pure::variants JavaScript Extensibility Guide

pure-systems GmbH

Version 5.0.8.685 for pure::variants 5.0

Copyright © 2003-2021 pure-systems GmbH

2021

Table of Contents

1. Introduction 2
2. Using 2
3. Extensibility Scope 3
4. JavaScript Support 3
    4.1. JavaScript File Creation 3
        4.1.1. Create in pure::variants Project 3
        4.1.2. Create in JavaScript Project 5
    4.2. JavaScript Library Registration 7
    4.3. JavaScript File Execution 9
        4.3.1. Direct JavaScript Execution 9
        4.3.2. Execution on Model File 10
        4.3.3. Execution in Model Editor 11
5. JavaScript Library Reference 13
    5.1. Script Context 13
    5.2. Script Stubs 14
    5.3. Project API 15
    5.4. Model Lifecycle API 15
    5.5. Model Navigation API 15
    5.6. Model Manipulation API 16
    5.7. Modularization API 16
    5.8. User Input API 17
    5.9. Script Output API 19
6. JavaScript Variant Project Template 20
7. JavaScript Examples 22
    7.1. Add two Alternative Features 22
    7.2. Replace Feature by two Alternatives 23
8. JavaScript Snippets for pure::variants 24
    8.1. Projects and Models 24
        8.1.1. Get Project Name 24
        8.1.2. Get Feature Models 24
        8.1.3. Get Family Models 25
        8.1.4. Get Variant Models 25
        8.1.5. Create Model 25
        8.1.6. Delete Model 25
        8.1.7. Get Root Element of Model 25
        8.1.8. Get All Elements of Model 26
    8.2. Elements 26
        8.2.1. Get Unique Name of Element 26
        8.2.2. Get Visible Name of Element 26
        8.2.3. Get Description of Element 26
        8.2.4. Get Direct Children of Element 27
        8.2.5. Get All Children of Element 27
        8.2.6. Get Variation Type of Element 27
        8.2.7. Rename Element 27
        8.2.8. Create Feature 28

8.2.9. Delete Element ... 28
8.2.10. Change Variation Type of Element ... 28
8.2.11. Change Range of Element with OR-Variation Type ... 29
8.3. Variants and Selections ... 29
8.3.1. Get All Selections in Variant Model ... 29
8.3.2. Get Element of Selection ... 29
8.3.3. Get Element Selection State ... 29
8.3.4. Change Element Selection ... 30
8.4. Relations ... 30
8.4.1. Add Relation ... 30
8.4.2. Change Relation Type ... 30
8.4.3. Delete Relation ... 31
8.5. Constraints ... 31
8.5.1. Add Constraint ... 31
8.5.2. Change Constraint ... 32
8.5.3. Delete Constraint ... 32
8.6. Restrictions ... 32
8.6.1. Add Restriction ... 32
8.6.2. Change Restriction ... 33
8.6.3. Delete Restriction ... 33
8.7. Attributes ... 33
8.7.1. Get Attributes of Element ... 33
8.7.2. Add Attribute to Element ... 34
8.7.3. Add Value to Attribute ... 34
8.7.4. Add Calculation to Attribute ... 35
8.7.5. Restrict Attribute Value ... 35
8.7.6. Delete Attribute ... 35
8.7.7. Delete Attribute Value ... 36
9. JavaScript Snippets for pure::variants for Simulink ... 36
9.1. Get Simulink Variability Model Information ... 36
9.1.1. Get all Variation Points in Model ... 36
9.1.2. Get Default Assignment of Variation Point ... 36
9.1.3. Get All Variations of Variation Point ... 37
9.1.4. Get Label of Variation ... 37
9.1.5. Get Value of Variation ... 37
9.1.6. Get Condition of Variation ... 37
9.2. Manipulation of Simulink Variability Model ... 37
9.2.1. Simulink Variability Model ... 37
9.2.2. Variation Point ... 38
9.2.3. Variation ... 39

1. Introduction

This manual describes how to extend the pure::variants functionality and user interface using JavaScript.

The reader is expected to have basic knowledge about and experiences with pure::variants. This manual is available in online help as well as in printable PDF format here.

2. Using

Depending on the installation method used either start the pure::variants-enabled Eclipse, or under Windows select the **pure::variants** item from the **Start** menu.

If the **Variant Management** perspective is not already activated, do so by selecting it from **Open Perspective** -> **Other...** in the **Window** menu.

3. Extensibility Scope

The JavaScript interface of pure::variants allows to run complex operations on pure::variants models. For this purpose the JavaScript interface provides a simplified API where only the minimal input data is needed. The JavaScript interface allows to make changes on Feature Models (*.xfm), Family Models (*.ccfm) or Variant Models (*.vdm). Complex changes on models can be coded in a single JavaScript script and run at once. This avoids changing the models step by step by the user and ensures consistent content of the models.

The interface to pure::variants is realized by the pure::variants JavaScript library which is documented in section Section 5, "JavaScript Library Reference".

The JavaScript API published by the pure::variants JavaScript library is based on the Java API of pure::variants. Thus implementations done in JavaScript can be easily converted to Java and vice versa. The documentation of the Java API is part of the **pure::variants Online Help**, which can be found in menu **Help**->**Help Contents**. It also describes the JavaScript API by marking Java classes, Java class methods, and Java class fields using the following annotations:

- **@jsClass** marks classes supported in JavaScript
- **@jsMethod** marks class methods supported in JavaScript
- **@jsField** marks class fields supported in JavaScript

4. JavaScript Support

pure::variants allows the editing of JavaScript scripts by providing a JavaScript editor, which is part of the Eclipse JavaScript Development Tools. This editor supports auto completion of classes as well as of methods exposed by the pure::variants JavaScript Library. It provides clearness by highlighting e.g. JavaScript key words and checks automatically implemented JavaScript code. Therefore a JavaScript file has to be created below a JavaScript project in order to enable these possibilities of the JavaScript editor.

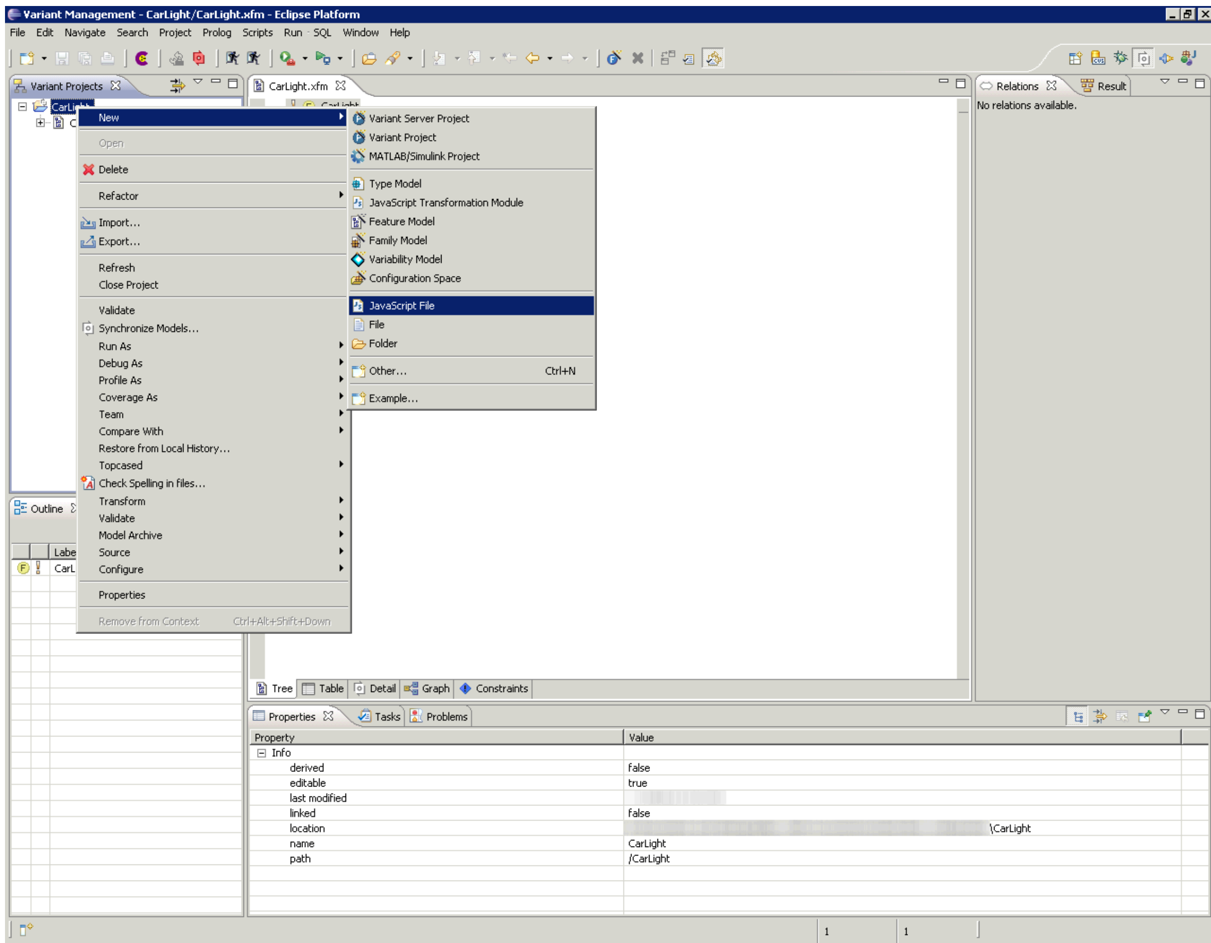
4.1. JavaScript File Creation

JavaScript files supposed to run in a pure::variants environment can be created in two ways. A JavaScript file can be created below an already existing pure::variants project, which is automatically adapted to a JavaScript project. And a JavaScript file can also be created below a JavaScript project with enabled pure::variants JavaScript support. A JavaScript file should have the extension **.js**.

4.1.1. Create in pure::variants Project

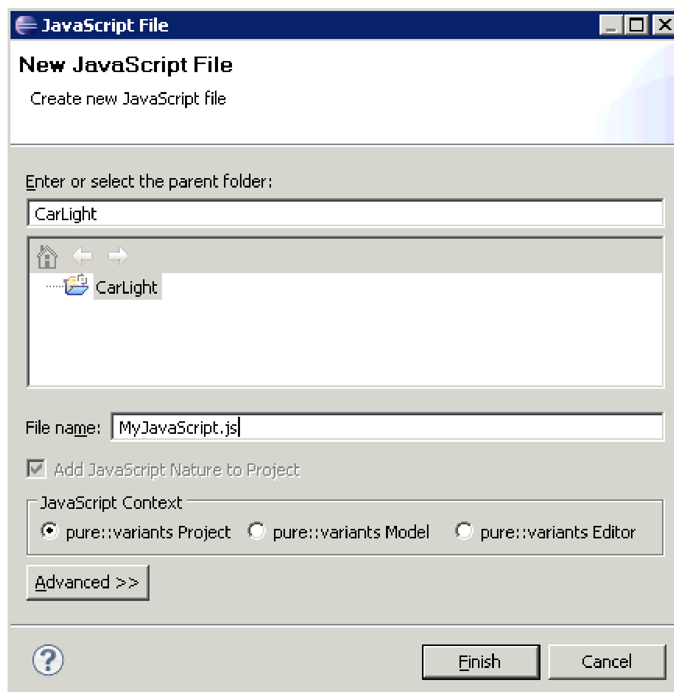
With the help of the context menu of a pure::variants project, the creation wizard of a JavaScript file can be started.

