
pure::variants Connector for Source Code Management Manual

pure-systems GmbH

Version 5.0.11.685 for pure::variants 5.0

Copyright © 2003-2022 pure-systems GmbH

2022

Table of Contents

| | |
|--|---|
| 1. Introduction | 1 |
| 1.1. Software Requirements | 1 |
| 1.2. Installation | 1 |
| 1.3. About this manual | 1 |
| 2. Using Connector | 1 |
| 2.1. Starting pure::variants | 1 |
| 2.2. Import a Directory Tree into a Family Model | 2 |
| 2.3. Updating Models from Directory Tree | 5 |
| 3. Using Relation Indexer | 5 |
| 3.1. Adding the Relation Indexer to a Project | 6 |
| 3.2. The Relations to the Source Code | 6 |
| 3.3. The Preferences | 7 |

1. Introduction

pure::variants Connector for Source Code Management (Connector) enables developers to manage source code variability using pure::variants. The Source Code Management of pure::variants provides a flexible opportunity to synchronize directory structures and source code files easily with pure::variants models. Thereby variants management can be applied practically even to complex software projects. Furthermore connections between pure::variants features and source code may be managed easier with the builder and are highly accessible via the Source Code Management.

1.1. Software Requirements

The pure::variants Connector for Source Code Management is an extension for pure::variants and is available on all supported platforms.

1.2. Installation

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**).

1.3. About this manual

The reader is expected to have basic knowledge about and experiences with pure::variants. Please consult its introductory material before reading this manual. The manual is available in online help as well as in printable PDF format [here](#).

2. Using Connector

2.1. 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.

2.2. Import a Directory Tree into a Family Model

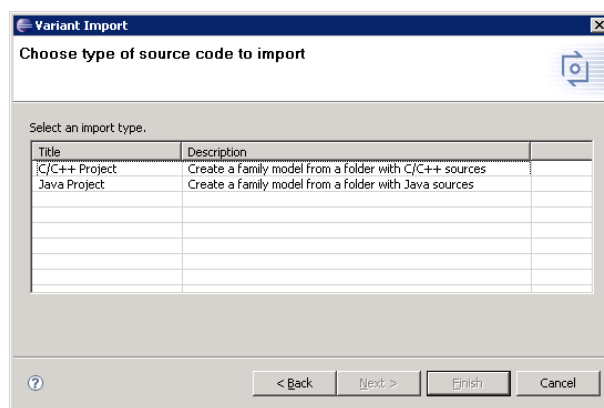
Before importing a directory tree into a Family Model, a variants project has to be created. Also it is suggestive to have features defined in an Feature Model already. Please consult the pure::variants documentation for help about these steps.

The actual import is started by selecting the Import... action either in the context menu of the Projects view or with **Import...** menu in the **File** menu. Select **Variant Models or Projects** from category **Variant Management** and press **Next**. On the following page select **Import a Family Model from source folders** and press **Next** again.

Choose type of source code to import

The import wizard appears (see [Figure 1, “Page of the import wizard to select the type of source code that may be imported”](#)). Select a project-type to import and press **Next**. Each type contains of a predefined set of file types to import to the model.

Figure 1. Page of the import wizard to select the type of source code that may be imported



Select Source and Target

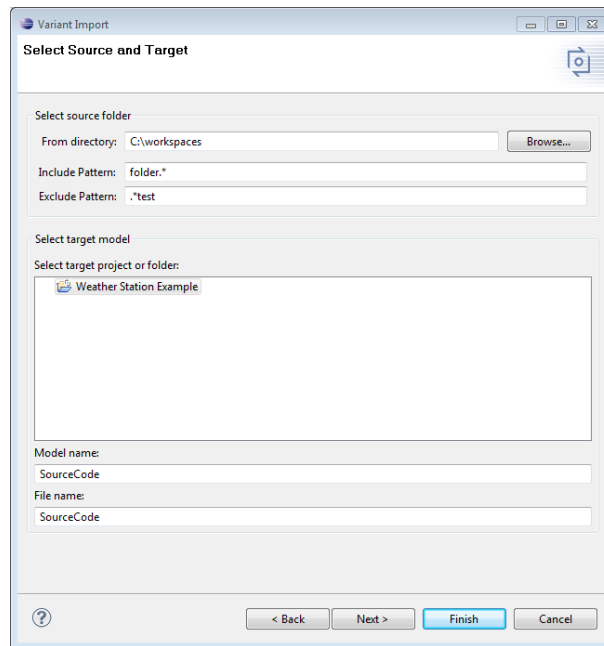
On the next wizard page ([Figure 2, “Page of the import wizard to select the source and the target for the import”](#)) the source directory and the target model must be specified.

