

pure::variants Eclipse Plug-in User's Guide

Version 1.1 for pure::variants 2.0

pure::variants Eclipse Plug-in User's Guide : Version 1.1 for pure::variants 2.0

Published 2005

Copyright © 2003-2005 pure-systems GmbH

Table of Contents

1. Introduction	1
What is pure::variants?	1
Other related documents	2
2. Getting Started	3
Software Requirements	3
Software Installation	3
How to install the software	3
Installation Problems	3
Obtaining and Installing a License	4
The Variant Management Perspective	4
Using Feature Models	5
Using Configuration Spaces	5
Viewing and Exporting Configuration Results	7
Transforming Configuration Results	7
Additional pure::variants plug-ins	8
Exploring Documentation and Examples	9
3. Concepts	11
Introduction	11
Common Concepts in pure::variants models	12
Element Relations	12
Element Attributes	13
Feature Models	14
Feature Attributes	15
Feature Restrictions	15
Family Models	15
Structure of the family model	16
Sample family model	17
Restrictions in Component Family Models	18
Family Model Element Relations	19
Model Evaluation	20
.....	22
The XML Transformation Engine XMLTS	22
The Transformation Process	22
pure::variants Transformation Input	23
4. The Standard Transformation	25
Using the standard transformation	25
Standard Part Types	26
Standard transformation for source elements	27
Using XSLT to transform	31
Example: Conditional Document Parts	31
5. Graphical User Interface Elements	35
Getting Started with Eclipse	35
Variant Management Perspective	36
Editors	37
Common Editor Actions	37
Common Editor Pages	38
Feature Model Editor	43
Family Model Editor	46
Variant Description Model Editor	47
Compare Model Editor	49
Views	50
Attributes View	50

Filter View	51
Outline View	51
Problem View/Task view	51
Properties View	51
Relations View	52
Result View	53
Variant Feature Matrix view	54
Variant Projects View	55
Property Editors	55
Configuration Space	55
Model Export and Import	58
Export	58
Import	60
6. Reference	61
Abbreviations	61
Feature/Element Relations	61
General Model Restriction Language pvProlog	62
Additional Restriction Rules for Variant Evaluation	67
Match Expression Syntax for getMatchingElements	69
Model Attributes	70
Feature Models	70
Supported Feature Types	70
Family Models	71
Predefined Family Model Part Types	71
Family Model Element Relations	72
Variant Description Models	72
Feature Selection List Entry Types	72
XSLT Extension Functions	73
7. Appendices	77
Software Configuration	77
User Interface Advanced Concepts	77
Console View	77
Glossary	77
Index	79

List of Figures

1.1. pure::variants transformation process	1
2.1. Initial layout of the “Variant Management” perspective (“Variant Projects” view is shown)	4
2.2. A simple feature model of a car	5
2.3. Variant model with a problematic selection	6
2.4. Variant model export wizard (HTML export of all models selected)	7
2.5. Transformation configuration in configuration space properties	8
2.6. Transformation button in Eclipse toolbar	8
3.1. Overview of family-based software development with pure::variants	11
3.2. (simplified) element meta model	12
3.3. (Simplified) element attribute meta-model	13
3.4. Basic structure of feature models	15
3.5. Basic structure of component family and component models	16
3.6. Sample component family model	17
3.7. Model Evaluation Algorithm (Pseudo Code)	20
3.8. XML Transformer	23
4.1. The Standard Transformation Type Model	25
4.2. Multiple attribute definitions for Value calculation	26
4.3. Variant project describing the manual	32
4.4. The manual for users and developers	34
5.1. Eclipse workbench elements	35
5.2. Variant management perspective standard layout	36
5.3. Filter definition dialog	37
5.4. Sample attribute definitions for a feature	40
5.5. Restrictions page shown in property editor. Right picture	41
5.6. Restriction editor	41
5.7. Element selection dialog	42
5.8. Open feature model editor with outline and property view	44
5.9. Feature property dialog	45
5.10. Open family model editor with outline and property view	46
5.11. Open variant description model editor with outline and tasks view	48
5.12. Outline view showing the list of available features in a variant description model	48
5.13. Model Compare Editor	49
5.14. Attributes view (right) showing the attribute Count for feature Gears	51
5.15. Relations view (different layouts) for feature with a “ps:requires” to feature 'Main Component Big'	52
5.16. Result View	53
5.17. Result View in Delta Mode	54
5.18. Feature Matrix view of a configuration space	55
5.19. Configuration space properties: Model Selection	56
5.20. Configuration space properties: Transformation input/output paths	57
5.21. Configuration space properties: Transformation Configuration	57
5.22. Directed Graph Export Output Configuration Dialog	58
5.23. Directed graph export example (options LR direction, Colored)	59
7.1. The configuration dialog of pure::variants fig-configuration-dialog	77

List of Tables

5.1. Keyboard short cuts in feature model tree editor	44
5.2. Variables available for path resolution in transformations	56
6.1. Supported Relations between Features/Element	61
6.2. Logic operators in pvProlog	63
6.3. Functions in pvProlog	63
6.4. Available rules for Value calculations and Restrictions	63
6.5. Additional rules available for variant evaluation	67
6.6. Meta-Model attributes in pvProlog	70
6.7. Feature tree relation types and icons	70
6.8. Predefined part types	71
6.9. Supported Element Relations only supported in CCFM	72
6.10. Types of feature selections	73
6.11. XSLT extension functions	73

List of Examples

4.1. A sample conditional document for use with the ps:condxml transformation	29
4.2. Generated code for a ps:flagfile for flag "DEFAULT" with value "1"	29
4.3. Generated code for a ps:makefile for variable "CXX_OPTFLAGS" with value "-O6"	30
4.4. Generated code for a ps:classalias for alias "PConn" with aliased class "NoConn"	30

Chapter 1. Introduction

What is pure::variants?

The pure::variants Eclipse plug-in extends the Eclipse IDE to support the development and deployment of software product lines. Using pure::variants, a software product line is developed as a set of integrated models: Feature models describing the problem domain, Component Family models describing the problem solution and Variant Description models specifying individual products from the product line.

Feature models describe the products of a product line in terms of the features that are common to those products and the features that vary between those products. Each feature in a Feature model represents a property of a product that will be visible to the user of that product. These models also specify relationships between features, for example, choices between alternative features. Feature models are described in more detail in the Section Figure 3.4, “Basic structure of feature models”.

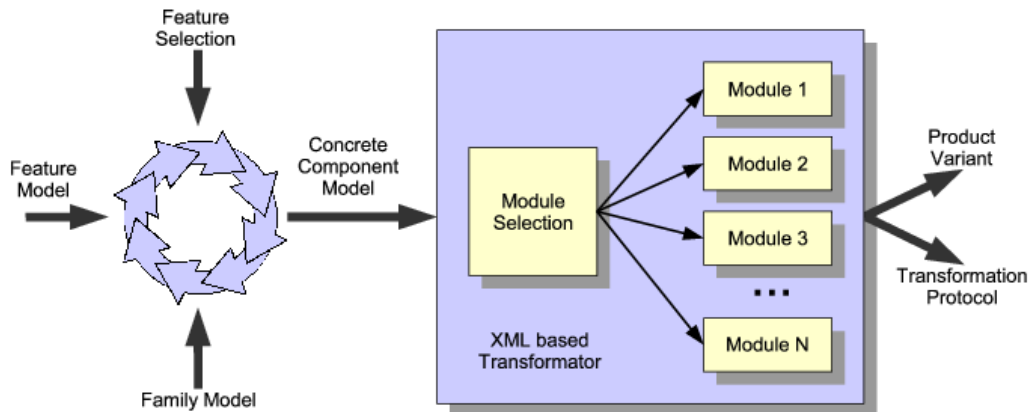
Component Family models describe how the products in the product line will be assembled or generated from pre-specified components. Each component in a Component Family model represents one or more functional elements of the products in the product line, for example software (in the form of classes, objects, functions or variables) or documentation. Component Family models are described in more detail in the Section the section called “Family Models”.

Variant Description models describe the set of features of a single product in the product line. Taking a Feature model and making choices where there is variability in the Feature model creates these models. Variant Description models are described in more detail in the Section ?

In contrast to other approaches, pure::variants captures the Feature model (problem domain) and the Component Family model (problem solution) separately and independently. This separation of concerns makes it simpler to address the common problem of reusing a Feature model or a Family model in other projects.

Figure 1.1, “pure::variants transformation process” gives an overview of the basic process of creating variants with pure::variants.

Figure 1.1. pure::variants transformation process



The product line is built by creating Feature and Component Family models. Once these models have been created, individual products may be built by creating Variant Description models. Responsibility for creation of product line models and creation of product models is usually divided between different groups of users.

Other related documents

The “Workbench User Guide” (“Help”->”Help Contents”) is a good starting point for familiarising yourself with familiar with the Eclipse user interface.

The pure::variants XML transformation system is described in detail in the XML Transformation System Manual (see Eclipse online help for a HTML version)

Features specific to the pure::variants Server Edition are described in a separate section in this document in the PVEP Plugins chapter (if reading the Eclipse help variant of the manual with an installed Server Edition feature) or in a separate PDF file.

The pure::variants Extensibility Guide is a reference document for information about extending and customizing pure::variants e.g. with customer-specific user interface elements or by integrating pure::variants with other tools.

Chapter 2. Getting Started

Software Requirements

The following software has to be present on the user's machine in order to support the pure::variants Eclipse plug-in:

Operating System:	Windows 2000, Windows XP, Linux or MacOS X 10.3
Eclipse:	Eclipse 3.0 or greater required. Eclipse is available from http://www.eclipse.org/ .
Java:	Eclipse requires a Java Virtual Machine (JVM) to be installed. We recommend using a Sun JDK 1.4 or 1.5 compatible JVM. See http://www.java.com/ for a suitable JVM.

Software Installation

How to install the software

pure::variants software is distributed and installed in one of three ways:

- *Installing from an Update site* Installation via the Eclipse update mechanism is a convenient way of installing and updating pure::variants from an internet site. The location of the site depends on the pure::variants edition, visit the pure-systems web site (<http://web.pure-systems.com>) or read your registration e-mail to find out which site is relevant for the version of the software you are using. Open the page in your browser to get additional information how to use update sites with Eclipse 3.0.
- *Archived Update Site* pure::variants uses now the format of archived update sites, distributed as ZIP files, for offline installation. pure::variants archived update site file names start with “updatesite” followed by an identification of the contents of the update site. Installation is almost identical to normal update site installation. Simply follow the instructions for normal update sites (above) but instead of using the “New Remote Site” button to navigate to and select the ZIP file the “Archived Site” button is used.
- *ZIP file* Some pure::variants extensions may be distributed in simple ZIP files instead as archived update site. To install such extensions unpack the contents of this file into the directory where Eclipse is installed. Additional Eclipse features and Eclipse plugins may also be installed during this process. (NB A “normal” ZIP distribution will no longer be provided for core pure::variants toolset.)

Installation Problems

If you experience problems when installing new or updated plug-ins it can help to remove any previous installation of the plug-in by removing all directories starting with *com.ps.consul* from the features and plug-ins subdirectories of your Eclipse installation.

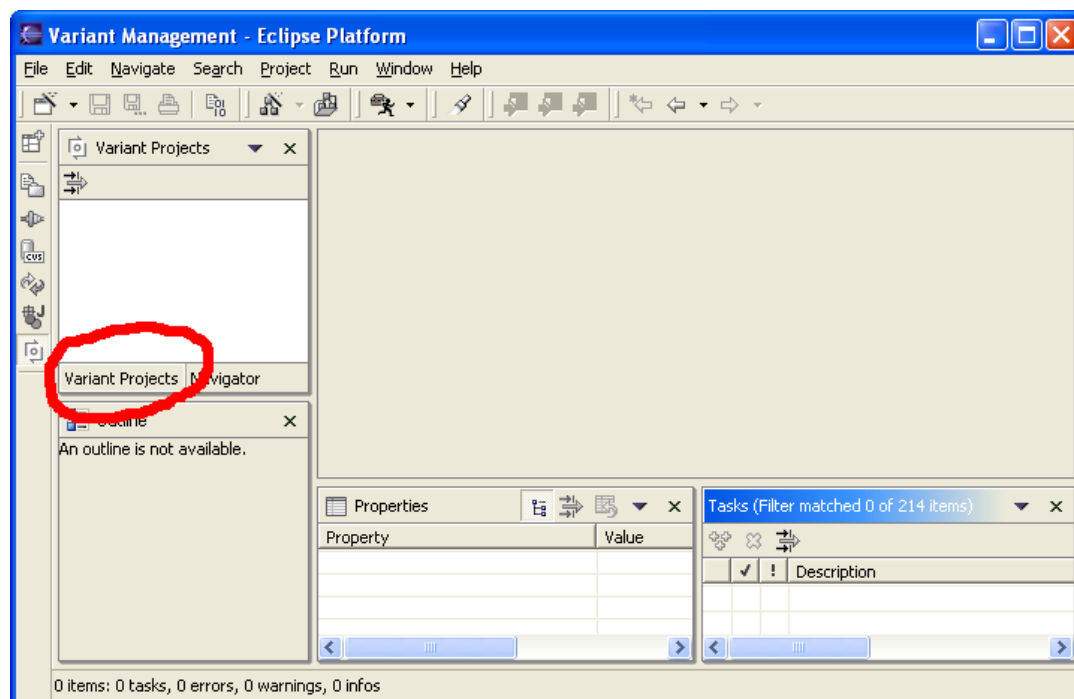
Obtaining and Installing a License

A valid license is required in order to use pure::variants. If pure::variants is started and no license is present, then the user is prompted to supply a license. By selecting the Request License button a software registration form is opened in the user's default web browser. After submitting the form, a license file is generated and sent to the e-mail address specified by the user. Select the Yes button and use the file dialog to specify the license file to install. The specified license will be stored in the current workspace. If the user has different workspaces, then the license file has to be installed again

The Variant Management Perspective

The easiest way to access the variant management functionality is to use the Variant Management perspective provided by the plug-in. Use Window->Open Perspective->Other and choose Variant Management to open this perspective in its default layout. The Variant Management perspective should now open as shown below.

Figure 2.1. Initial layout of the “Variant Management” perspective (“Variant Projects” view is shown)



Now select the Variant Projects view in the upper left side of the Eclipse window. Create an initial standard project using the context menu of this view and choose New->Variant Project or use the File->New->Project wizard from the main menu. The view will now show a new project with the given name.

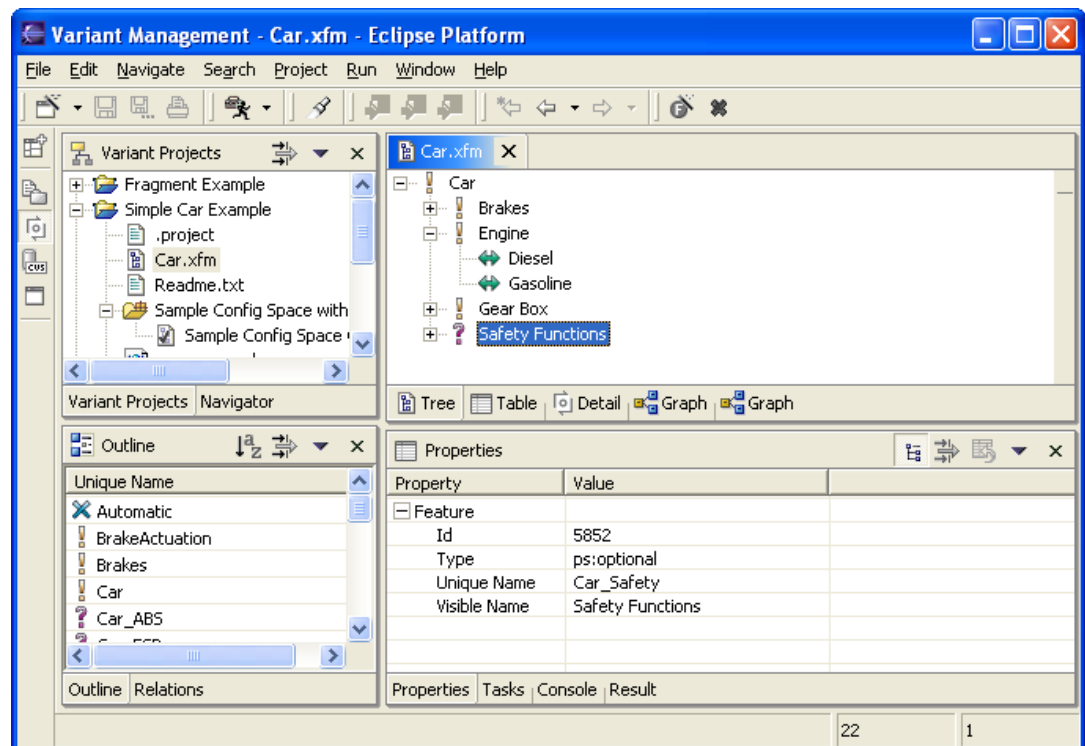
Once the standard project has been created, three editor windows will be opened automatically: one for the feature model, one for the family model and one for the variant description model.

Using Feature Models

When a new Variant project is created a new feature model is also created with a root feature of the same name as the project's name. This name can be changed using the properties dialog of the feature. To create child features, use the New entry of the context menu of the intended parent feature. A New Feature wizard allows a unique name, a visible name, and the type of the feature and other properties to be specified. All properties of a feature can be changed later using the Properties dialog.

The figure below shows a small example feature model for a car.

Figure 2.2. A simple feature model of a car



The outline view (lower left corner) shows configurable views of the selected feature model and allows fast navigation to features by double-clicking the displayed entry.

The properties view in the lower middle of the Eclipse window shows properties of the currently selected feature.

The Details tab of the feature model editor (shown in the upper right part) provides access to a different view on the feature model. This view uses a layout and fields inspired by the *Volere* requirements specification template to record more detailed aspects of a feature.

Using Configuration Spaces


In order to create Variant Description models it is first necessary to create configuration spaces. These are used to combine models for configuration purposes. The New->Configuration Space menu item starts the New Configuration Space wizard. Only the names of the configuration space and at least one feature model have to be specified. The

Viewing and Exporting Configuration Results

initially created standard project configuration space is already configured in this way.

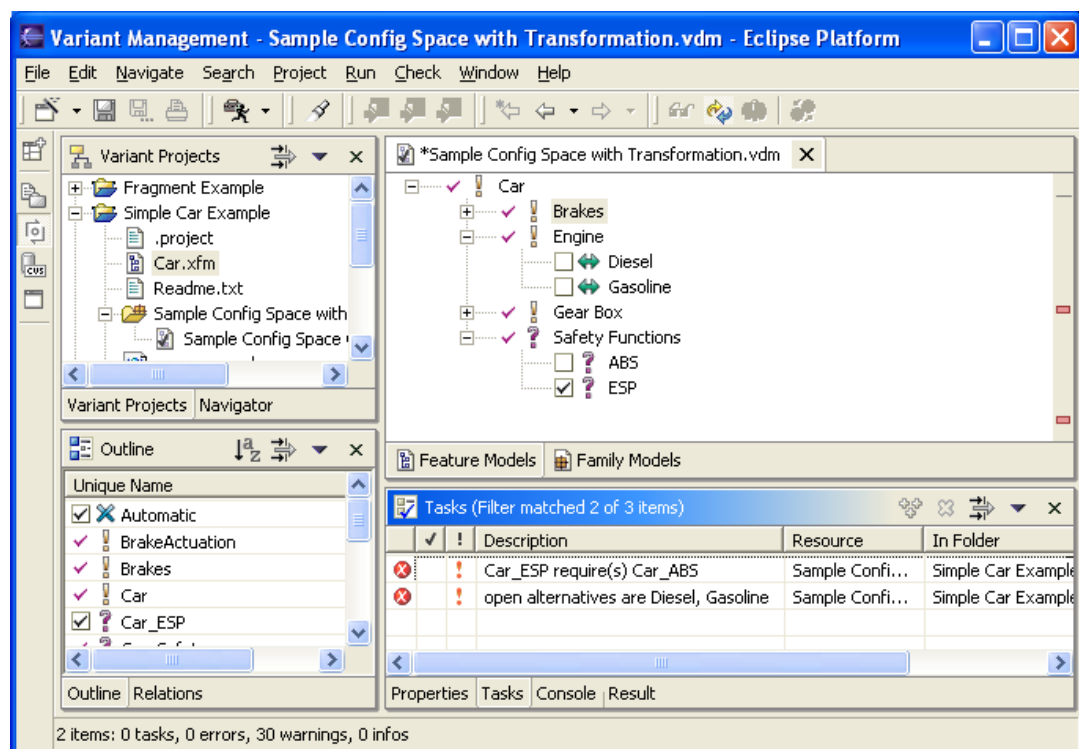
A variant model has to be created inside the configuration space for each configuration. This is done using the context menu of the configuration space.

The variant model editor is used to select the desired features for the variant. This editor is also used to perform configuration validation. The Check Model button on the toolbar, and the Variant -> Check menu item, are used to perform an immediate validation of the feature selection. The Variant->Auto Check menu item enables or disables automatic validation after each selection change. The Variant->Auto Resolve menu item enables or disables automatic analysis and resolution of selection problems.

The problem view¹ (lower right part) shows problems with the current configuration. Double clicking on a problem will open the related element(s) in the Variant Model editor. When used for the first time, Variant Management problems may be filtered out. To resolve this, simply click on the filter icon  and select “Variant Management Problems” as problem item to show. For some problems the “Quick fix” item in the context menu for the problem may offer options for solving the problem.

The figure below show an example of a problem selection.

Figure 2.3. Variant model with a problematic selection



The outline view shows a configurable list of features from all feature models in the configuration space.

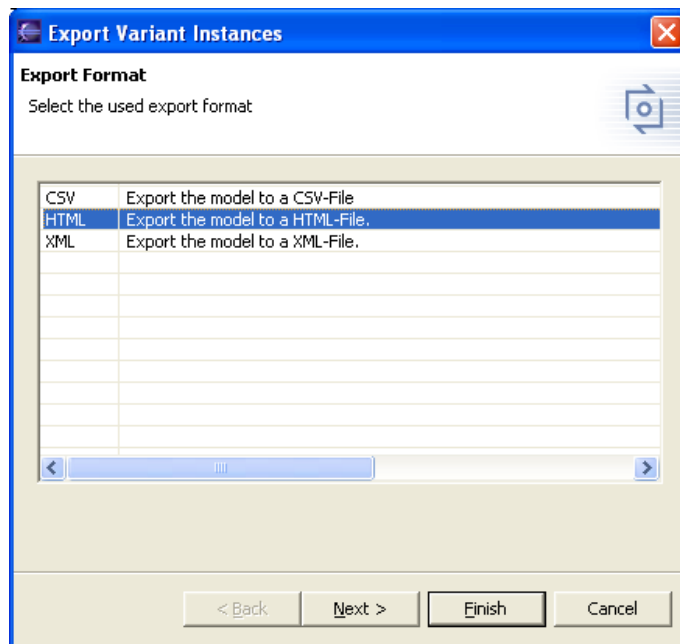
¹ Eclipse 3.0: The task view has been divided into tasks and problems view. Please open the problem view to see evaluation problems.

