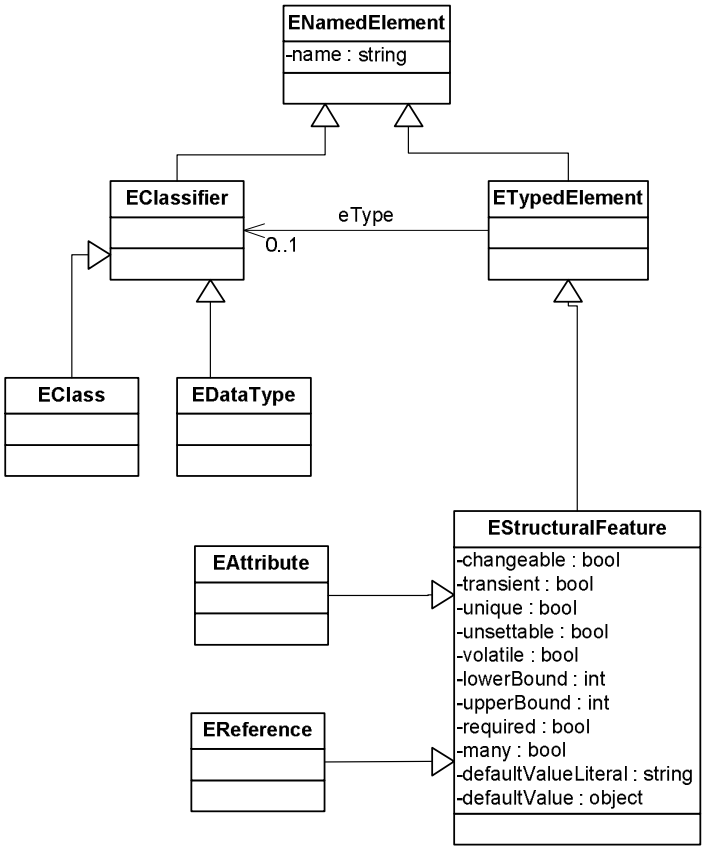

Figure 4: Ecore structural features [5]

by a name and have exactly one eAttributeType. name or height are EAttributes.

- EDataType models the attribute type of an EAttribute. It is identified by a name. Data types can be primitive types like int or object types like Integer [5]. EInt or EBoolean are EDatatypes in the example.

- EReference models associations of EClasses. It has a name attribute as identifier. The containment attribute defines if it is a containment refererence. Lower and upper bounds are defined by their attributes. An EReference may have an EOpposite if the association is bidirectional and must have an eReferenceType to define which type of EClass is referenced. EReferences in the flow network example are sinkEdge or nodeContainment.

Actually, there are a lot more meta attributes that EReferences and EAttributes share, as Figure 4 [5] shows.

As we see the name attribute was actually inherited from ENamedElement, which only defines this attribute. EClass and EDataType share the supertype EClassifier, which is referenced by ETypedElement. This is because both ERef-

erences and EAttributes as some other classes in Ecore have an eType reference which can references EClassifier. Most attributes EReferences and EAttributes share are inherited from EStructuralFeature [5]:

- changeable determines if the value of the feature may be externally set.
- transient is true if the feature will not be serialized.
- unique specifies for multiplicity-many features if a single value may only occur once.
- unsettable specified if the feature may be unset. If a feature is unset, it has no value. If a feature is not unsettable, it will be set to the default if an EObjects eUnset() method is called.
- volatile defines that the feature has no storage directly associated. This is usually the case if it can be derived from other features.
- upperBound and lowerBound define the multiplicity of a feature.
- required and many are conveniences and derived from the lower and upper bounds.
- defaultValueLiteral stores the default value of the feature as a String and defaultValue is derived from this by converting the String to the appropriate data type.

What sets EReferences and EAttributes apart are only a few meta attributes [5]. EAttributes have a boolean iD that determines if the attribute can be used to uniquely identify an EClass. This attribute will then be referenced by a reference eIDAttribute from the EClass. ERferences have the containment and container attributes. If containment is true, the referenced class is contained by the refererencing class. A contained class may not contain their container [5]. Container models the same relationship, but the other way around. In Figure 2 EdgeContainment is an example of a containment reference. Finally there is a boolean attribute resolveProxies which determines if EClasses from other Resources should be loaded if necessary.

Data types in Ecore represent "simple" data [5]. These comprise of primitive types like int, but also objects like String. This gives us the possibility to model conceptually simple data as objects without operations, although there may be operations in Java. Still, this is sensible to keep the models as simple as possible. One exception from this are enumerated types which are illustrated in Figure 5 [5].

Enumerated types are defined by their literals, which are a concrete list of values they can take. As we see the literals inherit from ENamedElement, so they also have a name. Thus, we can use enumerations to define our own distinct states for an attribute. The two different methods getEEnumLiteral() return an EEnumLiteral either identified by name or value.

Another important feature of EMF for us are packages and factories. Packages contain related classes and data types and their metadata. Factories are

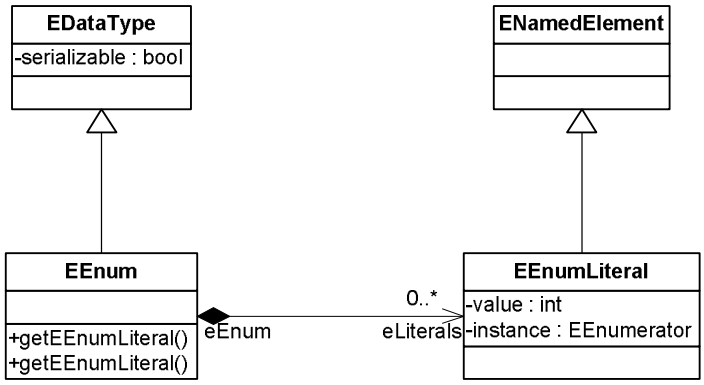

Figure 5: Ecore Enumerations [5]

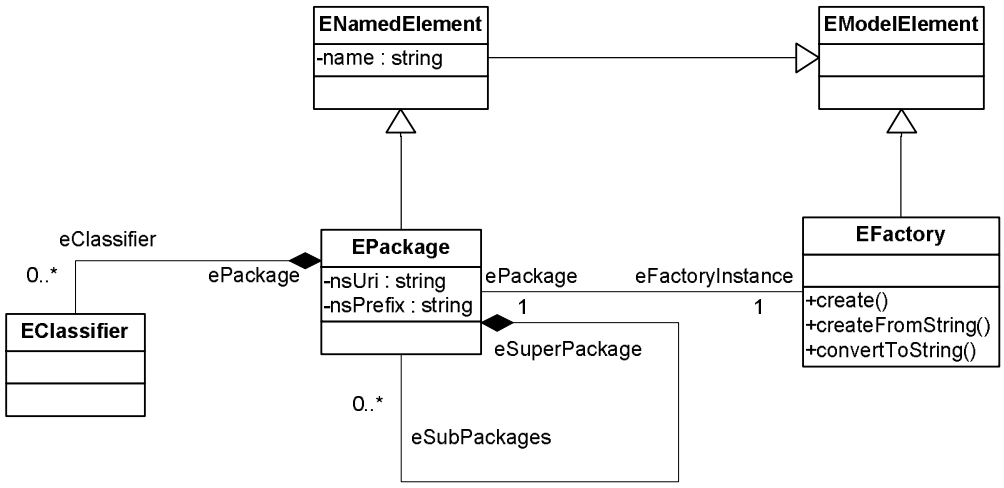

Figure 6: Ecore packages and factories [5]

used to create instances of these classes. Figure 6 [5] shows the relations between these.

EPackages are identified by their namespace URI (nsURI). This URI must be unique. They also have a name, but that may be not unique. As we see, the packages consist of EClassifiers, referenced by the eClassifiers association. Recall that EClasses or EDataTypes are EClassifiers (see Figure 4). A package may also have subpackages, where the names of EClasses may be duplicates of those in the main package. EPackages are stored in a package registry at runtime which can be used to access the EPackages when needed. All that is needed for this is the namespace URI. Each package has a unique EFactory associated by the eFactoryInstance association. This EFactory can be used to create instances of the modeled EClasses.

EClasses may also contain EOperations. These model operations, but no semantic information is modeled or will be generated. DMM could be used to actually model the behaviour of the operations. The transformation in this

work does not transform EOperations, so we will not describe them further.

