pure::variants - Connector for IBM Engineering Requirements Management - DOORS Manual

Parametric Technology GmbH

Version 7.0.0.685 for pure::variants 7.0

2025

Table of Contents

1. Introduction

pure::variants-Connector for IBM Engineering Requirements Management - DOORS enables DOORS users to manage requirements variability using pure::variants. By coupling pure::variants and DOORS, knowledge about Variability and variants can be formalized, shared, and automatically evaluated. This enables getting answers to questions about valid combinations of requirements in product variants quickly; permits easy monitoring of planned and released product variants at the requirements level, and also permits very efficient production of variant-specific requirements documents. out of the requirements repository.

The manual is available in online help inside the installed product as well as in printable PDF format. Get the PDF here.

1.1. About this Manual

The reader is expected to have basic knowledge about and experiences with both tools, IBM Rational DOORS and pure::variants. Please consult their introductory material before reading this manual.

1.2. Software Requirements

The following software has to be present on the user's machine in order to support the pure::variants - Connector for IBM Engineering Requirements Management - DOORS:

IBM Rational DOORS: IBM Rational DOORS 9.6.0.0 - 9.7.2.9 is required. pure::variants does support the 32 bit and the 64 bit client of IBM Rational DOORS. Compatibility with other IBM Rational DOORS releases is not guaranteed.

The pure::variants - Connector for IBM Engineering Requirements Management - DOORS is an extension for pure::variants and is available on all supported platforms.

DOORS and pure::variants communicate using the TCP protocol or OLE communication (Windows only). It is recommended to use the OLE communication method, as long as pure::variants is running on the same computer as the DOORS client.

For OLE communication, pure::variants and the DOORS client must be running on the same machine.

For using TCP/IP communication, the machine running pure::variants must be able to connect to the port that the DXL Server is using (default 5093), on the machine running the DOORS client. The machine running the DOORS client must be able to connect to the same port (default 5093) on the machine running pure::variants. The port assignment may be changed.Using the preferences page in pure::variants and the shown configuration dialog in the DOORS client while starting the p::v Connector Server.

1.3. Installation

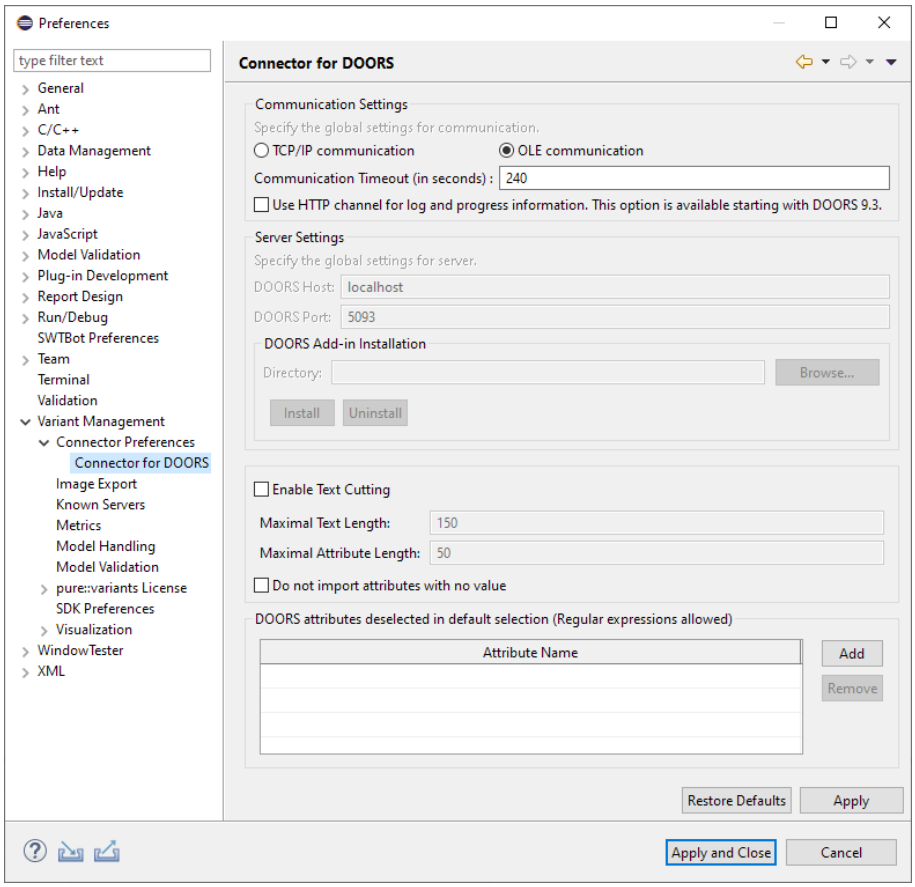
The connector consists of a small IBM Rational DOORS DXL extension, an extension module. for pure::variants, and a pure::variants Integration for defining variability information in DOORS made more user-friendly.

pure::variants

Please consult section **pure::variants Connectors** in the **pure::variants Setup Guide** for detailed information on how to install the connector (menu **Help** -> **Help Contents** and then **pure::variants Setup Guide** -> **pure::variants Connectors**).

Installation of the pure::variants extension model in Eclipse can be easily verified. Simply open the **Preferences** menu in Eclipse. Unfold the items below **Variant Management**. Below the entry **Connector References** an entry **Connector for DOORS** will appear when the installation has been successful (see Figure 1, “Preferences Page”).

Figure 1. Preferences Page



UI Integration Installation

Please consult section **pure::variants Integrations** in the **pure::variants Setup Guide** for detailed information on how to install the connector (menu **Help** -> **Help Contents** and then **pure::variants Setup Guide** -> **pure::variants Integrations**).

After finishing the installation successfully, the DOORS client has to be started with a command line option, which enables the Integration menu within DOORS.

The command line should look similar to this: `doors -a "<Menu installation path>\pure-variants"`. Without this command line option the Integration can not be triggered and so not be used.

On Windows platforms it is also possible to add the directory to the registry key `HKEY_LOCAL_MACHINE\SOFTWARE\Telelogic\DOORS\<DOORS version number>\Config`.

1. Open the Registry editor
2. Browse to DOORS installation in HKEY_LOCAL_MACHINE\SOFTWARE\Telelogic\DOORS\<DOORS version number>\Config
3. Right click config Key to add a new string value
   - Value Name set to **Addins**
   - Value Data set to the path of the pure::variants menu directory.

     e.g. C:\Program Files\Parametric Technology\pv_Enterprise_6.0\com.ps.consul.eclipse.ui.doors.integration\Doors_Menu\pure-variants

With administrative access to the DOORS installation the Add-In can also be installed for all users of this installation using the shared DXL library. See the DOORS Help topic "Configuring DOORS" for more information.

Note

This requires adaptions of the pure::variants Integration menu DXL scripts.

Now the Integration should be available in DOORS. To verify if the installation was successfull, open a DOORS module and select from the menu **pure::variants** the item **Open pure::variants Integration**. If the pure::variants Integration window opens, the Integration was installed correctly.

Note

The pure::variants Integration for IBM Rational DOORS needs several attributes to work properly.

- Two object attributes to store the created restrictions and constraints in. They are called *pvRestriction* and *pvConstraint*, both have to be of type *Text*. The Integration tries to create these attributes automatically if missing.
- For saving the currently opened models, so the Integration can load them automatically, if the Integration opens the next time a module attribute called *pvModels* of type *Text* is needed. This is optional, if you do not want to save the opened models, you do not need to create this attribute. The integration tries to create this attribute. automatically if it is missing.
- For saving user defined substitution character settings a module attribute with name *pv_substitution_settings* is needed. This attribute is optional. The Integration tries to create this attribute automatically if missing.

DOORS OLE Setup

No setup is required. Just make sure to select OLE Communication in the pure::variants preferences for the Connector for DOORS, which is the default.

TCP/IP Communication Only: DOORS DXL AddIn Installation

Note

This step is needed for TCP/IP communication only. If a DOORS client is running on the same machine as pure::variants OLE communication is recommended.

To be able to communicate with DOORS via a TCP/IP connection, pure::variants requires the execution of a DXL extension script. The script is distributed as part of the Connector. Open the preferences of the Connector (see Figure 1, “Preferences Page”). Choose a directory for **DOORS Add-In Installation** and press the **Install** button. This directory has to be added to the list of DOORS addin directories. This can be achieved using the command line switch -projectaddins "<.../DoorsAddins>" or -J "<.../DoorsAddins>" e.g.

```
> doors -J "C:\Program Files\Parametric Technology\pvSyncForDoors\DoorsAddins"
```

with the DOORS Add-In installation directory chosen above.

On Windows platforms it is also possible to add the directory to the registry key `HKLM\Software\Telelogic\Doors\Doors version\projectaddins`.

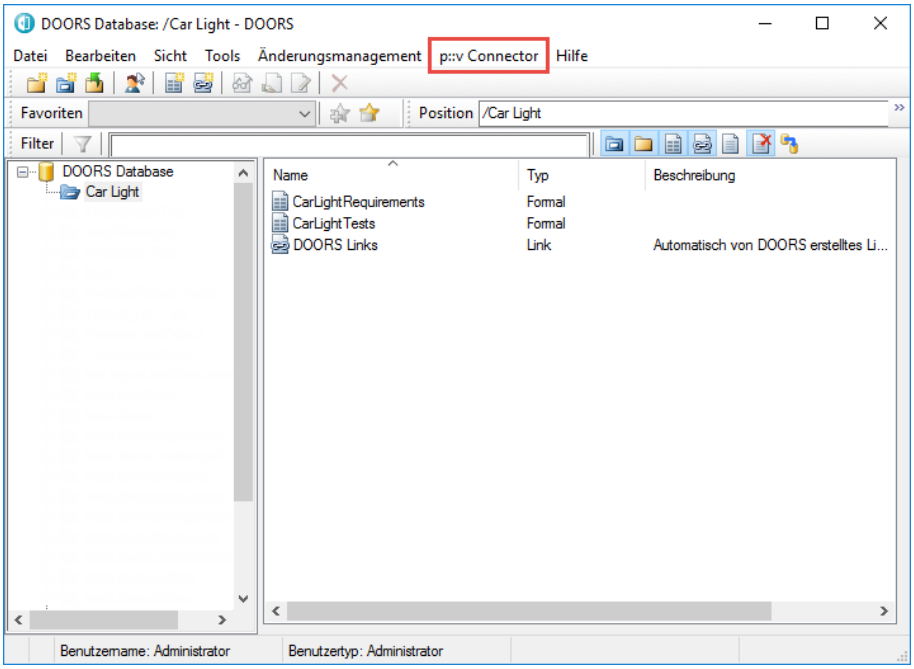
1. Open the Registry editor
2. Browse to DOORS installation in HKEY_LOCAL_MACHINE\SOFTWARE\Telelogic\DOORS\<DOORS version number>\Config
3. Right click config Key to add a new string value
   - Value Name set to **projectaddins**
   - Value Data set to the path of the pure::variants DoorsAddins directory

On Unix platforms the environment variable `DOORSPROJECTADDINS` can be used.

With administrative access to the DOORS installation the Add-In can also be installed for all users of this installation using the shared DXL library. See the DOORS Help topic "Configuring DOORS" for more information.

To verify correct installation of the DXL extension script, open DOORS. There should now be a new menu item **p::v Connector**. This menu will contain an item called **Start p::v Connector Server**. Do not start the server until the pure::variants Connector Eclipse Extension has been installed. See Figure 2, “Initial Screen of DOORS with Successfully Installed pure::variants Connector AddIn”

Figure 2. Initial Screen of DOORS with Successfully Installed pure::variants Connector AddIn



2. Using the Connector

2.1. Starting DOORS Client

Start the DOORS client with the required DOORS user name and password. pure::variants will use the client to access DOORS data, so the user must have sufficient access to the relevant data in the DOORS database.

2.2. TCP/IP Communication Only: Starting the DOORS pure::variants Connector Server

Note

This step is needed for TCP/IP communication only. If a DOORS client is running on the same machine as pure::variants OLE communication is recommended.

Before importing, exporting, or synchronizing data between pure::variants and DOORS, it is necessary to start the pure::variants Connector Server on DOORS. Simply select **Start p::v Connector Server** from **p::v Connector** in the DOORS main window. If this entry is not displayed, verify correct installation of the DOORS part of the pure::variants Connector.

Once the server is started, DOORS will not respond to any user interaction until the server is stopped using the **Shutdown DOORS Script Server** button in the pure::variants tool bar.

2.3. Starting pure::variants

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

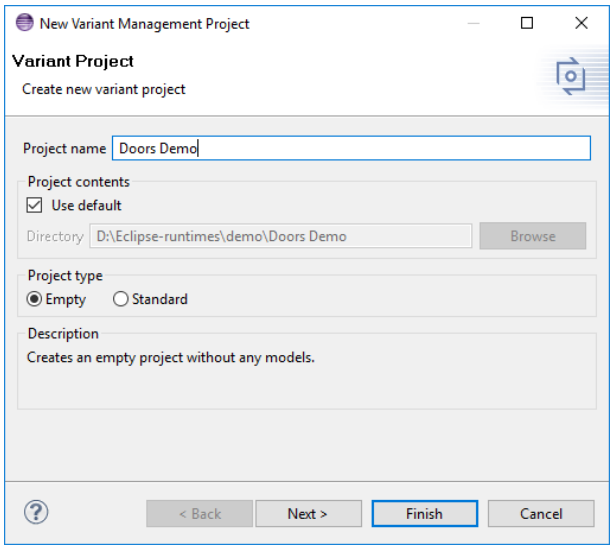
If the **Variant Management** perspective is not already activated, do so by selecting it from **Open Perspective -> Other** in the **Window** menu.

2.4. Creating the Initial Model(s)

The first step is always to create corresponding feature or family models for each relevant DOORS module. These initial feature or family models serve as starting points for using existing variability information or adding variability information to the requirements. The import procedure has to be executed **only once** for each DOORS module. One pure::variants model represents each module.

Before the actual import can be started, a Variant Management project has to be created, where the imported models will be stored. Select **Project** from **New** in the **File** menu. Choose **Variant Projects** below **Variant Management** in the first page of the "New project" wizard. Choose a name for the project and select **Empty** as project type (see Figure 3, “Creating an Empty Variant Management Project for DOORS Module Import”)

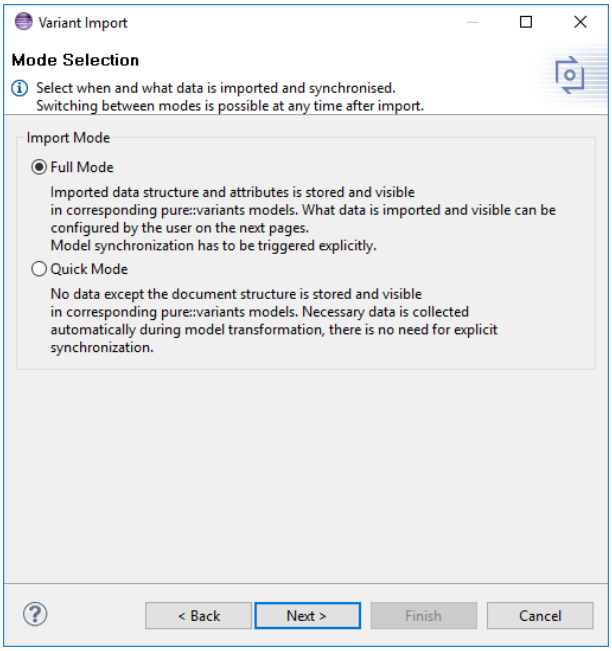
Figure 3. Creating an Empty Variant Management Project for DOORS Module Import



Import is started by selecting the import action either in the context menu of the Project view or with **Import** menu in the **File** menu. Select **Variant Models or Projects** and press **Next**. On the following page select *Import DOORS modules*.

The import wizard appears. With the first page you can decide whether you want to perform a full import of your DOORS Modules (**Full Mode**), or if you want just to import the module header (**Quick Mode**).

Figure 4. The Import Mode Selection Page in the DOORS Import Wizard



By using **Full Mode** all data of the selected DOORS modules can be imported. On the next pages, the user can configure which data will be imported can be configured by the user on the next pages. The imported data is stored and visible in the corresponding pure::variants models. The user can use the imported data to model variability within pure::variants. If using this mode, variability information should be modeled and stored in pure::variants models. The models need to be explicitly synchronized before transforming a variant.

If DOORS is master of the variability informations the **Quick Mode** should be used. The Quick Mode is used to just import information pure::variants needs to access the DOORS modules during synchronization, export, and transformation. The results are empty models. Each module is represented by a pure::variants model containing the linkinformation to the related DOORS module. In **Quick Mode** There is no need to explicitly synchronize the models imported from DOORS; this is done automatically. during transformation.

It is possible to switch between modes by performing a synchronisation and changing the mode in the occurring wizard.

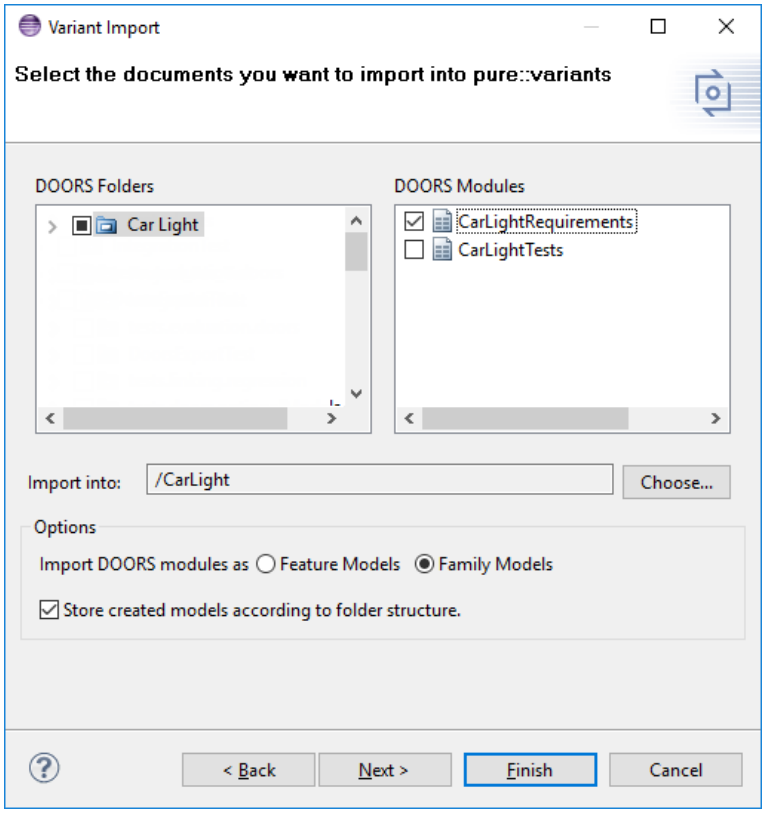
If the TCP/IP communication is used the wizard requests, on the next page, information about the host the DOORS p::v Connector server is running on. If the DOORS p::v Connector server and pure::variants are running on the same computer enter `localhost` here.

On the next page (see Figure 5, “The Module Selection Page in the DOORS Import Wizard”) the complete project and folder structure of the DOORS repository is shown. Navigate to the folders containing the modules of interest and select the check boxes on the right side. Selecting a check box on the left side marks all modules inside this folder and its sub-folders for import. Make sure that the import target location given next to **Import into:** is correct. The location can be changed using the **Choose** button.

If **Store created models according to DOORS folder structure** is checked, the folder structure created in pure::variants will resemble the DOORS folder structure. Projects are treated as normal folders in this case. If unchecked, all modules are stored directly in the selected target location. *Use this only when all Selected modules have unique names.*

Choose whether you like to import modules as feature or family model, by using the corresponding radio buttons.

Figure 5. The Module Selection Page in the DOORS Import Wizard



On the next page you can adjust several options for your module import.

If using Quick Mode you can select option **Consider Links**, if you want links to be considered during transformation.

Option **Import baselines** provides, if checked, possibility to use a specific baseline during import of modules. If checked a baseline can be chosen in the combobox located directly behind the option button. Only baselines are shown, which are available in all selected modules. If this option is not checked baseline**current** will be used for importing the modules. The imported baseline can be used during transformation, to copy the module from the specified baseline instead of using baseline **current**.

Click Finish to perform the import.

Check **Import links** if you like to import links from DOORS. If this option is unchecked no links are imported. That means even the variation link mechanism is not working, because no links are imported. Links can either be imported as separate relations per link, or as separate targets on the same relation.

Note

The semantic is different, if one relation with multiple targets are used, or if separate relations are used. Links of the same type should be merged in to the same relation.

Option **Import baselines** provides, if checked, the possibility to use a specific baseline during import of modules. If checked a baseline can be chosen in the combobox located directly behind the option button. Only baselines are shown, which are available in all selected modules. If this option is not checked baseline**current** will be used for importing the modules.

The following two options are synchronization options and have no effect during import, Both options are also changeable during the synchronization process.

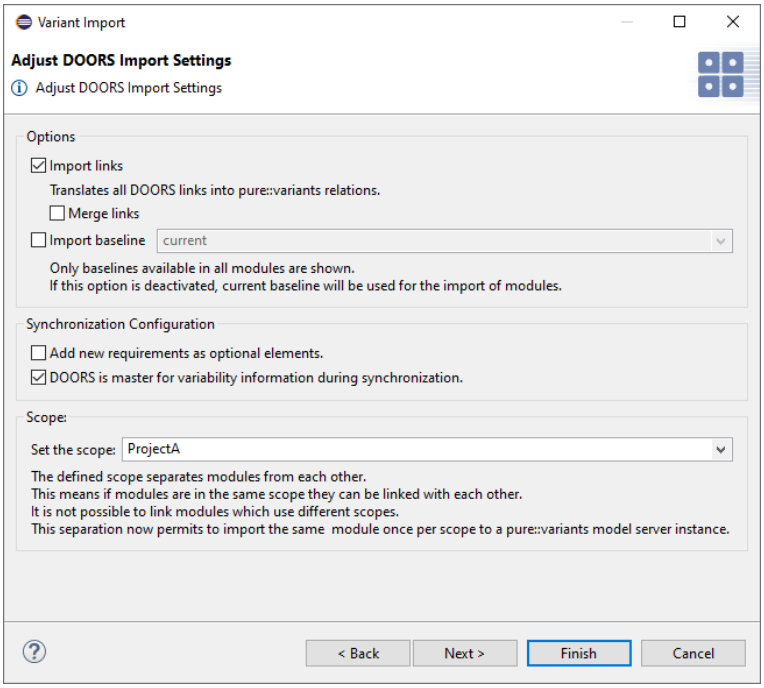
Add new requirements as optional elements provides the availability to add newly created requirements as optional during synchronization. This prevents you from accidentally changing the variant by adding a mandatory

element during synchronization. If a variation-type is specified with the DOORS attribute *pvVariationType* the specified variation-type is used, and this option has no effect.

If option **DOORS is master for variability information during sync** is checked, information defined in used Import Rule Sets is used during update. If this option is not checked changes in elements created by rule set will not be considered during update. For more information see Section 2.5, “Using Variability Information from DOORS Modules”.

Since 5.0.9 a scope can be defined during import of an Doors module. The defined scope separates modules from each other. This means if modules are in the same scope they can be linked with each other. It is not possible to link modules which use different scopes. This separation now allows the user to import modules more than once to a pure::variants model server instance. The import can be performed on different and the same projects, located on the same pure::variants model server instance. The scope is a simple string with no limitations. It is still possible to work without scope if no scope is given during the import. The scope is not changeable during update since the whole ID calculation is depending on the scope.

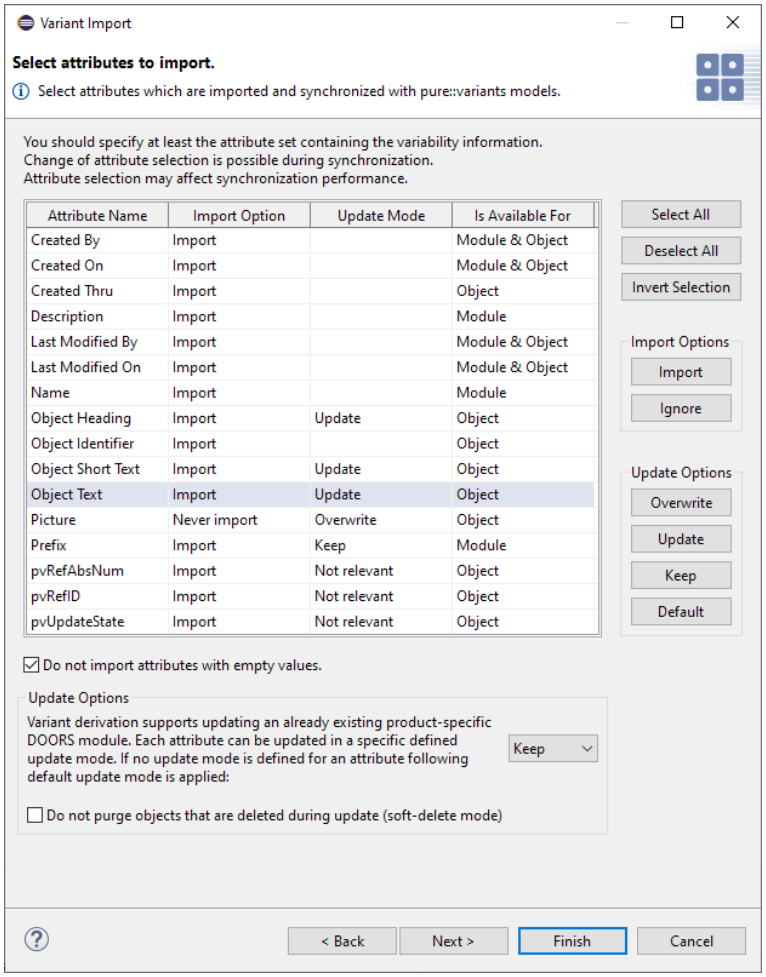
Figure 6. Select the Baseline of the DOORS Module to Import



The next page of the import dialog shows a list of DOORS attributes, the imported module has. The **Import Option** defines, whether the respective attribute should be imported or whether it should be ignored. This option can be switched using the buttons**Import** and **Ignored**. Only imported attributes can be used in constraints or restrictions. If an attribute cannot be imported, like *Picture*, it is marked with **Never import**. The **Update Mode** defines the handling of attribute values during update. Further information can be found in Section 3.5, “DOORS Transformation with Update Support”.

