Project Number 611125

DSL-tao User Manual

Version 0.2

March 2015

Draft

miso research group

http://www.miso.es

Autonomous University of Madrid

Project Partners: ARMINES, Autonomous University of Madrid, BME, IKERLAN, Soft-Maint, SOFTEAM, The Open Group, UNINOVA, University of York

TABLE OF CONTENTS

1. INTRODUCTION

DSL-tao is a meta-model editor to support in the creation of large meta-models and facilitate the application of patterns. The editor has been realized as an Eclipse plugin targeting EMF as the meta-modelling technology for DSL's meta-model specification. It provides a graphical front-end for the definition of meta-models, and an extensible library of patterns which can be applied to the meta-models being constructed. The editor is available at http://jdelara.github.io/DSL-tao/.

To facilitate the manipulation of large meta-models, the editor provides facilities like:

1. **Hierarchical organization** of elements into packages.
2. The construction of meta-models by applying **patterns**; including an assistant for applying the pattern, and the visualization of the applied patterns and roles.
3. The availability of predefined queries to **obtain and visualize useful derived information** for a given meta-model element. For example, given a class, we can obtain and visualize all inherited attributes, all parent and children classes through inheritance, and all reachable classes by containment. This feature improves navigation and understandability of large meta-models.
4. **Filters** that simplify the visualization of large meta-models (e.g. collapsing or hiding certain elements, like pattern information or references).

2. INSTALLATION AND GENERAL ORGANIZATION OF THE EDITOR

The editor has been implemented as an Eclipse plugin. It can be downloaded from http://jdelara.github.io/DSL-tao/.

The editor provides a suitable perspective to work with the meta-model diagrams (the MONDO perspective). The perspective consists of four areas, as seen in Figure 1:

1. The project explorer.

2. A canvas where the diagram is built and displayed.

3. The palette, which provides the elements that can be added to the diagram.

4. The view area, which collects specific views to work with this kind of diagrams (available patterns, applied patterns, pattern services, element properties and derived information).

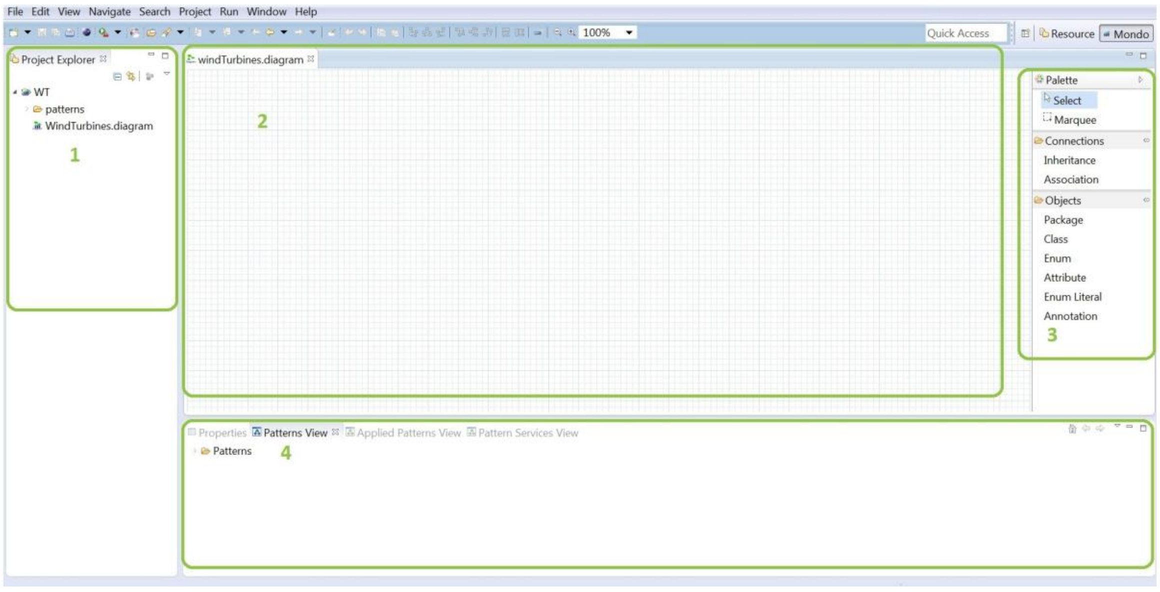


Figure 1: MONDO perspective, with the editor structure.

The next sections describe the main functionalities of the editor.

3. BASIC META-MODEL EDITING

Creation of MONDO projects and diagrams. The tool manages MONDO projects and diagrams. Figure 2 shows how to create a new MONDO project or diagram by invoking a wizard from the "File" menu of Eclipse.