In this thesis, mainly the capabilities of EMF for accessing the meta models generically are used. Using these techniques, we do not need to know the meta model at compile time, but only at run time. As explained before, we can get the metadata of a model by accessing the EPackage, which can be done using the EPackage registry. The implementation of the bidirectional transformation is designed to run as an Eclipse Plugin. We suppose that the model code of the used packages has been generated by EMF. This is necessary to create instances of the EClasses defined in the EPackage. EMF generates specialized factories to create these EClasses, but also generic methods that delegate to the specialized implementations. Using this generated code, we can then create a model conforming to an arbitrary core model if we know the names of the needed EClasses and EDataTypes. See Section 4 for details on how this was implemented.

2.3 Groove

Groove is a toolset that is designed to support model checking using graph transition systems [8, 9]. It can automatically compute a transformation system if it is supplied with a start state and a ruleset. As this thesis focuses on transformation for EMF models to Groove start states, we will not explain the format of the Groove rules here. Information about that can be found in the Groove manual [10]. Groove consists of five tools [10]:

- The Simulator is used to explore the graph, the resulting states and the ruleset.
- The Editor can be used to edit both start states and rules. The figures of Groove graphs in this thesis were made with the Groove Editor.
- The Generator computes the transition system of a graph from a start state and a grammar.
- The Imager can create pictures of graphs, rules and transition systems.
- The Model Checker verifies properties over graph transition systems.

We will now give an introduction to the format of Groove start states, as this is what the one side of the transformation produces and the other side takes as its input. Groove graphs consist of nodes without labels and labeled, directed edges. These graphs are stored in the Graph Exchange Language (GXL) format, which is defined by an Extensible Markup Language (XML) schema. As node labels are a wanted property, they are modeled by self edges of the nodes. In the Editor and Simulator these are displayed as labels inside the nodes. Figure 7 shows an example graph in the normal representation and with explicitly shown self edges. We left out the FlowNetwork root element and all attributes for simplification. As we see, displaying the self edges as node labels greatly simplifies the graph, so we will use this representation in the further sections of the thesis. We should keep in mind that all node labels in Groove graphs are actually labelled self edges.

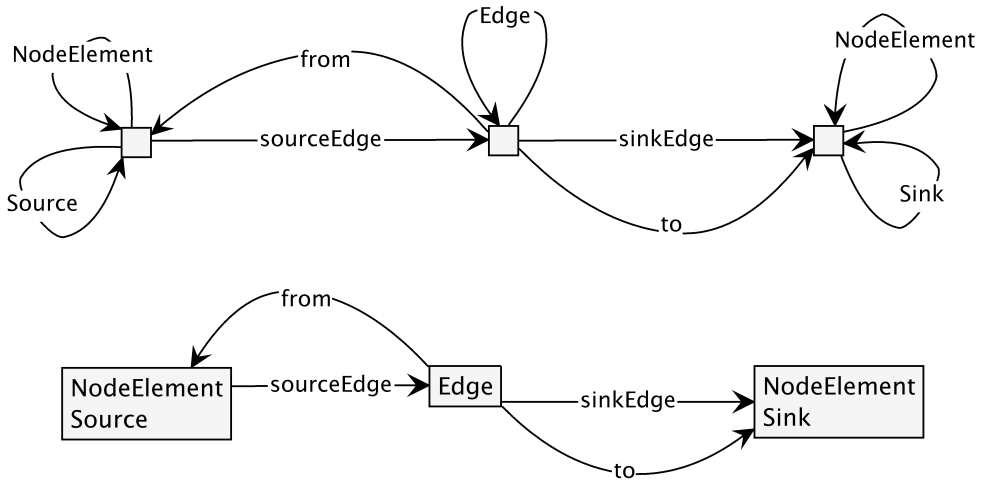

Figure 7: Groove graph with and without self edges explicitly shown.

In Figure 7 we also see how objects will be modeled in Groove in the remainder of this thesis. Each object is represented by a node with labels according to the object's class and all its supertypes. Figure 8 shows how attributes are modeled. They are represented by special nodes with self labels that begin with a codeword indicating that the node represents a variable. At the moment, Groove only supports integers (int:), floating point numbers (real:), strings (string:) and booleans (bool:). The objects the attributes belong to are connected by edges labeled with the attribute name.

Using graphs of this kind and according rules, Groove can compute the transition system of a graph and also automatically check for conditions. This can be used for model checking, i.e. to see if a petri net contains deadlocks.

3 Transforming EMF to Groove

This section presents the transformation of an EMF model to a Groove graph or state. This transformation is necessary to be able to use Groove to compute a transition system for the graph corresponding to the EMF model.

The transformation from an EMF model to a Groove start state is done by the method emfToGxl. The API allows to call the method with a File, a Resource or an EObject as the input parameter. If the method is called with a File as parameter, a Resource is loaded from the file and the method for a Resource is called. Then, since there is always a single root object in a model Resource, this root EObject is loaded and the method with an EObject as input parameter is called. Listing 1 shows the method in Pseudocode. Figure 9 shows a model that could be used as an input for the method. It is based on the flow network meta model from Figure 2.

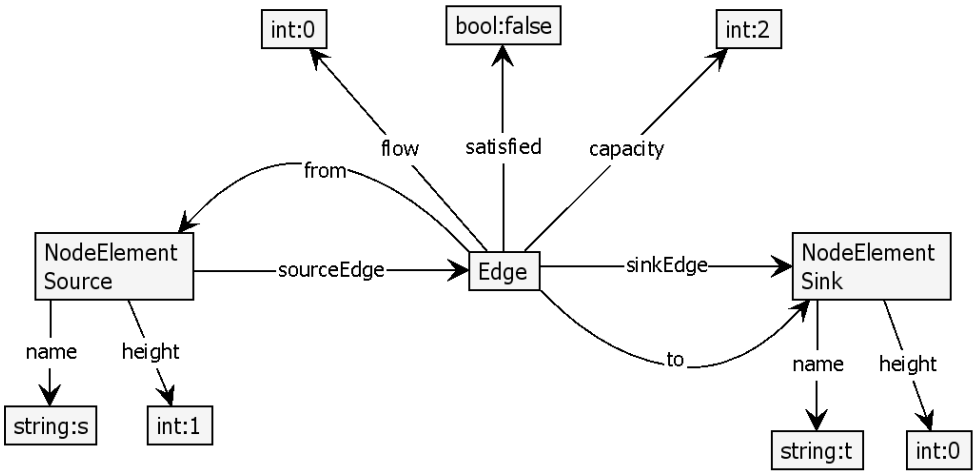

Figure 8: A Groove graph with attributes.

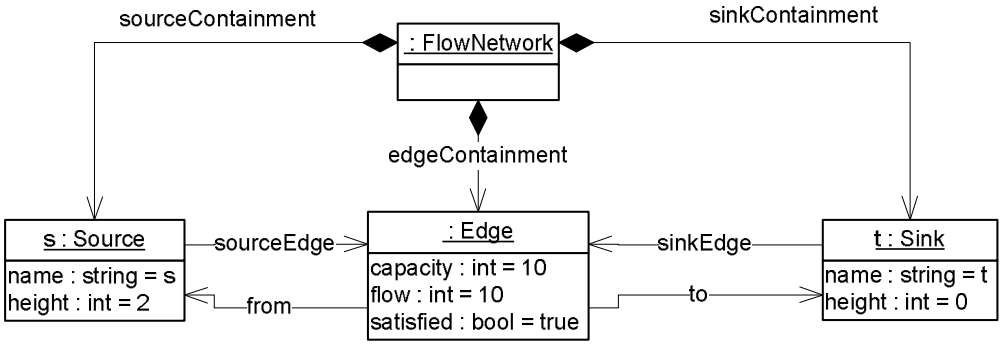

Figure 9: Model of a simple flow network

Listing 1: Main method of the conversion from EMF to Groove

```
1  GxlType emfToGxl(EObject rootObject) {
2    create an empty Gxl graph;
3    for (all objects reachable from rootObject)
4      get packages from package registry;
5    add a node with labels of all nsUris
6      of the packages;
7    preProcessPackage(List of nsUris);
8    createEnumNodes(package);
9    for (rootObject and all its ancestors)
10       addEObject(eObject);
11   for (rootObject and all its ancestors)
12       addReferences(eObject);
13 }
```

The output Groove start state is shown in Figure 10.
Remember from Section 2.2 that all meta data is stored in packages, which

