Generator

Generator User Guide

Introduction

Generator is a part of language specification that defines the denotational semantics for a language's concepts.

MPS follows the model-to-model transformation approach. Generator specifies the translation of constructions in the input language to constructions in the output language. The process of model-to-model transformation may involve many intermediate models and results in the output model where all constructions are in language whose semantics are already defined elsewhere.

For instance, most concepts in baseLanguage (classes, methods etc) are "machine understandable", wherefore baseLanguage is often used as an output language.

Target assets are created by applying model-to-text transformation, which must be supported by the output language. The language aspect for that is called TextGen and is available as a separate tab in concept's editor. MPS provides destructive update of generated assets only.

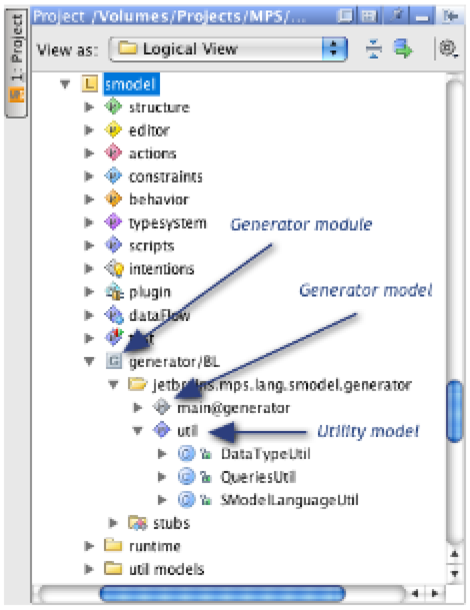
For instance, baseLanguage's TextGen aspect generates *.java files at the following location:
<generator output path>\<model name>\<ClassName>.java
where:
Generator output path - is specified in the module which owns the input model (see MPS modules).
Model name - is a path segment created by replacing '.' with the file separator in the input model's name.

Overview

Generator Module

Unlike any other language aspect, the generator aspect is not a single model. Generator specification can comprise many generator models as well as utility models. Generator model contains templates, mapping configurations and other constructions of the generator language.

Generator model is distinguished from a regular model by the model stereotype - 'generator' (shown after the model name as <name>@generator).
The screenshot below shows the generator module of the smodel language as an example.



Research bundled languages yourself
You can research the smodel (and any other) language's generator by yourself:

- download MPS (here);
- create new project (can be empty project);
- use the Go To -> Go to Language command in the main menu to navigate to the smodel language (its full name is jetbrains.mps.lang.smodel)

Creating a New Generator

New generator is created by using the New -> Generator command in the language's popup menu.

Technically, it is possible to create more than one generator for one language, but at the time of writing MPS does not provide full support for this feature. Therefore languages normally have only one (if any) generator. For that reason, the generator's name is not important. Everywhere in the MPS GUI a generator module can be identified by its language name.

When creating a new generator module, MPS will also create the generator model 'main@generator' containing an empty mapp ing configuration node.

Generator Properties

As a module, generator can depend on other modules, have used languages and used devkits (see Module meta-information).

The generator properties dialog also has two additional properties:

- depends on generators - specifies the dependencies on other generators; this allows making references on templates in another generator;
- mapping constraints - priority relationships between mapping rules can be specified. If such a relationship involves other generator rules, then declaring a dependency on that generator is also required. For details on mapping constraints, see Mapping Priorities, Generation Process: Defining the Order of Priorities, Demo 6: Dividing Generation Process into Steps.

Generating Generator

MPS generator engine (or the Generator language runtime) uses mixed compilation/interpretation mode for transformation execution.

Templates are interpreted and filled at runtime, but all functions in rules, macros, and scripts must be pre-compiled.

To avoid any confusion, always follow this rule: after any changes made to the generator model, the model must be re-generated (Shift+F9). Even better is to use Ctrl+F9, which will re-generate all modified models in the generator module.

Transformation

The transformation is described by means of templates. Templates are written using output language and are edited using the same cell editor as any other 'regular code' in that language. Therefore, the 'template editor' right away has the same level of tooling support - syntax/error highlighting, auto-completion, etc.