Press the **Browse...** button to select the directory where the source code exists that should be imported. By default the current workspace is selected because this might be a useful point to start navigating.

Below you can specify include and exclude pattern. These pattern have to be java regular expressions. Each input path, relative to the source root folder, is checked with these pattern. If the include pattern match, a folder is imported, if the exclude pattern does not match. Meaning the include pattern does pre select the folders to import, the exclude pattern does restrict this preselection.

After selecting the source code directory a target model must be defined. Therefore select a variant project or a folder where the model should be stored and enter a model name. The file name is extended automatically with the `.ccfm` extension if it is not given in this dialog. By default it will be set to the same name as the model name itself. This is the recommended setting.

After an expedient source folder and the desired model name are specified, the dialog might be finished by pressing **Finish**. If the **Next** button is pressed, a further page is coming up where additional settings can be done.

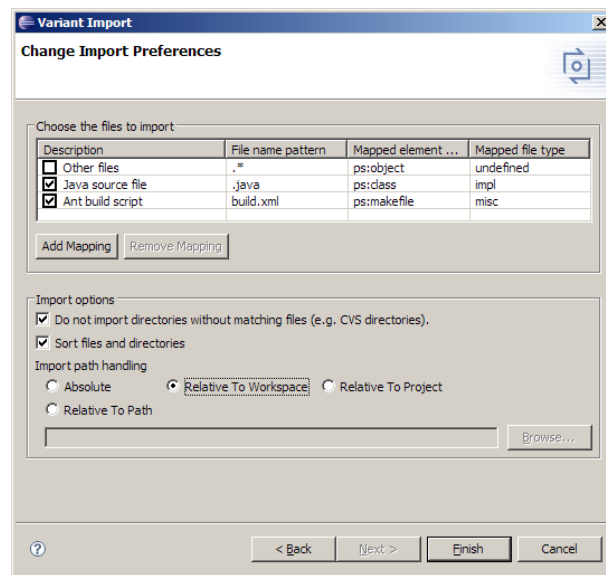
Figure 2. Page of the import wizard to select the source and the target for the import

Change Import Preferences

On the last wizard page ([Figure 3, “Page of the import wizard to define an individual configuration”](#)) there are preferences that can be done to customize the import behaviour for the imported software project.

The dialog page shows a table where the file types are defined, that will be considered by the import process. Each line consists of four fields.

- The **Description** field contains a short descriptive text to identify the file type.
- The **File name pattern** field is used to select files to be imported when they match to the fields value. The field uses the following syntax:
 1. The most common usecase may be a file extension. The usual syntax is `.EXT`, where `EXT` is the desired file extension (e.g. `.java`).
 2. Another common situation is a special file, like a makefile. Therefore, it is possible to match on the exact file name. To do this, just enter the file name into the field (e.g. `build.xml`).
 3. In some cases the mapping desires are more specific, so only files that match to a special pattern should be imported. To fit this requirement it is possible to use regular expressions in the **File name pattern** field. Describing the syntax of regular expressions would exceed the intend of this help. Please consult the regular expressions section of the reference chapter in the pure::variants user's guide (e.g. `.*`).
- The **Mapped element type** field sets the mapping between a file type and a pure::variants family element type. The family element type is a descriptor for the source file to provide further information to the mapped element in the imported model. Typical selections are `ps:class` or `ps:makefile`.
- The **Mapped file type** field sets the mapping between a file type and a pure::variants file type. The file type in pure::variants is a descriptor for the source file to provide further information to the mapped element in the imported model. Typical selections are `impl` for implementations or `def` for definition files.