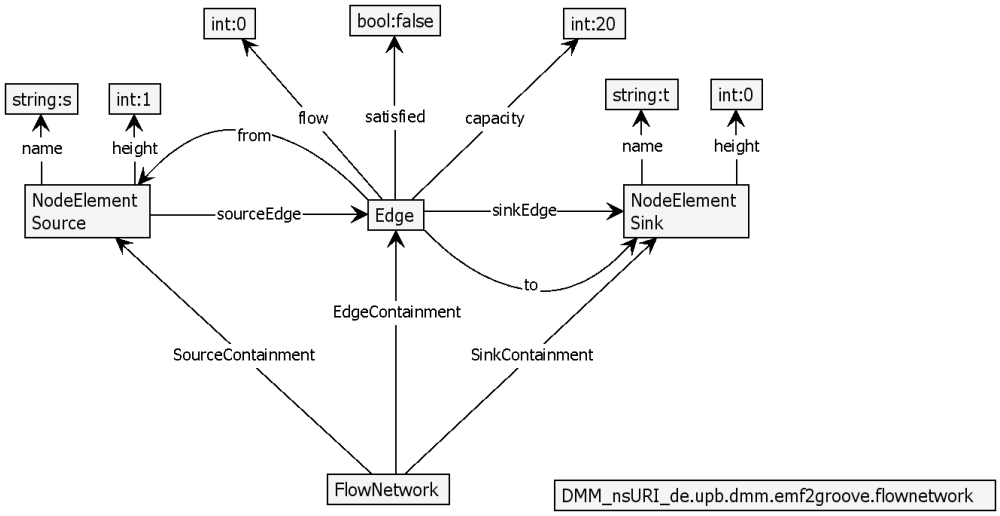

Figure 10: Groove start state for the flow network

are identified by their namespace URIs. We generate a list of all namespace URIs of packages that are referenced in the core model by going through all classes and their references. The first node we create in the new GXL graph is a special node indicating the namespace URIs of all packages referenced in the core model. The node labels, which are actually labeled self edges, see Section 2.3, are of a special format that identify them as system nodes which will be ignored in the further transformation process. The labels begin with "DMM_NsUri_" followed by the actual namespace URI. In the reverse transformation from a Groove graph to an EMF model, we will use this node to load the necessary packages (see Section 4).

The packages are then preprocessed to resolve duplicate class names, see Section 3.1. Next, nodes for all enum literals in the package are added, see Section 3.2.

Now, we first create nodes for all EObjects in the model. At first, we call addEObject() for the root EObject, adding a node with all appropriate class edges and attributes. Using the EMF method eAllContents(), we get all ancestors of rootObject as a TreeIterator. We iterate through all of these EObjects and call addEObject() for each of it. Listing 2 shows how this method works.

Listing 2: addEObject adds an EObject and the according attributes to the graph

```
1  String addEObject(EObject eObject) {
2    newNode = a new node in the graph;
3    eClass = eObject.eClass();
4    className = eClass name from the class name Map;
5    add a self edge to newNode with className as label;
6    for (all super types of eClass)
7      add a self edge to newNode with super type's
```

```
8         class name as label;
9     addAttributes(newNode, eObject);
10    add eObject to Map objectsDone;
11  }
```

First we first create a new node representing the EObject, then we add a self edge to this node for its eClass and all super types. At last we call addAttributes for the new node, also passing the EObject as an argument. The method addAttributes adds the attributes of an EObject as in Listing 3. Finally, we add the eObject to a Map objectsDone<EObject,String>which stores what node represents what EObject. We will need this information later to add the references to the graph. The String returned by addEObject() is the node ID, a string identifying all nodes in a GXL graph.

Listing 3: Adding the attributes of an EObject to the according node

```
1  void addAttributes(NodeType newNode, EObject eObject) {
2    for (all attributes of eObject's EClass)
3      {
4        if (the attribute is an enumeration)
5          addEnum(newNode, attribute);
6        else {
7        attributeNode = a new node in the graph;
8        add self edge to attributeNode containing
9          the attribute type and value;
10       add edge from newNode to attributeNode
11         with the attribute name as label;
12       }
13     }
14 }
```

As we can see, we again use the meta data from the package in this method. Here, we go through all attributes that eObject's EClass has defined. This also includes all attributes defined as volatile or derived (see Section 2.2), because these can also be interesting for Groove and we do not want to compute them on the Groove side by designing special rules for derived values. EMF also returns all inherited attributes when getEAllAttributes() is called, which we do in this method. If the current attribute in the loop is an enumeration, we call the method addEnum(), which is described in Section 3.2. As explained in Section 2.3, attributes in Groove are represented by nodes with self edges specifying type and value, and an edge from the object to the attribute. So, in line 7 we create a new node, in line 8 we add the type and value and in line 9 we add the edge from the eObject to the attribute. The type of the attribute can be found out by getting the attribute from the eObject and inspecting its EType. This is then transformed to a string according to Table 1. To this string the value of the attribute is appended by getting it from the eObject and transforming it to a String using the toString() method.

Figure 11 shows the graph after all objects were added. All EObjects have

Table 1: List of supported data types and their representations in the Groove graph

Data type	representation
EBigDecimal	real
EBigInteger	int
EBoolean	bool
EBooleanObject	bool
EByte	int
EByteObject	int
EChar	string
ECharacterObject	string
EDouble	real
EDoubleObject	real
EFloat	real
EFloatObject	real
EInt	int
EIntegerObject	int
ELong	int
ELongObject	int
EShort	int
EShortObject	int
EString	string
EENum	see Section 3.2

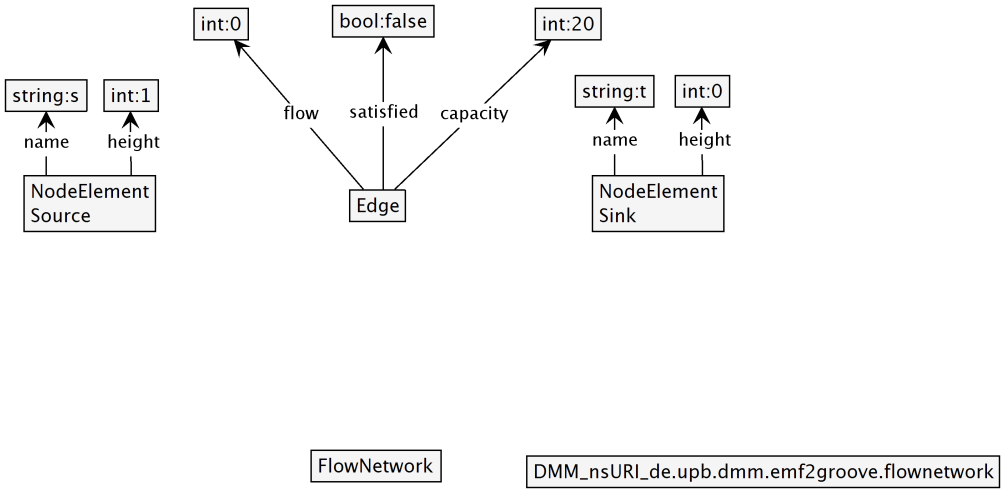

Figure 11: A Groove graph after the EObjects have been added, but before the EReferences are added.

been transformed by addEObject() and their respective attributes added to the graph, but there are no references yet. So, as we have seen in 1, the next step is to go through all EObjects again and add their references. This is done by the method addReferences() shown in Listing 4.

Listing 4: Method adding the references to the graph

```
1  void addReferences(EObject eObject,
2                     Map<EObject, String> objectsDone}
3    eClass = eObject's EClass;
4    for (all References defined in eClass
5          and it's super types)
6      {
7        if (reference is ordered)
8        addOrderedReference(eObject, reference);
9        else
10         for (all referenced EObjects)
11           add edge with label reference name
12             from eObject's node to referenced
13             EObject's node;
14     }
```

Here we again use EMF's generic API. We use getEAllReferences on the eObject's EClass, getting all EReferences that are defined in the meta model for the EClass and all super types. If the reference is ordered, we use the method addOrderedReference which is documented in Section 3.3. Otherwise, we can simply add an edge to the graph for each referenced EObject. Since we stored in a Map which EObject is represented by which node, we know where the edge has to go. The label of the edge is simply the reference name, which we also get from the meta model. If there are no EObjects being referenced

for a specific EReference, no edge is added. Now the transformation is done and the GXL graph is returned. It is shown in Figure 10. This graph can be loaded in Groove and with a corresponding ruleset a transition system can be computed. The following subsections will show how special cases in EMF models are transformed.