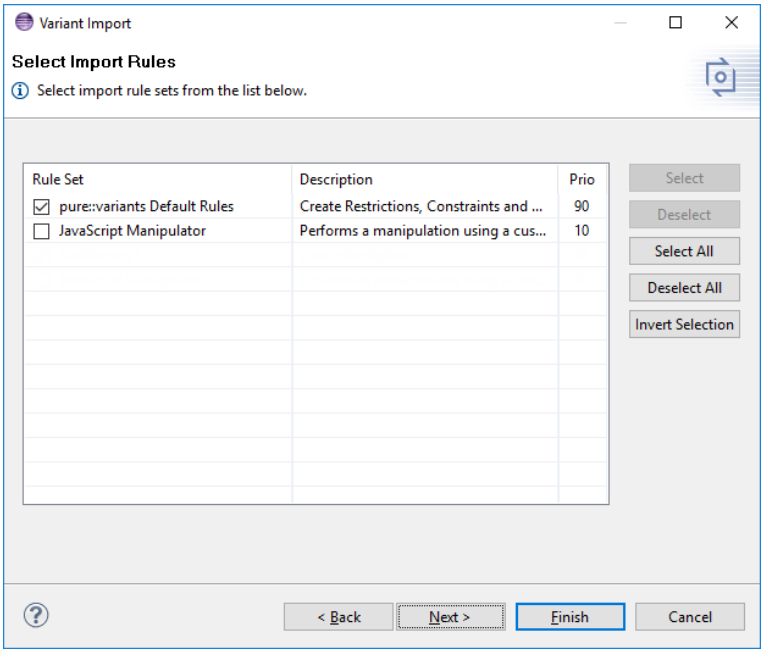
Do not import attributes with empty values ignores attributes with empty values during import, if checked.

Figure 7. Set of DOORS Attributes to Import



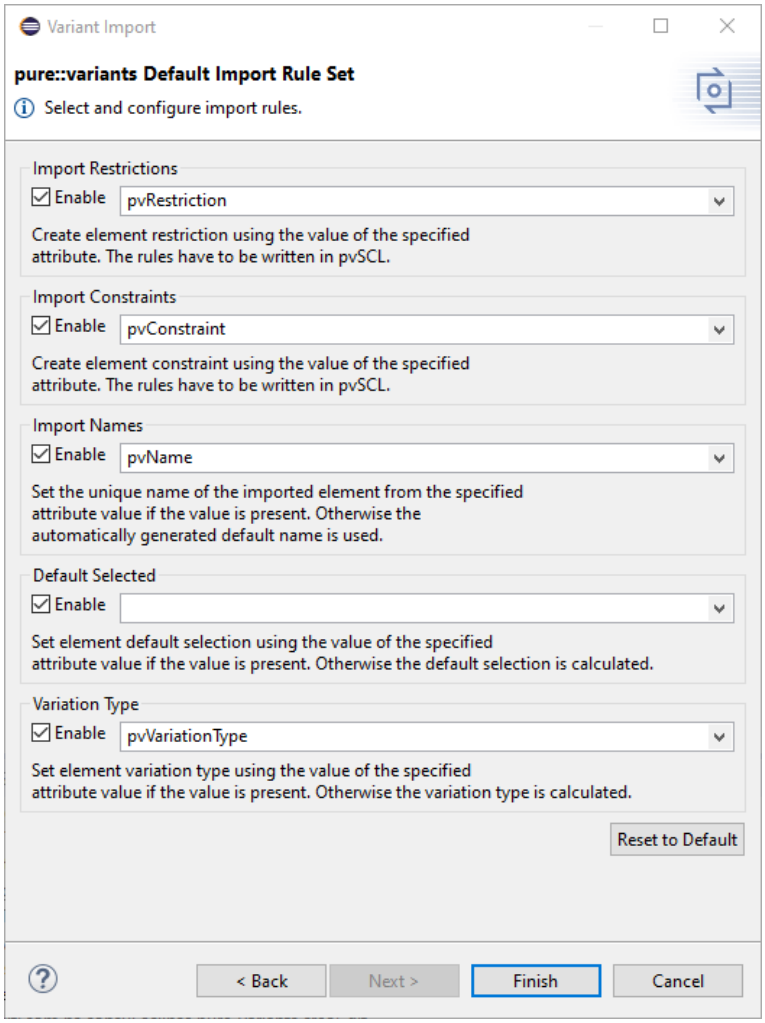
Pressing Next button brings up the Import Rules page. On this page you can select sets of Import Rules, which will be used to manipulate the resulting model after import. Import Rule Sets can be used to create specific pure::variants model elements like relations or restrictions from DOORS module information.

Figure 8. Select Set of Import Rule Sets to Use during Import



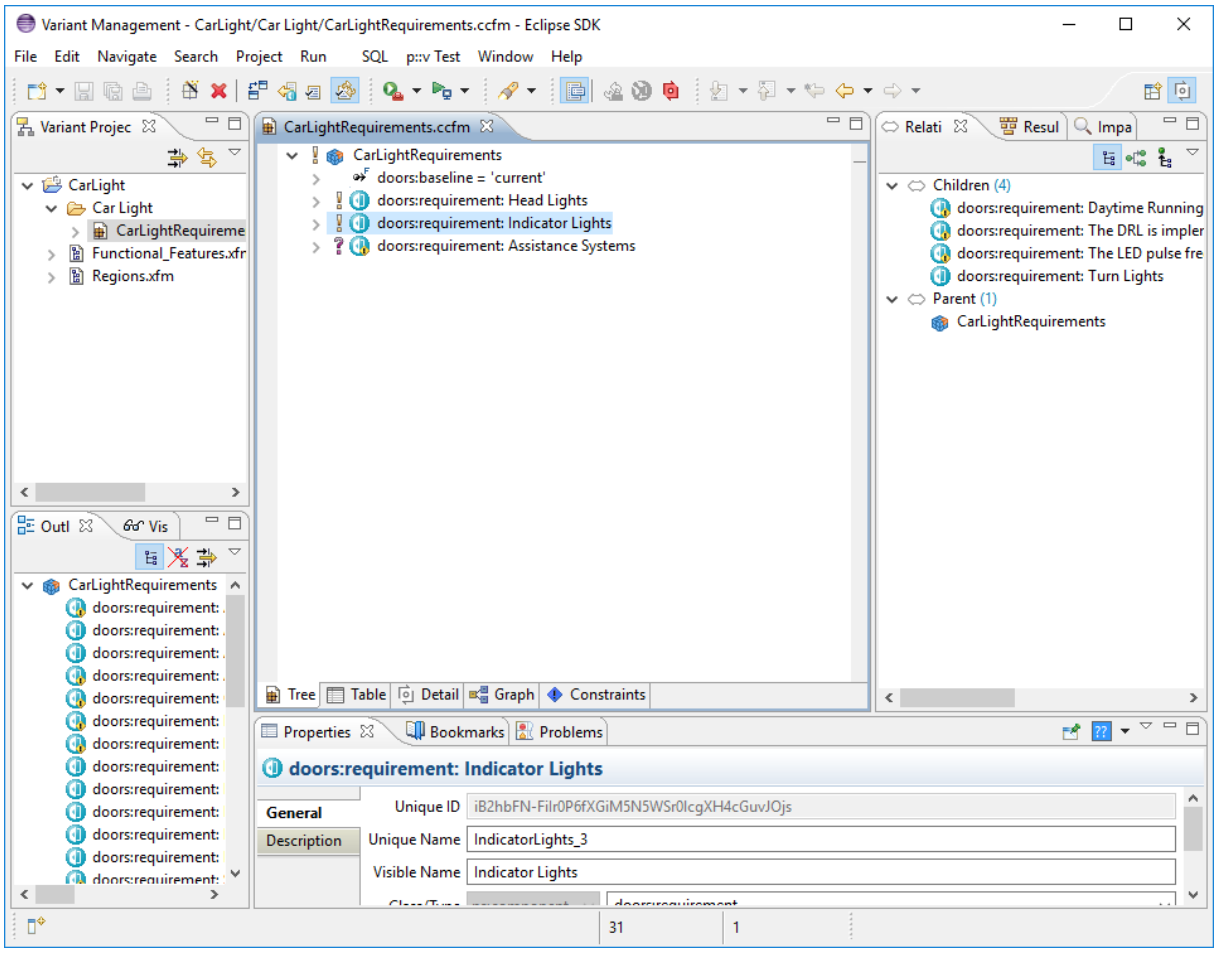
The last pages are showing the settings of the selected Import Rule Sets. For the pure::variants Default Import Rule Set you can choose which attribute value will be used for creating restrictions and constraints and setting the default selection on elements and which attribute value will be used as unique name for elements. Also you can disable the creation of restrictions, constraints and using specific attribute for unique name, default selection and variation type.

Figure 9. Settings for the pure::variants Default Import Rule Set



The import result will be visible in the Variant Project view. If nothing shows up, use the item **Refresh** in the project's context menu (right mouse click) or press **F5** after selecting the project in the view. Each module is now represented by one pure::variants model. Models can be opened by double-clicking on them or selecting **Open** in the context menu. Figure 10, “Result of Initial Import of a DOORS Module” shows the typical layout of a DOORS module after import.

Figure 10. Result of Initial Import of a DOORS Module



All requirements are represented as features (if imported as feature model) or as components (if imported as family model). The elements follow the hierarchical structure as found in the original DOORS module.

If element attributes are not visible in your feature model view, you should enable attribute display via the context menu (**Tree Layout** and select **Attributes**) Attribute values can also be restricted in length during import if they are not relevant for the pure::variants model. See Section 3.3, “Customizing Import of DOORS Attributes” for more information on this.

All requirement elements are imported as *mandatory* or default selected option (if a restriction was defined in DOORS) unless variability information was provided in the DOORS modules. This is explained in more detail in Section 3.1, “Adding Variability Information in DOORS”.

Note: Unicode control characters (characters between 0 and 31 and character 127) are invalid in XML except for characters 9 (tabulator), 10 (line feed) and 13 (carriage return). Invalid characters are replaced with white spaces during import. This has no effect of the transformation result, because in transformation the original content is copied directly from the source module in DOORS.

Table 1. Overview of Representation of DOORS Entities in pure::variants

DOORS Entity	pure::variants Representation
Project	folder in project
Folder	folder in project
Formal Module	feature model or family model
Object	feature in feature model / component in family model

Object Heading, Object Short text, Object Text	elements visible name (first non-empty value is used, long names are shortened)
Object attribute `pvRestriction`	element restriction in pvSCL language
Object attribute `pvConstraint`	element constraint in pvSCL language
Object attribute `pvVariationType`	element variation type. Valid input values are either `mandatory`, `or`, `optional`, `alternative`, `ps:mandatory`, `ps:or`, `ps:optional`, or `ps:alternative`.
Object attribute `pvName`	element unique name. Instead of generating the unique name from DOORS object properties, pure::variants will use the defined name as unique name. The name should be a valid unique name (see pure::variants User Guide for more information). If the name is not valid, pure::variants will generate a valid name from it. Uniqueness is not enforced during import, but later shown as model problem in the model editors.
Object attribute `pvDefaultSelected`	element default selection state. Valid input values are either `off` or `on`. Case is irrelevant, so `OFF` or `Off` are also valid input values.
Object id (absolute number)	element attribute `AbsoluteNumber`
Object attribute	element attributes with type ps:integer or ps:string
Link with attribute `pvRelation`	element relations which use the type name specified in the DOORS attribute `pvRelation`, attached to start of link.
Link	element relations of type `doors:link`, attached to start of link.
Access rights	not imported

2.5. Using Variability Information from DOORS Modules

During the synchronization, pure::variants can optionally use some information present in the DOORS modules to create the pure::variants model representation. If the option "DOORS is master for variability information during sync" was enabled, the information is used not only during initial import but also when synchronizing the model. In effect, the DOORS module becomes the master for variability related information. During synchronization all information including element hierarchy, restrictions and constraints is compared and if different from the information stored in the pure::variants model, shown a mergeable difference.

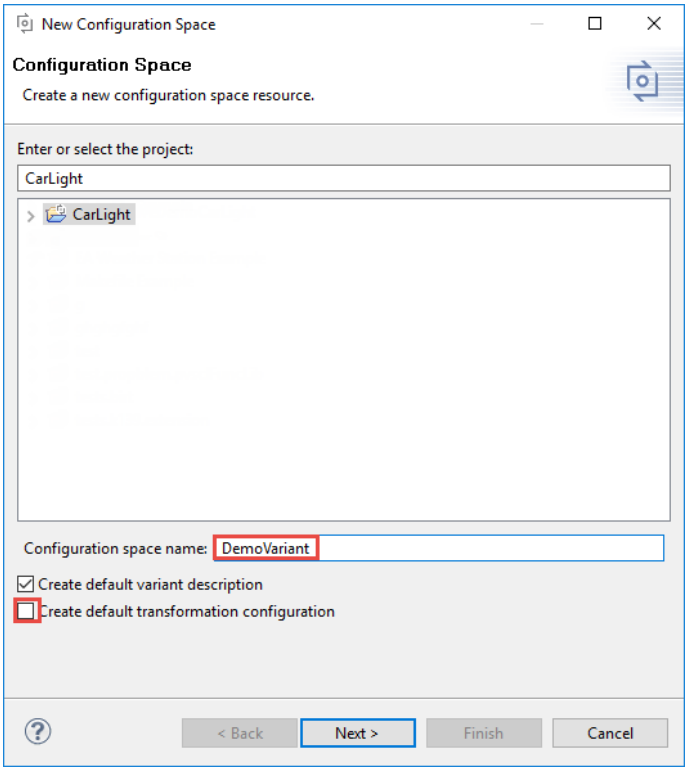
Which and how variability information can be represented in DOORS modules is explained in more detail in Section 3.1, “Adding Variability Information in DOORS”.

2.6. Defining a Variant

The next step is the definition of the actual variants of interest. Since the variability model usually permits the definition of a very large number of variants, pure::variants keeps track only of those variants which are of interest for the users. Typically this number is much smaller than the number of possible variants.

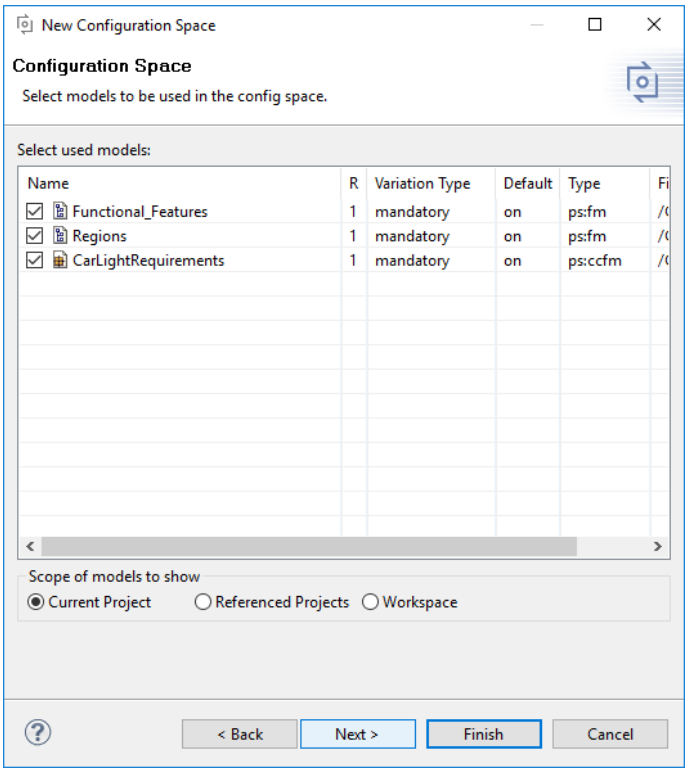
Variants are stored as separate entities called *Variant Description Models* (VDM). A VDM always belongs to a specific *Configuration Space*. Thus before defining variants, a configuration space has to be created. Select the project containing the imported models in the Variant Projects view and open the context menu. Below the item **New** select **Configuration Space**. A wizard is opened. On the first page (Figure 11, “The Configuration Space Wizard, page 1”), enter a name for the configuration space. The name has to follow strict rules (no spaces, no special characters). Uncheck the box before**Create standard transformation**, since for pure requirements models the standard transformation does not provide any relevant functionality (See the pure::variants User Manual for more information on transformations).

Figure 11. The Configuration Space Wizard, page 1



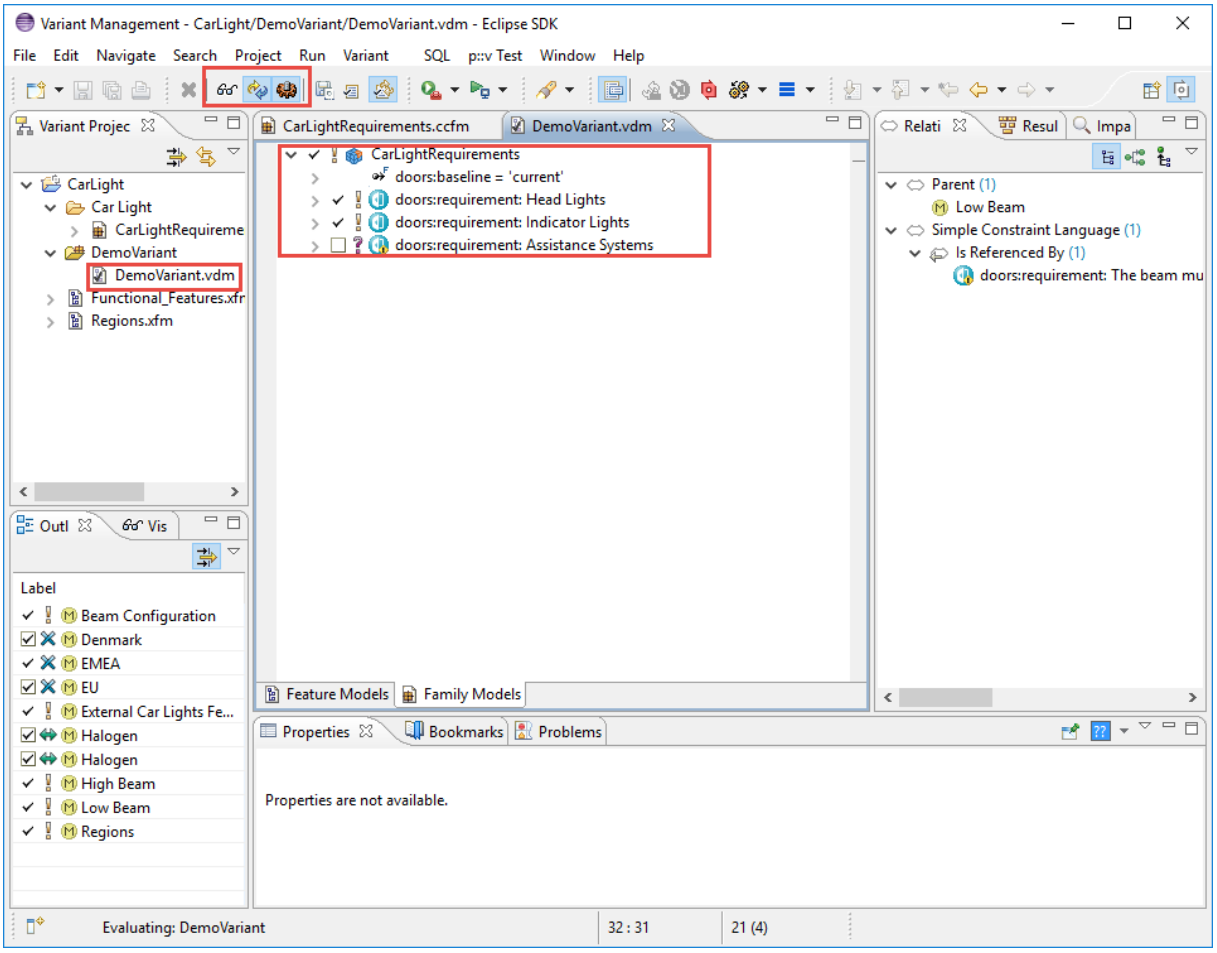
The next page is used to specify which feature models are to be included in this configuration space. Select here all models that represent the DOORS modules of interest. In the example below just one model is selected. Now press the **Finish** button.

Figure 12. The Configuration Space Wizard Model Selection Page



The resulting project structure is shown in Figure 13, “Initial Configuration Space Structure”. The `DemoVariants.vdm` is created and immediately opened. It resembles the structure of the previously defined model(s), but has a checkbox in front of each element to permit the user to select elements for this variant by clicking on it. The buttons marked in the toolbar control the evaluation of configurations. The left-most button ( ) initiates a manual check of the variant configuration. The middle button ( ) toggles between manually checking and automatic checking after changes to the VDM. Finally the right-most button ( ) toggles the auto-resolver on or off. The auto-resolver provides automatic resolution of configuration problems where possible.

Figure 13. Initial Configuration Space Structure



Tip: For small to medium sized models (up to several thousand element, depending on the speed of the processor), it is convenient to turn on both autochecking and autoresolving by clicking on the respective toolbar buttons.

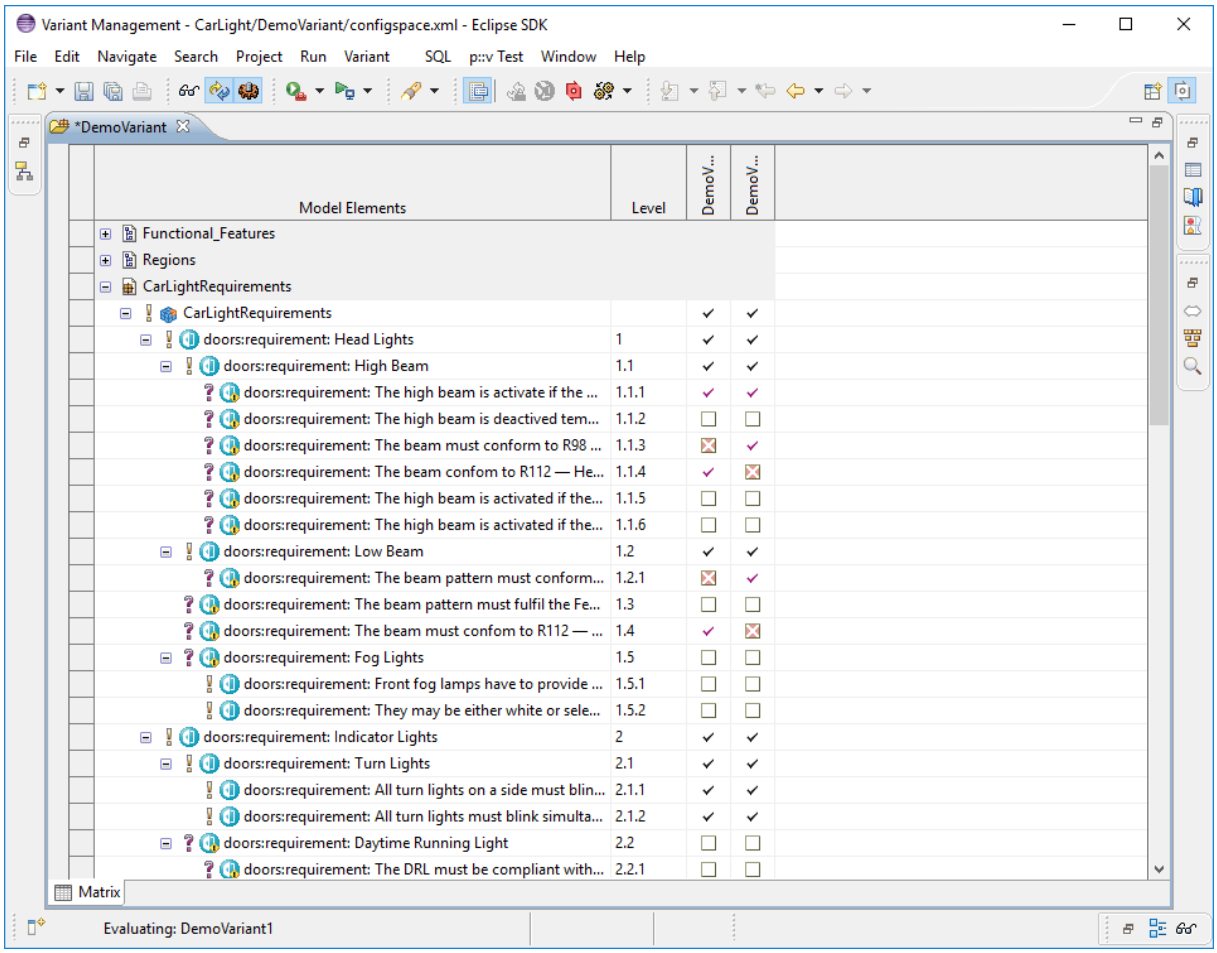
Problems may be indicated by pure::variants during the selection of requirement elements. There are several places where problems are shown. Firstly, the Problems view (usually located in the lower right part of the pure::variants perspective) lists all problems such as incompatible elements or a missing selection from *alternative* elements. In addition, problems are shown in the model editor directly in front of the element causing the problem. A tooltip (which can be seen by moving the mouse over the icon) explains the problem, and the context menu for the problem (right mouse button) provides possible fixes for the problem. E.g. for conflicting elements the fix is to deselect either one or the other element.