Templates applicability is defined using #Generator Rules, which are grouped into #Mapping Configurations.

Mapping Configurations

Mapping Configuration is a minimal unit, which can form a generation step. It contains #Generator Rules, defines mapping labels and may include pre- and post-processing scripts.

Generator Rules

Applicability of each transformation is defined by generator rules.
There are six types of generator rules:

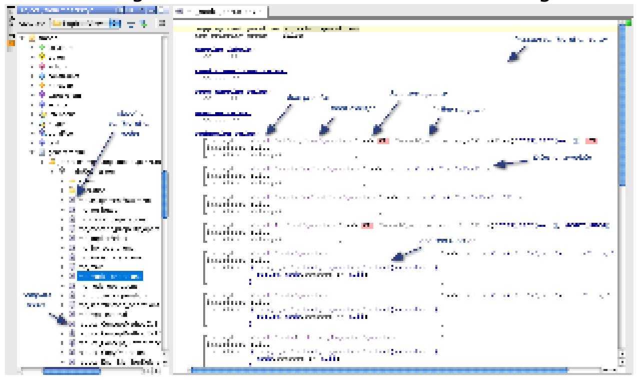
- conditional root rule
- root mapping rule
- weaving rule
- reduction rule
- pattern rule
- abandon root rule

Each generator rule consists of premise and consequence (except for the abandon root rule, whose consequence is predefined and is not specified by the user).

All rules except for the conditional root rule contain a reference on concept of input node (or just input concept) in its premises. All rule premises also contain an optional condition function.

Rule consequence commonly contains a reference to an external template (i.e. a template declared as a root node in the same or different model) or so-called in-line template (conditional root rule and root mapping rule can only have reference to an external template). There are also several other versions of consequences.

The following screenshot shows the contents of a generator model and a mapping configuration example.



Macros

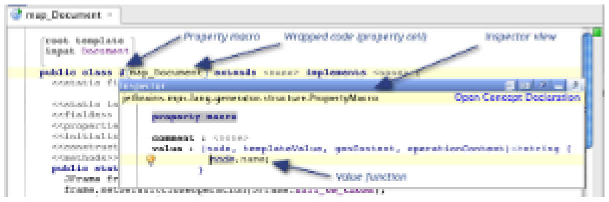
Template code can be parameterized by means of macros. The generator language defines three kinds of macros:

- property macro - computes property value;
- reference macro - computes the target (node) of a reference;
- node macro - is used to control template filling at generation time. There are several versions of node macro - LOOP-macro is an example.

Macro implements a special kind of so-called annotation concept and can wrap property, reference or node cells (depending on the kind of macro) in a template code.

Code wrapping (i.e. the creation of a new macro) is done by pressing Ctrl+Shift+M or by applying the 'Create macro' intention.

The following screenshot shows an example of a property macro.



```
root template
input Document

public class $[map_Document] extends <none> implements <none>
  <<static fields>>

  <<static initializer>>
  <<fields>>
  <<properties>>
  <<initializer>>
  <<constructor>>
  <<methods>>
  public static
    JFrame fr
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
jetbrains.mps.lang.generator.structure.PropertyMacro

property macro

comment : <none>
value : (node, templateValue, genContext, operationContext)->string {
          node.name;
        }
```

Macro functions and other parameterization options are edited in the inspector view. Property macro, for instance, requires specifying the value function, which will provide the value of the property at generation time. In the example above, output class node will get the same name that the input node has.

The node parameter in all functions of the generator language always represents the context node to which the transformation is currently being applied (the input node).

Some macros (such as LOOP and SWITCH-macro) can replace the input node with a new one, so that subsequent template code (i.e. code that is wrapped by those macros) will be applied to the new input node.

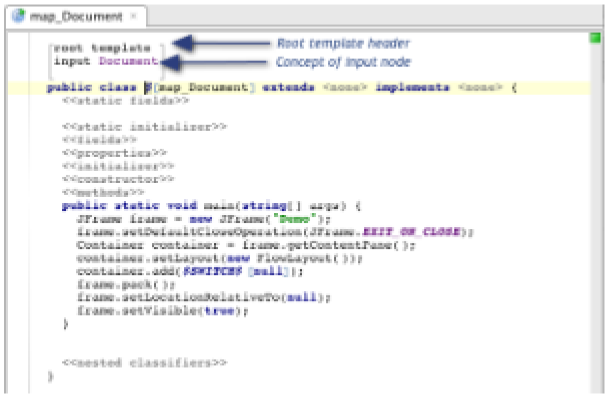
External Templates

External templates are created as a root node in the generator model.