3.1 Duplicate Class Names

Since a core model may contain references to EClasses in other packages, or subpackages, there is the possibility that the names of two EClasses from different packages are the same. In this case, simply using the EClass' name as the label of a self edge would lead to problems because the two classes would be undistinguishible in the Groove graph. So, to avoid this problem, we have to store more information in the self edges. We could use a schema like "nsURI#EClassName", which would be unambiguous, since the nsURI is required by EMF to be unique, as are EClass names within one package (see Section 2.2). But in this case the node labels would have very long names and would be hard to work with in Groove, especially when designing rules. So we want to store as little information as possible, but as much as necessary to identify each class.

Our example core model does not contain any duplicate file names yet, so let us add an EClass Node as in Figure 12. The package counting is a subpackage, but could also be an external package. This EClass has the same name and super types, so it would look the same in the output Groove state. It has another incoming containment reference and an additional attribute "counter" which could be used to distinguish it from the original Node EClass, but that would not always be possible.

Since we want to be able to distinguish between these two classes easily both in Groove and in the reverse transformation back to EMF, we will use the method preProcessPackage() to define distinct names for all EClasses. The same method will be used in the reverse transformation, so the resulting names will also be the same, as we will see by analyzing the method in Listing 5.

Listing 5: Package preprocessor to resolve duplicate class names

```
1  BidiMap<URI, String> preProcessPackage(List nsUriList) {
2    classNameMap = new BidiMap<URI, String>;
3    for (all nsUris in nsUriList)
4    {
5      get the package that is registered for nsUri;
6      for (all EClasses in package)
7      {
8        className = name of current EClass;
9        if (BidiMap contains className)
10       {
11         classPackageName = name of current EClass' package;
12         className = classPackageName + className;
13       }
```

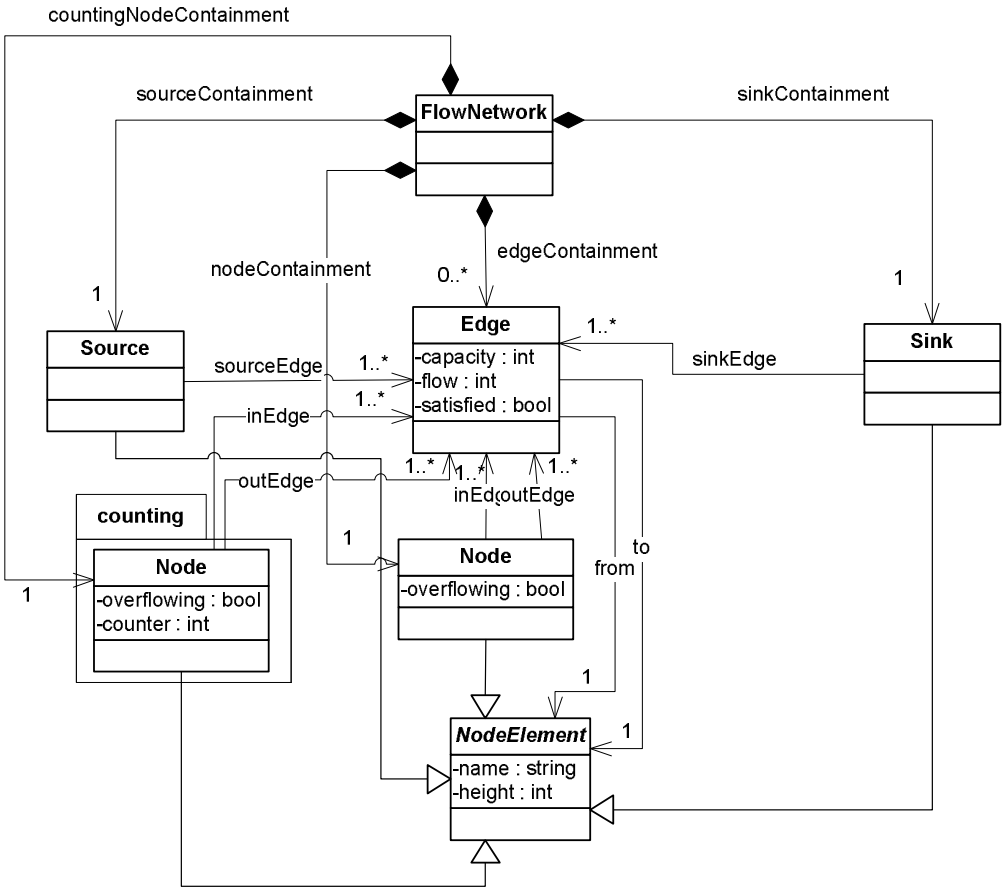

Figure 12: Flow network example containing a duplicate EClass name in a subpackage

```
14            classNameMap.put(ClassURI, className);
15        }
16    }
17    return ClassNameMap;
18  }
```

This method returns a bidirectional mapping between URIs and Strings and is called with a List nsUriList. nsUriList is a list of namespace URIs of all packages that are referenced in the core model. This can be found by going through all EClasses in the package and getting the package for each, adding it's namespace URI to the list if it is not already in it. Since we assume that the core model will not change between our transformations, the order that the packages are found in is well-defined.

The URIs that will be stored in the bidirectional map are the class URIs of the EClasses in the package. This is computed by reflectively getting the EClass' package's namespace URI and adding the EClass' name as the URI fragment. In our example, if the namespace of the main package is "de.upb.flownetwork", the class URI of the Node EClass will be "de.upb.flownetwork#Node". Call-

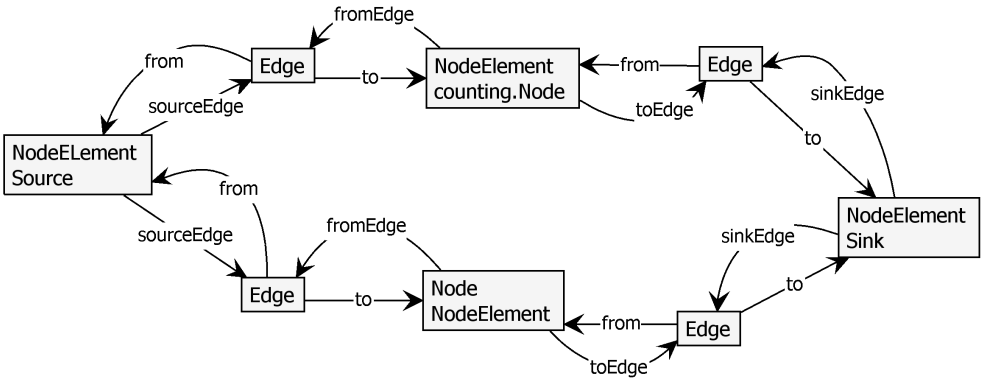

Figure 13: Groove graph with resolved class names. The node FlowNetwork has been left out for clarity.

ing EClass.getPackage() on the Source EClass in the subpackage will return "de.upb.flownetwork.counting", so the EClass URI will be "de.upb.flownetwork.-counting#Node". Since the namespace URI of all packages must be unique and there cannot be duplicate EClass names in the same package, the class URI is a distinct identifier for each EClass.

As we said before, we do not want to use these long identifiers in our Groove graph. For this reason we only use the class name when we add a class to the map, which is done in the order we find them in the core model. If we find a second class in the core model that has the same name, we prefix that with the package name. This is the last part of the namespace URI, so it would be flownetwork for classes in the main package and counting for classes in the subpackage. Figure 13 shows the output graph of a simple flow network containing both a Node object from the main package and one from the counting package. As we see, the first Node class from the main package is labeled Sink. The second Node class from the subpackage is labeled counting.Node to avoid ambiguity.

3.2 Enumerations

Enumerations offer us the possibility to define our own states for a variable. Let us change the example from Figure 2 to incorporate an enumeration. At the moment, EClass Node has a boolean variable named "overflowing". We will now model this through an enumeration named flowState that has the three literals "overflowing", "flowing" and "empty". As before, it is a derived value that is calculated from the flow going in and out of the Node. If there is more flow going in than out, than the node is overflowing. If there is as much flow going in as out, it is flowing, and if there is no flow going in or out at all, it is empty.

As explained in Section 2.2, an EEnum contains the EEnumLiterals and we can get them by using the eLiterals reference, see Figure 5. Listing 6 shows the method that was called at the beginning of the transformation (see Listing 1).