Each variant can be represented in its own VDM. To create a new VDM, either select **Clone** from the context menu (in Variant Project view) of an existing variant or use **New->Variant Model** in the context menu of the configuration space. When a valid variant is configured, it can be stored and exported to DOORS. The next section explains this in detail.

Variants may be compared at the elements level by using the matrix editor (see Figure 14, “Matrix Editor for Comparing Variants”). This editor is activated by double-clicking on the enclosing configuration space icon. The

Table Layout item in the context menu can be used to customize the list of variants to compare and the**Show Elements...** and **Filter** items in the same menu can be used to select elements of interest.

Figure 14. Matrix Editor for Comparing Variants



2.7. Exporting a Variant to DOORS

Variants stored in a variant description model can be made available in DOORS. The Connector currently supports three different ways of representing variants: *link-based* , *module-based* and *attribute-based*.

Link-Based Variant Representation

In the link-based representation each variant is represented by a single DOORS requirement object with outgoing links to all requirements included in the variant. The requirements to include are defined by the list of requirement element selected in the respective variant description model. When re-exporting an existing variant, the requirement representing the variant is updated.

Module-Based Variant Representation

The module-based representation creates variant-specific copies of each module in a designated DOORS folder, using the folder structure of the original DOORS modules. Only those modules that are included in the configuration space as pure::variants models are considered. In contrast to the link-based representation requirement objects may also be included in the variant-specific modules even if they have not been explicitly selected in the variant description model. This is the case if the requirement object has been selected by its parent object but has not been selected in the variant description model. These requirement objects must be included in order to keep the document structure intact.

Links between modules included in the variant are kept intact in this representation, as well as outgoing links from these modules to other modules not part of the variant. Incoming links from these "external" modules are not copied into the variant-specific copy of the modules.

The copied links between the modules are stored in link sets, these link sets between the copied modules are stored in link modules. Each link set has one source module and one target module. If the link set is stored in a link module, which is located in the same project as the source module of this link set, the link set is copied to a link module, which is located at the same location relative to the copy of the source module. If this link module is not existing by then, it is created. All link module attributes are preserved.

There are 4 different link module types (Many-to-Many, Many-to-One, One-to-One and One-to-Many). During copying the modules in DOORS the different types are handled equally.

pure::variants recreates the folder structure from the import project in the output folder per default. The module paths are recreated relative to the project the modules are located in. For nested projects, the path is recreated relatively to the first project only.

In case this is not sufficient customized module paths can be used or ff filtering of the requirements in a DOORS module is necessary, which is not covered by the variability information, the attribute *doors:filterView* can be used.

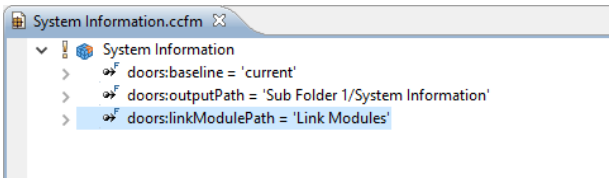
Please have a look to section the section called “Family model attributes used to customize transformation behavior” to find more information about the customization properties.

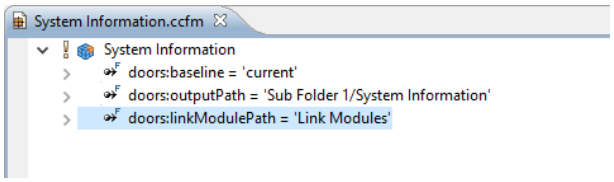
Family model attributes used to customize transformation behavior

The following attributes can be added to the root element of a family model, which represent a door module. Adding those attributes change the transformation behavior and thus can be used to customize the Module-Based Variant Representation.

All attributes are optional and can be freely combined.

Table 2. Transformation customization attributes

Attribute Name	Behavior
doors:outputPath	Defines the target path of the copied module. The path is relative to the *VariantRoot* defined in the transformation configuration. The entered path needs to contain the module name. If the module shall be renamed during transformation, just specify the new module name instead of the original one here. So the resulting path is: <VariantRoot>/<content of the attribute> For the update mode of the doors transformation this attribute defines the path of the working copy module and the path of the corresponding modules in the latest and ancester module. Example:  In the example above the module "System Information" will be copied to the folder <VariantRoot>/SubFolder 1/System Information.
doors:linkModulePath	Defines the path for all link modules which are used for links starting in the module represented by the family model defining the attribute.

Attribute Name	Behavior
	Example:  All link modules with links starting in this module will be created in the folder "<variant root>/Link Modules". **Note** Renaming of link modules is not supported
doors:filterView	This attribute is used to define a filter view. The filter view is a DOORS module view, which has to exist in the transformed DOORS module. If the filter view is defined, all requirements not visible in the view are removed before the processing of the copied module starts. **Note** If a parent requirement of an visible element is filtered by the view, then the parent requirement is not removed to maintain the document structure. This applies for all parents until the module root. The environment variable *PV_DOORS_PRE_FILTERING_VIEW_NAME* can be used to define the filter view as well. If the attribute*doors:filterView* is not set, but the environment variable is set, the value of the environment variable is used. If the*doors:filterView* is set it overwrites the environment variable.
doors:repositoryPath	Considered in update mode only. The reference repository path can be customized for each module. Therefore, in the corresponding family model, this attribute has to be added to the root element. The defined repository path is relative to the *VariantRoot* defined in the transformation configuration. So the location will be <VariantRoot>/<content of the attribute>
keepconstraints	This attribute specifies whether the variability informations shall be removed from the copied module. If this attribute exists and is set to false then the informations are removed. In all other cases the informations remain in the copied modules.

Attribute-Based Variant Representation

In the attribute-based representation there are two modi of representing a variant via attributes within the source DOORS module.

The first modus is the *Variant Matrix* modus, which creates a single attribute for each exported variant. The attribute is named with the name of the variant and a user defined prefix. The value for these attributes is *True* if the corresponding requirement is part of the variant, *False* otherwise.

The second modus is the *Variant Enumeration* modus. This modus does only use one attribute. If a requirement is part of a variant, the name of this variant is added to the value. The name of this attribute can be user defined. Default is *pvVariants*. This transformation uses a specific attribute type filled with the enumeration values used in the attribute. Since 5.0.9 the transformation uses one specific attribute type for each variant enumeration attribute.

The transformation parameter *cleanup* removed unused enumeration values and also switches the old behavior to the new behavior.

Note

This approach is only available as transformation, it can not be performed as described in the next section.

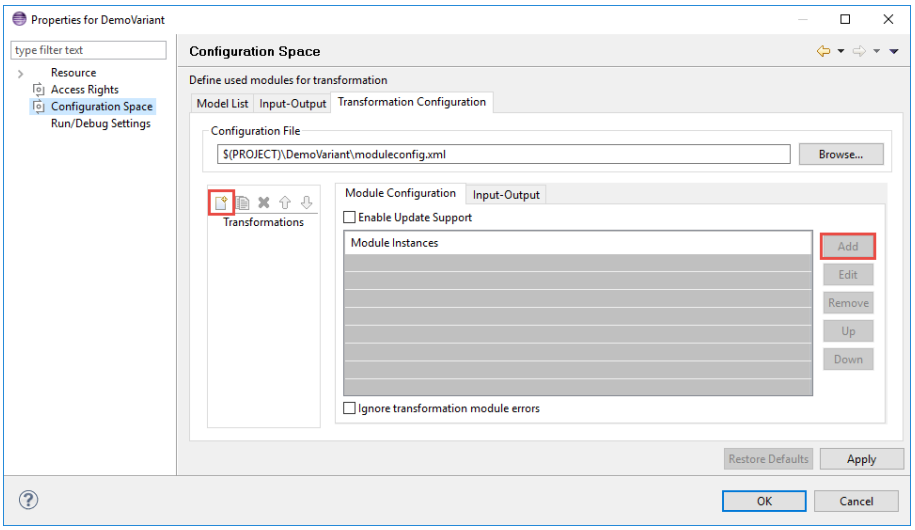
Using Transformation to Export Variant

To export variants to DOORS, using a transformation, first a *Transformation Configuration* has to be created. To create a Transformation Configuration click on **Transformation** button in the tool bar (see Figure 15, “Transform Model”) and choose *Open Transformation Config Dialog...*

Figure 15. Transform Model

The configuration space property dialog opens and the *Transformation Configuration* tab is shown. Next step is to add a new *Module Configuration, by clicking the marked tool bar item* (see Figure 16, “Transformation Configuration”). Now add a new Module to the Module Configuration, using the **Add** button.

Figure 16. Transformation Configuration



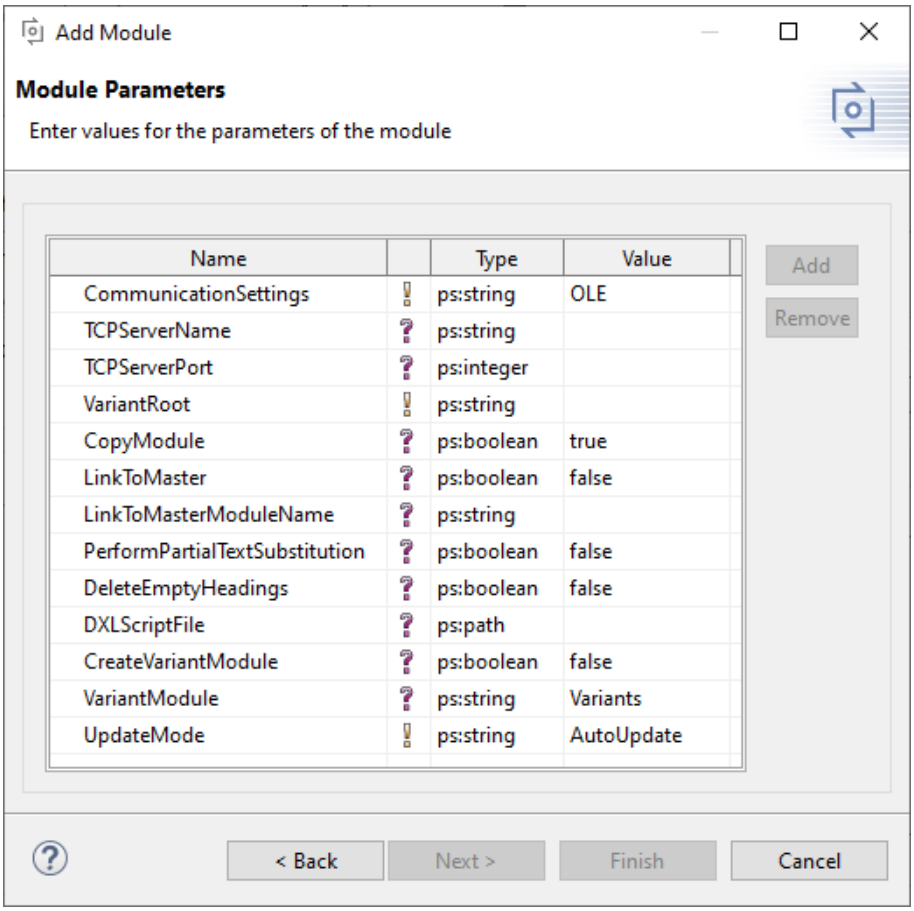
A new Dialog comes up, for the link-based and module-based approach (see the section called “Link-Based Variant Representation”) choose IBM Rational DOORS Module and enter a name.

The next page shows some parameters:

- CommunicationSettings - choose between communicating with DOORS over TCP/IP or OLE connection.
- TCPServerName - The server address. (Only necessary, if TCP/IP connection is used.)
- TCPServerPort - The server port. (Only necessary, if TCP/IP connection is used.)
- VariantRoot - The folder or project the variant should be exported to.
- CopyModule - If **true** is selected, a copy of the modules will be placed inside the selected folder during export.
- LinkToMaster - If **true** is selected, the modules in the export folder are linked to the original modules.

- LinkToMasterModuleName - The defined link module name will be used to create a link module for storing the trace links to the source requirements. If this is omitted "Variant Object Links" will be used a link module name.
- PerformPartialTextSubstitution - If **true** is selected, the partial text substitution is performed.
- DeleteEmptyHeadings - If **true** is selected, empty headings are removed after the transformation is performed. An empty heading is an object, which has the *Object Heading* attribute set and no more child objects are left after transformation.
- DXLScriptFile - If defined, the file path of a DXL script that will be executed during transformation (see Section 3.6, “Perform Custom DXL Script during transformation” for more information). The file has to be encoded in UTF-8.
- CreateVariantModule - If **true** is selected, the variant module is created.
- VariantModule - The name of the module used for the link-based approach. If this is omitted, the name Variants will be used as module name. To prevent name conflicts with created variant folders, this name should not be a name of a variant.
- UpdateMode - Defines the update mode if update support is enabled. **AutoUpdate** (default) means that all modules are updated automatically during the transformation. **ManualUpdate** means that the update has to be triggered in each module manually by the user. **OnlyUpdateLinks** means that it updates the links of all transformed modules. **OnlyUpdateObjects** means that it updates the objects of all transformed modules. Further information can be found in Section 3.5, “DOORS Transformation with Update Support”.

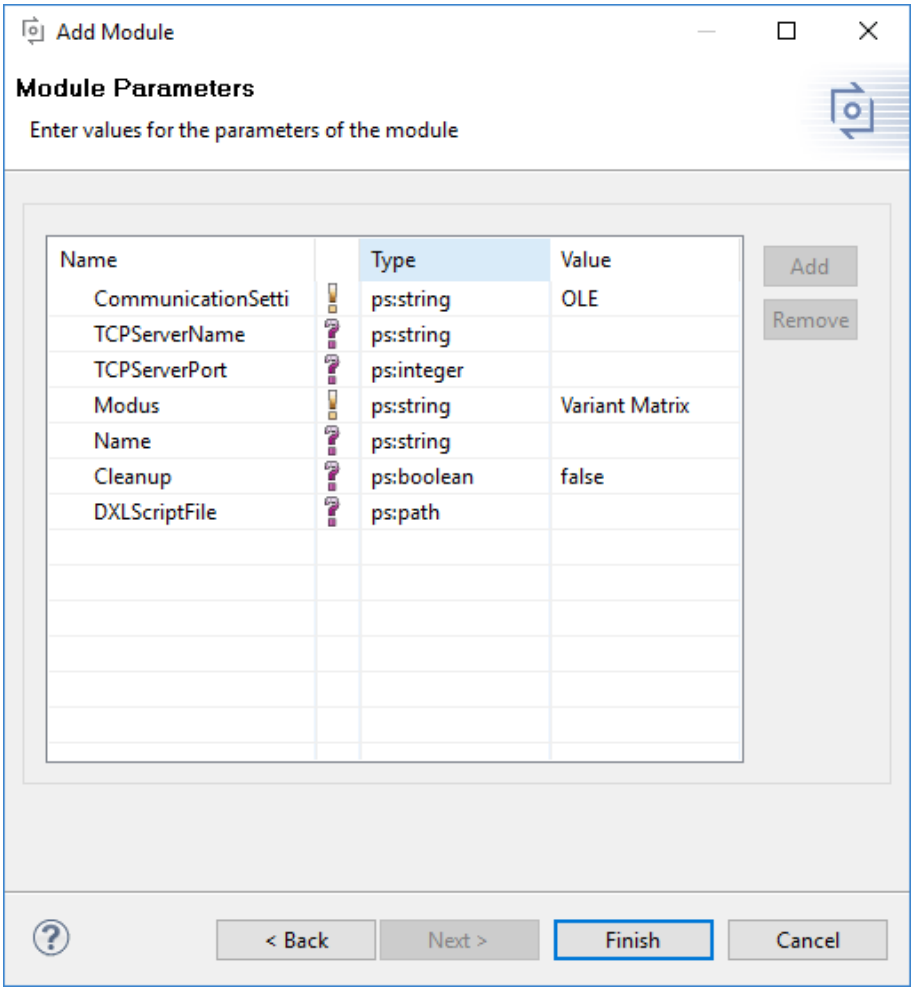
Figure 17. Module Parameter Page



If the attribute-based approach should be used choose *IBM Rational DOORS Configuration Exporter*. On the next page parameter for configuring the transformation module are shown.

- CommunicationSettings - choose between communicating with DOORS over TCP/IP or OLE connection.
- TCPServerName - The server address. (Only necessary, if TCP/IP connection is used.)
- TCPServerPort - The server port. (Only necessary, if TCP/IP connection is used.)
- Modus - Chose between the two modi *Variant Matrix and Variant Enumeration*.
- Name - Specifies the name for the enumeration attribute or the prefix for the matrix attribute names. If not set standard name (*pvVariants*) or no prefix is used.
- Cleanup - If **true** is selected, all existing variant attributes are removed before exporting the current variant.
- DXLScriptFile - If defined, the file path of a DXL script that will be executed during transformation (see Section 3.6, “Perform Custom DXL Script during transformation” for more information). The file has to be encoded in UTF-8.

Figure 18. Module Parameter Page



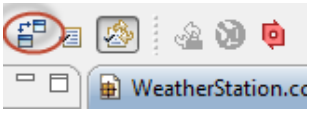
After finishing the dialogs the transformation can simply be used by clicking on the Transformation button (see Figure 15, “Transform Model”) in the tool bar and choosing the transformation in the pull down menu.

2.8. Updating Models from DOORS

Since there is no life connection between the DOORS database and pure::variants, it is necessary to update the pure::variants models with information from DOORS whenever relevant changes have been made. To facilitate the synchronization, pure::variants provides a **Synchronize** action. To start the update, open the model represent-

ing the DOORS module and press the **Synchronize** button in the tool bar (see Figure 19, “Synchronize model”). pure::variants will connect to DOORS and present the so called Compare Editor for pure::variants models (seeFigure 20, “Model Update from DOORS in Compare Editor”).

Figure 19. Synchronize model

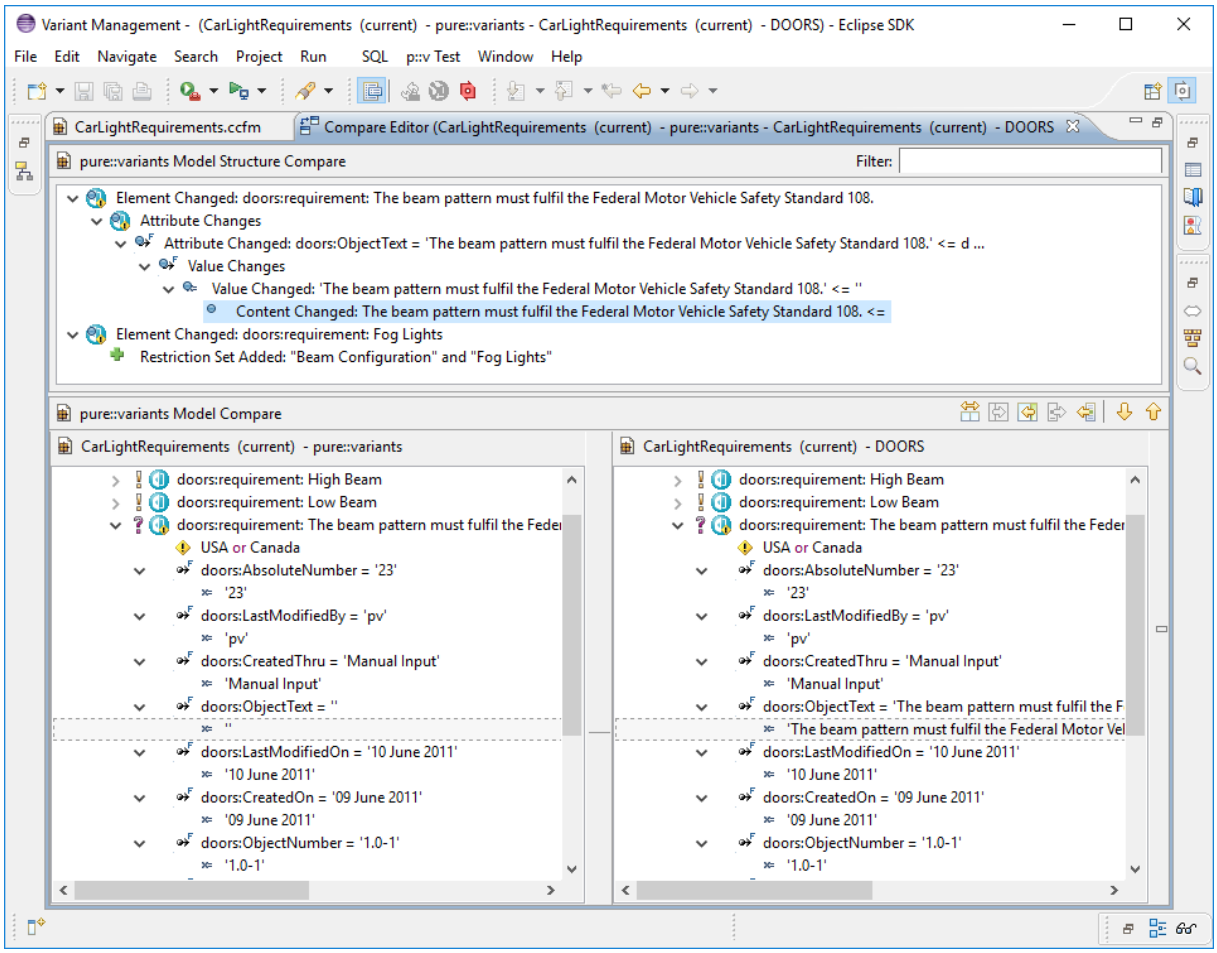


The compare editor is used throughout pure::variants to compare model versions but in this case is used to compare the DOORS data (displayed in the lower right side) with the current pure::variants model (lower left side). All changes are listed as separate items in the upper part of the editor, ordered by the affected elements. Selecting an item in this list highlights the respective change in both models. In the example, the changed attribute values are marked with boxes and connected with their respective counterparts in the other model.

Note

If the option **Add new requirements as optional elements** is enabled in the synchronize wizard there is one **important** difference between model import and model update. Requirements with no explicit variability type specified will become *mandatory* elements when imported and *optional* elements (default selected off) when they are introduced as part of the model update. The reason is straightforward. Adding new elements as optional elements during update retains full compatibility with existing variants, since optional elements are not automatically added to a selection, if they are default selected off. Thus a new evaluation of an existing variant description model yields the same selection.

Figure 20. Model Update from DOORS in Compare Editor



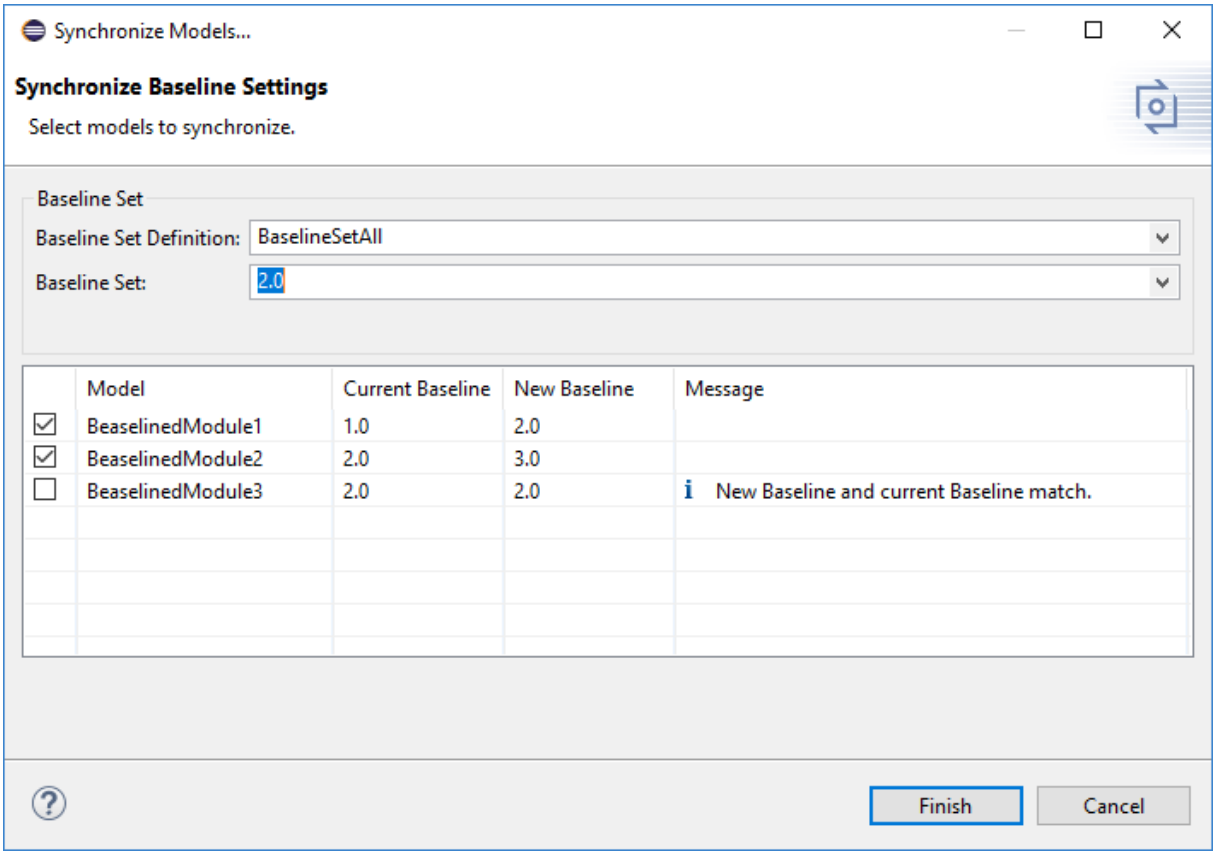
The Merge toolbar (see marked area between upper and lower editor windows at right) provides tools to copy single or even all (non-conflicting) changes as a whole from the DOORS model to the pure::variants model.

