White Paper by Eugen Schindler

Debugging in MPS

Every MPS language developer eventually hits a point where the language they developed behaves unexpectedly and it's not obvious why this is happening. In a "classic" development situation, this would be the point where you would reach for a debugger or the plain old "printf-debugging" method. But how do you debug your MPS language(s)?

1. A Quick Checklist

As a quick checklist, JetBrains has a *Finding Your Way Out* page. If you are still stuck, you need a more systematic way of analyzing your situation, and this article provides you with pointers.

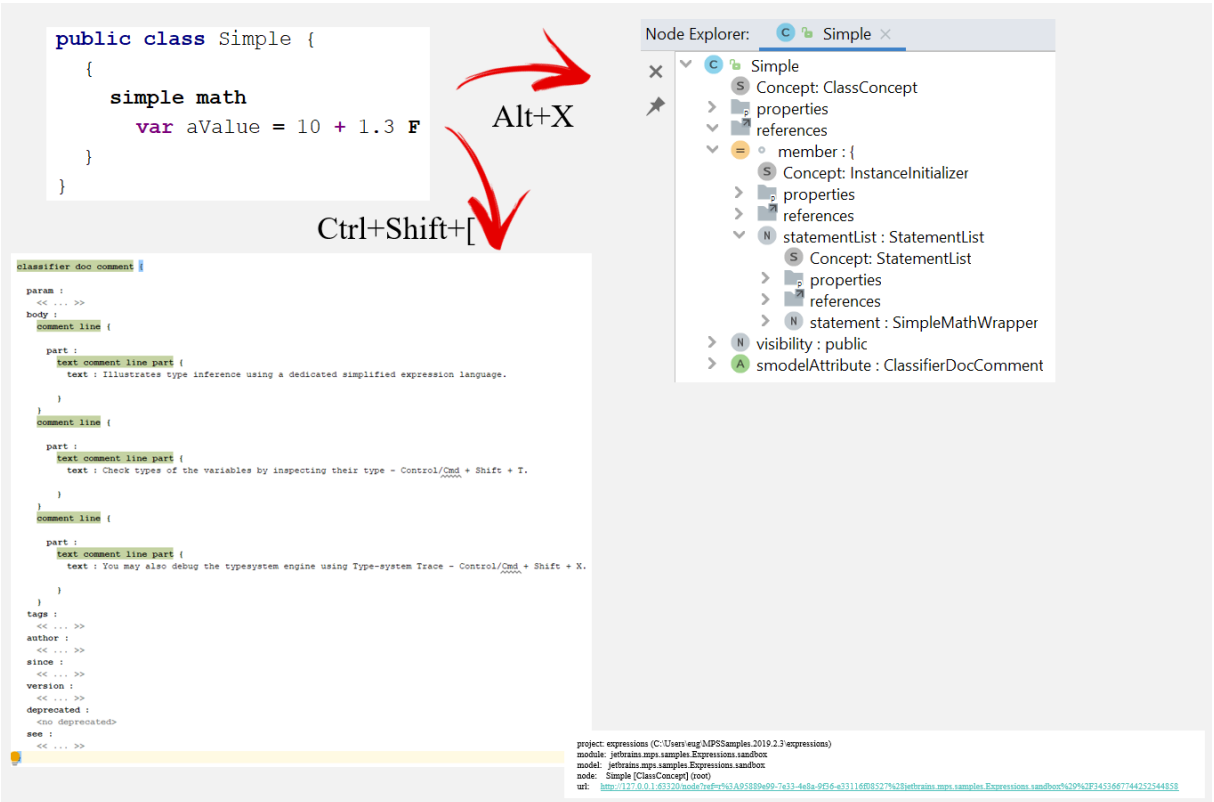
2. Built-in (and third-party) tracing tools

MPS provides built-in tracing and exploration tools for many subsystems to examine their inner workings. This section gives a description of these tools.

1. Structure Analysis

Reflective Editor and Node Explorer are tools for for discovering the true underlying structure of the model, as opposed to what the projectional editor is showing.