Listing 6: Creating the enum literal nodes

```
1  Map<String, NodeType> createEnumNodes
2                              (EPackage package) {
3    enumMap = new Map<String, NodeType>;
4    for (all EEnums in package)
5    {
6      create a node labeled with the EEnum's name;
7      for (all ELiterals of the EEnum)
8      {
9        create a node labeled with
10         the ELiterals's name;
11       create an edge from the ELiteral's node to the
12         EEnum's node labeled DMM_EEnum;
13       enumMap.put(literalName, node);
14     }
15   }
16 }
```

We create a node for each EEnum in the core model and nodes for each ELiteral linked to their according EEnum by an Edge labeled "DMM_EEnum". The literal nodes nodes are stored in a Map that links the ELiteral names to nodes in the graph. Mind that we do this independently of the actual model. That means we also add the nodes for EEnums and ELiterals that are not used by any objects in the model. This allows Groove rules to add Objects with these literals or to change the enumeration literal of an object to another one. We do not actually need the EEnum nodes for the transformation, but they can be used to identify to which EEnum a literal node belongs in Groove. When we add the attributes to a node created for an actual EObject, if we find an enumeration attribute we create an edge from the node representing that object to the according literal node. If more than one EObject has the same ELiteral as an attribute value, they will have edges to the same literal node. We can get the value from the EObject reflectively just as with normal attributes, in this case we will get an EEnumLiteral. We can then use the name of that literal to find the according node in the enumMap. Figure 14 shows an example Groove graph that has enumerated attributes.

3.3 Ordered References

As we saw in Listing 4, ordered references are transformed differently than non-ordered references. In the case of non-ordered references, we just add an edge between the nodes representing associated nodes. Even if there is more than one object referenced, we will just add more edges, all labeled with the name of the reference, but this does not preserve the order of the reference. Figure 15 shows an example of a flow network with multiple Nodes. The order of the Nodes is a, b, c. Since the nodeContainment EReference is ordered, we will use the methods described in this section.

The order of a single reference could be saved by using simple next-edges

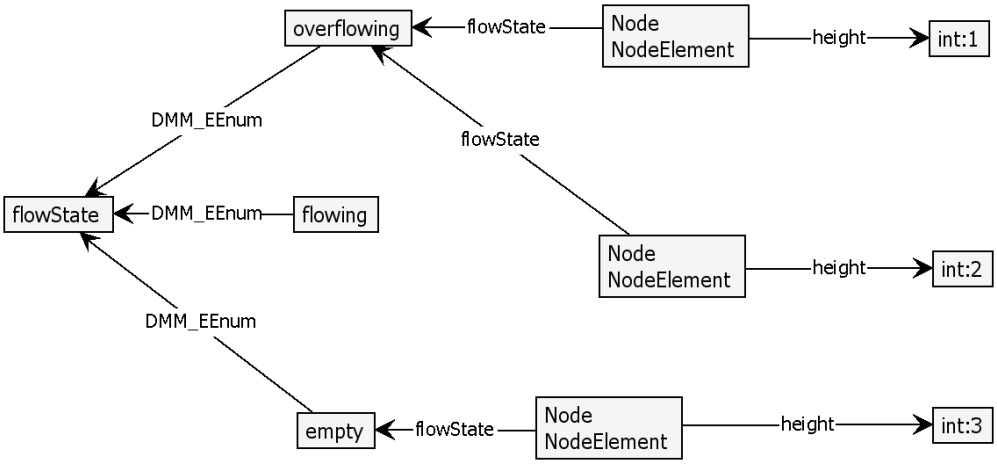

Figure 14: Example Groove graph with EEnumerations. Nodes without enumerations have been left out for clarity.

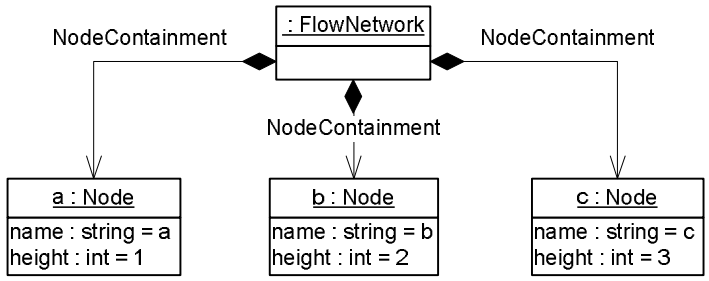

Figure 15: Flow network with three Nodes. The order of the Nodes is a, b, c. Edges, source and sink have been left out.

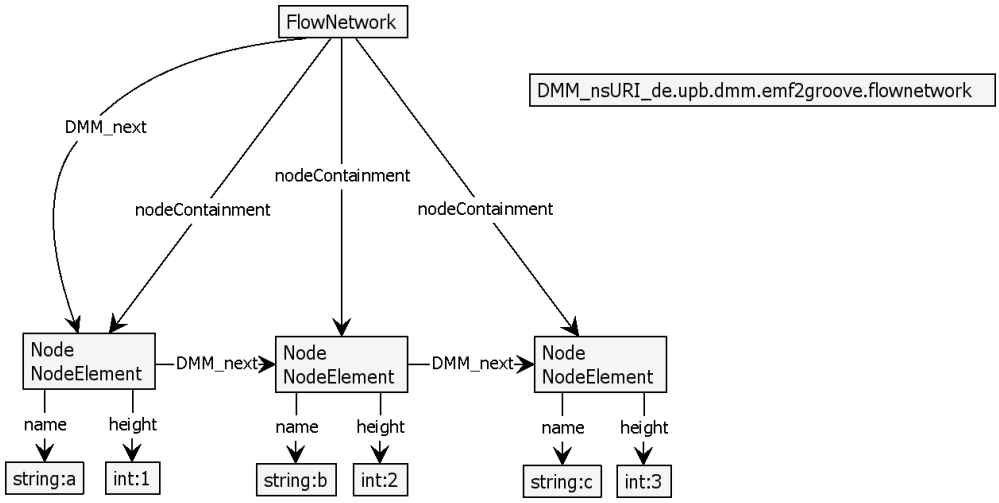

Figure 16: Excample Groove graph with simple next edges.

as in Figure 16. We labeled the edges "DMM_next" to mark them as edges used by the transformation system. This keeps the graph simple and allows to determine the order of the referenced objects, but this will not work if there are multiple references referencing the same objects.

If an object has multiple ordered references, there would be multiple next edges. This problem could be solved by labeling them with "DMM_next_⟨reference name⟩". Still, this would not suffice for a well-defined order in all cases. If there are two FlowNetwork objects referencing the same Nodes in a different order, we could not distinguish the next nodes. This leads us to the solution presented in Figure 17.

In this graph, there is a system edge labeled "DMM_next" from the referencing FlowNetwork to the first next node, and from there on to the next one and so on, sustained through a next node for each referenced object. These next nodes are labeled "DMM_next" and "DMM_next_⟨reference name⟩". Also, there is an edge labeled "DMM_ordered_reference" from each next node to the corresponding object node. This mechanism allows us to find the order of the referenced objects when transforming the graph back to EMF or in Groove using rules. Also, we can use rules in Groove to add further nodes to the reference anywhere in the chain of objects. In this case we would add a node for the object with the according edges and an according next node, updating the edges of the other next nodes in the chain. The next node chain basically works like a linked list with references to object nodes. The implementation of the funcionality described in this section simply adds the next nodes and edges by using EMF's reflective capacities.

4 Transforming Groove to EMF

The transformation from Groove states to EMF models can be used for example to inspect interesting states in the transition system computed by Groove inside

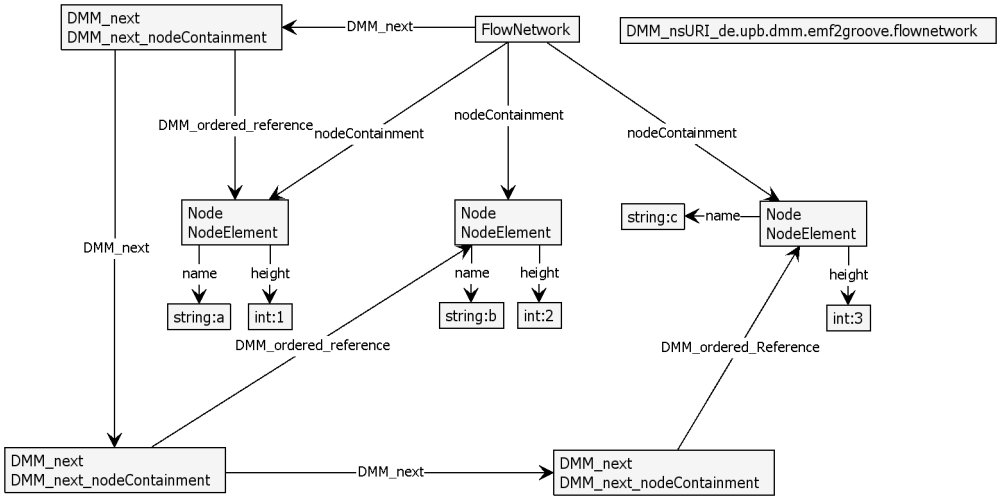

Figure 17: Finished Groove graph of the example from Figure 15

of the Eclipse platform. The API consists of two different calls to the gxlToEMF method. One method takes a File as input, loading a GXL graph from that and calling the other. The gxlToEMF method that is called with a GXL graph and a String is presented in Listing 7.