Updating all Quickmode Models connected to one Configuration Space

If baseline sets are used to group DOORS modules this information can be used to update baseline information in all Quickmode models in on configuration space. To use this functionality chose **Synchronize Models with Baseline Set** from the context menu of the config Space.

A dialog pops up. The dialog provides all Baseline Set Definitions connected to the input models of the configuration space. Models not imported from DOORS are not shown here.

Figure 21. Synchronize Models with Baseline Set Dialog



After selecting a Baseline Set Definition all Baseline Sets contained in the selected definition are shown. Selecting a Baseline Set results in showing the Baseline for each module in the **New Baseline** column of the table. The current in the models defined Baseline is shown in the **Current Baseline** column of the table. The **Message** column shown some information or warnings for the modules.

Clicking on the **Finish** button updated the Baseline information in the selected models.

3. Advanced Topics

3.1. Adding Variability Information in DOORS

Defining an Element Variability Type

A DOORS attribute named `pvVariationType` can be used to provide pure::variants with information about the intended variability type of an object. For each object with this attribute, the attributes value is matched against the four possible variability types for elements (use strings `mandatory`, `alternative`, `or`, `optional`, `ps:mandatory`,

`ps:alternative`, `ps:or`, `ps:optional`) during initial import or sync (if "DOORS is master for variability information during sync" was enabled). If this attribute is not defined or contains any other value than listed above, the default value of `ps:mandatory` or `ps:optional` (if the element has a restriction defined in attribute `pvRestric\tion`) is used.

If "DOORS is master for variability information during sync" was disabled and the DOORS attributes value is changed after import this will not be shown as mergable difference in Compare view. This is to prevent users from accidentally overwriting the variability information stored in their pure::variants models. However, the corresponding attribute of the pure::variants element (`doors:pvVariationType`) will reflect this change and will be shown as change in the Compare view.

Defining an Element Name

Each imported DOORS object gets an automatically generated pure::variants unique name. This name can be used in restrictions, constraints and calculations in pure::variants. If the automatically generated name is not suitable, it is possible to define a different unique name in DOORS.

If the DOORS attribute `pvName` is defined and not empty, this value is used as unique name. To prevent later problems the name is checked during import and synchronization. In case of violation of the naming conventions pure::variants automatically converts the name to a compatible name. However, pure::variants does not prevent creation of non-unique names during import and synchronization. If duplicate names are existing, they will be marked in the model editor.

Defining Element Restrictions

A DOORS attribute named `pvRestriction` can be used to generate a pure::variants restriction for the related pure::variants element. The language used for restriction definition is pvSCL. The pvSCL language is described in detail in the pure::variants User Guide.

Restrictions (as usual in pure::variants) may refer to elements in the same model or in any other model used together with the defining model in a configuration space. So if a requirement element should only be selectable, if a feature with unique name "MyFeature" or the feature "MyOtherFeature" is selected, simply use "`MyFeature or MyOtherFeature`" as value for the restriction attribute `pvRestriction`.

A restriction on a parent element, which is evaluating to false does deselect the whole subtree below the restricted element. None of the children is selected anymore. The restrictions on the child elements do not have any influence on the selection of the children in that case.

To edit the restriction attribute in a more user-friendly way, the pure::variants Integration can be used. It provides a pvSCL editor featuring auto completion, syntax highlighting and error check. If it is installed, the editor can be called in the **pure::variants** menu by selecting the **Edit Restriction** item. For more information see Section 3.2, “Using the pure::variants Integration for IBM Rational DOORS”.

Defining Element Constraints

A DOORS attribute named `pvConstraint` can be used to generate a pure::variants constraint for the related pure::variants element. The language used for constraints definition is pvSCL. The pvSCL language is described in detail in the pure::variants User Guide.

Constraints (as usual in pure::variants) may refer to elements in the same model or in any other model used together with the defining model in a configuration space. So if the selection of a requirement should imply the selection of two other elements with the names "MyRequirement" or the feature "MyOtherRequirement" , simply use "SELF implies `MyFeature or MyOtherFeature`" as value for the restriction attribute `pvConstraint`.

As with restrictions, the pure::variants Integration can be used to edit the constraint attribute in a more user-friendly way. It provides a pvSCL editor featuring auto completion, syntax highlighting and error check. If it is installed, the editor can be called in the **pure::variants** menu by selecting the **Edit Constraint** item. For more information see Section 3.2, “Using the pure::variants Integration for IBM Rational DOORS”.

Defining Element Default Selection

Each imported DOORS object gets an default selection state.

This is calculated using the following rules:

- If variation-type is ps:mandatory the element gets default selected
- If the variation-type is optional and the containing model is family model the element gets default selected
- If the variation-type is optional and the containing model is feature model the element gets not default selected
- If no variation-type is undefined and a restriction or constraint is defined, the element gets optional and default selected.
- In all other cases the element is default selected off.

If this does not fit your needs you can specify the default selection for each element by using the DOORS attribute `pvDefaultSelected`.

Allowed values are (ignoring case) `on` and `off`.

Defining Element Relations

To automatically create type relations in pure::variants like ps:conflicts or ps:requires from DOORS links, these links must have the special attribute `pvRelation`. If set to any non-empty value, pure::variants will used the type specified as string in the attributes value instead of the default`doors:link`. The relation concept is described in detail in the pure::variants User Guide.

3.2. Using the pure::variants Integration for IBM Rational DOORS

To support users of the pure::variants - Connector for IBM Engineering Requirements Management - DOORS in adding variability information to DOORS modules, the pure::variants Integration is provided. It can be used to define element restrictions (see the section called “Defining Element Restrictions”) and element constraints (see the section called “Defining Element Constraints”) or to edit calculations in attribute values of the currently selected requirement. To this end, it provides an editor featuring auto completion, syntax highlighting and error check. Furthermore, the Integration offers visualizations to preview variants and find errors in variability information (seethe section called “Visualizing Variability Information”).

The instructions for installing the Integration can be found in the section called “UI Integration Installation”.

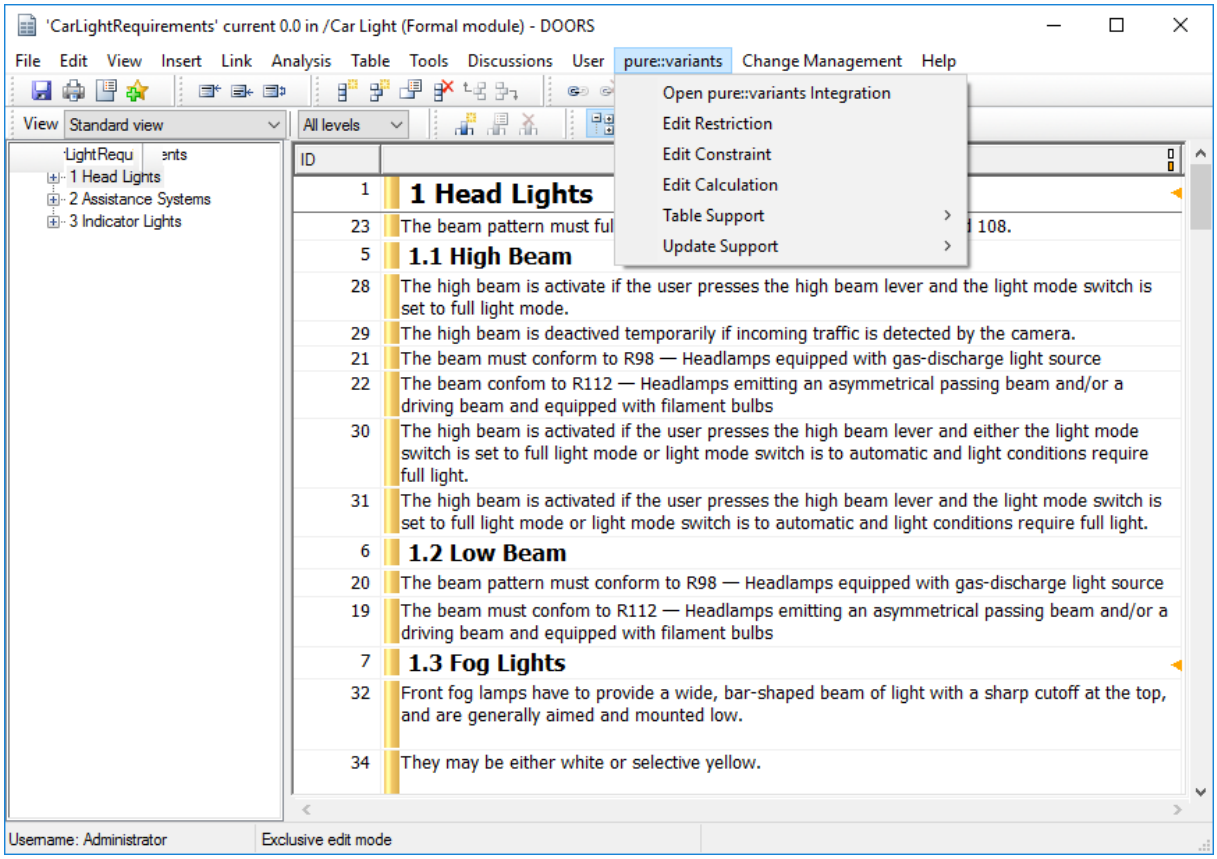
Opening the pure::variants Integration

To start the Integration, first open a DOORS module and select the requirement for which you want to define the restriction or constraint. If you want to define a restriction, select the item **Edit Restriction** from the **pure::variants** menu. For defining a constraint, select **Edit Constraint**. Calculations can be edited using the item **Edit Calculation**. Now the pure::variants Integration opens, and the constraint editor should be displayed.

You can also open the Integration window only, by selecting **Open pure::variants Integration**.

All other menu items are related to update support (see Section 3.5, “DOORS Transformation with Update Support”).

Figure 22. pure::variants Integration Menu in DOORS Module



First Use

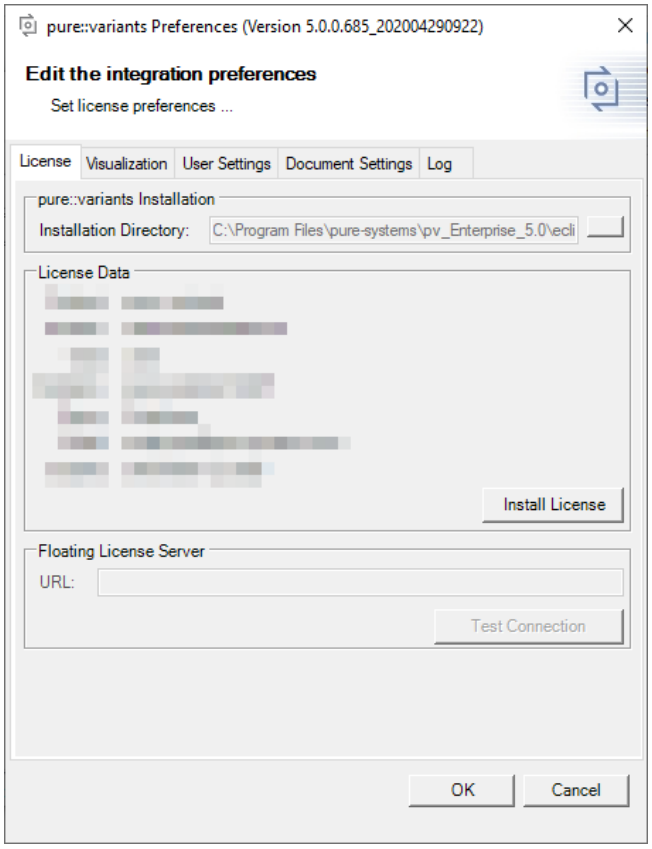
When you first use the Integration after installation, it is necessary to check whether the license preferences are correct. To this end, open the Integration preferences dialog via the button in the Integration window.

A dialog opens that shows the path to your pure::variants installation and your license information (see Figure 23, “Preferences Dialog”). If any of the information is missing, you need to enter it. Use the **...** button in the **pure::variants Installation** group to enter the installation directory, and the **Install License** button to specify your license.

If you are using a floating license and the URL in the **Floating License Server** group is not set already, you need to enter the URL. To test if the connection to the floating license server is established, press the button **Test Connection**.

Now you can use the Integration.

Figure 23. Preferences Dialog



Editing Variability Information

Connecting with pure::variants Models

For editing variability information and viewing visualizations, it is necessary to connect your Doors project with one or more pure::variants models. The following types of pure::variants models can be loaded:

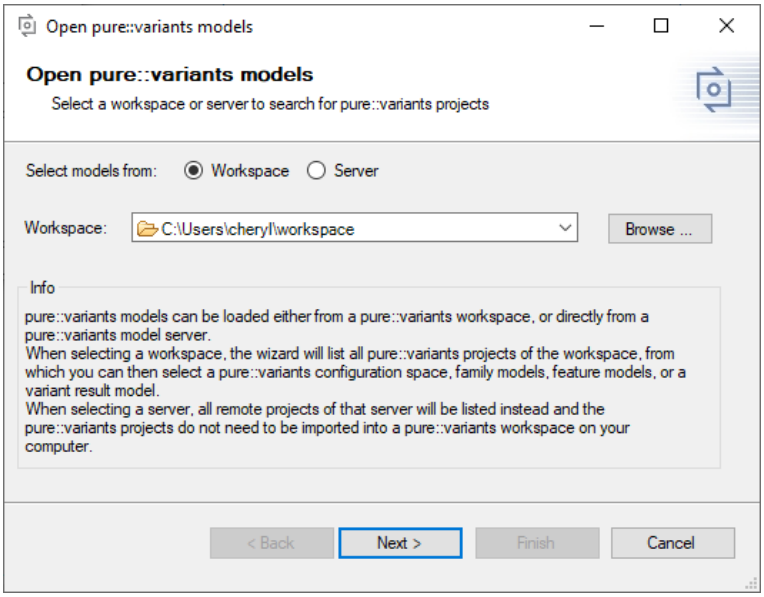
- Recommended: pure::variants configuration spaces, which enable selection of contained variant description models (.vdm)
- pure::variants variant result models (.vrm)
- pure::variants feature models (.xfm)
- pure::variants family models (.ccfm)

pure::variants models can be opened from two different sources: Either from a *pure::variants/Eclipse workspace* or from a *pure::variants model server*.

Opening models from a workspace

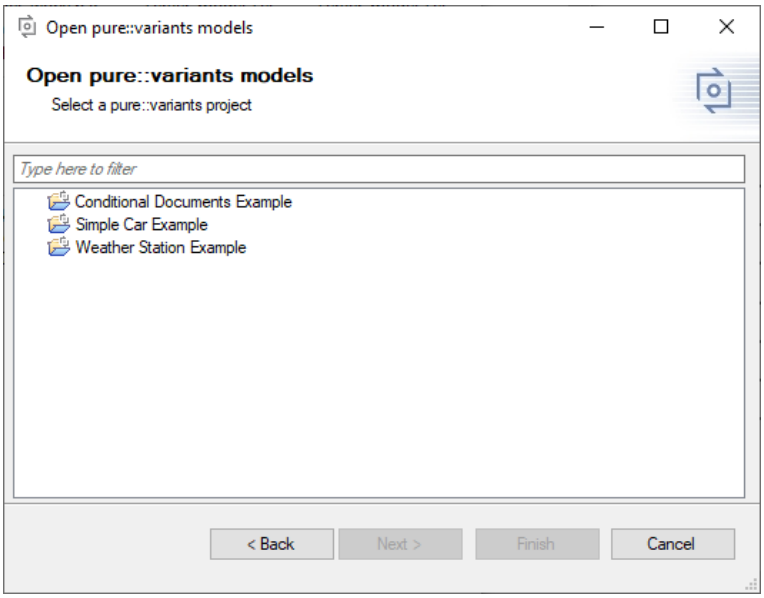
To open a model or configuration space, press on the Integration window. This will open a wizard, which first allows choosing the source (workspace or server). Choose *workspace* and then browse to find your pure::variants workspace folder. Already known workspaces are listed in the workspace dropdown box. If you later need to add or remove a workspace from the list, you can go to tab *User Settings* of the Integration preferences (accessible via).

Figure 24. Mode selection page



On the next page, all projects are listed that are located in the selected workspace folder or that are linked into the pure::variants/Eclipse workspace.

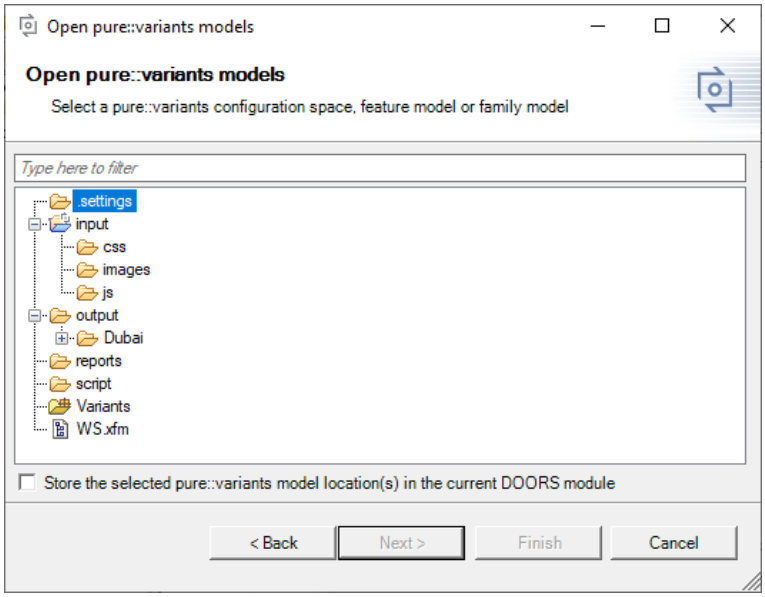
Figure 25. Project selection page



Select one and on the next page choose the model(s) or configuration space you want to open.

Checkbox *Store the selected pure::variants model location(s) in the current Doors module* allows you to save the selected model locations in the current Doors module, so that these models will be opened again once any user opens this Doors module. If you do not select the checkbox, the model locations will only be stored on your computer. For details, see the section called “Saving and Loading pure::variants Models”.

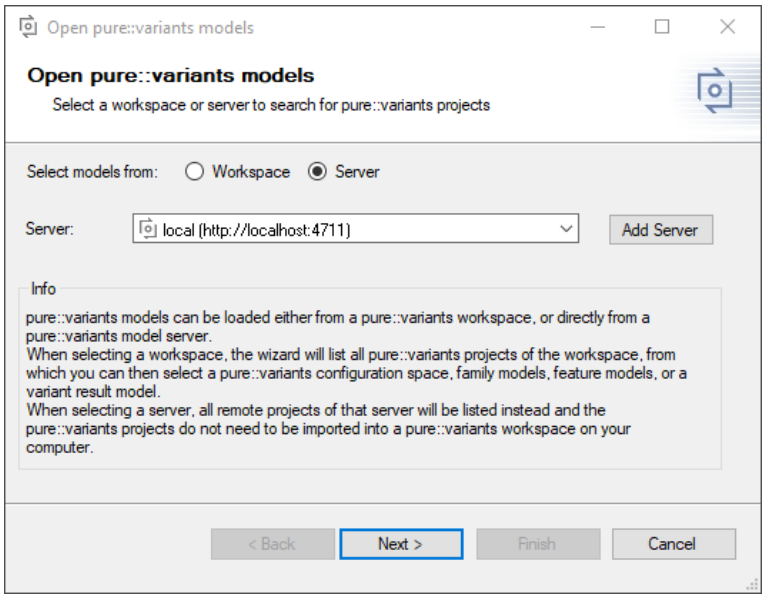
Figure 26. Model selection page



Opening models from a model server

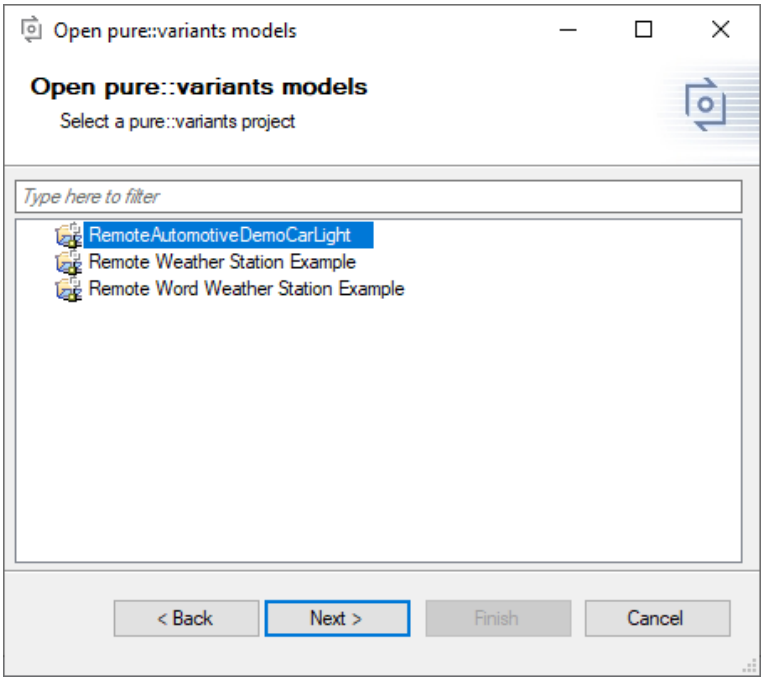
To open a model or configuration space directly from a pure::variants model server, also press . On the first page of the wizard, choose *server* and add the server address via button **Add Server**. Like in pure::variants, new servers need a name and the server address (e.g., https://yourserveraddress:443). Any known servers are listed in the server dropdown box. If you later need to add or remove a server from the list, open the Integration preferences by pressing . On tab *User Settings*, you can add or remove servers.

Figure 27. Mode selection page



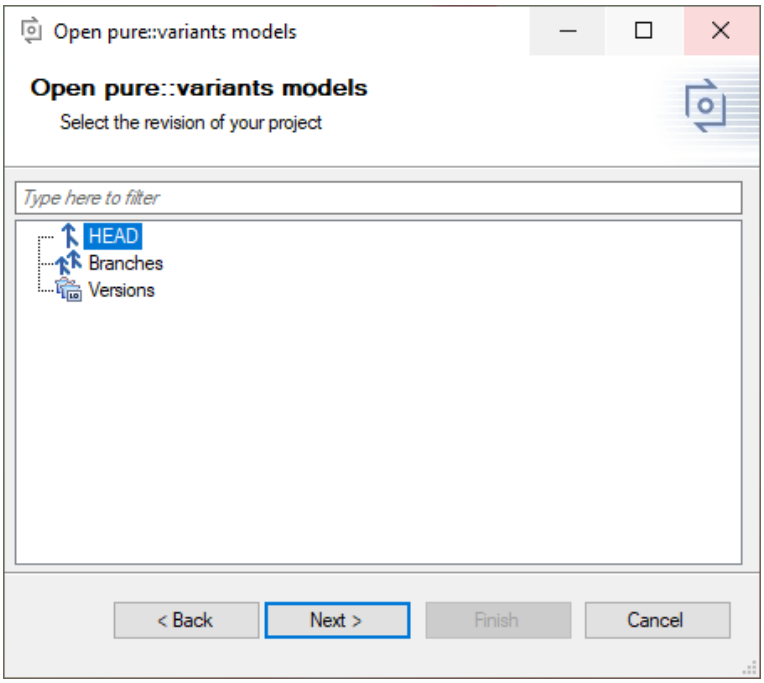
On the next page of the wizard, all projects of the server that the current user has read access to are listed.

Figure 28. Project selection page



Select one and on the next page choose the project's revision (branch or tag) from which you want to load a model.

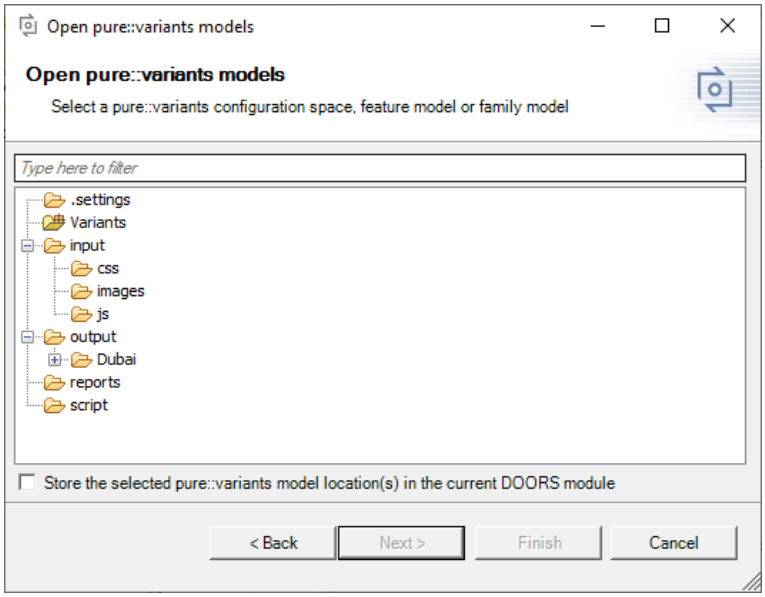
Figure 29. Project selection page



Finally, on the last page select the model(s) or configuration space you want to open.

Checkbox *Store the selected pure::variants model location(s) in the current Doors module* allows you to save the selected model locations in the current Doors module, so that these models will be opened again once any user opens this Doors module. If you do not select the checkbox, the model locations will only be stored on your computer. For details, see the section called “Saving and Loading pure::variants Models”.

