

pure::variants User's Guide

pure-systems GmbH

pure::variants User's Guide

Version 3.2.17 for pure::variants 3.2

Publication date 2015

Copyright © 2003-2015 pure-systems GmbH

Table of Contents

1. Introduction	1
1.1. What is pure::variants?	1
1.2. Link to PDF and Other Related Documents	1
2. Software and License Installation	3
2.1. Software Requirements	3
2.2. Software Installation	3
2.2.1. How to install the software	3
2.2.2. Updating the pure::variants	3
2.3. Obtaining and Installing a License	6
3. Introduction to Product Line Engineering with Feature Models	9
3.1. Introduction	9
3.2. Software Product Lines	9
3.3. Modelling the Problem Space with Feature Models	10
3.4. Modelling the Solution Space	12
3.5. Designing a variable architecture	13
3.6. Deriving product variants	14
4. Getting Started with pure::variants	15
4.1. Variant Management Perspective	15
4.2. Using Feature Models	15
4.3. Using Configuration Spaces	16
4.4. Transforming Configuration Results	17
4.5. Viewing and Exporting Configuration Results	18
4.6. Exploring Documentation and Examples	19
5. Concepts	21
5.1. Introduction	21
5.2. Common Concepts in pure::variants Models	21
5.2.1. Model Constraints	22
5.2.2. Element Restrictions	22
5.2.3. Element Relations	22
5.2.4. Element Attributes	22
5.3. Feature Models	25
5.3.1. Feature Attributes	26
5.4. Family Models	26
5.4.1. Structure of the Family Model	27
5.4.2. Sample Family Model	28
5.4.3. Restrictions in Family Models	28
5.4.4. Relations in Family Models	29
5.5. Variant Description Models	30
5.6. Hierarchical Variant Composition	30
5.7. Inheritance of Variant Descriptions	30
5.7.1. Inheritance Rules	31
5.8. Variant Description Evaluation	31
5.8.1. Evaluation Algorithm	31
5.9. Variant Transformation	33
5.9.1. The Transformation Process	33
5.9.2. Variant Result Models	34
6. Tasks	37
6.1. Evaluating Variant Descriptions	37
6.1.1. Configuring the Evaluation	37
6.1.2. Default Element Selection State	38
6.1.3. Automatic Selection Problem Resolving	39
6.1.4. Configuring the Auto Resolver	39
6.2. Reuse of Variant Descriptions	41
6.2.1. Hierarchical Variant Composition	41
6.2.2. Inheritance of Variant Descriptions	43

6.2.3. Load a Variant Description	44
6.3. Transforming Variants	44
6.3.1. Setting up a Transformation	44
6.3.2. Standard Transformation	48
6.3.3. User-defined transformation scripts with JavaScript	52
6.3.4. User-defined transformation scripts with XSLT	53
6.3.5. Transformation of Hierarchical Variants	57
6.3.6. Reusing existing Transformation	58
6.3.7. Ant Build Transformation Module	58
6.4. Validating Models	58
6.4.1. XML Schema Model Validation	59
6.4.2. Model Check Framework	59
6.5. Refactoring Models	62
6.6. Comparing Models	63
6.6.1. General Eclipse Compare	63
6.6.2. Model Compare Editor	63
6.6.3. Conflicts	64
6.6.4. Compare Example	64
6.7. Searching in Models	65
6.7.1. Variant Search	65
6.7.2. Quick Overview	68
6.8. Filtering Models	68
6.9. Computing Model Metrics	69
6.10. Extending the Type Model	70
6.11. Using Multiple Languages in Models	71
6.12. Importing and Exporting Models	72
6.12.1. Exporting Models	72
6.12.2. Importing Models	77
6.13. External Build Support (Ant Tasks)	79
6.13.1. pv.import	79
6.13.2. pv.evaluate	79
6.13.3. pv.transform	80
6.13.4. pv.inherit	80
6.13.5. pv.connect	80
6.13.6. pv.sync	80
6.13.7. pv.mergeselection	81
7. Graphical User Interface	83
7.1. Getting Started with Eclipse	83
7.2. Variant Management Perspective	84
7.3. Editors	84
7.3.1. Common Editor Pages	84
7.3.2. Feature Model Editor	97
7.3.3. Family Model Editor	100
7.3.4. Variant Description Model Editor	100
7.3.5. Variant Result Model Editor	102
7.3.6. Model Compare Editor	103
7.3.7. Matrix Editor	103
7.4. Views	104
7.4.1. Attributes View	104
7.4.2. Visualization View	105
7.4.3. Search View	106
7.4.4. Outline View	107
7.4.5. Problem View/Task View	107
7.4.6. Properties View	107
7.4.7. Relations View	108
7.4.8. Result View	109
7.4.9. Variant Projects View	111
7.5. Model Properties	112

7.5.1. Common Properties Page	112
7.5.2. General Properties Page	113
7.5.3. Inheritance Page	114
8. Additional pure::variants Plug-ins	115
8.1. Installation of Additional Plug-ins	115
9. Reference	117
9.1. Element Attribute Types	117
9.2. Element Relation Types	117
9.3. Element Variation Types	119
9.4. Element Selection Types	119
9.5. Predefined Source Element Types	119
9.5.1. aSourceElementType	120
9.5.2. ps:dir	120
9.5.3. ps:file	120
9.5.4. ps:fragment	121
9.5.5. ps:transform	121
9.5.6. ps:condxml	121
9.5.7. ps:condtext	123
9.5.8. ps:flagfile	124
9.5.9. ps:makefile	124
9.5.10. ps:classaliasfile	124
9.5.11. ps:symlink	125
9.6. Predefined Part Element Types	125
9.6.1. aPartElementType	126
9.6.2. ps:classalias	126
9.6.3. ps:class	126
9.6.4. ps:flag	126
9.6.5. ps:variable	127
9.6.6. ps:feature	127
9.7. Expression Language pvProlog	127
9.7.1. Element References	128
9.7.2. Logical Operators	129
9.7.3. Supported Functions	129
9.7.4. Additional Functions for Variant Evaluation	132
9.7.5. Match Expression Syntax for getMatchingElements	134
9.7.6. Accessing Model Attributes	134
9.7.7. Advanced pvProlog Examples	134
9.7.8. User-Defined Prolog Functions	136
9.8. Expression Language pvSCL	137
9.8.1. Comments	137
9.8.2. Boolean Values	137
9.8.3. Numbers	137
9.8.4. Arithmetics	137
9.8.5. Strings	138
9.8.6. Collections	138
9.8.7. Value Comparison	138
9.8.8. SELF and CONTEXT	139
9.8.9. Name and ID References	139
9.8.10. Element Existence Check	141
9.8.11. Attribute Access	141
9.8.12. Relations	142
9.8.13. Logical Combinations	142
9.8.14. Conditionals	143
9.8.15. Variable Declarations	143
9.8.16. Function Calls	143
9.8.17. Iterators	143
9.8.18. Accumulators	144
9.8.19. Function Definitions	144

9.8.20. Function Library	144
9.8.21. User-Defined pvSCL Functions	152
9.9. XSLT Extension Functions	152
9.10. Predefined Variables	157
9.11. Regular Expressions	157
9.11.1. Characters	158
9.11.2. Character Sequences	159
9.11.3. Repetition	160
9.11.4. Alternation	160
9.11.5. Grouping	160
9.11.6. Boundaries	160
9.11.7. Back References	160
9.12. Keyboard Shortcuts	160
10. Appendices	163
10.1. Software Configuration	163
10.2. User Interface Advanced Concepts	163
10.2.1. Console View	163
10.3. Glossary	163
Index	167

List of Figures

1.1. Overview of family-based software development with pure::variants	1
2.1. Update Site Selection	4
2.2. Pure::variants Plugin Selection	5
2.3. Licence Agreement	5
2.4. License Information Page	7
3.1. Overview of SPLE activities	10
3.2. Structure and notation of feature models (using pure::variants Directed Graph Export)	11
3.3. Feature Model for meteorological Product Line	11
3.4. Enhanced Feature Model for meteorological Product Line	12
3.5. pure::variants screen shot - solution space fragment shown at right	13
4.1. Initial layout of the Variant Management Perspective	15
4.2. A simple Feature Model of a car	16
4.3. VDM with a problematic selection	17
4.4. Transformation configuration in Configuration Space Properties	18
4.5. Transformation button in Eclipse toolbar	18
4.6. VDM export wizard	19
5.1. pure::variants transformation process	21
5.2. (simplified) element meta model	22
5.3. (Simplified) element attribute meta-model	23
5.4. Basic structure of Feature Models	26
5.5. Basic structure of Family Models	27
5.6. Sample Family Model	28
5.7. Model Evaluation Algorithm (Pseudo Code)	32
5.8. XML Transformer	33
6.1. VDM Editor with Outline, Result, Problems, and Attributes View	37
6.2. Model Evaluation Preferences Page	38
6.3. Automatically Resolved Feature Selections	39
6.4. Auto Resolver Preferences Page	40
6.5. Unique Names in a Variant Hierarchy	42
6.6. Example Variant Hierarchy	43
6.7. Load Selection Dialog	44
6.8. Multiple Transform Button	44
6.9. Configuration Space properties: Model Selection	45
6.10. Configuration Space properties: Transformation input/output paths	45
6.11. Configuration Space properties: Transformation Configuration	46
6.12. Transformation module selection dialog	47
6.13. Transformation module parameters	47
6.14. The Standard Transformation Type Model	49
6.15. Multiple attribute definitions for Value calculation	50
6.16. Sample Project using Regular Expressions	51
6.17. Variant project describing the manual	55
6.18. The manual for users and developers	57
6.19. Model Validation Preferences Page	59
6.20. New Check Configuration Dialog	60
6.21. Automatic Model Validation Preferences Page	61
6.22. Model Validation in Progress	62
6.23. Refactoring context menu for a feature	62
6.24. Model Compare Editor	65
6.25. The Variant Search Dialog	66
6.26. Quick Overview in a Feature Model	68
6.27. Filter definition dialog	69
6.28. Metrics for a model	70
6.29. Type Model Editor Example	71
6.30. Language selection in the element properties dialog	72
6.31. HTML Export Wizard	73

6.32. HTML Export Wizard	74
6.33. HTML Export Result	75
6.34. HTML Transformation Module	75
6.35. HTML Transformation Module Parameters	76
6.36. Directed graph export example	77
6.37. Directed graph export example (options LR direction, Colored)	77
6.38. JavaScript Manipulator Wizard Page	78
7.1. Eclipse workbench elements	83
7.2. Variant management perspective standard layout	84
7.3. Constraints view	86
7.4. Selected Element Selection Tool	88
7.5. Feature/Family Model Element Creation Tools	89
7.6. Family Model Element Properties	89
7.7. Element Relations Page	91
7.8. Sample attribute definitions for a feature	92
7.9. Restrictions page of element properties dialog	93
7.10. Constraints page of element properties dialog	94
7.11. Advanced pvSCL expression editor	94
7.12. Advanced pvProlog expression editor	95
7.13. pvProlog expression pilot	96
7.14. Element selection dialog	97
7.15. Feature Model Editor with outline and property view	98
7.16. New Feature wizard	99
7.17. Feature Model Element Properties	99
7.18. Open Family Model Editor with outline and property view	100
7.19. Specifying an attribute value in VDM with cell editor	101
7.20. Outline view showing the list of available elements in a VDM	102
7.21. VRM Editor with outline and properties view	103
7.22. Matrix Editor of a Configuration Space	104
7.23. Attributes view (right) showing the attributes for the VDM	105
7.24. Visualization view (left) showing 2 named filters and 2 named layouts	105
7.25. Variant Search View (Tree)	106
7.26. Variant Search View (Table)	106
7.27. Properties view for a feature	107
7.28. Description tab in Properties view for a relation	107
7.29. Properties view for a variant attribute	108
7.30. Relations view (different layouts) for feature with a ps:requires to feature 'Main Component Big'	109
7.31. Result View	110
7.32. Result View in Delta Mode	111
7.33. The Variant Projects View	112
7.34. Feature Model Properties Page	113
7.35. General Model Properties Page	113
7.36. Variant Description Model Inheritance Page	114
9.1. Prolog Code Library Model Property	136
9.2. pvSCL Code Library Model Property Page	152
10.1. The configuration dialog of pure::variants	163

List of Tables

5.1. Mapping between input and Variant Result Model types	34
6.1. Configuration Space Settings	58
6.2. Refactoring Operations	63
6.3. Table of CSS classes	74
6.4.	78
9.1. Supported Attribute Types	117
9.2. Supported relations between elements (I)	117
9.3. Supported Relations between Elements (II)	118
9.4. Element variation types and its icons	119
9.5. Types of element selections	119
9.6. Predefined source element types	119
9.7. Registered XSLT Extensions	122
9.8. Predefined part types	125
9.9. pvProlog Syntax (EBNF notation)	127
9.10. Element references	128
9.11. Examples	128
9.12. Logical operators in pvProlog	129
9.13. Logical functions in pvProlog	129
9.14. Functions for value calculations, restrictions, and constraints in pvProlog	130
9.15. Additional functions available for variant evaluation	132
9.16. Meta-Model attributes in pvProlog	134
9.17. Extension functions providing model information	153
9.18. Extension functions providing transformation information	154
9.19. Extension elements for logging and user messages	155
9.20. Extension functions providing file operations	156
9.21. Extension functions providing string operations	157
9.22. Supported Variables	157
9.23. Common Keyboard Shortcuts	161
9.24. Model Editor Keyboard Shortcuts	161
9.25. Graph Editor Keyboard Shortcuts	161

List of Examples

9.1. A sample conditional document for use with the ps:condxml transformation	122
9.2. Example use of pv:value-of	123
9.3. A sample conditional document for use with the ps:condtext transformation	124
9.4. Generated code for a ps:flagfile for flag "DEFAULT" with value "1"	124
9.5. Generated code for a ps:makefile for variable "CXX_OPTFLAGS" with value "-O6"	124
9.6. Generated code for a ps:classalias for alias "io::net::PConn" with aliased class "NoConn"	125

Chapter 1. Introduction

1.1. What is pure::variants?

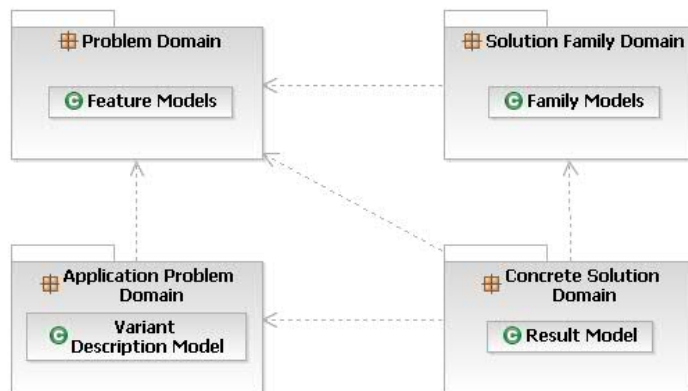
pure::variants provides a set of integrated tools to support each phase of the software product-line development process. pure::variants has also been designed as an open framework that integrates with other tools and types of data such as requirements management systems, object-oriented modeling tools,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

Figure 1.1, “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, is implemented as 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 (VDM), containing the selected feature set and associated values, represents a single problem from the problem domain. The Variant Result Model describes a single concrete solution drawn from the solution family.

Figure 1.1. 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 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 domain and to express the problems to be solved in terms of selected features and additional configuration information. This information is used to derive a Variant Result Model from the Family Model(s).
- The *application developer* generates a member of the solution family from the Variant Result Model by using the transformation engine.

1.2. Link to PDF and Other Related Documents

The *Workbench User Guide (Help->Help Contents)* is a good starting point for familiarizing yourself 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).

Any features concerning the pure::variants Server are described in the separate documents "pure::variants Server Support Plug-In Manual" and "pure::variants Server Administration Plug-In Manual". The server is available in the products "Professional" and "Enterprise".

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.

This document is available in online help as well as in printable PDF format [here](#).

Chapter 2. Software and License Installation

2.1. Software Requirements

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

Operating System:	<ul style="list-style-type: none">• Windows XP, Windows 7, Windows 2003 Server, Windows 2008 Server• Ubuntu Linux for x86• Intel MacOS X 10.5
Eclipse:	Eclipse 3.5 or higher required. Eclipse is available from http://www.eclipse.org/ .
Java:	Eclipse requires a Java Virtual Machine (JVM) to be installed. Minimum version is Java 5. We recommend using a Sun JDK 5 compatible JVM. See http://www.java.com/ for a suitable JVM.

2.2. Software Installation

2.2.1. How to install the software

pure::variants software is distributed and installed in several ways:

- *Stand-Alone Installation with an Installer [Windows only]* Installation via an Windows installer program. Download the appropriate Windows installer package from the internet. After unpacking the ZIP file start the Setup*.exe. The installer will create a stand-alone installation of pure::variants, which can be easily uninstalled without affecting any existing pure::variants or Eclipse installation. A Java Virtual Machine has to be installed separately before using pure::variants. Installation does not require administrative privileges, just writing privileges to the intended installation folder.

Installation into an existing Eclipse installation is not supported by the installer. The location of the download site depends on the pure::variants product variant. Visit the pure-systems web site (<http://www.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 install the software.

- *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. See task "Updating features with the update manager" resp. "Updating and installing software" in the Eclipse Workbench User Guide for detailed information on the Eclipse update mechanism (menu Help -> Help Contents and then Workbench User Guide->Tasks).

The location of the site depends on the pure::variants product variant. Visit the pure-systems web site (<http://www.pure-systems.com/pv>) 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 information how to use update sites with Eclipse 3.5.

- *Archived Update Site* pure::variants uses also the format of archived update sites, distributed as ZIP files, for offline installation into an existing Eclipse installation.

Archived update sites are available for download from the pure::variants internet update site. The location of the site depends on the pure::variants product variant. Visit the pure-systems web site (<http://www.pure-systems.com/pv>) 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 3.5. pure::variants archived update site file names start with *updatesite* followed by an identification of the contents of the update site. The installation process is similar to the internet update site installation.

2.2.2. Updating the pure::variants

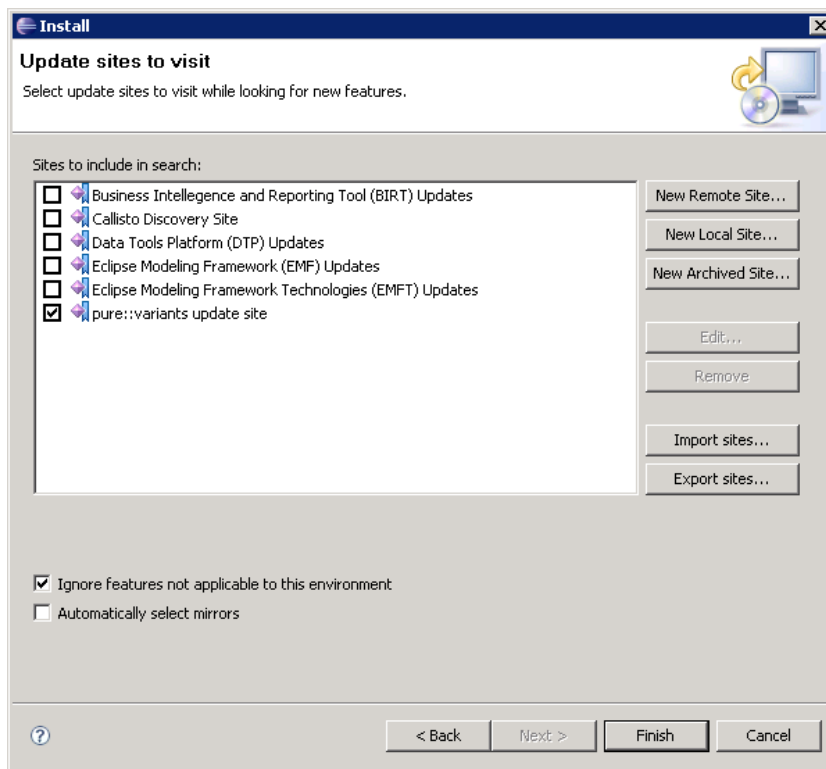
The quickest way to get a update for pure::variants is to run the software updater inside pure::variants:

- Start pure::variants (or the Eclipse in which pure::variants has been installed in).
- Select "Help"->"Software Updates"->"Find and Install...".
- Select "Search for new features to install" and "Next".
- Select "pure::variants update site" (all other check boxes should be deselected to speed up the process) and "Finish".

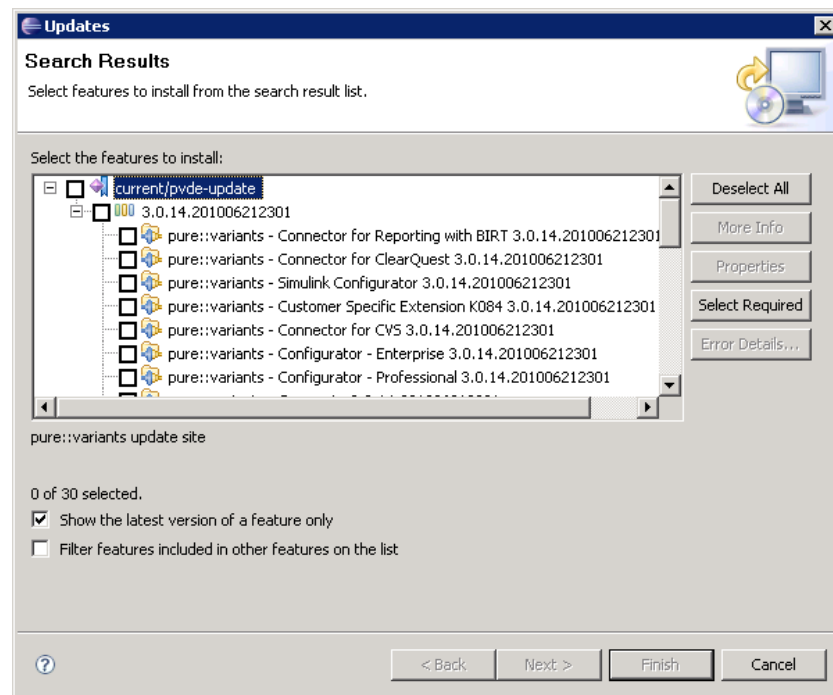
If location "pure::variants update site is not present, create "New Remote Site", enter a name and the url of the update site. Press "Ok", the new pure::variants update site from zip should be selected.

The location of the site depends on the pure::variants product variant. Visit the pure-systems web site (<http://www.pure-systems.com/pv>) or read your registration e-mail to find out which site is relevant for the version of the software your are using.

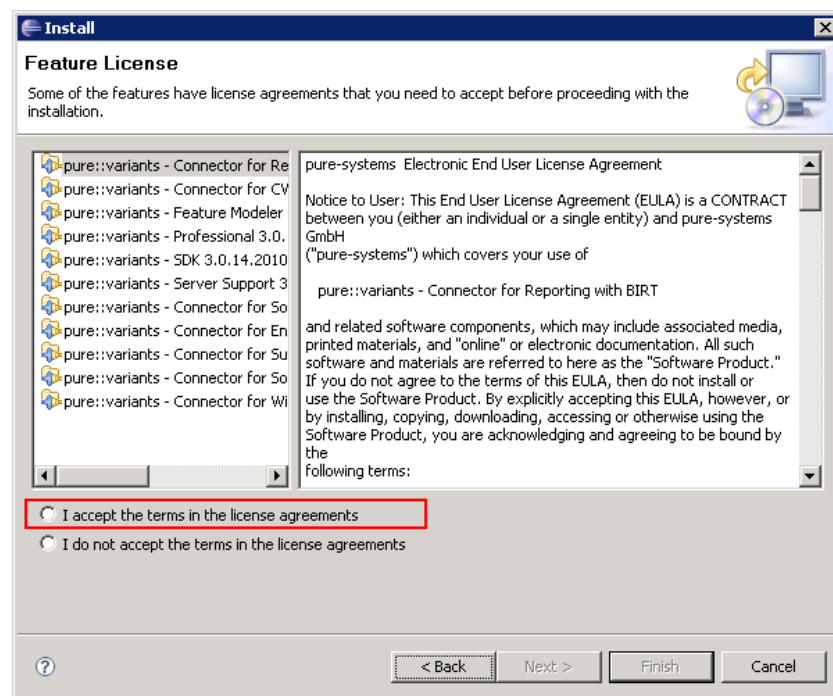
Figure 2.1. Update Site Selection



- Unfold all under pure::variants update site and select all features to be updated. If unsure, select all but the Connector for oAW (if visible). The Connector for oAW requires an openArchitectureWare installation which is by default not supplied. Select "Next".

Figure 2.2. Pure::variants Plugin Selection

- Accept license, hit "Next" and then "Finish".

Figure 2.3. Licence Agreement

- In the dialog select "Install all".
- Restart pure::variants when asked for.

