Henshin Cheat Sheet

INTRODUCTION TO HENSHIN

Henshin is a graph-based model transformation language for the **Eclipse Modeling Framework** (**EMF**).

- Henshin supports two types of transformations:
  - o **Endogenous**: direct transformations of EMF single model instances.
  - o **Exogenous**: translation of source model instances into a target language.
- **URL**: https://www.eclipse.org/henshin/

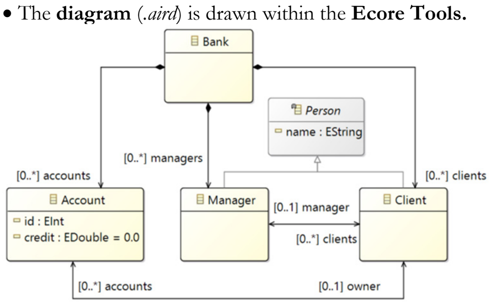
INSTALLATION

Henshin is a plug-in for the **Eclipse Modeling Tools.**

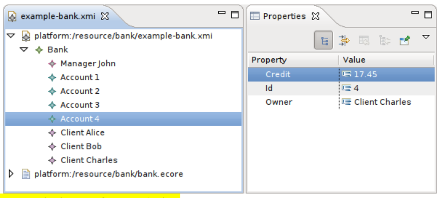
- **Eclipse URL:** https://www.eclipse.org/
- **Update sites**:
  - o **Current release:** http://download.eclipse.org/modeling/emft/henshin/updates/release
  - o **Nightly builds:** http://download.eclipse.org/modeling/emft/henshin/updates/nightly
- **Source code**: http://git.eclipse.org/c/henshin/org.eclipse.emft.henshin.git/

METAMODELS AND INSTANCE MODELS

- The **metamodel** is defined in **EMF** (*.ecore* files).
- The **diagram** (*.aird*) is drawn within the **Ecore Tools.**



- An **instance model** (*.xmi*) is typed in the **Sample Reflective Ecore Model editor**.

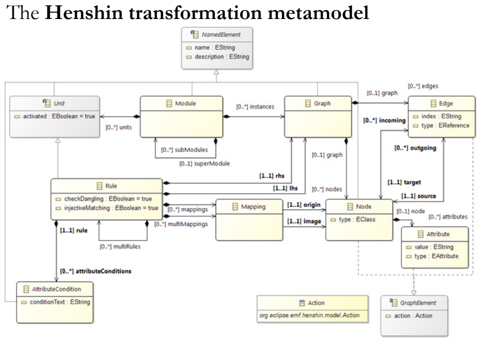


Dynamicity of models:

- **Static EMF:** the instance model (*.xmi*/*.bank*) is defined using the generated code from the metamodel. It references the metamodel purely by its URI.
- **Dynamic EMF:** the instance model (*.xmi*) is defined over a dynamic metamodel. It includes a **SchemaLocation** with the location of the metamodel (the *.ecore* file).
- A **static implementation** of the metamodel may exists,

HENSHIN TRANSFORMATION MODELS

- The **transformation model** is defined in a *.henshin* file.
- The **diagram information** (*.henshin_diagram)* is specified within the **Henshin Graphical Editor**.
- The **Henshin transformation metamodel**



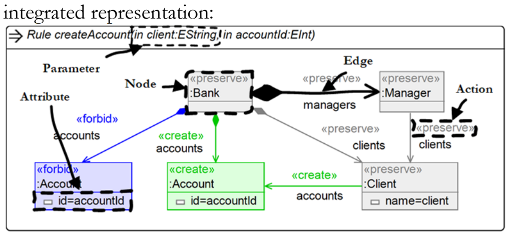
- A **Module** (*.henshin*) encapsulates a complete specification as a container for a set of units. There are two kind of units:
  - o **Rules** are the basic building blocks for **model transformations**.
  - o **Composite units** enable the orchestration of multiple rules in a control flow.
- **Static rules:** the module (*.henshin*) references the metamodel by its URI. *EPackage* imported from **Registry**.
- **Dynamic rules:** the module references the metamodel by its *.ecore* file. *EPackage* imported from **Workspace**.