Listing 7: Main method of the transformation from Groove to EMF.

```
1  Resource gxlToEMF(GraphType gxlgraph, URI outFileURI)
2  {
3    resource = new Resource for outFileURI;
4    nsURI = namespace URI from the first node
5              in gxlgraph;
6    ePackage = package from the registry
7                 identified by nsURI;
8    BidiMap<URI, String> classNameMap =
9      preProcessPackage(ePackage);
10   Map<NodeType, EObject> nodesDone = new Hashtable;
11   for (all nodes in the graph)
12   {
13     if (nodesDone does not contain the node)
14     transformNode(current node);
15   }
16 }
```

gxlToEMF will return a Resource containing all EObjects that are represented by nodes in the input graph. To create this resource in line 3, we need to have a URI that defines where the Resource would be saved in a file. We will not save it in a file but return it unsaved, the method calling gxlToEMF can simply save the Resource to a file by doing resource.save() if needed. gxlToEMF first acquires the meta data of the model we want to create by getting a package from the package registry. Section 2.2 describes how the package registry in

EMF works. We get the namespace URI of the package from the first node in the graph, where it has been saved in the transformation from EMF to Groove. This first node is well defined because the GXL format stores the nodes as a list. Using this we can get the package from the registry. Next we preprocess the package to have a Map linking class URIs and the strings representing the class in the graph. This is done using the same function preProcessPackage as in the EMF to Groove transformation, which is described in Section 3.1. Since we use the same method on the same package, we will get the same mapping as in the other transformation which allows us to unambigously identify an EClass by the labels of a node in the graph. gxlToEMF also creates another Map nodesDone, which will be used to store which nodes have already been transformed to which EObjects. Then, for all nodes in the graph transformNode is called if the current node has not already been transformed. transformNode will recursively add all nodes referenced by the node it is called with and add them to the nodesDone map. Listing 8 shows how the method works.

Listing 8: Transformation of a single node into an EObject

```
1  EObject transformNode(NodeType node,
2                        bool addDirectly) {
3    eObject = getEObject(node);
4    if (addDirectly == true)
5      add eObject to the resource;
6    nodesDone.put(node, eObject);
7    addAttributes(node);
8    for (all references of eObject)
9      addReference(node, reference);
10   }
11 }
```

The input is a node and a boolean that tells the method if it should add the node to the resource itself or if it will be added by the calling method. This will be further explained when we show how referenced objects are added. We use the reflective capabilities of EMF to get an EObject instance of the EClass represented by the current node (see Listing 9). Then we add the object to the resource as indicated by addDirectly. We also put the node in the nodesDone Map, because an instance has been created and added to the resource or will be added by the calling method. In both cases, we must not handle the same node again, which is averted by adding it to the Map, as we will see in Listing 11. Next we set the attributes of the object, as shown in Listing 10. Lastly we add the references the EObject may have, see Listing 11.

Listing 9: Find out which EClass is represented by a node and create an instance of it.

```
1  getEObject(NodeType node){
2    for (all self edges of node)
3    {
```

```
4       get EClass represented by the label of the
5         outer edge;
6       rightClass = true;
7       for (all other self edges of node)
8       {
9         get EClass represented by the label of the
10          inner edge;
11        if (inner EClass is not super type of
12              outer EClass)
13          {
14          rightClass = false;
15          break;
16          }
17      }
18      if (rightClass = true);
19        create instance of rightClass;
20    }
21  }
```

getEObject that is presented in Listing 9 is used to create an instance of the EClass represented by a node. Since there is a self edge at the node not only for the EClass of the represented object, but also for each of the inherited EClasses, we have to find out which one is the right EClass. This is done by two nested loops over all self edges of the node. Say we have a node with the self labels "NodeElement" and "Sink", as in Figure 10. Now assume we first find the self edge "NodeElement". We will get the EClass "NodeElement" from the map created in preProcessPackage (see Section 3.1 and Section 4.1). This is not the class of the object that the node represents but a super type of it. But to find out that, we will have to go through all other nodes and check if "NodeElement" is a super type of one of these, which is exactly what we do in the inner loop. If we find an EClass inheriting from "NodeElement" in one of the other self edges, we save that "NodeElement" is not the right EClass by setting rightClass to false and break out of the inner loop. In our example the only other self edge represents "Sink", which is really inheriting from "NodeElement", so we will break the loop. Then we will look at the next self edge and compare it to all other self edges again, until we have found the EClass that is the "lowest" in the inheritance chain. In the example no other self edge represents a class inheriting from "Sink", so we will have rightClass be true after we examined the self edge representing the "Sink" EClass. When we have found the right EClass, we use the factory instance explained in Section 2.2 to create an instance EObject of that class and return this EObject. For this the model code has to be generated and loaded as an Eclipse plugin, because otherwise EMF would not be able to create an EObject of type EClass. The newly created EObject has all attributes set to their default values and the references will be unpopulated.

Thus, the next thing we do (see Listing 8) is setting the attributes with the method setAttributes, shown in Listing 10.

Listing 10: Setting the attributes of an EObject

```
1  void setAttributes (NodeType node, EObject eObject)
2  {
3    for (all neighboring nodes of node)
4    {
5      if (the node represents an attribute)
6        set the value of the attribute to that
7          stored in the node;
8    }
9  }
```

This method goes through all neighboring nodes of the node representing the EObject being currently processed. If it finds a node representing an attribute, it sets the attribute accordingly. Since the attribute name is stored in the edge leading to the attribute node and the value in the only self edge that node has, this is no big deal. We parse the value from the self edge and use the generic set method of EObject to set the attribute. See section Section 4.2 to see how EEnums are handled in contrast to simple data types.

Now all that remains is polupating the references of the EObject. Listing 11 shows how this is done.

Listing 11: Populating the references of an EObject

```
1  void addReferences (NodeType node, EObject eObject)
2  {
3    for (all references of the EClass of eObject)
4    {
5      for (all edges representing the reference)
6      {
7        if (the referenced node has not been
8              transformed)
9          transform the referenced node using
10           transformNode;
11       else get the referenced EObject from nodesDone;
12       add the referenced EObject to the reference
13         list of eObject.
14     }
15   }
16 }
```

From the meta data in the package, we get all EReferences defined for the EClass of the current object. We then look for edges that are labeled with the reference name and recursively transform all nodes that they lead to. Here, ordered references have to be resolved as demonstrated in Section 4.3. When transforming the referenced nodes, we set addDirectly to false for the transformNode method, because we do not want the EObject to be added to the resource directly. Instead, we add the returned EObject to the reference list of our current EObject, populating the reference.

With this step, the transformation of the node from Groove to an EObject is done. transformNode adds the EObject to the resource if addDirectly is true and returns it to the calling method. Since we do this for all nodes that have not been recursively added (see Listing 7), we will have transformed the whole graph at the end of method gxlToEMF. The following sections show how the special cases shown in Section 2.2 are transformed back to EMF.

4.1 Resolving Duplicate Class Names in Groove Graphs

In the transformation from Groove to EMF, we use the same method preProcessPackage as in the transformation from EMF to Groove. See Section 3.1 for how this preprocessor computes an unambiguous mapping from EClasses to Strings. Since it produces a bidirectional Map, we can use this map without modifications. The package namespace URI was saved in a node in the graph, see Section 3. Under the condition that the package identified by that URI registered in the package registry when we start the reverse transformation is the same as that when we transformed the model from EMF to Groove, in other words, the meta model has not changed, we will get the same mapping as in the original transformation. This means we will transform the nodes identified by their labeled self edges back to instances of the same EClass they were before. Another prerequisite is that the rules applied by the Groove ruleset also used the same mapping of Strings to EClasses. Remember that we will simply take the EClass' name if possible (see Section 3.1). Since there will only be relatively few cases where duplicate class names actually occur, the need to use more than the EClass' name in Groove rules will occur very seldom.