If the direct remote update is not possible (often due to firewall/proxies preventing eclipse accessing external web sites), please go to the web site using an internet browser:

- For pure::variants Evaluation use www.pure-systems.com/pv-update
- For pure::variants Enterprise/Professional use www.pure-systems.com/pvde-update

and download the "Complete Updatesite" zip package:

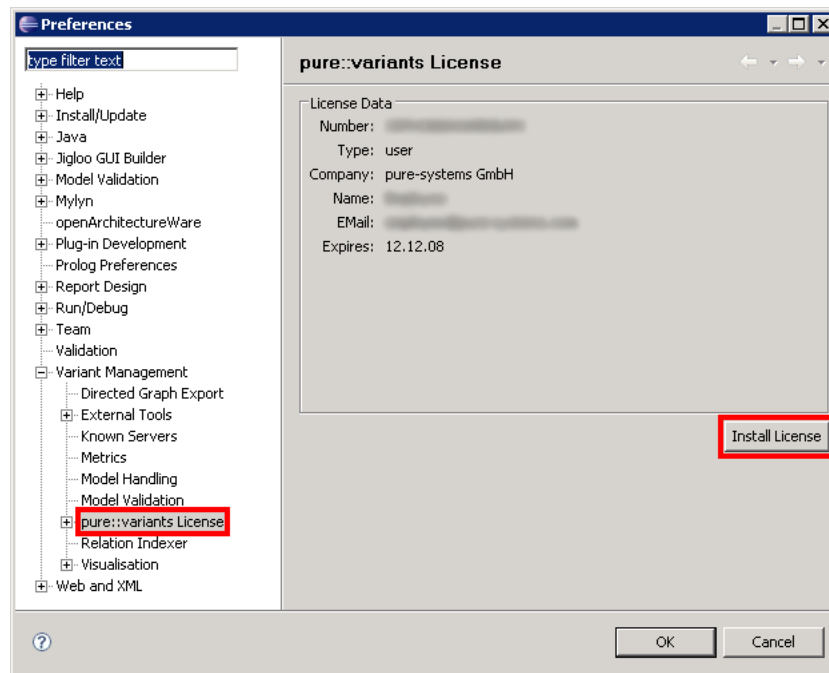
- Start pure::variants (or the Eclipse in which pure::variants has been installed in).
- Select "Help"->"Software Updates"->"Find and Install...".
- Select "Search for new features to install" and "Next".
- Click on button "Archived Update Site" or "Local Update Site".
- Use "Browse" to select the downloaded zip archive.
- Press "Ok", the new pure::variants update site from zip should be selected.
- All other check boxes should be deselected to speed up the process). Press "Finish".
- Unfold all under pure::variants update site and select all features to be updated. If unsure, select all but the Connector for oAW (if visible). The Connector for oAW requires an openArchitectureWare installation which is by default not supplied. Select "Next".
- Accept license, hit "Next" and then "Finish".
- In the dialog select "Install all".
- Restart pure::variants when asked for.

2.3. 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 in each of them.

To replace an expired license start pure::variants and open menu *Window->Preferences*. Navigate to page *Variant Management->pure::variants License* (see [Figure 2.4, "License Information Page"](#)). This page shows the currently installed license and allows to install new licenses. Click on button *Install License* and navigate to the license file. Select it and click on button *Open*. Now the page shows the new license information.

Figure 2.4. License Information Page



Chapter 3. Introduction to Product Line Engineering with Feature Models

3.1. Introduction

Although the term "(Software) Product line Engineering" is becoming more widely known, there is still uncertainty among developers about how it would apply in their own development context. The purpose of this chapter is to explain the design and automated derivation of the product variants of a Software Product Line using an easy to understand, practical example.

One increasing trend in software development is the need to develop multiple, similar software products instead of just a single individual product. There are several reasons for this. Products that are being developed for the international market must be adapted for different legal or cultural environments, as well as for different languages, and so must provide adapted user interfaces. Because of cost and time constraints it is not possible for software developers to develop a new product from scratch for each new customer, and so software reuse must be increased. These types of problems typically occur in portal or embedded applications, e.g. vehicle control applications. Software Product Line Engineering (SPLE) offers a solution to these not quite new, but increasingly challenging, problems. The basis of SPLE is the explicit modelling of what is common and what differs between product variants. Feature Models are frequently used for this. SPLE also includes the design and management of a variable software architecture and its constituent (software) components.

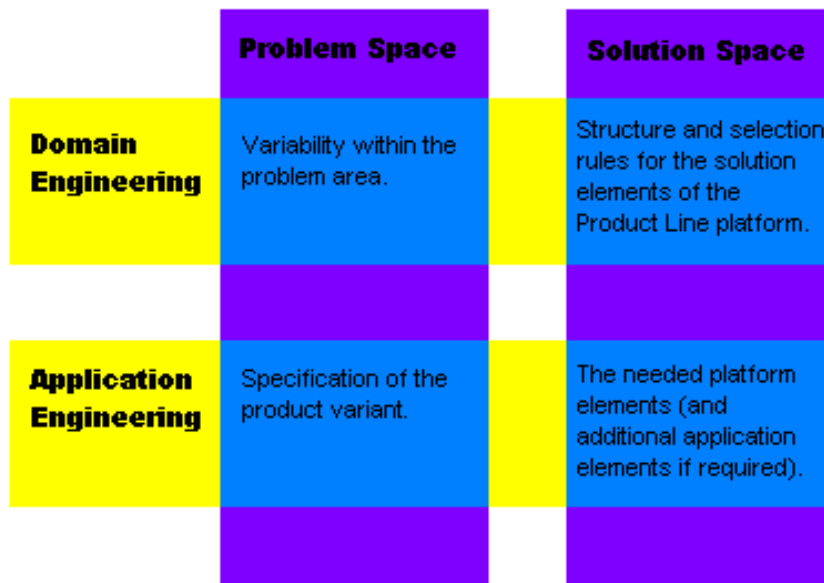
This chapter describes how this is done in practice, using the example of a Product Line of meteorological data systems. Using this example we will show how a Product Line is designed, and how product variants can be derived automatically using pure::variants.

3.2. Software Product Lines

However, before we introduce the example, we'll take a small detour into the basics of SPLE. The main difference from "normal", one-of-a-kind software development, is a logical separation between the development of core, reusable software assets (the platform), and actual applications. During application development, platform software is selected and configured to meet the specific needs of the application.

The Product Line's commonalities and variabilities are described in the Problem Space. This reflects the desired range of applications ("product variants") in the Product Line (the "domain") and their inter-dependencies. So, when producing a product variant, the application developer uses the problem space definition to describe the desired combination of problem variabilities to implement the product variant.

An associated Solution Space describes the constituent assets of the Product Line (often referred to as the "platform") and its relation to the problem space, i.e. rules for how elements of the platform are selected when certain values in the problem space are selected as part of a product variant. The four-part division resulting from the combination of the problem space and solution space with domain and application engineering is shown in [Figure 3.1, "Overview of SPLE activities"](#). Several different options are available for modelling the information in these four quadrants. The problem space can be described e.g. with Feature Models, or with a Domain Specific Language (DSL). There are also a number of different options for modelling the solution space, for example component libraries, DSL compilers, generative programs and also configuration files.

Figure 3.1. Overview of SPLE activities

In the rest of this chapter we will consider each of these quadrants in turn, beginning with Domain Engineering activities. We'll first look at modelling the problem space - what is common to, and what differs between, the different product variants. Then we'll consider one possible approach for realising product variants in the solution space using C++ as an example. Finally we'll look at how Application Engineering is performed by using the problem and solution space models to create a product variant. In reality, this linear flow is rarely found in practice. Product Lines usually evolve continuously, even after the first product variants have been defined and delivered to customers.

Our example Product Line will contain different products for entry and display of meteorological data on a PC. An initial brainstorming session has led to a set of possible differences (variation points) between possible products: meteorological data can come from different sensors attached to the PC, fetched from appropriate Internet services or generated directly by the product for demonstration and test purposes. Data can be output directly from the application, distributed as HTML or XML through an integrated Web server or regularly written to file on a fixed disk. The measurements to make can also vary: temperature, air pressure, wind velocity and humidity could all be of interest. Finally the units of measure could also vary (degrees Celsius vs. Fahrenheit, hPa vs. mmHg, m / s vs. Beaufort).

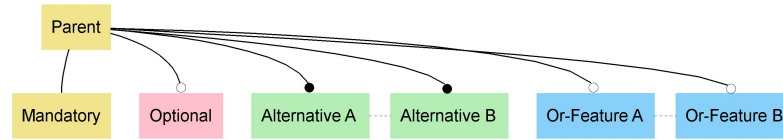
3.3. Modelling the Problem Space with Feature Models

We will now convert the informal, natural-language specification of variability noted above into a formal model, in order to be able to process it. Specifically, we will use a Feature Model. Feature models are simple, hierarchical models that capture the commonality and variability of a Product Line. Each relevant characteristic of the problem space becomes a feature in the model. Features are an abstract concept for describing commonalities and variabilities. What this means precisely needs to be decided for each Product Line. A feature in this sense is a characteristic of a system relevant for some Stakeholder. Depending on the interest of the Stakeholders a feature can be for the example a requirement, a technical function or function group or a non-functional (quality) characteristic.

Feature models have a tree structure, with features forming nodes of the tree. Feature variability is represented by the arcs and groupings of features. There are four different types of feature groups: "mandatory", "optional", "alternative" and "or".

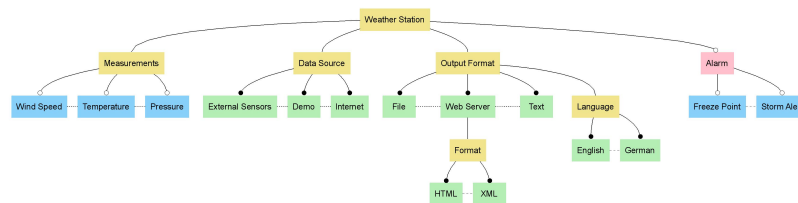
When specifying which features are to be included in a variant the following rules apply: If a parent feature is contained in a variant, all its mandatory child features must be also contained ("n from n"), any number of optional features can be included ("m from n, $0 < m \leq n$ "), exactly one feature must be selected from a group of alternative features ("1 from n"), at least one feature must be selected from a group of or features ("m from n, $m > 1$ ").

Figure 3.2. Structure and notation of feature models (using pure::variants Directed Graph Export)



There is no single standard for the graphical notation of feature models. We use a simplified notation created by pure::variants Direct Graph Export (see [the section called “Directed Graph Export”](#)). Alternatives and groups of or features are represented with traverses between the matching features. In this representation both colour and box connector are used independently to indicate the type of group. Our notation is shown in [Figure 3.2, “Structure and notation of feature models \(using pure::variants Directed Graph Export\)”](#). Using this notation, our example feature model, with some modifications, is shown in [Figure 3.3, “Feature Model for meteorological Product Line”](#): Each Feature Model has a root feature. Beneath this are three mandatory features – “Measurements”, “Data Source” and “Output Format”. Mandatory features will always be included in a product variant if their parent feature is included in the product variant. Mandatory features are not variable in the true sense, but serve to structure or document their parent feature in some way. Our example also has alternative features, e.g. “External Sensors”, “Demo” and “Internet” for data sources. All product variants must contain one and only one of these alternatives.

Figure 3.3. Feature Model for meteorological Product Line



At this stage we can already see one advantage that feature modelling has over a natural-language representation - it removes ambiguities - e.g. whether an individual variant is able to process data from more than one source. When taking measurements any combination of measurements is meaningful and at least one measurement source is necessary for a sensible weather station, to model this we use a group of Or. Usually simple optional features are used, such as the example of the freezing point alarm. Further improvements can also be made by refining the model hierarchy. So the strict choice between Web Server output formats - HTML or XML – can be made explicit.

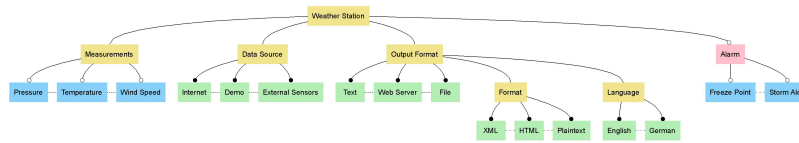
Feature models also support transverse relationships, such as requires (ps:requires) and mutually exclusive (ps:conflicts), in order to model additional dependencies between features other than those already described. So, in the example model, a selection of the “Freeze Point” alarm feature is only meaningful in connection with the temperature measurement capability. This can be modelled by an “Freeze Point” requires “Temperature” relationship (not shown in the figure). However, such relations should be used sparingly. The more transverse relations there are, the harder it is for a human user to visualize connections in the model.

When creating a feature model it can be difficult to decide exactly how problem space variabilities are to be represented in the model. In this case it is best to discuss this further with the customer. It is usually better to base these discussions around the feature model, since such models are easier for the customer to understand than textual documents and / or UML models. Formalising customer requirements in this way offers significant advantages later in Product Line development, since many architectural and implementation decisions can be made on the basis of the variabilities captured in the feature model.

In the example, the use of the output format XML and HTML can be clarified. The model explicitly defines that the choice of output format is only relevant for Web Server, a format selection is not possible for File or Text output. However, in the context of a discussion of the feature model it could be decided that HTML is also desirable for the on-screen (Window) representation and could also be applicable for file storage.

This results in the modified feature model shown in Figure 3.4, “Enhanced Feature Model for meteorological Product Line”.

Figure 3.4. Enhanced Feature Model for meteorological Product Line



We have added “Plaintext” to the existing features; this was implicitly assumed for output to the screen or to a file. We have modelled the mutual exclusion of XML and screen display (“Text”) using a (transverse) relationship between these features (not shown).

The previous discussion describes the basic feature model approach commonly found in the literature. However, pure::variants extends this basic approach. To complement the so-called hard relations between features (ps:requires and ps:conflicts) the weakened forms ps:recommends and ps:discourages have been added to many feature model dialects. pure::variants also supports the association of named attributes with features. This allows numeric values or enumerated values to be conveniently associated with features e.g. the wind force required to activate the storm alarm could be represented as a “Threshold” attribute of the feature “Storm Alert”.

An important and difficult issue in the creation of feature models is deciding which problem space features to represent. In the example model it is not possible to make a choice from the available hardware sensor types (e.g. use of a PR1003 or a PR2005 sensor for pressure). So, when specifying a variant, the user does not have direct influence on the selection of sensor types. These are determined when modelling the solution space. If the choice of different sensor types for measuring pressure is a major criterion for the customer / users, then appropriate options would have to be included in the feature model.

This means that the features in the problem space are not a 1:1-illustration of the possibilities in the solution space, but only represent the (variable) characteristics relevant for the users of the Product Line. Feature models are a user-oriented (or marketing-oriented) representation of the problem space, not the solution space.

After creating the problem space model we can use it to perform some initial analysis. For example, we can now calculate the upper limit on the number of possible variants in our example Product Line. In this case we have 1,512 variants (the model in Figure 2 only has 612 variants). For such a small number of variants the listing of all possible variants can be meaningful. However, the number of variants is usually too high to make practical use of such an enumeration.

3.4. Modelling the Solution Space

In order to implement the solution space using a suitable variable architecture, we must take account of other factors beyond the variability model of the problem space. These include common characteristics of all variants of the problem space that are not modelled in the feature model, as well as other constraints that limit the solution space.

These typically include the programming languages that can be used, the development environment and the application deployment environment(s). Different factors affect the choice of mechanisms to be used for converting from variation points in the solution space. These include the available development tools, the required performance and the available (computing) resources, as well as time and money. For example, use of configuration files can reduce development time for a project, if users can administer their own configurations. In other cases, using preprocessor directives (#ifdef) for conditional compilation can be appropriate, e.g. if smaller program sizes are required.

There are many possibilities for implementation of the solution space. Very simple variant-specific model transformations can be made with model-driven software development (MDSD) tools by including information from feature models in the Model-Transformation process, e.g. by using the pure::variants Connector for Ecore/openArchitectureWare or the pure::variants Connector for Enterprise Architect. Product Lines can also be implemented naturally using “classical” means such as procedural or object-oriented languages.

3.5. Designing a variable architecture

A Product Line architecture will only rarely result directly from the structure of the problem space model. The solution space which can be implemented should support the variability of the problem space, but there won't necessarily be a 1:1 correspondence of the feature models with the architecture. The mapping of variabilities can take place in various ways.

In the example Product Line we will use a simple object-oriented design concept implemented in C++. A majority of the variability is then resolved at compile-time or link-time; runtime variability is only used if it is absolutely necessary. Such solutions are frequently used in practice, particularly in embedded systems.

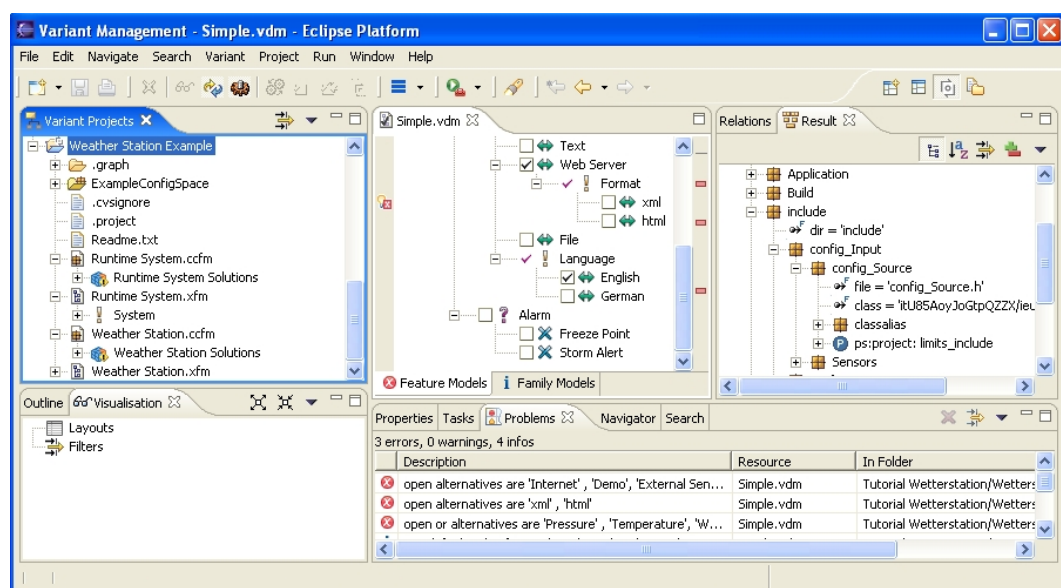
The choice of which tools to use for automating the configuration and / or production of a variant plays a substantial role in the design and implementation of the solution space. The range of variability, the complexity of relations between problem space features and solution constituents, the number and frequency of variant production, the size and experience of the development team and many further factors play a role. In simple cases the variant can be produced by hand, but quickly automation in the various forms like small configuration scripts, model transformers, code generators or variant management systems such as pure::variants will speed production.

For modelling and mapping of the solution space variability pure::variants and its integrated model transformation in most case is an ideal. This uses a Family Model to model the solution space, to associate solution space elements with problem space features, and to support the automatic selection of solution space elements when constructing a product variant.

Family models have a hierarchical structure, consisting of logical items of the solution architecture, e.g. components, classes and objects. These logical items can be augmented with information about "real" solution elements such as source code files, in order to enable automatic production of a solution from a valid feature model configuration (more on this later). For each family model element a rule is created to link it to the solution space. For example, the Web Server implementation component is only included if the Web Server feature has been selected from the problem space. To achieve this, a hasFeature("Web Server") rule is attached to the "Web Server" component. Any item below "Web Server" in the Family model can only be included in the solution if the corresponding Web Server feature is selected.

A pure::variants screen shot showing part of the solution space is shown in [Figure 3.5, "pure::variants screen shot - solution space fragment shown at right"](#).

Figure 3.5. pure::variants screen shot - solution space fragment shown at right



In our example, an architectural variation point arises, among other possibilities, in the area of data output. Each output format can be implemented with an object of a format-specific output class. Thus in the case of HTML

output, an object of type `HtmlOutput` is instantiated, and with XML output, an `XmlOutput` object. There would also be the possibility here of instantiating an appropriate object at runtime using a Strategy pattern. However, since the feature model designates only the use of alternative output formats, the variability can be resolved at compile-time and a suitable object can be instantiated using code generation for example.

In our example solution space a lookup in a text database is used to support multiple natural languages. The choice of which database to use is made at compile-time depending on the desired language. No difference in solution architectures can be detected between two variants that differ only in the target language. Here the variation point is embedded in the data level of the implementation. In many cases managing variable solutions only at the architectural level is insufficient. As has already been mentioned above, we must also support variation points at the implementation level, i.e. in our case at the C++ source code level. This is necessary to support automated product derivation. The constituents of a solution on the implementation level, like source code files or configuration files which can be generated, can also be entered in the family model and associated with selection rules.

So the existence of the Web Server component in a product variant is denoted using a `#define` preprocessor directive in a configuration Header file. In addition, an appropriate abstract variation point variable "WEB SERVER" must first be created of the type `ps:variable` in the family model. The value of this variable is determined by a `Value` attribute. In our case this value is always 1 if the variable is contained in the product variant. An item of type `ps:flagfile` can now be assigned to this abstract variable. This item also possesses attributes (file, flag), which are used during the transformation of the model into "real" code. The meaning of the attributes is determined by the transformation selected in the generation step. Here we use the standard `pure::variants` transformation for C / C++ programs, which produces a C-preprocessor `#define`-Flags in the file defined by file from these specifications.

Separating the logical variation point from the solution makes it very simple to manage changes to the solution space. For example, if the same variation point requires an entry in a Makefile, this could be achieved with the definition of a further source element, of the type `ps:makefile`, below the variation point "WEB SERVER".

3.6. Deriving product variants

The family model captures both the structure of the solution space with its variation points and the connection of solution and problem space. Not only is the separation of these two spaces important, but also the direction of the connection, since problem space models in most cases are much more stable than solution spaces; the linkage of the solution space to the problem space is more meaningful than the selection of solution items by rules in the problem space. This also increases the potential for reuse, since problem space models can simply be combined with other (new, better, faster) solutions. In `pure::variants` the linkage between models is determined by creating a configuration space with the relevant feature and family models as members.

Now we have all the information needed to create an individual product variant. The first step is to determine a valid selection of characteristics from the feature model. In the case of `pure::variants`, the user is guided towards a valid and complete feature selection. Once a valid selection is found, the specified feature list as well as the family model serve as input for the production of a variant model. Then, as is described above, the rules of the individual model items are checked. Only items that have their rules satisfied are included in the finished solution.

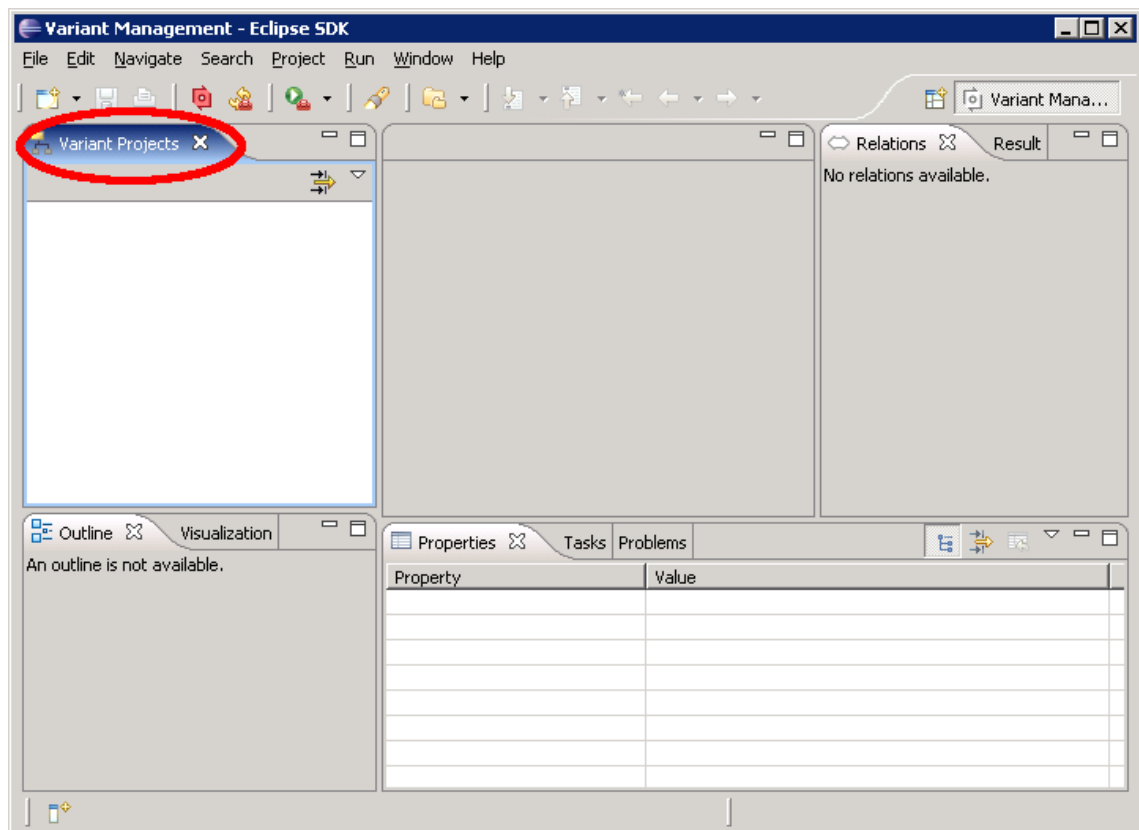
Since all these activities are done on `pure::variants` model level only, no "real" product has been created at this point. The last step is to execute the transformation, which interprets the models and creates an actual product variant. In `pure::variants` this transformation is highly configurable. In this example, source code would be copied from a file repository to a variant specific location, the configuration header file and some makefile settings would be generated. Also the generation of product variant specific UML models is a possible transformation. See following parts of the documentation for more information on the transformation process.