TRANSFORMATION RULES

A rule comprises two graphs:

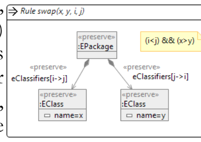
- **Left-hand side (LHS) graph**: it describes a pattern to be matched in the input model.
- **Right-hand side (RHS) graph**: it specifies a change on the input model.

A **graph** specifies model patterns on the abstract syntax level. The **graphical editor** merges LHS and RHS into an integrated representation:



- **Nodes** represent model elements.
- **Edges** represent references between model elements.

(1) Edges are identical if their types, sources and targets are identical. (2) If a ref. has an `EOpposite` ref., it's not required to specify an edge for the opposite ref. (3) In *ordered* ref., the positions of list entries can be modified using the *index* attribute.



Dangling condition: a rule execution may not leave behind **dangling edges**, being edges with a missing source or target.

- **Attributes** in nodes represent the attributes of the respective model elements. Their values may be literals (0, 'Hello'), parameters, or JavaScript expressions.
- **Parameters** allows to shape the behavior of units and rules with variable information. Parameters have a name, a description, a kind, and, optionally, a type.

Parameters kinds:

- o `in`: it's passed into the unit/rule from the its context.
- o `out`: it's passed out of the unit/rule into the its context.
- o `inout`: it's passed both into and out.
- o `var`: it's a variable used internally inside a unit/rule.

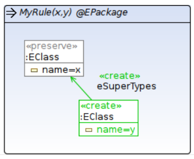
Alter parameter values of kind *inout* using `->`.

Parameters types (optional) can be an arbitrary `EClassifier`:

- o **EDataType:** `EString`, `EInt`, `EBoolean`,…
- o **EClass**

Transparent containers:

All objects (except the root) should be part of a containment (have a unique parent). **'@'** followed by a *type* after the rule name indicates that all objects in the rule will be automatically treated as children of an object of that type.



- **Actions:** nodes and edges are annotated with stereotypes («») which refer to actions:
  - o **«preserve»**: matches an object and preserves it.
  - o **«create»**: creates a new object or edge.
  - o **«delete»**: deletes an existing object or edge.
  - o **«forbid»**: forbid the existence of an object or edge.
  - o **«require»**: requires the existence of an object or edge.

Nodes and edges occurring in LHS are **«delete»**.
Nodes and edges occurring in RHS are **«create»**.
Node mappings between LHS and RHS are **«preserve»**.

Application conditions are graph patterns that restrict the LHS of a given rule

- o **Positive Application Conditions (PACs)** requires the presence of additional elements or relationships not included in the LHS. **«require»**
- o **Negative Application Conditions (NACs)** forbides the presence of elements or relationships. **«forbid»**

PACs and NACs aren't part of a computed match. A **match** only contains mappings for LHS nodes. This is important if you want to apply a rule for each computed match.

Parametrization to distinguish multiple PACs/NACs (#)

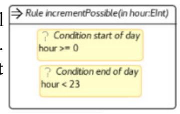
«require#1» Graph element is part of a PAC named *1*.
«forbid#myNAC» Graph is element of a NAC named *myNAC*.

Rule-nesting (*):

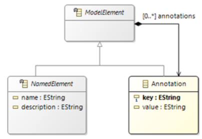
In **nested rules**, the outer rule is referred as **kernel rule** and the inner rule as **multi-rule**. During execution, the kernel rule is matched and executed once. Then, the match is used as a starting point to match and execute the multi-rule as often as possible. **Multi mappings** allow to specify identity between kernel and multi-rule nodes. Multi-rules nodes are indicated by a layered representation and an *. Examples:

«preserve*» Preserve all matching (default multi-rule).
«delete*/multi» Delete all matching (multi-rule *multi*).
«create*/my/nested/rule» Create an element in a nested multi-rule.
«require*/my/nested/rule#1» Named PAC in a nested multi-rule.
«forbid*/my/nested/rule#myNAC» Named NAC in a nested multi-rule.