Viewing and Exporting Configuration Results

Results of a configuration can be accessed in a number of ways. The Result view (Window->Show View->Other->Variant Management->Result) allows graphical review of the concrete component models that have been derived from the corresponding family models in the configuration space.

The context menu of the Variant Project view provides an Export operation. As shown in the figure below, configuration results (features and components) can be exported as HTML, XML, and CSV formats. The XML data format is the same as for importing models but contains only the configured elements. The export dialog asks the user for a path and name and the export data formats for the generated files, and the model types to export.

Figure 2.4. Variant model export wizard (HTML export of all models selected)



Transforming Configuration Results

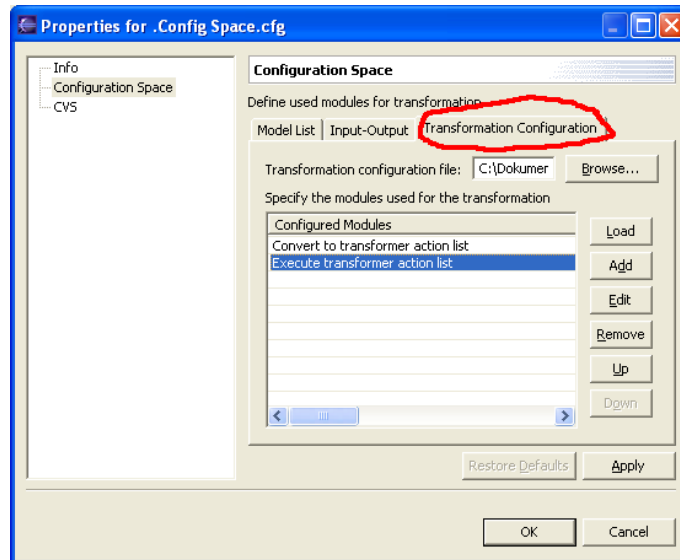
The last step in the automatic production of configured product variants is the transformation of the configuration results into the desired artefacts.

A modular, XML-based transformation engine is used to control this process (see the section called “ The XML Transformation Engine XMLTS ”). The transformation process has access to all models and additional parameters such as the input and output paths that have been specified in the configuration space properties dialog. The transformation file could be a single XSLT file, which is in turn executed with the configuration result as input, or a complete transformation module configuration.

The transformation configuration for a configuration space is specified in its properties dia-

log. The Transformation Configuration page (Figure 2.5, “Transformation configuration in configuration space properties”) of this dialog allows the creation and modification of transformation configurations. A default configuration for the standard transformation is created when the configuration space is created. See the section called “Configuration Space” for more information.

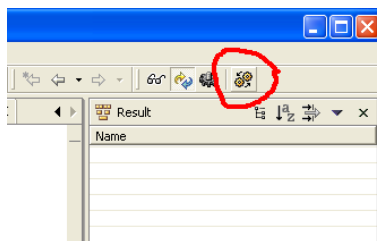
Figure 2.5. Transformation configuration in configuration space properties



The toolbar transformation button is used to initiate a transformation (see Figure 2.6, “Transformation button in Eclipse toolbar”). For more information on the XML transformation engine, see the document “XMLTS Transformation Engine”.

The distributed examples include some sample transformations.

Figure 2.6. Transformation button in Eclipse toolbar



Additional pure::variants plug-ins

The features offered by pure::variants may be further extended by the incorporation of additional software plug-ins. A plug-in may just contribute to the Graphical User Interface or it may extend or provide other functionality. For instance a plug-in could add a new editor tab for model editors or a new view. The online version of this user guide contains docu-

mentation for additional plug-ins. Printable documentation for the additional plug-in is distributed with the plug-ins and can be accessed from the online documentation via a hyperlink.

Currently available plugins provide TWiki [<http://www.twiki.org>] functionality for model elements, Bugzilla [<http://www.bugzilla.org>] integration, synchronization with Borland CaliberRM, access to version control systems such as CVS [<http://www.cvshome.org>] or Subversion [<http://www.cvshome.org>], and much more.

Exploring Documentation and Examples

Installing the "pure::variants User Documentation and Examples" feature gives access to online help and examples of pure::variants usage. Online documentation is accessed using "Help"->"Help Contents".

Examples can be installed as projects in the user's workspace by using "File"->"New"->"Example". The available example projects are listed in the dialog below the items "Variant Management" and "Variant Management SDK". Each example project typically comes with a Readme.txt file that explains the concept and use of the example.

Chapter 3. Concepts

Introduction

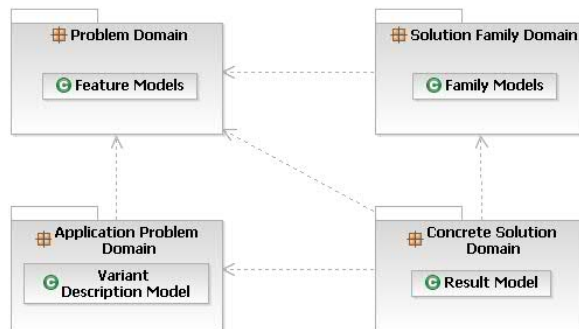
pure::variants provides a set of integrated tools to support each phase of the software product-line development process. pure::variants has also been designed as an open framework that integrates with other tools and types of data such as requirements management systems, object-oriented modeling tools, configuration management systems, bug tracking systems, code generators, compilers, UML or SDL descriptions, documentation, source code, etc.

Figure 3.1, “ Overview of family-based software development with pure::variants ” shows the four cornerstone activities of family-based software development and the models used in pure::variants as the basis for these activities.

When building the infrastructure for your Product Line, the problem domain is represented using hierarchical *Feature Models*. The solution domain, i.e. the concrete design and implementation of the software family, are implemented as *Component Family Models*.

The two models used for Application Engineering, i.e. the creation of product variants, are complementary to the models described above. The *Variant Description Model*, containing the selected feature set and associated values, represents a single problem from the problem domain. The *Concrete Component Model* describes a single concrete solution drawn from the solution family.

Figure 3.1. Overview of family-based software development with pure::variants



pure::variants manages the knowledge captured in these models and provides tool support for co-operation between the different roles within a family-based software development process:

- The *domain analyst* uses a feature model editor to build and maintain the problem domain model containing the commonalities and variabilities in the given domain.
- The *domain designer* uses a component family model editor to describe the variable family architecture and to connect it via appropriate rules to the feature models.
- The *application analyst* uses a variant description model to explore the problem do-

main and to express the problems to be solved in terms of selected features and additional configuration information. This information is used to derive a concrete component model from the family model(s).

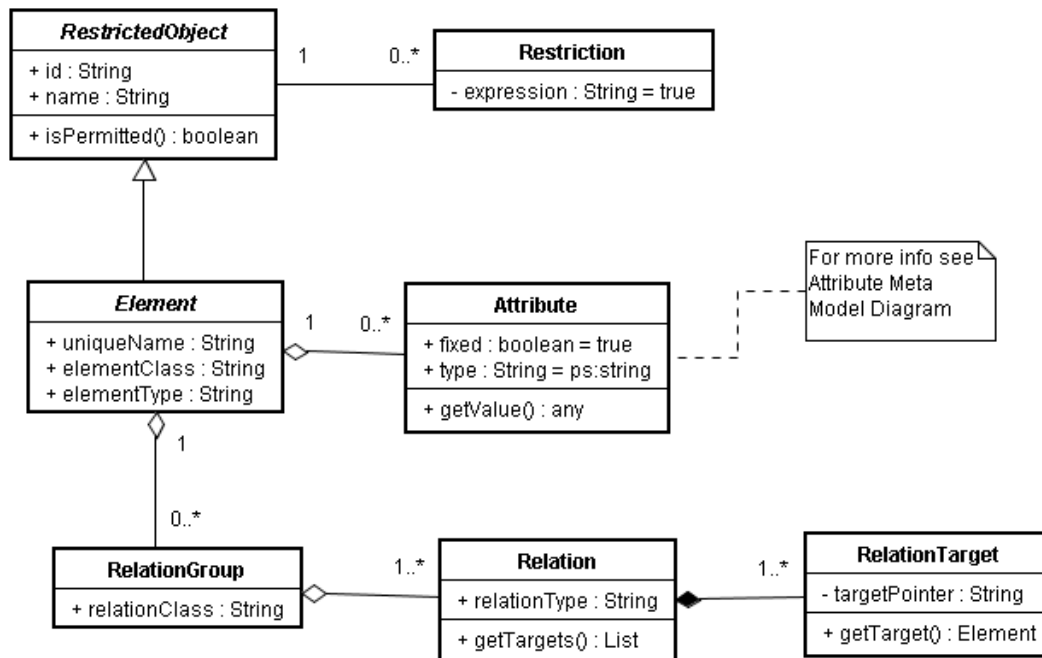
- The *application developer* generates a member of the solution family from the concrete component model by using the transformation engine.

Common Concepts in pure::variants models

This section described the common, generic structure on which both feature and family models are based.

Both models store elements (features in feature models, components, parts and source elements in family models) in a hierarchical tree structure. Elements (figure Figure 3.2, “(simplified) element meta model”) have an associated type and may have any number of associated attributes. An element may also have any number of associated relations. Additionally each element may be guarded by a number of restrictions. During model evaluation, an element cannot be part of a resulting configuration unless one of these restrictions evaluates to true. Detailed information about using restrictions is given in the section the section called “Model Evaluation”.

Figure 3.2. (simplified) element meta model



Element Relations

pure::variants allows arbitrary 1:n relations between model elements (feature/family model elements) to be expressed. The graphical user interface provides access to the most com-

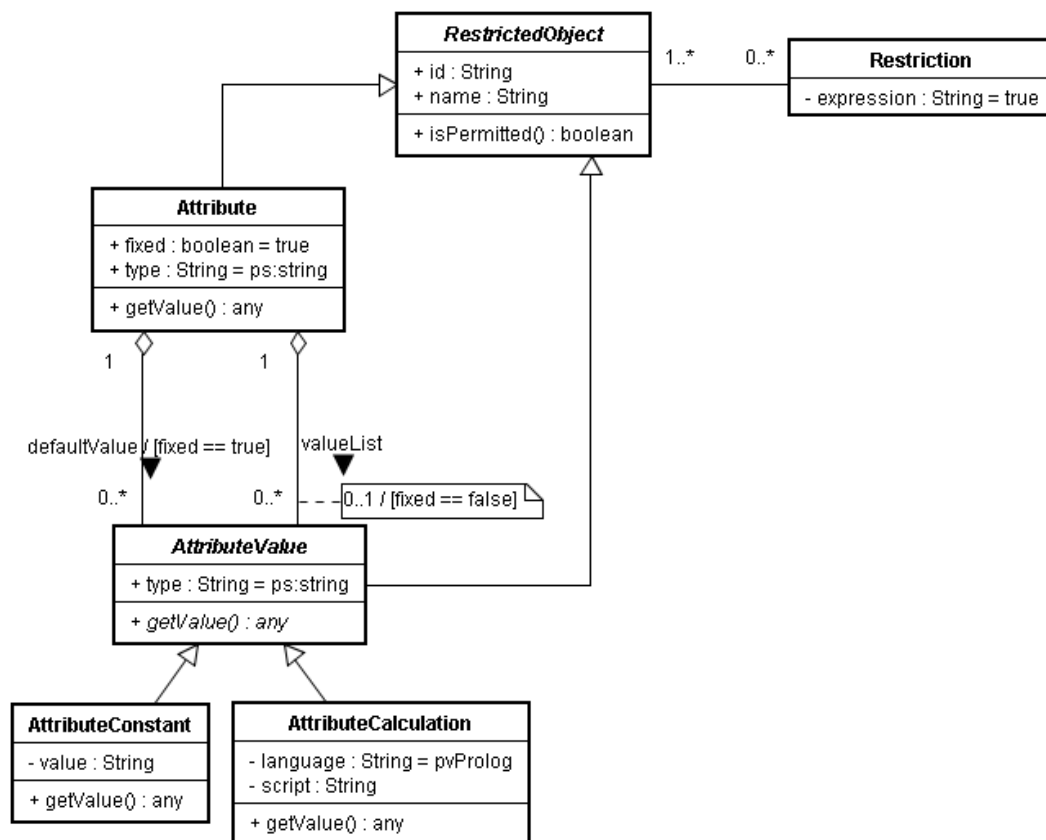
monly used relations. The extension interface allows additional relations to be accessed.


Examples of the currently supported relations are *requires*, *required_for*, *conflicts*, *recommends*, *discourages*, *cond_requires*, and *influences*. Use the relations page in the property dialog of a feature to specify feature relations. Table 6.1, “Supported Relations between Features/Element” documents the supported relations and their meanings..

Element Attributes

pure::variants uses attributes to specify additional information associated with an element. An attribute is a typed and named model element that can represent any kind of information (according to the values allowed by the type). An element may have any number of associated attributes. The Attributes of a selected model element are evaluated and their values calculated during the model evaluation process. A simplified version of the element attribute meta-model is shown below.

Figure 3.3. (Simplified) element attribute meta-model



Element attributes may be *fixed* (indicated with the checked  column in the UI) or *non-fixed*. The difference between a fixed and a non-fixed attribute is the location of the attribute value. The values of fixed attributes are stored together with the model element and are considered to be part of the model. A non-fixed element attribute value is stored in a variant description model, so the value may be different in other variant description models.

A non-fixed attribute may have a list of values that are used by default when the element is

selected and no valid value has been specified in the variant description model. Default values are stored in the model.

Guarding restrictions control the availability of attributes to the model evaluation process. If the restrictions associated with an attribute evaluate to *false*, the attribute is considered to be unavailable and may not be accessed during model evaluation.

A fixed attribute may have multiple value definitions assigned to it. A value definition may also have a restriction. In the evaluation process the value of the attribute is that of the first value definition that has a valid restriction (or no restriction) and successfully evaluates to *true*.

Attribute Value Types

The list of value types supported in `pure::variants` is defined in the `pure::variants` meta-model. Currently all types except `ps:integer` and `ps:float` are treated as string types internally. However, the transformation phase and some plug-ins may use the type information for an attribute value to provide special formatting etc..

The list of types provided by `pure::variants` is given in the reference section in table ???. Users may define their own types by entering the desired type name instead of choosing one of the predefined types.

Attribute Values

Attribute values may be represented using either constant values or calculations. Attribute values that are constant always have the same value. However, an attribute value can be calculated using a calculation expression to support complex usage scenarios. The syntax of the calculation expression depends on the expression language. `pure::variants` has an in-built expression language called `pvProlog` (see ???).

Feature Models

Feature models are used to express commonalities and variabilities efficiently. A feature model captures *features* and their *relations*. A *feature* is a property of the problem domain that is *relevant* with respect to commonalities of, and variation between, problems from this domain. The term *relevant* indicates that there is a stakeholder who is interested in an explicit representation of the given feature (property). What is relevant thus depends on the stakeholders. Different stakeholders may describe the same problem domain using different features.

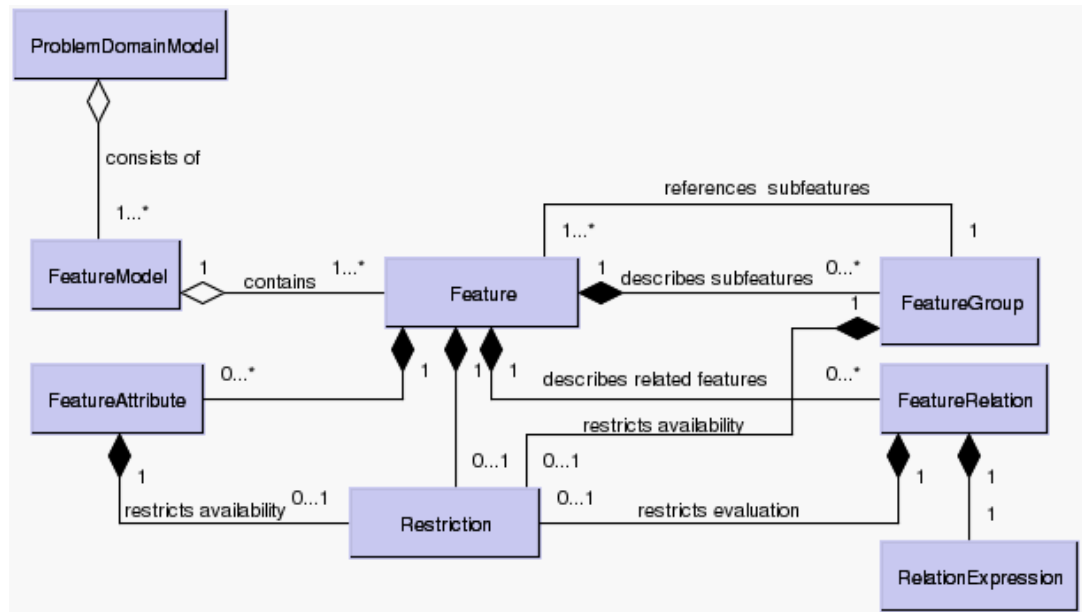
Feature relations can be used to define valid selections of combinations of features for a domain. The main representation of these relations is a *feature tree*. In this tree the nodes are features and the connections between features indicate whether they are *optional*, *alternative* or *mandatory*. Table Table 6.7, “Feature tree relation types and icons” gives an explanation on these terms and shows how they are represented in feature diagrams.

Additional constraints can be expressed as restrictions and/or element relations. Possible constraints could be allowing the inclusion of a feature only if two of three other features are selected as well, or disallowing the inclusion of a feature if one of a specific set of fea-

tures is selected.

Figure 3.4, “ Basic structure of feature models ” shows the principle structure of a pure::variants feature model as UML class diagram. A problem domain (ProblemDomainModel) consists of any number of feature models (FeatureModel). A feature model has at least one feature.

Figure 3.4. Basic structure of feature models



Feature Attributes

Some features of a domain cannot be easily or efficiently expressed by requiring a fixed description of the feature and allowing only inclusion or exclusion of the feature although for many features this is perfectly suitable. Feature attributes (i.e. element attributes in feature models) provide a way of associating arbitrary information with a feature. This significantly increases the expressive power of feature models.

However, it should be noted that this expressive power could come at a price in some cases. The main drawback is that for checking feature attribute values, the simple *requires*, *conflicts*, *recommends* and *discouraged* statements are insufficient. If value checks are necessary, for example to determine whether a value within a given range conflicts with another feature, pvOCL/pvProlog level restrictions will be required.

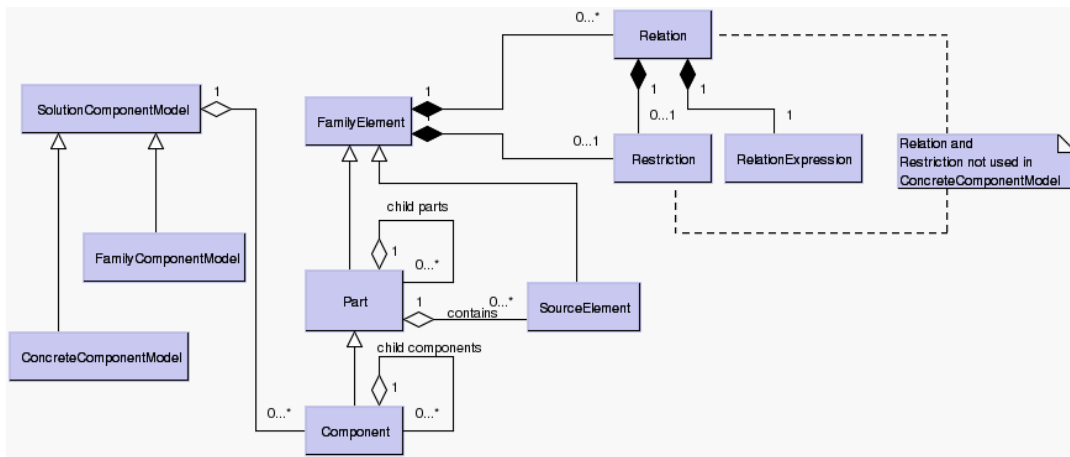
Feature Restrictions

Restrictions may be placed on features. These are checked during model evaluation. If no restriction on a feature evaluates to true, then the evaluation fails and a problem is reported to the user.

Family Models

The component family model (or family model) describes the solution family in terms of software architectural elements. Figure 3.5, “Basic structure of component family and component models” shows the basic structure of family models and the component models as a UML class diagram. Both models are derived from the `SolutionComponentModel` class. The main difference between the two models is that family models contain variable elements guarded by restriction expressions. Since component models are derived from family models and represent configured variants with resolved variabilities there are no restrictions used in component models.

Figure 3.5. Basic structure of component family and component models



Structure of the family model

The components of a family are organized into a hierarchy that can be of any depth. A component (with its parts and source elements) is only included in a result configuration when its parent is included and any restrictions associated with it are fulfilled. For top-level components only their restrictions are relevant.

Components:

A component is a named entity. Each component is hierarchically decomposed into further *components* or into *part elements* that in turn are built from *source elements*.

Parts:

Parts are named and typed entities. Each part belongs to exactly one component and consists of any number of *source elements*.

A part can be an element of a programming language, such as a class or an object, but it can also be any other key element of the internal or external structure of a component, for example an interface description. `pure::variants` provides a number of predefined part types, such as `ps:class`, `ps:object`, `ps:flag`, `ps:classalias`, and `ps:variable`. The family model is open for extension, and so new part types may be introduced, depending on the needs of the users.

Source elements:

Since parts are logical elements, they need a corresponding physical representation or representations. *Source elements* realise this physical representation. A source element is an unnamed but typed element. The type of a source element is used to determine how the source code for the specified element is generated. Different types of source elements are supported, such as `ps:file` that simply copies a file from one place to a specified destination. Some source elements are more sophisticated, for example, `ps:classaliasfile`, which allows different classes with different (aliases) to be used at the same place in the class hierarchy.