There are two kinds of external templates in MPS.

One of them is root template. Any root node created in generator model is treated as a root template unless this node is a part of the generator language (i.e. mapping configuration is not a root template). Root template is created as a normal root node (via Create Root Node menu in the model's popup).

The following screenshot shows an example of a root template.



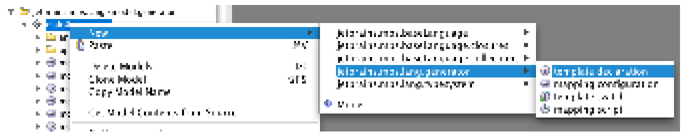
```
root template
input Document
public class $[map_Document] extends <none> implements <none> {
  <<static fields>>

  <<static initializer>>
  <<fields>>
  <<properties>>
  <<initializer>>
  <<constructor>>
  <<methods>>
  public static void main(string[] args) {
    JFrame frame = new JFrame("Demo");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Container container = frame.getContentPane();
    container.setLayout(new FlowLayout());
    container.add($SWITCH$ [null]);
    frame.pack();
    frame.setLocationRelativeTo(null);
    frame.setVisible(true);
  }


  <<nested classifiers>>
}
```

This root template will transform input node (a Document) into a class (baseLanguage). The root template header is added automatically upon creation, but the concept of input node is specified by the user.

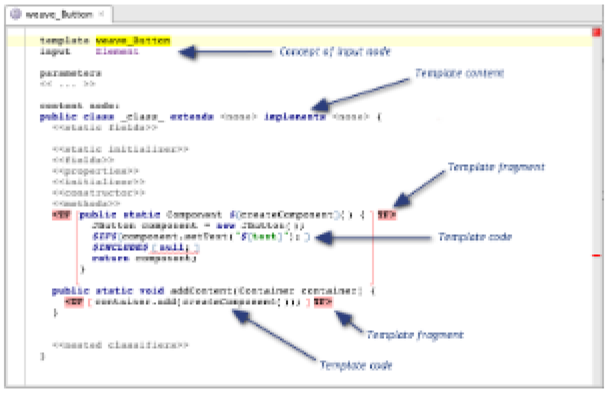
It is good practice to specify the input concept, because this allows MPS to perform a static type checking in a macro function's code.

Root template (reference) can be used as a consequence in conditional root rule and root mapping rule. (When used in a conditional root rule, the input node is not available).

The second kind of template is defined in the generator language and its concept name is 'TemplateDeclaration'. It is created via the 'template declaration' action in the Create Root Node menu.



The following screenshot shows an example of template declaration.



```
template weave_Button
input    Element

parameters
<< ... >>

content node:
public class _class_ extends <none> implements <none> {
  <<static fields>>

  <<static initializer>>
  <<fields>>
  <<properties>>
  <<initializer>>
  <<constructor>>
  <<methods>>
  <TF [public static Component $(createComponent)( ) {  ] TF>
         JButton component = new JButton( );
         $IF$[component.setText("$[text]"); ]
         $INCLUDE$ [ null; ]
         return component;
       }

  public static void addContent(Container container) {
    <TF [ container.add(createComponent( )); ] TF>
  }

  <<nested classifiers>>
}
```

The actual template code is 'wrapped' in a template fragment. Any code outside template fragment is not used in transformation and serves as a context (for example you can have a Java class, but export only one of its method as a template).