- **Conditions** impose additional Boolean conditions to rules. Conditions are **JavaScript** expressions.



- **Annotations** are a mechanism for supporting non-intrusive extensions of the Henshin language. Each model element from a Henshin module can be annotated with Annotations. To this end, each metaclass transitively inherits from `ModelElement`, which has an arbitrary number of annotations. An annotation has a key and a value, both being strings.



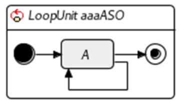
UNITS

Units specify control flow. Units have a fixed number of sub-units, allowing for arbitrary nesting.

Unary units: exactly one sub-unit.

Loop unit

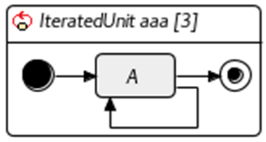
Apply A as often as possible.



- **Flags/Properties:** none
- **Execution successful:** always
- **Control flow:** sub-unit is executed as often as it can.

Iterated unit [x]

Apply the A x times.

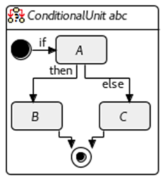


- **Flags/Properties:**
  - o *iterations* (int)
  - o *strict* (bool)
  - o *rollback* (bool)
- **Execution successful if:**
  - o *strict=true*: all iterations successful.
  - o *strict=false*: at least one iteration successful.
- **Control flow:** sub-unit is executed as often as specified in the *iterations* property.
  - o *strict=false*: if one inter. fails, the next iter. is exec.
  - o *strict=true, rollback=false*: if one iter. fails, exec. stops.
  - o *strict=true, rollback=true*: if one iter. fails, exec. stops and previous executions are reverted.

Conditional units: two or three sub-units.

Conditional unit

If A applicable then apply B, else apply C. (C is optional)



- **Flags/Properties:** none
- **Execution successful if:**

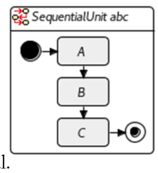
if unit and *then* unit are successful or *if* unit is unsuccessful while *else* unit is successful or not present.

- **Control flow:** if a match for the *if* unit can be found, the *then* unit is executed. Otherwise, if present, the *else* unit is executed.

Multi-units: arbitrary number of sub-units.

Sequential unit

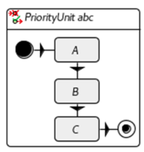
Apply A then B then C.



- **Flags/Properties:**
  - o *strict* (bool)
  - o *rollback* (bool)
- **Execution successful if:**
  - o *strict=true*: all sub-units successful.
  - o *strict=false*: at least one sub-unit successful.
- **Control flow:** sub-units are exec. in the given order.
  - o *strict=false*: if one sub-unit fails, the next one is exec.
  - o *strict=true, rollback=false*: if one fails, exec. stops.
  - o *strict=true, rollback=true*: if one sub-unit fails, exec. stops and previous executions are reverted.

Priority unit

Try to apply A. If A not applicable, try B etc.



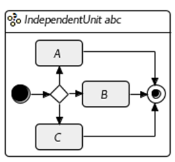
- **Flags/Properties:** none
- **Execution successful if:**

one sub-unit successful.

- **Control flow:** sub-units are checked in the given order for executability. The first found is executed.

Independent unit

Choose A, B or C randomly. If not applicable try another one randomly. If not applicable…



- **Flags/Properties:** none
- **Execution successful if:**

one sub-unit successful.

- **Control flow:** sub-units are checked randomly for executability. The first sub-unit found is executed.