Figure 1. JavaScript Script Creation



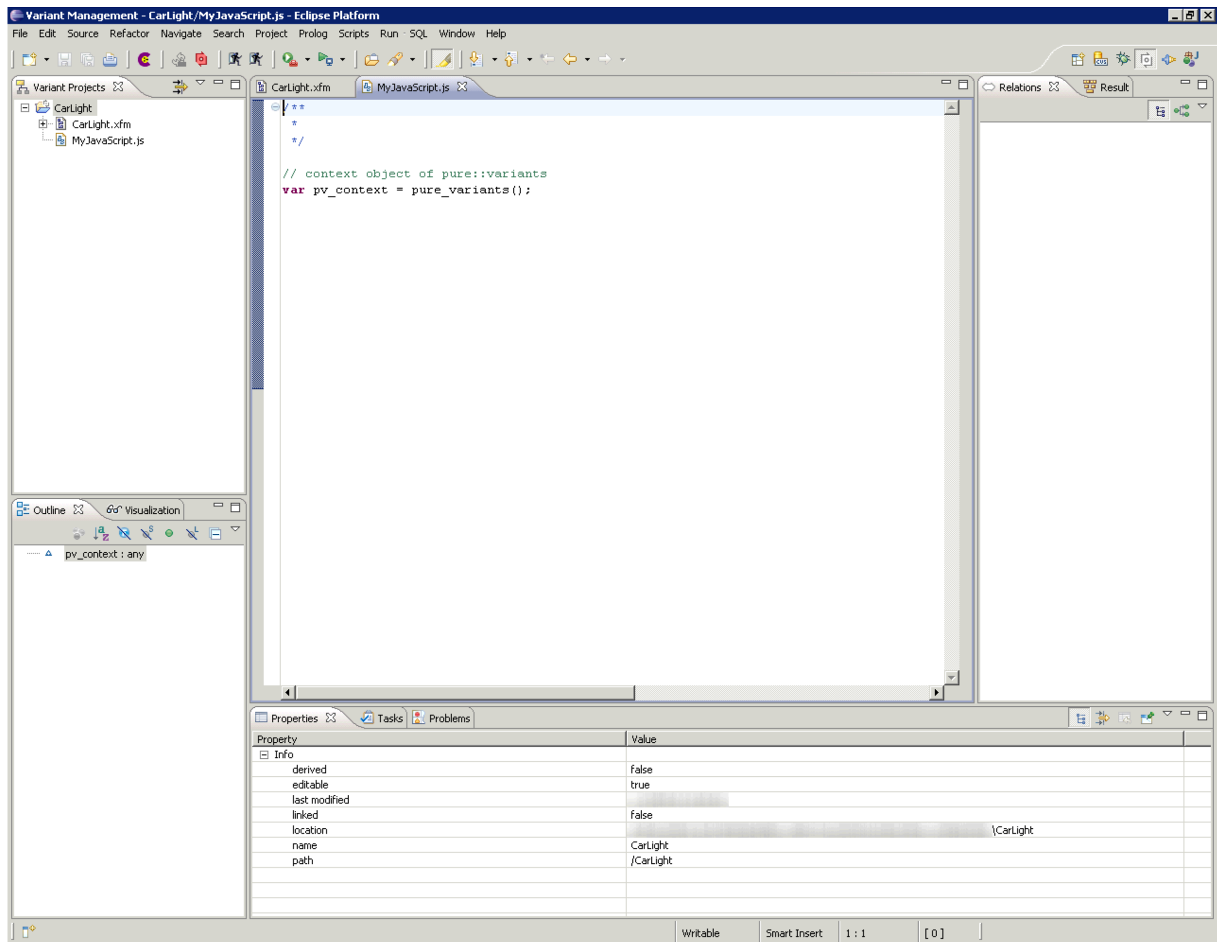
After entering the name of the JavaScript file, the context of the script has to be defined.

Figure 2. JavaScript Context Definition



Corresponding to the entered name, a new JavaScript file is created below the pure::variants project.

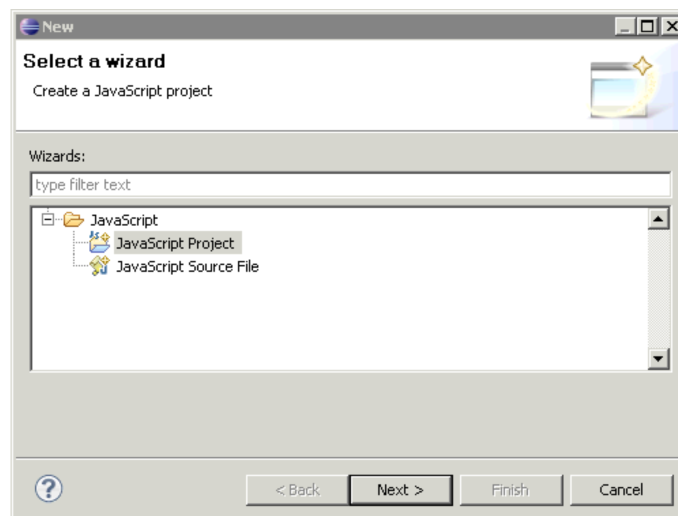
Figure 3. JavaScript Script



4.1.2. Create in JavaScript Project

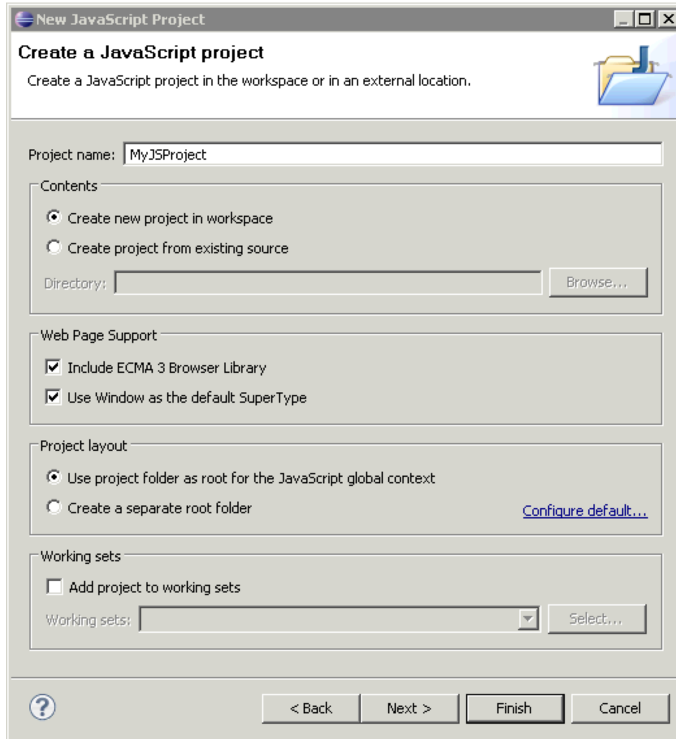
With the help of the context menu of the Variant Projects View (**New->Other...**) a new JavaScript project can be created.

Figure 4. JavaScript Project Creation



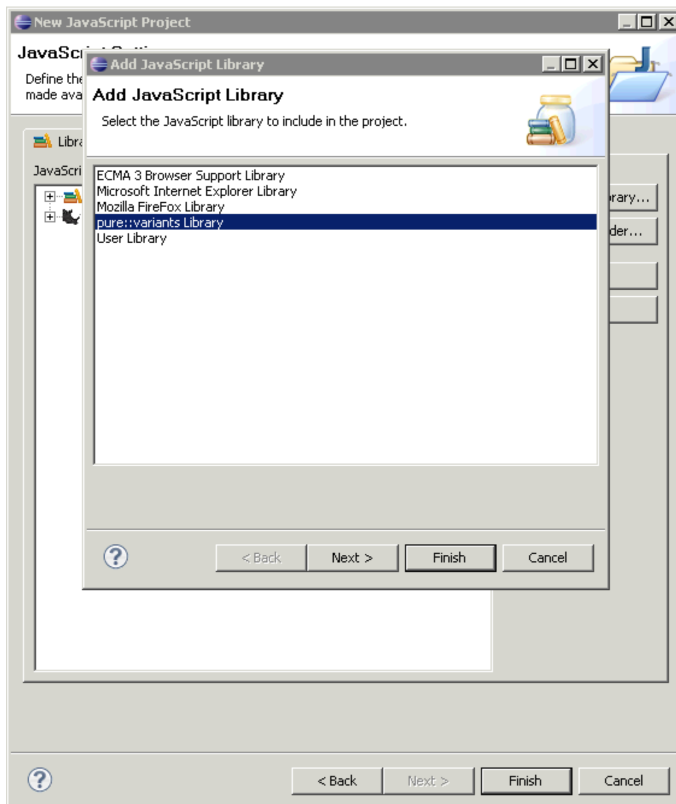
After entering a project name and managing the settings, the next page has to be opened.

Figure 5. JavaScript Project Settings



At this page the pure::variants JavaScript library has to be added to the list of already included JavaScript libraries.

Figure 6. pure::variants JavaScript Library



After adding the pure::variants JavaScript library the creation wizard can be finished. A new JavaScript file can created now below this JavaScript project.

4.2. JavaScript Library Registration

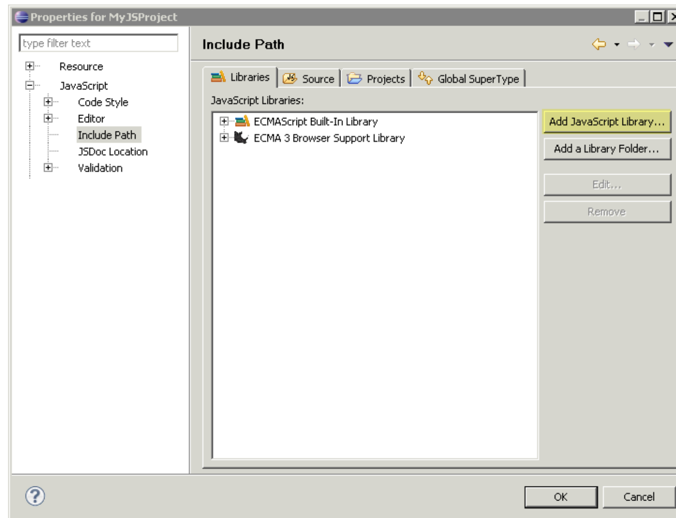
pure::variants exposes its API by the pure::variants JavaScript library. This library has to be part of the JavaScript library list of the JavaScript project. If the creation of the JavaScript file was done with the help of the pure::variants **New JavaScript File** wizard, the addition of the pure::variants JavaScript library is done automatically.

In order to check if the library is available, the properties dialog of a project has to be opened by using the context menu.

Figure 7. JavaScript Project Properties

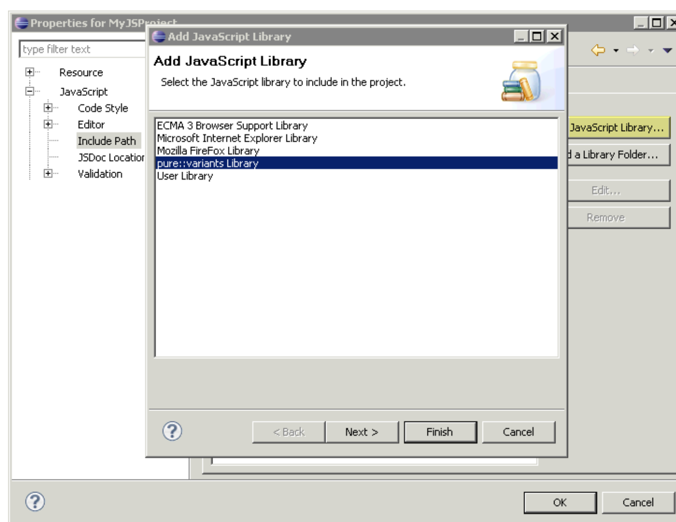
After opening the dialog, the entry **JavaScript**->**Include Path** has to be selected. The list of JavaScript libraries has to be checked, if the pure::variants JavaScript library is already included.

Figure 8. JavaScript Include Path Properties



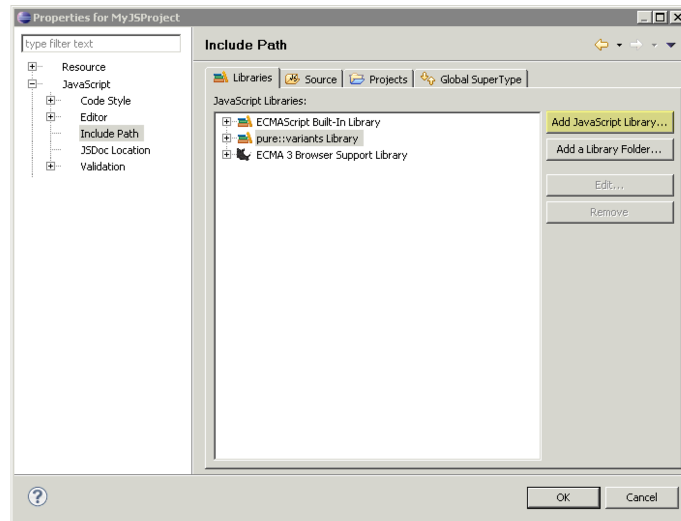
If the pure::variants JavaScript library is missing, it has to be added by clicking button **Add JavaScript Library...** and by selecting **pure::variants JavaScript Library**.