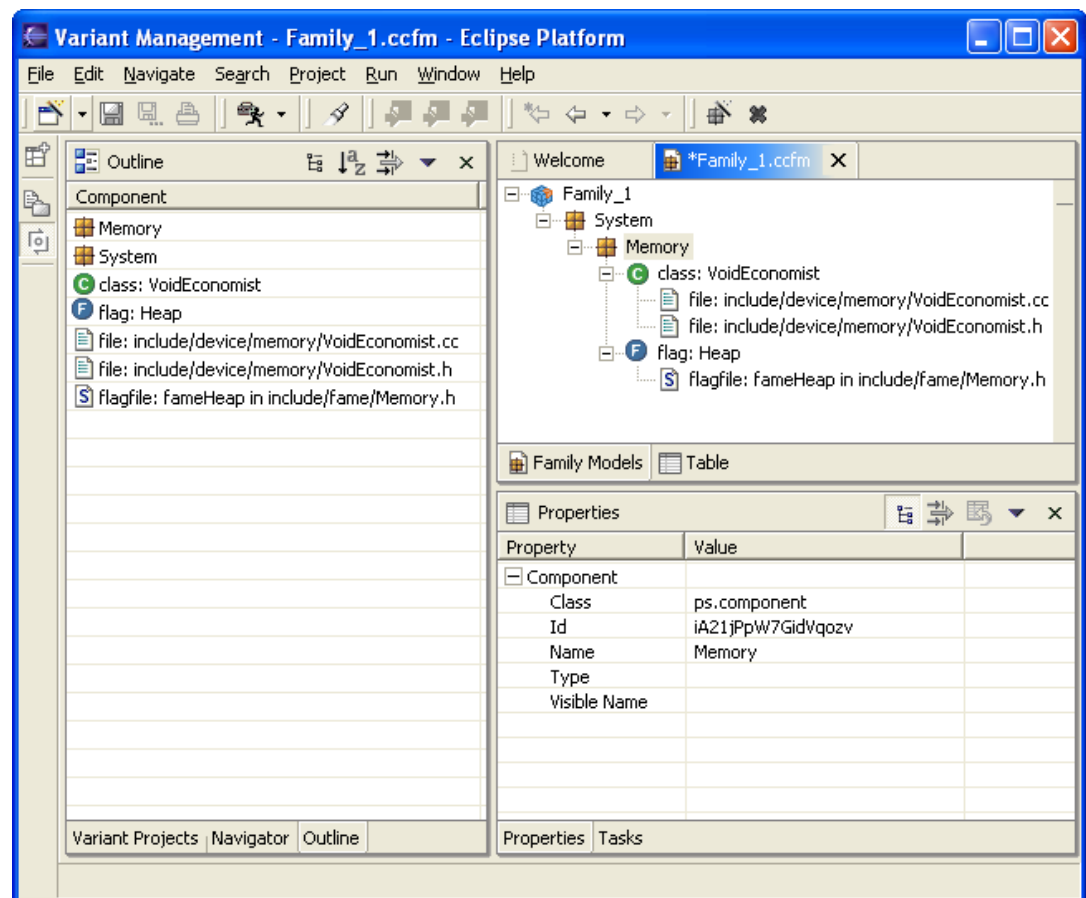
The actual interpretation of source elements is the responsibility of the `pure::variants` transformation engine. To allow the introduction of custom source elements and generator rules, `pure::variants` is able to host plug-ins for different transformation modules that interpret the generated concrete component model and produce a physical system representation from it.

The semantics of source element definitions are project, programming language, and/or transformation-specific.

Sample family model

An example family model is shown below:

Figure 3.6. Sample component family model



This model exhibits a hierarchical component structure. “System” is the top-level component, “Memory” its only sub component. Inside this component are two parts, a class, and a

flag. The class is realized by two source elements. Selecting an element of the family model will show its properties in the Properties view.

Using restrictions in Family Models:

A key capability that makes the family modelling language more powerful than other component description languages is its support for flexible rules for the inclusion of components, parts, and source elements. This is achieved by placing *restrictions* on each of these elements.

Each element may have any number of restrictions. An element is included if its parent is included and either there are no restrictions on it or at least one of its restrictions evaluates to *true*.

For example, assigning the restriction `not(hasFeature('Heap'))` to the class `VoidEconomist` in Figure 3.6, “Sample component family model” will cause the class and its child elements to be included when the feature `Heap` is not in the feature set of the variant. See the section called “Restrictions in Component Family Models” for more information.

Restrictions in Component Family Models

By default every element (component, part or source element) is included in a variant if its parent element is included, or if it has no parent element. specify conditions under which a configuration element may be excluded from a configuration.

It is possible to put restrictions on any element, and on element properties and relations. An arbitrary number of restrictions are allowed. Restrictions are evaluated in the order in which they are listed. If a restriction rule evaluates to *true*, the restricted element will be included.

A restriction rule may contain arbitrary (Prolog) statements. The most useful rule is `has_feature(<feature name/id>)` which evaluates to *true* if the feature selection contains the named feature.

Examples of Restriction Rules

Including an element only if a specific feature is present

```
hasFeature('Bar')
```

The element/attribute may be included only if the current feature selection contains the feature with identifier `Bar`.

Or-ing two restriction rules

Rule 1

```
not(hasFeature('BarFoos'))
```

Rule2

```
hasFeature('FoosBar')
```

This is a logical or of two statements. The element will be included if either feature Bar-Foos is not in the feature selection or FoosBar is in it.

It is also possible to merge both rules into one by using the or keyword.

Rule 1 or Rule 2

```
not(hasFeature('BarFoos')) or hasFeature('FoosBar')
```

Family Model Element Relations

As for features, each element (component, part, and source element) may have relations to other elements. The supported relations are shown in Table 6.9, “Supported Element Relations only supported in CCFM”.

When a configuration is checked, the configuration may be regarded as invalid if any relations are not satisfied.

Example using ps:exclusiveProvider/ps:requestsProvider relations

In the example below, the “Cosine” class element is given an additional *requestsProvider* relation to require that a cosine implementation must be present for a configuration to be valid. Relation statements are used in two different cosine implementations either of which could be used in some feature configurations (feature *FixedTime* and feature *Equidistant*), but which cannot both be (*ps:ExclusiveProvider*) in the resulting system.

```
ps:class("Cosine")
  Restriction: hasFeature('Cosine')
  Relation:    ps:requestsProvider = 'Cosine'

ps:file(dir = src, file = cosine_1.cc, type = impl):
  Restriction: hasFeature('FixedTime')
  Relation:    ps:exclusiveProvider = 'Cosine'

ps:file(dir = src, file = cosine_2.cc, type = impl):
  Restriction: hasFeature('FixedTime')
                and hasFeature('Equidistant')
  Relation:    ps:exclusiveProvider = 'Cosine'
```

Example for `ps:defaultProvider/ps:expansionProvider` relation

In the example given above an error message would be generated if the restrictions for both elements were valid, as it would not be known which element to include. Below, this example is extended by using the `ps:defaultProvider/ps:expansionProvider` relations to define a priority for deciding which of the two conflicting elements should be included. These additional relation statements are used to mark the two cosine implementations as an expansion point. The source element entry for `cosine_1.cc` specifies that this element should only be included if no more-specific element can be included (`ps:defaultProvider`). In this example, `cosine_2.cc` will be included when feature *FixedTime* and feature *Equidistant* are both selected, otherwise the default implementation, `cosine_1.cc` is included. If the autoresolver is activated then the appropriate implementation will be included automatically, otherwise an error message will highlight the problem.

```
ps:class("Cosine")
  Restriction: hasFeature('Cosine')
  Relation:    ps:requestsProvider = 'Cosine'

ps:file(dir = src, file = cosine_1.cc, type = impl):
  Restriction: hasFeature('FixedTime')
  Relation:    ps:exclusiveProvider = 'Cosine'
  Relation:    ps:defaultProvider = 'Cosine'
  Relation:    ps:expansionProvider = 'Cosine'

ps:file(dir = src, file = cosine_2.cc, type = impl):
  Restriction: hasFeature('FixedTime')
                  and hasFeature('Equidistant')
  Relation:    ps:exclusiveProvider = 'Cosine'
  Relation:    ps:expansionProvider = 'Cosine'
```

Model Evaluation

In the context of pure::variants, “model evaluation” is the activity of verifying that a variant description model (VDM) is complies with the feature and family models it is related to. Understanding this evaluation process is the key to successful use of restrictions and relations.

An outline of the evaluation algorithm is given in pseudo code below Figure 3.7, “Model Evaluation Algorithm (Pseudo Code)”.

Figure 3.7. Model Evaluation Algorithm (Pseudo Code)

```
modelEvaluation()
{
  foreach(current in modelRanks())
```



```
{
  checkAndStoreFeatSelection(getFeatMdlsByRank(current));
  selectAndStoreFromFamModels(getFamMdlsByRank(current),
                              class('ps:component'));
  selectAndStoreFromFamilyModels(getFamMdlsByRank(current),
                                 class('ps:part'));
  selectAndStoreFromFamilyModels(getFamMdlsByRank(current),
                                 class('ps:source'));
}
calculateAttributeValuesForResult();
checkFeatureRestrictions(getSelectedFeatures());
checkRelations();
}
```

The algorithm has certain implications on the availability of data to be used in restriction evaluation and attribute value calculations. But for simplicity we will consider for now that all feature and family models have the same model rank.

In the first evaluation step all feature selections made in the VDM are matched to the structure of their feature models. Firstly, all implicit features are determined and these are merged with the features selected by the user. Then, for this set, the structural rules for sub-feature selections are checked. This means, for example, that it is verified that one alternative is selected from an alternative feature group. Feature restrictions are not checked at this stage. Finally, the resulting set of selected features is stored so that it can be accessed later using, for example, *hasElement*.

The next step in the process involves accumulating elements from family models. This is achieved in three passes through the model. In the first pass, a breadth-first traversal of the components in the family model element hierarchy is performed. During this traversal, the restrictions of each component are evaluated. If a restriction evaluates to true, the respective component is added to the set of family model components accumulated so far. The accumulated set of components is now stored for later access. In the next pass all child part elements of the accumulated components have their restrictions evaluated in the same way as for components, and parts whose restrictions evaluate to true are added to the accumulated set of elements. The last pass evaluates the restrictions of all child source elements of the accumulated part elements and adds them to the accumulated set of elements if their restrictions evaluate to true. This evaluation order permits part element restrictions to safely access the component configuration, and source element restrictions to access the part configuration, since these configurations will not change once calculated.

Warning

This approach has a drawback in that calling "hasElement" on an element of the same class (e.g. 'ps:component'), and the same model rank, in the attribute value calculations or restrictions for an element will always yield 'false' as result. You must make sure that your family model element restrictions are "safe" by ensuring that they do not contain direct or indirect references to elements for which the selection has not yet been calculated (e.g. in attribute calculations or restrictions).

The algorithm has certain implications on the availability of information in restrictions and attribute value calculations. For simplicity we will consider for now that all feature and family models have the same model rank.

In the first evaluation step all feature selections stored in the VDM are matched to the structure of their feature models. First all implicit features are calculated and merged with the feature selected by the user. For this set it is now checked that structural rules for sub feature selections are fulfilled. This means that it is checked that one alternative is selected from an alternative feature group etc. Feature restrictions are not checked. This set of selected features is now stored for later access with “hasElement”.

The next step is to select elements from the family models. This is done in three iterations through the model. In a first run all components are checked in a breadth-first-traversal through the family model element hierarchy. For each component the restriction is evaluated. If the restriction evaluates to true, the respective component is added to the set of selected family model elements. When all components are checked, all child components of the selected components are checked until no more child components are found. The set of selected components is now stored for later access with “hasElement”. In the next run all restrictions of child part elements of selected components are evaluated in the same way as for components. The last run does this for all child parts of selected source elements. This evaluation order permits part element restrictions to safely access the component configuration, since it will not change anymore. The drawback is that it is not safe to reason about the component configuration in restrictions for components (of the same or lower ranks).

Warning

In pure::variants calling "hasElement" for an element of the same class (e.g. 'ps:component') and the same model rank will always yield 'false' as result. Make sure that family model element restrictions are "safe". That is, they do not contain directly or indirectly references to elements for which the selection is not yet calculated (e.g. in attribute calculations or restrictions).

The above steps are repeated for all model ranks starting with the earliest model rank and increasing to the latest model rank. (Note: the lower the model rank of a model, the earlier it is evaluated in this process e.g. a model of rank 1 is considered before a model of rank 2).

The last three steps in the model evaluation process are performed only once. Firstly, the attribute values for all selected elements are calculated. Secondly, the restrictions of the selected features are checked. Finally, the relations of the selected elements are checked. At this point all information about selected features and family model elements is available.

The XML Transformation Engine XMLTS

pure::variants supports a user-specified generation of product variants using an XML-based transformation component (XMLTS). XMLTS can process any kind of XML document by binding processing modules onto the nodes of the XML document according to a user-specified module configuration. These processing modules encapsulate the actions to be performed on a matching node in the XML document. A set of generic modules is supplied with XMLTS, e.g. a module to execute XSLT scripts and a module for collecting and executing transformation actions. The user may create custom modules and integrate these using the XMLTS module API.

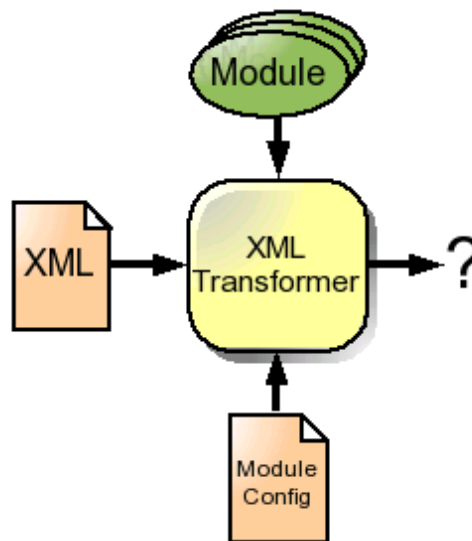
The Transformation Process

The XMLTS transformation process works by traversing the XML document tree. Each node visited during this traversal is checked to see whether any processing modules should be executed on it. If no module has to be executed, then the node is skipped. Otherwise the

actions of each module are performed on the node. Further modules executed on the node can process not only the node itself but also the results produced by previously invoked modules.

The processing modules to be executed are defined in a module configuration file. This file lists the applicable modules and includes configuration information for each module such as the types of nodes on which a module is to be invoked. The transformation engine evaluates this configuration information before the transformation process is started.

Figure 3.8. XML Transformer



The transformation engine initializes the available modules before any module is invoked on a node of the XML document tree. This could, for instance, give a database module the opportunity to connect to a database. The transformation engine also informs each module when traversal of the XML document tree is finished. The database module could now disconnect.

Before a module is invoked on a node it is queried as to whether it is ready to run on the node. The module must answer this query referring only on its own internal state.

A separately distributed XMLTS manual contains further information about the XML transformer. This manual shows how the built-in modules are used and how you can create and integrate your own modules.

pure::variants Transformation Input

In pure::variants, generation of a product variant is always based on the same XML language. An example of this XML is shown below:

```
<variant>
  <cfl>
    <element idref="element id"/>
    <novalue idref="property id"/>
    <value idref="property id" vid="property value id">
```

```
        eid="element id">
      ...
    </value>
    ...
  </cfl>
  <ccl>
    <element idref="element id"/>
    <novalue idref="property id"/>
    <value idref="property id" vid="property value id"
      eid="element id">
      ...
    </value>
    ...
  </ccl>
  <il>
    <inherited eid="element id" pid="property id"/>
    ...
  </il>
  <cm:consulmodels
    xmlns:cm="http://www.pure-systems.com/consul/model">
    <cm:consulmodel cm:type="ps:vdm" ...>
      ...
    </cm:consulmodel>
    <cm:consulmodel cm:type="ps:cfm" ...>
      ...
    </cm:consulmodel>
    <cm:consulmodel cm:type="ps:ccm" ...>
      ...
    </cm:consulmodel>
    ...
  </cm:consulmodels>
</variant>
```

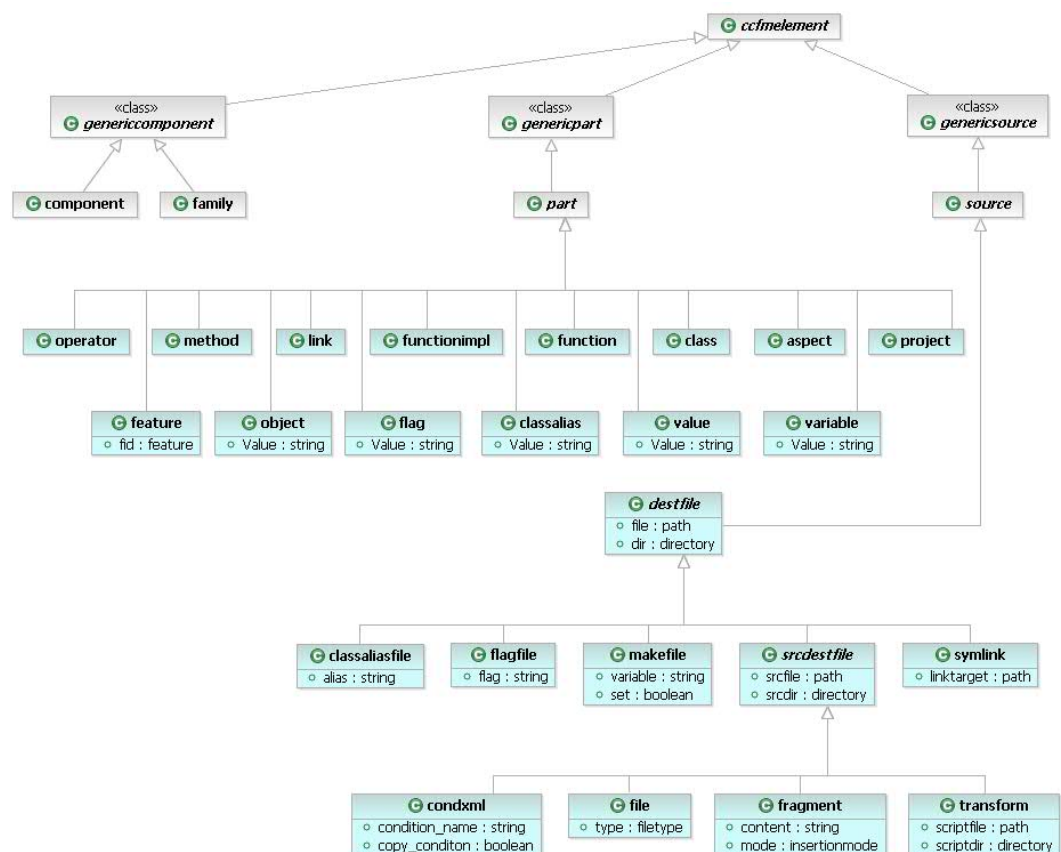
The <cfl>, <ccl>, and <il> subtrees list the concrete elements and property values of the product variant models. The models themselves are listed in the <cm:consulmodels> subtree. There can be several models of type “ps:cfm” (Concrete Feature Model) and type “ps:ccm” (Concrete Family Model) but always only one model of type “ps:vdm” corresponding to the original variant description model. pure::variants provides XSLT extension functions (see Table 6.11, “XSLT extension functions”) to simplify navigation and evaluation of this structure.

Chapter 4. The Standard Transformation

The standard transformation is suitable for many projects, such as those with mostly file-related actions for creating a product variant. This transformation also includes some special support for C/C++-related variability mechanisms like preprocessor directives and creation of other C/C++ language constructs (see the section called “Standard transformation for source elements”).

The standard transformation consists of a type model, describing the available element types for family models (see Figure 4.1, “The Standard Transformation Type Model”), and a corresponding transformation module, which converts a result model containing standard transformation elements into an action list for execution by the action list processor (also a transformation module).

Figure 4.1. The Standard Transformation Type Model



Using the standard transformation

The transformation configuration for the standard transformation is either created when a configuration space is created using the wizard, or can be (re-)created using the following instructions:

- Open the “Transformation Configuration” page in the configuration space properties
- Add the module “Standard transformation” using the “Add” button. Name it (as “Generate Standard Transformation Actionlist” for example)
- Add an Actionlist module. Leave the include pattern as “/variant” and all other parameters empty. Name it (as “Execute Actionlist” for example). In normal circumstances there should be only one Actionlist module for an include pattern, otherwise the action list gets executed twice (for each action list module matching the same node) on a tree node.

Standard Part Types

The provided part types are intended to capture the typical logical structure of procedural (ps:function, ps:functionimpl,) and object-oriented programs (ps:class, ps:object, ps:method, ps:operator, ps:classalias). Some general purpose types like ps:project, ps:link, ps:aspect, ps:flag, ps:variable, ps:value or ps:feature are also available.

Some of the part types have a mandatory attribute `Value`. The value of this attribute is used by child source elements of the part, for example to determine the value of the generated C preprocessor `#define` for a ps:flagfile source element. Unless noted otherwise any part element with a `Value` can be combined with any source element using an attribute `Value`. For example, it is possible to use a ps:value part with ps:flagfile and ps:makefile source elements to generate the same value into both a makefile and a preprocessor `#define` in some header file.

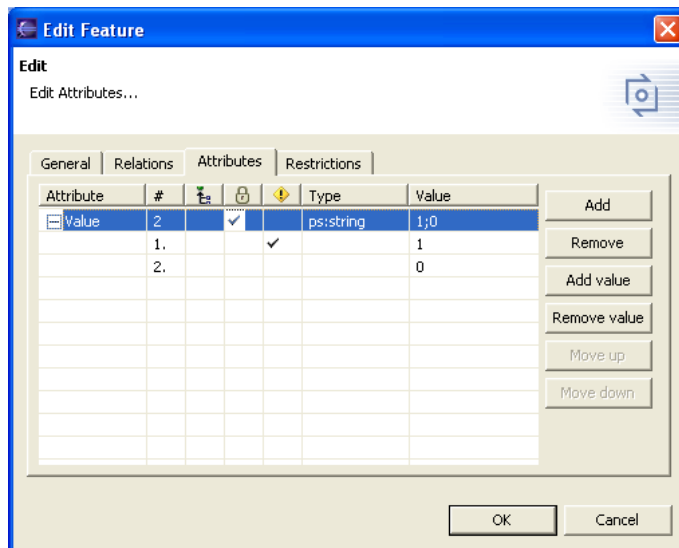
ps:feature is a special part type that can be used to define features which must be present if the part element is selected. If pure::variants detects a selected part of ps:feature with a valid `fid` attribute, the current feature selection must contain the feature with the calculated feature id. Otherwise the result is not considered to be valid. The autoresolver (if activated) tries to satisfy feature selections expected by ps:feature part elements. This functionality does not depend on the use of any specific transformation modules.

Assigning values to part elements

Calculation of a value for a ps:flag or ps:variable part is based on attribute values. At least one additional attribute `Value` definition for the part element is required. The attribute value may be a constant or a valid calculation. More than one attribute definition with the name `Value` can be used but such values should be guarded by restrictions. Definitions are evaluated in the order in which they are listed in the Attributes page in the element's Properties dialog. The first statement with a valid restriction and which evaluates to *true* is used.

Figure 4.2, “Multiple attribute definitions for Value calculation” shows typical `Value` attribute definitions. The value 1 is restricted and only set under certain conditions. Otherwise the unrestricted value 0 is used.

Figure 4.2. Multiple attribute definitions for Value calculation



Standard transformation for source elements

The standard transformation provides a rich set of source elements for file-oriented variant generation. Source elements can be combined with any part element (and also with part types which are not from the set of standard transformation part types) unless otherwise noted.

Source element definitions have the form:

aSourceElementType[attributeName1,attributeName2?]

The source element type `aSourceElementType` has one mandatory attribute named `attributeName1` and an optional attribute named `attributeName2`. The option is indicated by the trailing question mark.