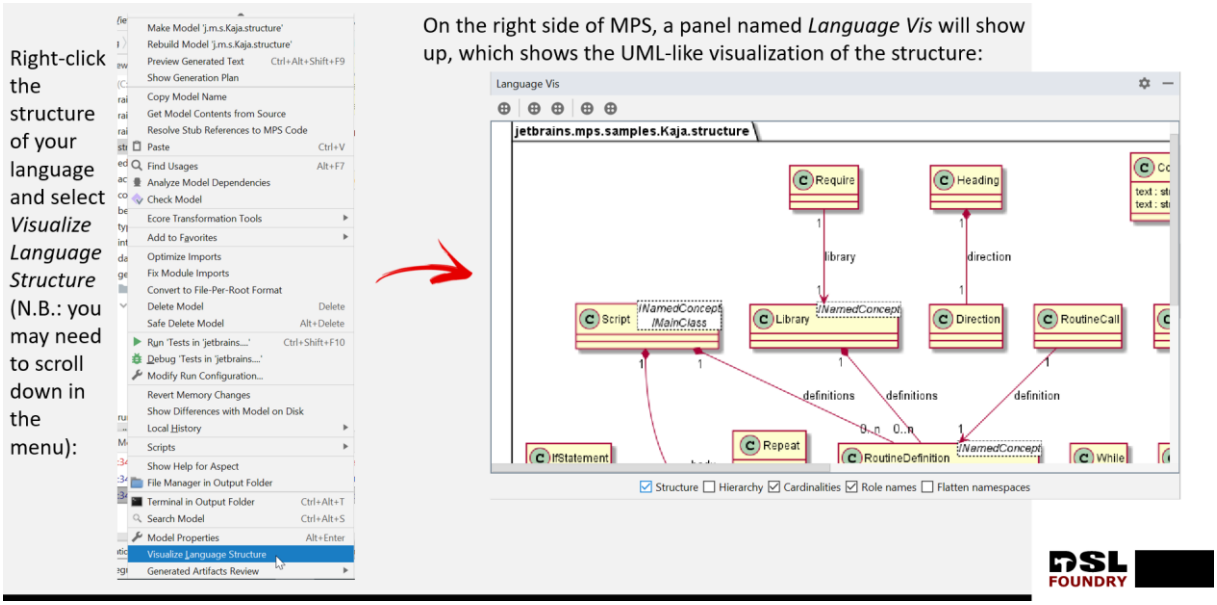
The following picture shows how to get the node explorer and the reflective editor for a piece of model (the shortcuts on Mac are Ctrl+X and Cmd+Shift+[):



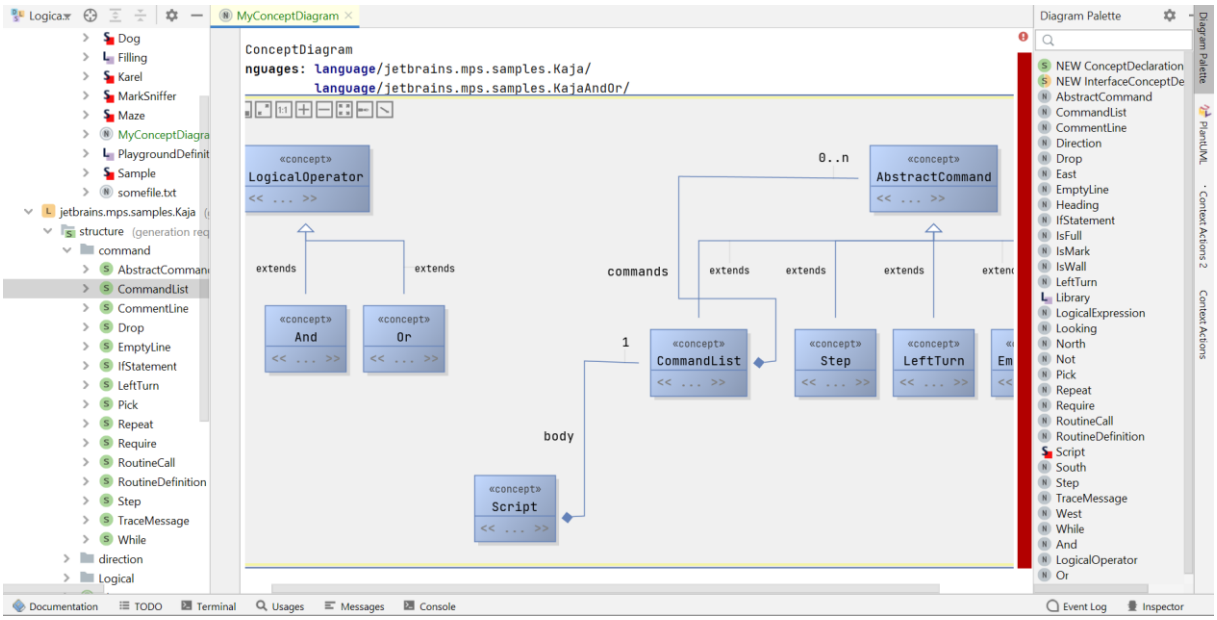
Another useful tool for analyzing structure is the hierarchy tool window.

The langvis plugin, a third-party tool from DSLFoundry, may also be helpful in getting insight into the structure of a language (or to document the structure in order for others to get insight), especially if you are more graphically oriented. This tool gives a graphical overview (very similar to a UML class diagram) of the structure of your language.