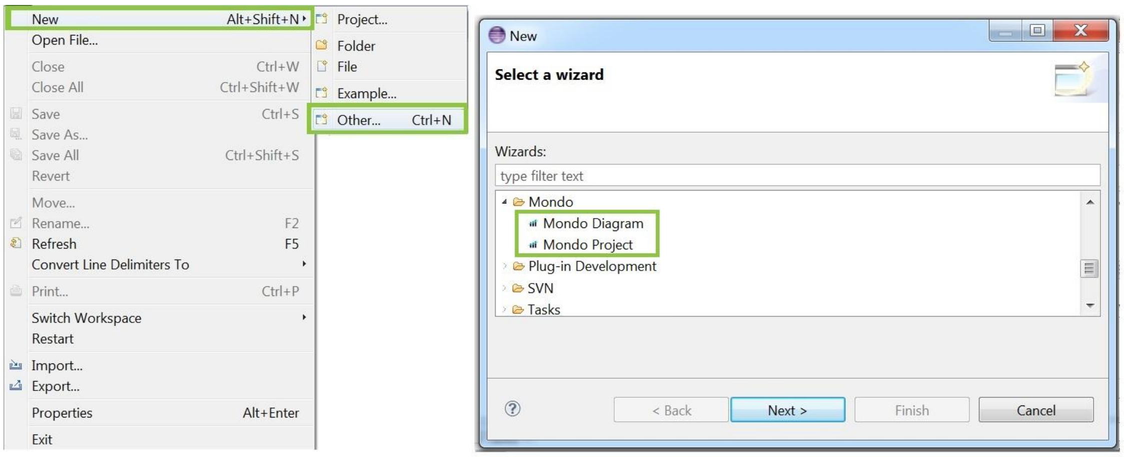


Figure 2: Creating a new MONDO project and diagram.

A MONDO project provides an environment with all the resources needed for the construction process of diagrams using patterns.

The new project includes a repository of patterns (within the folder called "patterns"), so the user has the possibility of adding new ones, and deleting and updating the predefined ones.

The project structure and the pattern definition model are showed in Figure 3.

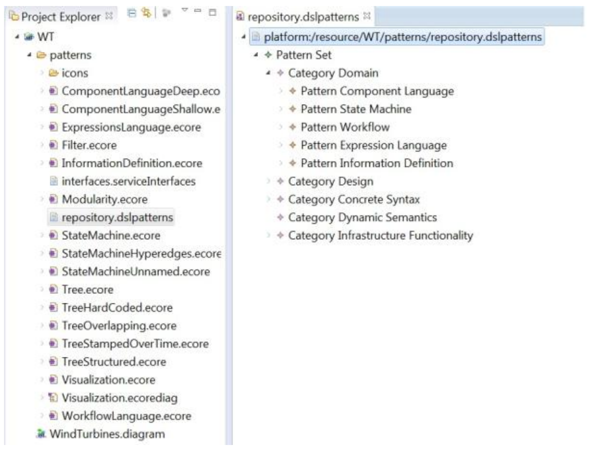

Project Explorer
WT
patterns
icons
ComponentLanguageDeep.eco
ComponentLanguageShallow.e
ExpressionsLanguage.ecore
Filter.ecore
InformationDefinition.ecore
interfaces.serviceInterfaces
Modularity.ecore
repository.dslpatterns
StateMachine.ecore
StateMachineHyperedges.ecore
StateMachineUnnamed.ecore
Tree.ecore
TreeHardCoded.ecore
TreeOverlapping.ecore
TreeStampedOverTime.ecore
TreeStructured.ecore
Visualization.ecore
Visualization.ecorediag
WorkflowLanguage.ecore
WindTurbines.diagram
repository.dslpatterns
platform:/resource/WT/patterns/repository.dslpatterns
Pattern Set
Category Domain
Pattern Component Language
Pattern State Machine
Pattern Workflow
Pattern Expression Language
Pattern Information Definition
Category Design
Category Concrete Syntax
Category Dynamic Semantics
Category Infrastructure Functionality

Figure 3: MONDO project structure.

A MONDO diagram consists of an underlying Ecore meta-model and a graphical representation for it. Diagrams can be created empty, or an existing Ecore meta-model can be imported and the tool provides an initial visualization for it.

There are two ways to import an existing Ecore meta-model into a MONDO diagram. The first one is from an existing MONDO diagram. This is done by right-clicking on the diagram canvas, as shown to the left Figure 4. As a result, the meta-model content is included on the diagram. The second possibility is to create a MONDO diagram from an Ecore meta-model, by right-clicking on the meta-model file in the package explorer, as shown to the right of Figure 4.

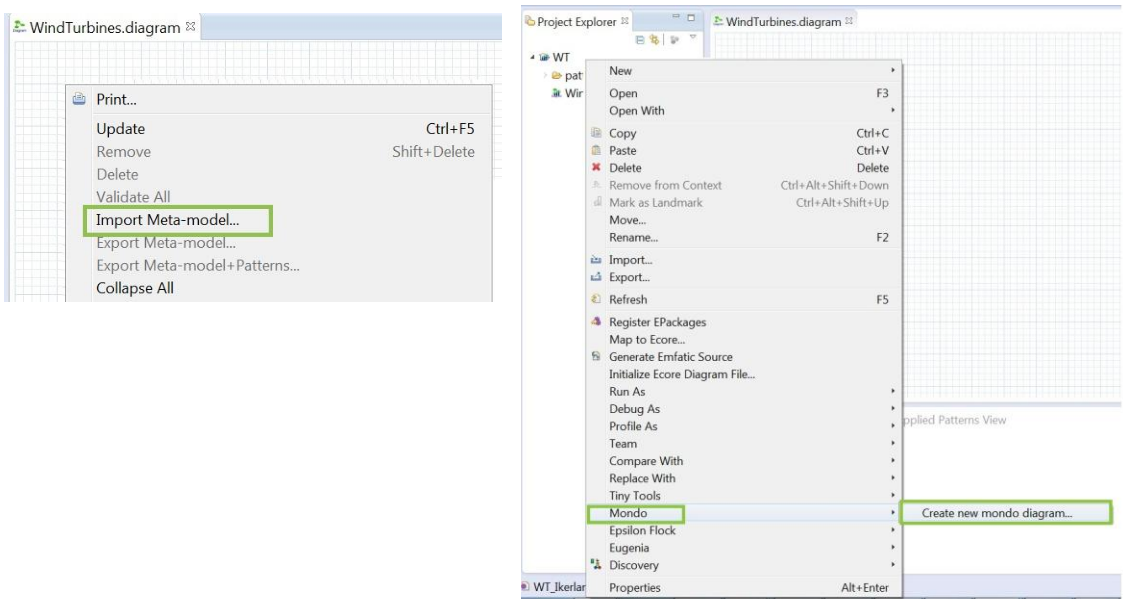


Figure 4: Importing an Ecore meta-model from the editing canvas (left). Creating a MONDO diagram from an Ecore meta-model (right).