Henshin Cheat Sheet

INTERPRETER API

- **Dependencies:**
  - o **org.eclipse.emf.henshin.interpreter** for Henshin.
  - o **org.eclipse.emf.ecore.xmi** for loading models/rules.
- **Loading and saving models:**

```
// Create a resource set for the working directory
HenshinResourceSet rs = new HenshinResourceSet("dir/");

// If static metamodels, register them
rs.getPackageRegistry().put(MetamodelPackage.eINSTANCE.
getNsURI(), MetamodelPackage.eINSTANCE);

// If dynamic metamodels, register them
rs.getPackageRegistry().put(metamodel.getNsURI(),
metamodel);

// Load a model
Resource res = rs.getResource("model.xmi");
EObject model = res.getContents().get(0);

// Load a Henshin module
Module module = rs.getModule("module.henshin", true);

// Apply the transformation (see below)...

// Save the model
Resource resT = rs.createResource("transformed.xmi");
resT.getContents().add(model);
resT.save(null);
```

- **Applying transformations:**

```
// Prepare the engine (it should be always reused)
Engine engine = new EngineImpl();

// Initialize the graph (reuse as possible)
EGraph graph = new EGraphImpl(model);

// Find the unit/rule to be applied
Unit unit = module.getUnit("myUnit");

// Prepare application of the unit/rule (don't reuse)
UnitApplication app = newUnitApplicationImpl(engine,
                                   graph, unit, null);

// Execute the unit/rule (see ApplicationMonitor below)
app.execute(null);
```

- **Alternative classes for transformations:**

```
// Update the contents of the resource based on EGraph
InterpreterUtil.applyToResource(unit, engine, res);
```

  - o `RuleApplication`: to apply a single rule and specify partial or complete matches.
- **Setting and getting parameters:**

```
// Assign parameters values before execution
app.setParameterValue("p1", "HelloWorld");
app.setParameterValue("p2", object);

// Retrieve the resulting values after execution
Object newValue = app.getResultParameterValue("p1");
```

- **Monitors: Canceling, Logging, and Profiling:**

`ApplicationMonitor` instances for the `execute()` method that allows to inspect and cancel unit/rule applications:

  - o `BasicApplicationMonitor` basic implementation.
  - o `LoggingApplicationMonitor` for logging.
    `setAutoSaveURI(URI)` saves intermediate results.
    `setMaxSteps(int)` aborts the execution after n steps.
  - o `ProfilingApplicationMonitor` for statistics.
    `printStats()` shows execution times for rule.
- **Finding matches:**

```
// Create a partial match
Match pMatch = new MatchImpl(rule);
pMatch.setParameterValue(p1, "foo");

// Iterate over all matches
for (Match m : engine.findMatches(rule,graph,pMatch)) {
  System.out.println(m);
}

// Alternative to find all matches
InterpreterUtil.findAllMatches(...)
```

- **Checking graphs/resources isomorphy:**

```
InterpreterUtil.areIsomorphic(graph1, graph2)
```

- **Engine options:**

```
Engine.getOptions().put(option, false);
```

  - o `Engine.OPTION_DETERMINISTIC`: deterministic rule application (default `true`).
  - o `Engine.OPTION_INJECTIVE_MATCHING`: Injective rule matching (default `true`). If `false`, it assigns two or more LHS nodes to the same model element.
  - o `Engine.OPTION_CHECK_DANGLING`: checks for dangling edges (default `true`). If `false`, the interpreter will delete the dangling edges.
  - o `Engine.OPTION_SORT_VARIABLES`: enabling/disabling automatic variable sorting.
  - o `Engine.OPTION_INVERSE_MATCHING_ORDER`: enabling/disabling inverse matching order.
  - o `Engine.OPTION_WORKER_THREADS`: setting the number of worker threads to be used. Use along with:
    `EGraph g = new PartitionedEGraphImpl(model, threads);`
  - o `Engine.OPTION_DESTROY_MATCHES`: allows the engine to destroy matching in `createChange(…)` method.