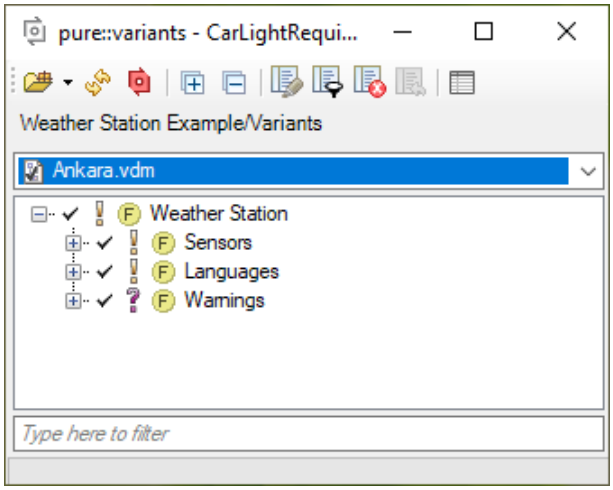
Figure 30. Model selection page



Opening pure::variants Configuration Spaces

To open a pure::variants configuration space, use the wizard as described above. On the last page, select a configuration space folder. Now the Integration window should show all used models of your configuration space. Please note that family models (.ccfm) are not opened per default. You can enable loading family models in the Integration preferences on the **Visualization** tab. After selecting a variant from the dropdown list, selections should be shown in front of features. To ease usage of configuration spaces with many variants, the latest opened variants are shown at the top of the list.

Figure 31. Configuration Space with Selected Variant



Opening Other pure::variants Models

Other pure::variants models, such as variant result models[1] (.vrm), feature models (.xfm), and family models (.ccfm) can also be opened via . Please note that family models (.ccfm) are not listed per default. You can enable loading family models in the Integration preferences on the **Visualization** tab.

[1]You can create a variant result model in pure::variants by clicking the **Save Result to File** button that is shown in the toolbar of a variant description model.

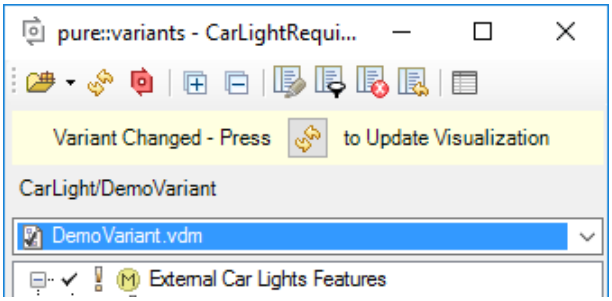
Live Connection with pure::variants

Since pure::variants 4.x, changes of the loaded pure::variants models are propagated live to the Integration. For example, directly after editing the name or changing the selection of a feature the loaded models are updated in the Integration window. To enable this live update, the following prerequisites need to be fulfilled:

- the opened model needs to be located in an Eclipse workspace
- the changes have to be done on the same Eclipse workspace using pure::variants 4.x or later
- either a configuration space, feature model or family model needs to be loaded (Variant result models can only be updated automatically when the .vrm file is saved)

If a visualization is active when a loaded model has changed, a pane is shown that informs you about a pending visualization update (see Figure 32, “Information about Pending Visualization Update”). When pressing the pane's refresh button, the visualization is updated.

Figure 32. Information about Pending Visualization Update



When the used models of a configuration space have changed or a new variant model was added to the configuration space, a live update of the currently loaded models is not possible. In this case, you can press to manually reload all pure::variants models and refresh the current visualization. To unload all models and free the pure::variants license, press .

Saving and Loading pure::variants Models

To ease the work with pure::variants Integrations, the last loaded model locations are saved, so that the model will be opened again automatically, next time you start the tool. Per default, these model locations are saved only for you on your local machine. If you want to save the last loaded model locations for all users, who are using a certain Doors module, you can select checkbox *Store the selected pure::variants model location(s) in the current Doors module* on the last page of the open model wizard. Then everytime a user opens that Doors module, the pure::variants models stored in the document will be opened instead of the locally stored models (if they can be found on the user's machine).

Furthermore, a list of the latest loaded models can be accessed via the small arrow next to the button.

Please note that models are saved relative to your current workspace, which is the workspace where your current model is located in or linked to. Therefore, you may be asked for your current workspace location when loading a model from a different workspace, or a model that is not located in a workspace (but may be linked into a workspace). Hint: If you want to know where exactly the loaded model is located, you can hover over the name of the model. A tooltip will show the full path of your currently loaded configuration space or model.

If you do not want to load a model again on startup, you can clear the stored model locations from the user and/or document settings. For details, see the section called “Manage Settings”.

Removing Old Model References

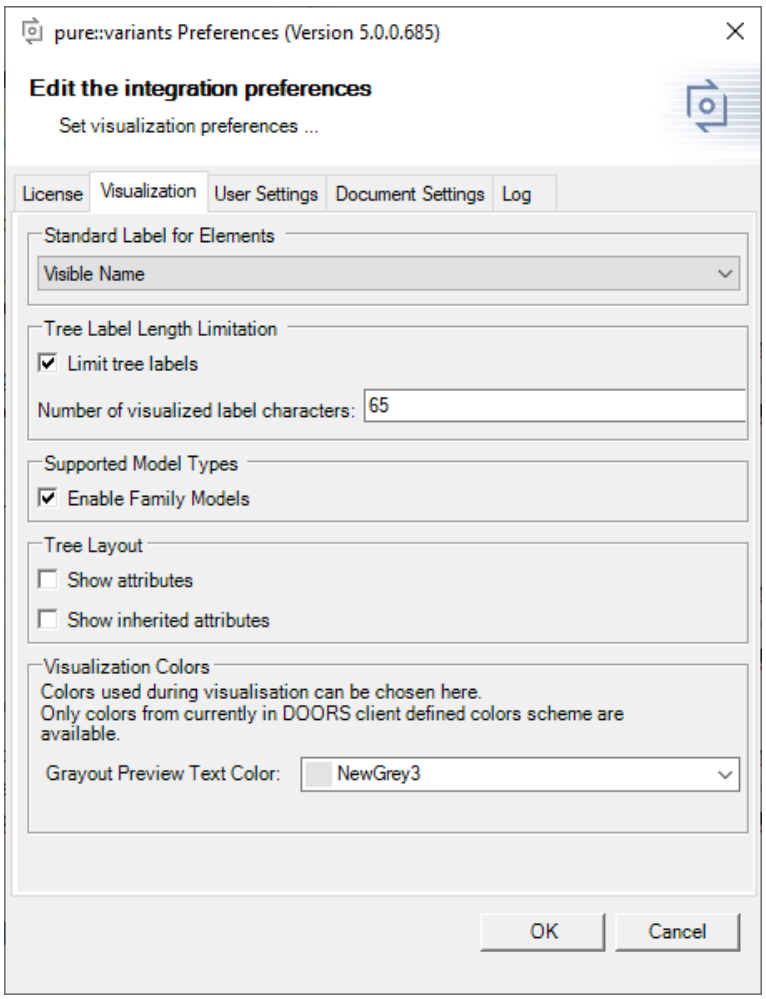
Since the loading of models has changed extensively in pure::variants Integrations of version 4.x, Integrations now save pure::variants models differently in the current document than they did in version 3.2.x. Therefore, references

to models loaded with version 3.2.x may still exist in your document, but are not used. This may be as intended if you still also need to open the document with an Integration of version 3.2.x. However, if this is not necessary you can remove these references from your document: Open the Integration preferences and press the **Remove References** button in the lower part of the **Log** tab (see Figure 44, “Preferences Dialog Log Tab”).

Model Visualization Preferences

In the Integration preferences, you can set how pure::variants models will be displayed and which model types are supported. To do that, open the Integration preferences by pressing and go to the **Visualization** tab (see Figure 33, “Preferences Dialog Visualization Tab”). The first dropdown box enables you to set how elements in the pure::variants model view are labeled. Furthermore, you can limit how many characters are shown for each element in the tree, enable or disable the loading of family models, and set whether attributes are shown in the model tree. To also show attributes inherited from parent elements, select **Show inherited attributes**.

Figure 33. Preferences Dialog Visualization Tab



Manage Settings

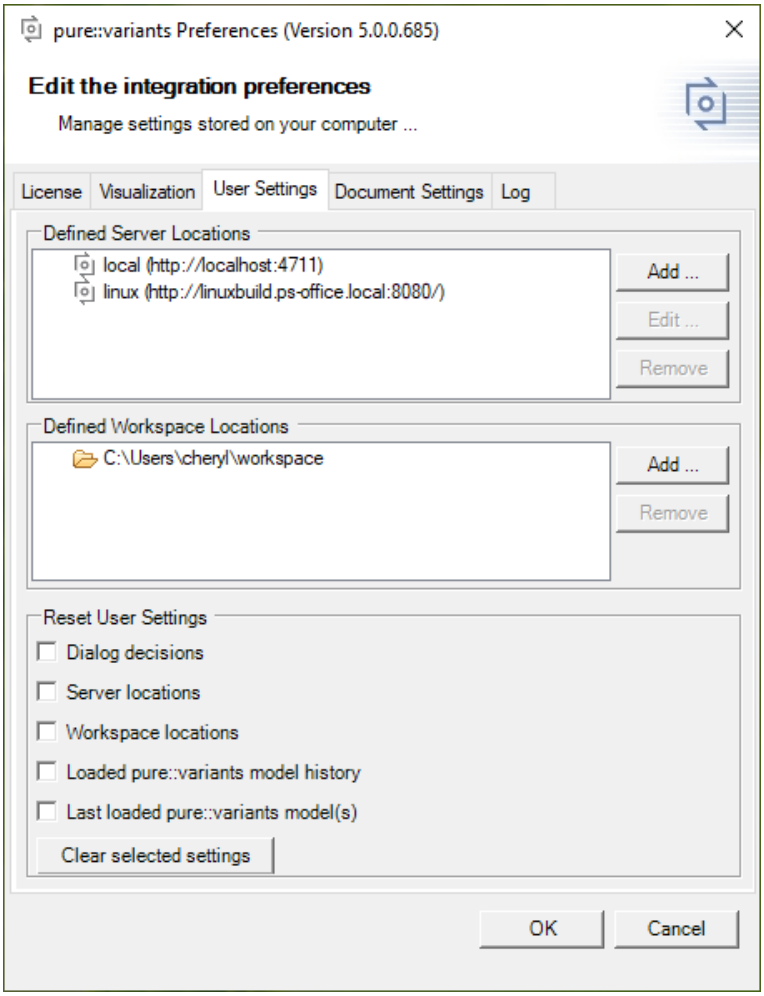
In the Integration preferences, you can also manage settings stored by the Integration. You can, for example, add or remove known workspace and server locations or clear certain settings. To do that, open the Integration preferences by pressing .

Here, you can find tabs *User Settings* and *Document Settings*.

On tab *User Settings*, you can manage settings that are stored only on your local machine, specifically for your user. (see Figure 34, “Preferences Dialog User Settings Tab”). This includes server and workspace locations,

dialog decisions, the history of previously loaded pure::variants models, and so on. The first section, **Defined Server Locations** enables you to add, edit and remove the server locations that are not locked. The second section, **Defined Workspace Locations** enables you to add and remove the workspace locations. The last section **Reset User Settings** enables you to clear the selected settings. For example, you can select checkbox *Last loaded pure::variants model(s)* and press the clear button, to make sure no pure::variants model is loaded from the user settings at startup of the Integration.

Figure 34. Preferences Dialog User Settings Tab

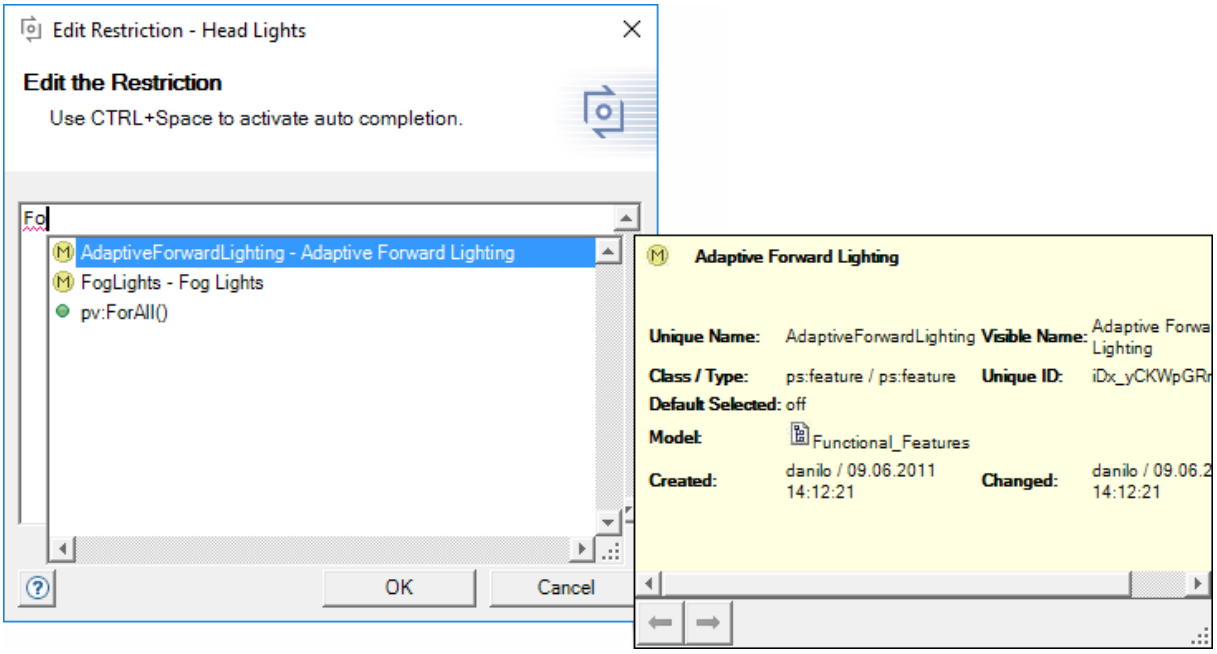


On tab *Document Settings* you can manage all settings that are stored in the current Doors module. For example, you can clear the last loaded pure::variants model from the current document, so that, when you or other users next open the current Doors module, the Integration will load no pure::variants model stored in the document.

Working with the pvSCL Editor

The pvSCL editor features auto completion and syntax highlighting (see Figure 35, “Using the Constraint Editor”). You can automatically complete words by pressing CTRL + space. If more than one word is possible, a list containing possible keywords and features including all pvSCL functions is displayed. For auto completion the loaded pure::variants models are evaluated.

Figure 35. Using the Constraint Editor



When you are done editing the pvSCL script and press the OK button, the entered expression is checked for errors. If no errors were found, the pure::variants Integration closes and the entered pvSCL expression is added to the selected requirement of the opened DOORS module. If an error is found, it is reported, so you can correct the pvSCL expression before writing it to the DOORS module.

Errors in pvSCL expressions are reported if the expression syntax is not pvSCL compliant, or if an element is unknown based on the loaded pure::variants models. Unknown elements are underlined in red.

The attributes used for storing the pvSCL restriction or the pvSCL constraint can be configured in the Integration preferences on the **Transformation and Preview** tab (see Figure 36, “Customizing Transformation and Preview Settings”). These settings are saved in the DOORS module, and thus can differ between modules. They are used during preview and transformation. Simply chose one of the available attributes in the combo boxes.

Note

To save the attribute settings, the module has to be opened in exclusive edit mode. The reason is that these settings are saved to a module attribute in the currently opened module. This enables the Doors module importer to also reuse these settings.. Write access to module attributes is only available in exclusive edit mode.

Working with the Calculation Editor

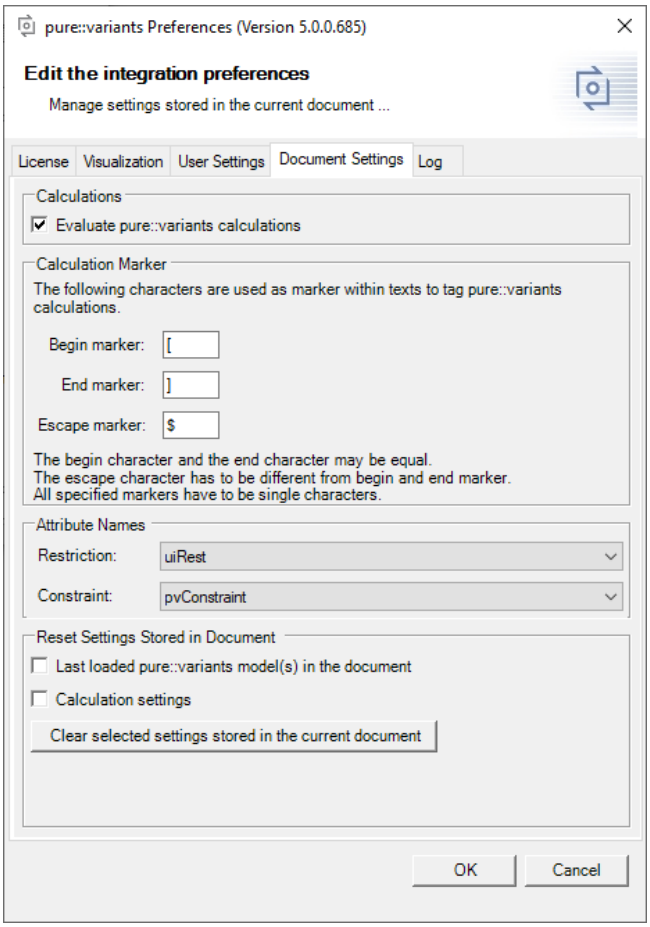
The calculation editor helps the user to add and edit pure::variants calculations in texts. Calculations are used to automatically insert pure::variants attribute values to text and string attributes in DOORS modules.

Before the user can define a pure::variants calculation he needs to tag the target location for the pure::variants attribute value. This is done by manually adding the begin (default: "[") and end (default: "]") marker for the pure::variants calculation in the requirement text. The markers can be configured in the Integration preferences on the **Document Settings** tab (see Figure 36, “Customizing Transformation and Preview Settings”). These settings are saved in the DOORS module, and thus can differ between modules. They are used during preview and transformation.

Note

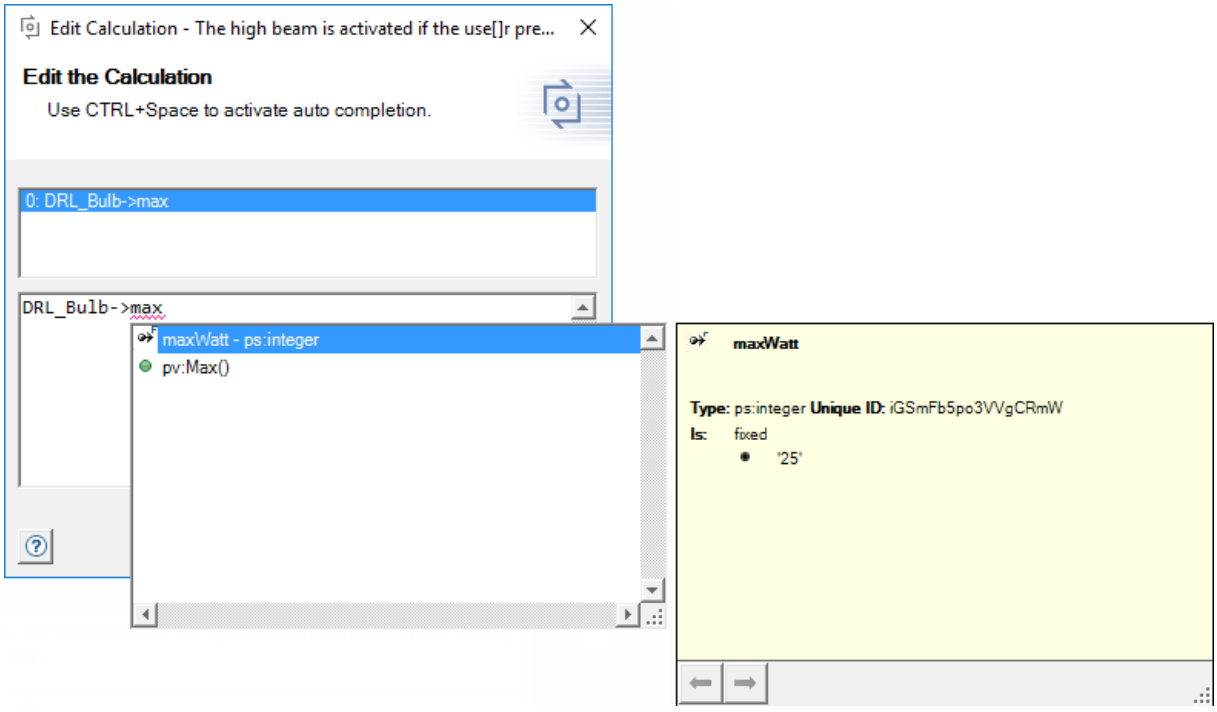
To save the calculation settings, the module has to be opened in exclusive edit mode. The reason is that these settings are saved to a module attribute in the currently opened module. This enables the transformation to also use the settings. Write access to module attributes is only available in exclusive edit mode.

Figure 36. Customizing Transformation and Preview Settings



After the markers are added to the requirement, the calculation editor can be opened via the pure::variants menu.

Figure 37. Using the Calculation Editor



The editor shows all calculations of one attribute of the selected requirement. If more then one attribute value has calculations, the attributes can be selected from a drop down menu shown on the top of the dialog. To edit one calculation just select the calculation in the above list and edit it in the text field below. The calculation editor is working similar to the pvSCL editor. Context help is available and problems with the rule are shown within the editor.

After editing the calculations close the editor. All calculations are checked and inserted at the marked locations. If errors are found, they are reported to the user before adding the calculations to the attribute texts.

Note

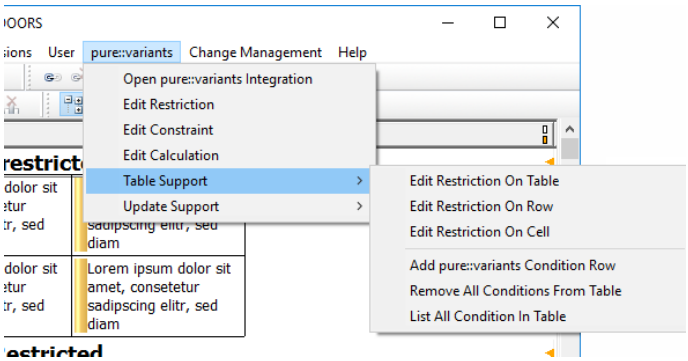
We do not recommend using calculations in DOORS attributes containing OLE objects. This may result in very long preview and transformation runs.

Adding Variability Information To Tables

pure::variants enables the user to add conditions to DOORS table elements. The conditions can be added to cells, rows, columns and the whole table. Since it is not possible in DOORS to edit attributes on rows, columns and tables, the pure::variants integration for DOORS has to be used to edit the conditions in tables. Conditions on columns, rows and tables remove the corresponding elements from the module. Conditions on cells only blank the cells, meaning all content is removed from the cells' objects, so the table layout remains the same.

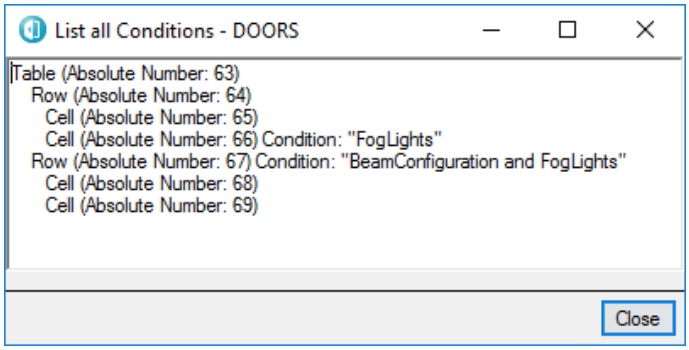
Adding a condition to cells, rows or tables is done the same way. Choose the corresponding item from the *Table Support* sub menu of the *pure::variants menu*. The restriction editor opens and allows the user to specify a condition. These conditions are not visible in the module.

Figure 38. The Table Editing Menu



To view existing conditions in a table, pure::variants provides the *List All Conditions In Table* menu entry. This opens a text dialog, which lists all elements in the table and their conditions. Figure 39, “The List Conditions Dialog” shows an example result. The example table has two rows with two cells each. The second cell in the first row has condition *FogLights* and the second row has condition *BeamConfiguration and FogLights*. There is no condition row in that table.

Figure 39. The List Conditions Dialog



Since DOORS does not have columns modeled in its modules, we introduce a pure::variants condition column to enable the user to define conditions on table columns. The condition row is added using *Add pure::variants Condition Row*. There is only one pure::variants condition row allowed per column. Conditions added to the cells of this condition row are considered for the complete column. The pure::variants condition row is removed automatically during transformation.

The *Remove All Conditions From Table* entry removes all conditions from all table elements. This includes removing the pure::variants condition row.

Visualizing Variability Information

After variability information has been added to the module, you can use visualizations to check whether the variability information is correct: Either preview variants for loaded variant models or mark errors in variability information. When you apply visualizations, they affect all visible module elements of the loaded DOORS module.

While performing a visualization it is possible to edit restrictions, conditions and calculations using the pvSCL editors available through the **pure::variants** menu. This will update the visualization. No other data can be edited in visualization mode. To edit other data than restrictions, constraints or calculations, you need to leave the visualization mode by pressing  in the Integration window.