When an Ecore meta-model is imported, either from the canvas or from an Ecore file, a new copy is created and linked to the MONDO diagram; thus, changes in the MONDO diagram will not be propagated to the original meta-model.

Interactive creation of meta-models. The editor permits creating classes, attributes, references, enumerations and enumeration literals. It allows organizing the elements in packages, and adding annotations to the different elements.

The addition of elements to the diagram is performed by "*drag and drop*" from a palette into the correct diagram container. For example, to create an attribute on a class, the attribute is to be dropped on the corresponding class container. A class is created by dropping a class into the diagram. The diagram contains an implicit package, which acts as the container for all classes.

Furthermore, association and inheritance connections can be created from de source class shape through its related popup menu.

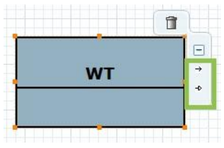
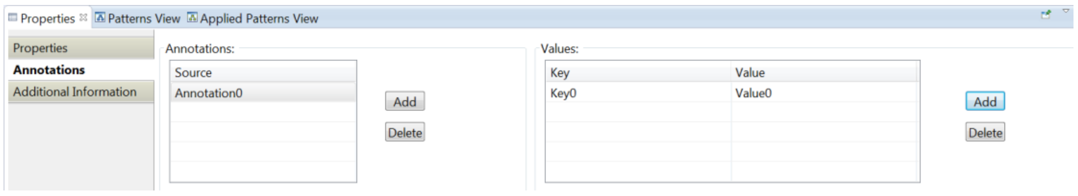


Figure 5: Connection popup menu

Elements in the meta-model can be annotated by the user. There are different mechanisms to work with annotations. First, the user can drop annotations into a given container. Second, annotations can be managed (addition, deletion and modification) using the property tab "*Annotations*", as shown in Figure 6.



Annotations:

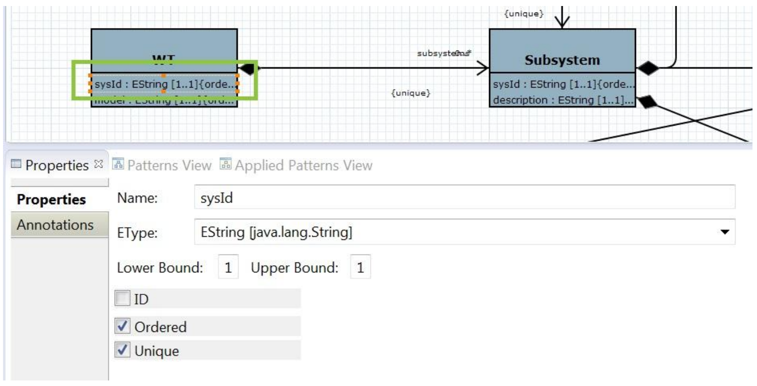
Source
Annotation0

Values:

Key	Value
Key0	Value0

Figure 6: Annotations tab in Properties view.

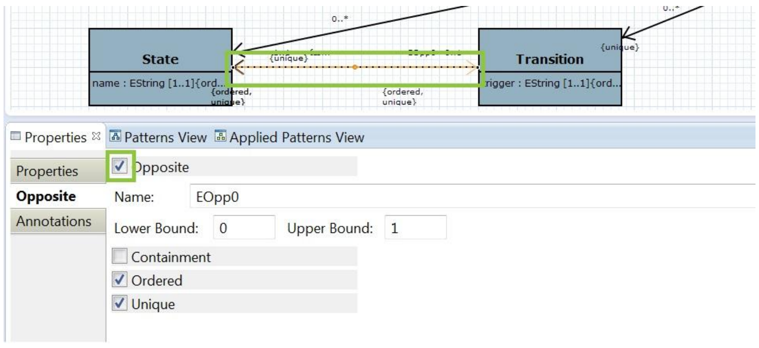
Editing properties. The different properties of elements can be edited. For this purpose, the editor provides a view that collects information about the selected element and permits changing its values, as shown in Figure 7 for the case of an attribute. Some properties, like the name of the class, can also be changed directly on the associated pictogram in the canvas.



Name: sysId
EType: EString [java.lang.String]
Lower Bound: 1 Upper Bound: 1
☐ ID
☑ Ordered
☑ Unique

Figure 7: Properties view.