HENSHIN'S VARIANT MANAGEMENT

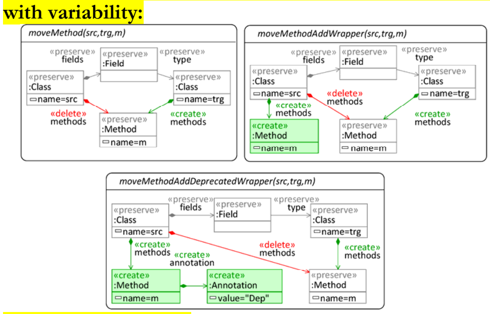
The **Henshin's Variant Management** feature allows to express variants of the same rule in a compact way, by making the commonalities and differences (*variabilities*) between the variants explicit.

Presence conditions specify conditions under which an element is present. The key idea is to annotate rule elements (e.g., nodes and edges) with presence conditions over a set of *features* in which the variants differ.

Presence conditions are propositional formulas, based on the connectives `& | ! xor`

Examples: `(A & B) | (C & !D) & xor(A,B,C)`

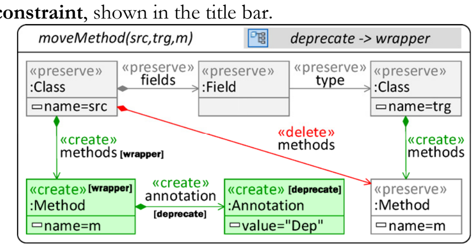
Example of three rule variants expressed using one rule with variability:



Rule with variability:

Several elements are annotated with presence conditions over the features *wrapper* and *deprecate*. The variants are obtained by configuring the rule (i.e., binding the features to *true* or *false*, and removing elements whose presence condition evaluates to false).

To avoid illegal configurations, the rule has a **configuration constraint**, shown in the title bar.



The variability rule needs three annotations attributes:

- o `featureConstraint`: the configuration constraint.
- o `injectiveMatchingPresenceCondition`: to support variant-specific differences (default `false`). If `true`, overrides *injectiveMatching*.
- o `features`: the list of features.

Execution via API:

Rules with variability are applied by using a class **VarRuleApplicationImpl**, which (optionally) takes as input a configuration, represented by a map of feature names.

```
// Applying rule variant with wrapper and deprecated
Map<String, Boolean> config = Map.of("wrapper", true,
                                     "deprecated", false);
RuleApplication vbRuleApp = new VarRuleApplicationImpl(
                  engine, egraph, rule, config, null);
vbRuleApp.execute(null);
```

It's allowed to provide a partial configuration, which does not include a value for each feature, and to provide `null` as the input configuration. In this case, if multiple variants are applicable, an applicable one will be non-deterministically applied. This behavior is particularly useful for batch transformations, where all rules of a module are applied as long as one is applicable.

MIXING STATIC AND DYNAMIC MODELS

When applying a Henshin rule to a model programatically, several configurations of doing so are possible. However, only a few configurations lead to the desired behavior. Models and rules can be either dynamic (d) or static (s). Before loading the model, an existing static implementation of the metamodel (MM) may be registered or not registered with the *ResourceSet* used to load the model. Cases *i* and *k* can be fixed by calling `rs.getModule("rules.henshin" , true)` to load the module.

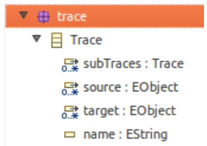
	Model	Rule	MM instance of Model	MM instance of Rule	MM Registered	ResourceSet used for loading model and rule	Result
a	d	d	s	d	yes	separate	not applicable
b	d	s	s	s	yes	separate	applicable
c	s	d	s	d	yes	separate	not applicable
d	s	s	s	s	yes	separate	applicable
e	d	d	d A	d B	no	separate	not applicable
f	d	s	d A	null	no	separate	not applicable
g	s	d	-	-	no	separate	exception
h	s	s	-	-	no	separate	exception
i	d	d	s	d	yes	same	not applicable
j	d	s	s	s	yes	same	applicable
k	s	d	s	d	yes	same	not applicable
l	s	s	s	s	yes	same	applicable
m	d	d	d A	d A	no	same	applicable
n	d	s	d A	d A	no	same	applicable
o	s	d	-	-	no	same	exception
p	s	s	-	-	no	same	exception