Figure 3. Page of the import wizard to define an individual configuration

New file types may be added by using the **Add Mapping** button. All fields are filled in with the value `undefined` and must be filled in by the user. To edit a value in a field, just click into the field with the mouse. The value becomes editable and can be changed. It is not possible to change the default file name patterns of the table. To make a customization flexible, it is possible to deselect a file type by deselecting the row. Deselected file name patterns stay in the configuration but will not be used by the importer. User defined file types may be removed again by using the **Remove Mapping** button.

By default an **Other files** file name pattern is available in the table but deselected. Typically it is not wanted to import all files but this can be easily changed by selecting the according row.

There are three general import options to customize the behavior of the importer.

- **Do not import directories without matching files (e.g. CVS directories).**

If the importer finds a directory where no matching file is in it and where no subdirectory has a matching file, the directory will not be imported. This is often useful, if projects are managed by version management systems like CVS. For CVS, each relevant directory contains a CVS-directory where irrelevant files are stored. If this option is selected and the CVS-files do not match to any file type defined above, the directory will not imported as a component into the Family Model.

- **Sort files and directories.**

Enable this option to sort files and directories each in alphabetical order.

- **Import path handling.**

For further synchronization the importer needs to store the original path of all imported elements into the model. In many cases Family Models are shared with other users. The directory structure may be different for each user. To support most common usage scenarios the importer can work in different modes:

Absolute The absolute path to the imported element will be stored into the model. For later synchronization and during the transformation the files have to be placed on the exactly same location as during the first import.

Relative to Workspace The paths are stored relative to the workspace folder. For synchronization the files have to be part of the Eclipse workspace. The transformation has to use the Eclipse workspace as input directory.

Relative to Project The paths are stored relative to the project. For synchronization the files are part of the project inside Eclipse. The transformation has to use the project folder as input directory.

Relative to Path The paths are stored relative to the given path. For synchronization the files have to be placed on the exactly same location. The transformation input directory is the same as the relative path during the import.

All preferences of this dialog are stored persistently. The personal customizations must not redone each time the import runs. This makes the import workflow easy and fast.

2.3. Updating Models from Directory Tree


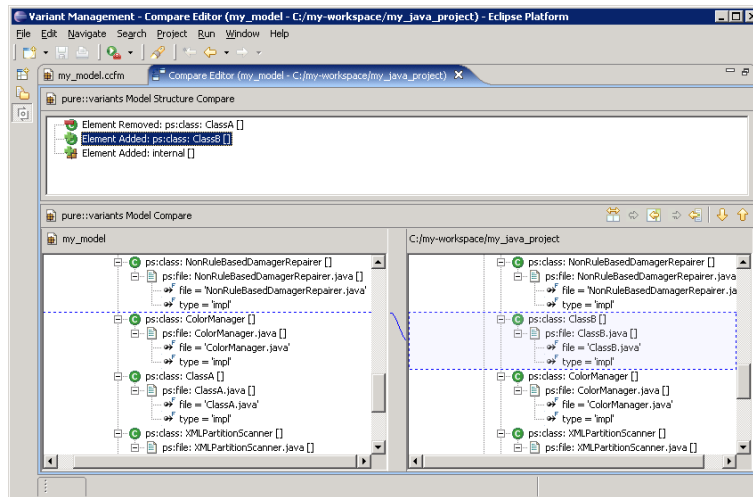
Press the **Synchronize** button  to synchronize an imported model with its directory path. The root path of the project is stored in the model so it will synchronize to the same directory as before. To enable the **Synchronize** button, open the model and select any element. After pressing the **Synchronize** button a *Compare Editor* is opened where the current Family Model and the model of the current directory structure is opposed (see [Figure 4, “Model update from Directory Tree in Compare Editor”](#)).

Figure 4. Model update from Directory Tree in Compare Editor



The compare editor is used throughout pure::variants to compare model versions but in this case is used to compare the physical directory structure (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, a added element is marked with a box on the right hand side and connected with its feasible position in the model on the left hand side.

The Merge toolbar between upper and lower editor windows provides tools to copy single or even all (non-conflicting) changes as a whole from the directory tree model to the Feature Model.

Note

The synchronization is done with the last used importer settings. This makes it possible to update the model with other settings as made while the import was done.