Chapter 4. Getting Started with pure::variants

4.1. Variant Management Perspective

The easiest way to access the variant management functionality is to use the Variant Management perspective provided by pure::variants. If not open by default, 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 4.1. Initial layout of the Variant Management Perspective



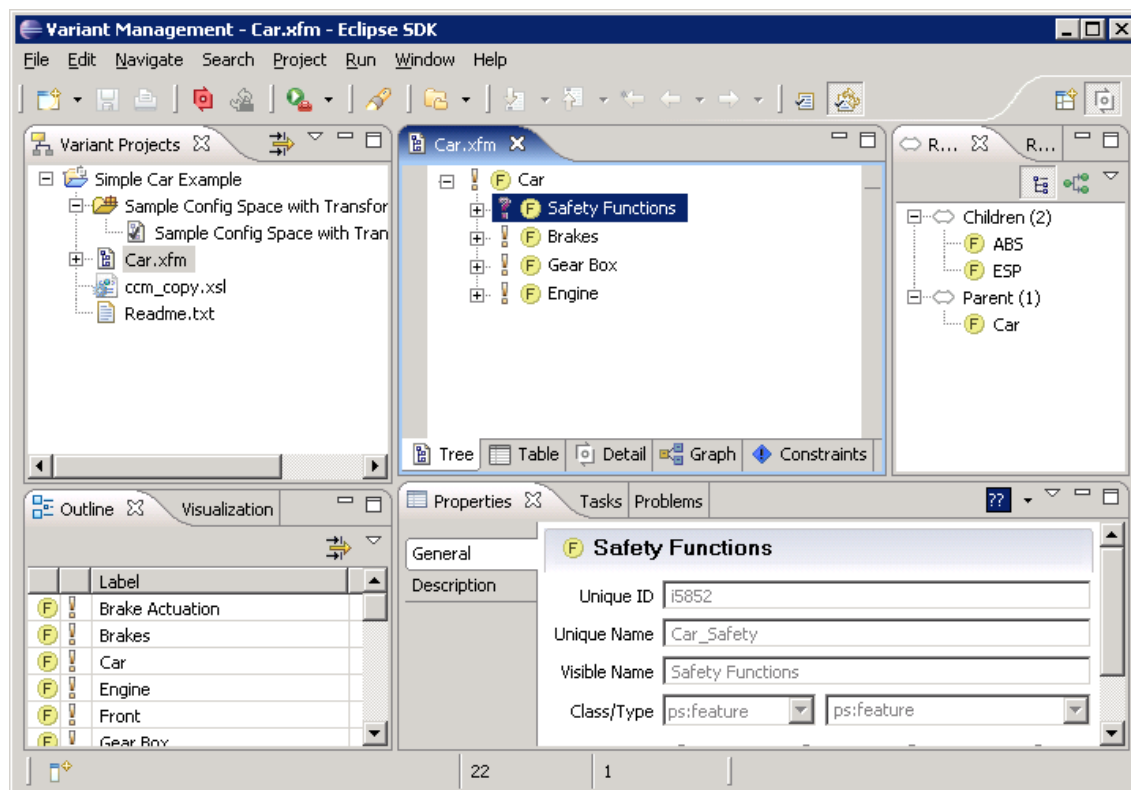
Now select the Variant Projects view in the upper left side of the Eclipse window. Create an initial standard project using the context menu of this view and choose New->Variant Project or use the File->New->Project wizard from the main menu. The view will now show a new project with the given name.

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 VDM.

4.2. Using Feature Models

When a new Variant project of project type *Standard* is created a new Feature Model is also created with a root feature of the same name as the project's name followed by *Features*. 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 4.2. 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 Table tab of the Feature Model Editor (shown in the lower left part) provides a table view of the model. It lists all features in a table, where editing capabilities are similar to the tree (same context menu, cell editors concept...). It allows free selection of columns and their order.

The Details tab of the Feature Model Editor provides a different view on the current feature. This view uses a layout and fields inspired by the *Volere* requirements specification template to record more detailed aspects of a feature.

The Graph tab provides a graphical representation of the Feature model. It also supports most of the actions available in the feature model Tree view.

The Constraints tab contains a table with all constraints defined in the model supporting full editing capabilities for the constraints.


4.3. Using Configuration Spaces

In order to create VDMs 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 initially created Standard project Configuration Space is already configured in this way.

A VDM has to be created inside the Configuration Space for each configuration. This is done using the context menu of the Configuration Space.

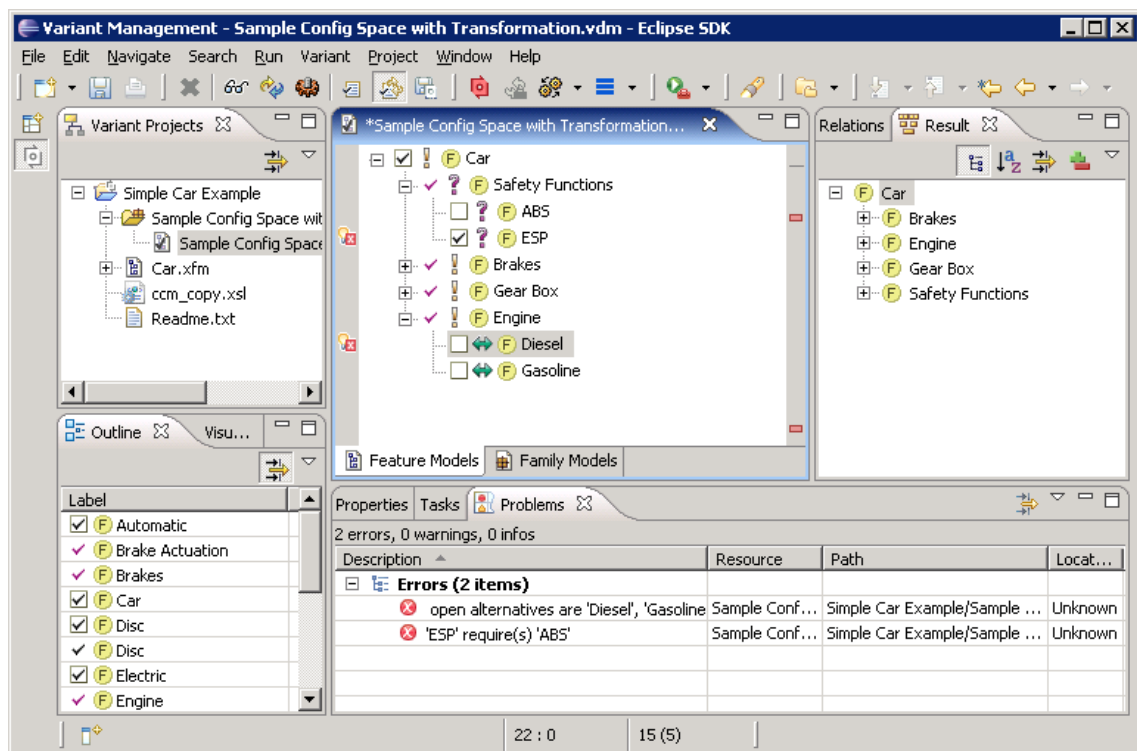
The VDM Editor is used to select the desired features for the variant. This editor is also used to perform configuration validation. The Evaluate Model button on the toolbar, and the *Variant->Evaluate* menu item, are used to

perform an immediate validation of the feature selection. The *Variant->Auto Evaluate* 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 problems view (lower right part) shows problems with the current configuration. Double clicking on a problem will open the related element(s) in the VDM 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 of the problem may offer options for solving the problem.

The figure below shows an example of a problem selection.

Figure 4.3. VDM with a problematic selection



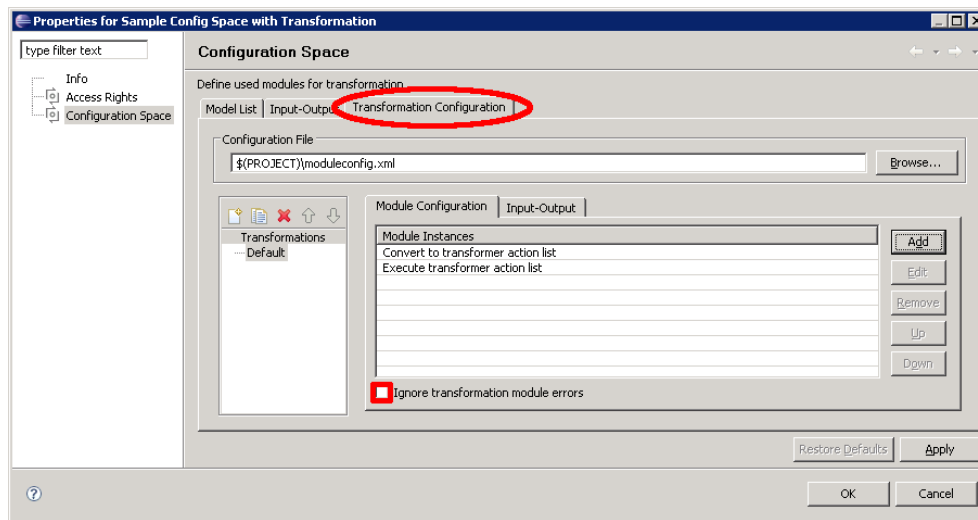
The Outline view shows a configurable list of features from all Feature Models in the Configuration Space.

4.4. Transforming Configuration Results

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

A modular, XML-based transformation engine is used to control this process (see [Section 5.9, "Variant Transformation"](#)). 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 dialog. The Transformation Configuration Page ([Figure 4.4, "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 [Section 6.3.1, "Setting up a Transformation"](#) for more information.

Figure 4.4. Transformation configuration in Configuration Space Properties

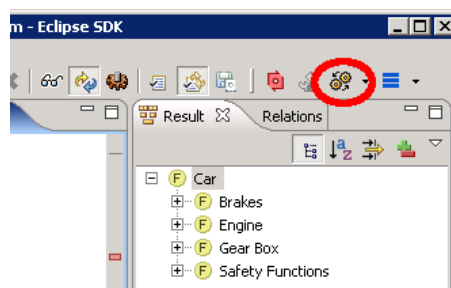
The toolbar transformation button is used to initiate a transformation (see [Figure 4.5, “Transformation button in Eclipse toolbar”](#)). If the current feature selection is invalid a dialog is opened asking the user whether to transform anyway.

Note

Transforming invalid configurations may yield incorrect product variants.

For more information on the XML transformation engine, see the document *pure::variants XML Transformation System Documentation*.

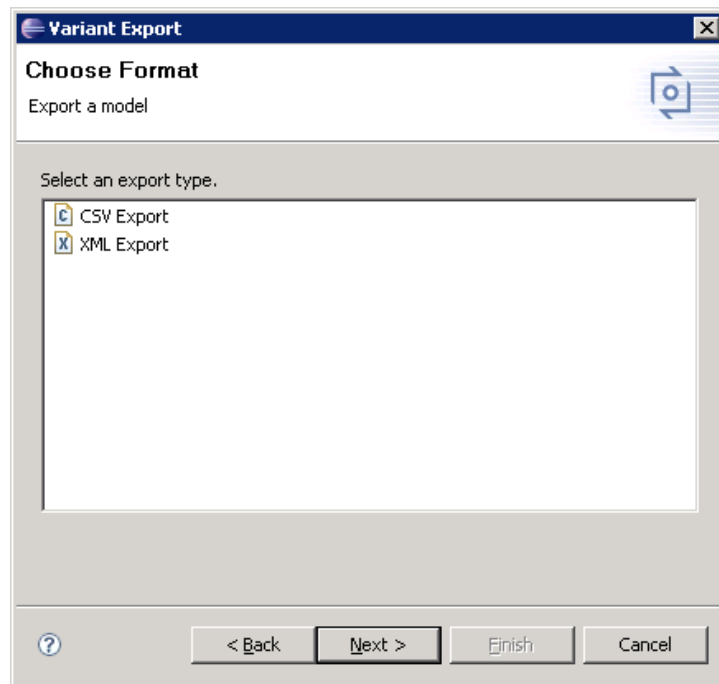
The distributed examples include some sample transformations.

Figure 4.5. Transformation button in Eclipse toolbar

4.5. 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 variant result models that have been derived from the corresponding models in the Configuration Space.

The context menu of the Variant Projects view provides an Export operation. As shown in the figure below, configuration results (features and components) can be exported as 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 4.6. VDM export wizard

4.6. Exploring Documentation and Examples

"pure::variants" gives an 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.

Additionally tutorials can be installed in the same way as the examples. The available tutorials are listed in the dialog below the items "Variant Management Tutorials". It contains the documentation itself in the pure::variants project and optional project contents.

Chapter 5. Concepts

5.1. Introduction

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 Feature Models describing the problem domain, Family Models describing the problem solution and Variant Description Models (VDMs) 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 [Section 5.3, “Feature Models”](#).

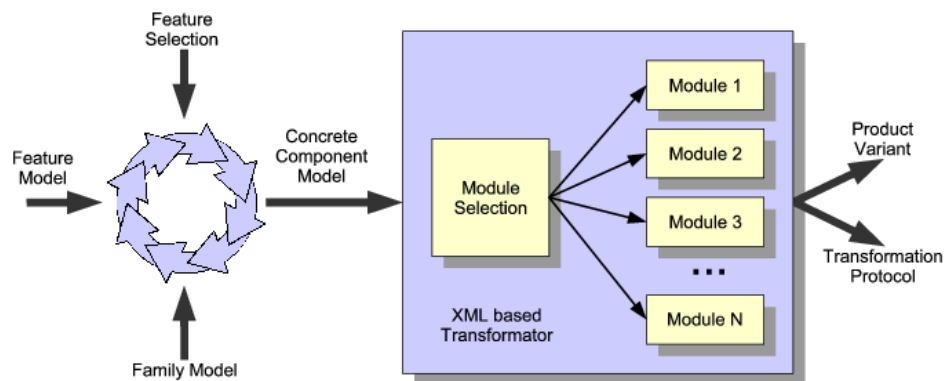
Family Models describe how the products in the product line will be assembled or generated from pre-specified components. Each component in a Family Model represents one or more functional elements of the products in the product line, for example software (in the form of classes, objects, functions or variables) or documentation. Family models are described in more detail in [Section 5.4, “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. VDMs are described in more detail in [Section 5.5, “Variant Description Models”](#).

In contrast to other approaches, pure::variants captures the Feature Model (problem domain) and the 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 5.1, “pure::variants transformation process”](#) gives an overview of the basic process of creating variants with pure::variants.

Figure 5.1. pure::variants transformation process



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

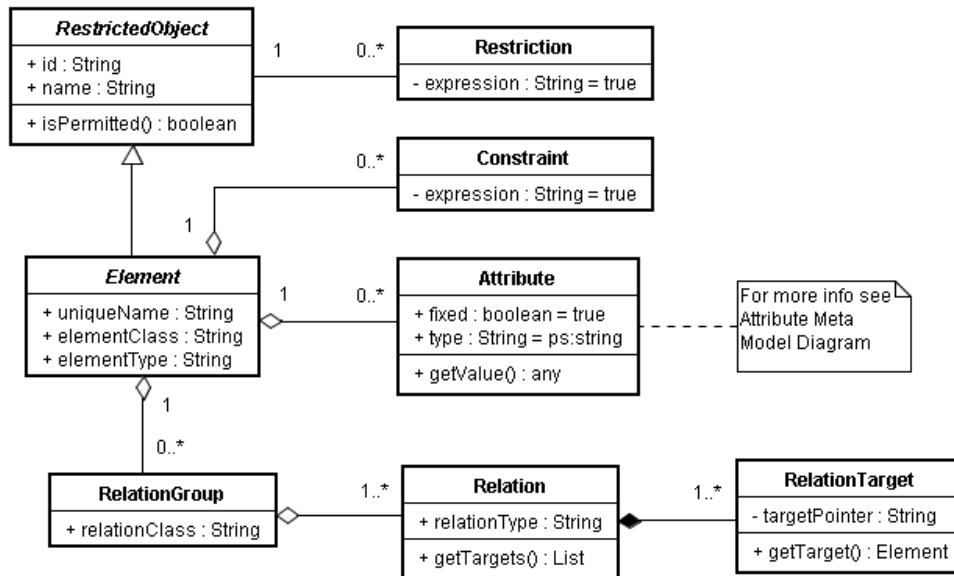
5.2. Common Concepts in pure::variants Models

This section describes the common, generic structure on which all models are based.

All models store elements (features in Feature Models, components, parts and source elements in Family Models) in a hierarchical tree structure. Elements ([Figure 5.2, “\(simplified\) 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 restrictions and constraints can be assigned to an element.

Figure 5.2. (simplified) element meta model



5.2.1. Model Constraints

Model constraints are used to check the integrity of the configuration (Variant Result Model) during a model evaluation. They can be assigned to model elements for clarity only, i.e. they have no effect on the assigned elements. All defined constraints have to be fulfilled for a resulting configuration to be valid. Detailed information about using constraints is given in [Section 5.8, “Variant Description Evaluation”](#).

5.2.2. Element Restrictions

Element restrictions are used to decide if an element is part of the resulting configuration. During model evaluation, an element cannot become part of a resulting configuration unless one of the restrictions defined on the element evaluates to true. Restrictions can not only be defined for elements but also for element attributes, attribute values, and relations. Detailed information about using restrictions is given in [Section 5.8, “Variant Description Evaluation”](#).

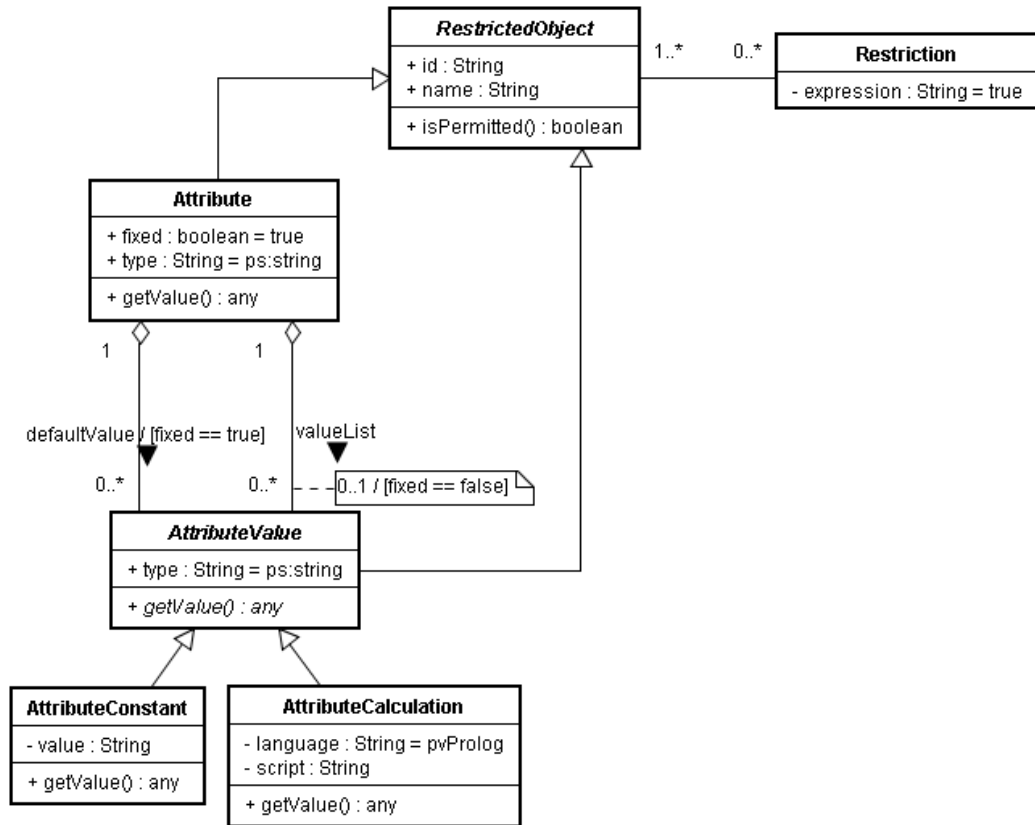
5.2.3. Element Relations

pure::variants allows arbitrary 1:n relations between model elements to be expressed. The graphical user interface provides access to the most commonly 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 9.2, “Supported relations between elements \(I\)”](#) documents the supported relations and their meanings.

5.2.4. 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 5.3. (Simplified) element attribute meta-model

Element attributes may be *fixed* (indicated with the checked **F** column in the GUI) 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 VDM, so the value may be different in other VDMs.

A non-fixed attribute must not, but can have values that are used by default when the element is selected and no value has been specified in the VDM.

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*.

Instead of selecting one value from a list of possible values, it is also possible to provide attributes which have a configurable collection of values. Each value in the collection is available in a variant if the corresponding restriction holds true. Two types of collections are available for use: Lists and Sets. List attributes mean to maintain an order of the values and allow multiple equal entries. Set attributes instead require each value to be unique. An order is not ensured. To use this feature, either square brackets ("[]") for lists or curly brackets ("{}") for sets have to be added after the data type, e.g. *ps:string{}*, *ps:boolean[]*, or *ps:integer[]*.

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 [Table 9.1, “Supported Attribute Types”](#). Users may define their own types by entering the desired type name instead of choosing one of the predefined types.

By adding square brackets (`[]`) or curly brackets (`{}`) to the name of a value type a list or set type can be specified, e.g. `ps:string[]`, `ps:boolean[]`, or `ps:integer{}`. A list or set type can hold a list of values of the same data type. In contrast to normal types each of the given values is available in a variant if its restriction holds true or it doesn't have a restriction.

Attribute Values

Attribute values can be constant or calculated. Calculations are performed by providing a calculation expression instead of the constant value. The result of evaluating the calculation expression is the value of the attribute in a variant. `pure::variants` uses either the built-in expression language *pvSCL* or *pvProlog* to express calculations.

Attributes with type `ps:integer` must have decimal or hexadecimal values of the following format.

```
('0x' [0-9a-fA-F]+) | ([+-]? [0-9]+)
```

Attributes with type `ps:float` must have values of the following format.

```
[+-]? (([0-9]+ ('.' [0-9]*)?) | ('.' [0-9]+)) ([eE] [+-]? [0-9]*)?
```

Attribute Value Calculations with pvSCL

When using *pvSCL* for value calculation, the following examples are a good starting point. For a detailed description of the *pvSCL* syntax, refer to [Section 9.8, “Expression Language pvSCL”](#).

Attribute calculation in *pvSCL* requires the returned value to be of the defined attribute type. Thus, to assign the value 1 to an attribute of type `ps:integer` use the following calculation expression:

```
1
```

To assign an attribute the value of another attribute `OtherAttribute` of an element `OtherElement`, use the following expression:

```
OtherElement->OtherAttribute
```

To return the half of the product of the value of two attributes, use:

```
(OtherElement->OtherAttribute*AnotherElement->AnotherAttribute)/2
```

Only the value of attributes of type `ps:float` and `ps:integer` should be used in arithmetic expressions.

Use the following expression to return a string based on another attribute.

```
'Text ' + OtherElement->OtherAttribute + ' more Text'
```

Attribute Value Calculations with pvProlog

When using *pvProlog* for value calculation, basic knowledge of the Prolog syntax and semantics are helpful. See [Section 9.7, “Expression Language pvProlog”](#) for a detailed description of the language. However, for many use cases the following examples are a good starting point.

Attribute calculation in *pvProlog* requires the value to be bound to a variable called `value`. Thus, to assign the value 1 to an attribute use the following calculation expression:

```
Value = 1
```

To assign an attribute the value of another attribute `OtherAttribute` of an element `OtherElement`, use the following expression:

```
getAttribute('OtherElement', 'OtherAttribute', OtherAttributeValue),
Value = OtherAttributeValue
```

`getAttribute` assigns the value to `OtherAttributeValue`, which is then assigned to the result variable `Value`. This expression can be written more compact as follows.

```
getAttribute('OtherElement', 'OtherAttribute', Value)
```

The result of an arithmetic expressions is assigned to the result variable using the keyword "is" instead of operator "=". To return the half of the product of the value of two attributes, use the following expression:

```
getAttribute('OtherElement', 'OtherAttribute', OAV),
getAttribute('AnotherElement', 'AnotherAttribute', AAV),
Value is (OAV*AAV)/2
```

On the right side of keyword "is", arithmetic expressions can be used similar to most other programming languages. Only the value of attributes of type *ps:float* and *ps:integer* should be used in arithmetic expressions.

Tip

Attribute values of type *ps:boolean* are represented as string constants 'true' and 'false'. They can not be used in direct comparisons with *pvProlog* `false` and `true`. Please make a string comparison instead, e.g. `BooleanAttrValue = 'true'`.

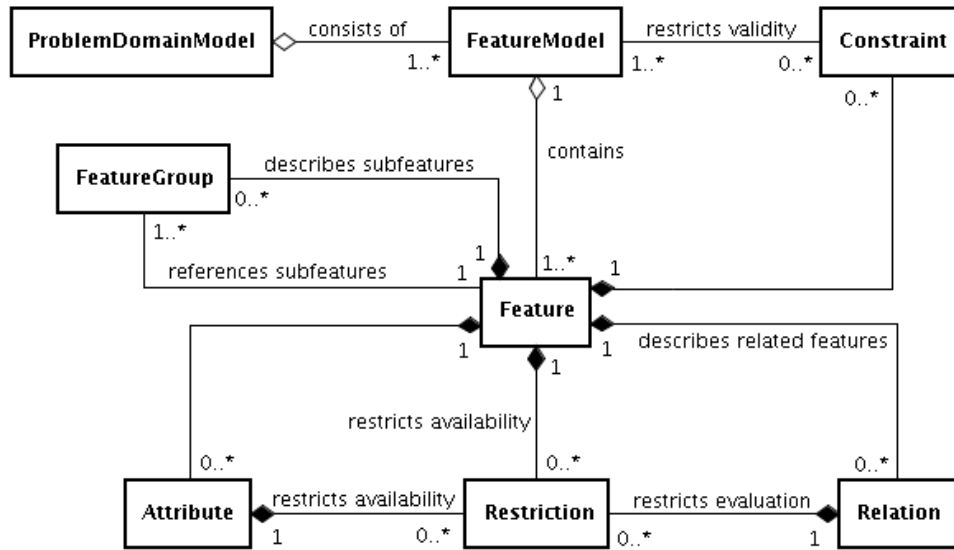
5.3. Feature Models

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 9.4, “Element variation types and its icons”](#) gives an explanation on these terms and shows how they are represented in feature diagrams.

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

[Figure 5.4, “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 5.4. Basic structure of Feature Models

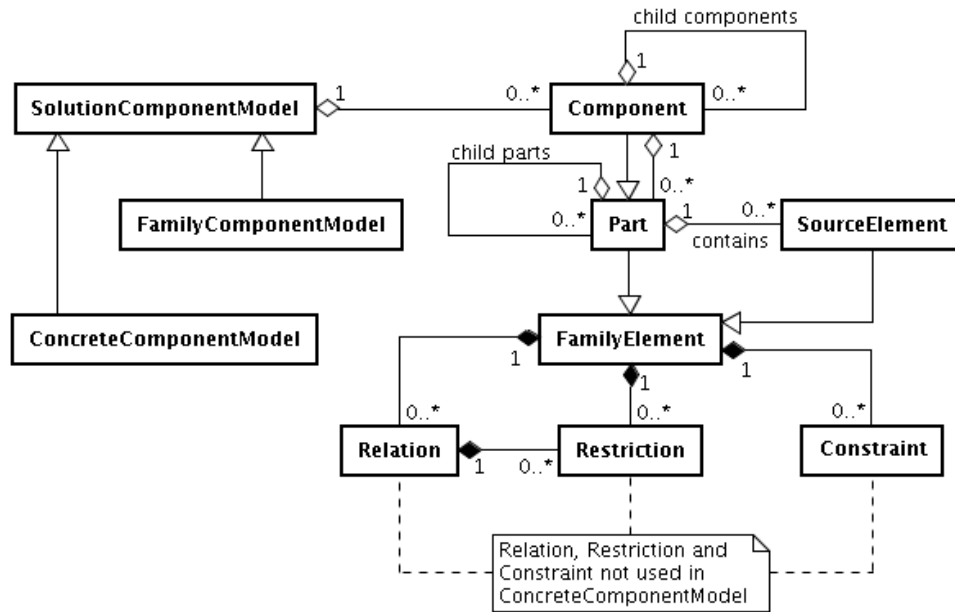
5.3.1. Feature Attributes

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, *pvProlog* or *pvSCL* level restrictions will be required.