This section lists the source element definitions supported by the standard transformation (see the section called “Using the standard transformation”):

All file-based standard transformations for elements derived from `ps:destfile` specify the location of the destination file using the two attributes `dir` and `file`. The resulting file will be placed in `<ConfigSpaceOutputDir>/<dir>/<file>`. Source elements derived from `ps:srcdestfile` are generated from some input file. The source file location is specified using the optional attributes `srcdir` and `srcfile`. If one or both of these attributes are not used, the values from `dir` and `file` are used. The source file location is relative to the `<ConfigSpaceInputDir>`. Both variables `ConfigSpaceInputDir` and `ConfigSpaceOutputDir` are stored as properties of the Configuration Space rather than in the model, since they may differ for different users of the same model.

**ps:file[ps:directory dir, ps:path file, ps:type type?,
ps:directory srcdir?, ps:path srcfile?]**

This definition is used for simple files that can be used without modification, but are located in a different place in the source hierarchy and/or have a different filename. The source file is copied from the source location to the destination file location.

The type should be `def` or `impl` when the file contains definitions (e.g. a C/C++ Header) or implementations. For most other files the type `misc` is appropriate.

type	description
<code>impl</code>	This type is used for files containing an implementation, e.g. <code>.cc</code> or <code>.cpp</code> files
<code>def</code>	This type is used for files containing declarations, e.g. C++ header files. In the context of <code>ps:classalias</code> calculations this information is used to determine the include files required for a given class.
<code>misc</code>	This type is used for any file that does not fit into the other categories.

`ps:fragment`[`ps:directory dir`, `ps:path file`, `ps:insertionmode mode`, `ps:string content?`, `ps:directory srcdir?`, `srcfile?`]

Appends content to a file. The content is taken either from a file if `srcdir` and `srcfile` are given, or from a string if `content` is given. The attribute `mode` is used to specify the point at which this content is appended to the file, i.e. “before” or “after” the child parts of the current node's parent part are visited. The default value is `before`.

`ps:transform`[`ps:directory dir`, `ps:path file`, `ps:directory, ps:directory scriptdir`, `ps:path scriptfile`, `ps:directory srcdir?`, `ps:path srcfile?`, `ps:string [xsltparameters]?`]

Transforms a document using an XSLT script and saves the transformation output to a file. The document to be transformed should be located in `<ConfigSpaceInputDir>/<srcdir>/<srcfile>`. The transformation output is written to `<ConfigSpaceOutputDir>/<dir>/<file>`. `<ConfigSpaceInputDir>/<scriptdir>/<scriptfile>` specifies the location of the XSLT script to be used. Any other attributes are interpreted as script parameters and so will be accessible as a global script parameter in the XSLT script containing the corresponding attribute value.

`ps:condxml`[`ps:directory dir`, `ps:path file`, `ps:directory srcdir?`, `ps:path srcfile?`, `ps:string condition_name?`, `ps:string copy_condition?`]

Copies an XML document and optionally saves the copy to a file. Special conditional XML attributes on XML nodes are dynamically evaluated to decide whether this node (and its subnodes) are to be copied. The name of the condition attribute to use is specified by the

ps:condxml attribute `condition_name` and defaults to “condition”. If the ps:condxml attribute `copy_condition` is not set to “false” then the condition attribute is copied into the target document as well. The condition itself must be a valid XPath expression and may use the pure::variants XSLT extension functions (see Table 6.11, “XSLT extension functions”).

Note

Before pure::variants release 1.2.4 the attribute names `pv.copy_condition` and `pv.condition_name` were used. These attributes are still supported in existing models but should not be used for new models. Support for these attribute names has been removed in pure::variants release 1.4.

In the example document given below the resulting XML document after processing with an ps:condxml transformation will only contain an introductory chapter if the corresponding feature `WithIntroduction` is selected.

Example 4.1. A sample conditional document for use with the ps:condxml transformation

```
<?xml version='1.0' ?>
<text>
<chapter condition="pv:hasFeature('WithIntroduction')">
  This is some introductory text.
</chapter>
<chapter>
  This text is always in the resulting xml output.
</chapter>
</text>
```

ps:flagfile[ps:directory dir, ps:path file, ps:string flag]

This definition is used to generate C/C++-Header files containing a `#define <flag> <flagValue>` statement. The `<flagValue>` is the value of the attribute `Value` of the parent part element. The name of the flag is specified by the attribute `flag`. See the section called “Assigning values to part elements” for more details. The same file location can be used in more than one ps:flagfile definition to include multiple `#define` statements in a single file.

Example 4.2. Generated code for a ps:flagfile for flag "DEFAULT" with value "1"

```
#ifndef __guard_DEBUG
#define __guard_DEBUG
#undef DEBUG
#define DEBUG 1
#endif
```

ps:makefile[ps:directory dir, ps:path file, ps:string variable]

This definition is used to generate makefile variables using a `<variable> += '<varValue>'` statement. `<varValue>` is the value of the attribute `Value` of the parent part element. The name of the variable is specified by the attribute `variable`. See the section called “Assigning values to part elements” for more details. The same file location can be used in more than one `ps:makefile` to include multiple makefile variables in a single file.

Example 4.3. Generated code for a ps:makefile for variable "CXX_OPTFLAGS" with value "-O6"

```
CXX_OPTFLAGS += "-O6"
```

ps:classaliasfile[ps:directory dir, ps:path file, ps:string alias]

Used to support different classes with different names that are concurrently used in the same place in the class hierarchy. This transformation is C/C++ specific and can be used as an efficient replacement for templates in some cases. This definition is only used in conjunction with the part type `ps:classalias`. The definition generates a `typedef aliasName aliasValue;` statement. `aliasValue` is the value of the attribute “Value” of the parent part element. Furthermore, during code generation the variant component model is searched for a class with name `aliasValue` and `#include` statements are generated for each of its `ps:file` source elements that have a “type” attribute equal to ‘def’.

Example 4.4. Generated code for a ps:classalias for alias "PCConn" with aliased class "NoConn"

```
#ifndef __PCConn_include__
#define __PCConn_include__
#include "C:\Weather Station Example\output\usr\wm-src\NoConn.h"
typedef NoConn PCConn;
#endif __PCConn_include__
```

ps:symlink[ps:directory dir, ps:path file, ps:string linktarget]

This definition creates a symbolic link to a file or directory named

`<linktarget>`.

Note

Symbolic links are not supported under Microsoft Windows operating systems. Instead files are copied.

Using XSLT to transform

A highly flexible way of generating product variants is to use XSLT in conjunction with the `pure::variants` XSLT extension functions. No special requirements are placed on the transformation you have to perform and using the extension functions is quite straightforward:

- Open the transformation configuration page in the configuration space properties.
- Add the “XSLT script execution” module using the “Add” button. Name it, (as “Execute XSLT script” for example).
- Change the module parameters page by pressing “Next” and enter the name of the XSLT script file you want to execute as value of the “in” parameter.
- An (optional) output file can be specified using the “out” parameter.
- Press Finish to close the transformation configuration page and start the transformation.

Example: Conditional Document Parts

To demonstrate how to use XSLT to generate a product variant, the following example will show the generation of a manual in HTML format with different content for different target groups (users, developers). This example uses the standard transformation and a user-provided XSLT script. The basic idea is to represent the manual in XML and then to use an XSLT script to generate the HTML representation. Attributes on the nodes of the XML document are used to discriminate between content for different target groups.

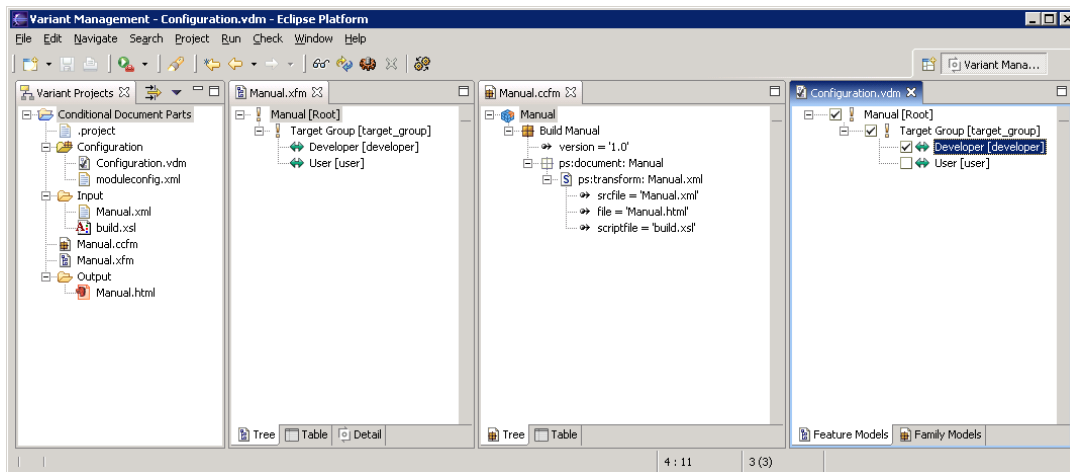
The example XML document looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<document>
  <title condition="pv:hasFeature('developer')">
    <par>Developer Manual</par>
  </title>
  <title condition="pv:hasFeature('user')">
    <par>User Manual</par>
  </title>
  <version>
    pv:getAttributeValue('build','ps:component','version')
  </version>
  <section name="Introduction">
    <par>Some text about the product...</par>
  </section>
  <section name="Installation">
    <subsection name="Runtime Environment">
      <par>Some text about installing
        the runtime environment...</par>
    </subsection>
    <subsection name="SDK"
      condition="not(pv:hasFeature('user'))">
      <par>Some text about installing
        the software development kit...</par>
    </subsection>
  </section>
```

```
<section name="Usage">
  <par>Some text about using the product...</par>
</section>
<section name="Extension API"
  condition="pv:hasFeature('developer')">
  <par>Some text about extending the product...</par>
</section>
</document>
```

The manual has a title, a version, sections, subsections, and paragraphs. The title and the presence of some sections and subsections are conditional on the target group. The attribute `condition` has been added to the dependent parts of the document to decide which part(s) of the document are to be included. These conditions test the presence of certain features in the product variant. Figure 4.3, “Variant project describing the manual” shows the corresponding feature and family models in a variant project using the standard transformation.

Figure 4.3. Variant project describing the manual



The feature model describes the different target groups that the manual's content depends on. The family model describes how to build the HTML document, i.e. which XSLT script is to be used to transform which XML document into HTML. For this purpose the standard transformation source element `ps:transform` has been used (see the section called “Standard transformation for source elements”). This source element refers to the XSLT script `build.xsl` shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:pv="http://www.pure-systems.com/purevariants"
  xmlns:cm="http://www.pure-systems.com/consul/model"
  xmlns:dyn="http://exslt.org/dynamic"
  extension-element-prefixes="pv dyn">

  <!-- generate text -->
  <xsl:output method="html"/>
```

```
<!-- match document root node -->
<xsl:template match="document">
  <html>
    <title></title>
    <body>
      <xsl:apply-templates mode="body"/>
    </body>
  </html>
</xsl:template>

<!-- match title -->
<xsl:template match="title" mode="body">
  <xsl:if test="not(@condition) or dyn:evaluate(@condition)">
    <h1><xsl:apply-templates mode="body"/></h1>
  </xsl:if>
</xsl:template>

<!-- match version -->
<xsl:template match="version" mode="body">
  <p><b><xsl:value-of
    select="concat('Version:',dyn:evaluate(.))"/></b>
  </p>
</xsl:template>

<!-- match section -->
<xsl:template match="section" mode="body">
  <xsl:if test="not(@condition) or dyn:evaluate(@condition)">
    <h2><xsl:value-of select="@name"/></h2>
    <xsl:apply-templates mode="body"/>
  </xsl:if>
</xsl:template>

<!-- match subsection -->
<xsl:template match="subsection" mode="body">
  <xsl:if test="not(@condition) or dyn:evaluate(@condition)">
    <h4><xsl:value-of select="@name"/></h4>
    <xsl:apply-templates mode="body"/>
  </xsl:if>
</xsl:template>

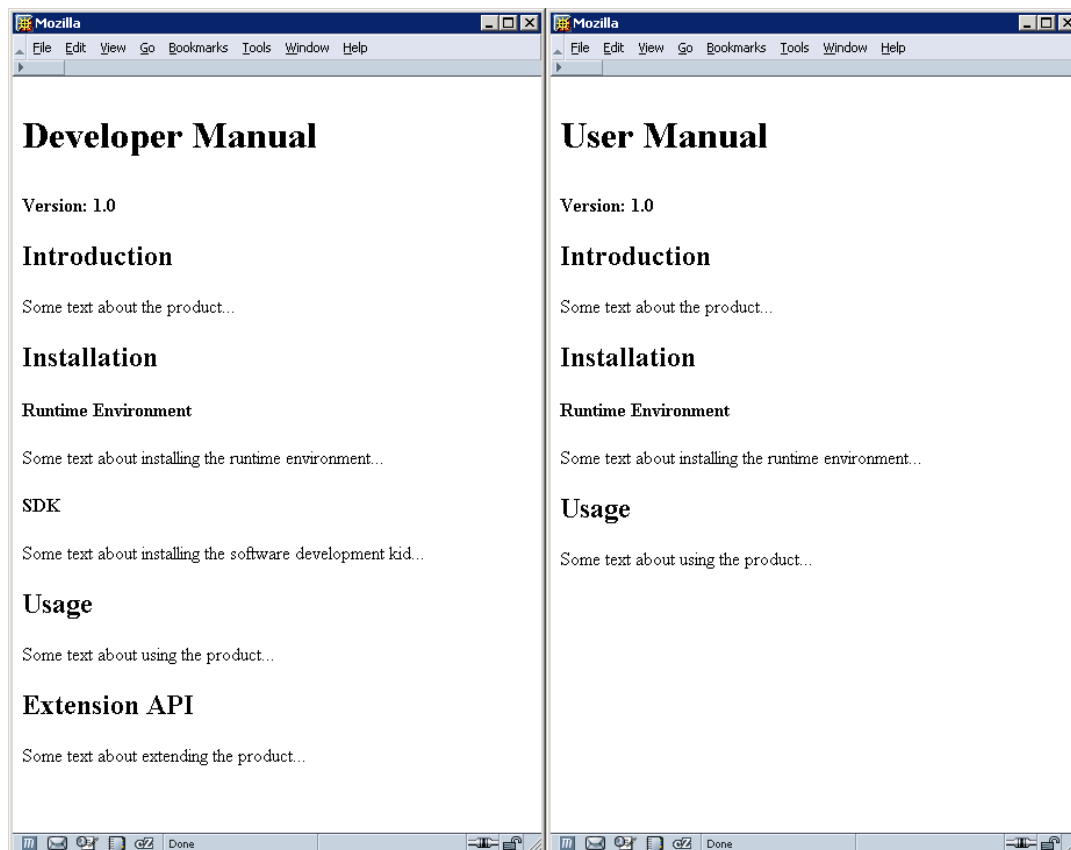
<!-- match paragraphs -->
<xsl:template match="par" mode="body">
  <p><xsl:value-of select="."/></p>
</xsl:template>
</xsl:stylesheet>
```

The script takes XML as input and produces HTML as output. It has several transformation parts, one for every manual element, where the condition attributes are dynamically evaluated. Note that these condition attributes are expected to be valid XPath expressions. In the XML description of the manual these expressions contain calls to the pure::variants XSLT extension function `hasFeature()`, this expects the unique name of a feature as an argument and returns *true* if this feature is in the product variant (see Table 6.11, “XSLT extension functions”). If a node of the XML document has such a condition and this condition fails, the node and all of its child nodes are ignored and are not transformed into HTML. For example, if a `<section>` node has the condition `hasFeature('user')` and the feature `user` is not selected in the product variant, then this section and all its subsections will be ignored.

In the XML description of the manual a second `pure::variants` XSLT extension function is called, `getAttributeValue()`. This function is used to get the manual version from the family model. It reads the value of the `version` attribute of the component build and returns it as string.

Figure 4.4, “The manual for users and developers” shows the two variants of the manual (HTML) generated selecting target group user and then developer.

Figure 4.4. The manual for users and developers



Chapter 5. Graphical User Interface Elements

The layout and usage of the pure::variants User Interface closely follows Eclipse guidelines. See the Workbench User Guide [<http://help/topic/org.eclipse.platform.doc.user/gettingStarted/qs-01.htm>] provided with Eclipse (“Help”->“Contents”) for more information on this.

Getting Started with Eclipse

This section gives a short introduction to the elements of the Eclipse UI before introducing the pure::variants UI. Readers with Eclipse experience may skip this section.

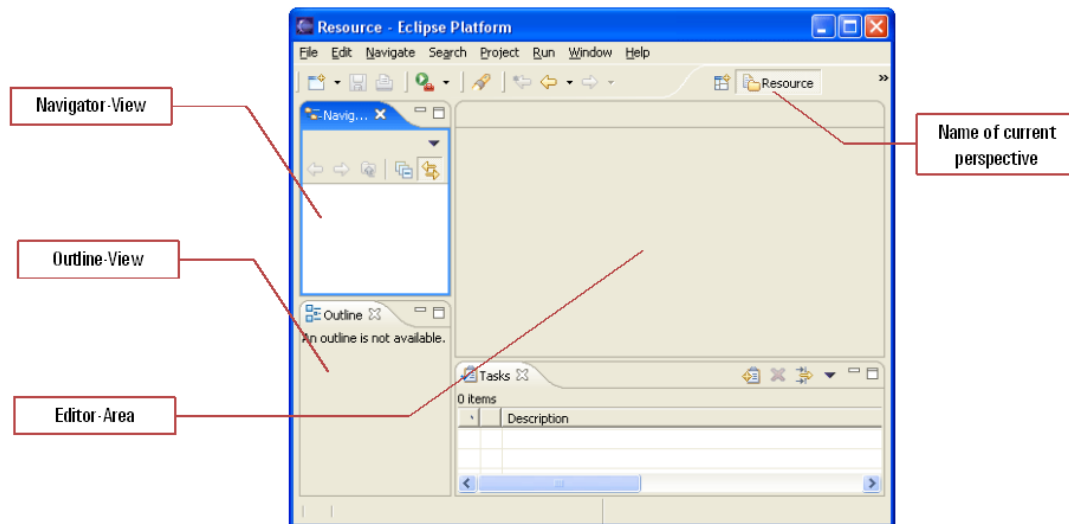
Eclipse is based around the concepts of *workspaces* and *projects*. Workspaces are used by Eclipse to refer to enclosed projects, preferences and other kinds of meta-data. A user may have any number of workspaces for different purposes. Outside of Eclipse, workspaces are represented as a directory in the file system with a subdirectory `.meta-data` where all workspace-related information is stored. A workspace may only be used by a single Eclipse instance at a time. Projects are structures for representing a related set of resources (e.g. the source code of a library or application). The contents and structure of a project depends on the nature of the project. A project may have more than one nature. For example, Java projects have a Java nature in addition to any project-specific natures they may have. Natures are used by Eclipse to determine the type of the project and to provide specialised behaviour. Project-specific meta information is stored in a `.project` file inside the project directory. This directory could be located anywhere in the file system, but projects are often placed inside a workspace directory. Projects may be used in more than one workspace by importing them using (“File”->“Import”->“Import Existing Project”).

Figure Figure 5.1, “Eclipse workbench elements” shows an Eclipse workbench window. A *perspective* determines the layout of this window. A perspective is a (preconfigured) collection of menu items, toolbar entries and sub-windows (*views* and *editors*). For instance this figure shows the standard layout of the Resource perspective. Perspectives are designed for performing a specific set of tasks (e.g. the Java perspective is used for developing Java programs). Users may change the layout of a perspective according to their needs by placing views or editors in different locations, by adding or closing views or editors, menu items and so on. These custom layouts may be saved as new perspectives and re-opened later. The standard layout of a perspective may be restored using “Window”->“Reset Perspective”.

Editors represent resources, such as files, that are in the process of being changed by the user. A single resource cannot be open in more than one editor at a time. A resource is normally opened by double-clicking on it in a *Navigator* view or by using a context menu. When there are several suitable editors for a given resource type the context menu allows the desired one to be chosen. The figure below shows some of the main User Interface elements:

Figure 5.1. Eclipse workbench elements

Variant Management Perspective

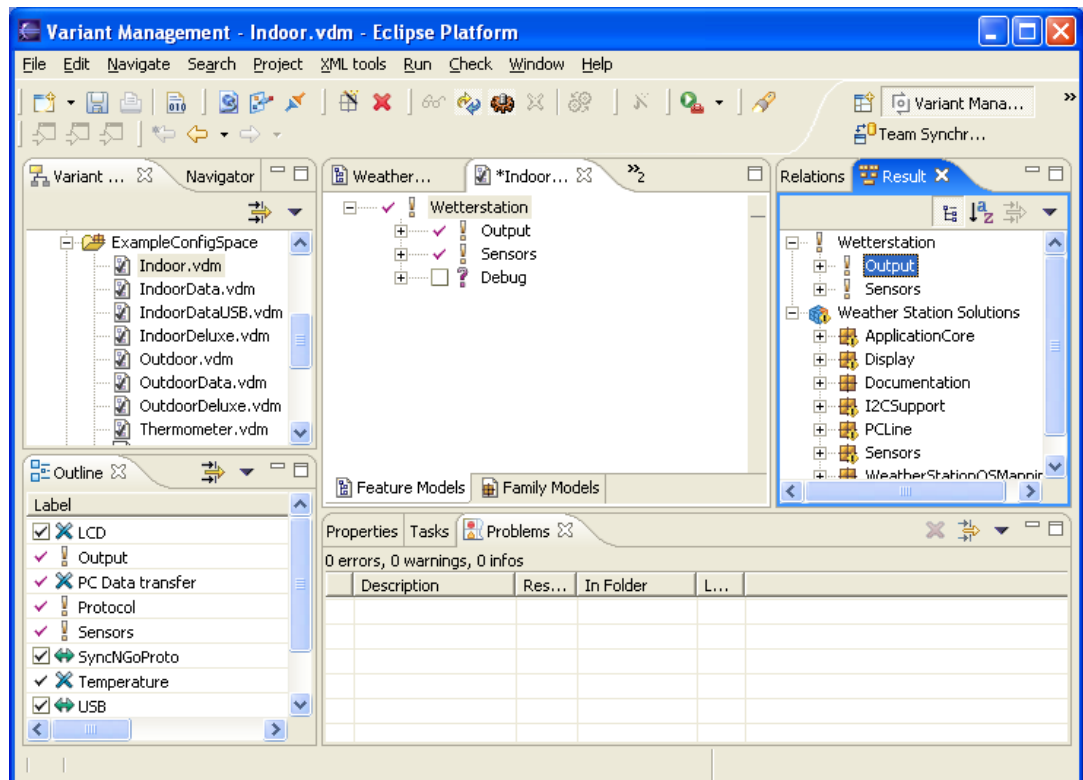


Eclipse uses *Views* to represent any kind of information. Despite their name, data in some types of view may be changed. Only one instance of a specific type of view, such as the Outline view, may be shown in the workbench at a time. All available views are accessible via Windows->Show View->Other.

Variant Management Perspective

pure::variants adds a Variant Management perspective to Eclipse to provide comprehensive support for variant management. This perspective is opened using “Window->Open Perspective->Other->Variant Management”. Figure 5.2, “ Variant management perspective standard layout ” shows this perspective with a sample project.