When you install the langvis plugin, keep in mind the prerequisite of PlantUML, which means you have to download plantuml.jar and put it in your home directory. The example below shows a structure visualization of the RobotKaja project from MPS Samples:



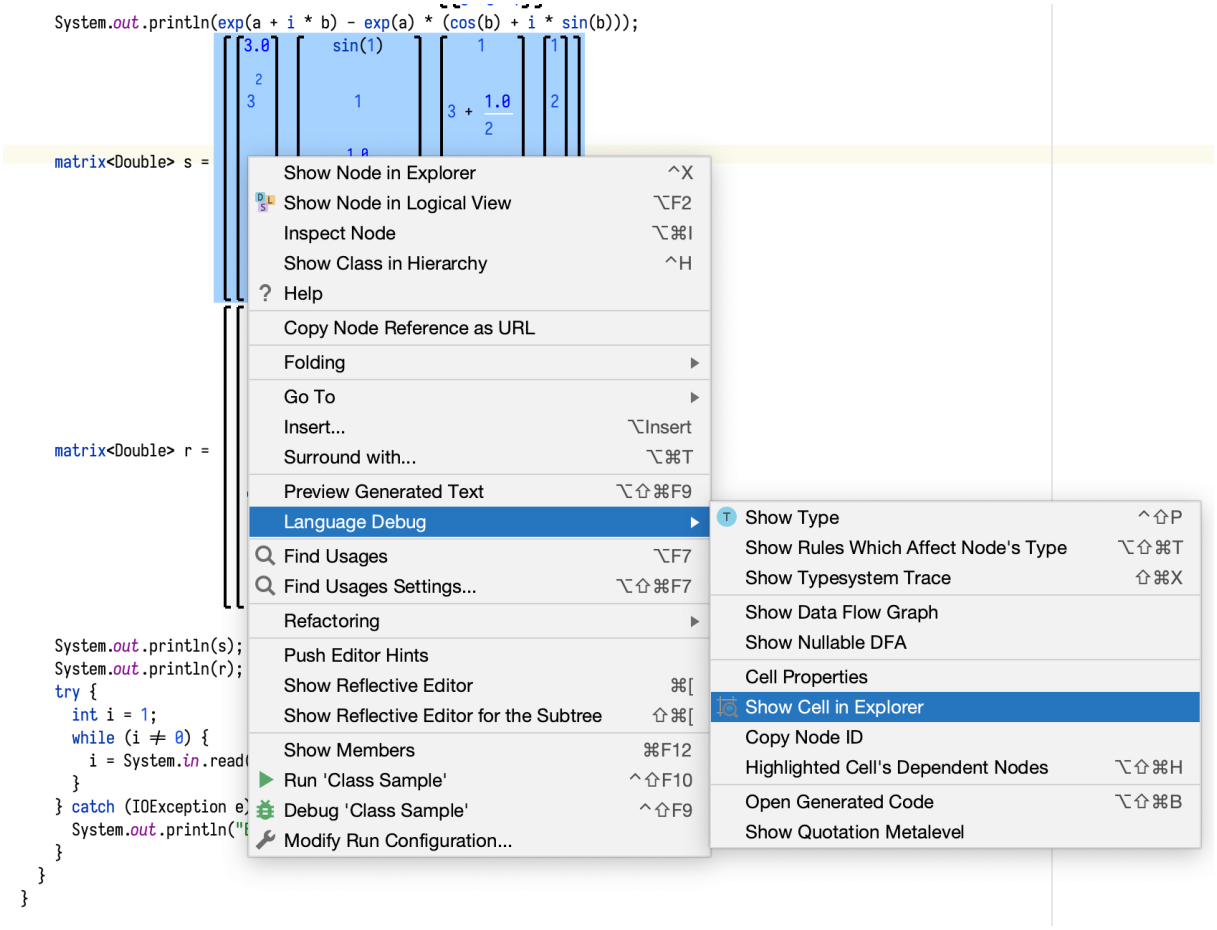
You can also use the *mbeddr conceptdiagram* language (this requires installation of mbeddr into your MPS). Because a concept diagram is an editable model, it enables you to manually filter out (or filter in) specific concepts in the overview by drag-and-drop from the diagram palette (on the right). To use *conceptdiagram*, simply add the language *com.mbeddr.mpsutil.conceptdiagram* to your used languages in a model and then create a new *conceptdiagram* root node. The following picture shows an example of a concept diagram:

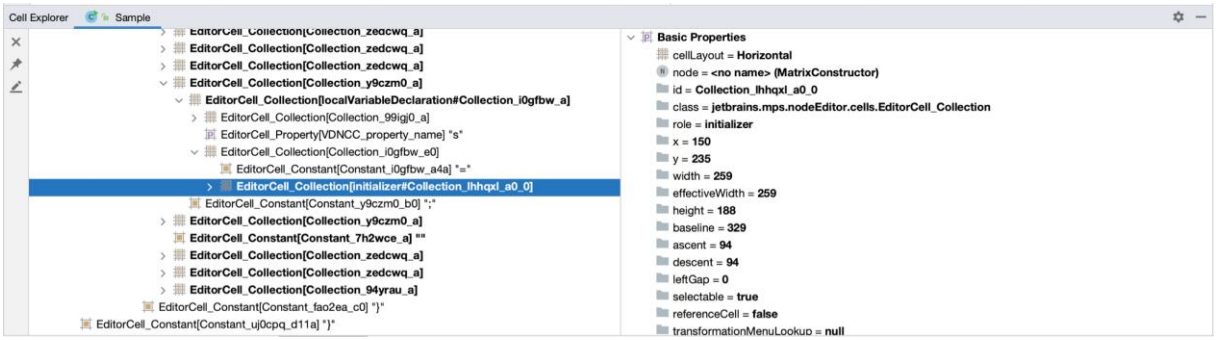


2. Editor Analysis

Menu Trace can be used to learn which transformation or substitute menu contributed a particular action to the completion menu or to the context assistant.