Figure 9. pure::variants JavaScript Library



After selecting the library the dialog as well as the project properties dialog has to be finished in order to include also the pure::variants JavaScript library in the list of libraries.

Figure 10. pure::variants Supported JavaScript Project



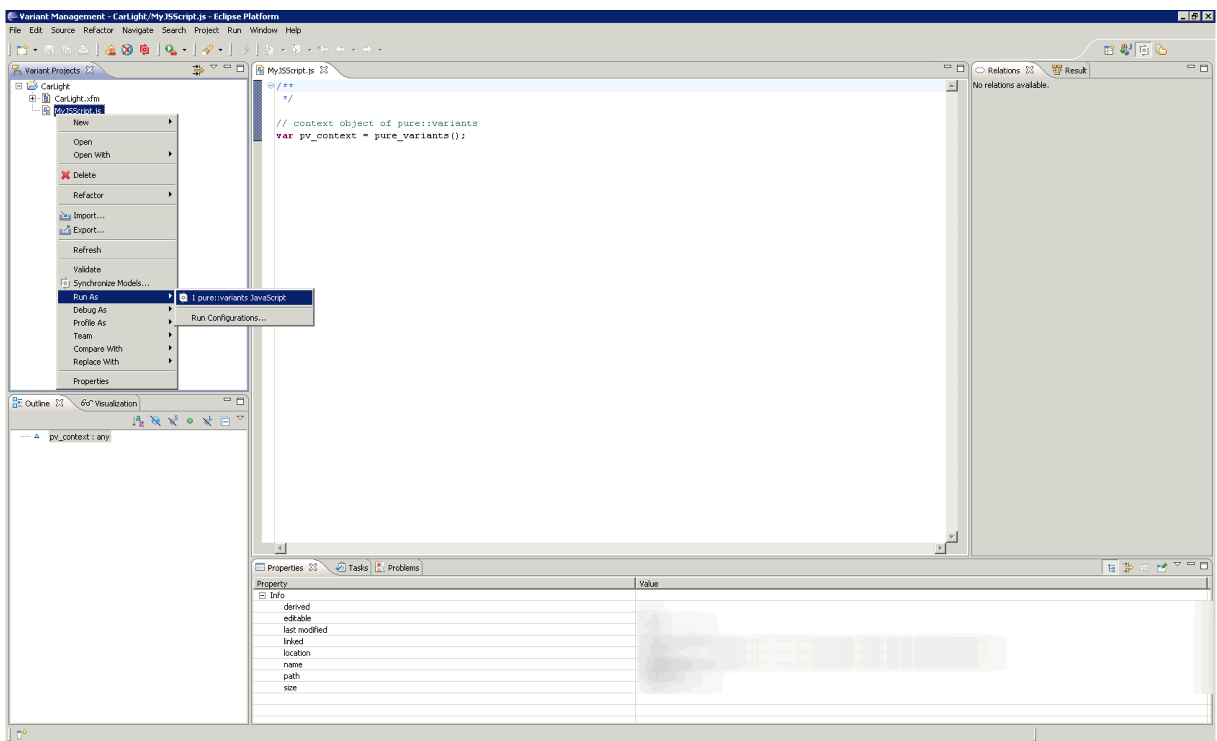
4.3. JavaScript File Execution

Depending on the selected context of a JavaScript script, it is started in different ways.

4.3.1. Direct JavaScript Execution

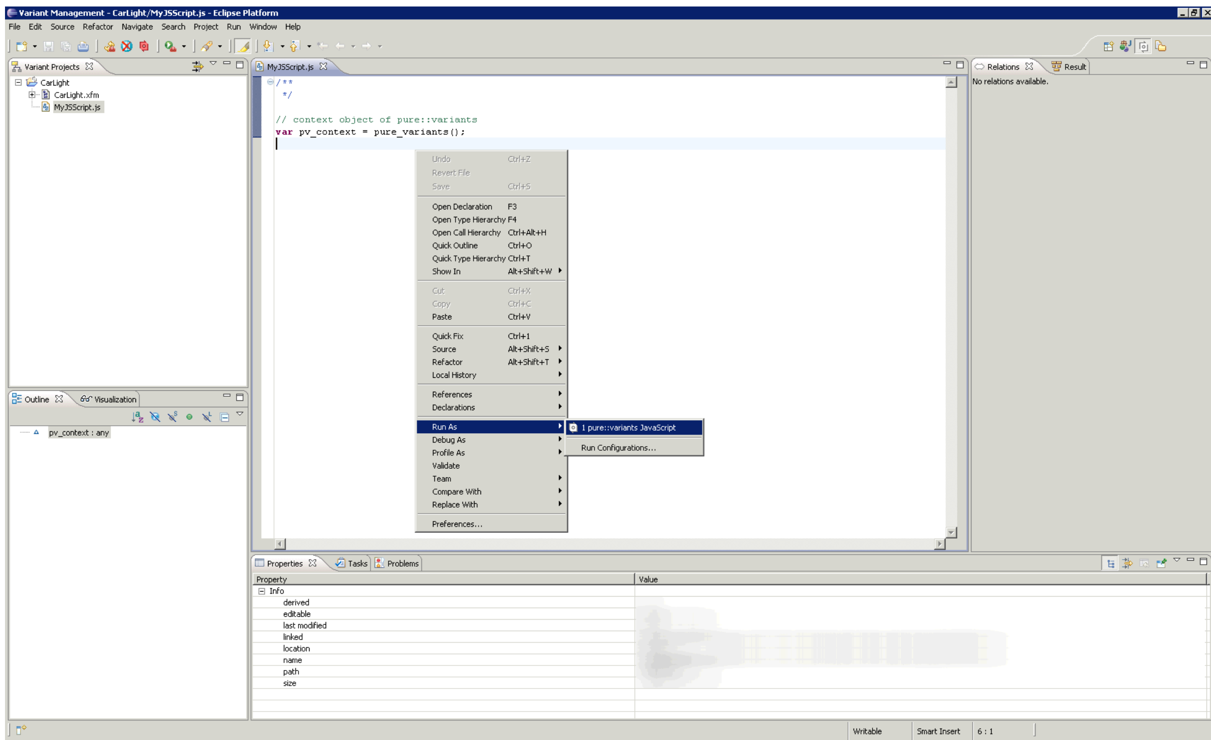
Starting a JavaScript script directly means using its context menu in the **Variant Projects View**. This context menu always contains the sub menu **Run As**, which provides the entry **pure::variants JavaScript**, for each JavaScript file. Starting a script this way runs it in the project context.

Figure 11. Direct JavaScript Execution (File)



Starting the JavaScript script in the JavaScript editor is another possibility to run a script directly. Therefore the editor context menu always contains the sub menu **Run As**, which also provides the entry **pure::variants JavaScript**.

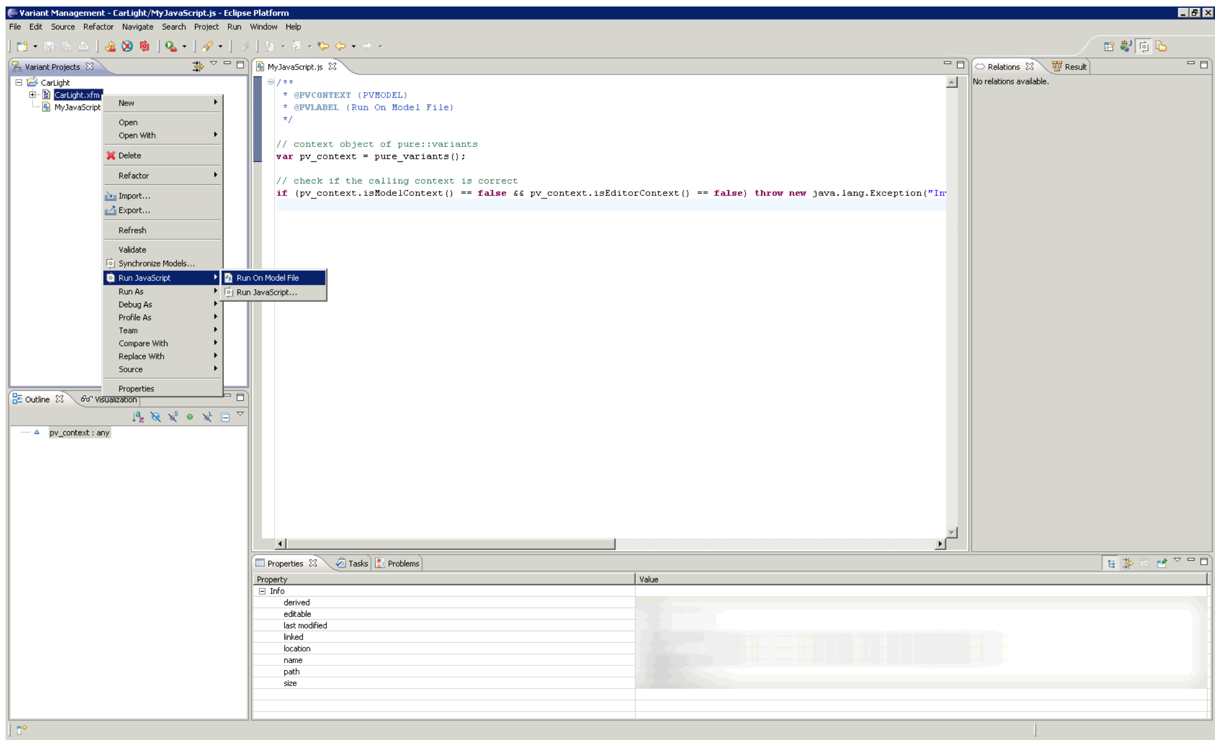
Figure 12. Direct JavaScript Execution (Editor)



4.3.2. Execution on Model File

If a Java script was created for the model file context, the script appears as an action directly in the context menu of a model file. Depending on the defined label the entry has the appropriate name. The script can be started by opening the context menu of the model file in the **Variant Projects View**. The context menu contains the pure::variants sub menu **Run JavaScript** which provides all actions which refer to implemented JavaScript scripts created for model file context.

Figure 13. Model File JavaScript Execution

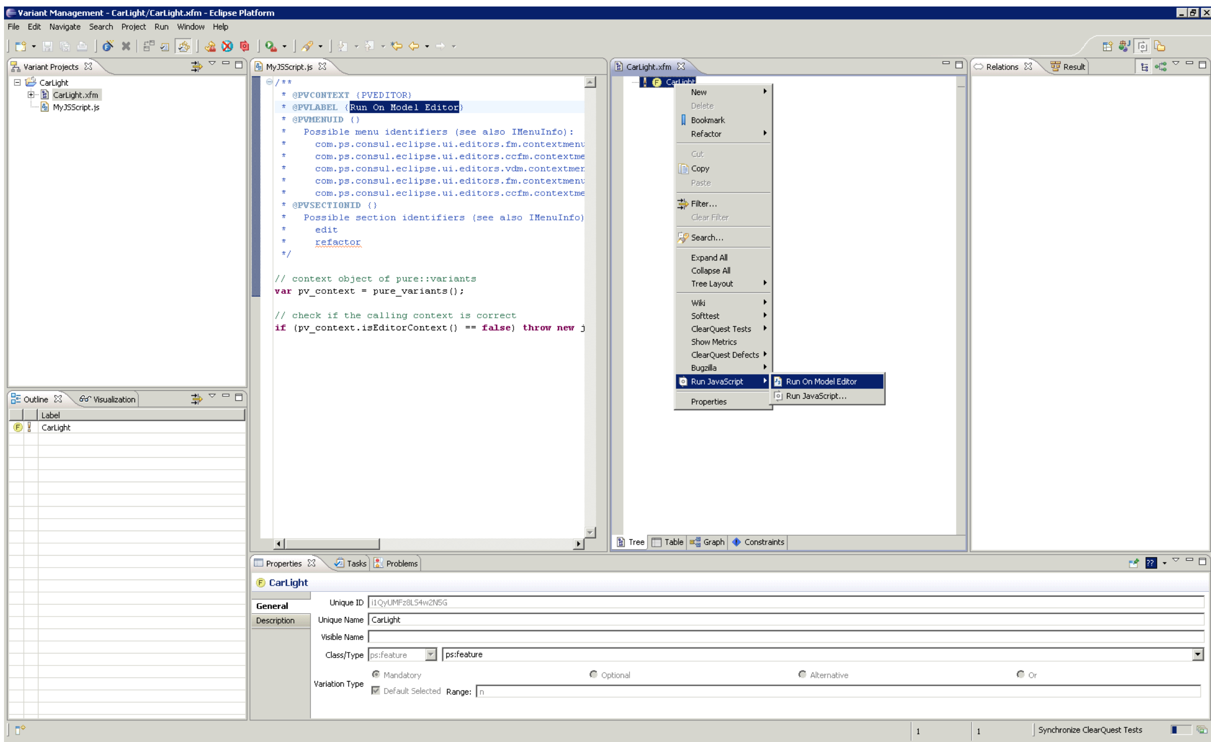


4.3.3. Execution in Model Editor

A JavaScript script can also be started from the model editor context menu. As for the model file context several entries can be available, with names as defined by the JavaScript scripts. The context menu of a model editor contains the sub menu **Run JavaScript** which provides all entries which refer to implemented JavaScript scripts created for model editor context.