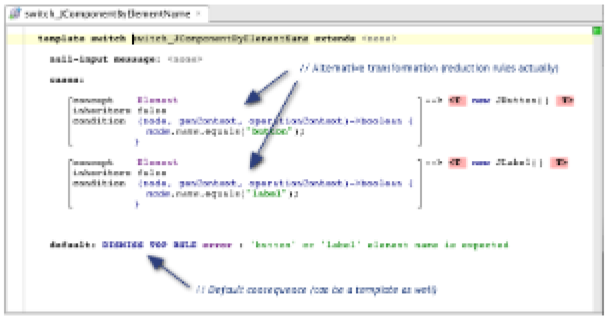
Template declaration can have parameters, declared in the header. Parameters are accessible through the #generation context
.

Template declaration is used in consequence of weaving, reduction and pattern rules. It is also used as an included template in INCLUDE-macro (only for templates without parameters) or as a callee in CALL-macro.

Template Switches

A template switch is used when two or more alternative transformations are possible in a certain place in template code. In that case, the template code that allows alternatives is wrapped in a SWITCH-macro, which has reference to a Template Switch. Template Switch is created as a root node in generator model via the Create Root Node menu (this command can be seen on the 'menu' screenshot above).

The following screenshot shows an example of a template switch.


switch_JComponentByElementName
// Alternative transformation (reduction rules actually)
// Default consequence (can be a template as well)

```
template switch switch_JComponentByElementName extends <none>

  null-input message: <none>

  cases:

    [concept     Element                                              ] --> <T new JButton() T>
    [inheritors  false
    [condition   (node, genContext, operationContext)->boolean {
    [              node.name.equals("button");
    [            }                                                    ]

    [concept     Element                                              ] --> <T new JLabel() T>
    [inheritors  false
    [condition   (node, genContext, operationContext)->boolean {
    [              node.name.equals("label");
    [            }                                                    ]


  default: DISMISS TOP RULE error : 'button' or 'label' element name is expected
```

Generator Language Reference

- Mapping Configuration
- Generator Rule
- Root Template
- External Template
- Mapping Label
- Macro
- Template Switch
- Generation Context (operations)
- Mapping Script

Mapping Configuration

Mapping Configuration is a container for generator rules, mapping label declarations and references on pre- and post-processing scripts. A generator model can contain any number of mapping configurations - all of them will be involved in generation process if the owner generator module is involved. Mapping configuration is a minimal generator unit that can be referenced in mapping priority rules (see Generation Process: Defining the Order of Priorities).

Generator Rule

Generator Rule specifies a transformation of an input node to an output node (except for the conditional root rule which doesn't have an input node). All rules consist of two parts - premise and consequence (except for the abandon root rule which doesn't have a consequence). Any generator rule can be tagged by a mapping label.

All generator rules functions have following parameters:

- node - current input node (all except condition-function in conditional root rule)
- genContext - generation context - allows searching of output nodes, generating of unique names and others (see #generation context)

Generator Rules:

Rule	Description	Premise	Consequence
conditional root rule	Generates root node in output model. Applied only one time (max) during the generation step.	condition function (optional), missing condition function is equivalent to a function always returning true.	root template (ref)
root mapping rule	Generates a root node in output model.	concept - applicable concept (concept of input node) inheritors - if true then the rule is applicable to the specified concept and all its sub-concepts. If false (default) then the sub-concepts are not applicable. condition function (optional) - see conditional root rule above. keep input root - if false then the input root node (if it's a root node) will be dropped. If true then input root will be copied to output model.	root template (ref)
weaving rule	Allows to insert additional child nodes into output model. Weaving rules are processed at the end of generation micro-step just before map_src and reference resolving. Rule is applied on each input node of specified concept. Parent node for insertion should be provided by context function. (see #Model Transformation)	concept - same as above inheritors - same as above condition function (optional) - same as above	• external template (ref) • weave-each context function - computes (parent) output node into which the output node(s) generated by this rule will be inserted.
reduction rule	Transforms input node while this node is being copied to output model.	concept - same as above inheritors - same as above condition function (optional) - same as above	• external template (ref) • in-line template • in-line switch • dismiss top rule • abandon input
pattern rule	Transforms input node, which matches pattern.	pattern - pattern expression condition function (optional) - same as above	• external template (ref) • in-line template • dismiss top rule • abandon input
abandon root rule	Allows to drop an input root node with otherwise would be copied into output model.	applicable concept (⚠ including all its sub-concepts) condition function (optional) - same as above	n/a