Figure 5.2. Variant management perspective standard layout



Editors

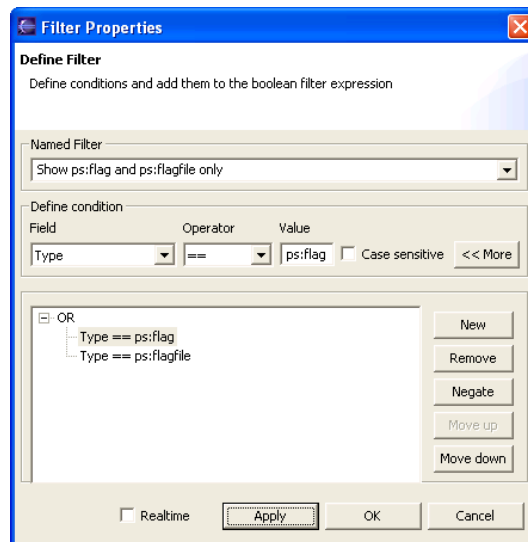
pure::variants provides specialized editors for each type of model. Each editor can have several pages representing different model visualizations (e.g. tree-based or table-based). Selecting the desired page tab within the editor window changes between these pages.

Common Editor Actions

Filtering

Most views and editors support filtering. Depending on the type of view, the filtered elements are either not shown (table like views) or shown in a different style (tree views). Filters can be defined, or cleared, from the context menu of the respective view/editor page. When the view/editor has several pages the filter is active for all pages.

Figure 5.3. Filter definition dialog



Arbitrarily complex filters based on comparison operations between feature/element properties (name, attribute values, etc.) and logical expressions (and/or/not) are supported. Comparison operations include conditions like equality and containment, regular expressions (matches) and checks for the existence of an attribute for a given element (empty/not empty).

Filters can be named for later reuse using the Named filter field. The drop-down box allows access to previously defined filters. Fast access to named filters is provided by the Filter view, which can be activated using the Windows->Views->Other->Variant Management->Filter item.

Common Editor Pages

Since most models are represented as hierarchical tree structures, different model editors share a common set of pages and dialogs.

Tree Editing Page

The tree-editing page shows the model in a tree-like fashion (like Windows Explorer). This page allows multiple-selection of elements and supports *drag and drop*. Tree nodes can also be cut, copied, and pasted using the global keyboard shortcuts or via a context menu.

Selection of a tree node causes other views to be updated, for instance the Properties view. Conversely, some views also propagate changes in selection back to the editor (e.g. the outline views).

A context menu enables the expansion or collapse of all children of a node.

Double-clicking on a node opens a property dialog for that node.

Table View/Editor Page

The table view is available in many views and editors. This view is a tabular representation of the tree nodes. The visible columns and also the position and width of the columns can be customized via a context menu (Table Layout->Change). Layout changes for each model are stored permanently in the Eclipse workspace. Clicking on a column header sorts that column. The sort direction may be reversed with a second click on the same column header.

Tip

Double clicking on a column header separator adjusts the column width to match the maximal width required to completely show all cells of that column.

Most cells in table views are directly editable. A single-click into a cell selects the row; a second click opens the cell editor for the selected cell. The context menu for a row permits addition of new elements or deletion of the row. A double-click on a row starts a property dialog for the element associated with the row.

Element Property Dialog

The property dialog for an element contains General, Relations, Attributes and Restrictions pages.

General Page

Unique Name	A unique name for the model element. The name must not begin with a numeric character and must not contain spaces. The uniqueness of the name is automatically checked against other model elements. The unique name can be used to identify elements instead of their unique identifier. Unique names are required for each feature, but not for other model elements; if another model element is given a unique name then it must be unique. The Unique name is displayed by default (in brackets if the visible name is also displayed).
Visible Name	An informal name for the model element. This name is displayed in views by default. This name can be composed of any characters and doesn't have to be unique.
Description	A description of the feature. The description may contain HTML or plain text.

Relations page

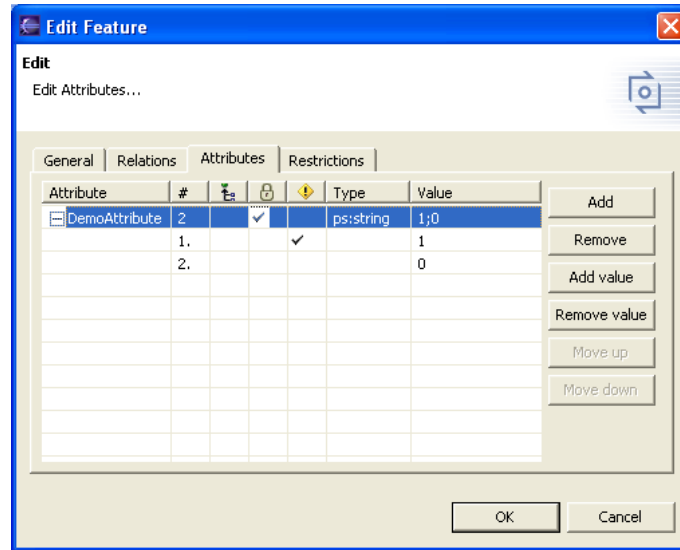
This page allows definition of additional relations between an element and other elements, such as features or components. Typical relations between features, such as requires or conflicts, can be expressed using a number of built-in relationship types. The user may also extend the available relationship types. More information on element relations can be found in the section called “Element Relations”.


Attributes page


Every element may have an unlimited number of associated attributes (name-value pairs).


The attributes page uses a table of trees to visualize the attribute declaration (root row) and (optional) attribute value definitions (child rows).

Each attribute has an associated Type and may have any number of Value definitions associated with it. The values must be of the specified Type. The number of attribute value definitions is shown in the “#” column. In the example the attribute “DemoAttribute” has two value definitions (“1” and “0”). In the example in Figure 5.4, “Sample attribute definitions for a feature”, the attribute DemoAttribute has two value definitions (1 and 0).

Figure 5.4. Sample attribute definitions for a feature

Attributes can be *inherited* from parent elements. Checking the inheritable cell (column icon ) in the parent elements Attribute page does this. An inherited attribute may be overridden in a child element by defining a new attribute with the same name as the inherited attribute. The new attribute may or may not be inheritable as required.

Attributes can be *fixed* by checking the cell in the  column. Fixed attributes are calculated from value definitions in the model in which they are declared, in contrast to non-fixed attributes for which the value is specified in a variant description model. Default values can be (optionally) defined here for non-fixed attributes. These are used if no value is specified in the variant description model.

An attribute may have a restricted availability. This is indicated by a check mark in the  column. Clicking on a cell in this column activates the Restrictions editor. To restrict the complete attribute definition use the restriction cell in the attribute declaration (root) row. To restrict an attribute value, expand the attribute tree and click into the restriction cell of the value. Restrictions can either be entered directly into a cell or by using the Restrictions editor. Clicking on the button marked ... which appears in the cell when it is being edited also opens this editor. See the section called “ Restrictions page and Restrictions Editor ” for detailed information.

During model evaluation, attribute values are calculated in the listed order. The Move Up and Move Down buttons on the right side of the page can be used to change this order. The first definition with a valid restriction (if any) and a constant, or a valid calculation result, defines the resulting attribute value.

Values can be entered directly into a cell or by using the Value editor. Clicking on the button marked ..., which appears in the cell when it is being edited, opens this editor. The editor also allows the value definition type to be switched between constant and calculation. The calculation type can use the pvProlog language to provide more complex value definitions. More information on calculating attribute values is given in ???.

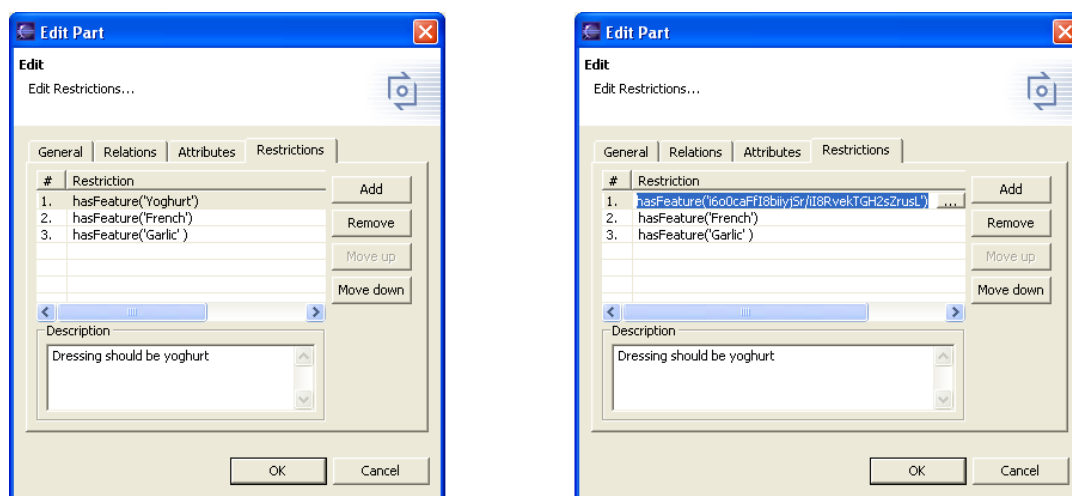
The use of attributes is covered further in the section called “ Element Relations ”.

Restrictions page and Restrictions Editor

The Restrictions page defines element restrictions. This page is identical to the restrictions editor used elsewhere in pure::variants, so this description applies to both.

Any element that can have restrictions can have any number of them. A new restriction can be created using the Add button; an unnecessary restriction can be removed using Remove. The order of restrictions may be changed using the Move Up and Move Down buttons on right side of the page.

Figure 5.5. Restrictions page shown in property editor. Right picture

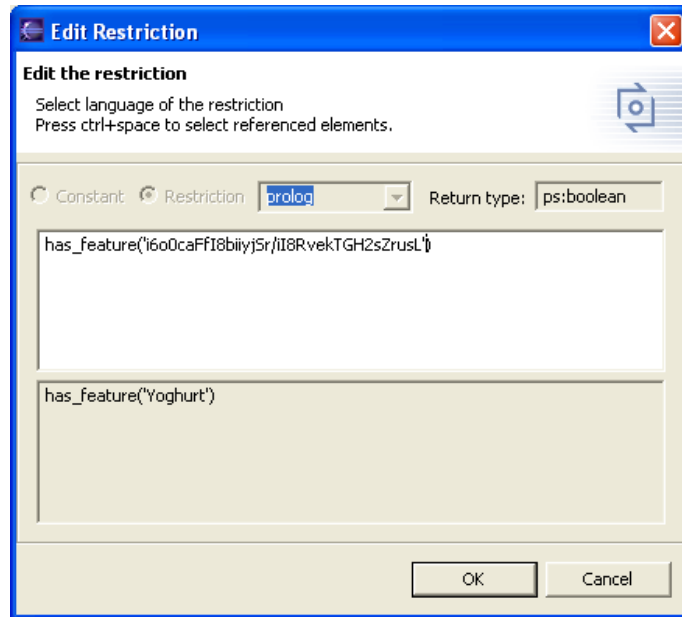


A restriction can be edited in place using the cell editor (shown in the right side of figure Figure 5.5, “ Restrictions page shown in property editor. Right picture ”). Note the difference in restriction #1 in the left and right sides of the figure. Unless they are being edited, the element identifiers in restrictions are shown as their respective unique names (e.g. 'Garlic') when available. When the editor is opened the actual restriction is shown (e.g. 'i6o.../...rusL'), and no element identifier substitution takes place. The ... button opens an editor dialog that is more suitable for complex restrictions. This editor (shown in figure Figure 5.6, “ Restriction editor ”) provides both views (without/with substituted element identifiers) in the upper and lower part of the window. An element selection dialog can be opened using **CTRL+SPACE** in order to help insert identifiers. All element identifiers selected from here are inserted into the restriction code as quoted strings.

Warning

The restriction syntax is not checked in this editor.

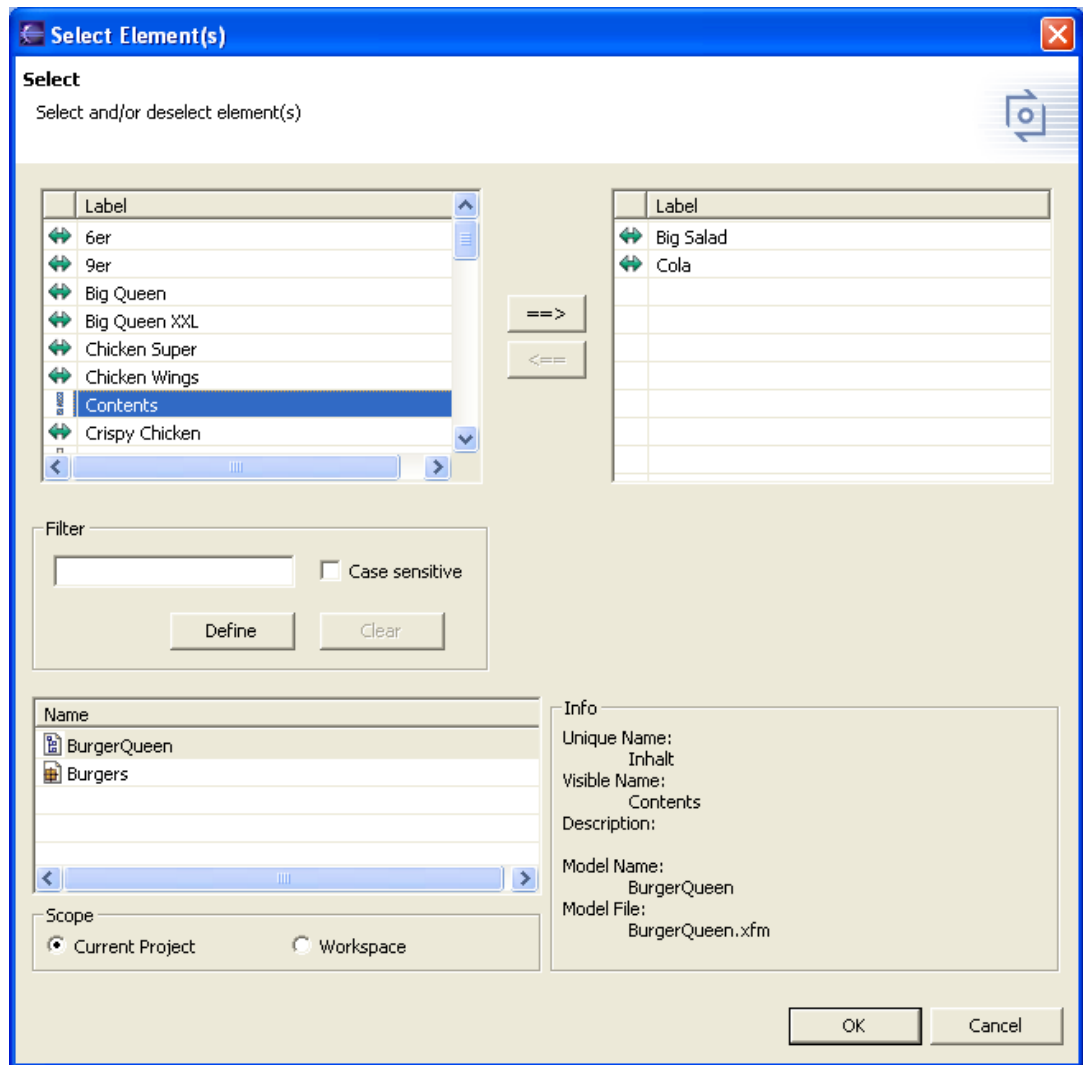
Figure 5.6. Restriction editor



Element Selection Dialog

The element selection dialog (figure Figure 5.7, “Element selection dialog”) is used in most cases when a single element or a set of elements has to be selected. The left pane lists the potentially available elements, the right pane lists the selected elements. To select additional elements, select them in the left pane and press the button “==>”. Multiple selection is also supported. To remove elements from the selection, select them in the right pane and use the button “<==”.

Figure 5.7. Element selection dialog



The model selection and filter fields in the lower part of the dialog control the elements that are shown in the left *Label* field. By default, all elements for all models within the current project are shown. If a filter is selected, then only those elements matching the filter are shown. If one or more models are selected, then only elements of the selected models are visible. If the scope is set to *Workspace* then all models from the current workspace are listed. The model selection is stored, so for subsequent element selections the previous configuration is used.

Tip

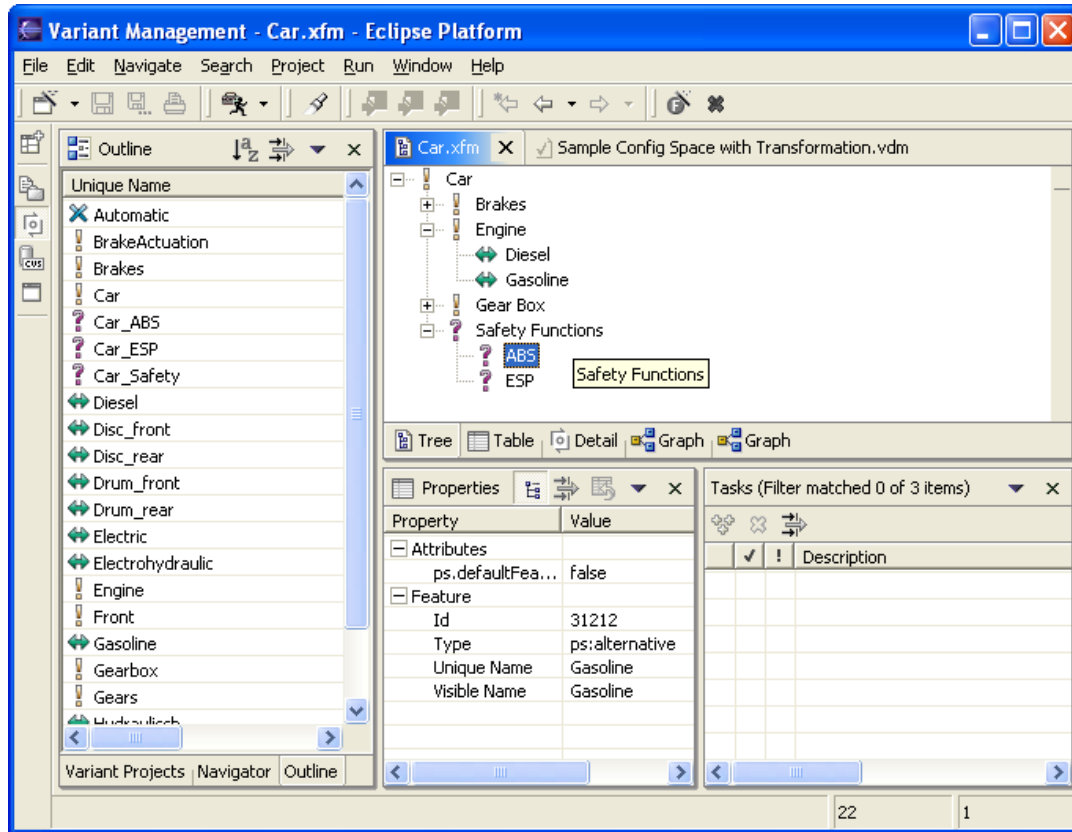
The element information shown in the left and right *Label* fields is configurable. Use “column properties” from the context menu to select and arrange the visible columns. See the section called “Table View/Editor Page” for additional information on table views.

Feature Model Editor

Every open Feature model is shown in a separate Feature model editor tab in Eclipse. This editor is used to add new features, to change features, or to remove features. Variant configuration is not possible using this editor. Instead, this is done in a variant description model editor (see the section called “Variant Description Model Editor” and the section called “Using Configuration Spaces” for more information).

The default page of a feature model editor is the tree-editing page. The root feature is shown as the root of the tree and child nodes in the tree denote sub-features. The icon associated with a feature shows the relation of that feature to its parent feature (see Table 6.7, “Feature tree relation types and icons”).

Figure 5.8. Open feature model editor with outline and property view



Some keyboard shortcuts are supported in addition to mouse gestures. These are documented in Table 5.1, “Keyboard short cuts in feature model tree editor”. Some shortcuts may not be supported on all operating systems platforms.

Table 5.1. Keyboard short cuts in feature model tree editor

Key	Action
ENTER	Show property dialog
ENTF	Delete feature
Up/Down cursor keys	Navigate tree
Left/Right cursor keys	Collapse or Expand tree nodes

CTRL+Z	Undo
CTRL+Y	Redo
CTRL+C	Copy into clipboard
CTRL+X	Cut into clipboard
CTRL+V	Paste from clipboard

Creating and Changing Features

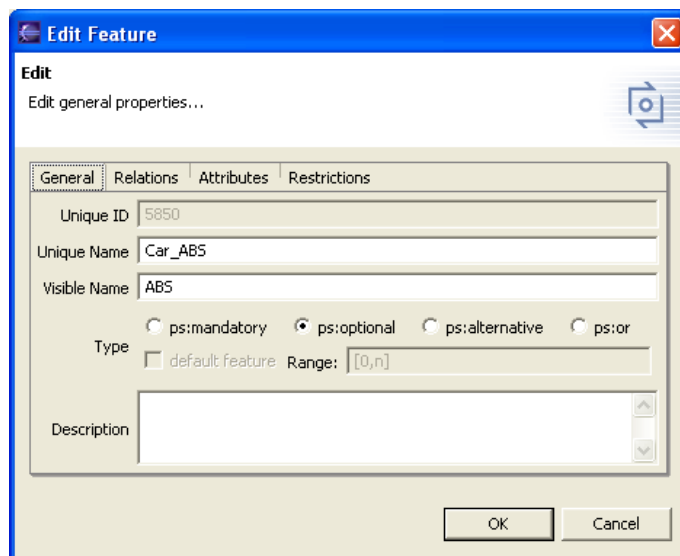
Whenever a new feature model is created, a root feature of the same name is automatically created and associated with the model.

Additional sub-features may be added to an existing feature using the New context menu item. This opens the new feature wizard (see Figure 5.9, “Feature property dialog”) where the user must enter a unique name for the feature and may enter other information such as a visible name or some feature relations. All feature properties can be changed later using the Property dialog (Context menu entry Properties). A detailed description of feature properties is given in the section called “Changing feature properties”.

A feature may be deleted from the model using the context menu entry Delete. This also deletes all of the features child features.

Cut, copy and paste commands are supported to manipulate sub-trees of the model. These commands are available on the “Edit menu”, the context menu of an element and as keyboard shortcuts.

Figure 5.9. Feature property dialog




Changing feature properties

Feature properties, other than a feature's Unique Identifier, may be changed using the Property dialog. This dialog is opened by double-clicking the feature or by using the context menu item "Properties".