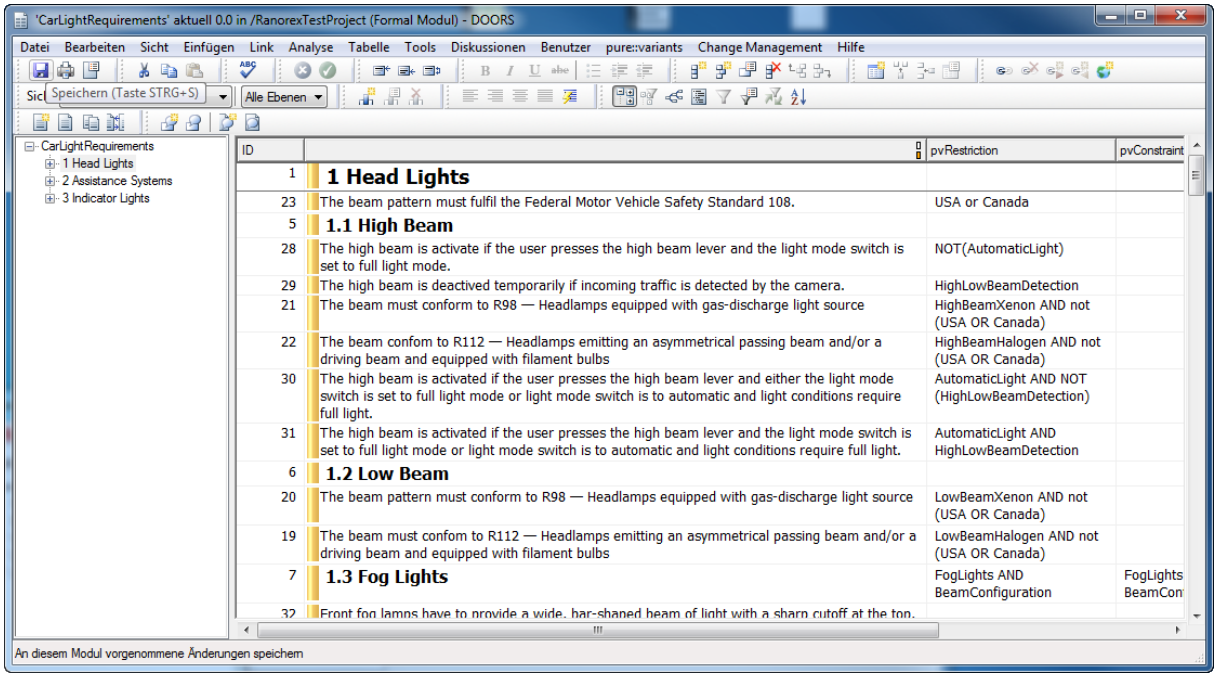
Variant Preview Visualizations

Through variant preview visualizations you can preview the results of a transformation with pure::variants. For preview visualizations you need to load a variant. This can be either a configuration space with a variant description model (.vdm) or a variant result model (.vrm). There are two types of variant previews: *Grayout* and *Filter Preview*.

Note

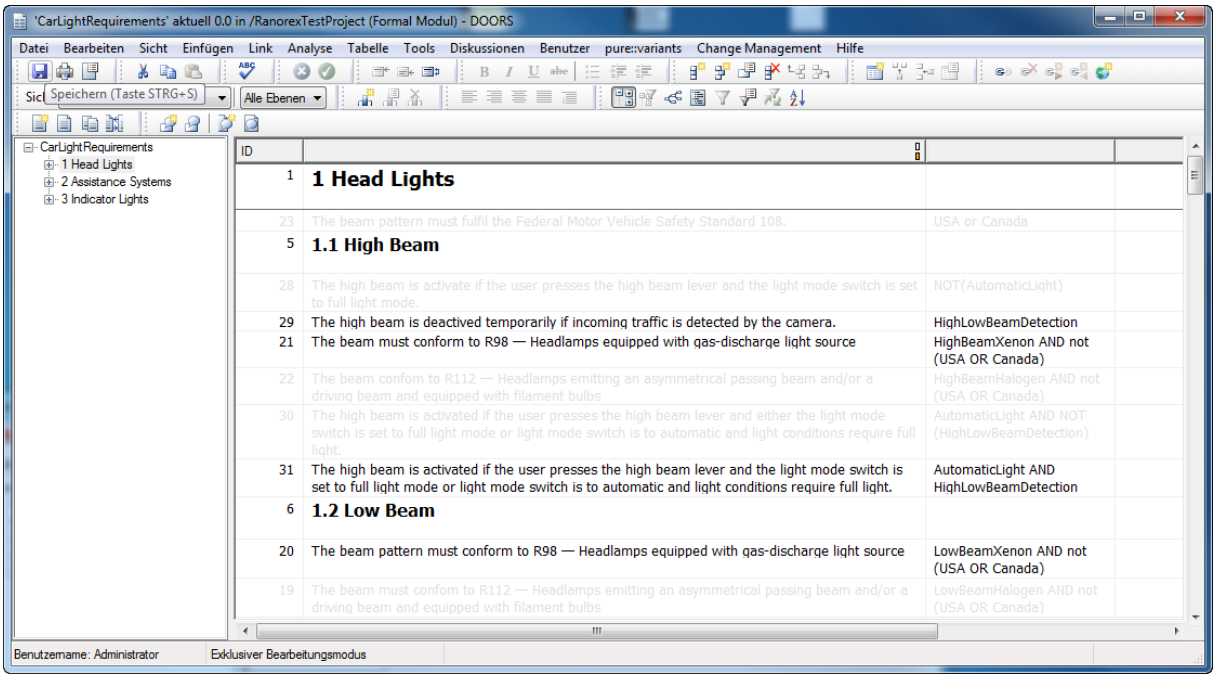
During preview visualisation the calculations added to DOORS attributes are considered for *Object Heading*, *Object Short Text* and *Object Text* only. During transformation all attributes of type *String* and *Text* are considered.

Figure 40. Example Module



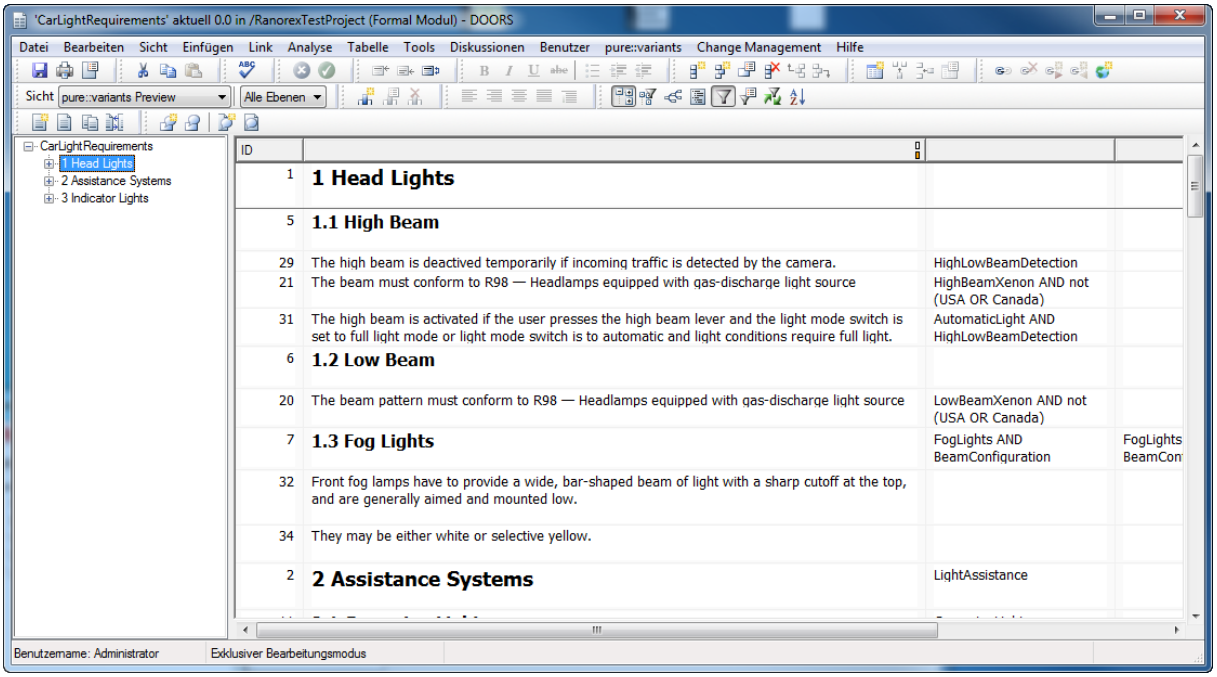
When you select the grayout preview by pressing  , all elements that would be excluded in the corresponding variant are grayed out. Figure 41, “Grayout Preview” shows such a visualization in a DOORS module. Some requirements are grayed out, since the assigned features are not selected in the loaded variant.

Figure 41. Grayout Preview



When you apply the filter preview by pressing , all elements that would be included in the variant, remain in the pure::variants Preview view, while all other requirements are filtered. Figure 42, “Filter Preview” shows the result of such a filter preview. The preview view shows only requirements, which will remain in the module if a transformation would be performed.

Figure 42. Filter Preview

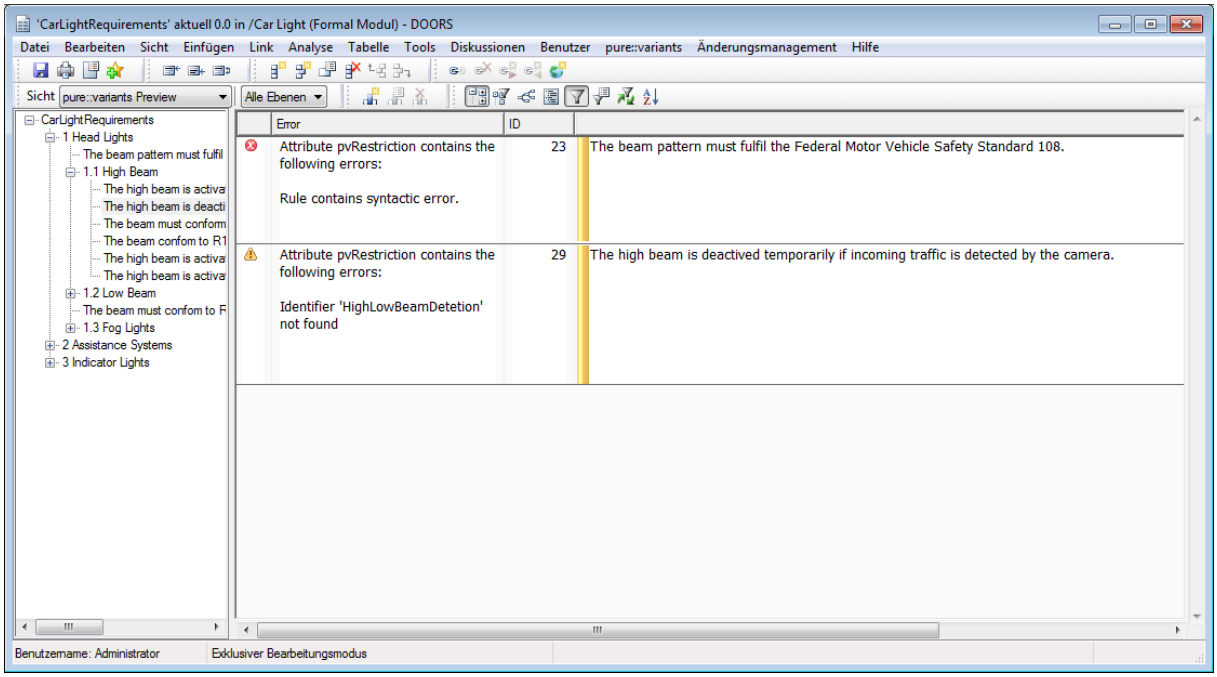


Error Visualizations

Error visualizations ( ) indicate where errors in variability information exist (errors in the pvSCL expression of pure::variants constraints). To be able to view error visualizations, a pure::variants model has to be loaded (see the

section called “Connecting with pure::variants Models”). There are two types of errors: Elements with syntactic errors are marked with , whereas elements with semantic errors are marked with . To help the user in finding the faulty rules, the error visualization filters the module contents and shows only requirements with faulty rules. Figure 43, “Syntax Error Visualization” shows an example of an error visualization. The BeamPattern requirement contains the rule *USA or*, which is semantically incorrect, because the second feature reference is missing. The semantic error is simply a misspelling of the feature references *HighLowBeamDetetcion*. *HighLowBeamDetection* would be correct.

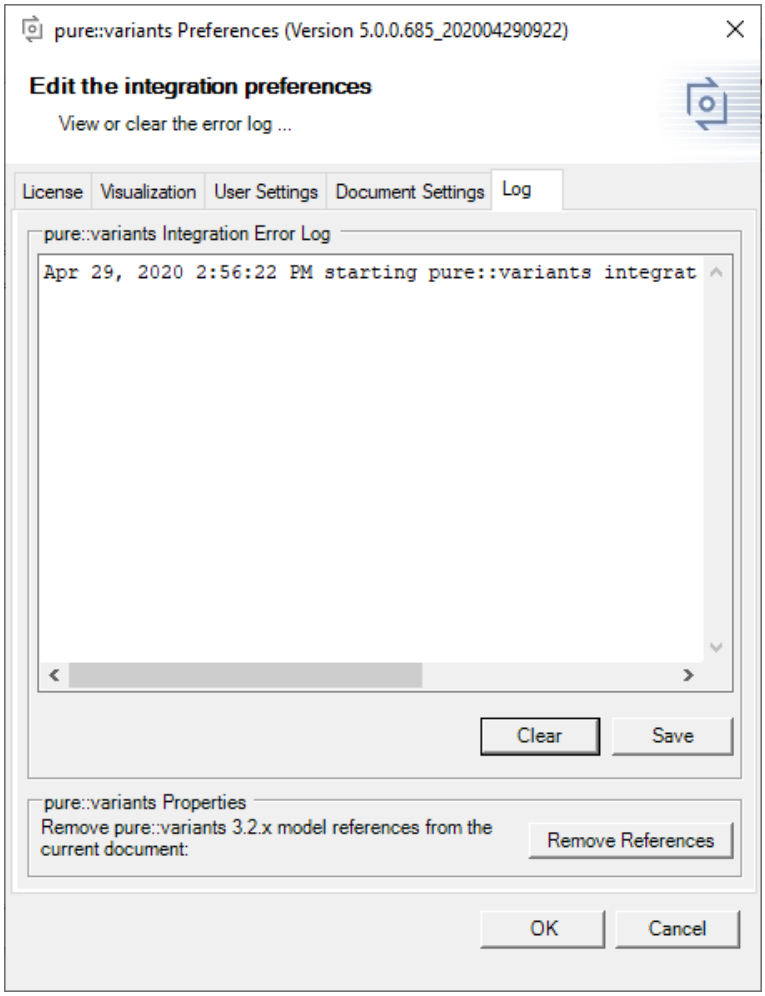
Figure 43. Syntax Error Visualization



Troubleshooting

If the Integration does not behave as expected, it may be useful to check its log file. You can find it in the Integration preferences ( ) on the **Log** tab (see Figure 44, “Preferences Dialog Log Tab”). It also enables you to save it to your disk or clear the contents of the log file. Furthermore, the **Log** tab provides the option to remove references to pure::variants models that can only be read by Integrations of version 3.2.x (see the section called “Removing Old Model References”).

Figure 44. Preferences Dialog Log Tab



3.3. Customizing Import of DOORS Attributes

Not all information stored in DOORS attributes has to be available in the pure::variants model. The truncation of attribute values during import can be toggled by the user in the Connector preferences page. `Object Text` and `Object Short Text` have their own settings (Maximal Text Length), all remaining attributes use a shared value (Maximal Attributes Length). Setting a value to 0 or negative numbers will result in empty attribute values.

The list of imported attributes can also be customized. This is also modifiable on the preferences page. Simply add the names of the attributes to be ignored into the list. **Tip:** It is possible to use regular expressions here. See the DOORS online help for a description of the syntax to use.

3.4. Calculations within Attribute Texts

Texts in DOORS attributes may have variable parts, while the most part of the text will remain equal in all of your variants.

In this case you can add a pvSCL statement to be evaluated by pure::variants. The statements will be replaced with actual values of your variant by pure::variants, if you perform *partial text substitution* during the export or transformation of your variant. The actual values are also shown during the preview performed with the pure::variants Integration for IBM Rational DOORS.

A pvSCL statement starts with an opening marker (default: '[') followed by an pure::variants pvSCL calculation rule and is ended with an closing marker (default: ']'). To escape a statement the escape character is used (default: "$"). This will prevent pure::variants from evaluating and replacing the escaped statement. The characters used for tagging a calculation are configurable. See Figure 36, “Customizing Transformation and Preview Settings”.

Example: *The maximum allowed speed is [Speed->Max] km/h.* in an DOORS attribute will be replaced with the value of attribute *"Max"* on Feature *"Speed"* in the exported variant. The result could be: *The maximum allowed speed is 100 km/h.*

Escaping the rule in the previous example, like *The maximum allowed speed is $[Speed->Max] km/h.*, forces pure::variants to ignore the rule. The result, would be *The maximum allowed speed is [Speed->Max] km/h.* in that case. The rule is not changed, but the escape character is removed.

Note

This functionality is an optional feature and has to be explicitly selected during export or transformation, because it may slow down your export process.

3.5. DOORS Transformation with Update Support

The pure::variants DOORS transformation supports the updating of exported variants, i.e., the application of changes done in the product line after the variant was exported without reverting user changes meanwhile done in the exported variant. For a general introduction for update support in transformations in pure::variants see the pure::variants User's Guide, Section 5.10. Variant Update.

How Update Works

The update is done based on three versions of the exported variant:

Working copy	a variant created by the DOORS transformation that was possibly edited by the user
Latest reference	a variant created by the DOORS transformation, which reflects the latest state of the product line
Ancestor reference	a variant created by the DOORS transformation, which is the common ancestor of both working copy and latest reference

The update process has the task to apply changes done in the latest reference, in comparison to the ancestor reference, to the current working copy without reverting user changes.

If the DOORS transformation is running in *enabled update support mode*, in the first run, i.e., if the working copy does not exist already, the transformation will create for each processed module a new working copy module and a new ancestor reference module as a copy of it. After that, the user can edit the working copy. The changes he has done can be determined at any time by comparing working copy and ancestor reference.

In the second and all following transformation runs, i.e., as long as a working copy exists, the transformation creates a new latest reference module for each processed module instead. If the transformation parameter **UpdateMode** is set to **AutoUpdate** the existing working copy will be updated automatically based on the existing ancestor reference and the new-created latest reference. If the **UpdateMode** is **ManualUpdate** this step is not done automatically. Instead, the user has to trigger the update on a per-module base using the pure::variants DOORS integration menu. If the **UpdateMode** is **OnlyUpdateLinks** only the links of all the transformed modules in the already existing set of latest modules are updated in the working copy. If the **UpdateMode** is **OnlyUpdateObjects** only the objects of all the transformed modules in the already existing set of latest modules are updated in the working copy.

The working copy modules are created in the same DOORS destination folder as when using the DOORS transformation without update support.

The ancestor and latest reference modules are created in a special folder structure, the reference repository, inside the variant root folder. If this is not sufficient, the reference repository path can be customized for each module. Therefore, in the corresponding family model, an attribute named *doors:repositoryPath*.

Please have a look to section the section called “Family model attributes used to customize transformation behavior” to find more information about the customization properties.

Module Instrumentation for Update Support

To permit the update of the working copy, the created modules contain some extra attributes, which must not be changed by the user:

pvRefID (object/linkset/link attribute)	An ID that is set for each generated object or link. This ID is used together with *pvRefIDScope* to map corresponding objects/links among working copy and references. User-added objects/linksets/links have no set*pvRefID*. For linksets, *pvRefID* is used to mark this linkset as generated.
pvRefIDScope (object/linkset/link attribute)	An ID that is optionally set for each generated object or link to differentiate the scopes of the *pvRefID*. New scopes are needed if working copy modules are used as a source for a new variant derivation. User-added objects/links have no set *pvRefIDScope*. For linksets, *pvRefIDScope* is used to mark this linkset as generated with scopes.
pvRefAbsNum (object attribute)	The absolute number of the object when created by transformation. It is used to recognize user-copied objects in the working copy and to check therefore the validy of the corresponding *pvRefID* value.
pvGenerated[_i] (module/linkset attribute)	Contain the encoded *pvRefIDScope*s and *pvRefID*s of the generated object/links. They are needed to recognize user-deleted objects/links, which were created by the transformation before.
pvGeneratedLinkModules (module attribute)	Contains the paths of all link modules of generated outgoing links. This is needed to find the link modules also in the case, when in the working copy all links were removed by the user.
pvLatestReferenceModule (module/linkset attribute)	Path of the corresponding latest reference module or latest reference link module. Set in working copy and ancestor reference modules and in working copy link modules.
pvAncestorReferenceModule (module attribute)	Path of the corresponding ancestor reference module. Set in working copy and latest reference modules.
pvWorkingCopyModule (module attribute)	Path of the corresponding working copy module or working copy link module. Set in latest and ancestor reference modules and in latest reference link modules.
pvUpdateUseSoftDeleteOnly (module attribute)	Defines if objects, which should be removed in the working copy, will be purged or only soft-deleted.

To control the update process, the following attribute can be changed by the user:

pvDoNotUpdate (object attribute)	Boolean attribute processed only in working copy. It is initially not set. Defines if an object and its subobjects should be excluded from update. If set to true, the objects and all its subobjects will not be updated. If set to false, the update of the object and its subobjects is allowed. Set values in objects of lower levels will overwrite values set in objects of higher levels.

The result of the update can be checked using following attribute:

pvUpdateState (object attribute)	Created in working copy only and is initially not set. After the update it contains the update result state of the object.
pvUpdateStateModAttr (module attribute)	Created in working copy only and is initially not set. After the update it contains the update result state of the module.

pvRefID Scopes

The usual use case for module-based variant derivation with update support is to derive a working copy from a user-created source module. In that case, the source module object's *Absolute Number* is used to initialize the *pvRefID* in the initial working copy module and in each latest reference module during instrumentation. Similarily,

the *pvRefID* of each link is initialized to the combination of the *Absolute Number*s of the link's source and target object.

However, with the new partial derivation mode in pure::variant 5.0 (but not limited to it) it is reasonable to also derive a working copy from another working copy created itself from a previous (partial) derivation. That is, the modules to be instrumented for update are already instrumented. In pure::variants 4.0 (and 5.0.0), such an existing instrumentation was ignored and the *pvRefID* was reassigned to the new *Absolute Number*(s) of the source working copy. This approach has the disadvantage that an update across serveral stages of variant derivations cannot be done properly in each case, since the potentially changed *pvRefID*s hamper the mapping of corresponding objects and links.

To support the update across several stages of working copy derivations, pure::variants 5.0.1 introduces *pvRefID scopes* to enable that *pvRefID*s do not change among the derivation stages and ensure that no collisions can occur when new objects or links are added by the user in the different stages. For that the new *pvRefIDScope* attribute defines the number of derivation stages the object or link was derived ago. An object or link with *pvRefIDScope* = 1 originates from the previous source module, *pvRefIDScope* = 2 means it originates from the source module two stages ago, etc.

So, the instrumentation works as follows since pure::variants 5.0.1:

- For the first derivation stage, i.e., if the source module is not instrumented yet, the *pvRefID* attribute is added to the variant module and is initialized as before, using the *Absolute Number*(s) of the object or link's source and target, respectively. For compatibility with older derivations, *pvRefIDScope* is not added in that stage. However, implicitly the *pvRefIDScope* has the value 1 for each derived object or link, since they originate from the source module. So, for the first derivation stage, there is no change in the behavior in comparison to pure::variants 4.0.

- In the second stage, the source working copy is instrumented yet: The previously generated objects and links have already a (valid) *pvRefID*. However, new user-added objects and links in the working copy will have no *pvRefID*. For instrumentation the new attribute *pvRefIDScope* is created. Objects and links, which have already a *pvRefID*, will keep their *pvRefID* and their *pvRefIDScope* is initialized to 2. The user-added objects or links will get a new*pvRefID* based on their *Absolute Number*(s) and their *pvRefIDScope* is initialized to 1.

- For all further stages, the source working copy is already instrumented with *pvRefIDScope* and *pvRefID*. Again, in each stage new objects or links can be added by the user. In the corresponding instrumentation, objects and links, which have already a *pvRefIDScope* and *pvRefID*, will again keep their *pvRefID*, but their*pvRefIDScope* is increased by one. So, in the *n*th stage, *pvRefIDScope* can have a maximum value of *n*. As in the second stage, the user-added objects or links will get a new *pvRefID* based on their *Absolute Number*(s) and their *pvRefIDScope* is initialized to 1.

So, beginning with the second derivation stage, the initialization of the *pvRefID* will differ between pure::variants 4.0 and 5.0. In pure::variants 4.0 no derivation stage has a *pvRefIDScope* attribute. So its stage cannot be recognized. But this incompatibility will be checked during the update process: If the new derived latest reference module has a*pvRefIDScope* attribute (i.e., it is at least of second stage) and the working copy to be updated has not (since it was created with pure::variant 4.0 before), the update process will be stopped with an error.

To be able to update a second-or-later-stage-derived working copy created by pure::variants < 5.0.1, the working copy and eventually the ancestor reference modules need to be migrated. This migration is not done automatically during the update process, but it can the done using the pure::variants integration menu in DOORS. During migration, the *pvRefID*s of the objects and outgoing links of the working copy and ancestor reference modules will be rewritten and the*pvRefIDScope*s will be initialized. This will be done based on the existing objects and links in the latest reference. If the corresponding object or link cannot be found in the lastest reference, the *pvRefID* will be kept and the *pvRefIDScope* will be initialized to the maximal scope number (2147483647). Since the object or link is missing in latest reference, the next update process will either remove this object or link, or in case of changed objects move it to the attic. As long as objects or links with that maximal scope exists, a new derivation from that working copy cannot be done.

Note

If a working copy with scopes (from a second or later derivation stage created with pure::variants >= 5.0.1) is updated using pure::variants < 4.0.23 or equal to 5.0.0, the *pvRefIDScope* will be ignored and potentially

incompatible *pvRefID*s will be used for the mapping and comparison. The result is that unwanted changes or removals will be done in the working copy. Beginning with pure::variants 4.0.23, the existence of *pvRefIDScope* will be recognized and the update process will be stopped with an error.

Update Algorithm

During the update process, the objects and outgoing links of the working copy module and the reference modules are compared. *pvRefIDScope* and *pvRefID* are used to identify, which objects and links correspond to each other in the three module versions. Using this three-way comparison, the addition, removal and change of objects can be reliably recognized. Depending on the result of the comparison, the changes done in the latest reference related to the ancestor reference can be automatically merged into the working copy.