5.4. Family Models

The Family Model describes the solution family in terms of software architectural elements. [Figure 5.5, “Basic structure of Family Models”](#) shows the basic structure of Family 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 Concrete Component Models are derived from Family Models and represent configured variants with resolved variabilities there are no restrictions used in Concrete Component Models. Please note, that older designations of Family Models are Family Component Model or even just Component Model. Following just Family Model will be used to designate those models with restrictions and thus unresolved variability.

Figure 5.5. Basic structure of Family Models

5.4.1. 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.

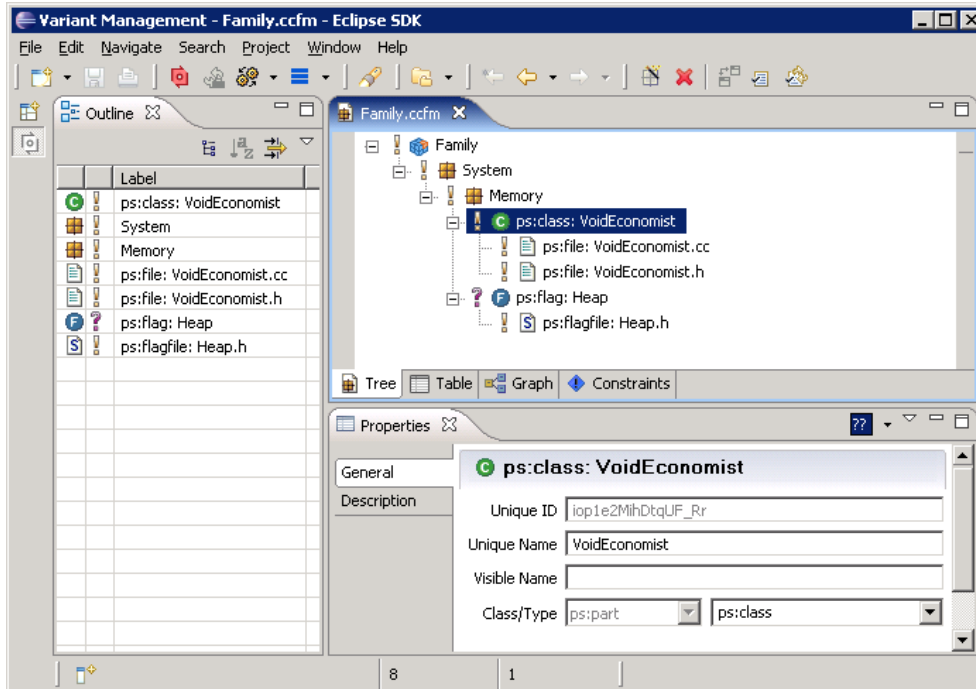
The actual interpretation of source elements is the responsibility of the `pure::variants` transformation engine. To allow the introduction of custom source elements and generator rules, `pure::variants` is able to host plug-ins for different transformation modules that interpret the generated Variant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

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

5.4.2. Sample Family Model

An example Family Model is shown below:

Figure 5.6. Sample 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 of 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 5.6, “Sample 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 Section 5.4.3, “Restrictions in Family Models” for more information.

5.4.3. Restrictions in 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. Restrictions 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 `hasFeature(<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 `BarFoos` 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')
```

5.4.4. Relations in Family Models

As for features, each element (component, part, and source element) may have relations to other elements. The supported relations are described in [Section 9.2, “Element Relation Types”](#).

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 *ps:requestsProvider* relation to require that a cosine implementation must be present for a configuration to be valid. *ps:exclusiveProvider* 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 it cannot be both implementations 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 Auto Resolver for selection problems 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'
```

5.5. Variant Description Models

Variant Description Models (VDM) describe the set of features of a single product in the product line. How to make a feature selection is described in [Section 7.3.4, “Variant Description Model Editor”](#). The validity of a feature selection is determined by the pure::variants model validation described in [Section 5.8, “Variant Description Evaluation”](#).

5.6. Hierarchical Variant Composition

See [Section 6.2.1, “Hierarchical Variant Composition”](#) for detailed information on how to create hierarchical variants.

5.7. Inheritance of Variant Descriptions

To share common feature selections/exclusions between several variants pure::variants supports VDM inheritance. This allows users to define the models for each VDM from which selections are to be inherited. Changes in the inherited model selection will be propagated automatically to all inheriting models. Inheritance is possible across Configuration Spaces and projects.

This kind of inheritance allows for example combination of partial configurations, restricting choices available to users only to the points where the inherited model left decisions explicitly open, or use of variant configurations in other contexts.

The list of models from which to inherit selections is defined on the properties page of the VDM (see [Section 7.5.3, “Inheritance Page”](#)). Models from the following locations can be inherited:

- from the same Configuration Space
- from another Configuration Space or folder of the same project
- from another Configuration Space or folder of a referenced project

Both single and multiple inheritance is supported. Single inheritance means that a VDM inherits directly from exactly one VDM. Multiple inheritance means directly inheriting from more than one VDM. It is not supported to directly or indirectly inherit a VDM from itself. But it is allowed to indirectly inherit a VDM more than once (diamond inheritance).

The following selections are inherited from a base VDM:

- selections explicitly made by the user
- exclusions explicitly made by the use
- selections the base VDM has inherited from other VDMs

Additionally attribute values defined in a inherited VDM are inherited if the corresponding selection is inherited. The applicable rules for the inheritance are listed in [Section 5.7.1, “Inheritance Rules”](#).

Inherited selections can not be changed directly. To change an inherited selection, the original selection in the inherited VDM has to be changed. Particularly if a selection is inherited that has a non-fixed attribute and no value is given in the inherited VDM, it is not possible to set a value for this attribute in the inheriting VDM. The value can only be set in the inherited VDM.

If both the inherited and the inheriting VDM are open, changes on the inherited VDM are immediately propagated to the inheriting VDM. This propagation follows the rules described in [Section 5.7.1, “Inheritance Rules”](#).

If the list of inherited VDMs for a VDM is changed, all inheriting VDMs have to be closed before.

5.7.1. Inheritance Rules

The following rules apply to the VDM inheritance:

1. If a model element is user selected in one inherited VDM it must not be user excluded in another. Otherwise it is an error and the conflicting selection is ignored.
2. There must be no conflicting values for the same attribute in different VDMs of the inheritance hierarchy, unless the corresponding selection is not inherited. Otherwise it is an error and the conflicting attribute value is ignored.
3. An inherited VDM has to exist in the current or in any of the referenced projects. Otherwise it is an error and the not existing VDM is ignored.
4. A VDM must not inherit itself, neither direct nor indirect. Otherwise it is an error.

5.8. Variant Description Evaluation

In the context of pure::variants, Model Evaluation is the activity of verifying that a VDM complies with the Feature and Family Models it is related to. Understanding this evaluation process is the key to a successful use of restrictions and relations.

5.8.1. Evaluation Algorithm

An outline of the evaluation algorithm is given in pseudo code below [Figure 5.7, “Model Evaluation Algorithm \(Pseudo Code\)”](#).

Figure 5.7. Model Evaluation Algorithm (Pseudo Code)

```

modelEvaluation()
{
  foreach(current in modelRanks())
  {
    checkAndStoreFeatSelection(
      getFeatModelsByRank(current));
    selectAndStoreFromFamModels(
      getFamModelsByRank(current), class('ps:component'));
    selectAndStoreFromFamilyModels(
      getFamModelsByRank(current), class('ps:part'));
    selectAndStoreFromFamilyModels(
      getFamModelsByRank(current), class('ps:source'));
  }
  calculateAttributeValuesForResult();
  checkFeatureRestrictions(getSelectedFeatures());
  checkRelations();
  checkConstraints();
}

```

```

modelEvaluation()
{
  foreach(current in modelRanks())
  {
    checkAndStoreFeatSelection(
      getFeatModelsByRank(current));
  }
  calculateAttributeValuesForResult();
  checkFeatureRestrictions(getSelectedFeatures());
  checkRelations();
  checkConstraints();
}

```

The algorithm has certain implications on the availability of information in restrictions, constraints, 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 four steps in the model evaluation process are performed only once. First, the attribute values for all selected elements are calculated. Then the restrictions and after that the relations of the selected features are checked. At this point all information about selected features and Family Model elements is available. Finally, the model constraints are evaluated deciding if the current selection is valid or not.

5.9. Variant Transformation

`pure::variants` supports a user-specified generation of product variants using an XML-based transformation component. Input to this transformation process is an XML representation of the Variant Result Model. Transformation modules are bound to 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 `pure::variants`, e.g. a module to execute XSLT scripts and a module for collecting and executing transformation actions. The list of available transformation depends on the `pure::variants` product and installed extensions.

The user may create custom modules and integrate these using the `pure::variants` API.

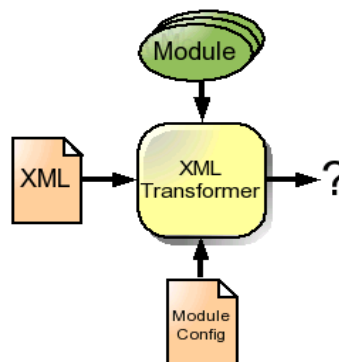
The transformation module configuration is part of the Configuration Space properties (see [Section 6.3.1, “Setting up a Transformation”](#)).

5.9.1. The Transformation Process

The transformation process works by traversing 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 5.8. 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.

Part of the SDK is a separately distributed 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.

5.9.2. Variant Result Models

For each Feature and Family Model of the Configuration Space a concrete variant is calculated during the model evaluation, called Variant Result Model. Restrictions and constraints are evaluated and removed from Variant Result Models. Attribute value calculations are replaced by their calculated values. Corresponding to these modifications the type of the models is changed to signal that a model is a concrete variant (see [Table 5.1, “Mapping between input and Variant Result Model types”](#)).

Table 5.1. Mapping between input and Variant Result Model types

Input Model Type	Result Model Type
ps:fm (<i>Feature Model</i>)	ps:cfm (<i>Concrete Feature Model</i>)
ps:ccfm (<i>Family Model</i>)	ps:ccm (<i>Concrete Family Model</i>)
ps:vdm (<i>Variant Description Model</i>)	ps:vdm (<i>Variant Description Model, identical to the input model</i>)

The Variant Result Models and additional variant information, collected in the so-called Variant Result Model, are the input of the pure::variants transformation. The Variant Result Model has the following structure.

```
<variant>
  <cil>
    <element idref="element id"/>
    <novalue idref="property id"/>
    <value idref="property id" vid="property value id"
      eid="element id">
      ...
    </value>
    ...
  </cil>
  <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>
```

```
<variant>
  <cil>
    <element idref="element id"/>
    <novalue idref="property id"/>
    <value idref="property id" vid="property value id"
      eid="element id">
      ...
    </value>
    ...
  </cil>
  <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:consulmodels>
</variant>
```

The `cil` subtree of this XML structure lists the concrete elements and property values of all concrete models in the variant. The `il` subtree contains a list of all inherited element attributes in all models of the variant. Finally the VDM and all Variant Result Models are part of the `cm:consulmodels` subtree.

This XML structure is used as input for the transformation engine as described above. `pure::variants` provides a certain set of XSLT extension functions (see [Table 9.17, “Extension functions providing model information”](#)) to simplify the navigation and evaluation of this XML structure in an XSLT transformation.

Tip

A copy of this XML structure can be saved using the "Save Result to File" button that is shown in the tool bar of a variant description model. In an XSLT transformation, access to the unmodified input models of the transformation can be gained using the `pure::variants` XSLT extension function `models()` (see [Table 9.17, “Extension functions providing model information”](#)).

Chapter 6. Tasks

6.1. Evaluating Variant Descriptions

In pure::variants a variant description, i.e. the selection of features in a VDM, can be evaluated and verified using the Model Evaluation. See [Section 5.8, “Variant Description Evaluation”](#) for a detailed description of the evaluation process.


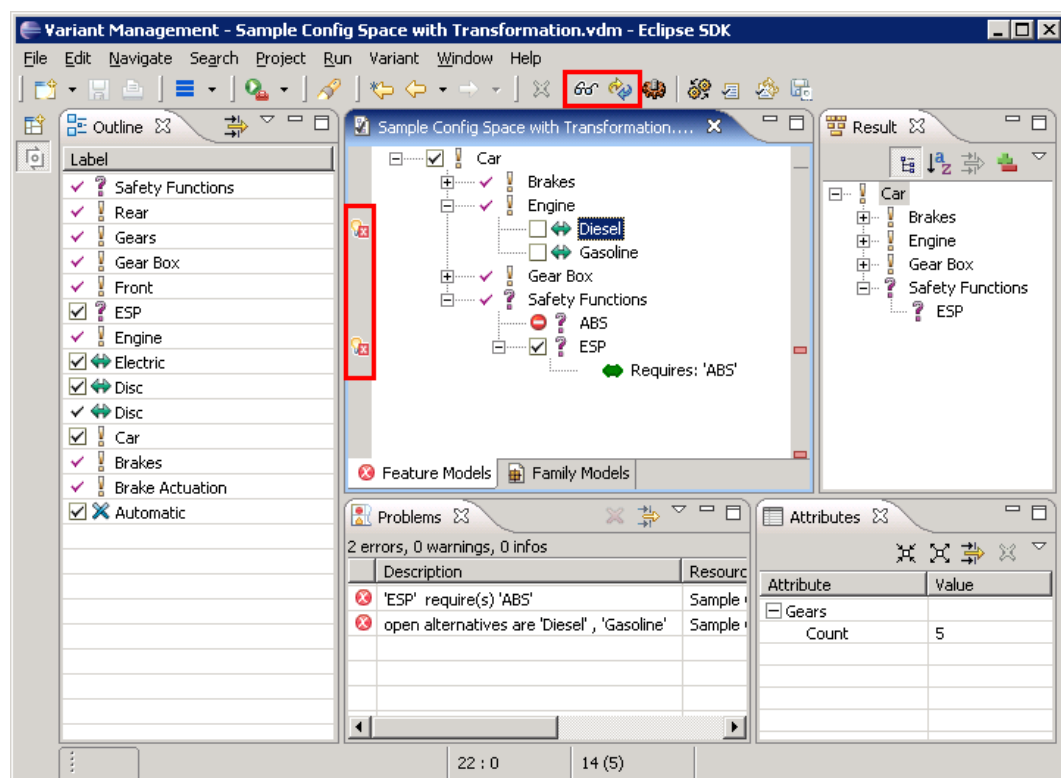


A variant description is evaluated by opening the corresponding VDM in the VDM Editor and clicking on button  in the Eclipse toolbar. Detected selection problems are shown as problem markers on the right side of the editor window and in the Problems View. On the left side of the editor window only those markers are shown that point to problems in the currently visible part of the model. Clicking on these markers may open a list with fixes for the corresponding problem.

Figure 6.1. VDM Editor with Outline, Result, Problems, and Attributes View

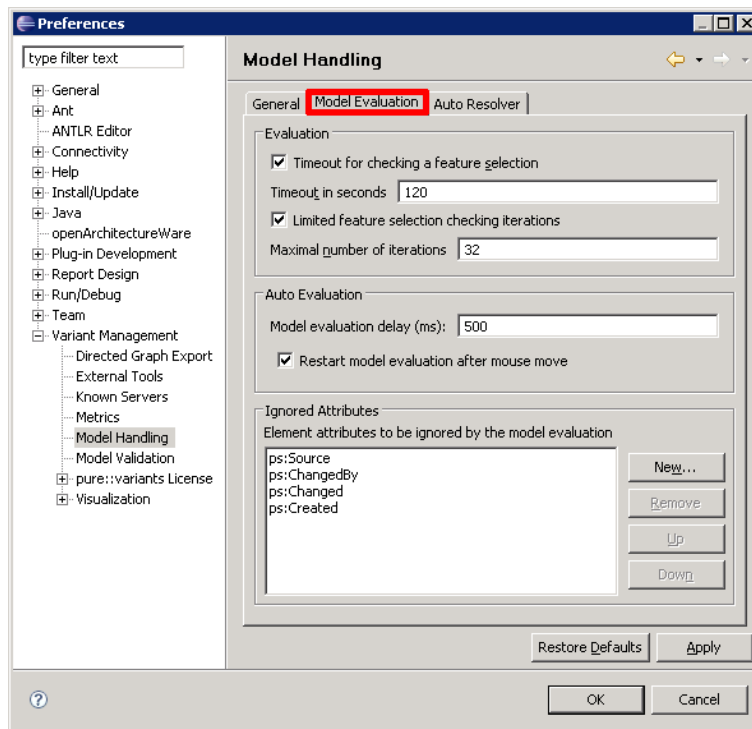


Automatic evaluation of the variant description is enabled by pressing button  in the Eclipse toolbar. This will cause an evaluation of the element selection each time it is changed.

If the variant description is valid, then the result of the evaluation are the concrete variants of the models in the Configuration Space shown in the Result View (see [Section 7.4.8, “Result View”](#)). The concrete variants of the models are collected in the Variant Result Model, that can be saved to an XML file using the button . Saved Variant Result Models can be opened with the VRM Editor. See [Section 5.9.2, “Variant Result Models”](#) for more information about Variant Result Models, and [Section 7.3.5, “Variant Result Model Editor”](#) for a detailed description of the VRM Editor.

6.1.1. Configuring the Evaluation

The model evaluation is configured on the Model Evaluation tab of the Variant Management->Model Handling preferences page (menu *Window->Preferences*, see [Figure 6.2, “Model Evaluation Preferences Page”](#)).

Figure 6.2. Model Evaluation Preferences Page

When the "Evaluate Model" button is clicked in the VDM Editor, the current feature selection is analysed to find and optionally resolve conflicting selections, unresolved dependencies, and open alternatives. Additionally the implicitly selected and mapped features are computed. For this analysis a timeout can be set. It defaults to two minutes which should be long enough even for big configuration spaces. The timeout can be disabled by unchecking the "Timeout for checking a feature selection" check box.

Finding mapped features is an iterative process. Mapped features can cause other features to be mapped and thus included into the selection. The default maximal number of iterations is 32. Depending on the complexity of the dependencies between the mapped features it may be necessary to increase this value. In this case pure::variants will show a dialog saying that the maximal number of iterations was reached. The iterations limit can be disabled by unchecking the "Limited feature mapping iterations" check box.

If the automatic model evaluation is enabled, changing the current feature selection in the VDM Editor causes an automatic evaluation of the Configuration Space. The evaluation process is not started immediately but after a short delay. The default is 500 milliseconds. With the "Restart model evaluation after mouse move" switch it is configured whether the timer for the evaluation delay is reset if the user moves the mouse.

It is possible to define a list of element attributes that are ignored during the model evaluation.

Note

For listed attributes it is not possible to access them in restrictions and calculations during the model evaluation process. These attributes also do not become part of the Variant Result Model, i.e. the concrete models of the variant.

The default list of ignored attributes contains the administrative attributes ps:Source, ps:Changed, ps:ChangedBy, and ps:Created.

6.1.2. Default Element Selection State

For each element in Feature and Family Models a default selection state may be defined. An element with the default selection state "selected" is selected implicitly if the parent element is selected. To deselect this element either the parent has to be deselected or the element itself has to be excluded by the user or the auto resolver.

Normally Family Model elements and mandatory features are created with the "selected" selection state. All other Feature Model elements are created with the "undefined" selection state.

6.1.3. Automatic Selection Problem Resolving

If a feature selection is evaluated to be invalid, selection problems may be occurred. Such selection problems are for instance failed constraints or restrictions. Certain selection problems are eligible to be resolved automatically, e.g. a not yet selected feature that is required by a relation can be selected automatically. pure::variants provides two levels of auto resolving, i.e. basic and extended auto resolving.

The pure::variants basic auto resolver component provides resolving failed relations and feature selection ranges. Auto resolving of failed relations is for instance the automatic selection of required features. Auto resolving of failed feature selection ranges is for instance the automatic selection of a feature of a group of features where at least one has to be selected.

The pure::variants extended auto resolver component additionally provides resolving failed restrictions and constraints. For instance, if only a feature A is selected and there exists a constraint "A requires B" then feature B becomes automatically selected if the extended auto resolver is enabled.

Note

The auto resolver does not change the selection state of user selected or excluded features.


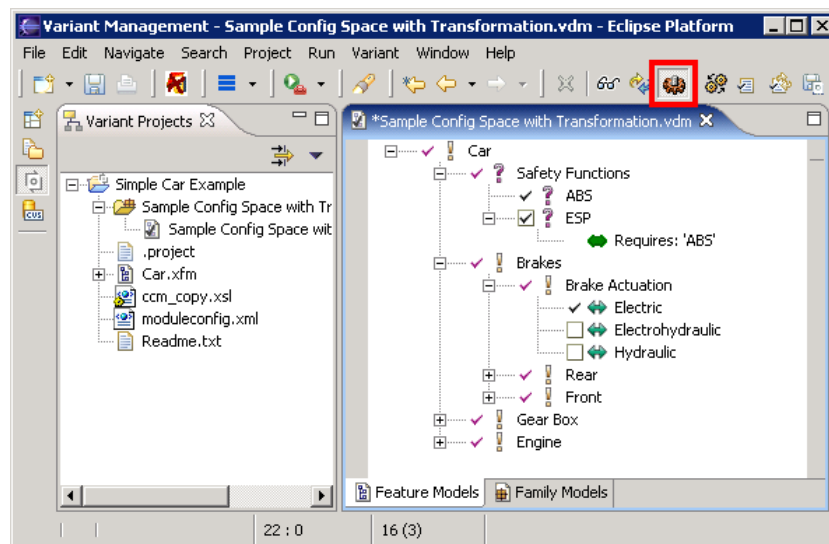
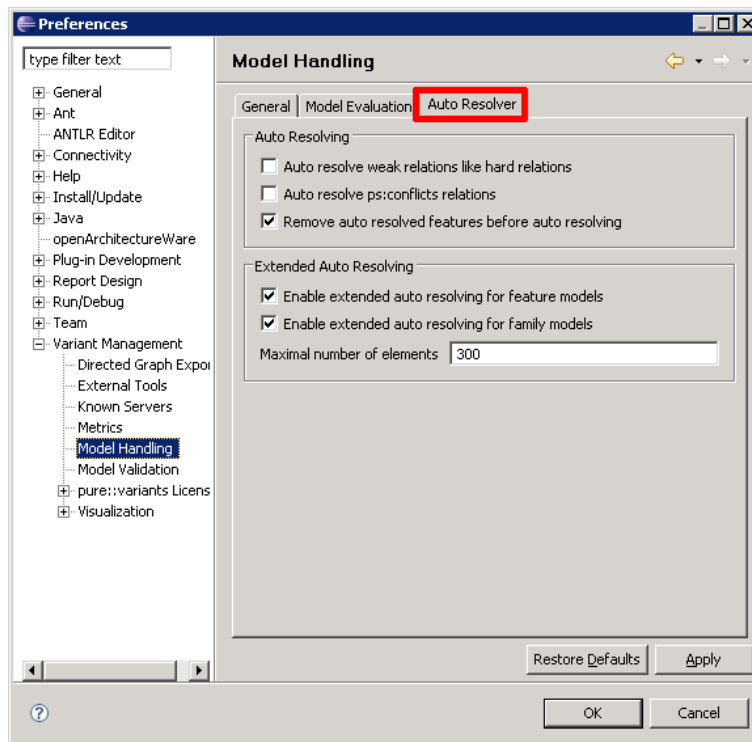
Auto resolving for a VDM is enabled by clicking button  in the tool bar. In [Figure 6.3, "Automatically Resolved Feature Selections"](#) a selection was auto resolved. The feature ABS was automatically selected due to the *Requires* relation on the user selected feature ESP. The feature Electric was automatically selected because it is the default feature of the alternative feature group Electric, Electrohydraulic, Hydraulic. The icons for the different selection types are described in [Section 9.4, "Element Selection Types"](#).

Figure 6.3. Automatically Resolved Feature Selections



6.1.4. Configuring the Auto Resolver

The auto resolving components are configured on the Auto Resolver tab of the Variant Management->Model Handling preferences page (menu *Window->Preferences*, see [Figure 6.4, "Auto Resolver Preferences Page"](#)).

Figure 6.4. Auto Resolver Preferences Page

Usually weak relation types like *ps:recommends* and *ps:discourages* are not considered by the auto resolver. Checking box "Auto resolve weak relations..." causes the auto resolver to handle weak relations like hard relations. In detail, *ps:recommends* is handled like *ps:requires*, i.e. select the required feature if possible. And *ps:discourages* is handled like *ps:conflicts*, i.e. exclude conflicting features if they were automatically selected by a *ps:recommends* relation.

Conflicts usually are not automatically resolved. Checking box "Auto resolve ps:conflicts relations" enables a special auto resolving for conflicts. If the conflicting feature was automatically selected due to a *ps:recommends* relation, then this feature becomes automatically excluded.

To get a clean selection before evaluating a model, i.e. a selection only containing user decisions, "Remove auto resolved features..." has to be enabled.

The extended auto resolver can be enabled for Feature and Family Models separately. Depending on the complexity of the Input Models, measured by counting the number of variation points, the extended auto resolver may exceed the memory and time limits of the model evaluation component of pure::variants. In this case the extended auto resolver aborts. To solve this problem following actions may be tried:

- Disable the extended auto resolver for Family Models. In most of the cases extended auto resolving is not interesting for Family Models.
- Review the models and try to reduce its complexity. This can be done for instance by flatten nested alternatives.
- Increase the model evaluation limits in the preferences.
- Disable the extended auto resolver.