Rule Consequences:

Consequence	Usage	Description
root template (ref)	• conditional root rule • (root) mapping rule	Applies root template
external template (ref)	• weaving rule • reduction rule • pattern rule	Applies an external template. Parameters should be passed if required, can be one of: • pattern captured variable (starting with # sign) • integer or string literal • null, true, false • query function
weave-each	weaving rule	Applies an external template to a set of input nodes. Weave-each consequence consists of: • foreach function - returns a sequence of input nodes • reference on an external template
in-line template	• reduction rule • pattern rule	Applies the template code which is written right here.
in-line switch	reduction rule	Consists of set of conditional cases and a default case. Each case specify a consequence, which can be one of: • external template (ref) • in-line template • dismiss top rule • abandon input
dismiss top rule	• reduction rule • pattern rule	Drops all reduction-transformations up to the point where this sequence of transformations has been initiated by an attempt to copy input node to output model. The input node will be copied 'as is' (unless some other reduction rules are applicable). User can also specify an error, warning or information message.
abandon input	• reduction rule • pattern rule	Prevents input node from being copied into output model.

Root Template

Root Template is used in conditional root rules and (root) mapping rules. Generator language doesn't define specific concept for root template. Any root node in output language is treated as a Root Template when created in generator model. The generator language only defines a special kind of annotation - root template header, which is automatically added to each new root template. The root template header is used for specifying of an expected input concept (i.e. concept of input node). MPS use this setting to perform a static type checking in a code in various macro-functions which are used in the root template.

External Template

External Template is a concept defined in the generator language. It is used in weaving rules and reduction rules.

In external template user specifies the template name, input concept, parameters and a content node.

The content node can be any node in output language. The actual template code in external templates is surrounded by templa te fragment 'tags' (the template fragment is also a special kind of annotation concept). The code outside template fragment serves as a framework (or context) for the real template code (template fragment) and is ignored by the generator. In external template for weaving rule, the template's context node is required (it is a design-time representation of the rule's context node), while template for reduction rule can be just one context-free template fragment. External template for a reduction rule must contain exactly one template fragment, while a weaving rule's template can contain more than one template fragments.

Template fragment has following properties (edited in inspector view):

- mapping label
- fragment context - optional function returning new context node which will replace main context node while applying the code in fragment. ⚠ Can only be used in weaving rules.

Mapping Label

Mapping Label is declared in mapping configuration and references on this declaration are used to label generator rules, macros and template fragments. Such marks allow finding of an output node by input node (see #generation context).

Properties:

- name
- input concept (optional) - expected concept of input node of transformation performed by the tagged rule, macro or template fragment
- output concept (optional) - expected concept of output node of transformation performed by the tagged rule, macro or template fragment

MPS makes use of the input/output concept settings to perform static type checking in get output ... operations (see #generati on context).

Macro

Macro is a special kind of an annotation concept which can be attached to any node in template code. Macro brings dynamic aspect into otherwise static template-based model transformation.

Property- and reference-macro is attached to property- and reference-cell, and node-macro (which comes in many variations - LOOP, IF etc.) is attached to a cell representing the whole node in cell-editor. All properties of macro are edited using inspector view.

All macro have the mapping label property - reference on a mapping label declaration. All macro can be parameterized by various macro-functions - depending on type of the macro. Any macro-function has at least three following parameters:

- node - current input node;
- genContext - generation context - allows searching of output nodes, generating of unique names and others;
- operationContext - instance of jetbrains.mps.smodel.IOperationContext interface (used rarely).

Many of macro have mapped node or mapped nodes function. This function computes new input node - substitution for current input node. If mapped node function returns null or mapped nodes function returns an empty sequence, then generator will skip this macro altogether. I.e. no output will be generated in this place.