Update Module Existence

If a module exists in the latest reference and the corresponding working copy module is missing, a new working copy module is created by cloning the latest reference module. There are two cases, when this happens: Either a new module was added to the configuration space and therefore is also added to the latest reference, or the working copy was manually removed by the user and should be recreated.

However, if a module only exists in working copy, i.e. when the module is removed from configuration space and therefore also from latest reference or it was added manually by the user, it will remain untouched in the working copy.

If the ancestor reference module is missing, it will be automatically re-created based on the existing or new latest reference module. Reasons for a missing ancestor reference module are a manual removal by the user or the changing of the reference repository path without moving the existing ancestor reference module to the new location. Note that in this re-creation case, changes between the lost ancestor reference and the latest reference will be recognized as local changes in the working copy.

Update Object Existence

The first step is to check if differences in the existence of the objects exist. Table 3, “Update of Objects” shows the possible comparison results for object existence. The first three columns show the existence state of the object in ancestor and latest reference, and working copy, respectively (X means exists, - means exists not). Since user-added objects in the working copy have no (valid) pvRefID, they are never mapped to objects in ancestor or latest reference and therefore are always handled as working-copy-only objects. So the case, where corresponding objects are added both in latest reference and in working copy, cannot occur. The change columns in the table show which changes has occured based on the ancestor reference. Considered changes are Add and Remove only. Attribute changes are not considered in this phase. The last column **Update action** defines the operations done in working copy during update.

Table 3. Update of Objects

	Ancestor reference	**Latest reference**	**Working copy**	**Change in latest reference**	**Change in working copy**	**Update action**
I	X	X	X	-	-	Update attributes
II	-	-	X	-	X added by user	No change
III	X	X	-	-	X removed by user	No change
IV	X	-	-	X removed	X removed by user	No change
Va	-	X	-	X added	-	Copy object X
Vb	-	X	- (previously removed by user)	X added	-	No change
VI	X	-	X	X removed	-	Remove object X, if attributes not changed

If the user has removed a formerly generated object (case III), this object will not be added again. Even if this object is removed (case IV) and later re-added (case Vb) in the latest reference, it will not be added to the working copy. The cases Va and Vb can be distinguished by the *pvGenerated* attribute. This behavior is applied to prevent a re-adding of the user-removed object during each new update.

An object, removed in the latest reference, will only be removed from the working copy, if there are no related changes. These changes are:

Attribute value changes	Removal of objects will also remove possible user changes of attribute values of these objects, which should be prevented. But only user changes have to be considered, which are not overwritten during each update. So only changes of attribute values are relevant, whose attributes are not in *Overwrite* mode. This holds also for object pictures.
Incoming links	Objects with incoming links cannot be removed, without removing the link before. If this link is user-added, its removal should be prevented.
User-added children	A generated object to be removed can contain user-added children objects, which would be also removed. This case should be also prevented.

Instead of being removed, these working copy objects are moved to a special *attic* section in the working copy module. There, the user can examine them and can remove them manually. For each moved object, the attic section will contain an additional parent object with the heading <NOT_REMOVED> <parent: [refID *refIDScope*/*refID* | ID *Absolute Number* | root]> containing the *refIDScope* and *refID* (if existing) or the *Absolute Number* of the original parent object. For objects on the first level, root is written instead. So it is easier to find the original location of the moved object.

The default behavior for removing objects is to delete and purge them (hard-delete mode). That is, when an object is removed, it is removed completely from the working copy module. If the corresponding object in the latest reference module is re-added again in one of the next updates, a new object with a new object ID (absolute number) is created. However, if *pvUpdateUseSoftDeleteOnly* is set in the working copy module, the purging of object is not done. The objects to remove are deleted only (soft-delete mode) and can be undeleted in later updates again. That is, after removing and re-adding, the object keeps its object ID.

Note

If the user calls *Purge* on a working copy module containing soft-deleted objects created by the update process, these soft-deleted objects are also removed completely. Re-added objects will get a new object ID in that case.

If a working copy with soft-deleted objects is used as a source for a new module-based variant derivation the soft-deleted objects will be ignored. They are not part of the new variants.

If objects (or one of its ancestors) are marked with *pvDoNotUpdate = true* in working copy, they are not updated. So if an object (tree) should be copied to a parent object in working copy, not to be allowed for update, the copy operation is not done. The same holds for removed objects: If the object to remove or its ancestors are not allowed to be updated, they are not removed.

Update Object Structure

Updating the position of objects in the object hierarchy is done using a two-way comparison and merge with the latest reference as master. That is, object positions of the working copy will be changed to object positions of the latest reference. So during the update process, object moves in latest reference will be applied to the working copy and object moves done by the user in the working copy will be reverted.

Object moves will be prevented if the target container object (or one of its ancestors) is marked with *pvDoNotUpdate = true* in working copy.

Update Table Existence and Structure

In DOORS modules, tables are described by special objects, which are structured hierachically in three levels: The top level is defined by one *table* object, which defines the table itself. The second level contains *row* objects,

defining the rows of the table. And finally the bottom level contains *cell* objects, defining the cells of each row. A concept for columns does not exist in this hierarchy.

The updating of table related objects works in the same way as for standard objects as described above. So the updating process supports the adding and removal of tables, rows, and cells. But since rows and cells in a table cannot be moved in DOORS, it is not possible to move user-added rows and cells to attic objects in the working copy, when their container objects (table and row, respectively) have to be deleted to update to the latest state. If a table to be removed contains user-added rows or cells, it will be moved to the attic as a whole.

Update Outgoing Link Existence

The check of existence for outgoing links is similar to that of objects. As for objects of corresponding modules, the mapping of the links is done based on *pvRefIDScope* and *pvRefID* of the links of corresponding linksets. So links in non-corresponding linksets/link modules are never mapped to each other. Like user-added objects, user-added links have no *pvRefID*, but they can be mapped to generated links based on their source and target objects. So a user-added link in working copy is updated based on an existing equivalent generated link. Table 4, “Update of Outgoing Links” shows the comparison result and the update actions done for outgoing links. Like for objects, user-removed links are not re-added during update (case IV and VIb).

Table 4. Update of Outgoing Links

	Ancestor reference	Latest reference	Working copy	Change in latest reference	Change in working copy	Update action
I	X	X	X	-	-	Update attributes
II	-	-	X	-	X added by user	No change
III	-	X	X	X added	X added by user	Convert X to generated, update attributes
IV	X	X	-	-	X removed by user	No change
V	X	-	-	X removed	X removed by user	No change
VIa	-	X	-	X added	-	Copy link X
VIb	-	X	- (previously removed by user)	X added	-	No change
VII	X	-	X	X removed	-	Remove link X

Note

The recognition of removed links in latest reference (case VII) and therefore the removal of corresponding links in the working copy does not work a) if the removed link was removed because a complete link module or linkset was removed, or b) if all links of the removed link's link module are removed. In that cases, these links are not removed in the working copy.

If an object that should be removed contains outgoing links, these links will also be removed. Objects with incoming links cannot and should not be removed. Instead, such objects are moved to the attic structure if necessary.

Updating External Links

When updating external links, we distinguish between incoming and outgoing external links. Since there is no unique identifier for external links, we use the *link path* of the link to identify corresponding external links across latest reference and working copy. That is, during update for each object we support only one external link per link path and link direction.

External links, which exist in the latest reference and do not exist in the working copy, are copied from the latest reference to the working copy. In that case, the external link is also marked as generated using the *pvRefID* attribute.

If a corresponding external link exists in the working copy (either created during update or added by the user), it will be updated, i.e., its properties will be overwritten by the values of the corresponding external link in the latest reference.

Any external links in the working copy, which are not created by the user, are removed, if there is no corresponding external link in the latest reference anymore. However, external links that are created by the user will never be removed.

Updating Attribute Types and Definitions

Attribute types and definitions, which exist in the latest reference and do not exist (yet) in the working copy, are copied from latest reference to working copy.

If in contrast an attribute type or definition exists already both in latest reference and working copy, the working copy's attribute type or definition will be updated. This update is limited to following use cases:

Module and object attribute definitions	If in latest reference a module (or module & object) attribute definition exists and in working copy the corresponding attribute definition applies only to objects, the working copy's attribute definition will be extended to a module & object attribute definition. Respectively, if in latest reference an object (or module & object) attribute definition exists and in working copy the corresponding attribute definition applies only to modules, the working copy's attribute definition will be also extended to a module & object attribute definition.
Multi-value enumeration attribute definitions	If in latest reference a multi-value enumeration attribute definition exists and in working copy the corresponding attribute definition is only single-valued, the working copy's attribute definition will be updated to a multi-value attribute definition.
Enumeration types	If enumeration types with the same name but with different values exist in latest reference and working copy, the working copy's enumeration type will be extended by the values existing only in the latest reference. So the updated working copy enumeration type contains the union of the values of latest reference's and original working copy's enumeration types.

If there are incompatible changes in the attribute definition, e.g. its type is changed from *string* to *integer*, an update of the attribute definition itself and also of the attribute values will not be done.

Attribute types and attribute definitions, which are added by the user to the working copy only, are not changed. User changes in attribute types or definitions in working copy are, except for the use cases mentioned above, not reverted or modified during the update.

Updating Attribute Values

The updating of attribute values is done for module attributes, object attributes (and therefore also for table, row, and cell attributes), linkset attributes, and link attributes, as long as the considered attribute container exists both in working copy and latest reference, and their attribute types and definitions are not incompatible. Link module attributes are not updated.

Table 5, “Update of Attribute Values” shows on the left side the different compare results. A, B, and C represent different values of the attribute. The compare result can be:

Up-to-Date	The value is the same in both references and in working copy.
Local Change	The value was changed in working copy only.
Reference Change	The value was changed in latest reference only.
Concurrent Change	The value was changed concurrently in both latest reference and working copy each to different values.

Same Change The value was changed concurrently in both latest reference and working copy each to the same value.

As result, different update strategies can be applied to the attribute values, whose result is shown in the right side of the table:

Overwrite mode The value in working copy will always be overwritten.

Keep mode The value in working copy will never be changed.

Update mode The value in working copy will be changed only if a *Reference Change* was recognized in the comparison. So user-changed attribute values will not be overwritten.

Table 5. Update of Attribute Values

	Ancestor reference	**Latest reference**	**Working copy**	**Compare state**	**Action in Overwrite mode**	**New state**	**Action in Keep mode**	**New state**	**Action in Update mode**	**New state**
I	A	A	A	Up-to-Date	-	Up-to-Date	-	Up-to-Date	-	Up-to-Date
II	A	A	C	Local Change	Overwrite	Up-to-Date	-	Local Change	-	Local Change
III	A	B	A	Reference Change	Overwrite	Same Change	-	Reference Change	Overwrite	Same Change
IV	A	B	C	Concurrent Change	Overwrite	Same Change	-	Concurrent Change	-	Concurrent Change
VI	A	B	B	Same Change	-	Same Change	-	Same Change	-	Same Change

Note

The *Update mode* is only available for module and object attributes.

To reflect the update result state for object attribute values to the user, the highest update state of the attribute values updated in *Update mode* is saved in the special object attribute *pvUpdateState*. The order of the states is (from lowest to highest): Up-to-Date, Same Change, Local Change, Reference Change, Concurrent Change. The update result states of object attributes in *Overwrite* and *Keep modes* are not reflected in the attribute *pvUpdateState*. As described before, objects in working copy marked with *pvDoNotUpdate = true* will not be updated. The value of *pvUpdateState* is set to *Up-to-Date* in that case.

Analogously, the update result state for module attributes is reflected in the special module attribute *pvUpdateStateModAttr*.

The update mode can be defined during the import and synchronisation of DOORS models in pure::variants. It can be specified for each single module and object attribute. Additionally, a default mode can be defined per module. It is used for module and object attributes, if not defined explicitely for the single attribute. For linkset attributes and for link attributes for outgoing links this default setting is used in general. Since the *Update* mode is not supported for these attributes, the *Update* mode is interpreted as *Keep* mode in that case.

Updating Object Pictures

In DOORS each object can contain a *picture*, which is usually added using menu **Insert > Picture...** in the DOORS module window and which is usually shown in the view's main column below the object heading and text. These

pictures are distinct from OLE objects, which are embedded directly in object text attribute values and can contain also image data.

Although the object's picture is not defined by an object attribute, pure::variants handles it as a virtual object attribute. That is, pictures are handled as each other attribute as defined in the section called “Updating Attribute Values”, whereas the picture's virtual attribute value is defined by the binary data of the picture. That is, the three-way comparison is done using the picture's binary data.

To control the update strategy for pictures, the read-only system attribute *Picture* is used as the representative for pictures during the import and synchronisation of DOORS models in pure::variants. For pictures all three update modes (*Update*, *Overwrite*, and *Keep*) are supported. If the *Update mode* is used for pictures, the *pvUpdateState* will also reflect the picture's comparison result.

Limitations

- All current releases of DOORS client 9.7 have an issue with text attribute values that contain certain *static* OLE content. During transformation this manifests in a high memory consumption and possible crashes of the DOORS client. This DOORS issue can also result in data loss, like cleared attribute values in the working copy. For that reason the DOORS update transformation creates warnings when attribute values with static OLE content will be copied. It is strongly recommended to replace such static OLE content by embedded OLE content.

- In update mode, actually unchanged text attribute values with embedded OLE content can result in the unwanted recognition of changes. This is caused by changes in the RTF representation of the OLE object, which is used for comparison.

- The updating of DOORS link set pairings (also known as link module descriptors) is not supported. But to avoid errors during creation of new links in the working copy, the link set pairing exclusive mode is deactivated temporarily during the update.

- The updating of DOORS discussions is not supported.

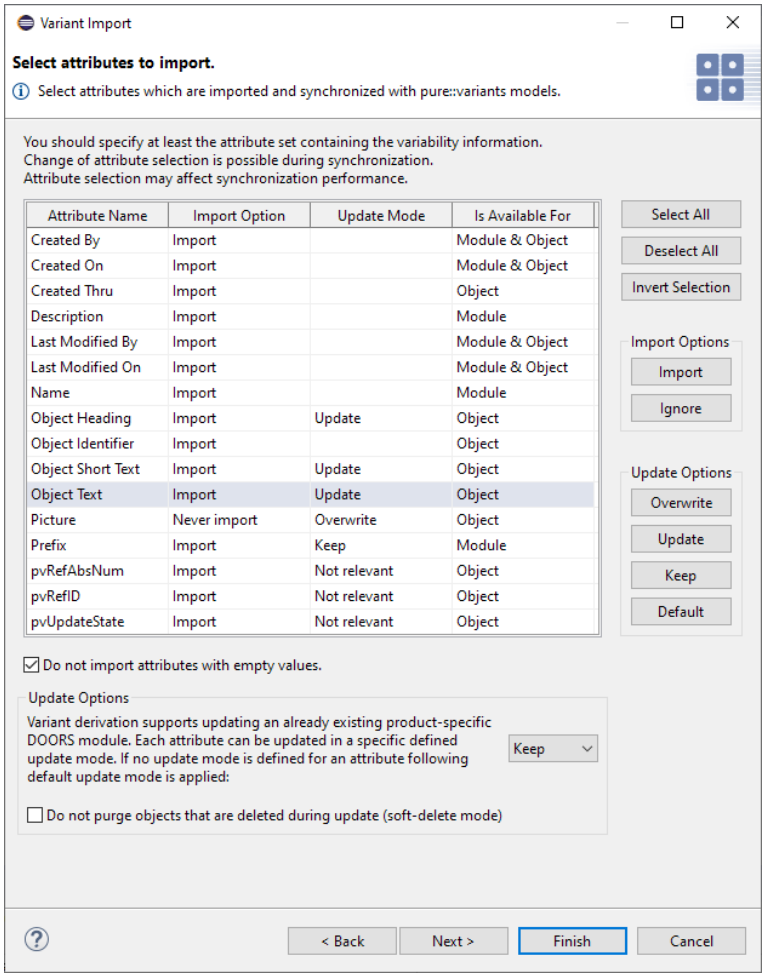
Usage

This section describes how the DOORS transformation with update support is used.

Defining Attribute Update Mode for Modules

During DOORS model import the update mode can be defined on the wizard page **Select DOORS attributes to import/synchronise for model** (see Figure 45, “Set of DOORS Attributes to Import”).

Figure 45. Set of DOORS Attributes to Import



For module and object attributes an update mode can be defined using the respective buttons (**Overwrite**, **Update**, **Keep**, and **Default**). **Default** means to use the module default defined using the combobox in the lower part of the wizard page. For such attributes the cells in the attribute table are empty. For special update-related module and object attributes, like *pvRefID*, the definition of an update mode does not make sense. For that reason the update mode is not changeable for them. They are marked as **Not relevant** in the attribute list.

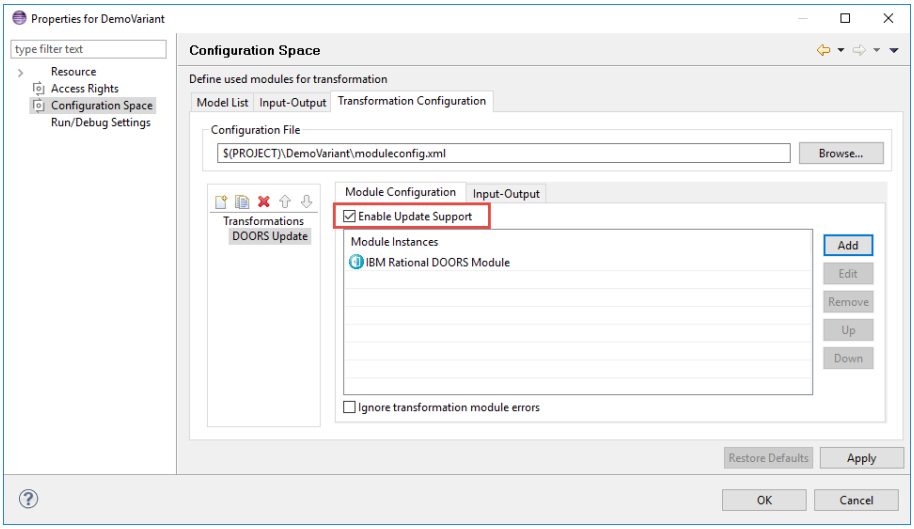
With *Do not purge objects that are deleted during update (soft-delete mode)* the soft-delete mode can be switched on.

During DOORS import model synchronization, the update mode settings can be changed. The changed settings are used during the next DOORS transformation run. Note: The soft-delete mode can be switched on any time. But once a working copy module was created or updated in this mode, switching it off is ignored in further updates.

Configuring Transformation for Update Support

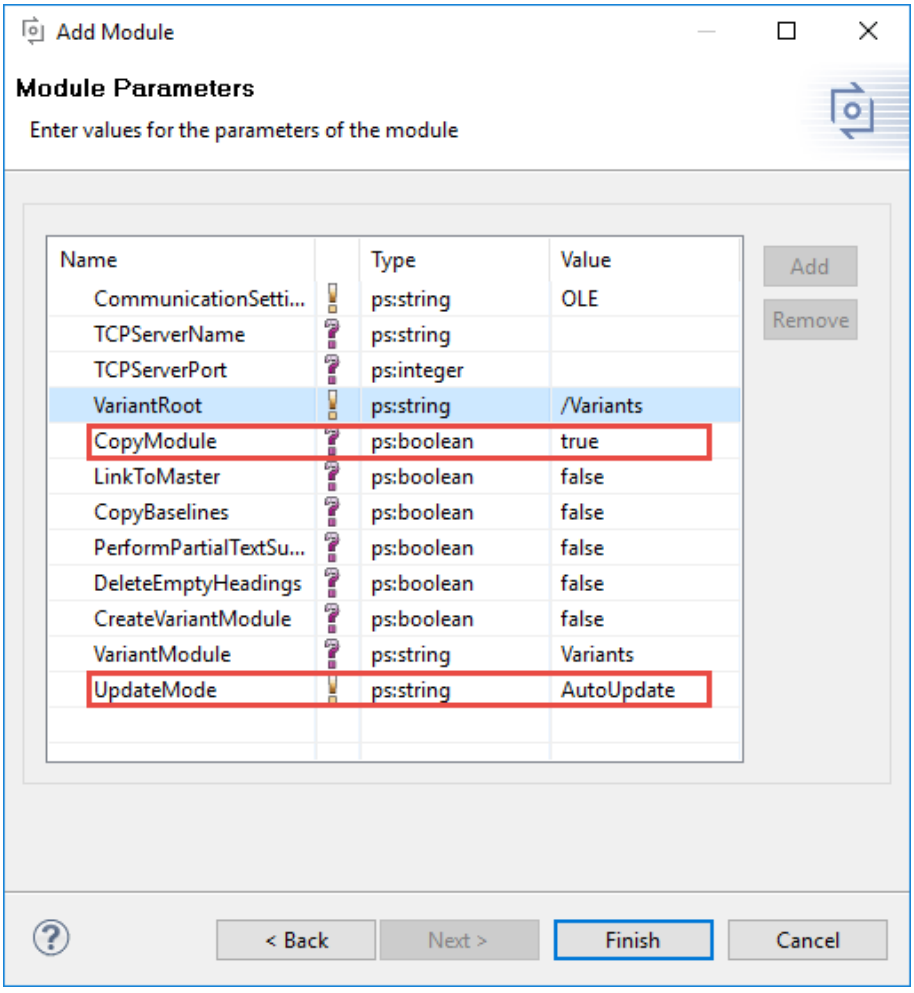
When adding the DOORS transformation in the **Transformation Module Configuration** the checkbox **Enable Update Support** has to be checked.

Figure 46. Transformation Module Configuration for Update Support



The DOORS transformation has to be configured to create module-based variants, i.e., transformation parameter **CopyModule** has to be set to true.

Figure 47. Module Parameter Page Setting for Update Support



The DOORS transformation knows two module update modes, which can be set with parameter **UpdateMode**:

AutoUpdate (default)	Each module processed by the DOORS transformation is updated automatically during the transformation according the rules described above.
ManualUpdate	During the DOORS transformation only the needed reference modules are created. The update itself has to be triggered manually per module by the user using the pure::variants DOORS integration.
OnlyUpdateLinks	During the transformation only the links of all the transformed modules in the already existing set of latest modules are updated in the working copy.
OnlyUpdateObjects	During the transformation only the objects of all the transformed modules in the already existing set of latest modules are updated in the working copy.

Creating Working Copy

If the working copy doesn't exist already, i.e., the intended output folder in the DOORS database does not contain any modules or folders, the DOORS transformation with enabled update support will automatically create a new working copy and will also create the needed ancestor reference. The content of the working copy modules can now be changed by the user.

To revert all changes done in the working copy, the output folder for the working copy have to be cleared. Then, running the DOORS transformation creates a new working copy.

Updating Working Copy

If the (potentially user changed) working copy should be updated to a changed product line variant, the DOORS transformation will have to be run again. As a working copy exists already, the DOORS transformation creates a new latest reference module instead. If **AutoUpdate** is enabled in the transformation settings, the existing working copy will be updated automatically.

Note

If a working copy was created before without update support, it cannot be updated. The DOORS transformation with enabled update support fails in this case and an error box is shown that the current working copy is not updateable.

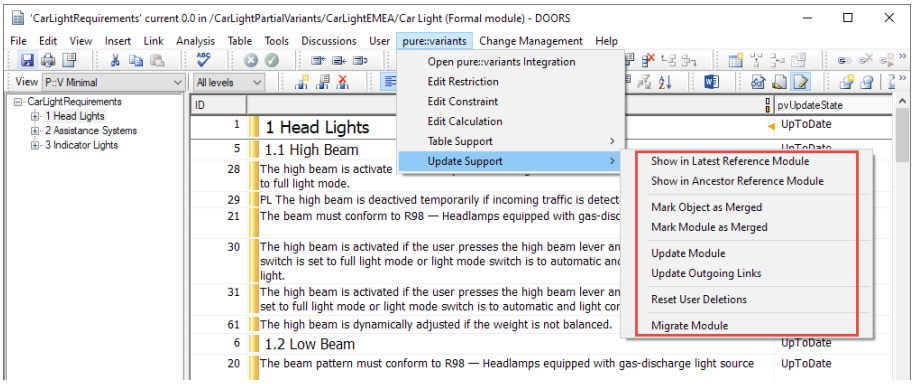
If the update support is disabled in the transformation configuration, the DOORS transformation will remove an eventually existing working copy and, since the reference modules are usually stored in the same output folder, will also remove the associated reference modules without user confirmation.

If working copy, latest reference and ancestor reference are incompatible relating the existence of pvRefID scopes, the update process will be stopped with an error.

Manual Updating Working Copy

If the DOORS transformation runs in **ManualUpdate** mode, the update will have to be started manually for each module. Therefore the module, which should be updated, has to be opened in DOORS. In the opened DOORS module window call **Update Module** in menu **pure::variants** to update objects and call **Update Outgoing Links** to update all links that have the opened module as source.

Figure 48. Update Support Submenu of pure::variants DOORS Integration Menu



These calls changes the working copy modules, but will not save them. So the change marks of DOORS can be used to review the object changes done during the module update. The changes can also be reverted by simply closing the respective module.

Note

As described above, incoming links from other modules will prevent the removal of the target objects in the update process. So it is necessary to update the outgoing links of the corresponding source module first. To ensure that, the user has to update the outgoing links of all modules first, before he can update the objects of all modules. However, since new links can only be added when the (new) source and target objects exist already, the user has to update the outgoing links of all modules again in a third step. These three steps are also considered in the automatic update process.

Reviewing Update Changes

In the next step each module has to be reviewed manually by the user and concurrent changes have to be resolved. To help to navigate between the working copy and the references, the pure::variants DOORS integration offers the functions **Show in Latest Reference Module** and **Show in Ancestor Reference Module** to find the correspondent object of the currently selected object in the reference modules.