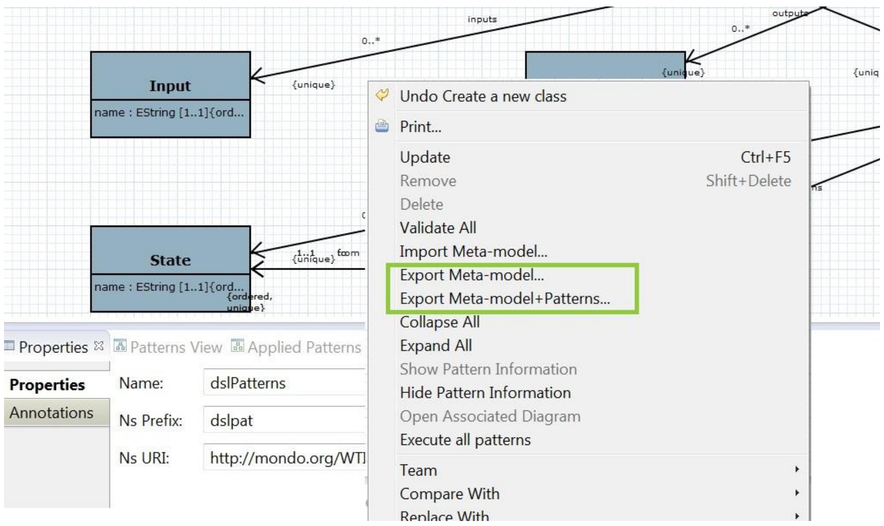
In addition, this view also enables the creation and editing of opposite references. Two opposite references are visualized as a single line, but the properties of each role can be edited separately in two different tabs, as shown in Figure 8. To configure whether a reference is bidirectional (i.e. it has an opposite role) or unidirectional, the "*Opposite*" check-box in the Opposite tab should be used, as shown highlighted in the figure.



☑ Opposite
Name: EOpp0
Lower Bound: 0 Upper Bound: 1
☐ Containment
☑ Ordered
☑ Unique

Figure 8: Opposite references tab in Properties view.

Exporting a meta-model into Ecore. At any moment, the meta-model being constructed can be exported into the Ecore format, using a contextual menu option, as shown in Figure 9.



Name:	dslPatterns
Ns Prefix:	dslpat
Ns URI:	http://mondo.org/WTI

Figure 9: Exporting a meta-model into Ecore.

Additionally, the information related to the patterns used can be exported as well using the option "Export Meta-model+Patterns..." as seen in Figure 9. This option will create two files into the destination directory selected by the user.

Before the export is performed, a validation of basic correctness properties of the meta-model is automatically performed, including:

- Uniqueness of attribute and reference names in inheritance hierarchies.
- Uniqueness of ID attribute in inheritance hierarchies.
- Valid URI and name for each package.

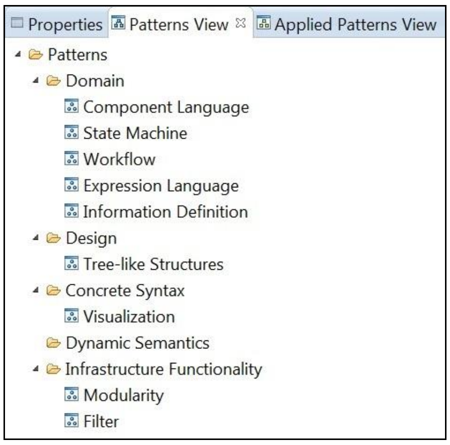
Other correctness properties, regarding for example valid names, or the acyclicity of inheritance relations are checked during the editing of the meta-model.

Note: Once exported, the diagram and the exported meta-model become independent, that is, the changes performed on the diagram will not be reflected on the meta-model, and vice versa.

4. PATTERN-BASED DEVELOPMENT

DSL-tao enables the construction of meta-models by means of libraries of reusable patterns. In this way, the meta-model developer can select patterns and apply them on the existing meta-model with the help of an assistant. Applying a pattern will result on the creation of new meta-model elements and the annotations of the new (or existing) elements with the different pattern roles. Such roles can be used by pattern processors, for example to contribute features to the generated DSL environment.

The editor contains a "patterns view", showing the collection of patterns that can be applied to the meta-models, organized in categories, as shown in Figure 10.

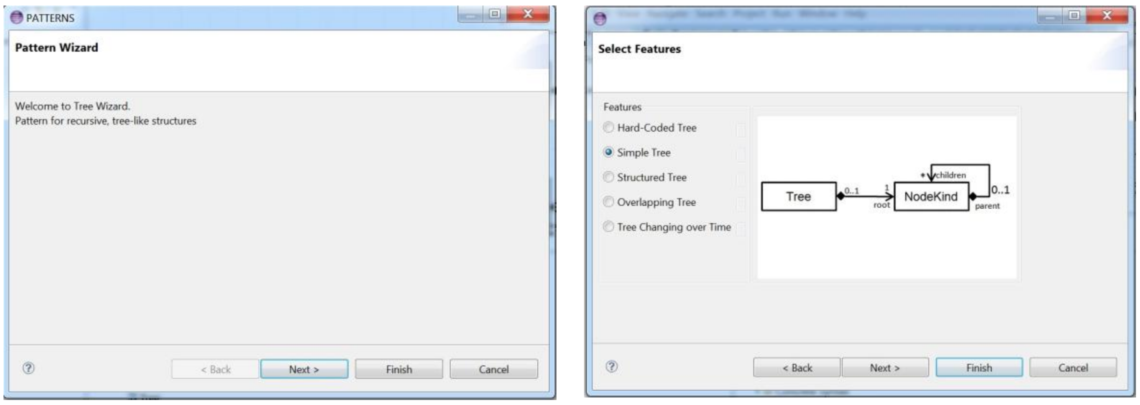


- Patterns
  - Domain
    - Component Language
    - State Machine
    - Workflow
    - Expression Language
    - Information Definition
  - Design
    - Tree-like Structures
  - Concrete Syntax
    - Visualization
  - Dynamic Semantics
  - Infrastructure Functionality
    - Modularity
    - Filter

Figure 10: Patterns view.

Next, we explain the different phases in pattern usage: (1) pattern selection, (2) pattern variant selection, (3) pattern role expansion, (4) pattern role binding, and (5) pattern application.