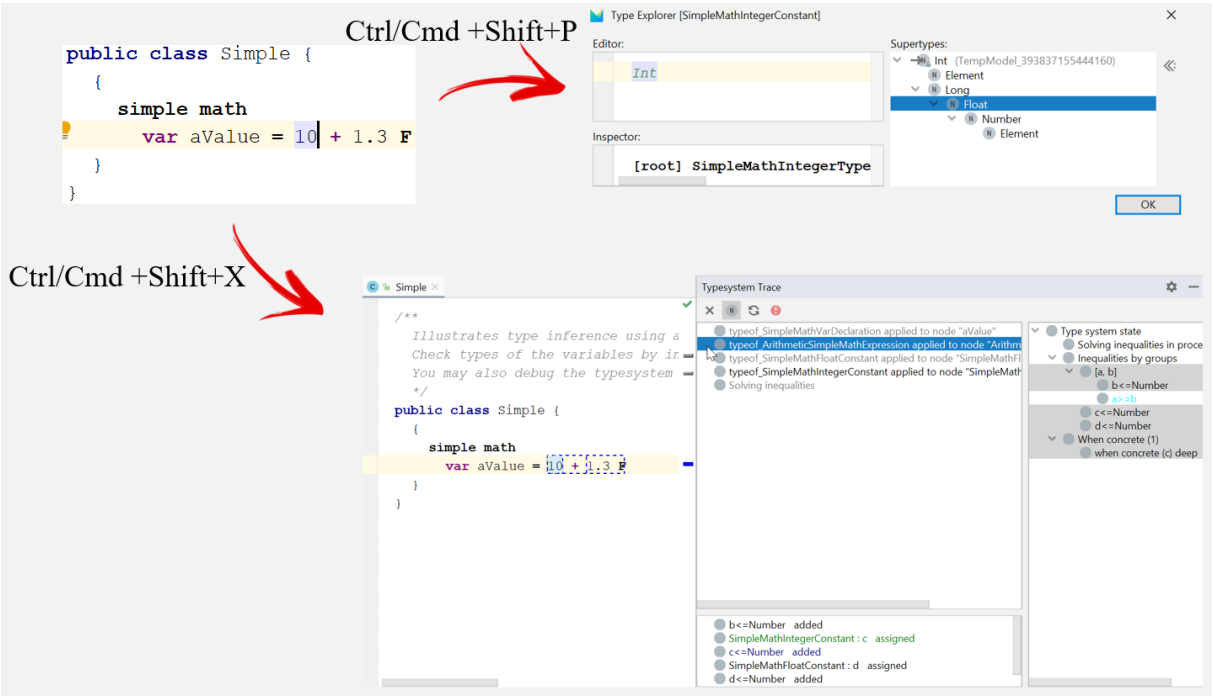
Cell Explorer can be used to explore the hierarchy of editor cells built for a particular node. The following picture shows how to open the cell explorer and examples of how it looks:





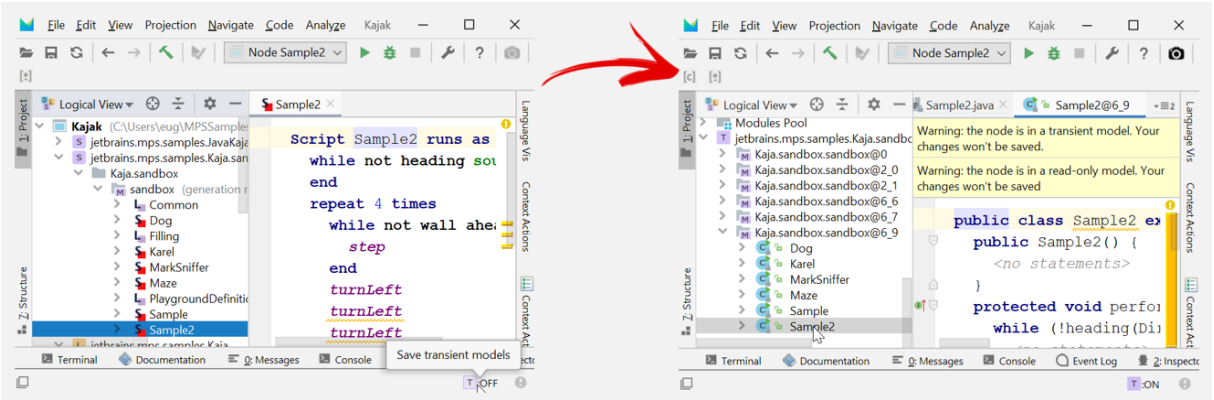
3. Typesystem Analysis

Typesystem Trace and Show Node Type action for getting insight into the typesystem engine. The image below summarizes how to get to each of those from a piece of a model and shows examples of how they look like:

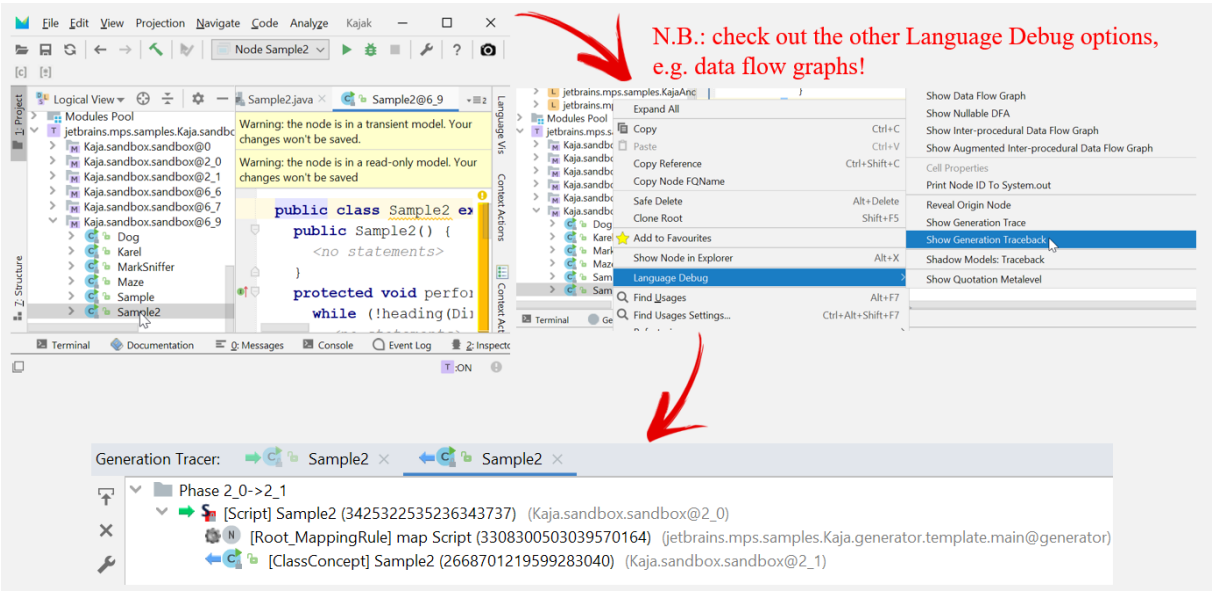


4. Generator Analysis

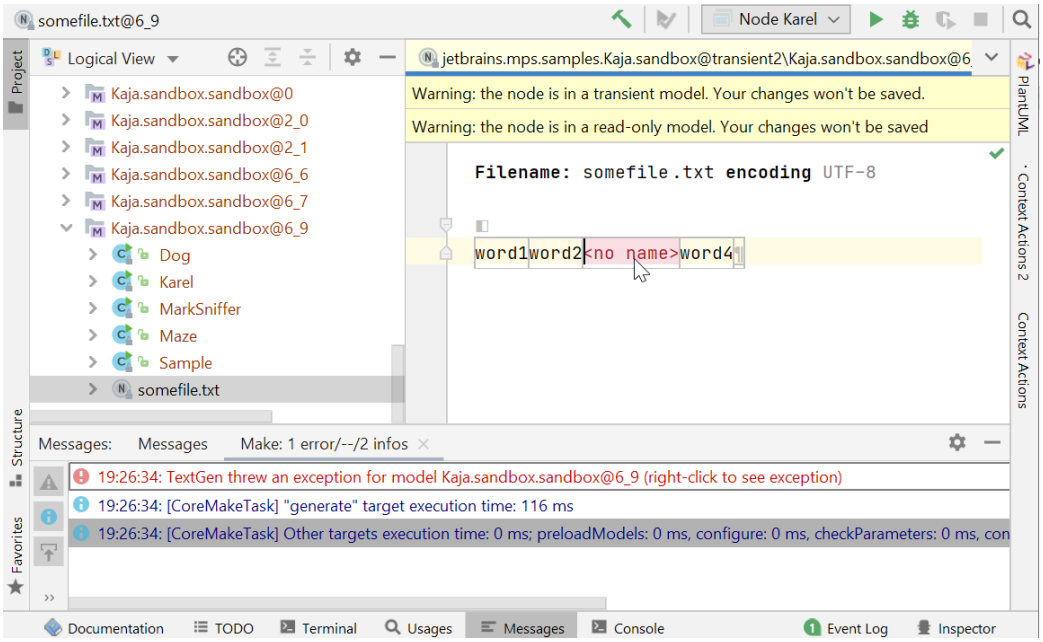
Transient Models and Generation Trace/Traceback are tools for analyzing what is happening in the generators. When saving transient models is enabled, all intermediate steps produced by the generator can be inspected in the transient models, at the bottom of the logical view, just under the modules pool. The following picture shows how to do that:



When transient models are enabled, you can also do a generation traceback through the various generation steps. In the following example (RobotKaja project from MPS Samples that are always delivered with MPS), right-click Sample2 class node in the logical explorer:



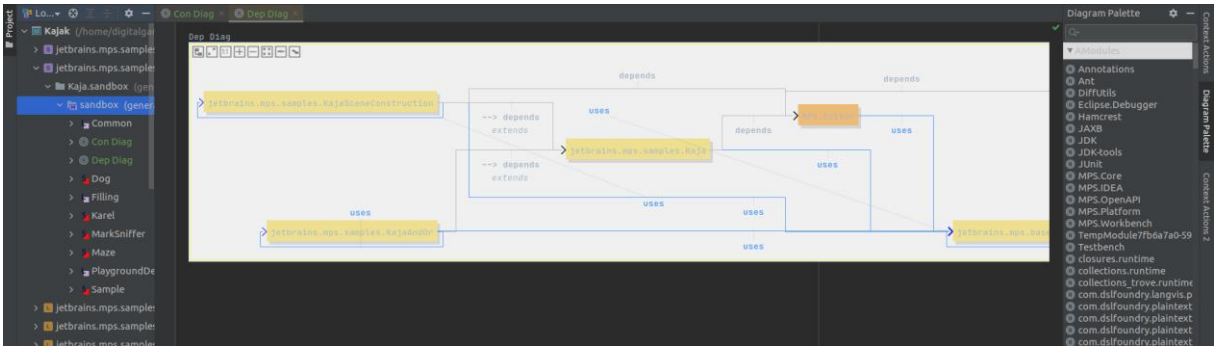
When using plaintextgen (which does no null-checking due to performance reasons), you can very quickly find the source of a NullPointer exception. Simply kick off the generator and then check the last transient plaintextgen-model for fields that have a red color, which means that there is something empty or null. In many cases, you can understand then immediately which template injects a null-value and causes the problem, or (in a more involved case) you can use the earlier described generation trace in order to trace back to transient models in earlier generation steps and the templates that are involved. The following example shows the last transient model of a plaintextgen template which produces a result node called somefile.txt, where a NullPointerException is thrown during the generation process (you can immediately see that the 3rd word in the result is the null-value, so you can now easily find that part in the original template):



5. Dependency Analysis

If you have difficulties getting insight into node-to-node-dependencies, the Find Usages subsystem can be used to analyze dependencies between nodes. This is also very useful to find examples of the usage of a certain model or metalanguage construction (especially when you set the scope to global).

For module- and model-dependencies, the built-in tool is the Module Properties view. If this is not enough or if you need more graphical insight or documentation about dependencies between modules and models, the mbeddr dependenciesdiagram language may help you out (this requires installation of mbeddr into your MPS). Because a concept diagram is an editable model, it enables you to manually add or remove specific models or modules in the overview by drag-and-drop from the diagram palette (on the right). To use dependenciesdiagram, simply add the language com.mbeddr.mpsutil.dependenciesdiagram to your used languages in a model and then create a new dependenciesdiagram root node. The following picture shows an example of a dependencies diagram of the Robot Jaja language from MPS Samples and some of its dependencies:



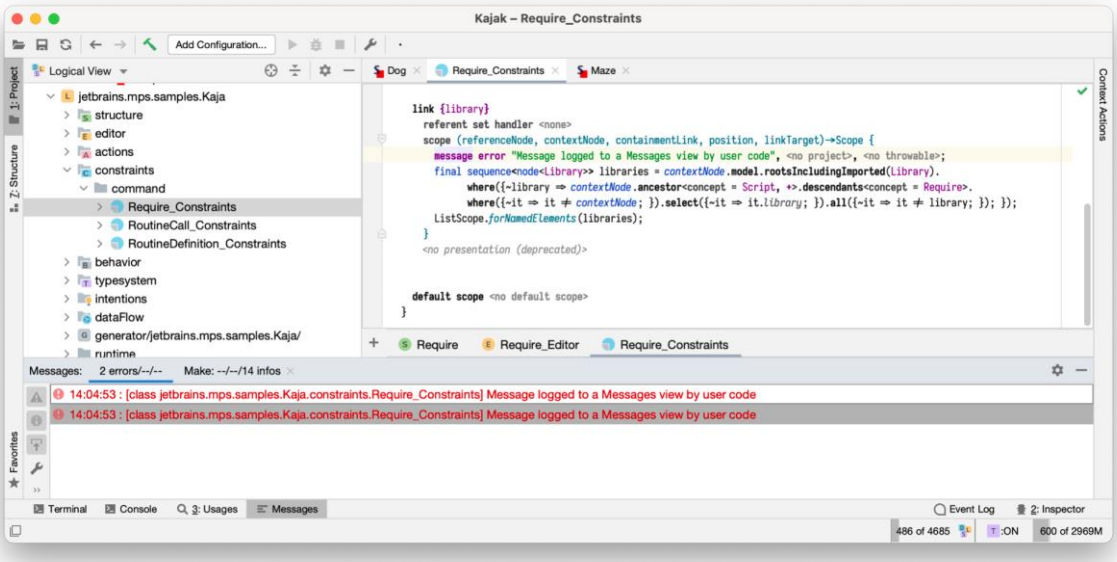
6. Other Analysis Tools

There are more specific debug tools, such as data flow graphs, trace to type system rule, nullable DFAs, show quotation metalevel, etc. These are often less used and currently out of scope of this text. However, if you have other analysis tools and example uses for them, please feel free to contact/comment.