THE TRACE MODEL

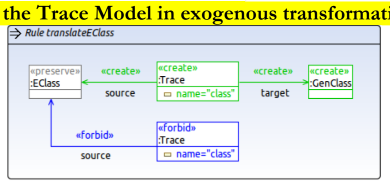
The **Henshin Trace Model** is an EMF model which provides generic and flexible support for traceability. It is used to keep track of the translated elements during the transformation, especially in exogenous transformations.

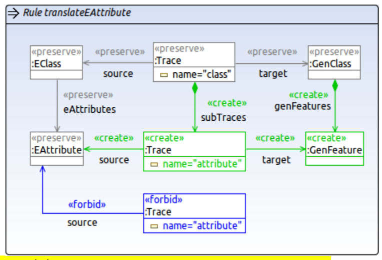
- **Import URI:** http://www.eclipse.org/emf/2011/Henshin/Trace

The Trace Model consists of a single class Trace which two references (source and target) of type EObject. Traces can be named and can contain subtraces.

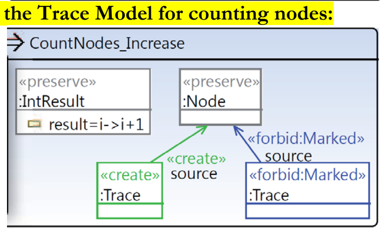


Using the Trace Model in exogenous transformations:





Using the Trace Model for counting nodes:



REFERENCES

- **Main reference:**
  - Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, Gabriele Taentzer. *Henshin: Advanced Concepts and Tools for In-place EMF Model Transformations.* MoDELS, 2010. https://doi.org/10.1007/978-3-642-16145-2_9
- **Variabiliy rules:**
  - Daniel Strüber et al. *Variability-based model transformation: formal foundation and application.* Formal Aspects Comput., 2018. https://doi.org/10.1007/s00165-017-0441-3
- **Henshin applied to search-based MDE:**
  - Stefan John et al. *Searching for Optimal Models: Comparing Two Encoding Approaches.* J. Object Technol., 2019. https://doi.org/10.5381/jot.2019.18.3.a6
  - *MDEOptimiser.* https://mde-optimiser.github.io/
- **Examples:**
  - Daniel Strüber et al. *Henshin: A Usability-Focused Framework for EMF Model Transformation Development.* ICGT, 2017. http://dx.doi.org/10.1007/978-3-319-61470-0_12
  - S. Jurack & J. Tietje. *Saying Hello World with Henshin.* TTC, 2011. https://arxiv.org/abs/1111.4756v1

Explanations:

On loading a model, a check is performed to see if the appropriate metamodel URI is already registered with the *ResourceSet* used to load it. If that is the case the registered metamodel instantiation is used to load the model. This is true for static as well as dynamic models. If the URI is not registered, for a dynamic model the referenced Ecore file is used to instantiate the metamodel dynamically and register it with the *ResourceSet*. Obviously, this fallback method is not available for static models.

On loading a rule, information about the metamodel package needs to be assigned to all Nodes, Edges and Attributes used in the rule. To that end, a lookup for the appropriate metamodel URI is perfomed in the registry of the containing ResourceSet. However, if the rule is a dynamic rule and the instantiation of the registered metamodel is static, a new dynamic instantiation of the metamodel is created and associated with the rule elements. When different *ResourceSets* are used for model and rule, a dynamic instantiation of the metamodel created during model loading cannot be accessed by the rule loading process. In this case, a second dynamic instantiation of the metamodel is created and used. As a result, the model elements of model and rule do not fit together.

Send comments and corrections to J.M. Horcas. CAOSD group, University of Málaga, Spain. horcas@lcc.uma.es
https://sites.google.com/view/josemiguelhorcas