(1)(2) Pattern and variant selection. To use a pattern, the user must select it, and a wizard will appear guiding the user in the application process, as shown in Figure 11. The first page in the wizard shows a brief description about the selected pattern. In the next one, the user can choose between different pattern variants and visualize their graphical representation. As an example, the figure shows some variants of a pattern for tree-based structures.



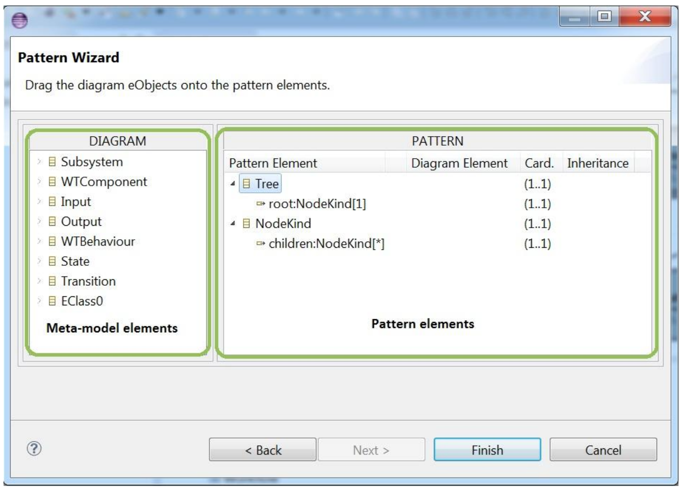
Welcome to Tree Wizard.
Pattern for recursive, tree-like structures

Features
- ○ Hard-Coded Tree
- ● Simple Tree
- ○ Structured Tree
- ○ Overlapping Tree
- ○ Tree Changing over Time

Figure 11: Assistant for pattern application.

(3) Pattern role expansion. Once a pattern variant is selected, the last page of the wizard allows the user to expand the pattern roles and bind those roles to meta-model elements, as shown in Figure 12.

There are two main parts in this page. The first part (on the left) is a tree view that contains the meta-model elements of the saved diagram. The second part (on the right) is a tree-table that collects information about the structure of the pattern, their allowed expansion cardinalities, and the application of the pattern (the elements of the meta-model to which they have been bound).


Pattern Wizard

Drag the diagram eObjects onto the pattern elements.

DIAGRAM
- Subsystem
- WTComponent
- Input
- Output
- WTBehaviour
- State
- Transition
- EClass0

Meta-model elements

PATTERN

Pattern Element	Diagram Element	Card.	Inheritance
Tree		(1..1)	
root:NodeKind[1]		(1..1)	
NodeKind		(1..1)	
children:NodeKind[*]		(1..1)	

Pattern elements

Figure 12: Wizard for pattern instantiation.

As explained in Section 1, applying a pattern consists on: *(i)* expanding the pattern roles according to their allowed expansion cardinality, and *(ii)* binding the pattern roles to existing meta-model elements. The pattern roles left unbound will be instantiated in the meta-model, while the bound elements in the meta-model will receive an annotation (a "*type*"), which may be used by pattern services to perform different tasks.

Pattern roles can be expanded according to their expansion cardinality. This is done by right-clicking on the pattern role, as shown to the left of Figure 13.

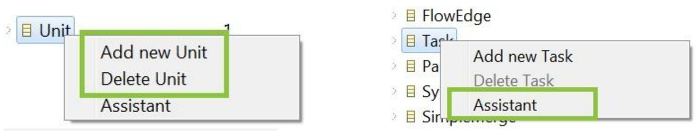


Figure 13: Expanding a pattern role (left). Invoking the assistant for role binding (right).

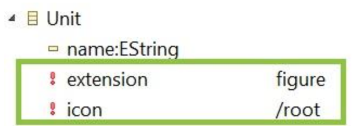
(4) Binding the pattern roles. Once expanded, the pattern roles should be bound to the meta-model elements. The wizard enforces well-formedness rules for the binding. For example, the tool only allows binding elements of the same nature, that is, classes with classes, references with references, and so on. In addition, the attributes can only be related when its container class is assigned and the references when the source and target classes are assigned.

Furthermore, the wizard contains an assistant that recommends the optimal element/s to be bound to, for a selected pattern role. This mechanism can be activated when the pattern role is selected, as shown to the right of Figure 13.

Two heuristics are used for the recommendation, one is generic (implemented in the tool), and the second is domain-specific, provided together with the pattern through an extension point. The generic heuristic works as follows:

- If the pattern element is a class: the heuristic recommends classes with the same number of attributes, containment references, reflexive references, and the rest of references. Please note that we consider structural matches, and hence, inherited features are also checked.
- If the pattern element is an attribute: the heuristic recommends attributes that belong to the class that is bound to the attribute owner, and have the same type.
- If the pattern element is a reference: the heuristic recommends references that belong to the class that is bound to the reference owner, have the same cardinality, and have the same type (i.e. point to the correct class). Opposite references are checked as well.

Class roles in patterns may include "*constant attributes*", which are not meant to be bound, but a value should be provided for them instead. Conceptually, patterns are located one meta-level above the current meta-model, and therefore, such attributes can be thought of having "*potency*" 1 as they have to be assigned a value at the meta-model level, while "*normal*" attributes are given a value at the model level. As Figure 14 shows, these two different attribute types are represented differently, and the figure shows the values given to the "*constant*" attributes extension and icon.



Unit
name:EString
extension figure
icon /root

Figure 14: "constant" and "normal" attributes in patterns.