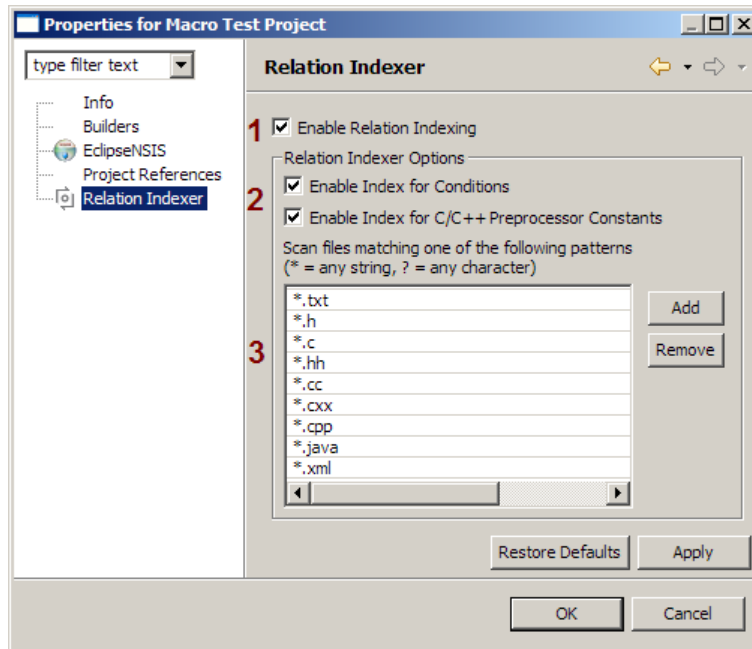
3. Using Relation Indexer

The Connector for Source Code Management enhances the **Relations View** with information about connections between pure::variants model elements and source code. Relations are added for features which are used in conditions of the `ps:condxml` and `ps:condtext` elements. For `ps:flag` and `ps:flagfile` elements the location of pre-processor constants in C/C++ source files are shown. In addition the locations of matching preprocessor constants are shown for a selected feature by using the mapping between feature unique names and preprocessor constants.

3.1. Adding the Relation Indexer to a Project

The relation indexer can be activated on a special project property page. Select the project and choose the **Properties** item in the context menu. In the upcoming dialog select the **Relation Indexer** page.

Figure 5. Project Property Page for the Relation Indexer



The relation indexer is activated for the project by selecting the **Enable Relation Indexer** option (1). After enabling the indexer there are some more options to define the project specific behavior. The indexing of **pure::variants Conditions** and **C/C++ Preprocessor Constants** can be activated separately (2). The list with file name patterns (3) is used to select the files for indexing. Only files which match one of the patterns are scanned. Add the "*" as pattern to scan all files of a project.

After activating the indexer for a project a builder is added to the project. This builder scans changed files for new relations to pure::variants model elements automatically.

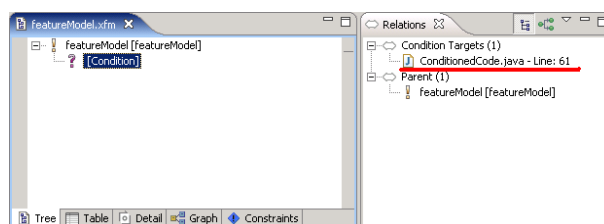
3.2. The Relations to the Source Code

With activated relation indexer the **Relations View** contains additional entries. These entries shows the name of the file and the line number of the variant point. The tool tip shows the appropriate section of the file. By double-clicking the entry the file will be opened into an editor.

pure::variants Conditions

The pure::variants condition can be used to include or exclude sections of a file depending on a feature selection. The **Condition Indexer** scans for such rules and extracts the referenced features. If such a feature is selected in the editor the **Relations View** will show all files and lines where a condition with the selected feature is located (see [Figure 6, "Representation of a Condition in the Relations View"](#)).