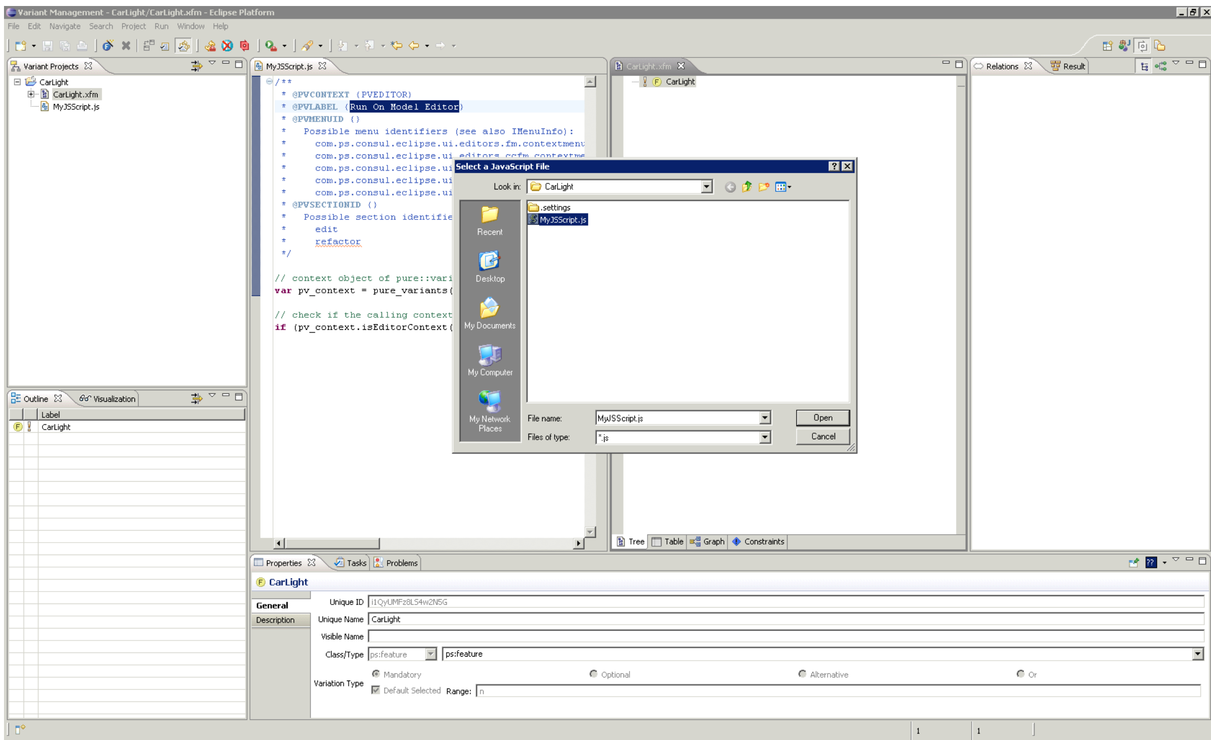
Please note, that it is also possible to restrict the appearance of a JavaScript script to a context menu of a particular model editor as well as to define another place inside a context menu. With the help of the annotation **@PVMENUID** a context menu identifier can be given in order to allow appearance only e.g. in a feature model editor. The annotation **@PVSECTIONID** defines a section inside a context menu e.g. edit section, which serves as place for the entry instead of the default section.

Figure 14. Model Editor JavaScript Execution



The context menu of a model editor as well as the context menu of a model file allows to select any JavaScript file. Both pure::variants sub menus **Run JavaScript** provides an action **Run JavaScript...** which allows to start any JavaScript script by selecting a JavaScript file inside a file selection dialog.

Figure 15. Any JavaScript Execution



5. JavaScript Library Reference

The API of the pure::variants JavaScript library can be easily explored and used while implementing a script due to auto completion and documentation available in the JavaScript editor. The auto completion works only if the type of the object is known. Thus an object which is transfered to another context such as a function must be declared by its type. For this, the type of the parameter and also the type of the return value has to be defined in the JavaScript documentation of the function. The next example illustrates how to specify the type of a function parameter and of the function's return value:

```
/** Do some work on the model. E.g. get root element.
 * @param {IPVModel} model The pure::variants model.
 * @return {IPElement} Any element
 */
function work(model) {
  // auto completion works for model.
  var element = model.getElementWithID(model.getElementsRootID());
  return element;
}

model = new PVModel();
var element = work(model);

// auto completion works for element.
var name = element.getName();
```

5.1. Script Context

pure::variants JavaScript scripts can be executed in different contexts. For this reason they have to define its context. pure::variants supports three contexts for JavaScript script execution:

- Project Context
- Model File Context
- Model Editor Context

A JavaScript script accesses pure::variants execution environment by using a pure::variants context object. This object is provided by function `pure_variants()`. Depending on the execution context of the script, the pure::variants context object contains different context related information as shown in the following table. Context independent methods are documented in the next sections.

Table 1. Context Related Methods

<table>
<tr><th>Context Object Method</th><th>Project Context</th><th>Model File Context</th><th>Model Editor Context</th></tr>
<tr><td>isProjectContext()
isModelContext()
isEditorContext()</td><td>Returns true.
Returns false.
Returns false.</td><td>Returns false.
Returns true.
Returns false.</td><td>Returns false.
Returns false.
Returns true.</td></tr>
<tr><td>getNameOfContextPro\ject()</td><td>The name of the project which contains the JavaScript file.</td><td colspan="2">The name of the project which contains the model.</td></tr>
<tr><td>getPathToContextMod\el()</td><td>An empty string.</td><td colspan="2">The absolute path of the model.</td></tr>
<tr><td>getContextModel()</td><td>null</td><td>An instance of the model.</td><td>The instance of the model used by the editor.</td></tr>
<tr><td>getContextElements()</td><td colspan="2">An empty list.</td><td>A (possibly empty) list of all elements selected in the editor.</td></tr>
</table>

Context Object Method	Project Context	Model File Context	Model Editor Context
`getContextElement()`	`null`		The first element selected in the editor, or `null` if no element selected.
`getContextItems()`	An empty list.		A (possibly empty) list of all items selected in the editor.
`getContextItem()`	`null`		The first item selected in the editor, or `null` if no item selected.

5.2. Script Stubs

As mentioned before, three contexts are defined in pure::variants and are supported while script execution. Depending on the selected context during the JavaScript file creation, different stubs of JavaScript files are generated.

If the script is intended to run directly in the context of a project, the following stub is generated.

```
// context object of pure::variants
var pv_context = pure_variants();
```

If the model file context was selected, a JavaScript comment describing the context is generated, as well as a check in order to guarantee the execution in that context. The model file context enables the Script to start directly by a provided action in the context menu of a model file in the **Variant Projects View**.

```
/**
 * @PVCONTEXT {PVMODEL}
 * @PVLABEL {Label in Model Context Menu}
 */

// context object of pure::variants
var pv_context = pure_variants();

// check if the calling context is correct
if (pv_context.isModelContext() == false && pv_context.isEditorContext() == false)
  throw new java.lang.Exception("Invalid invocation context!\n\nInvocation context must be a
 pure::variants model or a pure::variants editor.\n\n");
```

The created JavaScript file stub, which is intended to be runnable as action in the model editor context, contains an appropriate context check and additionally the possibility to place the appropriate action at different places in the context menu.

```
/**
 * @PVCONTEXT {PVEDITOR}
 * @PVLABEL {Label in Editor Context Menu}
 * @PVMENUID {}
 *   Possible menu identifiers (see also IMenuInfo):
 *     com.ps.consul.eclipse.ui.editors.fm.contextmenu
 *     com.ps.consul.eclipse.ui.editors.ccfm.contextmenu
 *     com.ps.consul.eclipse.ui.editors.vdm.contextmenu
 *     com.ps.consul.eclipse.ui.editors.fm.contextmenu.refactor
 *     com.ps.consul.eclipse.ui.editors.ccfm.contextmenu.refactor
 * @PVSECTIONID {}
 *   Possible section identifiers (see also IMenuInfo):
 *     edit
 *     refactor
 */

// context object of pure::variants
var pv_context = pure_variants();

// check if the calling context is correct
if (pv_context.isEditorContext() == false)
```

```
  throw new java.lang.Exception("Invalid invocation context!\n\nInvocation context must be a
 pure::variants editor.\n\n");
```

5.3. Project API

The project API allows to request several information about pure::variants projects. The pure::variants context object provides the necessary methods and can be accessed by calling function `pure_variants()`:

- `pure_variants().getNameOfContextProject()` returns the name of the context project
- `pure_variants().getPathOfContextProject()` returns the path to the context project
- `pure_variants().getProjectName(model)` returns the name of the project containing the given model
- `pure_variants().getFeatureModelPaths(project_name)` returns a list containing the paths to all feature models of the project with the given name
- `pure_variants().getFamilyModelPaths(project_name)` returns a list containing the paths to all family models of the project with the given name
- `pure_variants().getVariantModelPaths(project_name)` returns a list containing the paths to all variant description models of the project with the given name
- `pure_variants().getInputModels(variant_model)` returns all feature and family models of the configuration space containing the given variant description model

5.4. Model Lifecycle API

A pure::variants model has a well defined lifecycle as covered by the following methods:

- `pure_variants().createModel(path)` creates a new model at the given path
- `pure_variants().openModel(path)` opens a model located at the given path and returns an instance of it
- `pure_variants().closeModel(model)` closes the given model
- `pure_variants().deleteModel(path)` deletes a model located at the given path

Please note that opening a model always requires you to close that model again as demonstrated in the following example:

```
var model = pure_variants().openModel(path_to_model);
try {
  console().println(model.getName());
}
finally {
  pure_variants().closeModel(model);
}
```

5.5. Model Navigation API

Before navigating in a model, it has to be opened. Navigation in feature and family models differs from navigation in variant models. Feature and family models contain elements with their attributes, relations and restrictions and constraints which are organized in a tree hierarchy. Variant models contain selection elements which are managed as a flat list. A selection element in a variant model holds selection information for an element of a feature or family model. This selection information includes the selection state (e.g. selected or excluded) as well as attribute values for non-fixed attributes. A variant model does not have to have a selection element for each element of the feature and family models of the configuration space. Feature and family model elements for which no explicit selection is contained in the variant model, may be selected or excluded implicitly or not selected at all.