(5) Pattern application. Once the binding is performed, the pattern can be applied. The bound roles are translated into instances of a pattern meta-model (this model is stored within the diagram) The roles that are left unbound but are mandatory (minimum expansion cardinality of 1) are created in the meta-model. Figure 15 shows the result of a pattern application, where the pattern roles are shown in black.

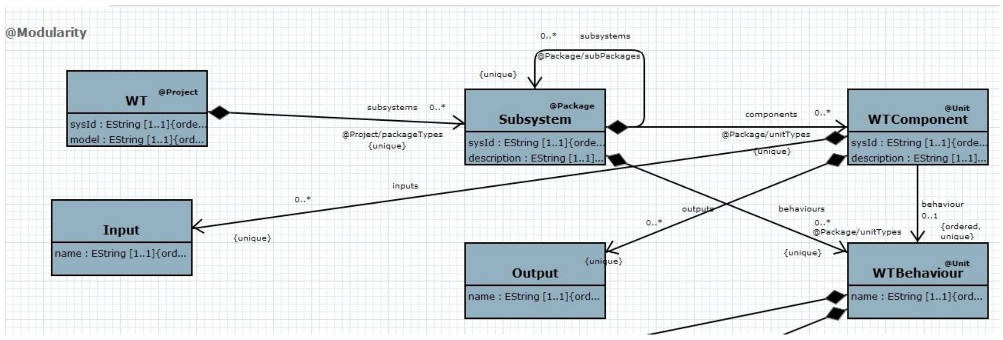

@Modularity
WT @Project
sysId : EString [1..1]{orde...
model : EString [1..1]{ord...
subsystems 0..*
@Project/packageTypes {unique}
0..* subsystems
@Package/subPackages
{unique}
Subsystem @Package
sysId : EString [1..1]{orde...
description : EString [1..1]...
components 0..*
@Package/unitTypes {unique}
WTComponent @Unit
sysId : EString [1..1]{orde...
description : EString [1..1]...
inputs 0..*
Input
name : EString [1..1]{ord...
{unique}
outputs 0..*
behaviours 0..* @Package/unitTypes
{unique}
behaviour 0..1 {ordered, unique}
Output
name : EString [1..1]{ord...
{unique}
WTBehaviour @Unit
name : EString [1..1]{ord...

Figure 15: Result of a pattern application.

4.1 PATTERN SERVICES

Each pattern may have associated a service, which can be invoked through the meta-model editor. Typically, these services are provided through an extension point, and will provide functionalities for the generated modelling environment.

This functionality is based on the extension point shown schematically in Figure 16.

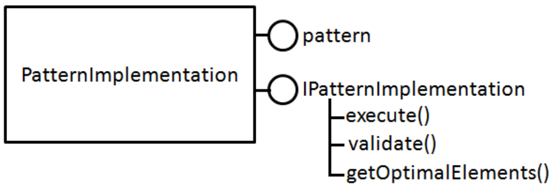


Figure 15: Scheme of extension point for patterns.

To define an implementation associated to a pattern, it is necessary to indicate the name of the pattern (attribute `pattern` in the figure), and implement the interface `IPatternImplementation`, which defines the methods `execute`(), `validate`() and `getOptimalElements`(). The `execute` method will typically call a code generator contributing to the generated plugin. The `validate` method provides domain-specific checkings to assert the correct instantiation of the pattern. The validation result is displayed in the diagram, as shown in Figure 17. The `getOptionalElements` method is used by the binding assistant to provide domain-specific heuristics on the most appropriate meta-model element to bind a given pattern role.

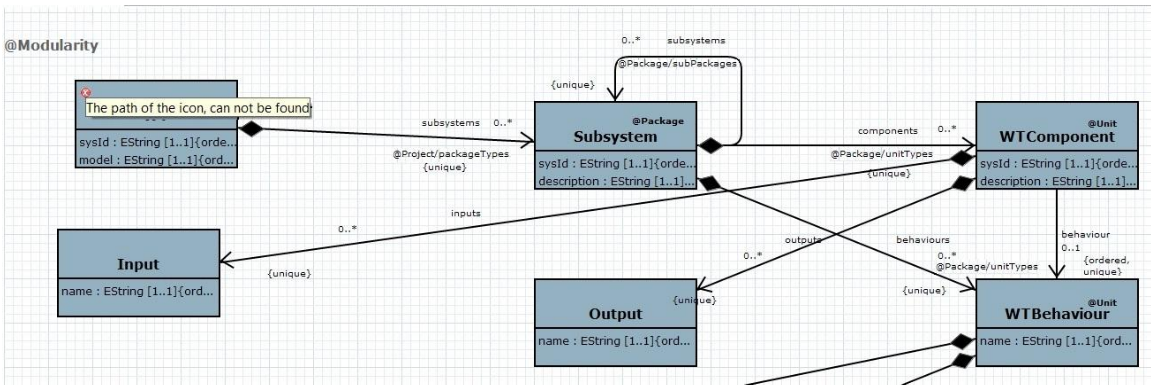

@Modularity
The path of the icon, can not be found
sysId : EString [1..1]{orde...
model : EString [1..1]{ord...
subsystems 0..*
@Project/packageTypes {unique}
0..* subsystems
@Package/subPackages
{unique}
@Package
Subsystem
sysId : EString [1..1]{orde...
description : EString [1..1]...
components 0..*
@Package/unitTypes {unique}
@Unit
WTComponent
sysId : EString [1..1]{orde...
description : EString [1..1]...
inputs
0..*
{unique}
Input
name : EString [1..1]{ord...
outputs
0..*
behaviours
0..*
@Package/unitTypes
{unique}
behaviour 0..1
{ordered, unique}
Output
name : EString [1..1]{ord...
@Unit
WTBehaviour
name : EString [1..1]{ord...