Figure 6. Representation of a Condition in the Relations View



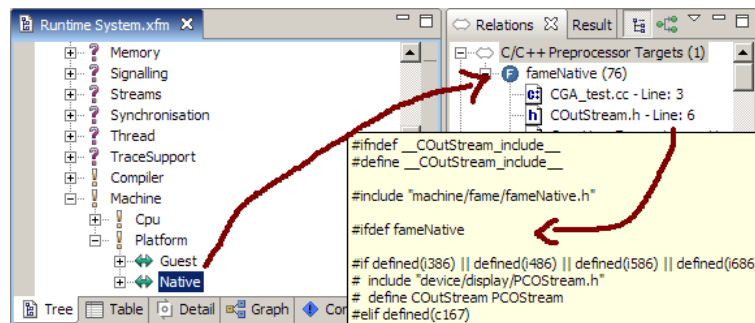
To get a detailed explanation on how to define conditions, consult the section `ps:condtext` of chapter 9.5.7 of the pure::variants User's Guide (*Reference-->Predefined Source Element Types-->ps:condtext*).

C/C++ Preprocessor Constants

The **C/C++ Preprocessor Indexer** scans files for constants used in preprocessor rules (e.g. `#ifdef`, `#ifndef`, ...). If a `ps:flag` or `ps:flagfile` element is selected the Relations View shows the usage of the defined preprocessor constant.

The Relations View also shows preprocessor constants connected to features by using mapping patterns. For this the patterns are expanded with the data of the selected feature. The resulting symbols are used to search for matching preprocessor constants. [Figure 7, "Representation of a C/C++ Preprocessor Constant in the Relations View"](#) shows an example with the pattern `fame{Name}`. The pattern is expanded with the unique name of the feature to `fameNative`. In the indexed code there are 76 locations where the preprocessor constant `fameNative` is used. This locations are shown in the **Relations View**. The patterns can be defined in the preferences (see [Section 3.3, "The Preferences"](#)).

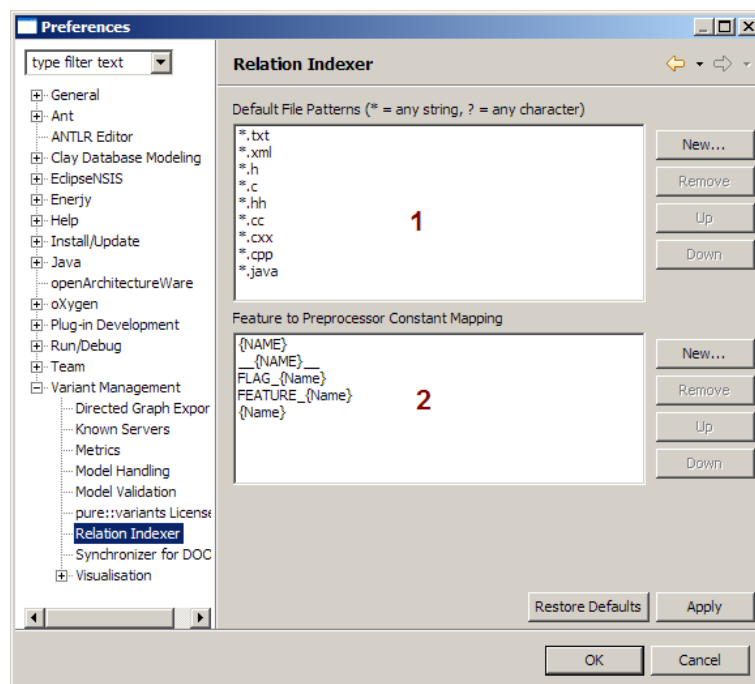
Figure 7. Representation of a C/C++ Preprocessor Constant in the Relations View



3.3. The Preferences

To change the default behavior of the indexer open the Eclipse preferences and select the **Relation Indexer** page in the **Variant Management** category. The page shows two lists.

Figure 8. Relation Indexer Preference page



The upper list contains the default file patterns for the indexer (1). This list is the initial pattern setting for newly enabled projects.

The lower list contains the mapping between features and preprocessor constants (2). This mapping is used for all projects. [Table 1, “Supported Mapping Replacements”](#) shows all possible replacements.

Table 1. Supported Mapping Replacements

| Wildcard | Description | Example: FeatureA |
|-----------------|--|-----------------------------|
| Name | the Unique Name of the selected feature | FLAG_{Name} - FLAG_FeatureA |
| NAME | the upper case Unique Name of the selected feature | FLAG_{NAME} - FLAG_FEATUREA |
| name | the lower case Unique Name of the selected feature | flag_{name} - flag_featurea |