To disable the extended auto resolver automatically if the input models exceed a certain count of elements, a model element count limit can be specified. The default is 300 elements. For models with a very low complexity this limit can be strikingly increased.

6.2. Reuse of Variant Descriptions

6.2.1. Hierarchical Variant Composition

pure::variants supports the hierarchical composition of variants as explained in [Section 5.6, “Hierarchical Variant Composition”](#). A variant hierarchy is set up by creating links to VDMs or Configuration Spaces in a Feature Model. Three different kinds of links are available:

- Variant Reference

A variant reference is simply a link in a Feature Model to a concrete VDM of another Configuration Space. The selections in the linked VDM are locked and can not be changed in the resulting variant hierarchy.

- Variant Collection

A variant collection is a link in a Feature Model to another Configuration Space. The VDMs defined in this Configuration Space are automatically linked. The selections in the linked VDMs are locked and can not be changed in the resulting variant hierarchy.

- Variant Instance

A variant instance is a link in a Feature Model to another Configuration Space. In a VDM of a Configuration Space with this Feature Model as input, it is possible to create concrete *Instances* below the variant instance link, which just means to construct a new linked VDM with an empty and free editable selection for the linked Configuration Space.

While Feature Models from a linked Configuration Space are directly linked below the link elements of the parent Feature Model, the Family Models from the linked Configuration Space are linked into the first Family Model of a corresponding Configuration Space, flat below the special element *LINKED_FAMILY_MODELS* that is automatically created.

Note

Intentionally there is no restriction towards linking VDMs and Configuration Spaces recursively. Thus it is possible for example to link a VDM which itself links other VDMs or whole Configuration Spaces.

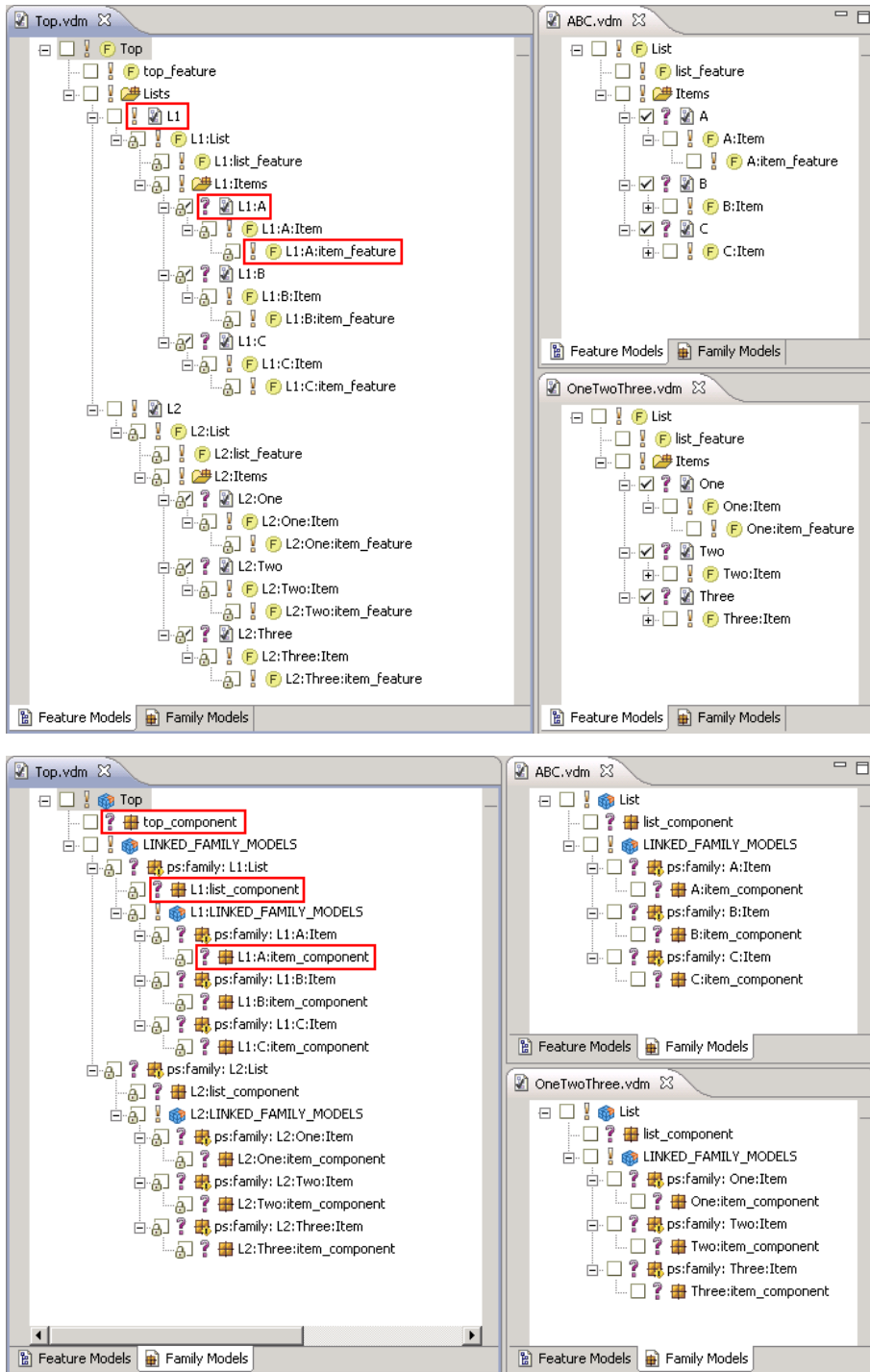
To create a link to a Configuration Space or VDM below an element of a Feature Model select that element, click right and select the wanted kind of link from the context menu (one of *Variant Reference*, *Variant Collection* or *Variant Instance*). This opens a wizard that allows to select the Configuration Space or VDM to link. In case of a variant collection link additionally the variation type of the link element has to be specified. The actual linking of VDMs and Configuration Spaces is not performed directly in the Feature and Family Models containing the links. It is performed when opening the VDMs of a corresponding Configuration Space.

If a variant instance link is created, then the VDM Editor provides two additional actions in the context menu on the corresponding link elements, i.e. *New->Instance* and *Remove Instance*. These actions allow to create and remove the concrete instances, i.e. VDMs, of the linked Configuration Space.

Relations between the variants of a variant hierarchy can be expressed using restrictions and constraints. See [Section 9.8.9, “Name and ID References”](#) and [Section 9.7.1, “Element References”](#) for details on how to reference elements from specific variants.

Unique Names and IDs in linked Variants

To distinguish multiple instances of the same variant in a variant hierarchy, all IDs and the element unique names in the models of each linked variant are changed according to the position of the variant in the hierarchy. Element unique names are prefixed with the unique name of the corresponding link element in the parent variant, separated by a colon (":"). If the parent variant is not the top of the variant hierarchy, then the unique names of its elements also are prefixed this way. [Figure 6.5, “Unique Names in a Variant Hierarchy”](#) and shows a hierarchy of three variants and how the unique names are prefixed in each variant.

Figure 6.5. Unique Names in a Variant Hierarchy


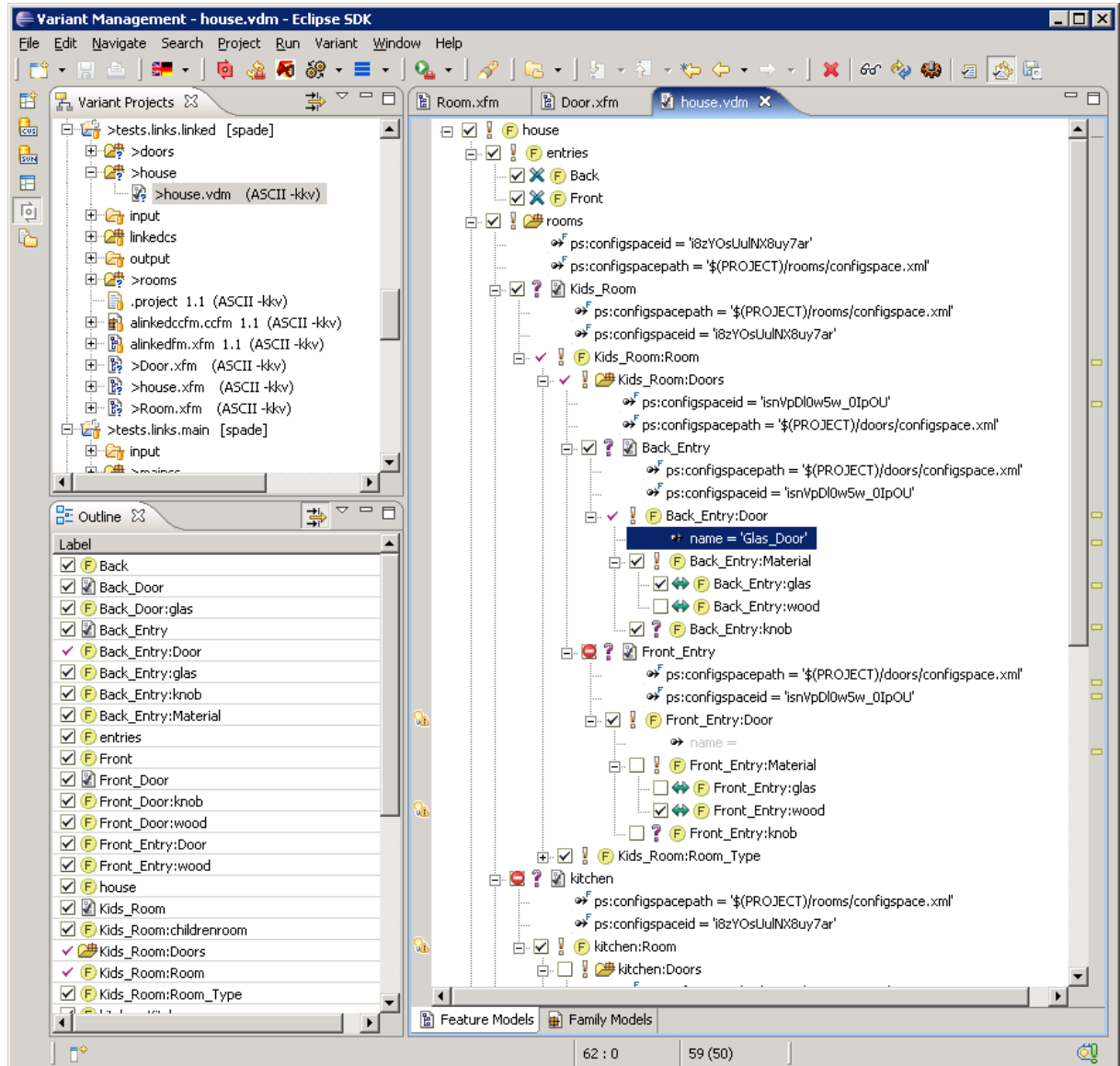
The unique IDs are prefixed in the same way except that the unique ID of the link elements is used as prefix.

Example Variant Hierarchy

Figure Figure 6.6, “Example Variant Hierarchy” shows how a simple house is modeled using Hierarchical Variant Composition. The VDM *house* is top-level and contains a Variant Instance Link named *rooms*. The house contains a kitchen, a kids room, a living room and a bedroom. The figure shows the kids room and the kitchen. These rooms are linked VDMs with the name *room*. This name is prefixed with the name of the corresponding Variant Instance Link element, i.e. *Kids_Room:Rooms*. This ensures uniqueness of the element unique names. Same rule is applied to the element IDs. The *room* VDM also contains a Variant Instance Link with name *doors*. It refers

to the *doors* Configuration Space, visible on the left. For the kids room two doors are available, i.e. *Back_Entry* and *Front_Entry*. Note the exclusions in this model. For the concrete house the kitchen is excluded, and for the kids room the back door is also excluded. The exclusion causes the Model Evaluator not to propagate selections of elements that are below the excluded element. Thus the selection is valid although for example *kitchen:Doors* or *Front_Entry:Material* are explicitly selected. Warnings are shown to give the user a hint for this fact, e.g. *Excluded 'kitchen' overwrites selection of kitchen:Room*.

Figure 6.6. Example Variant Hierarchy



6.2.2. Inheritance of Variant Descriptions

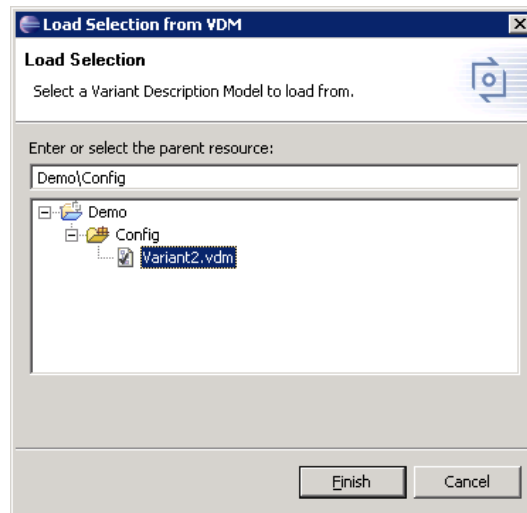
pure::variants supports sharing common feature selections/exclusions between several variant descriptions. This allows users to define the models for each VDM from which selections are to be inherited. Changes in the inherited model selection will be propagated automatically to all inheriting models. Inheritance is possible across Configuration Spaces and projects. See [Section 5.7, “Inheritance of Variant Descriptions”](#) for details.

The VDM inheritance hierarchy can be configured on the Inheritance Page of the Model Properties. See [Section 7.5.3, “Inheritance Page”](#) for a detailed description of this page.

6.2.3. Load a Variant Description

It is possible to load the feature selection from another VDM into the currently edited VDM. Right-click in the VDM Editor window and choose **Load Selection from VDM** from the context menu. This opens the dialog shown in [Figure 6.7, “Load Selection Dialog”](#).

Figure 6.7. Load Selection Dialog



In this dialog the VDM from which to load the selection has to be selected. All selections in the currently edited VDM are overwritten with the selections from the loaded VDM.

6.3. Transforming Variants

pure::variants supports user-defined generation of product variants, described by Variant Description Models, using an XML-based transformation component. See [Section 5.9, “Variant Transformation”](#) for a detailed information about the transformation process.


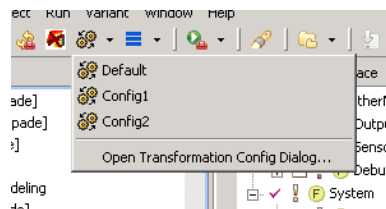
A VDM is transformed by opening it in the VDM Editor and clicking on button  in the Eclipse toolbar. If more than one transformation is defined in a Configuration Space then this button can be used to open the list of defined transformations and to choose one. Additionally this button allows to open the Transformation Configuration Page of the corresponding Configuration Space to add, remove, or modify transformations.

Figure 6.8. Multiple Transform Button



6.3.1. Setting up a Transformation

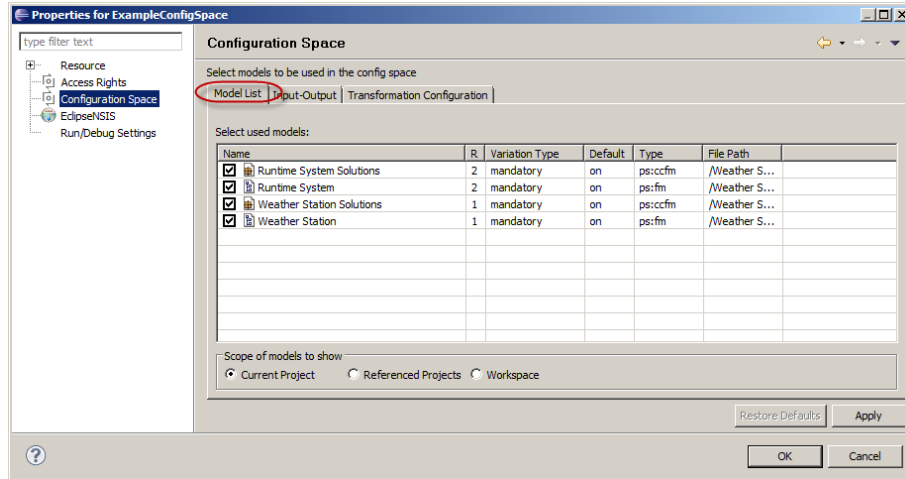
The transformation must initially be set up for a specific Configuration Space. Therefore the Configuration Space properties have to be opened from the Variant Projects view by choosing **Properties** from the context menu of the corresponding Configuration Space.

The editor is divided into three separate pages, i.e. the **Model List** page, the **Input-Output** page, and the **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 Space's project are shown.

Figure 6.9. Configuration Space properties: Model Selection



In the second column ("R") of the models list the rank of a model in this Configuration Space is specified. The 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 third column enables the user to select the variation type of a pure::variant model. Two variation types are available **mandatory** and **optional**. An optional model can be deselected in a variant, mandatory models are always part of the variant.

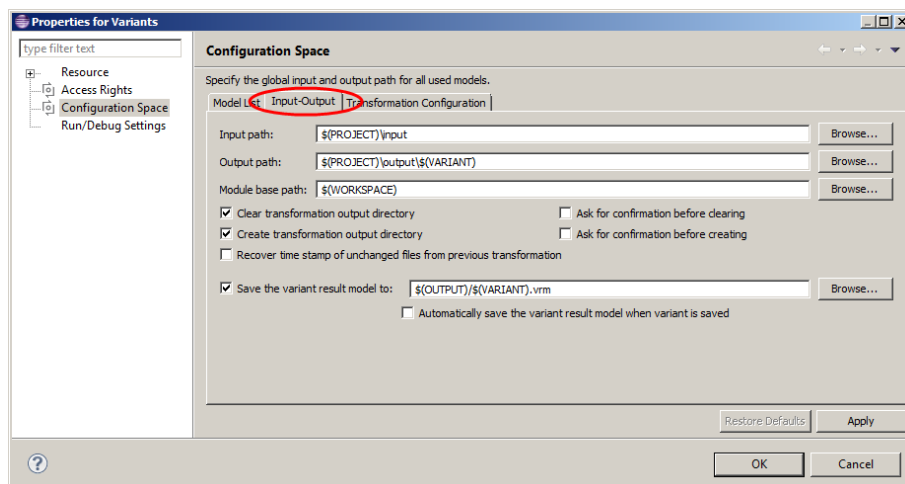
The next column ("Default") can be used to specify whether a optional model is default selected in the variants or not. This semantic is either equal to the default selected state of pure::variants model elements.

Clicking right in the models list opens a context menu providing operations for changing the model selection, i.e. *Select all*, *Deselect all*, and *Negate selection*.

Input-Output Page

This page is used to specify certain input and output options to be used in model transformations. The page can be left unchanged for projects with no transformations.

Figure 6.10. Configuration Space properties: Transformation input/output paths



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 module 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. To see which variables are available for path resolution in transformations refer to [Section 9.10, "Predefined Variables"](#)

The *Clear transformation output directory* check box controls whether pure::variants removes all files and directories in the Output path before a transformation is started. The *Ask for confirmation before clearing* check box controls whether the user is asked for confirmation before this clearing takes place. The remaining check boxes 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.

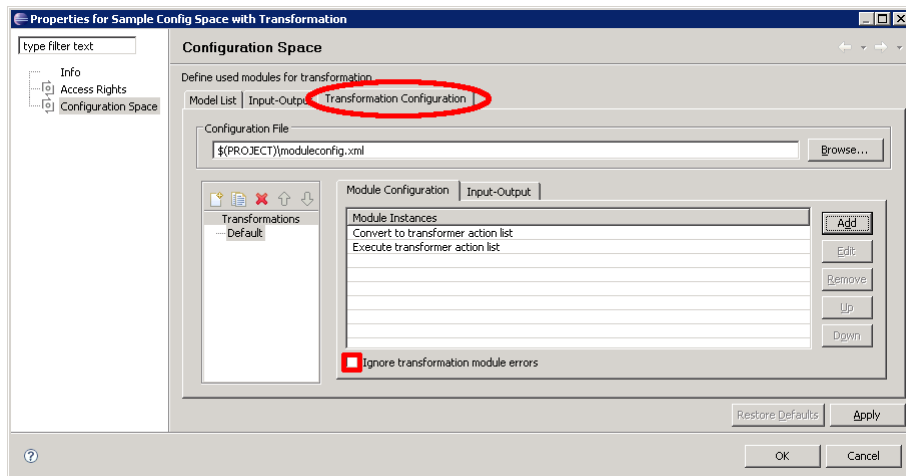
The "Save the variant..." option instructs the transformation framework to save the Variant Result Model to the given location. The Variant Result Model is the input of the transformation framework containing the concrete variants of the models in the Configuration Space.

The option "Automatically save the variant result model when variant is saved" does instruct pure::variants to save the Variant Result Model each time the corresponding Variant Description Model is saved.

Transformation Configuration Page

This page is used to define the model transformation to be performed for the Configuration Space. The transformation configuration 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 6.11. Configuration Space properties: Transformation Configuration



The Transformation Configuration Page allows to define a free number of *Transformation Configurations* which all will be available for the Configuration Space. The lower left part of the Transformation Configuration Page allows to create, duplicate, delete and move *Module Configuration* entries up and down. After pressing the left most button *Add a Module Configuration* a new entry is added immediately whose name can be changed as desired. If a complex *Module Configuration* is created it might be useful to create a copy of it and edit it afterwards. Use the button right to the add button *Copy selected Module Configuration* for this task. Following buttons allow to delete and move a *Module Configuration*.

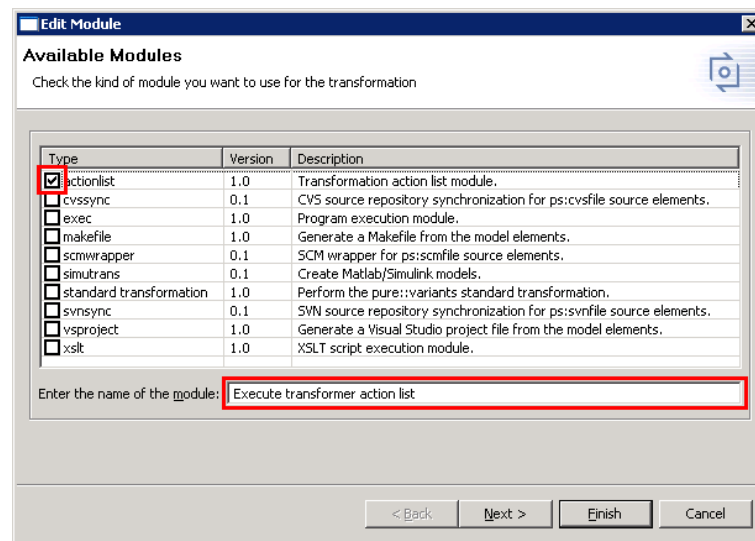
When a Transformation Configuration is selected on the left side, it can be edited with the lower right part of the Transformation Configuration Page. A Module Configuration consists of a list of configured modules. Since many

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 and by the kind of modules. This order in the list can be changed using the Up and Down buttons.

If the "Ignore transformation module errors" button on the bottom of the right page is checked, errors reported by transformation modules do not cause the current transformation to be aborted. Use this option with caution, it may lead to invalid transformation results.

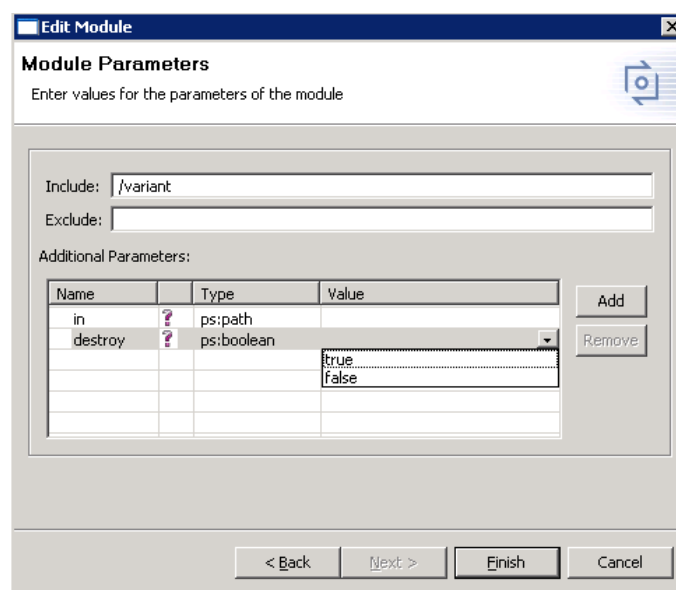
The buttons on the right side allow transformation modules to be added to or removed from the configuration, and to be edited. When adding or editing a transformation module a wizard helps to enter or change the module's configuration.

Figure 6.12. Transformation module selection dialog



In the transformation module selection dialog a name has to be entered for the chosen transformation module. The module parameters are configured in the "Module Parameters" dialog opened when clicking on button Next.

Figure 6.13. Transformation module parameters



A transformation module can have mandatory and optional parameters. A module can not be added to the list of configured modules as long as there are mandatory parameters without a value. Module parameters have a

name and a type. If there are values defined for a parameter, a list can be opened to choose a value from (see [Figure 6.13, “Transformation module parameters”](#)). If a default value is defined for a parameter, then this value is shown as its value if no other value was entered. Some modules accept additional parameters that can be added and removed using the Add and Remove buttons. Additional parameters are always optional and can have any name, type, and value.

The Include and Exclude input fields are used to specify the nodes of the transformation input document on which the module is executed during the transformation. Therefor the Include field contains an XPath expression describing the set of nodes on which the transformation module is to be bound. The Exclude field contains an XPath expression describing the set of nodes on which the transformation module is not to be bound. During the transformation the module is executed on each of the nodes described by the Include expression and not included in the set of nodes described by the Exclude expression. The structure of the transformation input document is explained in [Section 5.9.2, “Variant Result Models”](#).