Figure 17: Result of pattern validation.

The same pattern can be applied several times (the maximum number of instances is indicated in the pattern definition). Each application of a same pattern has its own options in the action menu as shown in Figure 18. In addition, all of them can be validated and executed from a unique menu option.

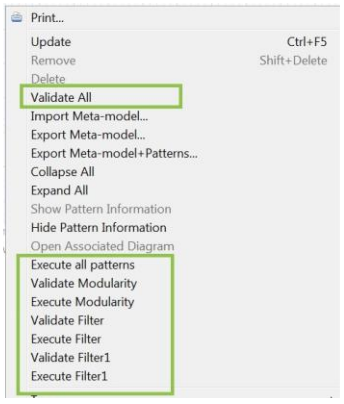
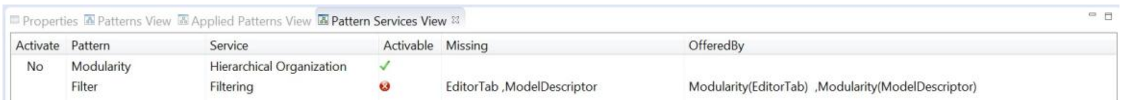


Figure 18: Service menu options.

The service provided by some pattern, may require the services provided by other patterns. Moreover, some services can be optional. The user can configure the services he wants to use, using the "Pattern Services View". Figure 19 shows how the service "Hierarchical Organization" is required by the service "Filtering". In the case of an unbound service, the view shows the patterns that offer such service.



Activate	Pattern	Service	Activable	Missing	OfferedBy
No	Modularity	Hierarchical Organization	✓		
	Filter	Filtering	✗	EditorTab ,ModelDescriptor	Modularity(EditorTab) ,Modularity(ModelDescriptor)

Figure 19: Pattern services view.

5. Visualization Mechanisms

The editor provides several mechanisms to facilitate the visualization and exploration of large-scale meta-models, which are described next.

- **Collapsing/expanding filters.** In order to visualize a meta-model in a simpler way, it is possible to filter some information and display only the name of the classes, references and applied pattern names (in *collapse* mode). Furthermore, the editor allows restoring to the expanded mode.

  This functionality can be used in two ways: individually by collapsing/expanding each class, or globally by applying the filter to all classes in the diagram. Both modes are illustrated in Figure 20.

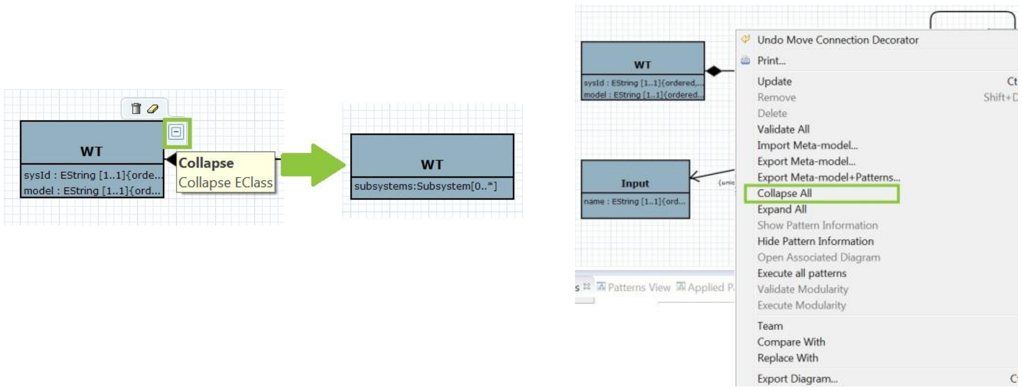


Figure 20: Collapse option for individual classes (left). Global collapse option (right).

- **Showing/hiding pattern role information.** The editor depicts on the diagram the pattern information applied to the different meta-model elements. This information can be hidden or displayed, as Figure 21 shows.

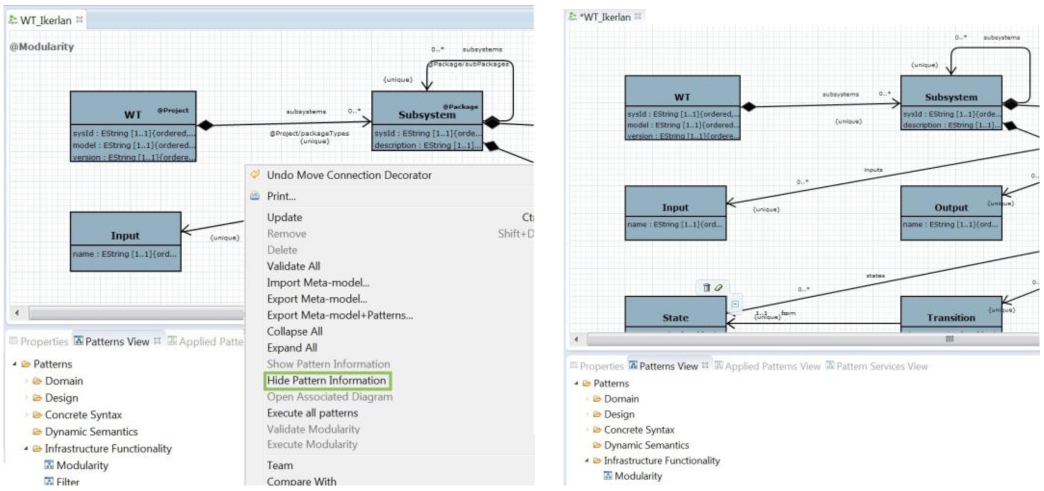


Figure 21: Showing and hiding annotations for pattern roles.