4.2 Setting the Values of EEnum Attributes

Recall how we represented ELiterals in the Groove graph (Section 3.2). We created a unique node for each ELiteral in the meta model. The nodes representing eObjects with EEnum attributes were linked to the according literal node. Now, in the transformation from Groove to EMF, we face the problem of creating the right ELiteral instance and setting the EEnum attribute to that. This is done by finding the edge representing the EEnum attribute and getting the label of the node that it leads to. This node is the representation of the right ELiteral. For example, if we look at Figure 14, Node a has the flowState "overflowing". This is represented by an edge labeled "flowState" leading to the literal node labeled "overflowing". Now, to transform Node a back, we look for the flowState edge and find that it leads to a node labeled "overflowing". We then use the EFactory in EMF to create an instance of that ELiteral and set the flowState EEnum to that instance.

4.3 Resolving Ordered References in Groove

Resolving the order of referenced objects from the structure we saved in the Graph as in Section 3.3 is done by the method addOrderedReference shown in Listing 12.

Listing 12: Populating an Ordered Reference

```
1  void addOrderedReference(NodeType node,
2                           Reference ref)
3  {
4    while (there is a next-node found)
5    {
6      look for the next next-node connected to node;
7      node = next-node;
8      transform the node that is linked
9        by the next-node;
10     add the EObject to the reference list;
11   }
12 }
```

addOrderedReference is called with the node representing the current EObject and the ordered reference we are populating. Let us look at Figure 17 as an example. FlowNetwork will be the node we begin with, the ordered EReference we are populating is nodeContainment. We will start by entering the while-loop and looking for a next-node connected to FlowNetwork. This node is identified as belonging to the nodeContainment reference by the label "DMM_next_nodeContainment". As there is one, we will set the current node to be that next-node. Then we transform the node representing the Node a and add it to the nodeContainment-list of FlowNetwork. The node representing Node a is found by looking for the edge labeled "DMM_ordered_reference" going out from the next-node. We first check if the node has already been transformed in the Map nodesDone and use the EObject stored there if possible. Then we look for the next next-node connected to the first next-node, find that indicating Node b as the second object in the reference, and so on until we do not find another next-node. Using this algorithm we add the referenced EObjects in an ordered reference in the right order.

5 Testing the Transformations

A number of tests have been conducted to test if the transformation works as expected. To test the system, we have designed test cases that cover the modeling capabilities of EMF. The test cases are defined by EMF models that contain specific model features like all supported kinds of attributes or different references. To test if both transformations work, we first transform the models to Groove states and then transform these back to EMF models. We are using the EMF Compare [1] plugin to test the models for equality. Compare tries to match the objects in the model to each other using a number of heuristics. Since the models are instances of the same core model, which is also available at runtime, Compare should have no problems with the matching. After that a DiffModel is built, which contains all differences between the models. This DiffModel is assumed to be empty by the tests. Additionally, the models are validated before and after the transformations to find discrepancies between the

core model and the instances.

We are also interested in the output Groove states, to investigate if the transformation from EMF to Groove was successful. These graphs have to be checked manually, because there is no way to automatically test the equivalence of Groove states and EMF models without using the transformation itself. For this reason, both the Groove states and the EMF models produced by the tests are stored in an output directory for manual inspection. As the test cases are deliberately small, manually checking the Groove states for correctness if necessary is no big problem.

All tests are written as JUnit 4 plugin tests [2], so they can be used to continually test the transformation if changes in the algorithms are implemented. Also, new tests can easily be implemented, as the main function of the test is independent of the input.

Now we will look at the tests individually. A special core model was used for the tests to ease identification of problems. The relevant parts of the model will be described and shown in the following sections.

5.1 Testing the Transformation of Attributes

In the testing of attributes, we differ between simple attributes and enumerations. Figure 18 shows the classes used in these test. The class AttrClass contains EAttributes of all supported EDataTypes (see Table 1) excluding EEnums. The attribute longDerived of type ELong is a derived value. This should make no difference in the transformation, but is tested for completeness. We test the transformation of these simple attributes using a model that contains two AttrClass objects. In one, the attributes all have their default values, in the other they all have non-default values. Again, the transformation should make no difference between this.

For EEnums, we have the class EnumClass with the attribute ENumAttr of type EEnum. EEnum is an enumeration with the literals LitA, LitB and LitC. Here we have two test models defined. In one, there is only one EEnumClass, in the other there are five, two with LitA, two with LitB and one with LitC as the value of EnumAttr. With these tests, the transformation of EEnums is completely tested.

5.2 Testing the Transformation of References

We test both ordered and non-ordered references in different tests. Containment references are tested in all tests, as the class Container is a root class containing all other classes, as customary in EMF core models. Figure 19 shows which classes are used in the tests of references. For reference test purposes, there is another class referenceClass that has different non containment references. Ordered references are tested using the classes orderedClass and singleOrdered. Container and referenceClass both have ordered references to these, for orderedClass it is a multiplicity-many reference, for singleOrdered it is a reference with the upper bound set to one. In the transformation, there should be next-nodes