Navigation in Feature and Family Model:

- `model.getRoot()` returns the root element of the model
- `model.getElementWithName(element_name)` returns the element with the given name, or `null` if there is no such element
- `element.getChildren()` returns the list of direct child elements of the element
- `element.getPropertyWithName(attribute_name)` returns the element's attribute with the given name, or `null` if there is no such attribute

Navigation in Variant Model:

- `model.getSelectionOfReference(element)` returns the selection element of the given feature or family model element, or `null` if there is no selection element for the given feature or family model element
- `model.getReferenceOfSelection(selection_element_id)` returns the feature or model element referred by the selection element with the given ID, or `null` if there is no such selection element

5.6. Model Manipulation API

An opened pure::variants models exposes only a read-only interface. Thus the model itself, its elements with their attributes, relations and so on, cannot be changed directly. To get a writable interface to the model, a model operation has to be created.

- `operation = pure_variants().changeModel(model)` creates a model operation in order to get write access to the given model
- `operation.addElement(element)` adds a new element to the model
- `element = operation.changeElement(element)` returns a writable copy of the given element
- `operation.removeElement(element)` deletes the given element from the model
- `pure_variants().updateModel(operation)` performs all changes collected by the given model operation
- `pure_variants().saveModel(model)` saves the given model in order to make the changes persistent

Please note that all changes applied to a local model are not persistent until the model has been saved as demonstrated in the following example:

```
var model = pure_variants().getContextModel();
var operation = pure_variants().changeModel(model);
var element = operation.changeElement(model.getRoot());
element.setName("TheRootElement");
pure_variants().updateModel(operation);
pure_variants().saveModel(model);
```

Models managed by a pure::variants server do not need to be saved. The changes persist immediately after being applied.

5.7. Modularization API

pure::variants supports the modularization of JavaScript scripts.

- `include(script_path)` includes other JavaScript scripts into the running script. That way the functionality can be distributed over several files, and you can reuse scripts in multiple other scripts. The parameter of the include function can be an absolute or relative path to the script file to include.

```
include("C:\\Shared Scripts\\core.js");
```

```
include("mapping.js");
include("reporting.js");
var mapping = map(pvcore.getContextModel());
report(mapping);
```

The algorithm for resolving the script path is as follows:

1. Resolve the path as an absolute path.
2. Resolve the path relative to the executed JavaScript file.
3. Resolve the path relative to the project of the executed JavaScript file.

5.8. User Input API

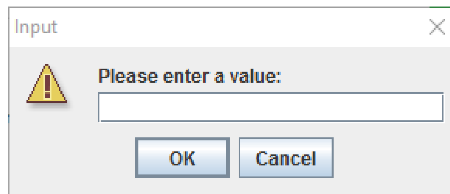
The execution of a JavaScript script can depend on user input, such as names, numbers, file paths, decisions, etc. Asking for user input is supported using the following functions:

- `askForInput(message, title)` opens a dialog window with the given title and showing the given message and an input field. If the title is omitted, then a default title is used. The function returns the text entered by the user, or `null` if the user cancelled the dialog.

  Example:

```
askForInput("Please enter a value:")
```

Figure 16. User Data

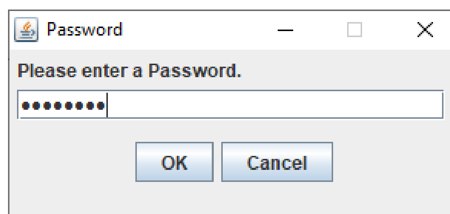


- `askForPassword(message, title)` opens a dialog window with the given title and showing the given message and a password input field. If the title is omitted, then a default title is used. The function returns the password entered by the user, or `null` if the user cancelled the dialog.

  Example:

```
askForPassword("Please enter a Password.")
```

Figure 17. User Password

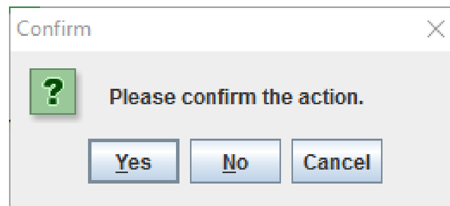


- `askForConfirmation(question, title)` opens a dialog window with the given title and showing the given question asking for confirmation. If the title is omitted, then a default title is used. The function returns `0` if the question was answered with yes, `1` if answered with no, and `2` if the dialog was cancelled by the user.

  Example:

```
askForConfirmation("Please confirm the action.")
```

Figure 18. User Confirmation

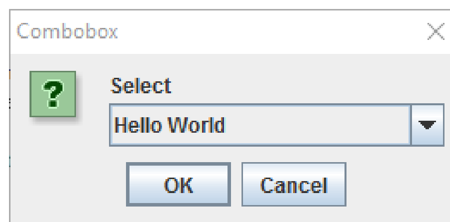


- `askUserComboBox(message, box_entries, title)` opens a dialog window with the given title and showing the given message and a selection box with the given box entries. If the title is omitted, then a default title is used. The function returns the selected entry, or `null` if nothing selected.

  Example:

```
askUserComboBox("Select", ["Hello World", "Hello", "World"], "Combobox")
```

Figure 19. User Selection



- `informUser(message, title)` opens a dialog window with the given title and showing the given message. If the title is omitted, then a default title is used.

  Example:

```
informUser("Hello World", "Information")
```

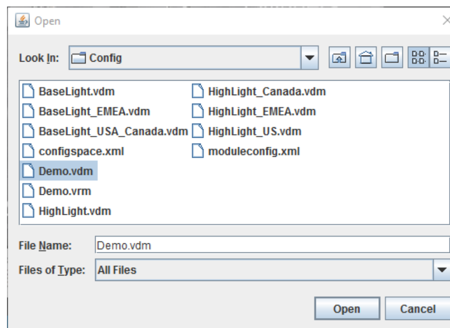
Figure 20. Informing User

- `selectFileToOpen(directory-path)` opens a file selection dialog window showing the files below the given directory path. The function returns the path to the file selected by the user, or `null` if the user cancelled the dialog.

  Example:

```
selectFileToOpen("C:\Workspace\Project\Config")
```

Figure 21. File Selection

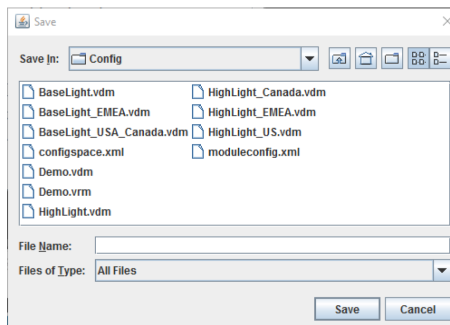


- `selectFileToSave(directory-path)` opens a file selection dialog window showing the files below the given directory path. The function returns the path to the file selected or entered by the user, or `null` if the user cancelled the dialog.

  Example:

```
selectFileToSave("C:\Workspace\Project\Config")
```

Figure 22. File Selection



5.9. Script Output API

The pure::variants JavaScript library provides the possibility to log information to a JavaScript console, and to return information to pure::variants.

- `console().write(message)` writes the given message to the JavaScript console which is part of pure::variants. The console opens automatically when new messages are written to it.

  Example:

```
console().write("Hello World!\n");
```

- `pure_variants().addSelection(item)` selects the given model item, such as an element, attribute, or relation, in the open model editor if the JavaScript script is executed in the context of a model editor. This allows for instance highlighting of newly created items or marking items for which the script has performed consistency checks.

  Example:

```
console().write("Hello World!\n");
```

6. JavaScript Variant Project Template

To use the *New -> Variant Project from Template* wizard, a JavaScript template is needed. The following example program listing shows how to create a pure::variants standard project.

This script has to be placed either in a project of the workspace, or in a central deployment folder below *%pure::variants Installation folder%/eclipse/configuration/com.ps.consul.eclipse.ui/javascripts*, or *%Eclipse configuration folder%/com.ps.consul.eclipse.ui/javascripts*.

The *Variant Project from Template* wizard searches these locations for JavaScripts with the PVNEWPROJECT context and provides them to the user.

```
/**
 * Context of this script. PVNEWPROJECT means this is a Variant Project template.
 * @PVCONTEXT {PVNEWPROJECT}
 *
 * Label for project template. This label is shown in the new project wizard.
 * @PVLABEL {Create new local standard project.}
 */

// context object of pure::variants
var pv_context = pure_variants();

/**
 * Returns the name of the project to create. This name is shown in the wizard as proposal.
 * @return The name of the project to create, or null.
 */
function getProjectName() {
  return "Example Project";
}

/**
 * Return whether the new project is a remote project.
 * @return true if new project is a remote project.
 */
function getIsRemoteProject() {
  return false;
}

/**
 * Used for remote projects only.
 * If the new project shall be a remote project the server URL has to be given.
 * @return The server URL. Must not be null if remote project is created.
 */
function getServerURL() {
  return "http://servername:port"
}

/**
 * Return whether a specific location for the project is necessary.
 * Use null here if default location shall be used.
 */
function getProjectLocation() {
  return null;
}

/**
 * Return an array with the names of referenced projects.
 * The projects have to exist in the workspace.
 */
function getReferencedProjects(){
  return [];
}

/**
 * This hook is called right after the project's creation. The project is still empty.
 * The remaining setup steps are performed after this method is called.
 */
function prepareProject() {
}
```

```
/**
 * Get the models to be created. For each model to be created return path where to
 * create the model (relative to the workspace), the model's name and type.
 * @param projectResource The project.
 * @return An array of JSON objects.
 */
function getModels(projectResource) {
  var projectName = projectResource.getName();
  var json = [
    {
      "modelPath" : projectName + "/" + projectName + ".xfm",
      "modelName" : projectName + "Features",
      "modelType" : "ps:fm"
    },
    {
     "modelPath" : projectName + "/" + projectName + ".ccfm",
     "modelName" : projectName + "Family",
     "modelType" : "ps:ccfm"
    }
  ];
  return json;
}

/**
 * Get the configurations of configuration spaces which shall be created.
 * @param projectResource The project.
 * @param modelInfos The list of models defined in getModels() function.
 * @return An array of configuration space configurations, or an empty array.
 */
function getConfigSpaces(projectResource, modelInfos) {
  var usedModels = [];
  var iter = modelInfos.iterator();
  while(iter.hasNext()){
    var modelInfo = iter.next();
    var modelJSON = {
      "modelURL" : modelInfo.getModelURL(),
    };
    usedModels.push(modelJSON);
  }

  var json = {
    "configSpacePath" : projectResource.getName() + "/config",
    "usedModels" : usedModels,
    "moduleConfigList" : {
      "moduleconfigs" : [
        {
          "name" : "Default",
          "modules" : [
            {
              "name" : "Convert to transformer action list",
              "tname" : "standard transformation",
            },
            {
              "name" : "Execute transformer action list",
              "tname" : "actionlist",
              "parameter" : [
                {
                  "name" : "destroy",
                  "value" : "false"
                }
              ]
            }
          ]
        }
      ]
    }
  };
  return json;
}

/**
```

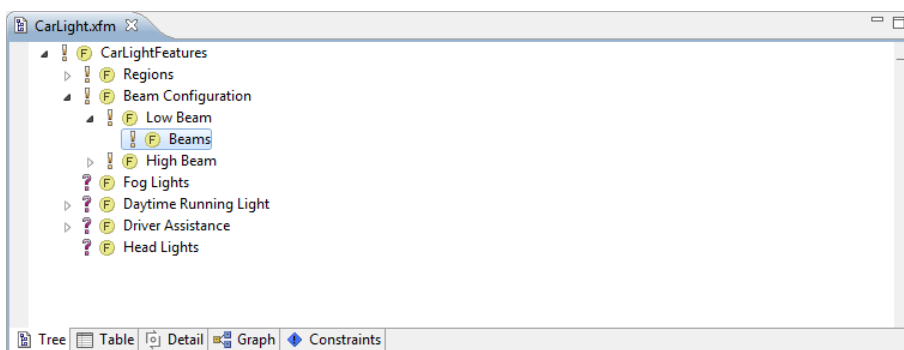
```
 * Get the models to be created. For each model to be created return
 * the path where to create the model (relative to the workspace),
 * the model's name and type.
 * @param projectResource The project.
 * @param configSpaceList The list of configuration spaces.
 * @return An array of JSON objects.
 */
function getVariantModels(projectResource, configSpaceList) {
  var cs = configSpaceList.get(0);
  var csRelPath = cs.getConfigSpaceResource().getParent().getProjectRelativePath();
  var csPath = projectResource.getName() + "/" + csRelPath.toOSString();
  var json = [
    {
      "modelPath" : csPath + "/" + projectResource.getName() + ".vdm",
      "modelName" : projectResource.getName() + "Variant"
    }
  ];
  return json;
}

/**
 * This hook is called after the project has been fully created. It is intended to be
 * used for project customization like adding initial content to the created models.
 * @param project The project.
 * @param models The list of models in the project.
 * @param variantsmodels The list of variant description models in the project.
 * @param configspaces The list of configuration spaces in the project.
 */
function customizeProject(project, models, variantmodels, configspaces) {
}
```

7. JavaScript Examples

The following examples illustrate different ways of creating features. The first example adds a group of two alternative features below an existing feature. The second example changes an existing feature and adds another to form a group of two alternative features.