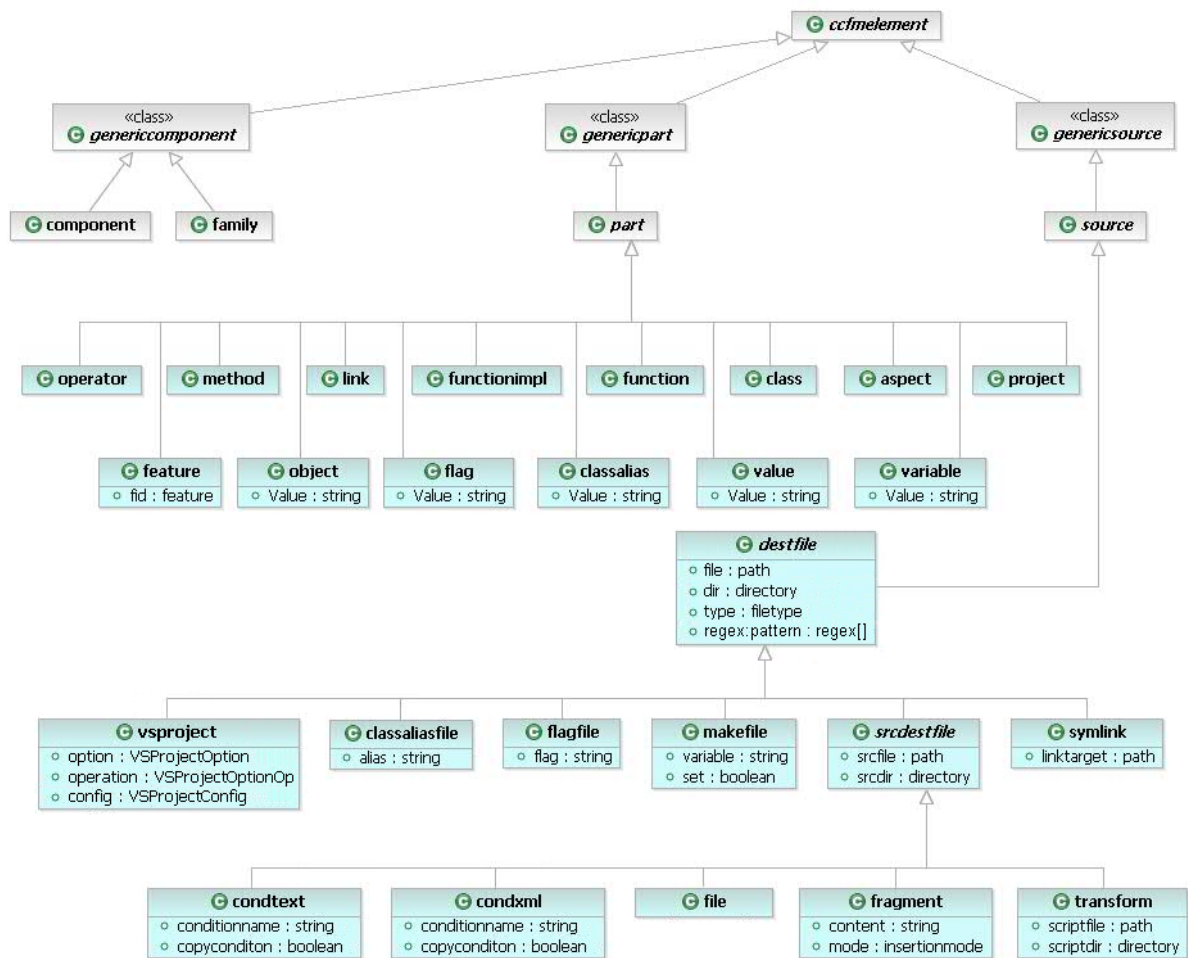
For a special Module Configuration it is also possible to specify special Input and Output paths, which overwrite the settings from Configuration Space. The Input and Output paths can be edited when selecting the *Input-Output* tab as shown in [Figure 4.4, “Transformation configuration in Configuration Space Properties”](#). Layout and behavior are identical to the Input-Output Page of the Configuration Space Properties Dialog with the exception that *Module base path* and the *Save the variant result model to* fields are not available. The use of Module Configuration specific Input and Output paths can be enabled with the check button *Use configuration specific input-output settings*.

Please see [Section 5.9, “Variant Transformation”](#) for more information on model transformation.

6.3.2. 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.

The standard transformation is based on a type model describing the available element types for Family Models (see [Figure 6.14, “The Standard Transformation Type Model”](#)).

Figure 6.14. The Standard Transformation Type Model

The standard transformation supports a rich set of part and source elements for file-oriented variant generation. For each source and part element type a specific transformation action is defined in the standard transformation. 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. For a detailed description of the standard transformation relevant source element types see [Section 9.5, “Predefined Source Element Types”](#).

The supported part element 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. For a detailed description of the standard transformation relevant part element types see [Section 9.6, “Predefined Part Element Types”](#).

Setting up the Standard Transformation

The transformation configuration for the standard transformation is either set up when a Configuration Space is created using the wizard, or can be set up by hand 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 for instance "Generate Standard Transformation Actionlist".
- Add an "Actionlist" module. Leave the include pattern as */variant* and all other parameters empty. Name it for instance *Execute Actionlist*. Usually there should be only one "Actionlist" module for an include pattern, otherwise the action list gets executed twice.

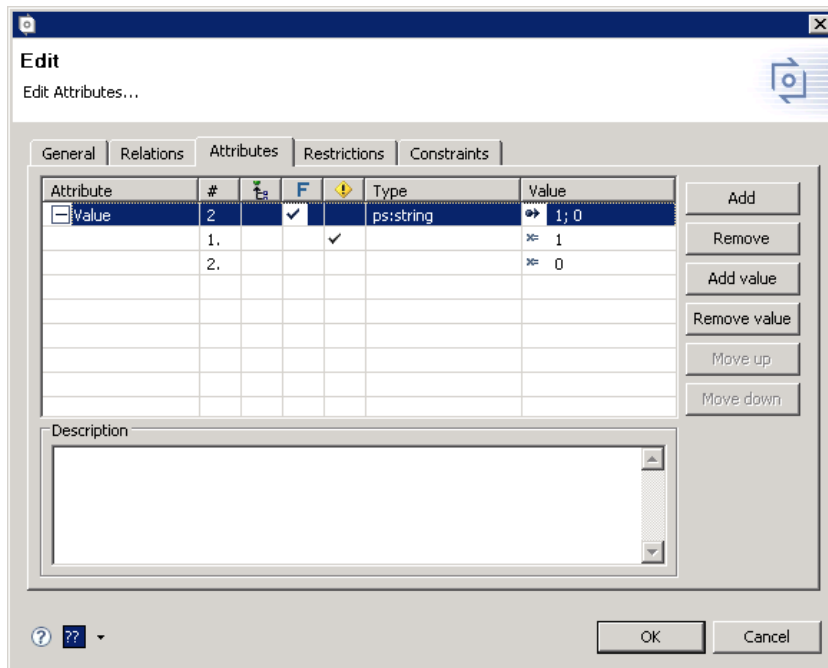
Providing Values for Part Elements

Some of the part element 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 a C preprocessor `#define` generated by a *ps:flagfile* source element. Unless noted otherwise any part element with an attribute `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 (as Makefile variable) and a header file (as preprocessor `#define`).

Calculation of the value of a *ps:flag* or *ps:variable* part element is based on the value of attribute `value`. The value may be a constant or calculation. There may be more than one attribute `value` defined on a part with maybe more than one value guarded by restrictions. The attributes and its values are evaluated in the order in which they are listed in the Attributes page of the element's Properties dialog. The first attribute resp. attribute value with a valid restriction that evaluates to *true* or without a restriction is used.

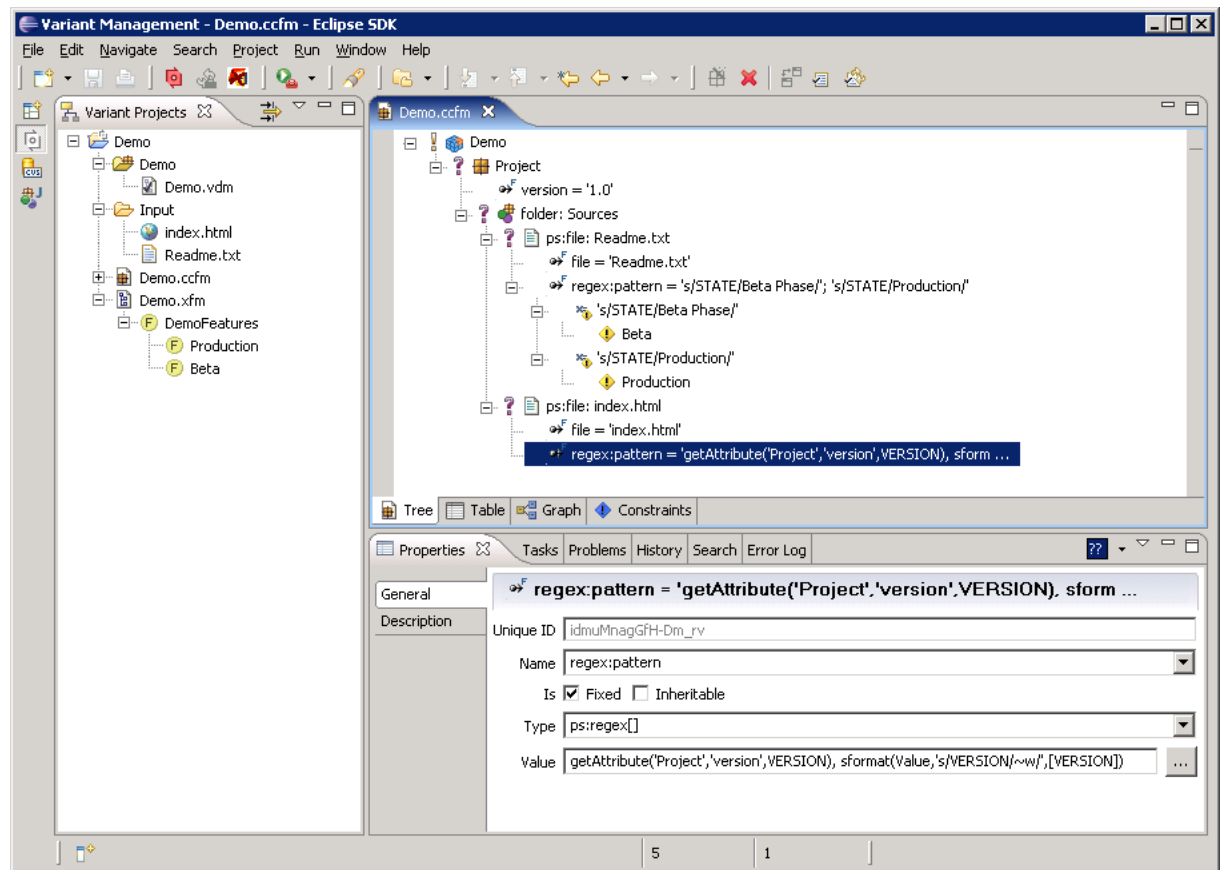
Figure 6.15, “Multiple attribute definitions for Value calculation” shows typical `value` attribute definitions. The value 1 is restricted and only set under certain conditions. Otherwise the unrestricted value 0 is used.

Figure 6.15. Multiple attribute definitions for Value calculation



Modify Files using Regular Expressions

Text based files can be modified during the transformation using a search and replace operation based on regular expressions. For this purpose the file must be modelled by a source element with a type derived from type *ps:destfile*. The regular expression to modify the file is provided in the attribute *regex:pattern* that has to be added to the source element. This attribute can have several values, each containing a regular expression, that are applied to the file in the order they are given.

Figure 6.16. Sample Project using Regular Expressions

Regular Expression Syntax

The syntax of the regular expressions is *sed* based:

```
s/pattern/replacement/flags
```

Prefix *s* indicates to substitute the replacement string for strings in the file that match the pattern. Any character other than backslash or newline can be used instead of a slash to delimit the pattern and the replacement. Within the pattern and the replacement, the pattern delimiter itself can be used as a literal character if it is preceded by a backslash.

An ampersand (`&`) appearing in the replacement is replaced by the string matching the pattern. This can be suppressed by preceding it by a backslash. The characters `"\n"`, where *n* is a digit, are replaced by the text matched by the corresponding back reference expression. This can also be suppressed by preceding it by a backslash.

Both the pattern and the replacement can contain escape sequences, like `"\n"` (newline) and `"\t"` (tab).

The following flags can be specified:

- n** Substitute for the *n*-th occurrence only of the pattern found within the file.
- g** Globally substitute for all non-overlapping strings matching the pattern in the file, rather than just for the first one.

See <http://www.opengroup.org/onlinepubs/000095399/utilities/sed.html> for more details about the *sed* text replacement syntax.

6.3.3. User-defined transformation scripts with JavaScript

In conjunction with the pure::variants JavaScript extension functions JavaScripts can be used to generate product variants. 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 *JavaScript Transformation* module using the *Add* button. Name it for instance *Execute JavaScript*.
- The module parameters can be changed on next page.
- Enter the path to the script file you want to execute as value of the *javascriptfile* parameter.
- An (optional) output file can be specified using the *outputfile* parameter.
- Press Finish to finish set up of the JavaScript transformation.

Example:

To demonstrate how to use JavaScripts for generating a product variant, the following example will show the generation of a text file, which contains a list of used features and some additional information about them. This example uses a user-provided JavaScript. The used JavaScript can also be found in the *Javascript Transformation Example* project.

Within the JavaScript the pure::variant extensibility options can be used. An API documentation is part of the pure::variants Extensibility SDK.

The example JavaScript looks like this:

```
//global variables
var gmodels;
//Package definitions
var ClientTransformStatus = com.ps.consul.eclipse.core.transform.ClientTransformStatus;
var ModelConstants = com.ps.consul.eclipse.core.model.ModelConstants;
var ModelLogic = com.ps.consul.eclipse.core.ModelLogic;

/**
 * This function initializes the script. Global variables are set and all
 * necessary work is done, before transformation can start.
 */
function init(vdm, models, variables, parameter) {
    // initialize global variables - we only use the models here
    gmodels = models;
    // if no error occurred return OK status
    return ClientTransformStatus.STATUS_OK;
}

/**
 * Function work() actually does the transformation work.
 */
function work() {
    try {
        var index;
        //iterator over all models
        for (index in gmodels) {
            var model = gmodels[index];
            // we only want to process Feature Models
            if (model.getType().equals(ModelConstants.CFM_TYPE) == true) {
                var rootid = model.getElementsRootID();
                printFeatures(model.getElementWithID(rootid));
            }
        }
    } catch (e) {
        // If something went wrong, catch error and return error status with specific error
        message.
    }
}
```



```

    return new ClientTransformStatus(ClientTransformStatus.ERROR, e.getMessage());
}
// if no error occurred return OK status
return ClientTransformStatus.STATUS_OK;
}

function printFeatures(element) {
    // add information to output file
    out.println("Visible Name: " + element.getVName());
    out.println("Unique Name: " + element.getName()+"\n");
    out.println("Description:\n");

    /**
     * Because the description of a feature is stored in HTML in this model, and
     * we don't want to see the HTML tags in our output file, we are doing some formatting here.
     */
    out.println(element.getDesc(null, "text/html"));
    out.println("-----\n");

    // get Children of current element from ModelLogic
    var iter = ModelLogic.getOrderedChildren(element).iterator();
    while (iter.hasNext()) {
        printFeatures(iter.next());
    }
}

/**
 * Function done() does necessary work after the transformation. Nothing to do in this
 * example.
 */
function done() {
    // if no error occurred return OK status
    return ClientTransformStatus.STATUS_OK;
}

```

The script consists of three main functions. These three functions will be called by the transformation module. All three have to be used.

- `init()`

Necessary work is done here, before transformation starts, like initializing the script. Gets necessary information from transformation module, like the used variant model, the used models in this variant, some variables and the transformation parameters.

- `work()`

Does the whole transformation work.

- `done()`

After transformation is finished, this function is called, to provide possibility to do some work after transformation.

If the transformation parameter *outputfile* was used, the variable *out* can be used to write directly to the given file. Otherwise the variable *out* writes to the Java standard output.

6.3.4. User-defined transformation scripts with XSLT

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 for instance *Execute XSLT script*.

- 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.

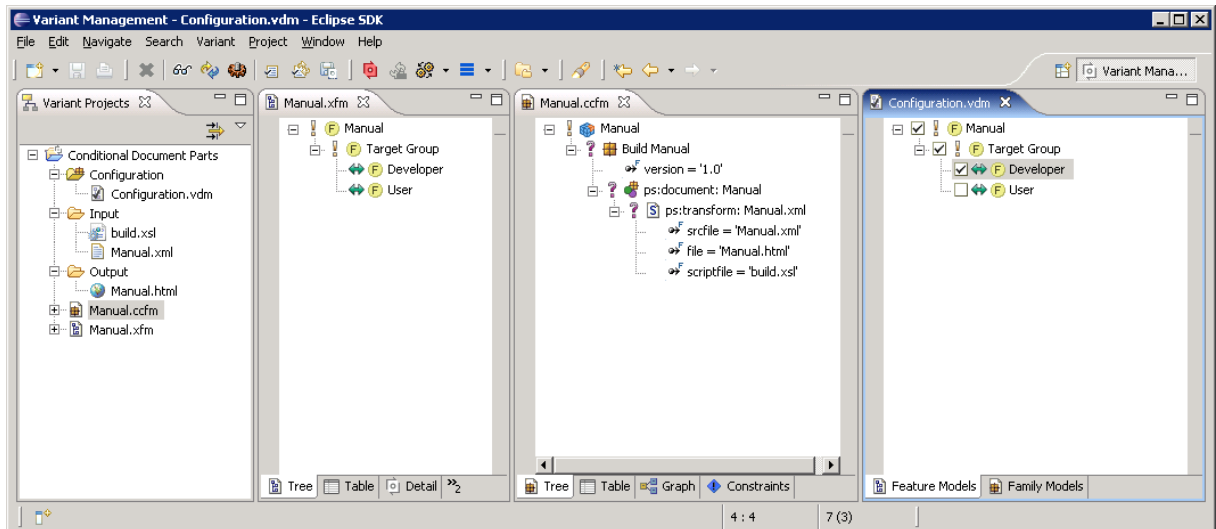
Example: Conditional Document Parts

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 implementing a lite version of the *ps:condxml* source element functionality. 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 6.17, “Variant project describing the manual”](#) shows the corresponding Feature and Family Models in a variant project using the standard transformation.

Figure 6.17. 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 [Section 9.5, “Predefined Source Element Types”](#)). This source element refers to the XSLT script `build.xml` 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>
```

```
</xsl:template>

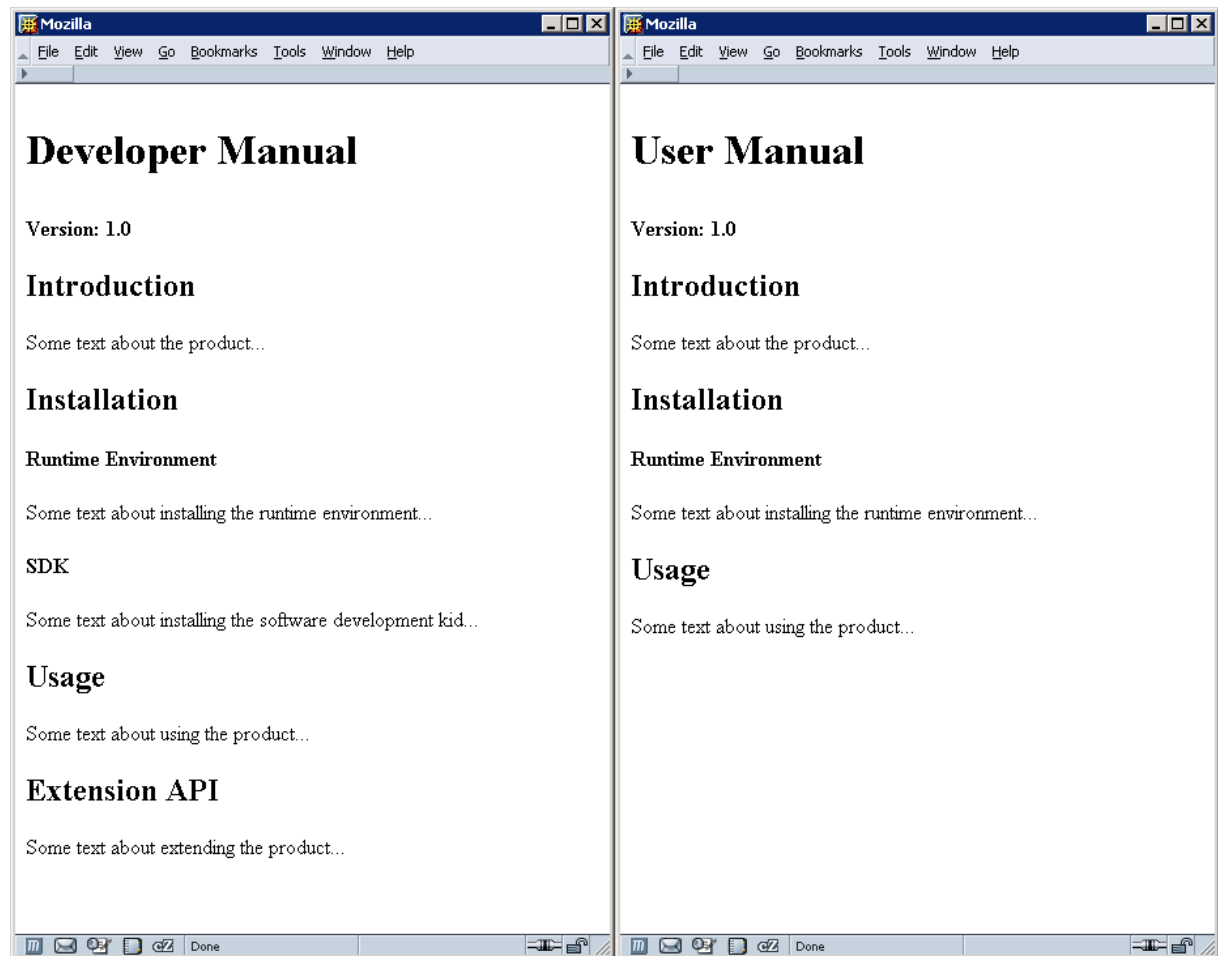
<!-- match subsection -->
<xsl:template match="subsection" mode="body">
  <xsl:if test="not(@condition) or dyn:evaluate(@condition)">
    <h4><xsl:value-of select="@name"/></h4>
    <xsl:apply-templates mode="body"/>
  </xsl:if>
</xsl:template>

<!-- 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 9.17, “Extension functions providing model information”](#)). 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, `getAttribute\value()`. 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 6.18, “The manual for users and developers”](#) shows the two variants of the manual (HTML) generated selecting target group user and then developer.

Figure 6.18. The manual for users and developers

6.3.5. Transformation of Hierarchical Variants

When a transformation of a hierarchical variant is performed then a single transformation is performed for each variant in the hierarchy. Only those transformations of linked variants are executed that have the name "Default" or the name of the top-level variant transformation (if not "Default").

The order of the transformations is top-down, i.e. first the top-level variant is transformed, then the variants below the top-level variant, and so on. Each single transformation is performed on the whole Variant Result Model, stating two lists of model elements, i.e. the transformation *Entry-Points* list and the transformation *Exit-Points* list. These lists describe the section of the Variant Result Model that represents the variant to transform. Some transformation modules may not support these lists and always work on the whole Variant Result Model.

There is a special variable `$(VARIANTSPATH)` that should be used in a transformation of hierarchical variants to specify the transformation output directory. This variable contains the name of the currently transformed variant (VDM) prefixed by the names of its parent variants (VDMs) according to the variant hierarchy. The variant names are separated by a slash ("/"). Using this variable makes it possible to build a directory hierarchy corresponding to the variant hierarchy. This may also avoid that the results of the transformation of one variant are overwritten by the results of the transformation of another variant. See [Section 9.10, "Predefined Variables"](#) for more information on the use and availability of variables.

Transformations of linked variants have to handle the prefixed unique names and IDs in the models of the variant (see [the section called "Unique Names and IDs in linked Variants"](#)). Especially XSLT and Conditional Text resp. Conditional XML transformations have to reference elements with their full, i.e. prefixed, name. If for instance the condition in a file transformed with Conditional Text is `"pv:hasFeature('Foo')"` then this condition always will fail if evaluated in the context of a linked variant. The correct condition would be `"pv:hasFeature('Link1:Foo')"`, if linked below the link element with unique name "Link1".

6.3.6. Reusing existing Transformation

The transformation module *Reuse Transformation* provides the possibility to reuse already existing transformation configurations. These existing configurations can be run with the first vdm, the last vdm or with each vdm of a configspace or vdm selection.

The *Reuse Transformation* module has two mandatory parameter.

The first parameter *Triggered by* defines for which vdm of the current transformation the reused transformation configuration is triggered. The three allowed values *First VDM*, *Each VDM* and *Last VDM* are provided in a combo box. *Each VDM* is the default.

The second parameter *Transformation* defines the name of the transformation configuration, which will be triggered by this module.

The configuration space settings are inherited as follows:

Table 6.1. Configuration Space Settings

Input Directory	Used from the <i>Reuse Transformation</i> configuration, if defined. From Configuration Space otherwise.
Output Directory	Used from the <i>Reuse Transformation</i> configuration, if defined. From Configuration Space otherwise.
Create Output Directory	Used from the <i>Reuse Transformation</i> configuration, if defined. From Configuration Space otherwise.
Cleanup Output Directory	Used from the <i>Reuse Transformation</i> configuration, if defined. From Configuration Space otherwise.
Create Output Directory	Used from the <i>Reuse Transformation</i> configuration, if defined. From Configuration Space otherwise.
Confirm Create Output Directory	Used from the <i>Reuse Transformation</i> configuration, if defined. From Configuration Space otherwise.
Confirm Cleanup Output Directory	Used from the <i>Reuse Transformation</i> configuration, if defined. From Configuration Space otherwise.
Recover Timestamps	Used from the <i>Reuse Transformation</i> configuration, if defined. From Configuration Space otherwise.
Force Transformation	Always true, because decision was made by user before running <i>Reuse Transformation</i> already.
Save Variant Result Model	Always false, because cannot be defined in transformation configurations. It is configuration space settings only.
Ignore Transformation Errors	Used from the <i>Reuse Transformation</i> configuration.