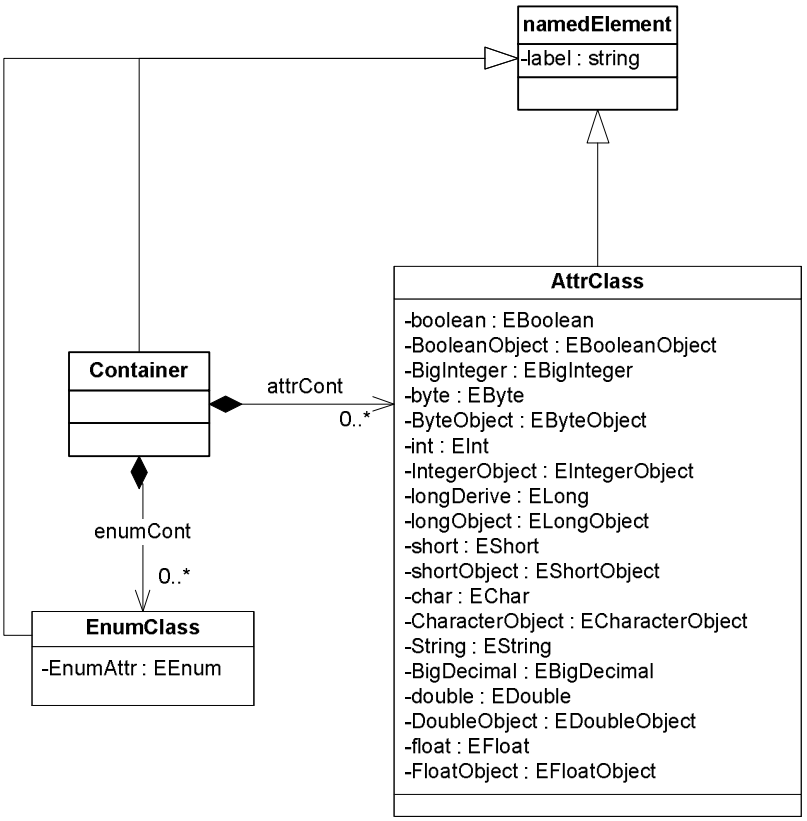

Figure 18: Test model classes used for attribute testing

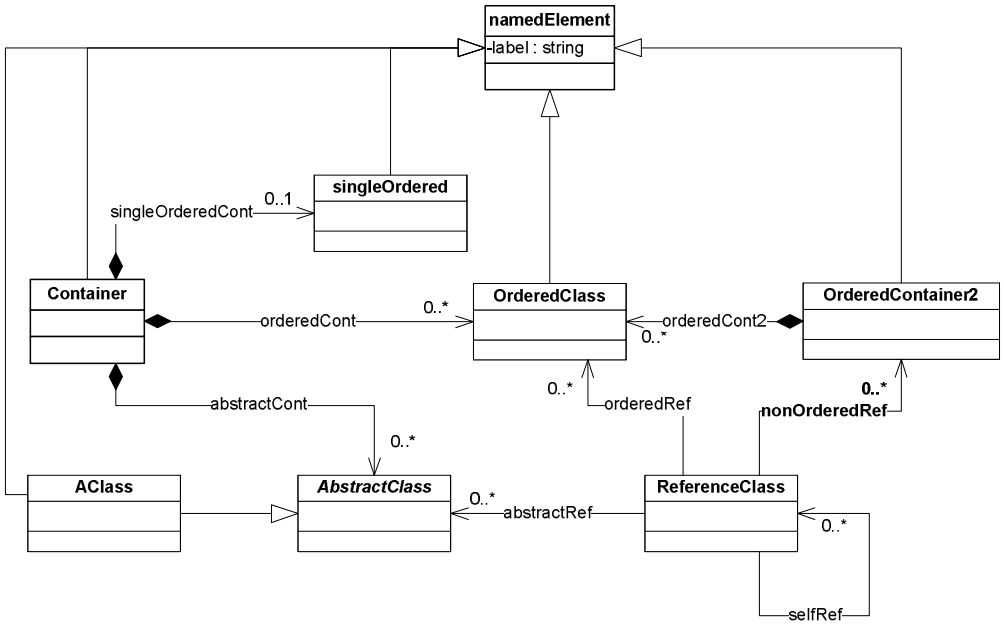

Figure 19: Test model classes used for reference testing

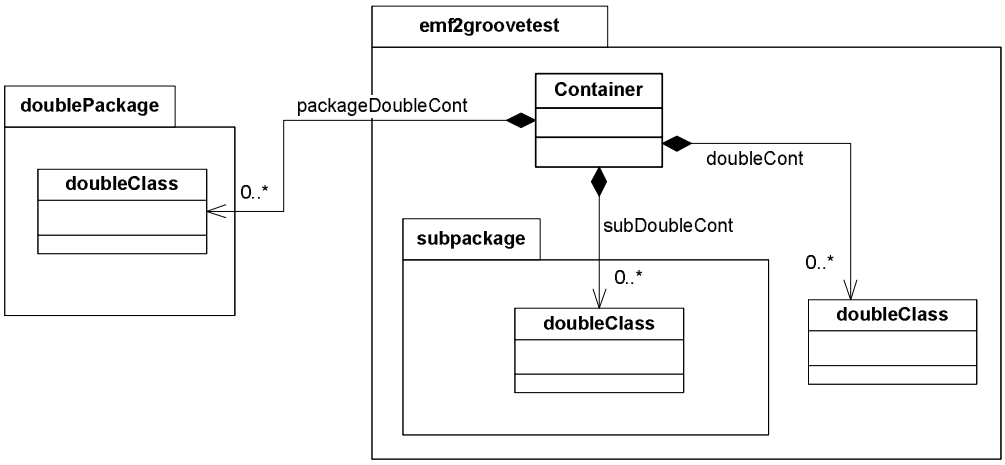

Figure 20: Test model classes used for testing of duplicate class names

created for the references that are ordered and many, but not for those that are ordered but not many.

There are five test models for references:

- manyOrderedOne uses the containment reference from Container to reference one orderedClass.
- manyOrderedFive uses the ordered reference from Container to orderedClass referencing five objects in order.
- singleOrderedCont uses the ordered single reference from Container to singleOrdered.
- twoOrderedRefs has two classes, Container and orderedContainer2, referencing the same orderedClasses in different orders.
- references tests non containment references using referenceClass.

5.3 Testing Duplicate Class Names

Figure 20 shows the classes used for the testing of duplicate class names. There are three classes doubleClass, all contained by Container. One doubleClass is a direct content of the emf2groovetest package, one is in a subpackage called subpackage and one is cross-referenced in a package doublePackage. The classes in the emf2groovetest package are also inheriting the label attribute from namedElement, which is not shown for clarity. We use a model with one of these classes each and one with five each. The transformations should transform these to Groove as doubleClass, subpackage.doubleClass and doublePackage.doubleClass, respectively. After the transformation back to an EMF model, the labels can be used to identify which doubleClass should be from which package.

5.4 Other Tests

There are two other test models. One tests inheritance using a class inheritClass that inherits from most other classes in the core model. Inheritance is also tested implicitly in the other tests, as all classes inherit their label attribute from namedElement, but this test does an explicit test with ten inheritances for the same object. Additionally, there is one "big" test model containing multiple objects of every class defined in the core model to test if the functionality tested with specialized test cases also works in the "big picture" of the whole model.

At the time this thesis was written, all tests returned positive results. They can also be used to test the transformation should changes in the implementation be made.

6 Conclusion and Outlook

To enable software developers to use graphical and at the same time formal models, Engels, Hausmann et al. developed Dynamic Meta Modeling (DMM) [7]. Using a DMM rule editor presented by Röhs [11] and a transformation of these to Groove rules, we can use the Groove toolset to inspect the transition system of models. What was missing was an automatic transformation from Eclipse Modeling Framework models to Groove states. Additionally, we would like to transform states identified by Groove as matching conditions that make them interesting to us back to EMF.

This thesis presented the bidirectional transformation between EMF models and Groove states necessary for that. We transform the EMF models to GXL graphs that can be read by Groove using the information from the ecore meta model. We support all attribute types that are supported by Groove, resolution of duplicate class names and ordered references. Using the generated code of the ecore model, we transform the state graphs output by Groove back to EMF models, so they can be inspected in the EMF editors again. Using a series of JUnit tests, we showed that the transformations work as expected for all supported EMF features. The bidirectional transformation presented in this thesis fills the gap in the connection of EMF and Groove that was present before.

We will now give an outlook on what the transformations could be used for. We could conceive an automatic DMM debugger using EMF models for the static and dynamic parts of the meta model. These are then transformed to Groove using the transformation from this thesis and an appropriate transformation for DMM rulesets. Using the ruleset, Groove can compute the transition system of the model and identify interesting states. These are then shown to the user and transformed back to EMF if he decides to do so, using the backwards transformation presented in this thesis. Figure 1 in Section 1 gives an overview over the static parts of this debugger.

What remains for the work on this transformation is to implement a graphical user interface. At the moment, the transformation runs as an Eclipse plugin but only provides a programming API that can be used by other Java programs to perform transformations from EMF to Groove or Groove to EMF.

References

[1] Eclipse modeling framework technology (emft) – emf compare, August 2008. `http://www.eclipse.org/modeling/emft/?project=compare`.

[2] junit.org resources for test driven development, August 2008. `http://www.junit.org`.

[3] Omg's metaobject facility (mof) home page, August 2008. `http://www.omg.org/mof/`.

[4] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows*. Prentice Hall, 1993.

[5] Frank Budinski, David Steinberg, Ed Merks, Raymond Ellersiek, and Timothy J. Grose. *Eclipse Modeling Framework*. Addison Wesley, 2003.

[6] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[7] Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel, and Stefan Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in uml. In *UML*, pages 323–337, 2000.

[8] Harmen Kastenberg. Towards attributed graphs in groove: Work in progress. *Electr. Notes Theor. Comput. Sci.*, 154(2):47–54, 2006.

[9] Arend Rensink. The groove simulator: A tool for state space generation. In *AGTIVE*, pages 479–485, 2003.

[10] Arend Rensink, Harmen Kastenberg, and Tom Staijen. *User Manual for the GROOVE Tool Set*. Department of Computer Science, University of Twente, March 2007.

[11] Malte Röhs. A visual editor for semantics specifications using the eclipse graphical modeling framework, 2008.

A List of Acronyms

API Application Programming Interface

DMM Dynamic Meta Modeling

EMF Eclipse Modeling Framework

EMOF Essential MOF

GMF Graphical Modeling Framework

Groove GRaphs for Object Oriented VErification

GXL Graph Exchange Language

MDA Model Driven Architecture

MOF Meta Object Facility

UML Unified Modeling Language

VML Visual Modeling Language

XML Extensible Markup Language

B List of Listings

1 Main method of the conversion from EMF to Groove 11
2 addEObject adds an EObject to the graph 12
3 Adding the attributes of an EObject to the according node . . . 13
4 Method adding the references to the graph 15
5 Package preprocessor to resolve duplicate class names 16
6 Creating the enum literal nodes 19
7 Main method of the transformation from Groove to EMF. 22
8 Transformation of a single node into an EObject 23
9 Find which EClass is represented by a node 23
10 Setting the attributes of an EObject 25
11 Populating the references of an EObject 25
12 Populating an Ordered Reference 27

C List of Figures

1 Overview over a bidirectional transformation 3
2 Example core model modeling a flow network 5
3 The simplified Ecore Kernel [5] 5
4 Ecore structural features [5] 6
5 Ecore Enumerations [5] 8
6 Ecore packages and factories [5] 8
7 Groove graph with and without self edges explicitly shown. . . . 10
8 A Groove graph with attributes. 11
9 Model of a simple flow network 11
10 Groove start state for the flow network 12
11 Groove graph before ERferences are added 15
12 Flow network with duplicate EClass name 17
13 Groove graph with resolved class names 18
14 Example Groove graph with EEnumerations 20
15 Flow network with ordered Nodes 20
16 Excample Groove graph with simple next edges. 21
17 Finished Groove graph of the example from Figure 15 22
18 Test model classes used for attribute testing 29
19 Test model classes used for reference testing 29
20 Test model classes used for testing of duplicate class names . . . 30