Debugger

Debugger

MPS provides an API for creating custom debuggers as well as integrating with debugger for java. See Debugger Usage page fo r a description of MPS debugger features.

- Integration with MPS java debugger engine
  - Nodes to trace and breakpoints
  - Startup of a run configuration under java debugger
  - Custom viewers
- Creating a non-java debugger
  - Traceable Nodes
  - Breakpoint Creators

Integration with MPS java debugger engine

To integrate your java-generated language with MPS java debugger engine, you should specify:

- on which nodes breakpoints could be created (see Breakpoint creators);
- nodes which should be traced (see Traceable nodes);
- how to start your application under debug;
- custom viewers for your data (see Custom viewers).

Not all of those steps are absolutely necessary; which are – depends on the language. See next parts for details.

Nodes to trace and breakpoints

Suppose you have a language, let's call it high.level, which generates code on some language low.level, which in turn is generated directly into text (there can be several other steps between high.level and low.level). Suppose that the text generated from low.level consists of java classes, and you want to have your high.level language integrated with MPS java debugger engine. See the following explanatory table:

	low.level is baseLanguage	low.level is not baseLanguage
high.level extends baseLanguage (uses concepts `Statement`,`Expression`, `BaseMethodDeclaration` etc)	Do not have to do anything.	Specify which concepts in low.level are traceable.
high.level does not extend baseLanguag e	Use `breakpoint creators` to be able to set breakpoints for high.level.	Specify which concepts in low.level are traceable. Use `breakpoint creators` to be able to set breakpoints for high.level.

Startup of a run configuration under java debugger

MPS provides a special language for creating run configurations for languages generated into java – jetbrains.mps.baseLangua ge.runConfigurations. Those run configurations are able to start under debugger automatically. See Run configurations for languages generated into java for details.

Custom viewers

When one views variables and fields in a variable view, one may want to define one's own way to show certain values. For instance, collections could be shown as a collection of elements rather than as an ordinary object with all its internal structure.

For creating custom viewers MPS has jetbrains.mps.debug.customViewers language.

A language jetbrains.mps.debug.customViewers enables one to write one's own viewers for data of certain form.

A main concept of customViewers language is a custom data viewer. It receives a raw java value (an objects on stack) and returns a list of so-called watchables. A watchable is a pair of a value and its label (a string which cathegorizes a value, i.e. whether a value is a method, a field, an element, a size etc.) Labels for watchables are defined in `custom watchables container`. Each label could be assigned an icon.

The viewer for a specific type is defined in a `custom viewer` root. In the following table `custom viewer` parts are described:

Part	Description
for type	A type for which this viewer is intended.
can wrap	An additional filter for viewed objects.
get presentation	A string representation of an object.
get custom watchables	Subvalues of this object. Result of this funtion must be of type `watchable list`.

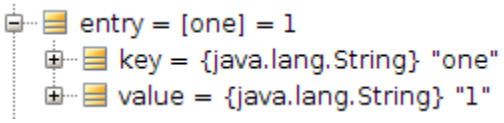
Custom Viewers language introduces two new types: `watchable list` and `watchable`.

This is the custom viewer specification for `java.util.Map.Entry` class:

MapEntryViewer

```
custom viewer MapEntryViewer

for type: Map.Entry

can wrap:
<no canWrap>

get presentation:
(value)->string {
  Object key = value.getKey();
  Object entryValue = value.getValue();
  return "[" + (key == null ? "null" : key.toString()) + "] = " + (entryValue ==
      toString());
}

get custom watchables
(value)->watchable list {
  watchable list result = new watchables array list;
  Object key = value.getKey();
  Object entryValue = value.getValue();
  result.add(new watchable key ( key ));
  result.add(new watchable value ( entryValue ));
  return result;
}
```

And here we see how a map entry is displayed in debugger view:



Creating a non-java debugger

Debugger API provided by MPS allows to create a non-java debugger. All the necessary classes are located in the "Debugger API for MPS" plugin. See also Debugger API description.

Traceable Nodes

This section describes how to spesify which nodes require to save some additional information in `trace.info` file (like information about positions text, generated from the node, visible variables, name of the file the node was generated into etc.).

`trace.info` files contain information allowing to connect nodes in MPS with generated text. For example, if a breakpoint is hit, java debugger tells MPS the line number in source file and to get the actual node from this information MPS uses information from `trace.info` files.

Specifically, `trace.info` files contain the following information:

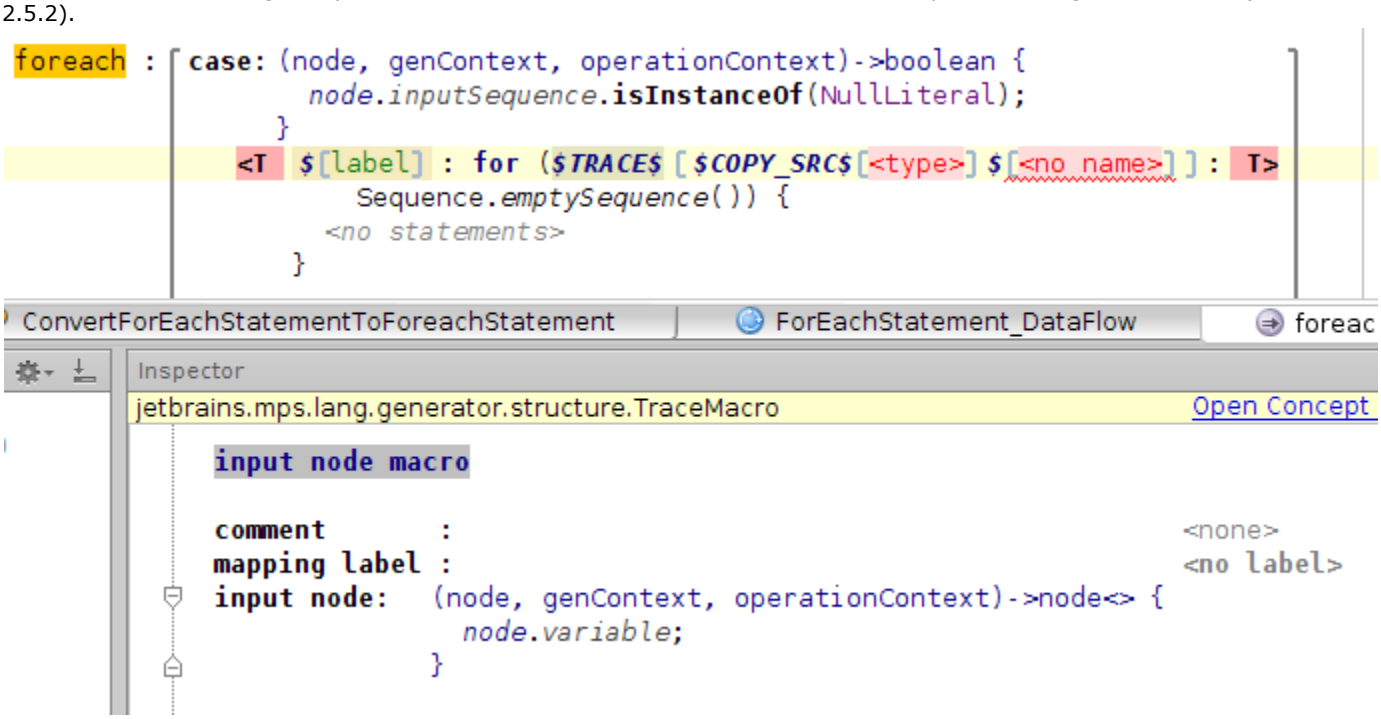
- position information: name of text file and position in it where the node was generated;
- scope information: for each "scope" node (such that has some variables, associated with it and visible in the scope of the node) – names and ids of variables visible in the scope;
- unit information: for each "unit node" (such that represent some unit of a language, for example a class in java) – name of the unit the node is generated into.

Concepts `TraceableConcept`, `ScopeConcept` and `UnitConcept` of language `jetbrains.mps.lang.traceable` are used for that purpose. To save some information into `trace.info` file, user should derive from one of those concepts and implement the specific behavior method. The concepts are described in the table below.

Concept	Description	Behavior method to implement	Example
`TraceableConcept`	Concepts for which location in text is saved and for which breakpoints could be created.	`getTraceableProperty` – some property to be saved into `trace.info` file.	
`ScopeConcept`	Concepts which have some local variables, visible in the scope.	`getScopeVariables` – variable declarations in the scope.	
`UnitConcept`	Concepts which are generated into separate units, like classes or inner classes in java.	`getUnitName` – name of the generated unit.	

`trace.info` files are created on the last stage of generation – while generating text. So the decsribed concepts are only to be used in languages generated into text.

When automatical tracing is impossible, $TRACE$ macro can be used in order to set input node for generated code (since MPS 2.5.2).



Breakpoint Creators

To specify how breakpoints are created on various nodes, root `breakpoint creators` is used. This is a root of concept `BreakpointCreator` from `jetbrains.mps.debug.apiLang` language. The root should be located in the language plugin model. It contains a list of `BreakpointableNodeItem`, each of them specify a list of concept to create breakpoint for and a method actually creating a breakpoint. `jetbrains.mps.debug.apiLanguage` provides several concepts to operate with debuggers, and specifially to create breakpoints. They are described below.

- DebuggerReference – a reference to a specific debugger, like java debugger;
- CreateBreakpointOperation – an operation which creates a location breakpoint of specified kind on a given node for a given project;
- DebuggerType – a special type for references to debuggers.

On the following example `breakpoint creators` node from `baseLanguage` is shown.

```
breakpoint creators
  for concepts:
    Statement
  create breakpoint:
    (debuggableNode, project)->ILocationBreakpoint throws DebuggerNotPresentExceptio
      return debugger<Java>.create(Java Line Breakpoint, debuggableNode, project);
    }
  for concepts:
    FieldDeclaration
    StaticFieldDeclaration
  create breakpoint:
    (debuggableNode, project)->ILocationBreakpoint throws DebuggerNotPresentExceptio
      return debugger<Java>.create(Java Field Breakpoint, debuggableNode, project);
    }
```

In order to provide more complex filtering behavior, instead of a plain complex list breakpoint creators can use `isApplicable` f unction. There is an intention to switch to using this function.

Previous Next