6.3.7. Ant Build Transformation Module

The transformation module *Ant Build Module* provides the possibility to call an Ant build during the transformation. The module has two parameter.

The first parameter *Build File* defines the location of the Ant build file.

The second parameter *Target* defines the target for the build. If no target is given the default target of the Ant build file will be used.

6.4. Validating Models

In the context of pure::variants, *Model Validation* is the process of checking the validity of feature, family, and variant description models. Two kinds of model validation are supported, i.e. validating the XML structure of

models using a corresponding XML Schema and performing a configurable set of checks using the model check framework.

6.4.1. XML Schema Model Validation

This model validation uses an XML Schema to check if the XML structure of a pure::variants model is correct. This is pure syntax check, no further analyses of the model are performed.

The XML Schema model validation is disabled per default. It can be enabled selecting option "Validate XML structure of models..." on the Variant Management->Model Handling preferences page (menu *Window->Preferences*). If enabled all pure::variants models are validated when opened.

Note

Invalid models will not be opened correctly if the XML Schema model validation is enabled.

For more information about XML Schema see the [W3C XML Schema Documentation](#).

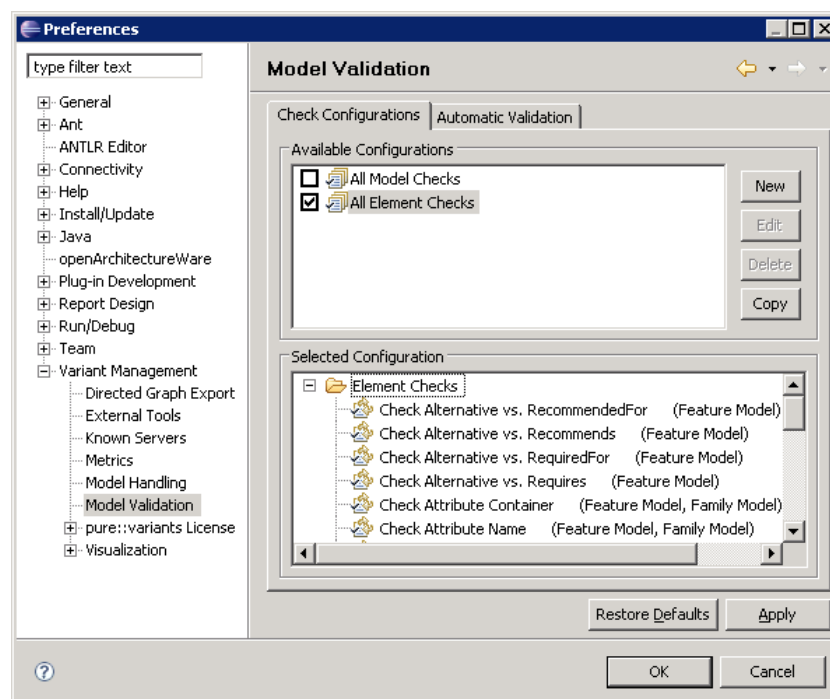
6.4.2. Model Check Framework

The model check framework allows the validation of models using a configurable and extensible set of rules (called "model checks"). There are no restrictions on the complexity of model checks.

Configuring the Framework


The model check framework is configured on the **Variant Management->Model Validation** preference page (menu **Window->Preferences**). On the **Check Configurations** tab the model check configurations can be managed and activated (see [Figure 6.19, "Model Validation Preferences Page"](#)).

Figure 6.19. Model Validation Preferences Page



The two default configurations "All Model Checks" and "All Element Checks" are always available. "All Model Checks" contains all model checks that perform whole model analyses. Compared with "All Element Checks" containing all checks that perform analyses on element level. The configuration "All Element Checks" is enabled per default if the pure::variants perspective is opened the first time.

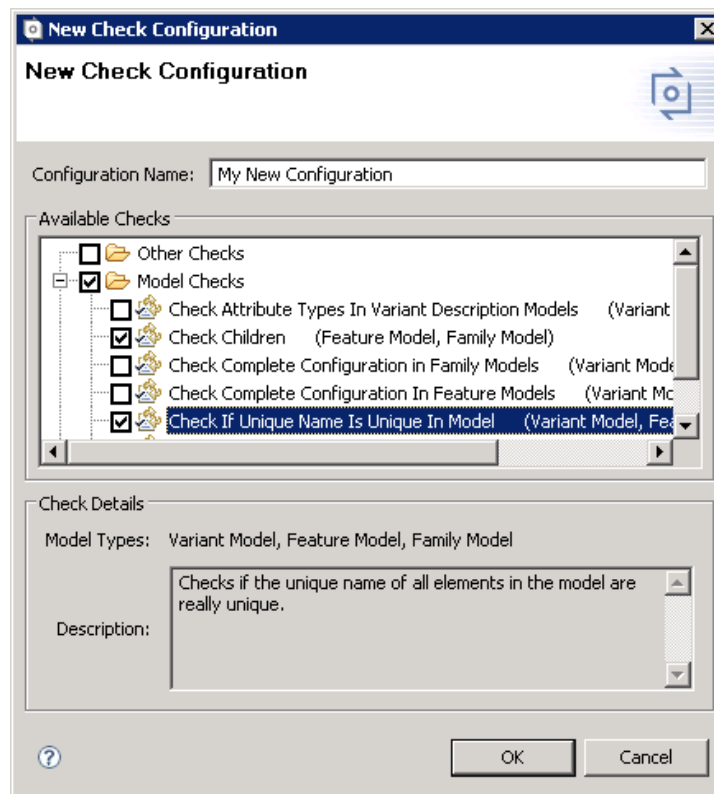
A model check configuration is activated by selecting it in the **Available Configurations** list. If more than one configuration is selected, the checks from all selected configurations are merged into one set that becomes activated.

The checks contained in a configuration are shown in the **Selected Configuration** list by clicking on the name of the configuration. The checks are listed by its names followed by the list of model types supported by a check. Additionally the icon  reveals if the check is enabled for automatic model validation (see [the section called “Performing Model Checks”](#)). A brief description of a check is shown by moving the mouse pointer over the check name.

All but the two default configurations "All Model Checks" and "All Element Checks" can be deleted by clicking first on the name of the configuration and then on button **Delete**.

A new configuration can be created by clicking on button **New**. This will open the **New Check Configuration** dialog as shown in [Figure 6.20, “New Check Configuration Dialog”](#).

Figure 6.20. New Check Configuration Dialog



For a new check configuration a unique name for the configuration has to be entered. The available checks are shown in the **Available Checks** tree and can be selected for the new configuration by clicking on the check boxes of the checks. Clicking on the root of a sub-tree selects/deselects all checks of this sub-tree.

Detailed information about a check are displayed in the **Check Details** area of the dialog if the name of a check is selected. The **Model Types** field shows the list of model types for which the corresponding check is applicable. The **Description** field shows the description of the check.

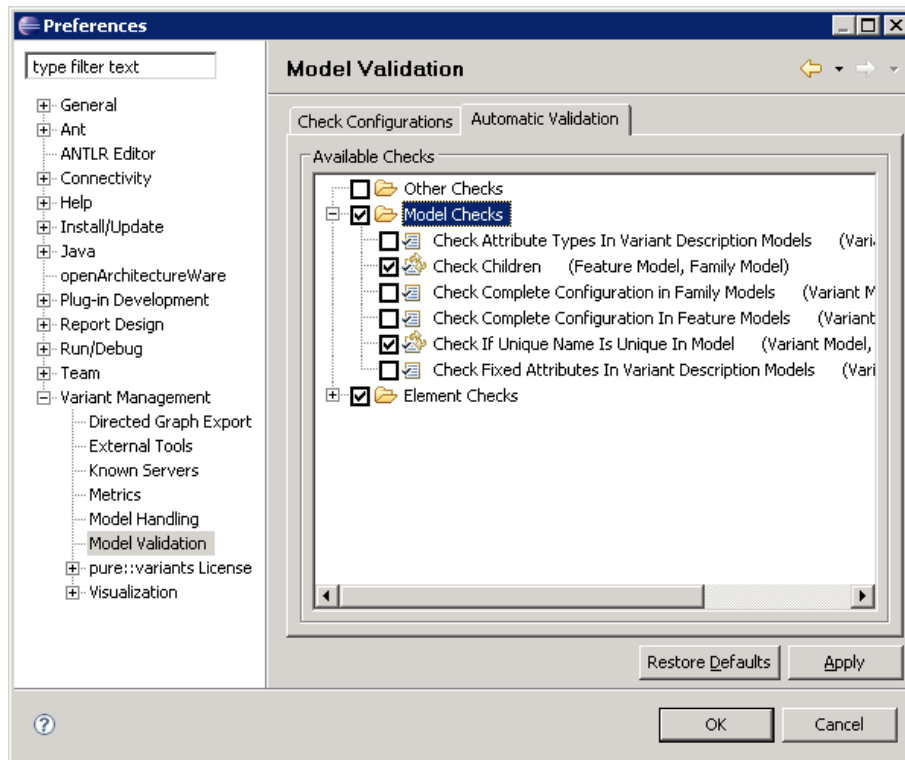
The same dialog appears for editing and copying check configurations using the **Edit** and **Copy** buttons. Only non-default configurations can be edited.

And with the "Enable check for..." button (or clicking on the icon  of a check)

Automatic Model Validation

On the Automatic Validation tab it can be configured which checks are allowed to be performed automatically (see [Figure 6.21, “Automatic Model Validation Preferences Page”](#)). If the automatic model validation is enabled, after every change on the model those checks are performed from the active check configurations that are enabled for automatic model validation.

Figure 6.21. Automatic Model Validation Preferences Page



The **Available Checks** tree shows all known checks independently from the selected check configuration. Clicking on the check box of a check toggles the automatic validation state of the corresponding check. Clicking on the root of a sub-tree toggles all checks of this sub-tree.

A description of the check is shown by moving the mouse pointer over the check name.

Performing Model Checks


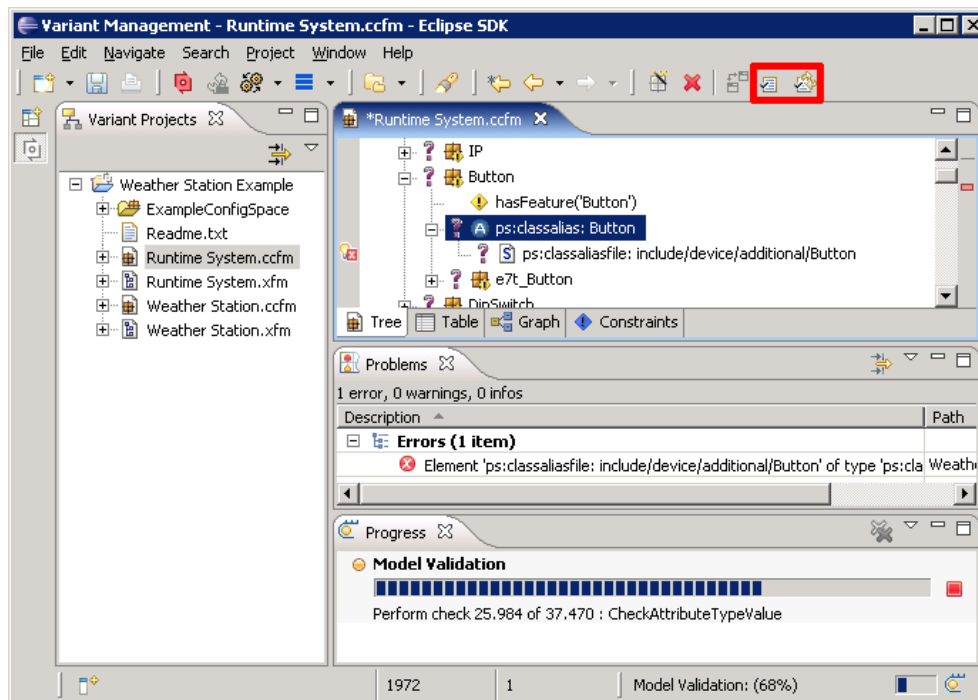

A model can be checked using the selected model check configurations by opening the model in a corresponding model editor and pressing button  in the tool bar. This will start a single model validation cycle. The progress of the model validation is shown in the Progress view.

Figure 6.22. Model Validation in Progress

If no model check configuration is selected a dialog is opened inviting the user to choose a non-empty check configuration. This dialog can be disabled by enabling the "Do not show again" check box of the dialog.

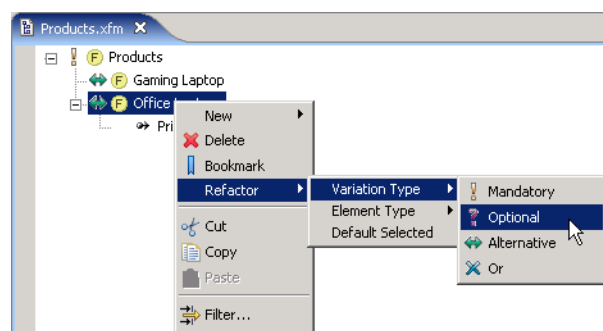
The button  is used to enable automatic model checking, i.e. after every change on the model a new check cycle is started automatically. In contrast to the single model validation cycle only those checks are performed from the active check configurations that are enabled for automatic model validation. Automatic model validation is enabled by default.

The result of a model check cycle is a list of problems found in the model. These problems are shown in the Problems view and as markers on the model. A list of quick fixes for a problem can be shown either by choosing "Quick Fix" from the context menu of the problem in the Problems view or by clicking on the corresponding marker on the model. For some problems special quick fixes are provided fixing all problems of the same kind.

6.5. Refactoring Models

To simplify the editing of Feature and Family Models pure::variants provides a set of refactoring operations. They support the user to efficiently change model objects like elements, relations, restrictions and attributes.

The refactoring operations can be accessed via the context menu of the Feature and Family Model Editors, see [Figure 6.23, "Refactoring context menu for a feature"](#).

Figure 6.23. Refactoring context menu for a feature

The refactoring operations provided in the context menu depend on the selection made in the editor. For instance, select two or more features and right-click on one of the selected features to open the context menu. The appearing *Refactoring* menu contains for example items for changing the variation type. This operation allows to modify the variation type for all selected features at once. Compared with the conventional way that opens the element properties dialog for each feature, refactoring operations save a lot of time. There are numerous operations that can be performed on model objects.

The following list summarizes the available refactoring operations.

Table 6.2. Refactoring Operations

Operation on	Available Operations
Elements	Variation Type Change Element Type Change Default-Selected State Change
Attributes	Attribute Name, Type, and Value Change Inheritable and Fixed State Change
Restrictions and Constraints	Restriction/Constraint Code Change
Relations	Relation Type Change Relation Targets Change

6.6. Comparing Models

In pure::variants two models can be compared using the Model Compare Editor. It is based on the Eclipse Compare.

6.6.1. General Eclipse Compare

In general, comparison of resources is divided into two different types. One is to compare two resources with each other. This is called a two-way compare. A two-way compare can only reveal differences between resources, but can not recognize in which resource a change was performed. A two-way compare in Eclipse is obtained by selecting two resources and then choosing *Compare With->Each Other* from the context menu. Other two-way comparisons supported by Eclipse are *Compare With->Revision* and *Compare With->Local History*.

A more comfortable compare is the so called three-way compare. In addition it has an ancestor resource from which is known that this is the unchanged resource. In this way it can be determined which change was performed in which resource. Such compare editors are opened for instance for synchronizing resources with CVS repositories which always maintain a third ancestor resource by using *Compare With->Latest from Head* and *Compare With->Another Branch or Version*.

The compare editor is divided into an upper and a lower part. The upper part shows structural changes in a difference tree. The lower part presents two text editors located next to each other. Changes are highlighted in colored lines or rectangles on both sides. Those belonging to one change are connected with a line. For two-way comparisons the changes are always grey-colored. In three-way comparisons outgoing (local) changes are grey-colored, incoming (remote) changes blue-colored, and changes on both sides which are conflicting are red-colored.

A resource compare can be used to view changes for two resources. In addition it provides the possibility to apply single changes to local models. Therefore the compare editor provides a toolbar, located between the upper and the lower part, with actions which can be used to apply changes: **Copy All from Left to Right**, **Copy All Non-Conflicting Changes from Right to Left**, **Copy Current Change from Left to Right**, **Copy Current Change from Right to Left**, **Select Next Change**, **Select Previous Change**. You can step through the changes and apply them if the specific buttons are enabled. As stated above refer to the Eclipse [Workbench User Guide](#) for detailed information on this.

6.6.2. Model Compare Editor

In general the Eclipse text compare editor is opened for any resource after calling the actions described in the previous section. For pure::variants models the special pure::variants Model Compare Editor is opened. This makes

it easier to recognize changes in pure::variants models. Typical changes are for example *Element Added*, *Attribute Removed*, *Relation Target Changed*.

The upper part of the editor, i.e. the structure view, displays a patch tree with a maximum depth of three. Here all patches are grouped by their affiliation to elements. Thus *Element Added* and *Element Removed* are shown as top level patches. All other patches are grouped into categories below their elements they belong to. Following categories exist: **General**, **Attributes**, **Relations**, **Restrictions**, **Constraints** and **Misc**. The names of the categories indicate which patches are grouped together. Below the category **Misc** only patches are shown that are usually not displayed in the models tree viewer. As in the Eclipse text compare you can step through the patches with the specific buttons. Each step down always expands a model patch if possible and steps into it. The labels for the patch consist of a brief patch description, the label of the patched model item and a concrete visualization of the old and the new value if it makes sense. Here is an example: Attribute Constant Changed: attrname = 'newValue' <-oldValue. In this attribute patch's label a new value is not additionally appended, because it is part of the attributes (new) label "attrname = 'newValue' ".

The lower part of the model compare editor is realized using the usual model tree viewers also used in the model editors. They are always expanded to ensure that all patches are visible. As in the text compare editors, patches are visualized by colorized highlighted rectangle areas or lines using the same colors. In opposite to the text compare they are only shown if the patch is selected in the upper structure view. For two-way comparisons it is ambiguous which model was changed. Because of this an additional button is provided in the toolbar which allows to exchange two models currently opened in the model compare editor. This leads from a remove-patch into an add-patch, and for a change the new and the old value are exchanged.

The model compare editor compares two model resources on the model abstraction layer. Hence textual differences may exist between two models where the model compare editor shows no changes. Thus conflicts that would be shown in a textual compare are not shown in the model compare editor. This allows the user to apply all patches in one direction as desired and then to override into the other direction.

6.6.3. Conflicts

In three-way comparisons it may occur that an incoming and an outgoing patch conflict with each other. In general the model compare editor distinguishes between fatal conflicting patches and warning conflicts. In the tree viewer conflicts are red-colored. A fatal conflict is for example an element change on one side, while this element was deleted on the other side. One of these patches is strictly not executable. Usually warning conflicts can be merged, but it is not sure that the resulting model is patched correctly. Typical misbehaviour could be that some items are order inverted. To view which patch conflicts with which other path just move the mouse above one of the conflicting patches in the upper structure view. This and the conflicting patch then change their background color either to red for fatal conflicts or yellow for conflict warnings.

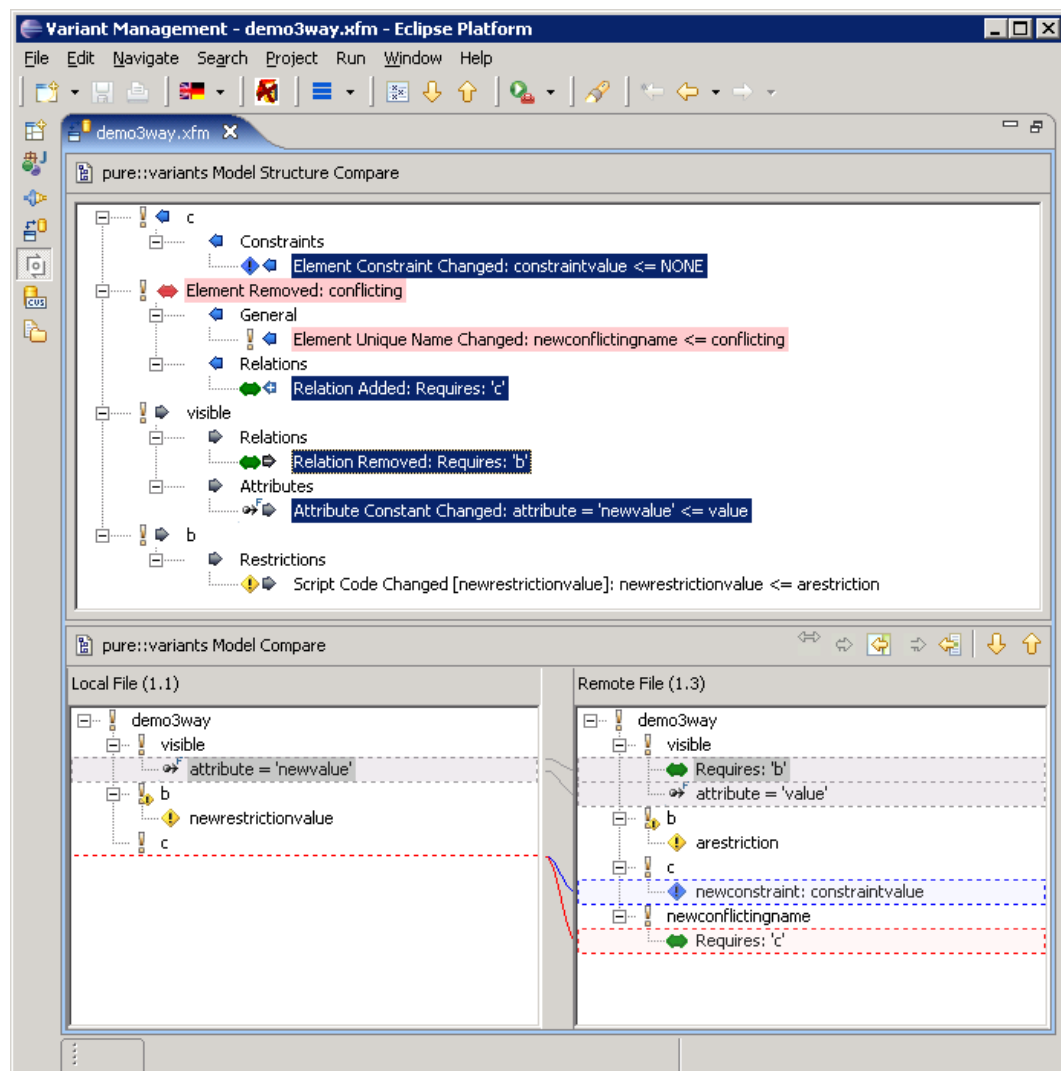
In general a sophisticated algorithm tries to determine conflicts between two patches. These results are very safe hints, but 100% safety is not given. For a conflicting or non-conflicting patch it may occur that it can not be executed. Conflict warning patches may be executed without problems and lead to a correct model change. In general the user can try to execute any patch. If there are problems then the user is informed about that. If there are problems applying a non-conflicting patch, the editor should be closed without saving and reopened. Then another order of applying patches can solve this problem. The actions *Apply All Changes ...* do only apply incoming and non-conflicting changes. Other patches must be selected and patched separately.

6.6.4. Compare Example

Figure 6.24, “Model Compare Editor” shows an example how a model compare editor could look like for a model that is synchronized with CVS. The upper part shows the structure view with all patches visible and expanded representing the model differences. A CVS synchronize is always a three-way compare. There are incoming changes (made in the remote CVS model) and outgoing (local) changes. As to see in the figure the incoming changes have a blue left arrow as icon, while outgoing changes have a grey right-arrow as icon. Added or removed items have a plus or a minus composed to the icon. Conflicting changes are marked with a red arrow in both directions displayed only at the element as the patches top level change. In this example a conflict arises at the element conflicting. In CVS its unique name changed and a relation was added while this element was deleted locally. Two patches show a red background because the mouse hovered above one of these patches which is not visible

in the figure. Note that the tree viewers in the lower part show only the patches which are selected above. The colors correspond to the patch direction.

Figure 6.24. Model Compare Editor

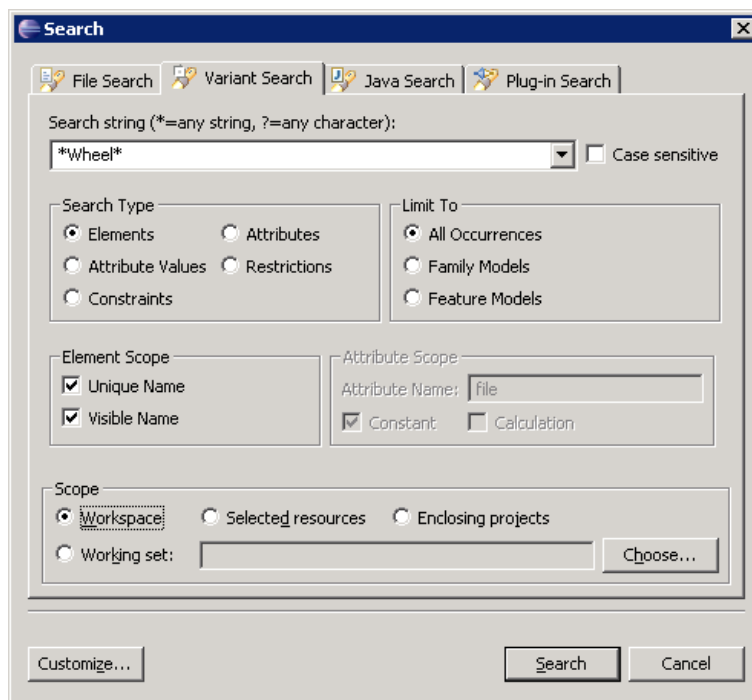


6.7. Searching in Models

6.7.1. Variant Search

Feature and Family Models can be searched using the Variant Search dialog. It supports searching for elements, attributes, attribute values, restrictions, and constraints.

The Variant Search dialog is opened either by choosing the **Search->Variant** menu item, by clicking on the Eclipse **Search** button and switching to the **Variant Search** tab, or by choosing **Search** from the context menu of the model editor.

Figure 6.25. The Variant Search Dialog

The dialog is divided into the following sections.