The picture below shows the initial state of the feature model before any of the examples is executed, highlighting the feature **Beams** for which both examples are intended to be executed.

Figure 23. Initial Feature Model



7.1. Add two Alternative Features

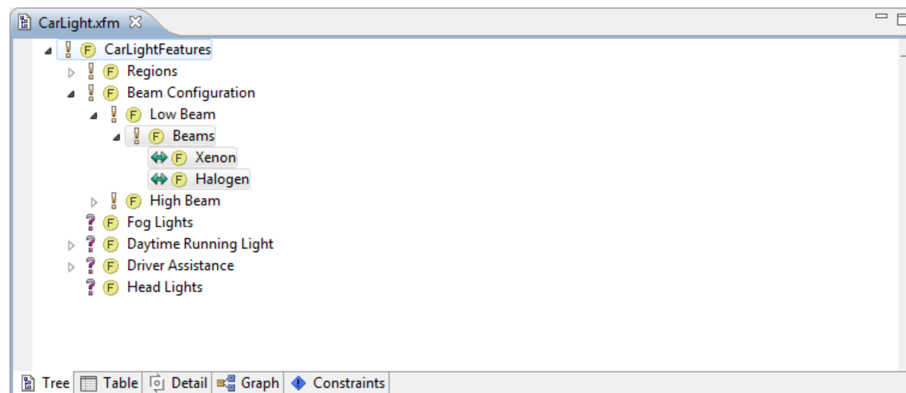
In this example two alternative features named **Xenon** and **Halogen** are created as direct children of the context feature. The following JavaScript is intended to be executed on the feature **Beams**.

```
// Prepare new feature "Xenon"
var xenon = Element();
xenon.setKlass(ModelConstants().FEATURE_CLASS);
xenon.setType(ModelConstants().FEATURE_CLASS);
xenon.setName("Xenon");
```

```
// Prepare new feature "Halogen"
var halogen = Element();
halogen.setKlass(ModelConstants().FEATURE_CLASS);
halogen.setType(ModelConstants().FEATURE_CLASS);
halogen.setName("Halogen");

// Get the feature on which this script was executed
// As mentioned above, this shall be feature "Beams"
var beams = pure_variants().getContextElement();

// Prepare the feature model to be changed
var operation = pure_variants().changeModel(pure_variants().getContextModel());

// Add both "Xenon" and "Halogen" as alternative features below "Beams"
operation.addElement(xenon, beams, ModelConstants().ALTERNATIVE_TYPE);
operation.addElement(halogen, beams, ModelConstants().ALTERNATIVE_TYPE);

// Apply the changes
pure_variants().updateModel(operation);
```

The following picture shows the result.

Figure 24. Added Features



7.2. Replace Feature by two Alternatives

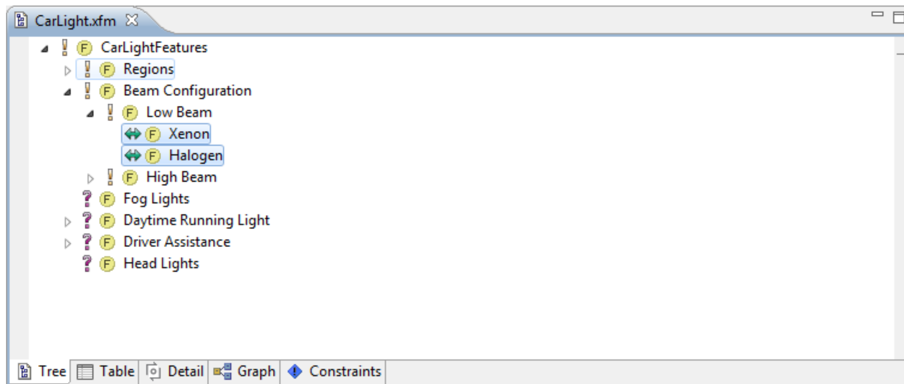
In this example the feature **Beams** is changed into an alternative feature named **Xenon**, and a second alternative feature named **Halogen** is added. The following JavaScript is intended to be executed on the feature **Beams**.

```
// Prepare the feature model to be changed
var operation = pure_variants().changeModel(pure_variants().getContextModel());

// Get the feature on which this script was executed
// As mentioned above, this shall be feature "Beams"
var beams = pure_variants().getContextElement();

// Rename the feature "Beams" to "Xenon"
var xenon = new Element(operation.changeElement(beams));
xenon.setName("Xenon");

// Change the variation type of "Xenon" to alternative
// This has to be done on the parent element "Low Beams" of "Xenon"
var lowBeams = new Element(operation.changeElement(xenon.getParentID()));
Operations().setVariationType(lowBeams, xenon, ModelConstants().ALTERNATIVE_TYPE);

// Prepare new feature "Halogen"
var halogen = Element();
halogen.setKlass(ModelConstants().FEATURE_CLASS);
halogen.setType(ModelConstants().FEATURE_CLASS);
halogen.setName("Halogen");

// Add "Halogen" as second alternative feature beside "Xenon"
```

```
operation.addElement(halogen, lowBeams, ModelConstants().ALTERNATIVE_TYPE);

// Apply the changes
pure_variants().updateModel(operation);
```

The following picture shows the result.

Figure 25. Changed and Added Features



8. JavaScript Snippets for pure::variants

In the following section JavaScript snippets are listed that show the basic functionality of the pure::variants JavaScript library.

8.1. Projects and Models

8.1.1. Get Project Name

```
/**
 * Gets the context project's name.
 * @return {String} The project's name.
 */
function getProjectName() {
  var project_name = pure_variants().getNameOfContextProject();
  return project_name;
}
```

```
/**
 * Gets the name of the project containing the model.
 * @param {IPVModel} model The model.
 * @return {String} The project's name.
 */
function getProjectName(model) {
  var project_name = pure_variants().getProjectName(model);
  return project_name;
}
```

8.1.2. Get Feature Models

```
/**
 * Gets all Feature Models of the project.
 * @param {String} project_name Name of the project.
 * @return {List} List of feature models.
 */
function getFeatureModels(project_name) {
  var featuremodel_path_list = pure_variants().getFeatureModelPaths(project_name);
  return featuremodel_path_list;
}
```

8.1.3. Get Family Models

```
/**
 * Gets all Family Models of the project.
 * @param {String} project_name Name of the project.
 * @return {List} List of family models.
 */
function getFamiliyModels(project_name) {
  var familymodel_path_list = pure_variants().getFamilyModelPaths(project_name);
  return familymodel_path_list;
}
```

8.1.4. Get Variant Models

```
/**
 * Gets the Variant Models of the project.
 * @param {String} project_name Name of the project.
 * @return {List} List of feature models.
 */
function getVariantModels(project_name) {
  var variantmodel_path_list = pure_variants().getVariantModelPaths(project_name);
  return variantmodel_path_list;
}
```

8.1.5. Create Model

```
/**
 * Creates a feature model of given name at the given path.
 * @param {String} path The path where to create the model.
 * @param {String} name The name of the new model.
 * @param {String} type The type of the model.
 */
function createFeatureModel(path, name, type) {
  pure_variants().createModel(path, name, type);
}
```

The following model types are to be used:

- `ModelConstants().FM_TYPE` for creating feature models
- `ModelConstants().CCFM_TYPE` for creating family models
- `ModelConstants().VDM_TYPE` for creating variant description models

8.1.6. Delete Model

```
/**
 * Deletes the model at the given path.
 * @param {String} path The path to the model.
 */
function deleteModel(path) {
  pure_variants().deleteModel(path);
}
```

8.1.7. Get Root Element of Model

```
/**
 * Gets the root element of the given model.
 * @param {IPVModel} model The model.
 * @return {IPVElement} The root element.
 */
function getRootElement(model) {
  var root_element = model.getRoot();
  return root_element;
}
```

8.1.8. Get All Elements of Model

```
/**
 * Print each element in the model.
 * @param {IPVModel} model The model.
 */
function printModelElements(model) {
  var element_list = model.getElementList();
  var idx = element_list.iterator();
  while (idx.hasNext()) {
    var element = idx.next();
    console().println(element.getVName() + " (" + element.getName() + ")");
  }
}
```

8.2. Elements

8.2.1. Get Unique Name of Element

```
/**
 * Gets the unique name of the element.
 * @param {IPVElement} element The element.
 * @return {String} The unique name.
 */
function getUniqueName(element) {
  // Features must have a unique name, whereas elements of
  // family and variant models can have a unique name.
  var uname = element.getName();
  return uname;
}
```

8.2.2. Get Visible Name of Element

```
/**
 * Gets the visible name of the element.
 * @param {IPVElement} element The element.
 * @return {String} The visible name of the element.
 */
function getVisibleName(element) {
  // The Visible Name is optional and can be any String.
  var vname = element.getVName();
  return vname;
}
```

8.2.3. Get Description of Element

The mimetype of a description is defined in the model and can be retrieved with `element.getModelContainer().getMimeType()`. All descriptions are added using this mimetype.

```
/**
 * Gets the description of the element.
 * @param {IPVElement} element The element.
 * @return {String} The description.
 */
function getDescription(element) {
  // Read standard description (e.g. plain/html).
  var desc = element.getDesc(element.getModelContainer().getMimeType());
  return desc;
}
```

```
/**
 * Gets the description plain text of the element.
 * @param {IPVElement} element The element.
 * @return {String} The description text.
 */
function getText(element) {
```

```
  // Read plain text description (plain/text).
  var text = element.getText();
  return text;
}
```

8.2.4. Get Direct Children of Element

```
/**
 * Print the direct children of the given element.
 * @param {IPVElement} element The element.
 */
function printDirectChildren(element) {
  var element_list = element.getChildren();
  var idx = element_list.iterator();
  while (idx.hasNext()) {
    var child_element = idx.next();
    console().println(child_element.getVName() + " (" + child_element.getName() + ")");
  }
}
```

8.2.5. Get All Children of Element

```
/**
 * Print all children of the given element.
 * @param {IPVElement} element The element.
 * @return {Array} Array with all the children.
 */
function getAllChildren(element) {
  var children_array = new Array();
  var element_list = element.getChildren();
  var idx = element_list.iterator();
  while (idx.hasNext()) {
    var child_element = idx.next();
    children_array.push(child_element);
    children_array = children_array.concat(getAllChildren(child_element));
  }
  return children_array;
}
```

8.2.6. Get Variation Type of Element

```
/**
 * Gets the variation type of the given element.
 * @param {IPVElement} element The element.
 * @return {String} The variation type.
 */
function getVariationType(element) {
  var variation_type = element.getVariationType();
  return variation_type;
}
```

The following variation types are supported:

- `ModelConstants().MANDATORY_TYPE`
- `ModelConstants().OPTIONAL_TYPE`
- `ModelConstants().ALTERNATIVE_TYPE`
- `ModelConstants().OR_TYPE`

8.2.7. Rename Element

```
/**
 * Sets the unique and visible names of the element. The visible name
 * uses the default encoding (UTF-8) and the default mime type (text/plain).
```

```
 * @param {IPVElement} element The element to change.
 * @param {String} uname The new unique name.
 * @param {String} vname The new visible name.
 */
function renameFeature(element, uname, vname) {
  var model = element.getModelContainer();
  var operation = pure_variants().changeModel(model);

  element = operation.changeElement(element);
  element.setName(uname);
  Operations().setVName(element, vname, model.getMimeType());

  pure_variants().updateModel(operation);
}
```