The feature properties dialog extends the standard element properties dialog (see the section called "Element Property Dialog") with the addition of the following feature specific items on the "General" page

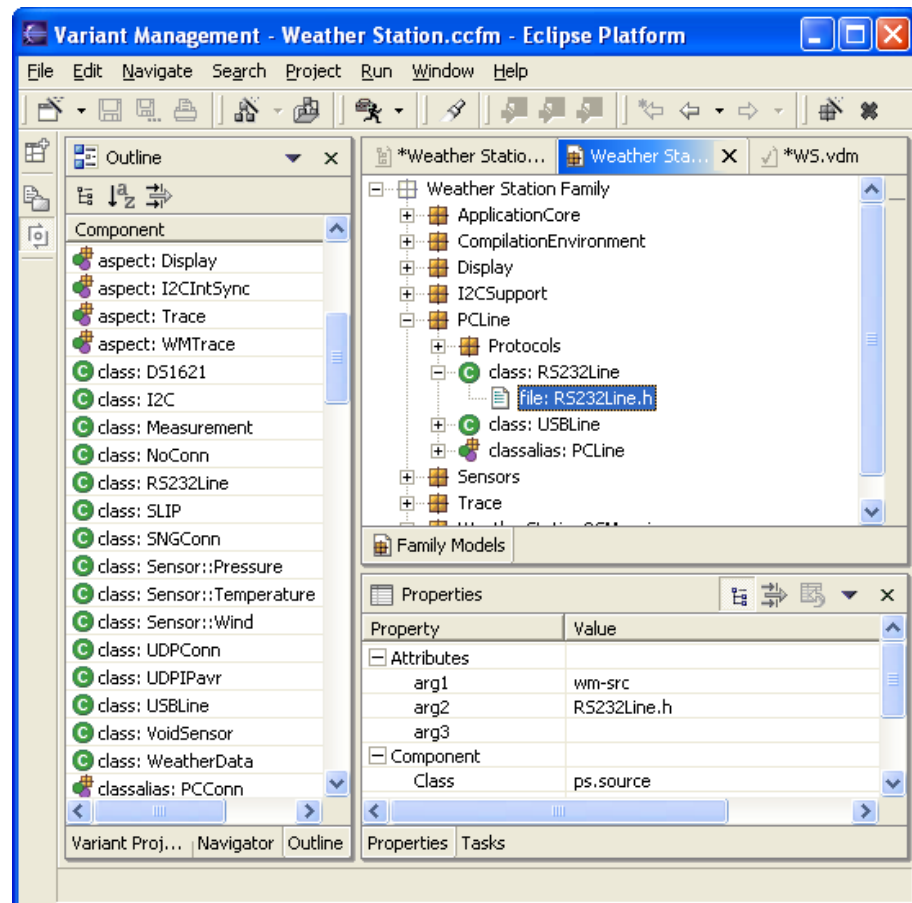
Feature Type	feature type is one of <i>ps:mandatory</i> (default), <i>ps:optional</i> , <i>ps:alternative</i> or <i>ps:or</i> .
Default feature	this property is used during model evaluation when the Auto Resolve option is active. If a default features parent feature is selected then the default feature is also selected if no other feature from the feature group has been selected. However, this behaviour only applies to the <i>ps:alternative</i> and <i>ps:or</i> feature types.
Range	for "ps:or" features it is possible to specify the number of features that have to be selected in a valid configuration in terms of a range expression. These range expressions can either be a number e.g. 2, or an inclusive number range given in square brackets e.g. [1,3] or a set of number ranges delimited by commas e.g. [1,3], [5, 8]. The asterisk character * or the letter n may be used to indicate that the upper bound is equal to the number of elements in the <i>ps:or</i> group.

Family Model Editor

The component family model editor shows a tree view of the components, parts, and source elements of a solution space. Each element in the tree is shown with an icon representing the type of the element (see Table 6.8, "Predefined part types"). The element may additionally be decorated with the restriction sign  if it has associated restriction rules.

For more information on family model concepts see the section called "Family Models".

Figure 5.10. Open family model editor with outline and property view



The Properties dialog is similar to that for Feature models but has an additional restrictions page. See the section called “ Restrictions in Component Family Models ” for more information on that topic.


Variant Description Model Editor

The variant description model editor is used to specify the configuration of an individual product variant. This editor allows the user to make and validate feature selections, to set attribute values, and to exclude model elements from the configuration.

In this editor, a tree view shows all feature models in the configuration space. A specific feature can be included in the configuration by marking the check box next to the feature. Additional editing options are available in a context menu.

Features may also be selected automatically e.g. by Auto Resolver. However, the context menu allows the exclusion of a feature; this prevents the Auto Resolver from selecting the feature.

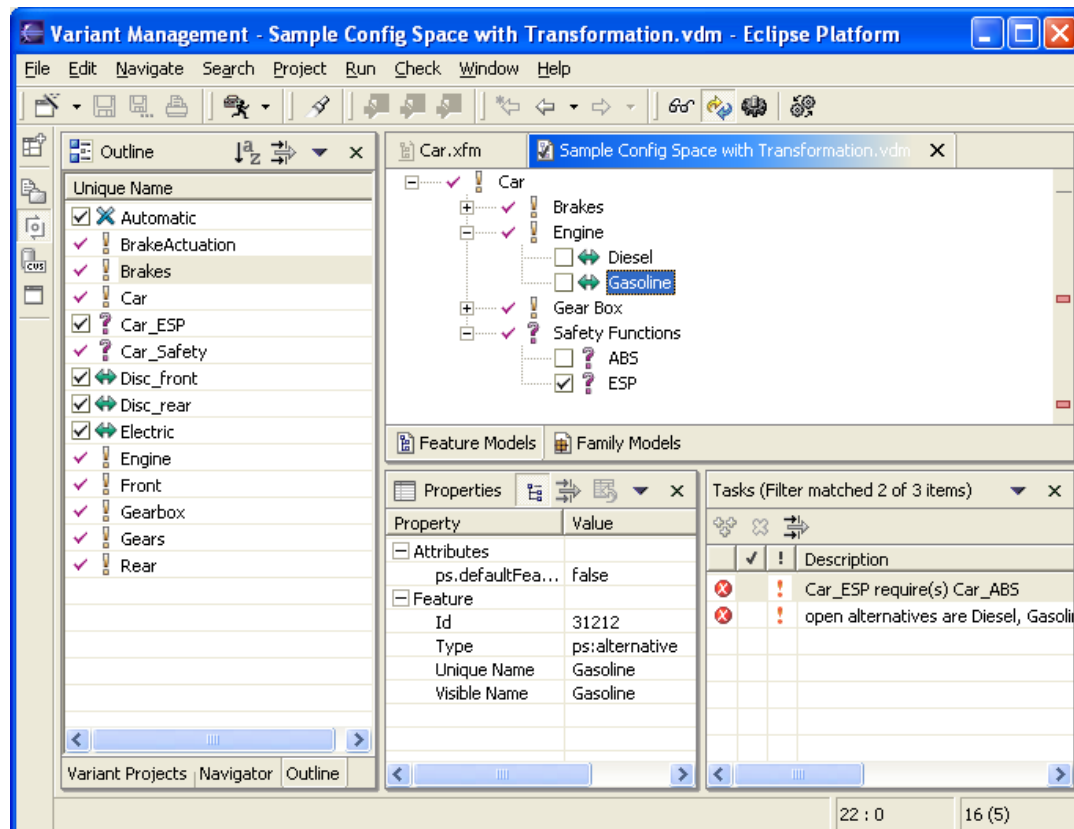
Each selected feature is shown with an icon indicating how the selection was made. The different types of icon are documented in Table 6.10, “ Types of feature selections ”. If the user selects a feature that has already been selected automatically its selection type becomes user selected and only the user can remove the selection.

When the  icon is shown instead of the selection icon, the selection of the feature is inadvisable since it will probably cause a conflict.

All configuration problems are shown with problem markers on the right side of the editor window and in the tasks/problems view.

Figure 5.11, “Open variant description model editor with outline and tasks view ” shows a sample Variant Description model. Note the different icons for implicit and user selection of features and the problems indicated in the right sidebar.

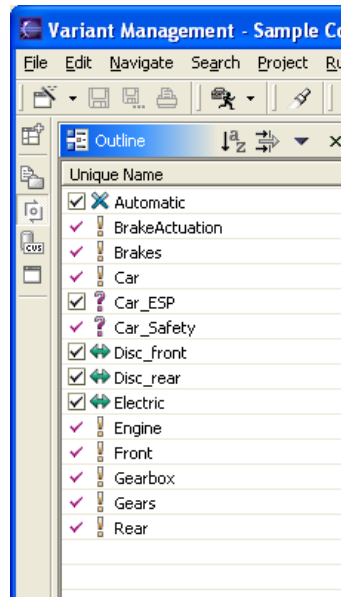
Figure 5.11. Open variant description model editor with outline and tasks view



Outline View for Variant Description Model Editor - Feature Set

The outline view of the variant description model shows the selected features with their selection state. This view may be filtered from the views filter icon or context menu.

Figure 5.12. Outline view showing the list of available features in a variant description model



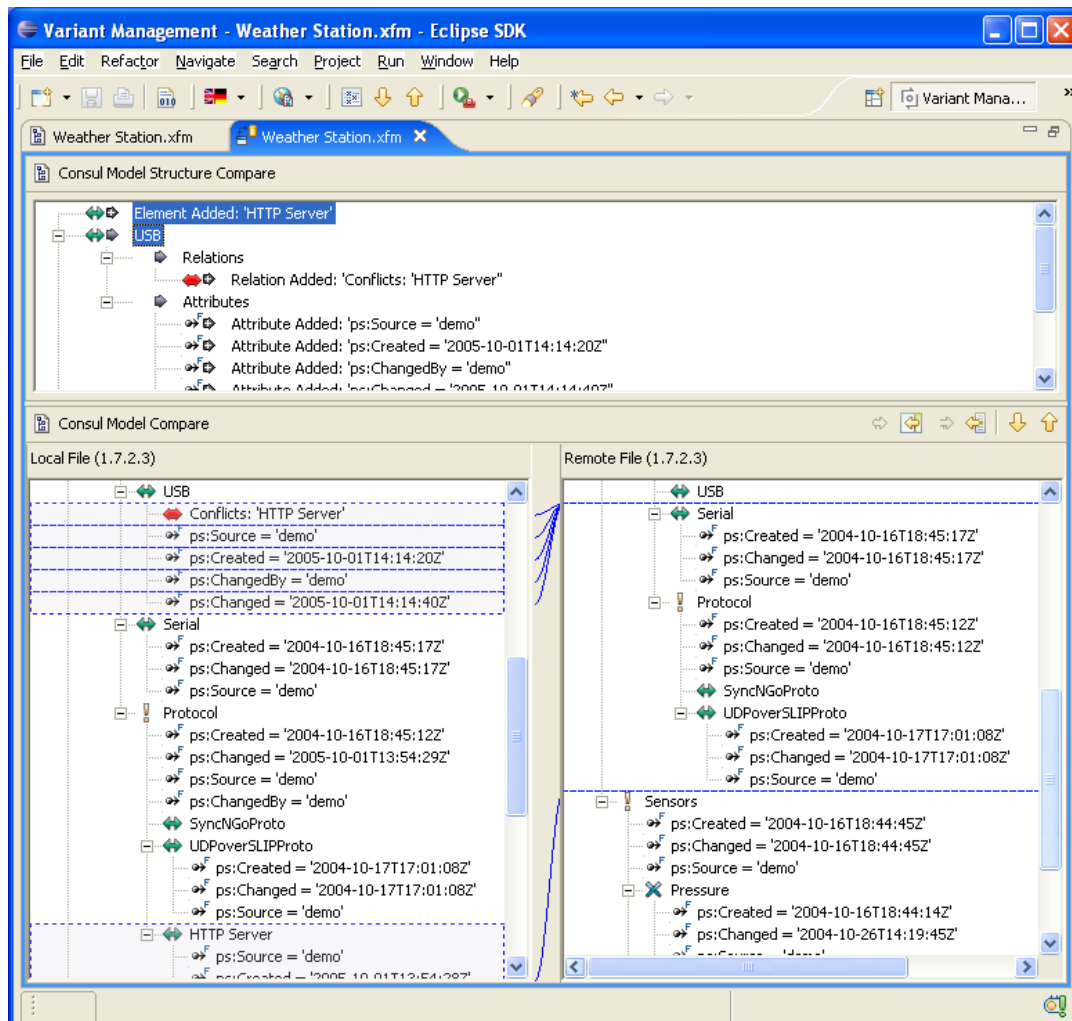
Compare Model Editor

Different versions of feature or family models may be compared using the Eclipse “compare” action. The compare editor permits the merging of changes from one model version to another model version. Depending the type of comparison (e.g. local file against version in CVS or two local files), merges may be permitted in both directions or only in one direction.

The figure Figure 5.13, “ Model Compare Editor ” shows a compare editor. The local model has been changed after being checked out from the CVS repository with version 1.7.2.3 . The upper part of the editor shows a list representation of the differences between both models. The topmost items always represent elements (features or family model elements). Changes made to an element are listed under element. Changes are categorized into changes to *general* element properties, *attributes*, *relations*, *restrictions* and other *miscellaneous* information.

The lower part of the editor is split into two windows, each representing one of the models to be compared. The highlighted areas in these windows represent the change locations for all changes selected above. In the figure the change items for the elements 'HTTP Server' and 'USB' are selected and marked. Buttons in the middle of the editor can be used to perform the change merging (copy one/all changes, goto next/previous change). For further information please refer to the Eclipse Workbench User Guide. The “Task” section contains a subsection “Comparing resources” which explains the use of compare editors in detail.

Figure 5.13. Model Compare Editor



Merging Restrictions

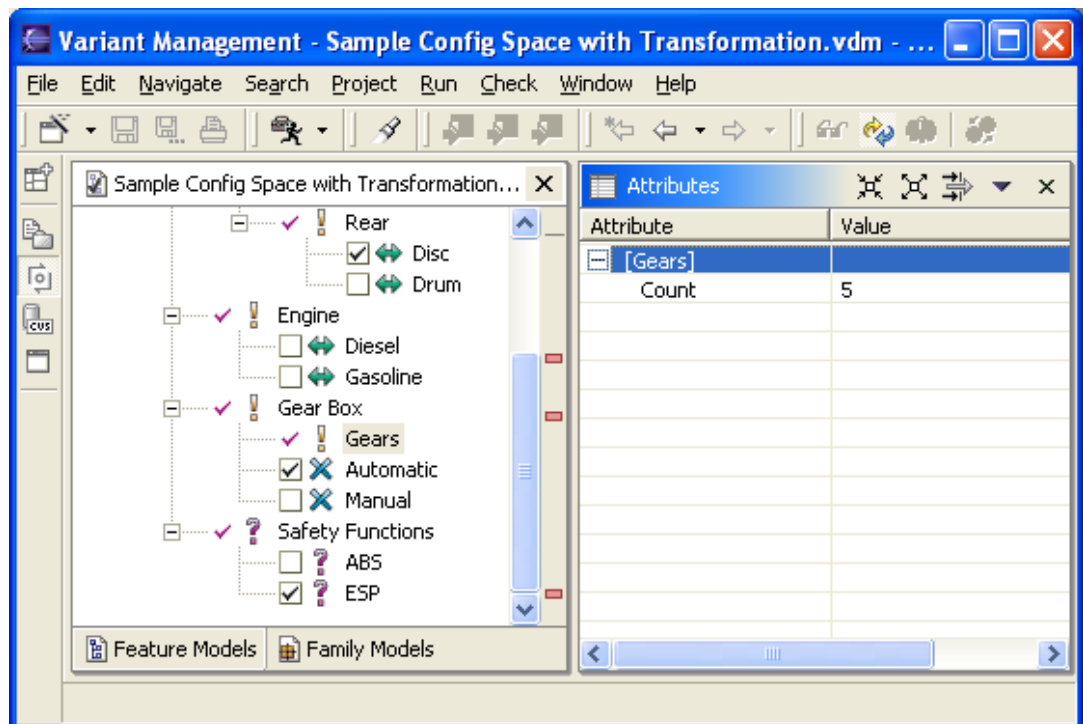
In contrast to purely text-based merge scenarios some changes cannot be merged without other changes previously merged into the model due to the internal structure of pure::variants models. For example, it is not possible to undo an element removal if its parent has also been removed. The user must merge the parent element first before the removed element can be successfully put back into the model. When trying to execute such a change, the editor will inform the user and prevent merges which would corrupt the internal model structure.

Views

Attributes View

The attributes view for a Variant Description model shows the available attributes for the models associated feature models. The user can set the values of non-fixed attributes in this view. This view may also be filtered to show only the attributes of selected features and/or where no values have been set.

Figure 5.14. Attributes view (right) showing the attribute Count for feature Gears



Filter View

Model editors may be filtered using Named Filters. The Filter view shows all currently available named filters. Additional filters may be imported from a file by using a context menu. To apply a filter to a model editor, select the model editor and then double-click on the filter name in the Filter view. Filters may also be deleted, renamed or exported using context menu commands. Exported filters are stored in a file, which can be imported into another Eclipse installation or shared in the project's team repository.

Outline View

The Outline view shows information about a model and allows navigation around a model. The outline view for some models has additional capabilities. These are documented in the section for the associated model editor.

Problem View/Task view

pure::variants uses the standard Eclipse “Tasks” (Eclipse 2.1.x) or “Problems” (Eclipse 3.0) view to indicate problems in models. If more than one element is causing a problem, clicking on the problem selects the first element in the editor. For some problems a “Quick fix” (see context menu of task list entry) may be available. Eclipse 3.0 users must open the “Problems” view manually (Window->Open View->Other->Basic->Problems view).

Properties View

pure::variants uses the standard Eclipse “Properties” view. This view shows important information about the selected element and allows editing of most property values.

Relations View


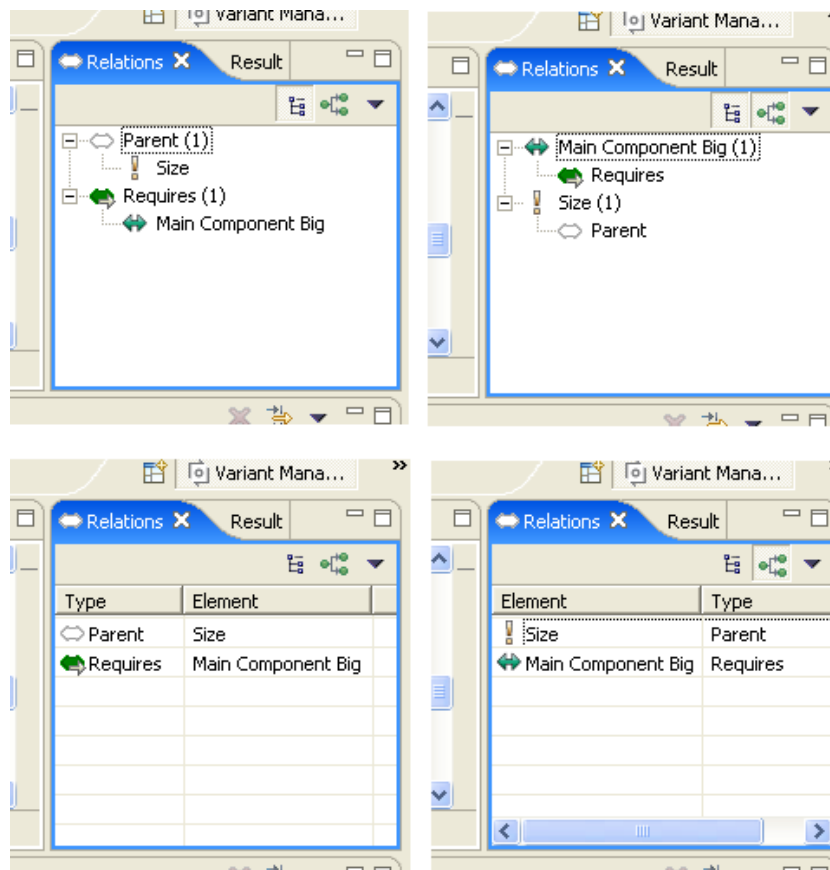
The Relations view shows the relation of the currently selected element (feature/component/part/source element) to other elements. Double-clicking on a related element selects that element in the editor. The small arrow in the lower part of the relation icon shows the direction of the relation. This arrow always points from the relation source to the relation destination. For some relations the default icon  is shown. The number in parentheses shown after an elements name is the count of child relations. So, in the figure below the element has one requires relation indicated by “(1)”.

Figure 5.15. Relations view (different layouts) for feature with a “ps:requires” to feature 'Main Component Big'






The Relations view is available in four different layout styles: - two tree styles combined with two table styles. These styles are accessed via icons or a menu on the Relations view toolbar.


The relations view supports filtering based on relation types. To filter the view use the Filter Types menu item from the menu accessible by clicking on the down arrow icon in the view's toolbar.

Result View

The result view shows the results of model evaluation after a selection check has been performed. It lists all selected features and family models elements representing the given variant.

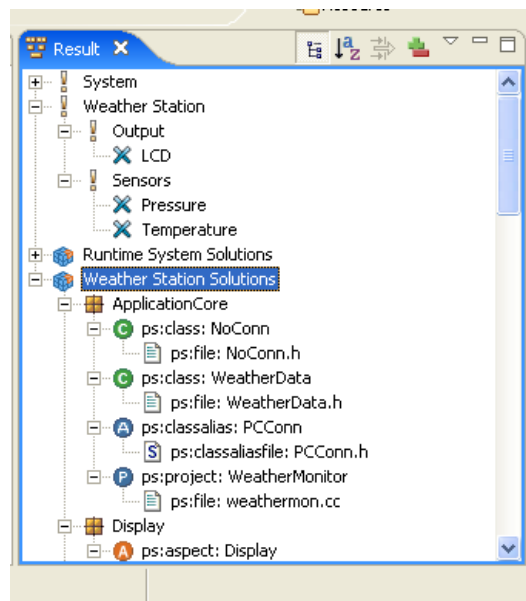
The result view also provides a special operation mode, where instead of a result, the difference (delta) between two results are shown, similar to the model compare capability for feature and family models.

Toolbar icons allow the view to be shown as a tree or table (), allow the sort direction to be changed (), and control activation/deactivation of the result delta mode ().


Filtering is available for the linear (table like) view, (). The “Model Visibility” item in the result view menu (third button from right in toolbar) permits selection of the models to be shown in the result view.

The result view displays a result corresponding to the currently selected VDM. If no VDM is selected, the result view will be empty. The result view is automatically updated whenever a VDM is evaluated.