Macro	Description	Properties (if not mentioned above)
Property macro	Computes value of property.	value function: • return type - string, boolean or int - depending on the property type. • parameters - standard + templateValue - value in template code wrapped by the macro.
Reference macro	Computes referent node in output model. Normally executed in the end of a generation micro-step, when the output model (tree) is already constructed. Can also be executed earlier if a user code is trying to obtain the target of the reference.	referent function: • return type - node (type depends on the reference link declaration) or, in many cases, string identifying the target node (see note). • parameters - standard + outputNode - source of the reference link (in output model).
IF	Wrapped template code is applied only if condition is true. Otherwise the template code is ignored and an 'alternative consequence' (if any) is applied.	condition function alternative consequence (optional) - any of: • external template (ref) • in-line template • abandon input • dismiss top rule
LOOP	Computes new input nodes and applies the wrapped template to each of them.	mapped nodes function
INCLUDE	Wrapped template code is ignored (it only serves as an anchor for the INCLUDE-macro), a reusable external template will be used instead.	mapped node function (optional) include template - reference on reusable external template

CALL	Invokes template, replaces wrapped template code with the result of template invocation. Supports templates with parameters.	mapped node function (optional) call template - reference on reusable external template argument - one of • pattern captured variable • integer or string literal • null, true, false • query function
SWITCH	Provides a way to many alternative transformations in the given place in template code. The wrapped template code is applied if none of switch cases is applicable and no default consequence is specified in #template switch.	mapped node function (optional) template switch - reference on template switch
COPY-SRC	Copies input node to output model. The wrapped template code is ignored.	mapped node function - computes the input node to be copied.
COPY-SRCL	Copies input nodes to output model. The wrapped template code is ignored. Can be used only for children with multiple aggregation cardinality.	mapped nodes function - computes the input nodes to be copied.
MAP-SRC	Multifunctional macro, can be used for: • marking a template code with mapping label; • replacing current input node with new one; • perform none-template based transformation; • accessing output node for some reason. MAP-SRC macro is executed in the end of generator micro-step - after all node- and property-macro but before reference-macro.	mapped node function (optional) mapping func function (optional) - performes none-template based transformation. If defined then the wrapped template code will be ignored. Parameters: standard + parentOutputNode - parent node in output model. post-processing function (optional) - give an access to output node. Parameters: standard + outputNode
MAP-SRCL	Same as MAP-SRC but can handle many new input nodes (similar to LOOP-macro)	mapped nodes function mapping func function (optional) post-processing function (optional)

Note
Reference resolving by identifier is only supported in BaseLanguage.
The identifier string for classes and class constructors may require (if class is not in the same output model) package name in square brackets preceding the class name:
[package.name]ClassName

Template Switch

Template switch is used in pair with SWITCH-macro (can be re-used in many different SWITCH-macro). Template switch consists of set of cases and one default case. Each switch case is a reduction rule, i.e. template switch contains list of reduction rules actually (see #reduction rule).

The default case consequence can be one of:

- external template (ref)
- in-line template
- abandon input
- dismiss top rule
  .. or can be omitted. In this case the template code surrounded by corresponding SWITCH-macro will be applied.

Template switch can inherit reduction rules from other switches via the extends property. When generator is executing a SWITCH-macro it tries to find most specific template switch (available in scope). Therefore the actually executed template switch is not necessarily the same as it is defined in template switch property in SWITCH-macro.

In the null-input message property user can specify an error, warning or info message, which will be shown in MPS messages view in case when the mapped node function in SWITCH-macro returns null (by default no messages are shown and macro is skipped altogether).

Generation Context (operations)

Generation context (genContext parameter in macro- and rule-functions) allows finding of nodes in output model, generating unique names and provides other useful functionality.

Generation context can be used not only in generator models but also in utility models - as a variable of type gencontext.

Operations of genContext are invoked using familiar dot-notation: genContext.operation

Finding Output Node

get output <mapping label>	Returns output node generated by labeled conditional root rule. Issues an error if there are more than one matching output nodes.
get output <mapping label> for (<input node>)	Returns output node generated from the input node by labeled generator rule, macro or template fragment. Issues an error if there are more than one matching output nodes.
pick output <mapping label> for (<input node>)	⚠ only used in context of the referent function in reference-macro and only if the required output node is target of reference which is being resolved by that reference-macro. Returns output node generated from the input node by labeled generator rule, macro or template fragment. Difference with previous operation is that this operation can automatically resolve the many-output-nodes conflict - it picks the output node which is visible in the given context (see search scope).
get output list <mapping label> for (<input node>)	Returns list of output nodes generated from the input node by labeled generator rule, macro or template fragment.
get copied output for (<input node>)	Returns output node which has been created by copying of the input node. If during the copying, the input node has been reduced but concept of output node is the same (i.e. it wasn't reduced into something totally different), then this is still considered 'copying'. Issues an error if there are more than one matching output nodes.