A detailed list of changes done in the working copy during DOORS transformation can be found in the transformation log. The path of this log file can be configured in the properties of the configuration space in tab **Input-Output**.

When doing changes in the working copy to resolve conflicts or to revert local changes, the update states can be updated by running the manual module update using the integration menu action **Update Module**.

Note

If an attribute value is changed in working copy, whose update mode is *Overwrite*, this value will be overwritten using the **Update Module** call.

Mark Module as Merged

When the user has resolved all concurrent changes and has reviewed the working copy, he has to call **Mark Module as Merged** to confirm that all desired changes done in latest reference are applied to the working copy. Otherwise the changes of the current latest reference will reappear in the next update.

The result of this call is that the ancestor reference module corresponding to the current working module is replaced by the corresponding latest reference module. Before that, the user is asked whether the old ancestor module should be archived for later reviews. On confirmation the old ancestor module is stored, suffixed by the current timestamp, in the ancestor reference folder.

To reflect the change, the update states of the objects in the marked working copy module will be updated. Therefore the current working copy module has to be opened in exclusive edit mode. Since the latest reference module and the ancestor reference module are equal now, the new update states are either *Up-To-Date* or *Local Change*.

Note

If an attribute value is changed in working copy, whose update mode is *Overwrite*, this value will be overwritten during the **Mark Module as Merged** operation.

The action **Mark Module as Merged** has to be called for each updated working copy module. After that, the working copy can be edited again until the next update will be necessary.

Mark Object as Merged

Marking a complete module as merged confirms that all changes in the latest reference are confirmed. However, in some cases only a subset of changes should be confirmed at a certain time. For that cases the user can select the objects he want to mark as merged and call **Mark Object as Merged**. This also works for selected tables and table cells.

The result of that call is that the corresponding objects in the ancestor reference module are replaced by the corresponding objects in the latest reference module. If a corresponding object in the latest reference module was removed, the corresponding object in the ancestor reference module will also be removed. Eventually existing child objects in the ancestor reference module, which would inhibit the removal, will be moved one level higher before.

To reflect this change, the update states of the selected objects will be updated. This requires that the current working copy module is opened in exclusive edit mode.

Reset User Deletions

As described above, previously generated objects/links, which are removed by user are not re-added during the update process. The action **Reset User Deletions** will reset the list of user deletions, by recreating the content of *pvGenerated* attribute in the current module and in all linksets that have the current module as source. So, during the next update process the previously user-removed objects/links are re-added to the working copy again. Resetting single object or link removals is not possible.

Migrate Module

As described above, working copies of a second or later derivation stage created with pure::variants < 5.0.1 cannot be updated anymore beginning with pure::variants 5.0.1. Using action **Migrate Module** the migration of the module itself and of all linksets of outgoing links can be done. Before the migration is started, the user has to confirm the migration. The **Migrate Module** action saves all changes done during migration. If the migration fails, the DXL error log will show the reason. It is safe to run the migration several times, since it is checked whether the migration needs to be done and which parts (module or single linksets) needs still a migration.

3.6. Perform Custom DXL Script during transformation

The pure::variants - Connector for IBM Rational DOORS provides some hooks, where the user can introduce one custom DXL script into the transformation process. Depending on the transformation mode these hooks are:

Transformation Mode	**Hooks**
Module-Based Transformation	Before transformation starts. After modules are copied. Before update starts. This hook is available for enabled update mode only. After transformation finished.
Link-Based Transformation	Before transformation starts. After transformation finished.
Attribute-Based Transformation	Before transformation starts. After transformation finished.

The custom script may implement the following methods, depending on the hooks the script should use. If a hook is not applicable for the executed transformation mode, but the corresponding method is implemented, the hook is ignored during script execution. Methods not needed do not have to be implemented. It is allowed to have only the necessary methods in the custom script.

Hook	**DXL Method**
Before transformation starts.	preTransform()
After modules are copied.	postModuleCopy()
Before update starts.	preUpdate()
After transformation finished.	postTransform()

The methods are called by pure::variants adding some utility functions to the standard DOORS functions. All methods and variables used by pure::variants are prefixed with *pv*. So custom scripts should not use any variable name or method name starting with this prefix.

The file path of the custom script is added to the transformation using the transformation parameter *DXLScriptFile* (see the section called “Using Transformation to Export Variant”). The file encoding has to be UTF-8.

The following example is available in the context menu of the projects view. (New -> Custom DXL Script)

```
/**
 * Function triggered before the transformation starts.
 *
 * The function is always called.
 *
 */
void preTransform() {

}

/**
 * Function triggered after the modules are copied..
 *
 * The function is not called for transformation configurations,
 * which are not performing a module copy.
 *
 */
void postModuleCopy() {

}

/**
 * Function triggered before the update meachnism is triggered.
 *
 * The function is not called, if the transformation failed before update
 * or update is not enaled.
 *
 * This function is never called for the link based variant representation and
 * for the configuration export transformation.
 */
void preUpdate() {

}

/**
 * Function triggered after the transformation has run.
 *
 * The function is not called, if the transformation failed.
 *
 */
void postTransform() {
  int numberOfModules = pvGetNumberOfModules();
  pvLog("Number of transformed modules: " numberOfModules "");

  int index = 0;
  while(index < numberOfModules){
```

```
    pvLog("Path to Source Module: " pvGetSourcePath(index));

    Baseline b = pvGetSourceBaseline(index);
    if(null b){
      pvLog("Used Source Baseline: current");
    }
    else{
      pvLog("Used Source Baseline: " (major b) "." (minor b) (suffix b) "");
    }

    pvLog("Target Path: " pvGetTargetPath(index));
    pvLog("Latest Path: " pvGetLatestPath(index));
    pvLog("Ancestor Path: " pvGetAncestorPath(index));

    index++;
  }

  pvLog("Is Update Enabled: " pvIsUpdateEnabled() "");
  pvLog("Is Initial Transformation: " pvIsInitialTransformation() "");

  pvLog("Is Copy Based Transformation: " pvIsCopyBasedTransformation() "");
  pvLog("Is Link Based Transformation: " pvIsLinkBasedTransformation() "");

  // for link based transformation only
  pvLog("Path to Variant Module: " pvGetVariantModulePath());

  pvLog("Is Attribute Based Transformation: " pvIsAttributeBasedTransformation() "");

  // for attribute transformation only
  pvLog("Attribute Based Transformation Mode: " pvGetAttributeBasedTransformationMode());
  pvLog("Attribute name used in Attribute Based Transformation: "
    pvGetAttributeBasedTransformationAttributeName());
}
```

Provided DXL API

The following methods provide access to the modules, which are used in the current transformation.

void pvSendResponse()

Prepare and send response to pure::variants.

This adds infos, warnings, errors and logs to the response. The informations are processed by pure::variants.

This method is added automatically during script preparation by the DXLServer.

string pvGetProvidedData(int index, string dataArray[])

Gets the data specified by the index from the given string array.

If the index is out of bounds this method adds an error and returns *null*.

Parameter	Description
index	The index of the string to retrieve from the array.
dataArray	The array to retrieve the data from.

Baseline pvGetProvidedBaselineData(int index, Baseline dataArray[])

Gets the data specifies by the index from the given Baseline array.

The method returns the baseline from the index or *null*.

Null represents the current state of the module.

If the index is out of bounds this method adds an error and returns *null*.

Parameter	Description
index	The index of the baseline to retrieve from the array.
dataArray	The array to retrieve the data from.

Baseline pvGetSourceBaseline(int index)

Get the Baseline with the given index.

Parameter	Description
index	The index of the baseline to retrieve.

ModName_ pvGetSourceModName(int index)

Get the ModName_ of the source module with the given index.

Parameter	Description
index	The index of the ModName to retrieve.

ModName_ pvGetTargetModName(int index)

Get the ModName_ of the target module with the given index.

May be null, if the module does not yet exist.

Parameter	Description
index	The index of the ModName to retrieve.

ModName_ pvGetWorkingCopyModName(int index)

Get the ModName_ of the working copy module with the given index.

May be null, if the module does not yet exist.

Parameter	Description
index	The index of the ModName to retrieve.

ModName_ pvGetLatestModName(int index)

Get the ModName_ of the latest module with the given index.

May be null, if the module does not yet exist.

Parameter	Description
index	The index of the ModName to retrieve.

ModName_ pvGetAncestorModName(int index)

Get the ModName_ of the ancestor module with the given index.

May be null, if the module does not yet exist.

Parameter	Description
index	The index of the ModName to retrieve.

string pvGetSourcePath(int index)

Get the path of the source module with the given index.

Parameter	Description
index	The index of the source module path to retrieve.

string pvGetTargetPath(int index)

Get the path of the target module with the given index.

May be null.

Parameter	Description
index	The index of the target module path to retrieve.

string pvGetWorkingCopyPath(int index)

Get the path of the working copy module with the given index.

May be null.

Parameter	Description
index	The index of the working copy module path to retrieve.

string pvGetLatestPath(int index)

Get the path of the latest module with the given index.

May be null.

Parameter	Description
index	The index of the latest module path to retrieve.

string pvGetAncestorPath(int index)

Get the path of the ancestor module with the given index.

May be null.

Parameter	Description
index	The index of the ancestor module path to retrieve.

int pvGetNumberOfModules()

Get the number of transformed modules.

bool pvIsUpdateEnabled()

Returns true, if the update mode is enabled in this transformation.

bool pvIsInitialTransformation()

Returns true, if the update mode is enabeld and this run of the transformation is the initial run.

In the initial run the update instrumentalisation is performed.

string pvGetVariantName()

Get the name of the variant model.

If instances are used in the variant model the name returned here is the name of

the top level variant model.

string[] pvInstanceNames

Array of type string providing all the instance names. The first entry is the top level variant name followed by the instances down to the currently transformed instance.

string pvGetVariantModulePath()

Get the path of the variant module used for the link based transformation.

May be null.

ModName_ pvGetVariantModulePath()

Get the ModName_ of the variant module used for the link based transformation.

May be null, if the module does not yet exist.

string pvGetAttributeBasedTransformationMode()

Get the mode of the attribute based transformation.

The value is either "*Variant Matrix*" or "*Variant Enumeration*".

string pvGetAttributeBasedTransformationAttributeName()

Get the variant enumeration attribute name or the prefix for the variant matrix attribute names.

If nothing is defined in the transformation configuration the default value is returned.

bool pvIsCopyBasedTransformation()

Returns true, if copy based transformation is performed.

bool pvIsLinkBasedTransformation()

Returns true, if link based transformation is performed.

bool pvIsAttributeBasedTransformation()

Returns true, if attribute based transformation is performed.

Module pvOpenModule(ModName_ modRef, bool readMode, bool visible)

This method open a module and returns the module handle.

Parameter	Description
modRef	The ModName_ of the module to open.

Parameter	Description
	If the given ModName_ is null, or points to an delete module, this method raises an error.
readMode	If parameter readMode is true, the module is opened in read-only mode, the module is opened in exclusive edit mode otherwise.
visible	If parameter visible is true, the module is opened visible to the user, invisible otherwise.

Module pvOpenBaseline(ModuleVersion modVers, bool visible)

This method opens the given baseline and return the module handle.

The returned module handle is read-only.

Parameter	Description
modVers	The ModuleVersion of the module to open. If the given ModuleVersion is null, this method raises an error.
visible	If parameter visible is true, the module is opened visible to the user, invisible otherwise.

void pvCloseModules()

This method closes all modules, which were opened by the pvOpenModule() and pvOpenBaseline() methods.

void pvError(string msg)

Adds an error, which will be send to pure::variants.

The script execution is NOT stopped.

But the transformation will fail, after the script finishes.

void pvWarning(string msg)

Adds a warning, which will be send to pure::variants.

The script execution is NOT stopped.

The transformation will NOT fail, the warning is shown in the transformation result dialog after the transformation finished.

void pvInfo(string msg)

Adds an information, which will be send to pure::variants.

The script execution is NOT stopped.

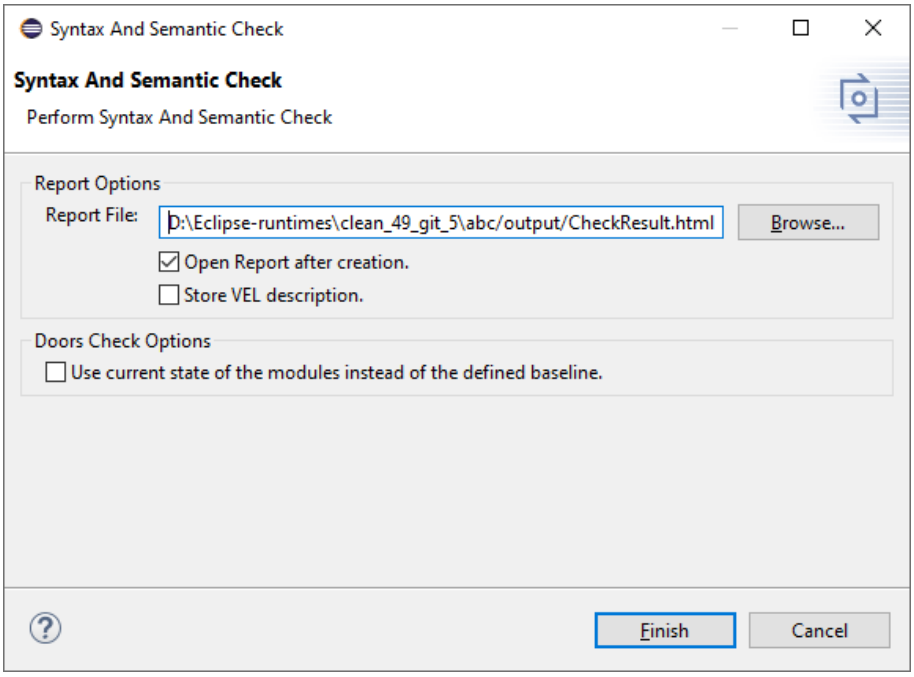
The transformation will NOT fail, the information is shown in the transformation result dialog after the transformation finished.

3.7. Checking all Doors modules connected to one Configuration Space for semantic and syntactic problems

Before transforming a lot of modules it is possible to check all pvscl rules in these modules. This ensures the transformation will not fail due to problems with the pvscl rules or a misconfiguration. To use this functionality choose **Perform Syntax and Semantic Check** from the context menu of the config space or the context menu of a selection of variant models.

A dialog pops up. In the dialog specify the options and an output path to the result report and click finish. The check now imports the variability information form DOORS to a VEL model and checks all the pvscl rules. Afterwards all selected variant models are evaluated to make sure there is no misconfiguration.

Figure 49. Syntactic and Semantic check dialog.



With the result report a log file is written. This contains detailed information if the process fails.

With enabeling the option *Store VEL description* the imported VEL descriptions are stored into the same folder as the report is stored. The VEL XML files can be imported to pure::variants for further analysis of the problems.

4. Troubleshooting

4.1. General

Locked DOORS Modules

Sometimes locked modules are reported by pure::variants, but DOORS does not show locked models and a second attempt succeeds. In general locked modules are listed in DOORS. Use **Manage Open Modules** in the **Tools** menu, to list the lock state of open modules.

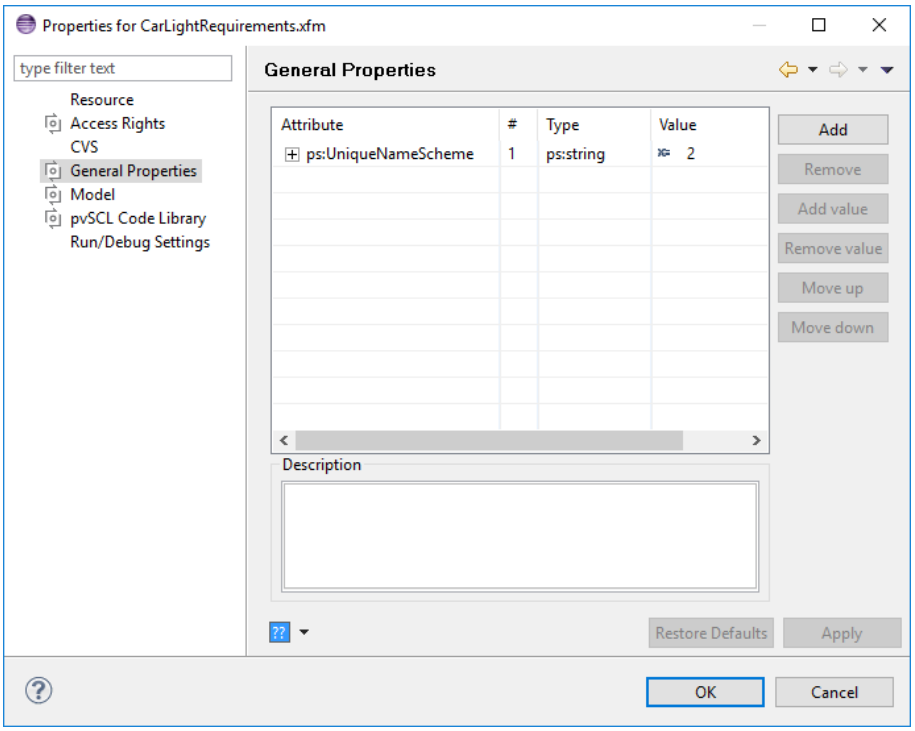
If **Manage Open Modules** does not show any locks you need to restart the DOORS client.

Unique Names are Too Long

Starting with pure::variants 4.0.5 the unique name generation was changed. Family model elements get no unique name now, since it is not necessary. In feature models the imported element get a generated unique name as before, but shortened to 100 characters. Existing models are not switched to the new unique name scheme automatically to prevent changing all unique names.

If the new unique name scheme should be used, a model property has to be added to the existing models. The name of the model property is *ps:UniqueNameScheme* and the value has to be *2*. Model properties are added using the properties dialog of the model.

Figure 50. Unique Name Scheme Property



Doors client is running out of memory

If during the transformation the DOORS client reaches about 3GB memory consumption the Client may stop working with a memory access violation exception. To prevent this we support switching the transformation behavior from leaving the used modules open as long as possible to prevent reopening them several times to close the modules as soon as possible to stop the DOORS client to run out of memory.

Starting with pure::variants 4.0.17 you can define an environment variable *PV_DOORS_BIG_DATA_SUPPORT*. If this variable is set to true, then the behavior is switch to the memory saving behavior.

pure::variants is running into communication timeout

While transformation of many modules or bis modules the Doors connector may run into a communication timeout. This can be solved in two ways. The first is to open the Preferences of the Doors Connecotr and set the comminucation timeout to more then 240 seconds. We recommend to at least double the timeout each time a timeout occurrs.

Starting with pure::variants 5.0.6 you can define an environment variable *PV_DOORS_CONNECTION_TIMEOUT*. If this variable is set the specified timeout will be used by the Doors connector. This enables you to define a communication time out even if the transformation or synchronisation is triggered with an ANT task with random workspaces.

Doors crashes during script execution

A Doors crash may occur during scripts execution. This can have various reasons. In that case the transformation log is oncomplete since the log is written after the current DXL script was successful.

Starting with pure::variants 5.0.7 it is possible to use a different communication channel, which sends log entries directly to pure::variants using HTTP requests. So so log is complete even if the script execution does not succeed.

To enable it go to the **Preferences** menu in Eclipse. Unfold the items below **Variant Management**. Below the entry **Connector References** navigate to entry **Connector for DOORS**. Enable the option "Use HTTP channel for log and progress information".

4.2. Module Transformation with Update Support

If the transformation with update support fails, check following list of solutions for fixing:

Working copy module *path* could not be opened. Reference paths cannot be updated.	During instrumentation of the new generated latest reference module, in the corresponding working copy the paths to the reference modules are updated. This is needed only if the path to the reference modules is changed by the user. If these paths are not updated, the pure::variants DOORS integration menu actions could fail. The indicated working copy module cannot be opened. So please ensure that the working copy module is not locked by another user. After that re-run the transformation.
Failed to remove previous ancestor reference module *path*. Reason: *errormsg*	When the complete working copy is removed, the transformation will create a new working copy and also the initial ancestor reference. If there are still existing ancestor reference modules, they will be removed. If that removal fails, the transformation will also fail. When starting with a new working copy from scratch, not only remove the working copy itself, but also the corresponding references modules.
[Working copy|Latest reference|Ancestor reference] module *path* does not exist or is soft-deleted. [Module|Link] update cannot be done.	For update, a working copy, a latest reference, and an ancestor reference are needed. If one is missing the update cannot be done. Usually the transformation creates all needed modules, but in some cases this could not be done. One reason is that the indicated module was deleted, but not purged (also called soft-deleted). Purging of soft-deleted modules is not done automatically by the transformation. So please check if the indicated module is soft-deleted and purge it manually.
[Working copy|Latest reference|Ancestor reference] module *path* could not be opened. [Module|Link] update cannot be done.	For update, a working copy, a latest reference, and an ancestor reference are needed. If one cannot be opened the update will not be done. The indicated module cannot be opened. So please ensure that the module is not locked by another user. After that re-run the transformation.
[Working copy|Latest reference|Ancestor reference] module *path* cannot be used for [module|link] update, since special update related attributes are missing.	For update, the working copy, the latest reference, and the ancestor reference modules need special attributes (namely pvRefID and pvAbsNum). They are created automatically during instrumentation. If they are missing, e.g. they were removed accidently by the user, the update cannot be done. In that case the indicated module is broken and have to be recreated. If a latest reference module is broken, check the transformation log for an error. The latest reference module will be recreated during the next transformation run. If that does not help, contact the pure::variants support. If the ancestor reference module is broken, a recreation can be forced by removing or renaming the module manually. During the next transformation, it will be automatically recreated by copying the currently existing latest reference module. The changes between the old broken ancestor module and the existing latest reference module will not be available anymore for update. If the working copy is broken, a recreation can be forced by removing or renaming the module manually. During the next transformation, it will be recreated by copying the new generated latest reference module. Be aware that in that case all user-changes in the working copy will get lost!

Latest reference module *path* cannot be used for ancestor recreation, since special update related attributes are missing.	When a working copy exists and an ancestor reference is missing (e.g. removed by the user), the ancestor will be recreated by an already existing version of the latest reference module before the a new latest reference module is created. So changes done between existing and new latest reference can be used during update. If the latest reference is broken, this copying will not be done and the indicated error is returned. To fix this case, the indicated latest reference module have to be removed manually. In the next transformation run, a new latest reference module will be generated and the missing ancestor reference module will be created.

4.3. TCP/IP Communication

TCP/IP Connection between DOORS Server and pure::variants Fails

To test the connectivity, simply start the DOORS DXL server (p::v Connector) and try to stop the server using **Disconnect DOORS Server**  button in the pure::variants tool bar. If DOORS starts responding normally, communication from pure::variants to the DOORS Server is working properly. If not, please check the hostname of the DOORS machine and the value given in the pure::variants connection dialog match. When both are on the same machine,`localhost` should be used here. Match the DOORS port setting (shown in the server dialog box) with the setting in the Connector (default is 5093). The port setting for incoming connections must be different from the server port settings (default is 5092).

If the server disconnects, but no other operation like importing modules work, it is most likely that the DOORS server is not able to connect back to pure::variants. This might be a firewall issue or a misconfiguration of the network name service. DOORS gets the host name reported by Java on the pure::variants machine as host to connect to. Please check with your local network administrator, if your network setup has problems.

Stopping the DOORS DXL Server

If it is not possible to stop the DOORS DXL Server using the pure::variants toolbar button, there is no other way than to terminate the DOORS client using operating system commands such as ALT+F4 on Windows platforms.

5. Known Restrictions

- If the IBM Rational DOORS 32 bit and 64 bit client is installed on the same computer at the same time, there might be a problem that pure::variants does not recognize the used DOORS client. We recommend to not install both versions on the same computer.
- DOORS tables are currently not supported in the pure::variants preview. Tables are fully supported in import and transformation.
- OLE objects inside attribute values are currently not supported. They are ignored during import.
- Images are currently not supported. Only the string `"<picture>"` will be shown after import.
- If a calculation is escaped and font style is changed within the calculation and their calculation marker, the rule may not be ignored and thus evaluated, which may result in errors.
- Automatic login into DOORS using user name and password is not working. Users must manually login and start the server.
- TCP/IP communication only: DOORS is blocked when the DXL server is running. Starting another instance of DOORS permits simultaneous work without stopping the DXL server. This however might use two licenses of DOORS. Please check the license policy of your DOORS installation.
- The attribute `Object Number` representing the hierarchical position of the requirement in the DOORS module is not silently updated. Thus, every change of this number will show up in the Compare view during the update. The ObjectNumber cannot be excluded using the attribute ignore list. This will be changed in future versions.

- If attribute refinement with variation links in DOORS is used, pure::variants creates a special requirement hierarchy in order to maintain the semantic of this approach. The pure::variants user should not be able to change type or hierarchy of DOORS requirement elements, which are part of the attribute Refinement. This would break the semantic of the variation links and would prevent pure::variant from correctly export or transform the variant.

  At the moment it is impossible to prevent the user from changing the hierarchy or types of the imported DOORS requirement elements.

- If two modules, which are located in different folders, are linked through a link module of type One-to-One or One-to-Many, the module based export will fail. (see the section called “Module-Based Variant Representation”)

- Exporting modules, which are linked together and are located in different folders, may leave a lock on the ascending folders. This lock can only be removed by the user by restarting the DOORS client. This problem will not be solved, because it is a DOORS issue, which leaves lock on ascending folders, if module is unlocked.

- Searching in the pure::variants preview view may fail due to a limitation of the DOORS client. We use layout DXL to show a preview for attribute substitutions. But the DOORS client is not able to search in layout DXL. A DXL error is shown.

- Change Indicator and Link Indicator columns are not supported during transformation in DOORS version < 9.6.0. The columns will be created in the copied modules but will not have any functionality.