Figure 5.16. Result View



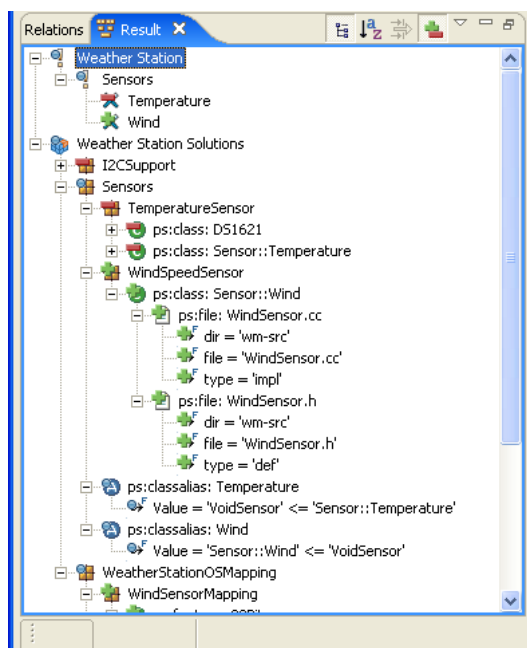
Result Delta Mode

The result delta mode is enabled with the plus-minus button () in the result view's toolbar. In this mode the view displays the difference between the current evaluation result and a *reference result* - either the result of the previous evaluation (default) or an evaluation result set by the user as a fixed reference. In the first case, the reference result is updated after each evaluation to become the current evaluation result. The delta is therefore always calculated from the last two evaluation results. In the second case the reference result does not change. All deltas show the difference between the current result and the fixed refer-

ence result.

The fixed reference can be either set to the current result or can be loaded from a previously saved variant result (a .vrm file). The reference result is set from the result view menu (third button from right in toolbar). To set a fixed result as reference use “Set current result as reference”. To load the reference from a file use “Load reference result from file”. To activate the default mode use “Release reference result”. The “Switch Delta Mode” sub-menu allows the level of delta details shown to be set by the user.

Figure 5.17. Result View in Delta Mode



Icons are used to indicate if an element, attribute or relation was changed, added or removed. A plus sign indicates that the marked item is only present in the current result. A minus sign indicates that the item is only present in the reference result. A dot sign indicates that the item contains changes in its properties or its child elements. Both old and new values are shown for changed attribute values (left hand side is new, right hand side is old).

Variant Feature Matrix view


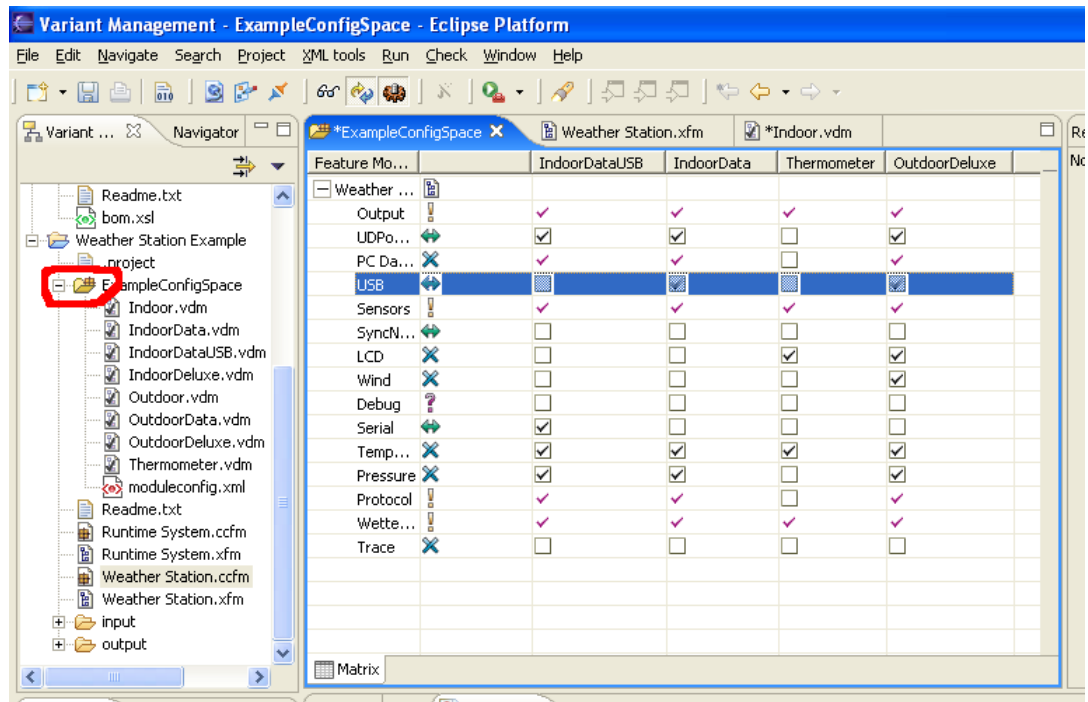
The feature matrix view gives an overview of feature selections and attribute values across the variants in a configuration space. The view is opened by double-clicking on the configuration space icon  in the “Variant projects” view (see Figure 5.18, “Feature Matrix view of a configuration space”). The view may be filtered based on the selection states of features in the individual Variant Description models: one filter shows the features that have not been selected in any model, one filter shows the features that have been selected in all models, and one filter shows the features that have been selected in at least one model. The filters are accessed via the context menu for the view (Show elements). The general filtering mechanism can also be used to further specify which features are visible (also accessible from the context menu).

Figure 5.18. Feature Matrix view of a configuration space

Variant Projects View

The Variant Projects View (upper left part in Figure 5.2, “Variant management perspective standard layout”) shows all variant management projects in the current workspace. Wizards available from the project's context menu allow the creation of feature models, family models, and configuration spaces. Double-clicking on an existing model opens the model editor, usually shown in the upper right part of the perspective. In Figure 5.2, “Variant management perspective standard layout” two editors are shown. The left one shows a feature model, the right one the corresponding variant description model with some features selected.

Property Editors

Configuration Space

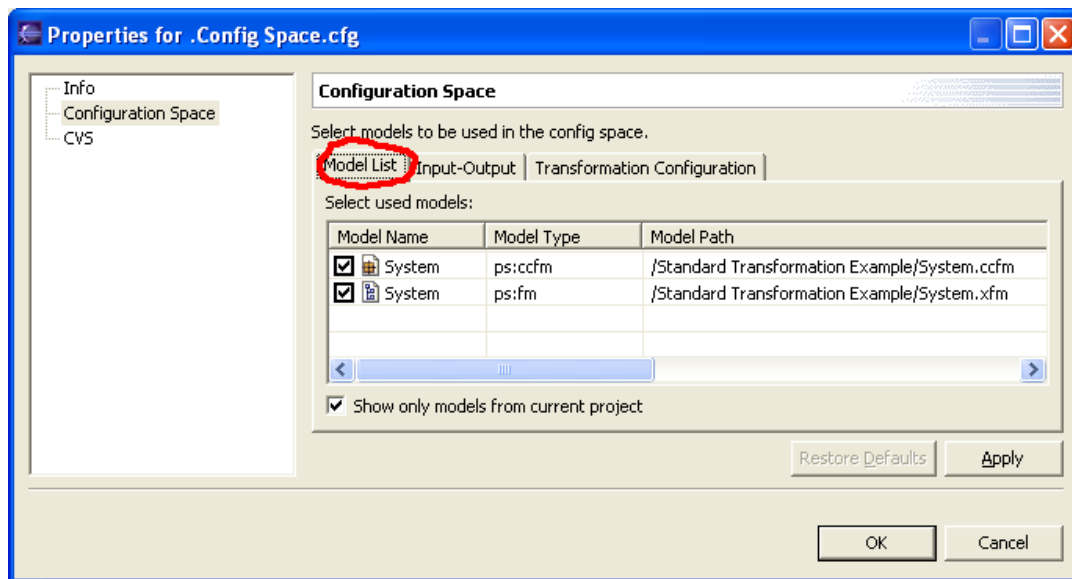
The configuration space properties dialog can be opened from the Variant Projects view using a context menu Configuration Space option. This option is only available for the Variant Projects view.

The dialog is divided into three separate pages a Model List page, an Input-Output page and a Transformation Configuration page.

Model List Page

This page is used to specify the list of models to be used in the configuration space. At least one model must be selected. By default, only models that are located in a configuration spaces project are shown.

Figure 5.19. Configuration space properties: Model Selection



Input-Output Page

This page is used to specify certain input and output options to be used in model transformations. The page need not be used for projects with no transformations.

The input path is the directory where the input files for the transformation are located. The output path specifies the directory where to store the transformation results. The modul base path is used when looking up module parameters specifying relative paths. All path definitions may use the following variables. The variables are resolved by the transformation framework before the actual transformation is started.

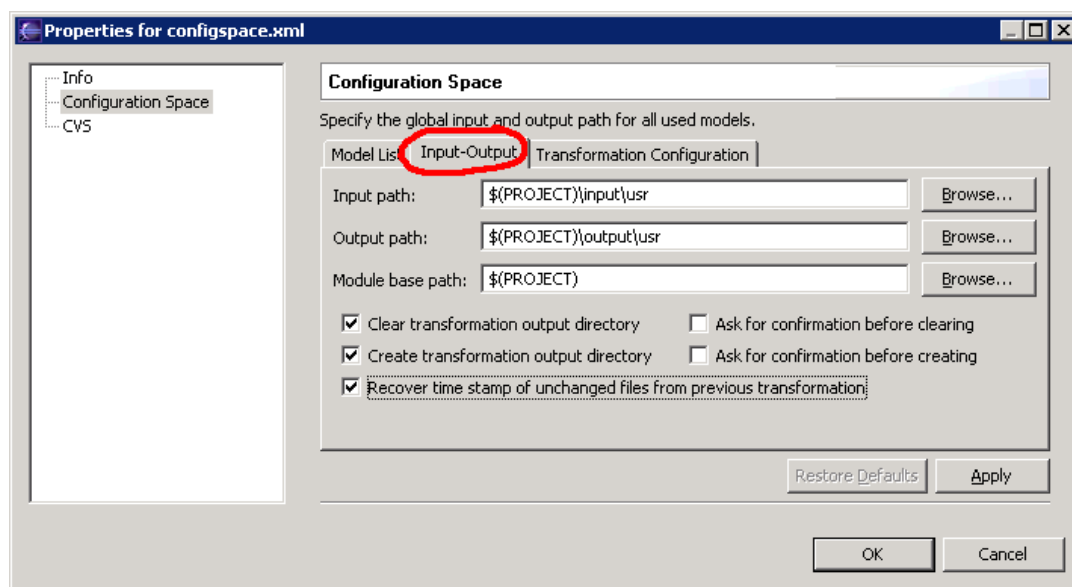
Table 5.2. Variables available for path resolution in transformations

Variable Name	Variable Content
\$(WORKSPACE)	The absolute path to the Eclipse workspace directory
\$(PROJECT)	The absolute path to the project directory
\$(INPUT)	The absolute path defined by the “Input path” option
\$(OUTPUT)	The absolute path defined by the “Output path” option
\$(MODULEBASE)	The absolute path defined by the “Module base path” option
\$(VARIANT)	The name of the variant model used for the transformation

The “Clear transformation output directory” checkbox controls whether pure::variants removes all files and directories in the Output path before a transformation is started. The “Ask for confirmation before clearing” checkbox controls whether the user is asked for confirmation before this clearing takes place. The remaining checkboxes work in a similar manner and control what happens if the Output path does not exist when a transformation is started.

The “Recover time stamp...” option instructs the transformation framework to recover the time stamp values for output files whose contents has not been changed during the current transformation. I.e. even if the output directory is cleared before transformation, a newly generated or copied file with the same contents retains its “old” time stamp. Enable this option if you use tools like “make” which use the files time stamp to decide if a certain file changed.

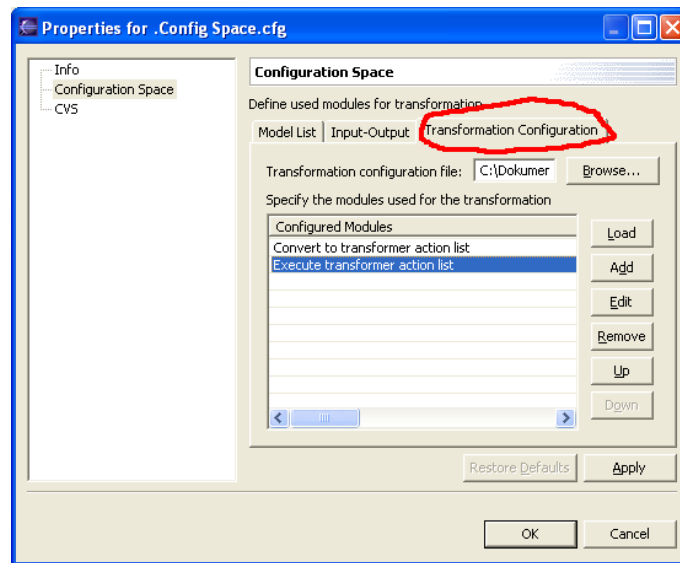
Figure 5.20. Configuration space properties: Transformation input/output paths



Transformation Configuration Page

This page is used to define the model transformation to be performed for the configuration space. The transformation is stored in an XML file. If the file has been created by using the wizards in pure::variants it will be named moduleconfig.xml and will be placed inside the configuration space. However, there is no restriction on where to place the configuration file, it may be shared with other configuration spaces in the same project or in other projects, and even with configuration spaces in different workspaces.

Figure 5.21. Configuration space properties: Transformation Configuration



Buttons on the right allow transformation modules to be added to or removed from the configuration and to be edited. When adding or editing a transformation module a wizard helps the user to enter or change the modules configuration. Since many modules have dependencies on other modules they must be executed in a specific order. The order of execution of the transformation modules is specified by the order in the Configured Modules list, this order can be changed using the Up and Down buttons. Please see the section called “ The XML Transformation Engine XMLTS ” for more information on model transformation.

Model Export and Import

Export

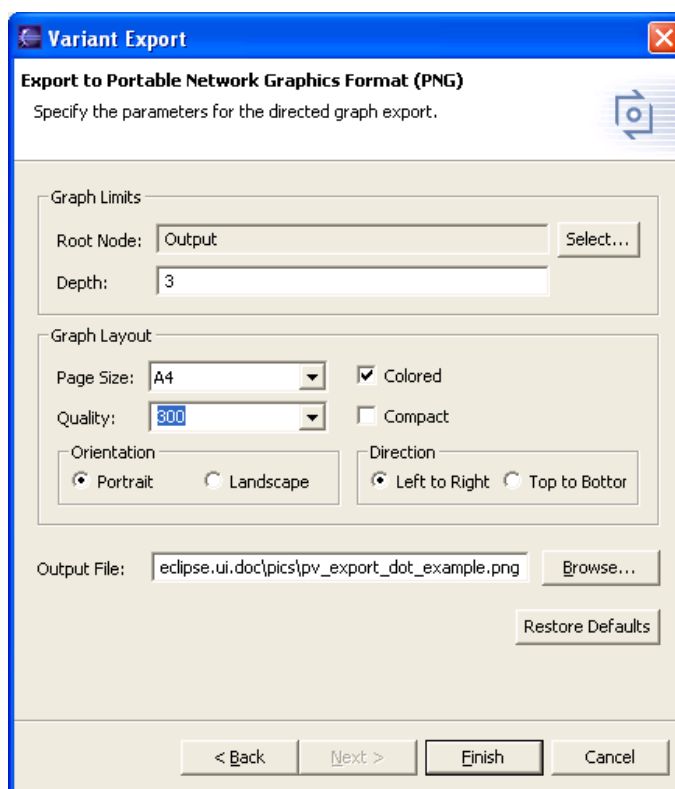
Models may be exported from pure::variants in a variety of formats. An Export item is provided in the Navigator and Variants Project views context menus and in the File menu. Select Variant Resources and choose one of the provided export formats.

Currently supported export data formats are HTML, XML, CSV and Directed Graph. The Directed Graph format is only supported for some models. Additional formats may be available if other plug-ins have been installed.

HTML export format is a hierarchical representation of the model. XML export format is an XML file containing the corresponding model unchanged.

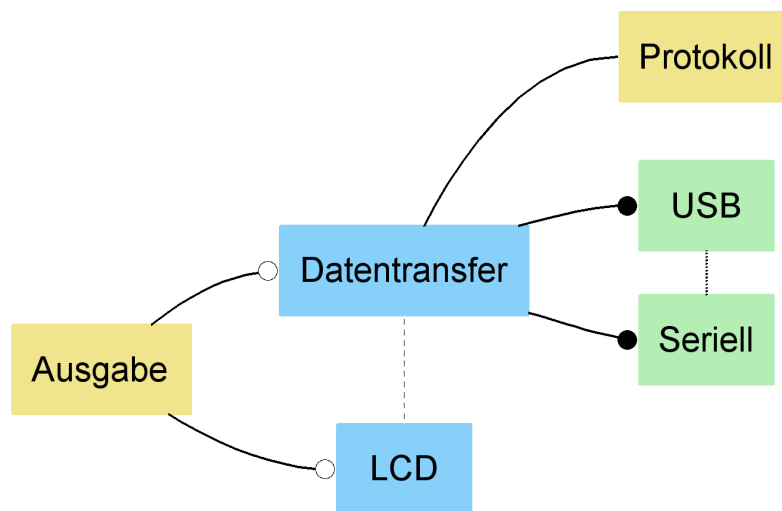
CSV export format is a text file that can be opened with most spreadsheet programs (e.g. Microsoft Excel or OpenOffice). CSV export respects filters, i.e. only matching elements are exported. The export wizard permits the columns to be generated in the output file to be selected.

Figure 5.22. Directed Graph Export Output Configuration Dialog



The directed graph export format generates a graph in the DOT language. This can be used for generation of images for use in documentation or for printing. If the DOT language interpreter from the GraphViz package (<http://www.graphviz.org/>) is installed in the computers executable path or the packages location is provided as a preference (Windows->Preferences->Variant Management->Directed Graph Export), many image formats can be generated directly. The dialog shown in Figure 5.22, “Directed Graph Export Output Configuration Dialog” permits many details of the output, such as paper size or the layout direction for the model graph, to be specified. Graphs for sub-models may be exported by setting the root node to any model element. The “Depth” field is used to specify the distance below the root node beyond which no nodes are exported. The “Colored” option specifies whether feature models are exported with a colored feature background indicating the feature relation (yellow=“ps:mandatory”, blue=“ps:or”, magenta=“ps:option”, green=“ps:alternative”). Figure 5.23, “Directed graph export example (options LR direction, Colored)” shows the results of a feature model export using the Left to Right graph direction and Colored options.

Figure 5.23. Directed graph export example (options LR direction, Colored)



Import

An Import item is provided in the Navigator and Variants Project views context menus and in the File menu. Select Variant Models or Projects and choose one of the provided import sources.

Currently only a generic family model import from source directories is provided. This import creates a family model or parts of a family model from an existing directory structure of Java or C/C++ source code files. Additional formats may be available if other plug-ins are installed.

Chapter 6. Reference

Abbreviations

The following abbreviations are used in this section:

AID	attribute id. The full id path of the attribute (modelId/attributeId).
AN	attribute name. This can be the name of an attribute or the full id path (modelId/attributeId).
CID,PID,SID	component/part/source id. This must be the full id path (modelId/elementId).
CN,PN,SN	component/part/source name. This can be the unique name of the component/part/source or the full id path (modelId/elementId).
EN	element name (can point to any element type). This can be the unique name of the element or the full element id path (modelId/elementId).
EID	element id. This must be the full id path (modelId/elementId).
EL	element id list. List of elements given as full id paths (modelId/elementId).
FID	feature id. This must be the full feature id path (modelId/featureId).
FN	feature name. This can be the unique name of the feature or the full feature id path (modelId/featureId).

Feature/Element Relations

Table 6.1. Supported Relations between Features/Element

Relation	Description
ps:requires(EL)	At least one of the specified targets in EL has to be selected when the source is selected.
ps:conflicts(EL)	If all specified targets are selected, the source must not be member of the selection.
ps:recommends(EL)	Like ps:requires, but not treated as error (only notification in task view)
ps:discourages(EL)	Like conflicts, but not treated as error (only notification in task view)

General Model Restriction Language pvProlog

ps:conditionalRequires(EL)	Similar to requires. The “requires” relation is checked only for targets whose parent is currently selected.
ps:influences(EL)	The features in FL are influenced in some way by the selection of the feature. The interpretation of the influence is up to the user.
ps:requiredFor(EL)	If at least one of the specified targets in EL is selected, the source has to be selected too.
ps:recommendedFor(EL)	Like ps:requiredFor, but not treated as error (only notification in task view)

General Model Restriction Language pv-Prolog

The pure::variants restriction language *pvProlog* is a dialect of the Prolog programming language. However, pvProlog expressions more closely resemble those in languages such as OCL and Xpath, than expressions in Prolog do. In most cases the provided logical operators and functions are sufficient to specify restrictions. If more complicated computations have to be done, the full power of the underlying Prolog engine can be used. This, however, will require a manual expression conversion when switching between OCL and Prolog in future versions of pure::variants. See <http://www.swi-prolog.org> for more information on SWI-Prolog syntax and semantics.

pvProlog Logic Expressions

[1]	Expr	::=	Func UnaryOpExpr OpExpr '(' Expr ')'
[2]	OpExpr	::=	Expr BinOp Expr
[3]	UnaryOpExpr	::=	UnaryOp '(' Expr ')'
[4]	BinOp	::=	'xor' 'equiv' 'and' 'implies' 'or'
[5]	UnaryOp	::=	'not'
[6]	Func	::=	FuncName '(' Args ')'
[7]	Args	::=	Argument Args ',' Argument
[8]	Argument	::=	String Number
[9]	String	::=	""[.]*""
[10]	Number	::=	['+' '-']? ['0'-'9']+ ['.' ['0'-'9']+]
[11]	FuncName	::=	['a'-'z'] ['a'-'z' 'A'-'Z' '0'-'9' '_']*

Table 6.2. Logic operators in pvProlog

Name/Symbol	Assoc.		Description
xor	right	Binary	logical exclusive or
equiv	none	Binary	not(A xor B)
and	left	Binary	logical and
implies	left	Binary	logical implication
or	left	Binary	logical or

Table 6.3. Functions in pvProlog

Name/Symbol	Description
not(EXP)	True if EXP is false

Table 6.4. Available rules for Value calculations and Restrictions

Rule	Description	Applicable in model
alternativeChild(FN, FN2) [deprecated alternative_child(FN, FN2)]	True, if the feature FN has an alternative group and one of the alternative features is in the current feature selection. FN2 is unified with the selected alternative feature name.	family model
isElement(EID)	True if the element with id EID is found in a feature or family model.	all
isFamilyModelElement(EID)	True if the element with id EID is found in a family model.	
isFeatureModelElement(EID)	True if the element with id EID is found in a feature model.	all
hasAttribute(AID) hasAttribute(EN, AN)	These methods check the existence of a definition for the specified attribute. Attribute is iden-	all