Generating of Unique Name

unique name from <base name> in context <node>

The uniqueness is secured throughout the whole generation session.
⚠ Clashing with names that wasn't generated using this service is still possible.

The context node is optional, though we recommend to specify it to guarantee generation stability. If specified, then MPS tries its best to generated names 'contained' in a scope (usually a root node). Then when names are re-calculated (due to changes in input model or in generator model), this won't affect other names outside the scope.

Template Parameters

#patternvar	Value of captured pattern variable ⚠ available only in rule consequence
param	Value of template parameter ⚠ available only in external template

Getting Contextual Info

inputModel	Current input model
originalModel	Original input model
outputModel	Current output model
invocation context	Operation context (jetbrains.mps.smodel.IOperationContext java interface) associated with module - owner of the original input model
scope	Scope - jetbrains.mps.smodel.IScope java interface

templateNode	Template code surrounded by macro. Only used in macro-functions
get prev input <mapping label>	Returns input node that has been used for enclosing template code surrounded by the labeled macro. Only used in macro-functions.

Transferring User Data

During a generation MPS maintains three maps user objects, each have different life span:

- session object - kept throughout whole generation session;
- step object - kept through a generation step;
- transient object - only live during a micro step.

Developer can access user object maps using an array (square brackets) notation:

```
session object [ <key> ]
step object [ <key> ]
transient object [ <key> ]
```

The key can be any object (java.lang.Object).

binding user data with particular node
The session- and step-object cannot be used to pass a data associated with a particular input node across steps and micro-steps because neither an input node nor its id can serve as a key (output nodes always have different id).
To pass such data use methods: putUserObject, getUserObject and removeUserObject defined in class jetbrains.mps.smodel.SNode.
The data will be transferred to all output copies of the input node. The data will be also transferred to output node if a slight reduction (i.e. without changing of the node concept) took place while the node copying.

Logging

```
show info <message text> -> <node>
show error <message text> -> <node>
show warning <message text> -> <node>
```

Creates message in MPS message view. If the node parameter is specified then clicking on the message will navigate to that node. In case of an error message, MPS will also output some additional diagnostic info.

Utilities (Re-usable Code)

If you have duplicated code (in rules, macros, etc.) and want to say, extract it to re-usable static methods, then you must create this class in a separate, non-generator model.

If you create an utility class in the generator model (i.e. in a model with the 'generator' stereotype), then it will be treated as a root template (unused) and no code will be generated from it.

Mapping Script

Mapping script is a user code which is executed either before a model transformation (pre-processing script) or after it (post-processing script). It should be referenced from #Mapping Configuration to be invoked as a part of it's generation step. Mapping script provides the ability to perform a non-template based model transformation.

Pre-processing scripts are also commonly used for collecting certain information from input model that can be later used in the course of template-based transformation. The information collected by script is saved as a transient-, step- or session-object (see generation context).

Script sample:

```
mapping script myPreProcessingScript

script kind : pre-process input model
modifies model : false

(model, genContext, operationContext)->void {
  for (node<MyConcept> te : model.nodes(MyConcept)) {
    genContext.transient object [ MyConceptUtil.KEY_ID ] = ....;
  }
}
```

Properties:

script kind	• pre-process input model - script is executed in the beginning of generation step, before template-based transformation; • post-process output model - script is executed at the end of generation step, after template-based transformation.
modifies model	only available if script kind = pre-process input model If set true and input model is the original input model, then MPS will create a transient input model before applying the script. If set false but script tries to modify input model, then MPS will issue an error.

Code context:

model	Current model
genContext	Generation context to access transient/session or step objects.
invocation context	Operation context (jetbrains.mps.smodel.IOperationContext java interface) associated with module - owner of the original input model

The Generator Algorithm

The process of generation of target assets from an input model (generation session) includes 5 stages:

- Defining all generators that must be involved
- Defining the order of priorities of transformations
- Step-by-step model transformation
- Generating text and saving it to a file (for each root in output model)
- Post-processing assets: compiling, etc.