8.2.8. Create Feature

```
/**
 * Adds a feature to the given model.
 * @param {IPVElement} parent The feature's parent feature.
 * @param {String} uname The feature's unique name.
 * @param {String} vname The feature's visible name.
 * @param {String} variationType The feature's variation type.
 * @return {IPVElement} The new feature.
 */
function addFeature(parent, uname, vname, variationType) {
  var feature = new Element();
  feature.setKlass(ModelConstants().FEATURE_CLASS);
  feature.setType(ModelConstants().ELEMENT_FEATURE_TYPE);
  feature.setName(uname);
  Operations().setVName(feature, vname);

  var model = parent.getModelContainer();
  var operation = pure_variants().changeModel(model);
  operation.addElement(feature, parent, variationType);
  pure_variants().updateModel(operation);

  return model.getElementWithID(feature.getID());
}
```

8.2.9. Delete Element

```
/**
 * Deletes the given element from the given model.
 * @param {IPVElement} element The element to delete.
 */
function deleteElement(element) {
  var model = element.getModelContainer();
  var operation = pure_variants().changeModel(model);

  operation.removeElement(element);

  pure_variants().updateModel(operation);
}
```

8.2.10. Change Variation Type of Element

```
/**
 * Changes the variation type of the given element to the given type.
 * @param {IPVElement} element The element to change.
 * @param {String} variationType The new variation type of the element.
 */
function changeVariationType(element, variationType) {
  var operation = pure_variants().changeModel(element.getModelContainer());

  var parent = operation.changeElement(element.getParentID());
  element = operation.changeElement(element);

  Operations().setVariationType(parent, element, variationType);
```

```
  pure_variants().updateModel(operation);
}
```

8.2.11. Change Range of Element with OR-Variation Type

```
/**
 * Changes the range of the given OR-grouped element.
 * @param {IPVElement} element The element.
 * @param {String} range The new range of the element.
 */
function changeRange(element, range) {
  var operation = pure_variants().changeModel(element.getModelContainer());

  var parent = operation.changeElement(element.getParentID());
  element = operation.changeElement(element);

  Operations().setVariationType(parent, element, ModelConstants().OR_TYPE, range);

  pure_variants().updateModel(operation);
}
```

8.3. Variants and Selections

8.3.1. Get All Selections in Variant Model

```
/**
 * Gets all selections of the given variant model.
 * @param {IPVVariantModel} variantModel The variant description model.
 * @return {List} The list of selections.
 */
function getSelections(variantModel) {
  var selections_list = variantModel.getSelectionList();
  return selections_list;
}
```

8.3.2. Get Element of Selection

```
/**
 * Returns the element for which a selection decision was made.
 * @param {IPVVariantElement} selection The selection.
 * @return {IPVElement} The element regarding the selection.
 */
function getElementFromSelection(selection) {
  var element = selection.getElement();
  return element;
}
```

8.3.3. Get Element Selection State

```
/**
 * Print the selection state of the given element in the given variant.
 * @param {IPVVariantModel} variantModel The variant description model.
 * @param {IPVElement} element The model element.
 */
function printSelectionState(variantModel, element) {
  var selection_state = variantModel.getState(element).getSelection();
  if (selection_state == VariantElementState().SELECTION) {
    console().println("Element " + element.getName() + " is selected");
  } else if (selection_state == VariantElementState().EXCLUSION) {
    console().println("Element " + element.getName() + " is excluded");
  } else if (selection_state == VariantElementState().NONSELECTION) {
    console().println("Element " + element.getName() + " is not selectable");
  } else if (selection_state == VariantElementState().UNSELECTION) {
    console().println("Element " + element.getName() + " is not selected");
  } else {
    console().println("Element " + element.getName() + " is not selected");
```

```
  }
}
```

```
/**
 * Get the selection state of the given selection.
 * @param {IPVVariantElement} selection The selection element.
 * @return {String} One of "SELECTED", "NOTSELECTED", "EXCLUDED", and "NONSELECTABLE".
 */
function getSelectionState(selection) {
  var state = selection.getSelection();
  var result = "";
  if (state == VariantElementState().SELECTION) {
    result = "SELECTED";
  } else if (state == VariantElementState().EXCLUSION) {
    result = "EXCLUDED";
  } else if (state == VariantElementState().NONSELECTION) {
    result = "NONSELECTABLE";
  } else if (state == VariantElementState().UNSELECTION ||
             state == VariantElementState().NOT_AVAILABLE) {
    result = "NOTSELECTED";
  }
  return result;
}
```

8.3.4. Change Element Selection

```
/**
 * Changes the selection of a given element in the given variant.
 * @param {IPVVariantModel} variantModel The variant description model.
 * @param {IPVElement} element The element for which to change the selection.
 * @param {Number} newState The new selection state of the element.
 */
function changeSelection(variantModel, element, newState) {
  var operation = pure_variants().changeVariantModel(variantModel);
  var selection = variantModel.getSelectionOfReference(element);
  if (selection != null) {
    operation.changeSelection(element, newState);
  }
  else {
    operation.addSelection(element, newState);
  }
  pure_variants().updateModel(operation);
}
```

8.4. Relations

8.4.1. Add Relation

```
/**
 * Add a requires-dependency from the given element to the root element.
 * @param {IPVElement} element The element.
 */
function addRequiresRelation(element) {
  var model = element.getModelContainer();
  var operation = pure_variants().changeModel(model);

  element = operation.changeElement(element);
  var target = model.getRoot();
  Operations().addDependency(element, ModelConstants().REQUIRES_TYPE, target);

  pure_variants().updateModel(operation);
}
```

8.4.2. Change Relation Type

```
/**
 * Changes the element's relations of the given type to another type.
```

```
 * @param {IPVElement} element The element.
 * @param {String} type The relation type to change.
 * @param {String} newtype The new relation type.
 */
function changeRelationType(element, type, newtype) {
  var model = element.getModelContainer();
  var operation = pure_variants().changeModel(model);
  element = operation.changeElement(element);

  var relation_list = element.getRelationListOfKlass(ModelConstants().DEPENDENCIES_CLASS);
  var idx = relation_list.iterator();
  while (idx.hasNext()) {
    var relation = idx.next();
    if (relation.getType().equals(type)) {
      relation.setType(newtype);
    }
  }

  pure_variants().updateModel(operation);
}
```

```
/**
 * Changes the type of the given relation to another type.
 * @param {IPVRelation} relation The relation to change.
 * @param {String} newtype The new relation type.
 */
function changeRelationType(relation, newtype) {
  var model = relation.getModelContainer();
  var operation = pure_variants().changeModel(model);

  relation = operation.change(relation);
  relation.setType(newtype);

  pure_variants().updateModel(operation);
}
```

8.4.3. Delete Relation

```
/**
 * Remove all requires-relations from the given element.
 * @param {IPVElement} element The element.
 */
function deleteRequiresRelations(element) {
  var model = element.getModelContainer();
  var operation = pure_variants().changeModel(model);
  element = operation.changeElement(element);

  var relations = element.getRelationListOfKlass(ModelConstants().DEPENDENCIES_CLASS);
  var idx = relations.iterator();
  while (idx.hasNext()) {
    var relation = idx.next();
    if (relation.getType().equals(ModelConstants().REQUIRES_TYPE)) {
      var targets = relation.getTargetList().iterator();
      while (targets.hasNext()) {
        var target = targets.next().getTarget();
        Operations().remDependency(element, ModelConstants().REQUIRES_TYPE, target);
      }
    }
  }

  pure_variants().updateModel(operation);
}
```

8.5. Constraints

8.5.1. Add Constraint

```
/**
```

```
 * Adds a constraint to the given element.
 * @param {IPVElement} element The element.
 * @param {String} expr The constraint expression.
 */
function addConstraint(element, expr) {
  var model = element.getModelContainer();
  var operation = pure_variants().changeModel(model);
  var language = ModelConstants().SIMPLE_CONSTRAINT_LANGUAGE;

  var constraintSet = element.getConstraintSet();
  if (constraintSet == null) {
    constraintSet = Operations().makeConstraint(expr, language);
    operation.addConstraintSet(element, constraintSet);
  }
  else {
    constraintSet = operation.changeRestSet(constraintSet);
    Operations().addConstraint(constraintSet, expr, language);
  }

  pure_variants().updateModel(operation);
}
```

8.5.2. Change Constraint

```
/**
 * Change the expression of a constraint.
 * @param {IPVRestSet} constraintSet The constraint set containing the constraint.
 * @param {IPVRestriction} constraint The constraint to change.
 * @param {String} expr The new constraint expression.
 */
function changeConstraint(constraintSet, constraint, expr) {
  var model = constraintSet.getModelContainer();
  var operation = pure_variants().changeModel(model);

  constraintSet = operation.changeRestSet(constraintSet);
  var script = constraintSet.getFirstRestriction().getFirstScript();
  script.setContent(expr);

  pure_variants().updateModel(operation);
}
```

8.5.3. Delete Constraint

```
/**
 * Deletes all constraints from the given element.
 * @param {IPVElement} element The element.
 */
function deleteConstraints(element) {
  var model = element.getModelContainer();
  var operation = pure_variants().changeModel(model);

  element = operation.changeElement(element);
  var constraintSet = element.getConstraintSet();
  if (constraintSet != null) {
    operation.removeRestSet(constraintSet);
    element.unsetConstraintID();
  }

  pure_variants().updateModel(operation);
}
```

8.6. Restrictions

8.6.1. Add Restriction

```
/**
 * Adds a restriction to the given element.
```

```
 * @param {IPVElement} element The element.
 * @param {String} expr The restriction expression.
 */
function addRestriction(element, expr) {
  var model = element.getModelContainer();
  var operation = pure_variants().changeModel(model);
  var language = ModelConstants().SIMPLE_CONSTRAINT_LANGUAGE;

  var restrictionSet = element.getRestrictionSet();
  if (restrictionSet == null) {
    restrictionSet = Operations().makeRestriction(expr, language);
    operation.addRestrictionSet(element, restrictionSet);
  }
  else {
    restrictionSet = operation.changeRestSet(restrictionSet);
    Operations().addRestriction(restrictionSet, expr, language);
  }

  pure_variants().updateModel(operation);
}
```

8.6.2. Change Restriction

```
/**
 * Change the expression of a restriction.
 * @param {IPVRestSet} restSet The restriction set containing the restriction.
 * @param {IPVRestriction} restriction The restriction to change.
 * @param {String} expr The new restriction expression.
 */
function changeConstraint(restSet, restriction, expr) {
  var model = restSet.getModelContainer();
  var operation = pure_variants().changeModel(model);

  restSet = operation.changeRestSet(restSet);
  var script = restSet.getFirstRestriction().getFirstScript();
  script.setContent(expr);

  pure_variants().updateModel(operation);
}
```

8.6.3. Delete Restriction

```
/**
 * Deletes all restrictions from the given element.
 * @param {IPVElement} element The element.
 */
function deleteRestrictions(element) {
  var model = element.getModelContainer();
  var operation = pure_variants().changeModel(model);

  element = operation.changeElement(element);
  var restSet = element.getRestrictionSet();
  if (restSet != null) {
    operation.removeRestSet(restSet);
    element.unsetRestrictionID();
  }

  pure_variants().updateModel(operation);
}
```

8.7. Attributes

8.7.1. Get Attributes of Element

```
/**
 * Print the attributes of the given element.
 * @param {IPVElement} element The element.
```

```
 */
function printAttributes(element) {
  var attributes = element.getPropertyList().iterator();
  while (attributes.hasNext()) {
    var attribute = attributes.next();
    console().println(attribute.getName() + " [" + attribute.getType() + "]:");
    var values = attribute.getValueList().iterator();
    while (values.hasNext()) {
      var value = values.next();
      if (value instanceof com.ps.consul.eclipse.core.model.IPVConstant) {
        console().println("  Constant: " + value.getContent());
      }
      else if (value instanceof com.ps.consul.eclipse.core.model.IPVCalculate) {
        var calc = new Calculate(value);
        if (calc.hasScriptList()) {
          console().println("  Calculation: " + calc.getFirstScript().getContent());
        }
      }
    }
  }
}
```