**General Model Restriction
Language pvProlog**

<p>hasAttribute(ET, EN, AN)</p> <p>hasAttribute(EC, ET, EN, AN)</p>	<p>tified by its id (AID), by the symbolic name of its associated element and its symbolic name (EN, AN) or similarly by additionally specifying the element type ET. To ensure correct operation of hasAttribute variants using symbolic names, symbolic element names EN must be unique inside the configuration space or inside the element space of the configuration space [(ET, EN), (EC, ET, EN)] and the symbolic attribute name AN must be unique inside the attribute space of the element.</p>	
<p>getAttribute(AID, VALUE)</p> <p>getAttribute(EN, AN, VALUE)</p> <p>getAttribute(ET, EN, AN, VALUE)</p> <p>getAttribute(EC, ET, EN, AN, VALUE)</p>	<p>These methods get or check the existence and value of the specified attribute. Attribute is identified by its id (AID), by the symbolic name of its associated element and its symbolic name (EN, AN), or similar by additionally specifying the element type ET. When VALUE is a constant, getAttribute checks that the attribute has the specified value. If VALUE is a variable, then subsequent rules can access the attribute's value using the specified variable name. To ensure correct operation of hasAttribute variants using symbolic names, symbolic element names EN must be unique inside the configuration space or inside the element space of the configuration space [(ET, EN), (EC, ET, EN)] and the symbolic attribute name (AN) must be unique inside the attribute space of the element.</p>	all
<p>getAttributeName(AID, ANAME)</p>	<p>ANAME is unified with the attribute name of the attribute specified with AID.</p>	
<p>getAttributeType(AID, ATYPE)</p>	<p>ATYPE is unified with the meta-model attribute type of the attribute specified with AID.</p>	
<p>isTrue(VALUE)</p>	<p>If VALUE is equal to the internal representation of the 'true' value for an attribute of type</p>	<p>Example usage in a restriction: getContext(EID) and getAttribute(EID, 'ABoolean', BV) and is-</p>

**General Model Restriction
Language pvProlog**

	'ps:boolean', it will evaluate to true.	True(BV)
isFalse(VALUE)	If VALUE is not equal to the internal representation of the 'true' value for an attribute of type 'ps:boolean', it will evaluate to true.	
getVariantId(MID)	MID is unified with the unique id of the variant description model (.vdm) currently being evaluated.	
getModelList(MIDL)	MIDL is unified with the list of all models currently being evaluated. This gives access to ids of the feature models, family models and variant description models in the current configuration space	
getElementModel(EID,MID) getElementModel(MID)	MID is bound to the model id associated with the unique element id EID. If EID is not given, the context element is used as EID.	
userMessage(TYPE, STRING, RELATEDEIDS, CONTEXTEID)	Issues a problem message, to be shown, for example, in the Eclipse problems view. TYPE is one of {'error', 'warning'} STRING is the text which describes the problem RELATEDEIDS is a list of elements with some relation to the problem CONTEXTEID is the id of the element that caused the problem.	Example: userMessage('error','Something happened',MYEID,[REID1,REID2])
userMessage(TYPE,STRING,RELATEDEIDS)	Issues a problem message as above but automatically sets the current element to be the context element.	
warningMsg(STRING, RELATEDEIDS) errorMsg(STRING, RELATEDEIDS)	Convenience methods for userMessage, sets TYPE automatically.	
warningMsg(STRING) errorMsg(STRING)	Convenience methods for userMessage, set TYPE automatically and uses empty RELATEDEIDS list.	errorMsg('An unknown error occurred')

**General Model Restriction
Language pvProlog**

getContext(EID) getSelf(SELF) getContext(EID,SELF)	These methods can be used to determine the restriction/calculation context. EID is bound to the unique id of the element that is the immediate ancestor of the restriction or calculation. So, inside an attribute calculation it will be bound to the id of the element containing the attribute definition. SELF is the unique id of the calculation/restriction itself.	Access the attribute 'X' of the same element in a calculation: getContext(EID), getAttribute(EID, 'X', XValue)
getElementChildren(EID,CEIDS)	CEIDS is unified with the list of children of the element specified with EID or an empty list if no children exist.	
getElementParents(EID,PARIDS)	PARIDS is unified with the list of parents of the element specified by EID or an empty list if no parents exist.	
getElementRoot(EID,ROOTID)	ROOTID is the root element for the element specified by EID. For elements with several root elements only one is chosen.	
getElementName(EID,ENAME)	ENAME is unified with the unique name of the element specified with EID.	
getElementClass(EID,ECLASS)	ECLASS is unified with the type model element class of the element specified with EID.	The standard meta model uses the classes ps:feature, ps:component, ps:part and ps:source
getElementType(EID,ETYPE)	ETYPE is unified with the type model element type of the element specified with EID.	
getMatchingElements(MatchExpr,MEIDS) getMatchingElements(CTXID,MatchExpr,MEIDS)	MEIDS is unified with a list of all the elements which comply with the specified match expression MatchExpr. The context of the match expression is the current element context (see getContext) unless CTXID is used to specify a different context. Match expressions are explained below.	Put all features below the current element with unique names starting with "FEA_X" into LIST use getAttribute('**'.FEA_X*',LIST).
getMatchingAttrib-	AIDS is unified with all attrib-	

Additional Restriction Rules for Variant Evaluation

<code>getMatchingElements(MatchTokenExpr, EID, AIDS)</code>	<p>Matches the elements of the element specified with the unique id EID which match with the pattern in MatchTokenExpr. The match pattern is the same as for getMatchingElements, but it must not contain dot characters. Match expressions are explained below.</p>	
<code>subnodeCount(ECLASS,ENAME,COUNT)</code> <code>subnodeCount(ECLASS,ETYPE,ENAME,COUNT)</code> <code>subnodeCount(EID,COUNT)</code>	<p>These methods count the number of selected children of a given element. COUNT is bound to the number of selected child elements. Whether the element itself is selected is not checked.</p>	<p>Example: A restriction checking whether three children of component 'X' are selected: <i>subnodeCount('ps:component', 'X', 3)</i></p>
<code>subfeatureCount(FNAME,COUNT)</code>	<p>COUNT is bound to the number of selected child features of feature FNAME. Convenience method for subnodeCount('ps:feature',_,FNAME,COUNT).</p>	
<code>singleSubfeature(FNAME)</code>	<p>True if feature FNAME has just a single child. Convenience method for subnodeCount('ps:feature',_,FNAME,1).</p>	

Additional Restriction Rules for Variant Evaluation

Table 6.5. Additional rules available for variant evaluation

Rule	Description	Applicable
<code>hasElement(EID)</code> <code>has(EID)</code>	<p>True if the element EID is in the variant. Fails silently otherwise.</p>	<p>If hasElement is used inside restrictions and constraints inside feature models the element identified by EID has to be contained in models with higher ranks.</p> <p>If used in family model the element has to be in feature models of the same rank or in any model of higher ranks.</p>

Additional Restriction Rules for Variant Evaluation

hasFeature(FN), [deprecated: has_feature]	True if the feature FN is found in the current set of selected features. Fails silently otherwise.	see hasElement
hasComponent(CN), hasPart(PN), hasSource(SN) [deprecated: has_component, has_part, has_source]	True if the component/ part/source xN is found in the current set of selected components in the current component configuration. Fails silently otherwise.	see hasElement. hasPart may also refer to components from same family model, hasSource may also refer to parts from same model.
requiresFeature(FN FNL) [requires_feature(FN)]	True if the feature FN, or at least one feature of the features in the list FNL, is found in the current set of selected features. Issues an error message and fails otherwise. May be used to request inclusion of specific features in the set.	see hasElement
requiresComponent(CN CNL) [requires_component(CN CNL)]	True if the component CN or at least one component from the list CNL is found in the current set of selected components. Issues an error message and fails otherwise. May be used to request inclusion of specific components.	see has_component
conflictsFeature(FN FNL) [conflicts_feature(FN FNL)]	True if the feature FN or at least one feature from the list FNL is not found in the current set of selected features. Issues an error message and fails otherwise. May be used to prevent inclusion of specific feature.	see hasElement
conflictsComponent(CN CNL) [conflicts_component(CN CNL)]	True if the component CN or at least one component from the list CNL is not found in the current set of selected components. Issues an error message and fails if component is found. May be used to prevent inclusion of specific components.	see has_component
getAllSelectedChildren(EID, IDL)	Binds IDL to contain all selected children and children of children below and not including EID.	EID must be an element of a model with the same or higher rank when this rule is used in attribute calculations. EID must be an element of a model with high-

Match Expression Syntax for getMatchingElements

		er rank when used in restrictions. In family model restrictions EID can also be an element of the same model rank.
getMatchingSelectedElements(MatchExpr,MEIDS) getMatchingSelectedElements(CTXID,MatchExpr,MEIDS)	Similar to getMatchingElement described above, but the list is unified only with the elements which are in the current configuration.	
sumSelectedSubtreeAttributes(EID,AN,Value)	Calculates the numerical sum of all attributes with the name AN for all selected elements below element with id EID not including the elements attributes itself.	see getAllSelectedChildren

Match Expression Syntax for getMatchingElements

The `getMatchingElements` rules use simple match expressions to specify the elements. A match expression is a string. Match expressions are evaluated relative to a given context or absolutely (i.e. starting from the root element of the context element) when the expression's first character is a dot `'.'`. The expression is broken into individual matching tokens by dots `'.'`.

Each token is matched against all elements at the given tree position. The first token is matched against all children of the context or all children of the root element in the case of absolute paths. The second token is matched against children of the elements which matched the first token and so on.

Tokens are matched against the unique names of elements.

The match pattern for each token is very similar the the Unix *cs*h pattern matcher but is case insensitive, i.e. the pattern `V*` matches all names starting with small `v` or capital `V`. The following patterns are supported.

- ? Matches one arbitrary character.
- *
- Matches any number of arbitrary characters.
- [...]
- Matches one of the characters specified between the brackets. `<char1>-<char2>` indicates a range.
- {...}
- Matches any of the patterns in the comma separated list between the braces.
- **
- If the token is `**`, the remainder of the match expression is applied recursively for all subhierarchies below.

For example, path expression `'A?.BAR'` matches all elements named `BAR` below any element with a two letter name whose first letter is `A` and which is relative to the current context element. The expression `'**.D*'` matches all model elements whose unique name

starts with D and that are in the model of the context element.

The context element (or root element in an absolute expression) itself is never included in the matching elements.

Model Attributes

Information stored as model metadata can be accessed using the pvProlog function *getAttribute()*. The table below lists the available attributes and their meanings. Use the model id (from *getModelList*, *getVariantId*) as the EID. For example, to bind the name of the variant model to the variable VDMNAME in a calculation or restriction, use the following pvProlog expression:

```
getVariantId(VID), getAttribute(VID, 'name', VDMName)
```


Table 6.6. Meta-Model attributes in pvProlog

Attribute Name	Description
name	The descriptive name of the model.
date	The creation date of model.
version	An arbitrary user-defined string to identify the version of the model.
time	The creation time of the model.
author	The user who created the model.
file	The file name of the model (without directory path, see path below).
path	The absolute path leading to the model directory.

Feature Models

Supported Feature Types

Table 6.7. Feature tree relation types and icons








Short name	Relation Type	Description	Icon
mandatory	ps:mandatory	Mandatory features are selected if the parent feature is	

		selected.	
optional	ps:optional	Optional features are selected independently.	?
alternative	ps:alternative	Alternative features are organised in groups. At least one feature has to be selected from a group if the parent feature is selected (Although this can be changed using range expressions). pure::variants allows only one or group for the same parent feature.	↔
or	ps:or	Or features are organised in groups. At least one feature has to be selected from a group if the parent feature is selected (Although this can be changed using range expressions). pure::variants allows only one or group for the same parent feature.	✕

Family Models

Predefined Family Model Part Types

Table 6.8. Predefined part types

Part type	Description	Icon
ps:class	Maps directly to a class in an object-oriented programming language.	
ps:classalias	Different classes may be mapped to a single class name. Value restrictions must ensure that in every possible configuration only one class is assigned to the alias.	
ps:object	Maps directly to an object in an object-oriented programming language.	
ps:variable	Describes a configuration variable name, usually evaluated in make files. The variable can have a value assigned.	
ps:flag	A synonym for ps:variable. This part type maps to a source code flag. A flag can be undefined or can have an associated value that is calculated at configuration time. ps:flag is usually used in conjunction with the flagfile source element, which generates a C++-preprocessor #define <flagName> <flagValue> statement in the specified file.	
ps:project	ps:project can be used as the part type for anything that does not fit into other part types.	
ps:aspect	Maps directly to an aspect in an aspect-oriented language (e.g. AspectJ or AspectC++)	

Family Model Element Relations






Table 6.9. Supported Element Relations only supported in CCFM

Relation	Description	Use for	Partner relation
ps:exclusiveProvider(id)	In a valid configuration at most one exclusiveProvider for a given id is allowed. Thus, the relation defines a mutual exclusion relation between elements.	concurrent implementations for an abstract concept	requestsProvider
ps:requestsProvider(id)	In a valid configuration for each requestsProvider with the given id there must be an exclusiveProvider with the same id. There may be any number of requestsProvider relations for the same id.	Request existence of an abstract concept	exclusiveProvider
ps:expansionProvider(id)	In a valid configuration at most one expansionProvider for a given id is allowed. Thus, the relation defines a mutual exclusion relation between elements.	Provides mechanism for implementing variation points with a default solution.	defaultProvider
ps:defaultProvider(id)	If an element marked as expansionProvider is additionally marked as defaultProvider for the same given id and there is more than one possible element claiming to be an expansionProvider for this id, then the defaultProvider is excluded.		expansionProvider

Variant Description Models

Feature Selection List Entry Types

Table 6.10. Types of feature selections

Type	Description	Icon
user	Explicitly selected by the user. Auto resolver will never change the selection state of a user feature.	
auto resolved	A feature selected by the auto resolver to correct problems in the feature selection. Auto resolver may change the state of an auto resolved feature but does not deselect these features when the user changes a feature selection state.	
mapped	The auto resolver detected a valid feature-mapping request for this feature in a feature map and in turn selected the feature. The feature mapping selection state is automatically changed/rechecked when the user changes the feature selection.	
implicit	All features from the root to any selected feature and mandatory features below a selected feature are implicitly selected if not selected otherwise.	
excluded	The user may exclude a feature from the selection process (via a context menu). When the selection of an excluded or any children features of an excluded feature is required, an error message is shown.	

XSLT Extension Functions

The following extension functions are available when using the integrated XSLT processor in the pure::variants XML Transformation System for model transformations and model exports. The extension functions are defined in the namespace "http://www.pure-systems.com/purevariants".

Table 6.11. XSLT extension functions

Function	Description
<code>nodeset models()</code>	Get all models known to the transformer.
<code>nodeset model-by-id(string)</code>	Get all models known to the transformer having the given id.
<code>nodeset model-by-name(string)</code>	Get all models known to the transformer having the given name.
<code>nodeset model-by-type(string)</code>	Get all models known to the transformer having the given type. Valid types are "ps:vdm", "ps:cfm", and "ps:ccm".

XSLT Extension Functions

<code>boolean hasFeature(string)</code>	Return <i>true</i> if the feature, given by its unique name or id, is in the variant.
<code>boolean hasComponent(string)</code>	Return <i>true</i> if the component, given by its unique name or id, is in the variant.
<code>boolean hasPart(string)</code>	Return <i>true</i> if the part, given by its unique name or id, is in the variant.
<code>boolean hasSource(string)</code>	Return <i>true</i> if the source, given by its unique name or id, is in the variant.
<code>boolean hasElement(string id)</code>	Return <i>true</i> if the element, given by its unique id, is in the variant.
<code>boolean hasElement(string name,string class,string type?)</code>	Return <i>true</i> if the element, given by its unique name, class, and (optionally) type, is in the variant.
<code>nodeset getElement(string id)</code>	Return the element given by its unique id.
<code>nodeset getElement(string name,string class,string type?)</code>	Return the element given by its unique name, class, and (optionally) type.
<code>boolean hasAttribute(string id)</code>	Return <i>true</i> if the attribute, given by its unique id, is in the variant.
<code>boolean hasAttribute(nodeset element,string name)</code>	Return <i>true</i> if the attribute, given by its name and the element it belongs to, is in the variant.
<code>boolean hasAttribute(string eid,string name)</code>	Return <i>true</i> if the attribute, given by its name and the id of the element it belongs to, is in the variant.
<code>boolean hasAttribute(string ename,string eclass,string etype?,string name)</code>	Return <i>true</i> if the attribute, given by its name and the unique name, class, and (optionally) type of the element it belongs to, is in the variant.
<code>nodeset getAttribute(string id)</code>	Return the attribute given by its unique id.
<code>nodeset getAttribute(nodeset element,string name)</code>	Return the attribute given by its name and the element it belongs to.
<code>nodeset getAttribute(string eid,string name)</code>	Return the attribute given by its name and the id of the element it belongs to.
<code>nodeset getAttribute(string ename,string eclass,string etype?,string name)</code>	Return the attribute given by its name and the unique name, class, and (optionally) type of the element it belongs to.
<code>boolean hasAttributeValue(nodeset attribute)</code>	Return <i>true</i> if the given attribute has a value.

XSLT Extension Functions

boolean hasAttributeValue(string id)	Return <i>true</i> if the attribute given by its unique id has a value.
boolean hasAttributeValue(nodeset element,string name)	Return <i>true</i> if the attribute, given by its name and the element it belongs to, has a value.
boolean hasAttributeValue(string eid,string name)	Return <i>true</i> if the attribute, given by its name and the id of the element it belongs to, has a value.
boolean hasAttributeValue(string ename,string eclass,string etype?,string name)	Return <i>true</i> if the attribute, given by its name and the unique name, class, and (optionally) type of the element it belongs to, has a value.
string getAttributeValue(nodeset attribute)	Return the value of the given attribute.
string getAttributeValue(string id)	Return the value of the attribute given by its unique id.
string getAttributeValue(nodeset element,string name)	Return the value of the attribute given by its name and the element it belongs to.
string getAttributeValue(string eid,string name)	Return the value of the attribute given by its name and the id of the element it belongs to.
string getAttributeValue(string ename,string eclass,string etype?,string name)	Return the value of the attribute given by its name and the unique name, class, and (optionally) type of the element it belongs to.

Further XSLT extension functions are described in the external document “XMLTS Transformation Engine”.

Chapter 7. Appendices

Software Configuration

`pure::variants` may be configured from the configuration page (located in Window->Preferences->Variant Management Preferences). The available configuration options allow the license status to be checked, the plug-in logging options to be specified and the configuration of some aspects of the internal operations of the plug-in to be specified. Pure-systems support staff may ask you to configure the software with specific logging options in order to help identify any problems you may experience.

Figure 7.1. The configuration dialog of `pure::variants`
fig-configuration-dialog

User Interface Advanced Concepts

Console View

This view is used to alter the information that is logged during program operation. The amount of information to be logged is controlled via a preferences menu and this can be changed at any time by selecting the log level icon in the view's toolbar. The changed logging level is active only for the current session. [Note If the preferences menu is used instead to change the logging level then this applies to this session and every subsequent session.]

Glossary

Context Menu	A menu, which is customized according to the user interface item the user is currently pointing at (with the mouse). On Windows, Linux and MacOS X (with two or more mouse buttons), the right mouse button is usually configured to open the context menu. Under MacOS X (with single button mouse) the command key and then the mouse button have to be pressed (while still holding the command key) to open the context menu.
CSV	Comma Separated Value list. A simple text format often used to exchange spreadsheet data. Each line represents a table row, columns are separated with a comma character or other special characters (e.g. if the comma in the user's locale is used in floating point numbers like in Germany).
DOT	The name of a tool and its input format for automatic graph layouting. The tool is part of the GraphViz package available as open source from www.graphviz.org
EBNF	Extended Backus-Naur Form. A common way to describe programming language grammars. The Backus-Naur Form (BNF) is

	a convenient means for writing down the grammar of a context-free language. The Extended Backus-Naur Form (EBNF) adds the regular expression syntax of regular languages to the BNF notation, in order to allow very compact specifications. The ISO 14977 standard defines a common uniform precise EBNF syntax,
HTML	Hyper Text Markup Language.
Model Rank	Model rank is a positive integer that is used to control the model evaluation order. Models are evaluated from higher to lower ranks i.e. all models with rank 1 (highest) are evaluated before any model with rank 2 or lower. The rank of a model is configuration space-specific and can be set in the configuration space properties. The default rank is 1.
OCL	Object Constraint Language. A standardized declarative language for specifying constraints on UML models. See http://www.omg.org .
Prolog	PROgramming in LOGic. A programming language based on predicate logic.
XML	eXtensible Markup Language. A simple standardized language for representing structured information. See http://www.w3.org
XML Namespace	To provide support for independent development of XML markup elements (DTD/XML Schema) without name clashes, XML has a concept to provide several independent namespaces in a single XML document. See http://www.w3.org
XMLTS	XML Transformation System. The name for the pure::variants-specific transformation system for generating variants from XML based models.
XPath	XPath is part of the XML standard family and is used to describe locations in XML documents but also contains additional functions e.g. for string manipulation. XPath is heavily used in XSLT.
XSLT	XML Stylesheet Language Transformations. A standardized language for describing XML document transformation rules. See http://www.w3.org
UML	Unified Modeling Language. A standardized language for expressing software architectures and similar information. See http://www.omg.org
URL	Uniform Resource Locator. A standardized format for expressing the type and location of a resource (i.e. a file or service access point). Most commonly used for referring to HTML pages on an HTTP web server (e.g. http://my.server.org/index.html)

Index

A

- Attribute
 - Calculation
 - Syntax,
 - Element, 13
 - Feature, 15
 - Value, 14
 - Value Types, 14
- Attributes
 - Editor, 39
 - View, 50

C

- Compare
 - Models, 49

D

- Dialog
 - Element Selection, 42

E

- Editor
 - Attributes, 39
 - Common actions, 37
 - Common pages, 38
 - Family Model, 46
 - Feature Model, 43
 - Filter, 37
 - Restrictions, 41
 - Variant Description Model, 47
- Editor Pages
 - Table, 38
 - Tree, 38
- Element
 - Attribute, 13
 - Selection Dialog, 42
- Element Properties
 - Attributes Page, 39
 - Dialog, 39
 - General Page,
 - Restrictions Page, 41
- Evaluation
 - Variant Description Model, 20
- Export
 - Model, 58

F

- Family Model, 15
 - Editor, 46
- Family model
 - Restrictions, 18
- Feature

- Attributes, 15
 - Relations, 12
 - Restrictions, 15
- Feature Model, 14
 - Editor, 43
 - Feature Types, 70
- Features
 - Matrix View, 54
- Filter
 - View, 51

I

- Import
 - Model, 60

M

- Model
 - Compare, 49
 - Export, 58
 - Family, 15
 - Feature, 14
 - Import, 60
 - Meta Attributes, 70

O

- Outline
 - View, 51
- Outline View
 - Variant Description Model, 48

P

- Problems
 - View, 51
- Projects
 - View, 55
- Properties
 - View, 51
- pvProlog,

R

- Relations
 - Feature, 12
 - View, 52
- Restriction
 - Syntax,
- Restrictions
 - Editor, 41
 - family model, 18
 - Feature, 15
- Result
 - View, 53

T

- Tasks
 - View, 51
- Type

Attribute Value, 14

V

Value

Attribute, 14

Variant

Matrix View, 54

Variant Description Model

Editor, 47

Evaluation, 20

Outline, 48

Variant Projects

View, 55

Views

Attributes, 50

Feature Matrix, 54

Filter, 51

Outline, 51

Problems, 51

Properties, 51

Relations, 52

Result, 53

Tasks, 51

Variant Projects, 55