We will discuss the first three stages of this process in detail.

Defining the Generators Involved

To define the required generators, MPS examines the input model and determines which languages are used in it. Doing this job MPS doesn't make use of 'Used Languages' specified in the model properties dialog. Instead MPS examines each node in the model and gathers languages that are actually used.

From each 'used language' MPS obtains its generator module. If there are more than one generator module in a language, MPS chooses the first one (multiple generators for the same language are not fully supported in the current version of MPS). If any generator in this list depends on other generators (as specified in the 'depends on generators' property), then those generators are added to the list as well.

After MPS obtains the initial list of generators, it begins to scan the generator's templates in order to determine what languages will be used in intermediate (transient) models. The languages detected this way are handled in the same manner as the languages used in the original input model. This procedure is repeated until no more 'used languages' can be detected.

Explicit Engagement

In some rare cases, MPS is unable to detect the language whose generator must be involved in the model transformation. This may happen if that language is not used in the input model or in the template code of other (detected) languages. In this case, you can explicitly specify the generator engagement via the Languages Engaged on Generation section in the input model's properties dialog (Advanced tab).

Defining the Order of Priorities

As we discussed earlier, a generator module contains generator models, and generator models contain mapping configurations. Mapping configuration (mapping for short) is a set of generator rules. It is often required that some mappings must be applied before (or not later than, or together with) some other mappings. The language developer specifies such a relationship between mappings by means of mapping constraints in the generator properties dialog (see also #Mapping Priorities and the Dividing Generation Process into Steps demo).

After MPS builds the list of involved generators, it divides all mappings into groups, according to the mapping priorities specified. All mappings for which no priority has been specified fall into the last (least-priority) group.

tip
You can check the mapping partitioning for any (input) model by selecting Show Mapping Partitioning action in the model's popup menu.
The result of partitioning will be shown in the MPS Output View.

Model Transformation

Each group of mappings is applied in a separate generation step. The entire generation session consists of as many generation steps as there were mapping groups formed during the mapping partitioning. The generation step includes three phases:

- Executing pre-mapping scripts
- Template-based model transformation
- Executing post-mapping scripts

The template-based model transformation phase consists of one or more micro-steps. The micro-step is a single-pass model transformation of input model into a transient (output) model.

While executing micro-step MPS follows the next procedure:

1. Apply conditional root rules (only once - on the 1-st micro-step)
2. Apply root mapping rules
3. Copy input roots for which no explicit root mapping is specified (this can be overridden by means of the 'keep input root' option in root mapping rules and by the 'abandon root' rules)
4. Apply weaving rules
5. Apply delayed mappings (from MAP_SRC macro)
6. Revalidate references in the output model (all reference-macro are executed here)

There is no separate stage for the application of reduction and pattern rules. Instead, every time MPS copies an input node into the output model, it attempts to find an applicable reduction (or pattern) rule. MPS performs the node copying when it is either copying a root node or executing a COPY_SRC-macro. Therefore, the reduction can occur at either stage of the model transformation.

MPS uses the same rule set (mapping group) for all micro-steps within the generation step. After a micro-step is completed and some transformations have taken place during its execution, MPS starts the next micro-step and passes the output model of the previous micro-step as input to the next micro-step. The whole generation step is considered completed if no transformations have occurred during the execution of the last micro-step, that is, when there are no more rules in the current rule set that are applicable to nodes in the current input model.

The next generation step (if any) will receive the output model of previous generation step as its input.

tip
Intermediate models (transient models) that are the output/input of generation steps and micro-steps are normally destroyed immediately after their transformation to the next model is completed.
To keep transient models, enable the following option:
Settings -> Generator Settings -> Save transient models on generation

See also:

- Saving Transient Models
- Using Generation Tracer Tool

Unable to render {include} The included page could not be found.

Examples

If you're feeling like it's time for more practical experience, check out the generator demos.
The demos contain examples of usage of all concepts discussed above.

Previous Next