8.7.2. Add Attribute to Element

```
/**
 * Adds an attribute to the given element.
 * @param {IPVElement} element The element.
 * @param {String} name The name of the attribute.
 * @param {String} type The data type of the attribute.
 */
function addAttribute(element, name, type) {
  var model = element.getModelContainer();
  var operation = pure_variants().changeModel(model);

  element = operation.changeElement(element);
  var attribute = new Property();
  attribute.setName(name);
  attribute.setFixed(true);
  attribute.setInheritable(false);
  attribute.setInvisible(false);
  attribute.setType(type);
  element.addProperty(attribute);

  pure_variants().updateModel(operation);
}
```

8.7.3. Add Value to Attribute

```
/**
 * Adds a value to an attribute.
 * @param {IPVElement} element The element.
 * @param {String} name The name of the attribute.
 * @param {String} value The value to add.
 */
function addAttributeValue(element, name, value) {
  var model = element.getModelContainer();
  var operation = pure_variants().changeModel(model);

  element = operation.changeElement(element);
  var attribute = element.getPropertyWithName(name);
  if (attribute != null) {
    var constant = new Constant();
    constant.setType(attribute.getType());
    constant.setContent(value);
    attribute.addConstant(constant);
  }

  pure_variants().updateModel(operation);
}
```

8.7.4. Add Calculation to Attribute

```
/**
 * Adds a calculation to an attribute.
 * @param {IPVElement} element The element.
 * @param {String} name The name of the attribute.
 * @param {String} expr The calculation expression.
 */
function addAttributeCalculation(element, name, expr) {
  var model = element.getModelContainer();
  var operation = pure_variants().changeModel(model);
  var language = ModelConstants().SIMPLE_CONSTRAINT_LANGUAGE;

  element = operation.changeElement(element);
  var attribute = element.getPropertyWithName(name);
  if (attribute != null) {
    var calculate = new Calculate();
    calculate.setType(attribute.getType());
    attribute.addCalculate(calculate);

    var script = new Script();
    script.setType(attribute.getType());
    script.setLanguage(language);
    script.setContent(expr);
    calculate.addScript(script);
  }

  pure_variants().updateModel(operation);
}
```

8.7.5. Restrict Attribute Value

```
/**
 * Restrict the given attribute value.
 * @param {IPVValue} value The attribute value (constant or calculation).
 * @param {String} expr The restriction expression.
 */
function restrictAttributeValue(value, expr) {
  var model = value.getModelContainer();
  var operation = pure_variants().changeModel(model);
  var language = ModelConstants().SIMPLE_CONSTRAINT_LANGUAGE;

  var restrictionSet = value.getRestrictionSet();
  if (restrictionSet == null) {
    restrictionSet = Operations().makeRestriction(expr, language);
    operation.addRestrictionSet(value, restrictionSet);
  }
  else {
    restrictionSet = operation.changeRestSet(restrictionSet);
    Operations().addRestriction(restrictionSet, expr, language);
  }

  pure_variants().updateModel(operation);
}
```

8.7.6. Delete Attribute

```
/**
 * Delete an attribute of an element.
 * @param {IPVModel} model The model.
 * @param {String} ename The name of the element.
 * @param {String} aname The name of the attribute.
 */
function deleteAttribute(model, ename, aname) {
  var operation = pure_variants().changeModel(model);

  var element = model.getElementWithName(ename);
  if (element != null) {
    element = operation.changeElement(element);
```

```
    var attribute = element.getPropertyWithName(aname);
    if (attribute != null) {
      element.remProperty(attribute);
    }
  }

  pure_variants().updateModel(operation);
}
```

8.7.7. Delete Attribute Value

```
/**
 * Delete an attribute value.
 * @param {IPVValue} value The attribute value to delete.
 */
function deleteAttributeValue(value) {
  var property = value.getParentContainer();
  var element = property.getParentContainer();
  var model = element.getModelContainer();
  var operation = pure_variants().changeModel(model);

  element = operation.change(element);
  property = element.getPropertyWithID(property.getID());
  property.remValue(value);

  pure_variants().updateModel(operation);
}
```

9. JavaScript Snippets for pure::variants for Simulink

The following section contains some fragments of JavaScript scripts showing the Simulink functionality. This is only available if the **pure::variants Connector for Simulink** is installed.

9.1. Get Simulink Variability Model Information

9.1.1. Get all Variation Points in Model

```
/**
 * Gets all VariationPoints of the given model.
 * @param {IPVModel} variabilityModel The model
 * @return {Array} result An array of VariationPoints
 */
function getVariationPoints(variabilityModel) {
  var variationpoint_list = SimulinkLogic().getVariationPointList(variabilityModel);
  var idx = variationpoint_list.iterator();
  var result = new Array;
  var variationPoint;

  while (idx.hasNext()) {
    variationPoint = idx.next();
    result.push(variationPoint);
  }

  return result;
}
```

9.1.2. Get Default Assignment of Variation Point

```
/**
 * Gets the DefaultAssignment of a given variation point.
 * @param {IPVElement} variationPoint The variation point
 * @return {IPVElement} variation The default selected Variation
 */
function getDefaultAssignment(variationPoint) {
  var default_variation = SimulinkLogic().getDefaultAssignment(variationPoint);
  return default_variation;
```

```
}
```

9.1.3. Get All Variations of Variation Point

```
/**
 * Gets all variations of a given variation point.
 * @param {IPVElement} variationPoint The VariationPoint
 * @return {Array} variations An Array of Variations
 */
function getVariations(variationPoint) {
  var variation_list = SimulinkLogic().getVariationList(variationPoint);
  var idx = variation_list.iterator();
  var variations = new Array;

  while (idx.hasNext()) {
    variation = idx.next();
    variations.push(variation);
  }

  return variations;
}
```

9.1.4. Get Label of Variation

```
/**
 * Gets the label of a given Variation.
 * @param {IPVElement} variation The Variation
 * @return {String} label The label of the Variation
 */
function getLabel(variation) {
  var label = SimulinkLogic().getLabelOfVariation(variation);
  return label;
}
```

9.1.5. Get Value of Variation

```
/**
 * Gets the value of a given Variation.
 * @param {IPVElement} variation The Variation
 * @return {String} value The value of the Variation
 */
function getLabel(variation) {
  var value = SimulinkLogic().getValueOfVariation(variation);
  return value;
}
```

9.1.6. Get Condition of Variation

```
/**
 * Gets the condition of a given Variation.
 * @param {IPVElement} variation The Variation
 * @return {String} condition The condition of the Variation
 */
function getLabel(variation) {
  var condition = SimulinkLogic().getConditionText(variation);
  return condition;
}
```

9.2. Manipulation of Simulink Variability Model

9.2.1. Simulink Variability Model

Create Simulink Variability Model

```
/**
```

```
 * Creates a VariabilityModel from the given path and name.
 * @param {String} path The path to the model
 * @param {String} name The name of the model
 */
function createVariabilityModel(path, name) {
  SimulinkOperations().createVariabilityModel(path, name);
}
```

Delete Simulink Variability Model

```
/**
 * Deletes a VariabilityModel from the given path.
 * @param {String} path The path to the model
 */
function deleteVariabilityModel(path) {
  pure_variants().deleteModel(path);
}
```

9.2.2. Variation Point

Add Variation Point

```
/**
 * Adds a VariationPoint with the given name and comment to the model.
 * @param {IPVModel} variabilityModel The model
 * @param {String} variationpoint_name
 * @param {String} variationpoint_comment
 * @param {String} parameter_name
 */
function addVariationPoint(variabilityModel, variationpoint_name, variationpoint_comment,
 parameter_name) {
  var operation = pure_variants().changeModel(variabilityModel);

  SimulinkOperations().createVariationPoint(operation, variationpoint_name,
    variationpoint_comment, parameter_name);
  pure_variants().updateModel(operation);
}
```

Rename Variation Point

```
/**
 * Changes the name of a given variationPoint.
 * @param {IPVModel} variabilityModel The model
 * @param {IPVElement} variationPoint The variation point to change
 * @param {String} variationpoint_new_name The new name
 * @param {String} variationpoint_comment The comment
 */
function changeVariationPoint(variabilityModel, variationPoint, variationpoint_new_name,
 variationpoint_comment) {
  var operation = pure_variants().changeModel(variabilityModel);
  var variationpoint_comment = variationPoint.getDesc(variabilityModel.getMimeType());

  SimulinkOperations().changeVariationPoint(operation, variationPoint,
    variationpoint_new_name, variationpoint_comment);
  pure_variants().updateModel(operation);
}
```

Change Default Assignment of Variation Point

```
/**
 * Changes the default value to the given variation.
 * @param {IPVModel} variabilityModel The model
 * @param {IPVElement} variation The new default variation
 */
function changeDefaultVariation(variabilityModel, variation) {
  var operation = pure_variants().changeModel(variabilityModel);
  var variation_label = SimulinkLogic().getLabelOfVariation(variation);
```

```
  var variation_value = SimulinkLogic().getValueOfVariation(variation);
  var is_default_assignment = true;

  SimulinkOperations().changeVariation(operation, variation, variation_label,
    variation_value, is_default_assignment);
  pure_variants().updateModel(operation);
}
```

Delete Variation Point

```
/**
 * Deletes the given VaitioationPoint.
 * @param {IPVModel} variabilityModel The model
 * @param {IPVElement} variationPoint The VariationPoint to delete
 */
function deleteVariabilityPoint(variabilityModel, variationPoint) {
  var operation = pure_variants().changeModel(variabilityModel);

  SimulinkOperations().removeVariationPoint(operation, variationPoint);
  pure_variants().updateModel(operation);
}
```

9.2.3. Variation

Add Variation

```
/**
 * Adds a Variation to the given VariationPoint.
 * @param variationModel The model
 * @param variationPoint The VariationPoint to add the Variation to
 * @param variation_label The label of the Variation
 * @param variation_value The value of the Variation
 * @param variation_is_default_assignemnt Whether or not the new Variation is default
 */
function addVariation(variationModel, variationPoint, variation_label, variation_value,
 variation_is_default_assignemnt) {
  var operation = pure_variants().changeModel(variationModel);
  SimulinkOperations().createVariation(operation, variationPoint,
    variation_label, variation_value, variation_is_default_assignemnt);
  pure_variants().updateModel(operation);
}
```

Rename Variation

```
/**
 * Changes the name of a given variation.
 * @param {IPVModel} variabilityModel The model
 * @param {IPVElement} variation The variation to change
 * @param {String} variation_new_label The new label
 * @param {String} variation_value The value
 */
function changeVariation(variabilityModel, variation, variation_new_label, variation_value) {
  var operation = pure_variants().changeModel(variabilityModel);
  var variation_value = SimulinkLogic().getValueOfVariation(variation);
  var is_default_assignment =
 variation.getDefault().equals(ModelConstants().ELEMENT_DEFAULT_ON);

  SimulinkOperations().changeVariation(operation, variation, variation_new_label,
    variation_value, is_default_assignment);
  pure_variants().updateModel(operation);
}
```

Change Condition of Variation

```
/**
 * Changes the condition of the given Variation.
 * @param {IPVModel} variabilityModel The model
```

```
 * @param {IPVElement} variation The variation to change
 * @param {String} variation_condition The condition
 */
function changeCondition(variabilityModel, variation, variation_condition) {
  var operation = pure_variants().changeModel(variabilityModel);
  var variation_label = SimulinkLogic().getLabelOfVariation(variation);
  var variation_value = SimulinkLogic().getValueOfVariation(variation);
  var is_default_assignment =
 variation.getDefault().equals(ModelConstants().ELEMENT_DEFAULT_ON);
  var variation_comment = variation.getDesc(variabilityModel.getMimeType());

  SimulinkOperations().changeVariation(operation, variation, variation_label,
    variation_value, is_default_assignment, variation_comment,
    variation_condition);
  pure_variants().updateModel(operation);
}
```

Delete Variation

```
/**
 * Deletes the given variation from the model.
 * @param {IPVModel} variabilityModel The model
 * @param {IPVElement} variation The variation to delete
 */
function deleteVariation(variabilityModel, variation) {
  var operation = pure_variants().changeModel(variabilityModel);

  SimulinkOperations().removeVariation(operation, variation);
  pure_variants().updateModel(operation);
}
```