- **Highlighting applied patterns**. To emphasize how the patterns have been applied, the editor provides a view with the applied patterns and their roles. When a pattern is selected, all their elements are highlighted and the rest of the meta-model is hidden. This feature is illustrated to the left of Figure 22.

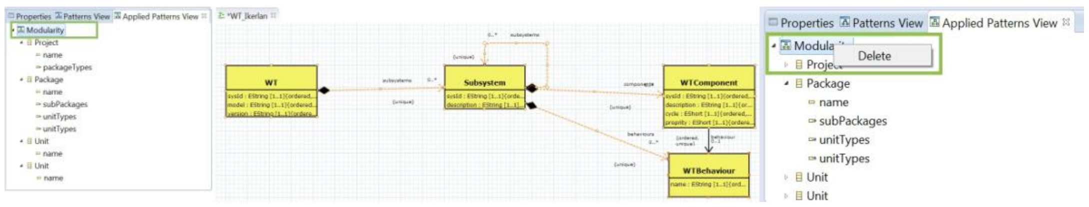


Figure 22: Highlighting applied patterns (left). Unbinding pattern roles to meta-model elements (right).

This view also enables the management of the applied patterns, allowing the deletion of pattern role bindings from the different meta-model elements, as shown to the right of Figure 22.

- **Highlighting derived information.** To facilitate the exploration of large meta-models, we provide some predefined queries to visualize derived information. Hence, given a selected class, we provide all its supertypes, subtypes, attributes and references (owned and inherited). This feature is illustrated in Figure 23.

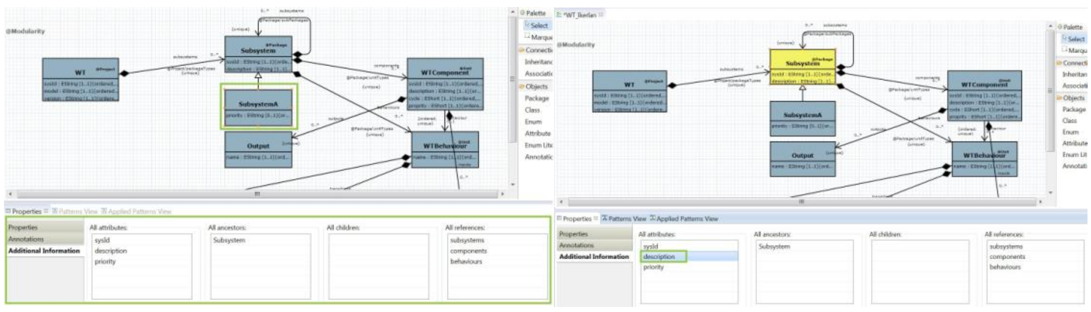
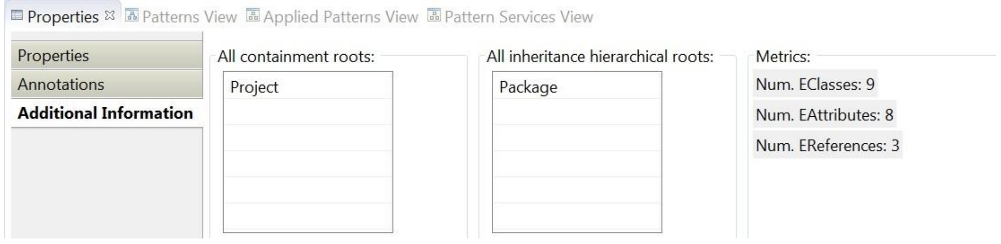


Figure 23: Showing derived information for a given class.

For packages, their additional information tab shows the list of all the containment and inheritance hierarchical roots which are contained by the package. In addition, information about some metrics such as the total number of classes, attributes and references is included as seen in Figure 24.



Properties | Patterns View | Applied Patterns View | Pattern Services View

Properties
Annotations
Additional Information

All containment roots: Project

All inheritance hierarchical roots: Package

Metrics:
Num. EClasses: 9
Num. EAttributes: 8
Num. EReferences: 3

Figure 24: Showing additional information for a given package.

- **Package navigation.** The editor supports the organization of a meta-model into packages, and the exploration of package contents, as shown in Figure 25.

The content of a package is shown in another diagram, which can be edited as usual. It is also possible to navigate back to the container package.

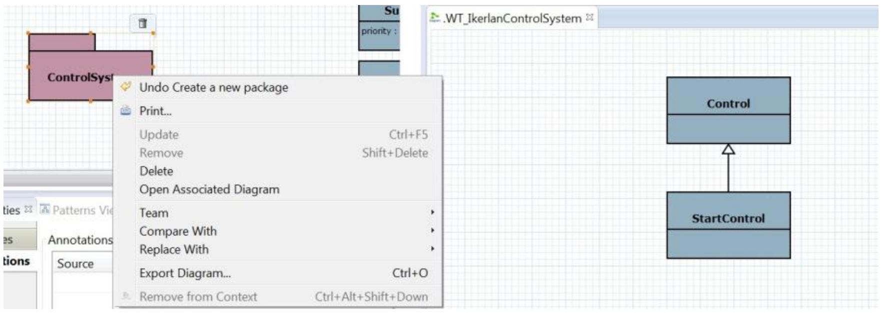


Figure 25: Expanding the contents of a package.