Finally, it's good to mention that the test frameworks of MPS can often also be powerful (and even automatically reproducible) analysis tools, because they enable one to isolate small examples to better understand and at the same time get good tests for your language.

3. Logging to the Tool Messages Window

You can make your own message appear in the Messages tool window, looking like this:



1. Import the jetbrains.mps.baseLanguage.logging language into your model using Ctrl+L/Cmd+L.
2. Add message statements (message error, message info, etc.) to your code. Provide the current project if possible.
3. After you rebuild and rerun the affected code, the message will appear in the Messages view.

Note that the message will appear in a separate unnamed tab, not in the usually selected *Make* tab.

Providing the project to the message statement is optional but recommended. If you do not provide a project and you have multiple project windows open at the same time, the messages could appear in the wrong one.

4. Logging to the Console or Log File

Rather than using the Messages tool window, you can have your code log to the MPS log file:

1. Import the jetbrains.mps.baseLanguage.logging language into your model using Ctrl+L/Cmd+L.
2. Add log statements (log error, log info, etc.) to your code. Compared to logging to the Messages tool window you don't need to provide the current project.
3. By default, debug and trace level logging is disabled. To enable debug logging, use the *Help → (Diagnostic Tools →) Debug Log Settings* menu action.

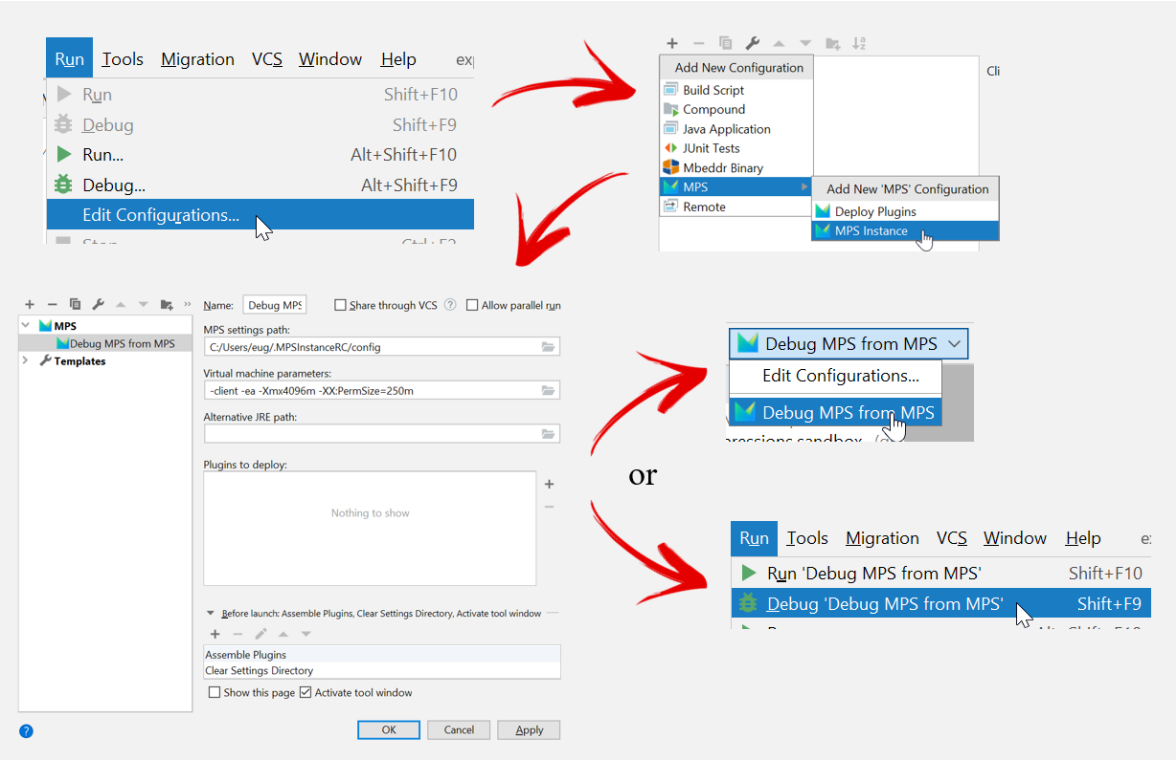
To view the logs, you can do one of the following:

- Help → Show Log in (Finder/Explorer) to find the log; this typically opens your local file browser and points to the log.
    - The file is usually called idea.log, but it rotates, so if you have a lot of logging, the log entries you are looking for may be already rotated away into idea.log.1, idea.log.2, etc.
- You can use the Fernsprecher plugin to view logs in the browser
- If you heavily use the log and need a performant, filterable solution, you can use Apache Chainsaw (see a howto here), because the MPS log is in log4j format.
- Start MPS from the Terminal/Console:
    - On Windows, modify mps.bat or mps.sh to use java instead of javaw, so that you can see raw logging in real time in a console that pops up on MPS start.
    - Be careful, selecting text in the console may block MPS on Windows!
    - On Mac, start MPS from the Terminal by running MPS<version>.app/Contents/MacOS/mps
    - MPS 2020.3 disables console logging so you will not see any output. MPS 2021.1 brings console logging back.

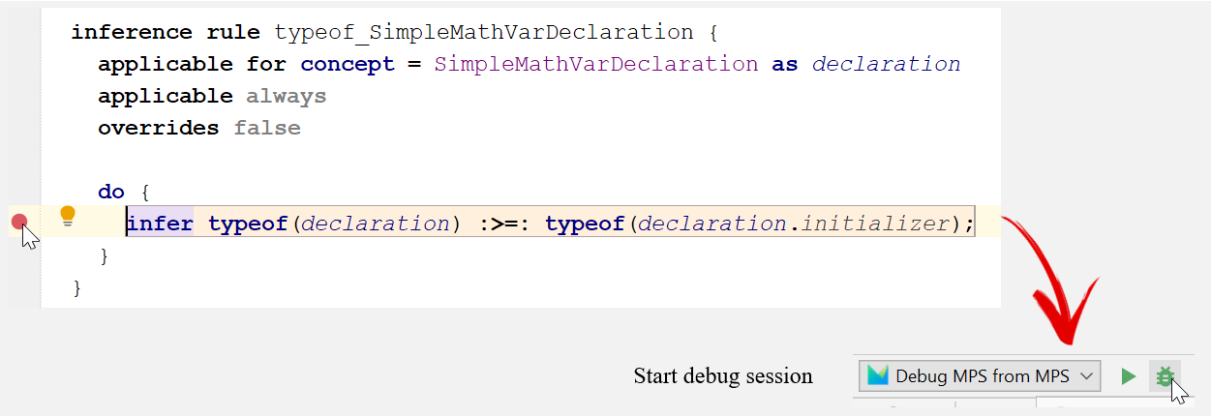
5. The Built-in MPS Debugger

You may have seen that MPS has a built-in debugger. Our experience with it is mixed. This debugger is fairly easy to set up and use (see further down for a more up-to-date instruction), compared to full-blown remote debugging (which is described in the following section).

To use the built-in debugging feature, you have to first define a run configuration to be able to start a debugging session. This can be done as follows:



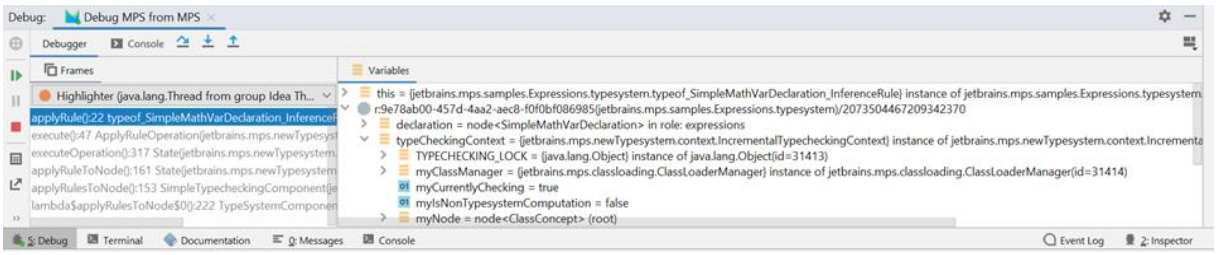
Once you have the run configuration set up, you can simply set a break-point in any piece of code in any of the aspects (e.g. a typesystem inference rule) by left-clicking in the gutter and then start the debug session:



This will start a new MPS session in which you can get the IDE into a state that triggers the breakpoint. Here, for the expressions example, we change the 10 to something else:

```
public class Simple {
  {
    simple math
      var aValue = 10 + 1.3 F
  }
}
```

Switching back to the original MPS from which the debug session was started, will give a “classic” debugging screen, which enables inspection of variables, values, frames, etc. (things that you are used to from a typical JetBrains-tool debugger):



Be aware that this type of debugging lets you step through code that was generated by MPS, but if you want to step through the part of MPS code that was written in Java (i.e. the MPS framework itself), you would be out of luck. This means that you need to have some idea of how the MPS framework calls the generated code, otherwise you may get lost in the debugging process.

If your language generates into Java via base language, your language *users* could also use the MPS debugger to debug *their* code.

6. The Nuclear Option: Remote Debugging

When all else fails you can also debug MPS remotely, just like any other Java application. In this case, “remotely” doesn’t necessarily mean “from another computer”. After some configuration you can attach to the running MPS instance a debugger running on the same computer.

Setting up remote debugging is more involved than the previous approaches but it can be much more powerful because you are able to debug MPS internals, add watch points, interactively inspect data structures, or evaluate code, and so on. In short, you can use all the debugging tools that a modern IDE such as IntelliJ IDEA provides.

Thanks to Sergej Koščejev from SpecificLanguages for contributing to this whitepaper.