Search String

The search string input field specifies the match pattern for the search. This pattern supports the wildcards "*" and "?".

Wildcard	Description
?	match any character
*	match any sequence of characters

Case sensitive search can be enabled by checking the "Case sensitive" check box. The settings for previous searches can be restored by choosing a previous search pattern from the list displayed when pressing the down arrow button of the Search String input field.

Search Type

In this group it is specified what kind of model elements is considered for the search.

Elements	Search element names matching the pattern.
Attributes	Search element attribute names matching the pattern.
Attribute Values	Search element attribute values matching the pattern.
Restrictions	Search restrictions matching the pattern.
Constraints	Search constraints matching the pattern.

For refining the search the "Element Scope" group is activated for search type Elements and the "Attribute Scope" group is activated for search type Attribute Values.

Limit To

This group is used to limit the search to a specific model type. The following limitations can be made.

All Occurrences	All model types are searched.
Family Models	Only Family Models are searched.
Feature Models	Only Feature Models are searched.

Element Scope

This group is only activated if Elements search type is selected. Here it can be configured against which element name the search pattern is matched.

Unique Name	Match against the unique name of the element.
Visible Name	Match against the visible name of the element.

At least one of the options has to be chosen.

Attribute Scope

This group is only activated if Attribute Values search type is selected. In this group the following refinements can be made.

Calculations	Match against attribute value calculations.
Constants	Match against constant attribute values.

At least one has to be selected. To limit the search to values of attributes with a specific name, this name can be inserted into the Attribute Name input field.

Scope

This group is used to limit the search to a certain set of models. The following options are available.

Workspace	Search in all variant projects of the workspace.
Selected resources	Search only in the projects, folders, and files that are selected in the Variant Projects view.
Enclosing projects	Search only in the enclosing projects of selected project entries in the Variant Projects view.
Working set	Search only in projects included in the chosen working set.

For more information about working sets, please consult the [Workbench User Guide](#) provided with Eclipse (*Help->Help Contents*, section "Concepts"->"Workbench"->"Working sets").

Search Results

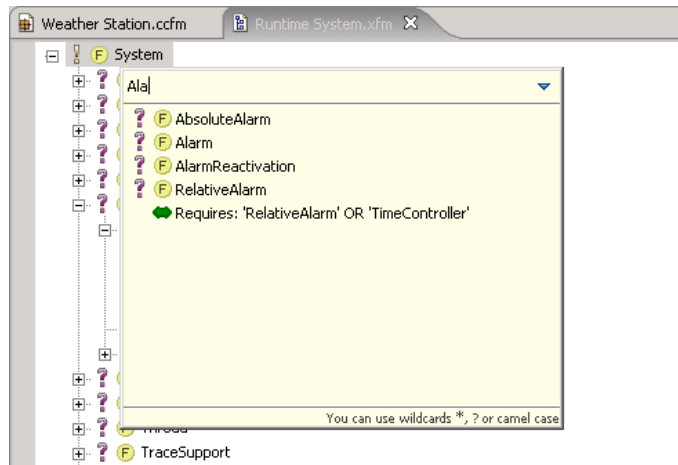
The results of the search are listed in the Variant Search view supporting a tree and table representation and a search result history. For more information about the Variant Search view see [Section 7.4.3, "Search View"](#).

After the search is finished blue markers are created on the right side of models containing matches. These markers visualize the matches in the model and provide an easy way to navigate to the matched model items simply by clicking on a marker.

6.7.2. Quick Overview

Within a model editor it is possible to search using the Quick Overview. Especially in large models it is sometimes hard to find an element with a known name or a known part of the name. To shorten the navigation through tree nodes or tables in model editors pure::variants provides a quick overview which you may already know from Eclipse as *Quick Outline*. If a model editor (e.g. a Feature Model Editor) is active then pressing the shortcut **CTRL+O** opens a small window with a sorted and filtered list of all model elements. [Figure 6.26, “Quick Overview in a Feature Model”](#) shows an example for the quick overview.

Figure 6.26. Quick Overview in a Feature Model



After the quick overview popped up a filter text can be entered. Shortly after the modification of the filter text the list of the quick overview will be updated according to the given filter. The filter can contain wildcards like the question mark `?` and the asterisk `*` as place holders for one arbitrary character and an arbitrary sequence of characters, respectively. You may also use CamelCase notation. Camel case means that between each capital letter and the letter in front of it a `*` wildcard is placed internally to the filter text. For example, typing *ProS* as filter text would also find elements like *Protocol Statistics* or *Project Settings*.

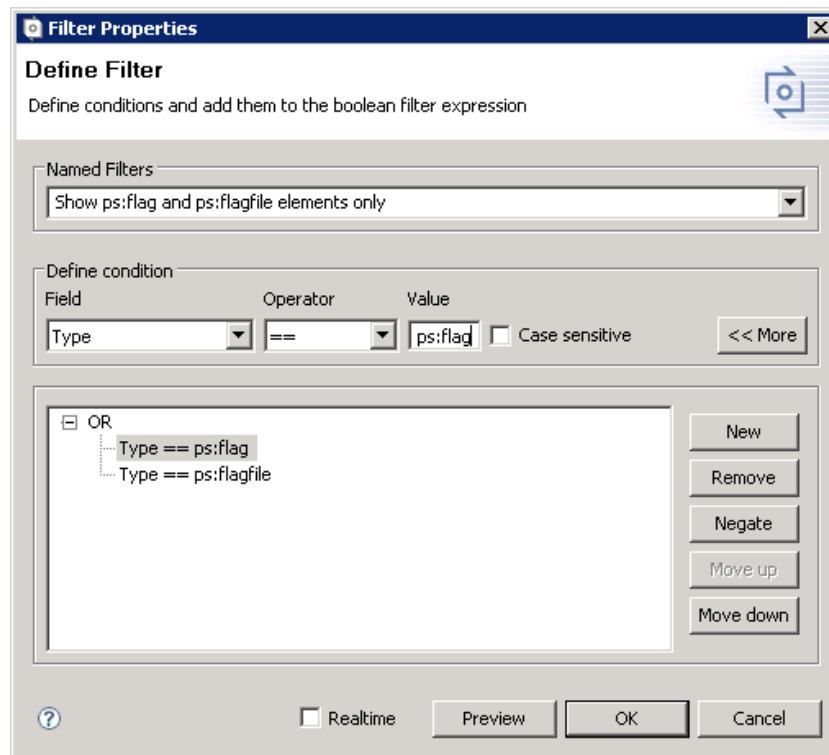
Finally, if the desired element is shown in the quick overview then a double-click on it lets the editor navigate to that element. You can also use the arrow keys to select the item from the list and press **ENTER** to get the same effect.

Note

The quick overview presents only those model objects which the active model editor shows. For instance, if the editor shows relations then the quick overview presents them, too. Additionally the filter set to the editor has effect to visibility of elements in the quick overview.

6.8. Filtering Models

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 6.27. Filter definition dialog

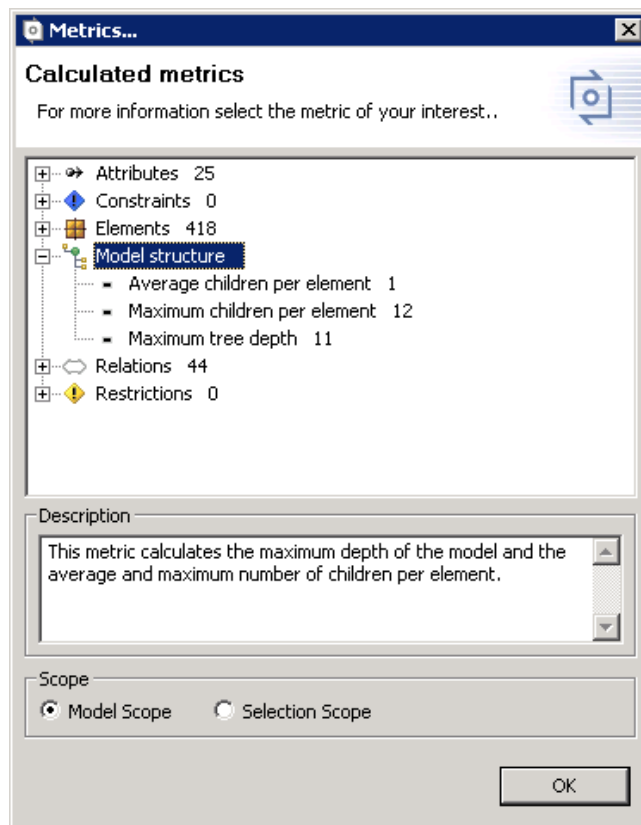
Arbitrarily complex filters based on comparison operations between feature/element properties (name, attribute values, etc.) and logical expressions (and/or/not) are supported. Comparison operations include conditions like equality and containment, regular expressions (matches) and checks for the existence of an attribute for a given element (empty/not empty). See [Section 9.11, “Regular Expressions”](#) for more information on regular expressions.

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 Visualization view, which can be activated using the Windows->Views->Other->Variant Management->Visualization item. See [Section 7.4.2, “Visualization View”](#) for more information on the view.

6.9. Computing Model Metrics

All pure::variants model editors provide an extensible set of metrics for the opened models. These metrics can be displayed by choosing **Show Metrics** from the context menu of a model editor. If metrics shall be displayed only for a sub-tree of a model, the root of this sub-tree has to be selected before the context menu is opened.

Figure 6.28. Metrics for a model



The available metrics are listed in a tree showing the name and overall results of the metrics on top level. Partial results and detailed information provided by a metric are listed in the corresponding subtree. An explaining description of a metric is displayed in the **Description** field if the name of the metric is marked.

The radio buttons at the bottom of the metrics dialog are used to switch between whole model and selected elements metric calculation. For VDMs, metrics are always calculated for the whole model. If a VDM has not been evaluated yet, the calculated metrics may be outdated and can show incorrect values.

On the **Variant Management->Metrics** preferences page (menu **Window->Preferences**), the set of metrics to apply can be configured.

6.10. Extending the Type Model

For every project a Type Model can be created extending the global Type Model. This model belongs to the project and can be shared like any other pure::variants model. This is an easy and a straight forward way to contribute own types to be used in the Feature and Family Models of the project containing the Type Model.

To create a Type Model right-click on a project in the Variant Project View and choose *New->Type Model* from the context menu. This creates a new file in the project named like the project and with extension ".typemodel". Note that only one Type Model can be created per project. The new Type Model is opened in the Type Model Editor. This editor also is opened by double-clicking on an existing Type Model file (see [Figure 6.29, "Type Model Editor Example"](#)).

The Type Model Editor consists of two parts. The left part shows the list of types defined in the model, while the right part provides an editing area for the type selected on the left. Additionally the left part provides a context menu for adding and removing types of the type model.

The Type Model Editor allows to add element and attribute types. After adding an attribute type the right part allows to change the *Name*, *Base Type* (that is the type which this type is specializing), whether this type is *Abstract*

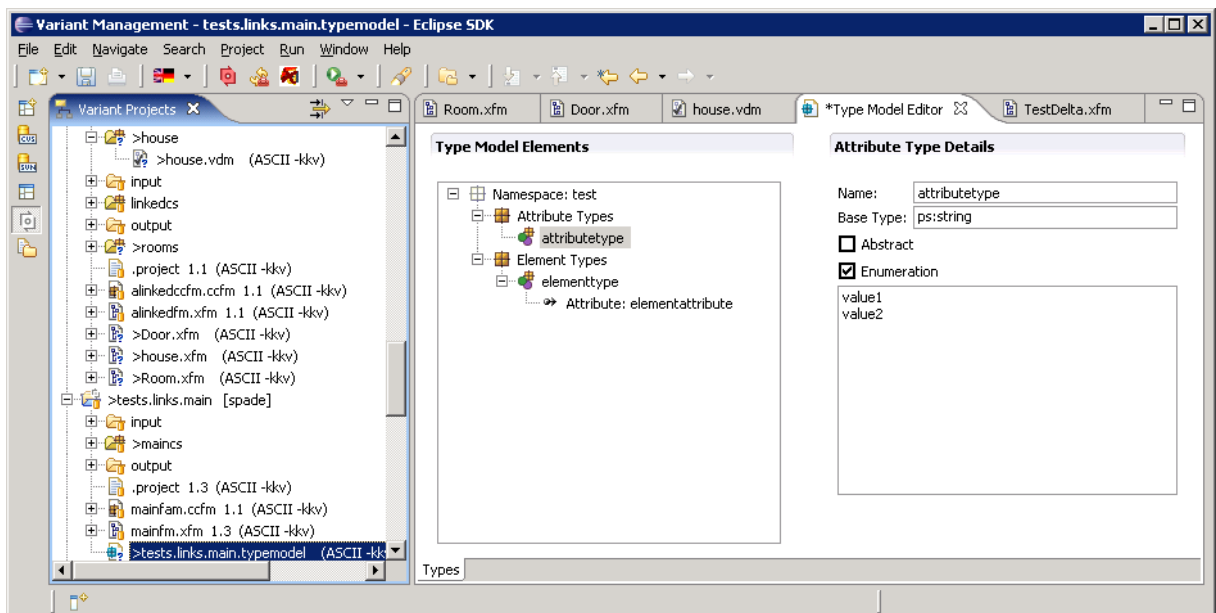
(and thus can only be used as base type for other types), and whether this is an enumeration type only allowing one of the listed values.

The editor provides for element types to change the *Label*, *Name* and the *Base Type*. Additionally the element type may be set *Abstract* and if there shall be a generic New Wizard, which would allow to easily create an element of that type.

For an element type attributes can be created. Those attributes present the default attributes which are defined for a concrete element of that type. For each attribute a *Name*, a *Type*, whether it is a *Single Value*, *List* or *Set* can be specified. Following flags can be set for an attribute: *Optional* (whether this attribute is required for an element), *Fixed* (whether it has a constant value or can be overridden in a VDM), *Read Only* (whether the user can provide a value for it) and *Invisible* (whether it is visible to the user).

After a Type Model was created or changed, the types defined in the Type Model are immediately available for modeling in the corresponding project.

Figure 6.29. Type Model Editor Example

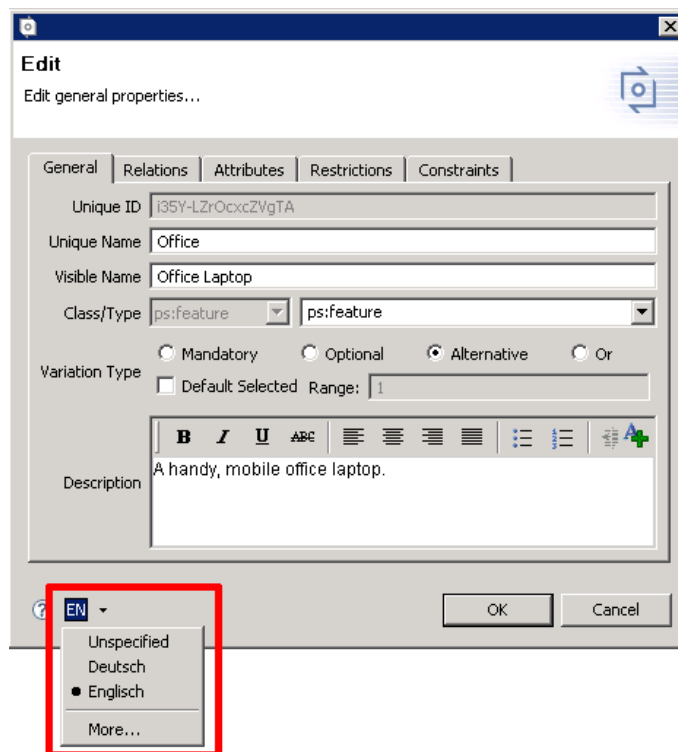


6.11. Using Multiple Languages in Models

pure::variants is able to deal with multiple languages for the visible name of elements and for all descriptions. This allows to define Feature and Family Models in more than one language.

The default language for models is defined in the preferences on the visualization page. Select *Window->Preferences...* from menu and then *Variant Management->Visualization* to change it. The default language is used for all views and editors.

To edit visible names or descriptions for a particular language use the language button (DE) in the element properties dialog as in Figure 6.30, “Language selection in the element properties dialog”. Clicking on the arrow of that button shows a list of languages currently in use in the model. By selecting a language from that list the visible name and all descriptions in the element properties dialog are shown in that language. You can change them, switch to another language and then change them again. pure::variants saves the visible name and all descriptions for each chosen language. If the desired language is not present in the language list then select the *More...* item to choose the language in the upcoming dialog. The selected language will be added to the language list.

Figure 6.30. Language selection in the element properties dialog

Note

There is a language with name *Unspecified* and abbreviation ?? available. This *language* can be used like others. Typically, it is used when the language of visible names and descriptions do not play a role. After installation of pure::variants it is set as the first default language. All texts of old models are treated as if they were entered for the language *Unspecified*.

The visible name and the description fields sometimes show texts from another language than the active, usually with an annotation like *[Language: EN]*. This occurs when no visible name or description was entered for the active language, to point out that there is a text for another language (in the example EN stands for English). However, simply modify the text to specify a text for the active language. Or, you may replace it by its translation.

Multiple languages of visible names and descriptions are also supported in the properties view (see [Section 7.4.6, “Properties View”](#)) and in the model properties page as well as in the general properties page of a model (see [Section 7.5.1, “Common Properties Page”](#) and [Section 7.5.2, “General Properties Page”](#)). Look for the language button and use it like described above.

6.12. Importing and Exporting Models

6.12.1. Exporting Models

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** from category **Variant Management** 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, character separated values, export format results in a text file that can be opened with most spreadsheet programs (e.g. Microsoft Excel or OpenOffice). CSV export respects the filters set in the editor of the model to export, i.e. only the matching elements are exported. The export wizard permits the columns to be generated in the output file to be selected.

HTML Export

The HTML Export generates representations for feature and family models in HTML. The generated HTML file can be opened by any browser (e.g. "Internet Explorer", "Firefox", etc.).

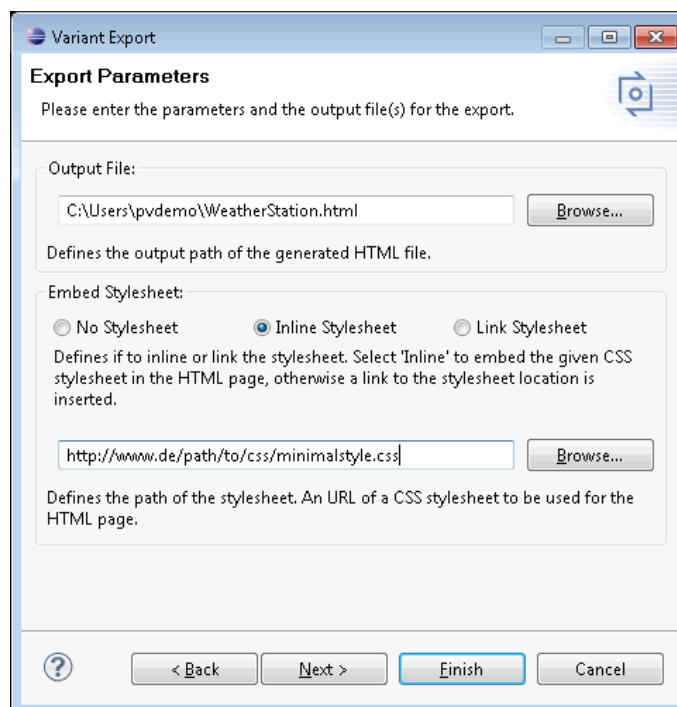
The export will generate a navigation section which represents all model elements hierarchical in a tree and the data of the elements on the right side of the generated html page. The navigation tree will help to navigate to elements quickly. The selected element in the navigation section will be shown on top of the content section. Each section of an element includes the following paragraphs:

- *General Properties*
- *Description*
- *Properties*
- *Relations, Restrictions and Constraints*

The *General Properties* paragraph shows information like *Unique Name*, *Element Class*, *Variation Type*, *Element Type* and *Default Selected*.

The following two pictures are showing the HTML Export wizard. The first page enables the user to define an absolute path for the output file. Using pure:variants path variables is supported. The style of the html output can be adjusted individually by referencing your own stylesheet (*.css) either as web URL or local file. The stylesheet can either be linked or inlined in the html output file.

Figure 6.31. HTML Export Wizard

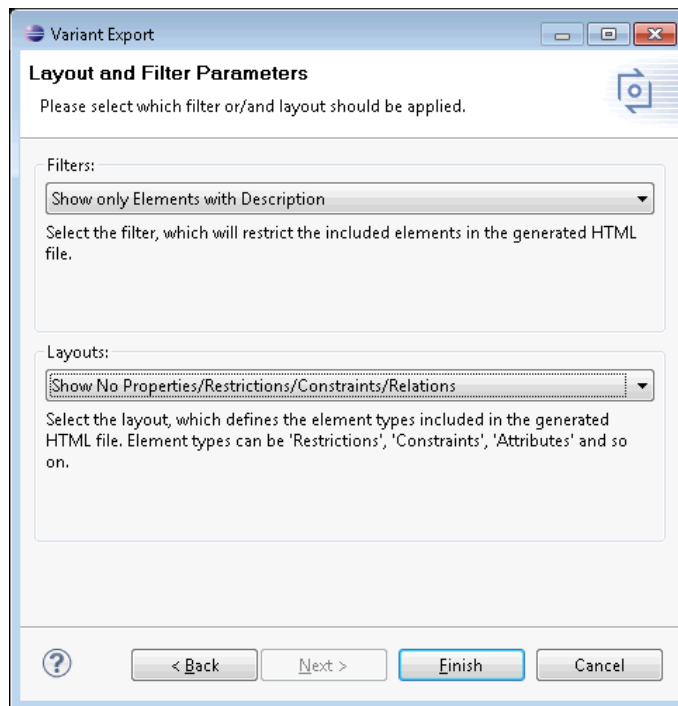


Define output path and css file path.

On the second configuration page a filter can be selected, which applies to the selected model. Elements which apply to the filter are not included in html output. Please see [Section 6.8, "Filtering Models"](#) for further instructions.

To hide specific information (e.g. "Restrictions", "Specific Attributes",...) in the selected model a tree layout can be selected in the combo box *Layouts*. For further Information see [the section called "Tree Editing Page"](#).

Figure 6.32. HTML Export Wizard



Define filter and tree layout.

The following stylesheet classes are supported in the HTML Export.

Table 6.3. Table of CSS classes

CSS Class	Description
.section	All sections including "General Properties", "Description", "Properties" and ...
.ps-general	"General Properties" section placed beneath Feature headline
.ps-description	"Description" section placed beneath "General Properties"
.ps-properties	"Properties" section placed beneath "Description"
.ps-relations	"Relations, Restrictions, Constraints" placed beneath "Properties"
.ps-breadcrumb	Breadcrumb navigation path beneath Feature's headline
.ps-feature	Section of a Feature

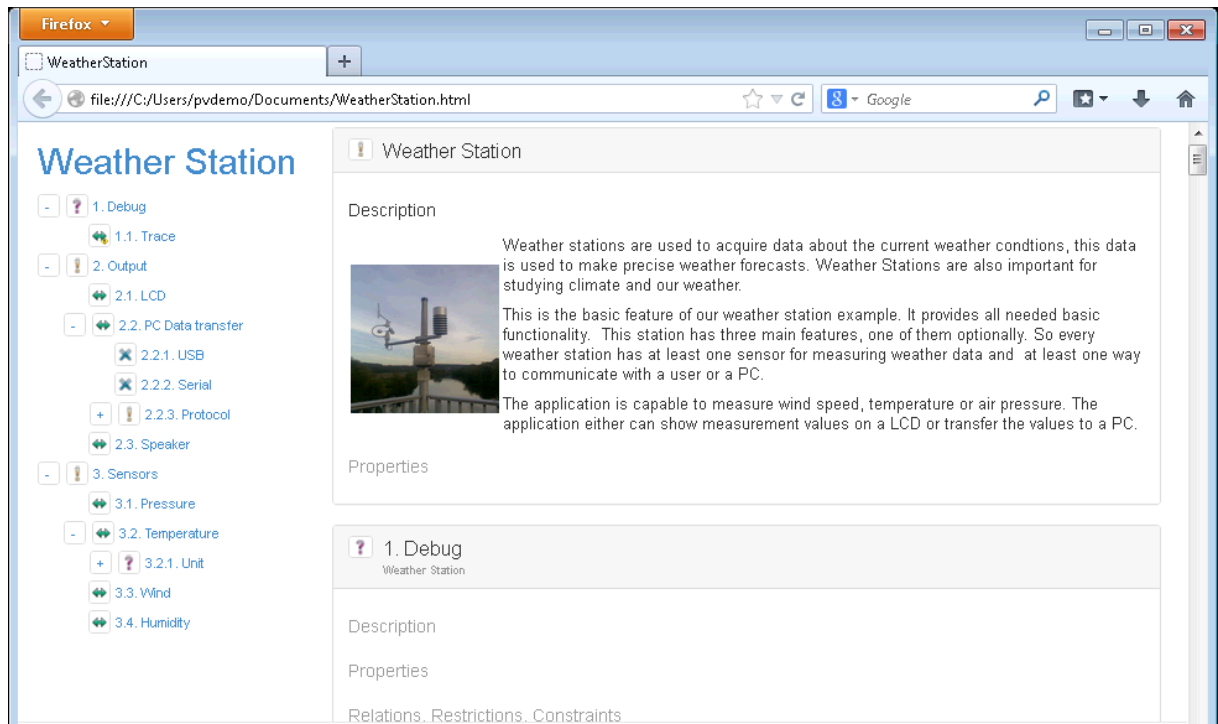
Is the html output opened in a browser the following interactions are available:

- Breadcrumb navigation placed beneath each element headline to navigate quickly to the parents of the element.
- Expand/Collapse tree buttons on the bottom of the navigation on the left side of the website to expand/collapse the navigation tree.
- Expand/Collapse model buttons on the right bottom of the website to expand/collapse all element sections.
- Expand/Collapse buttons on any element sections and headline to expand/collapse all element sections and headline of the same type in the whole html document.

- Elements having a relation have a hyperlink to quickly navigate to the related elements.

The following image shows a typical html export.

Figure 6.33. HTML Export Result



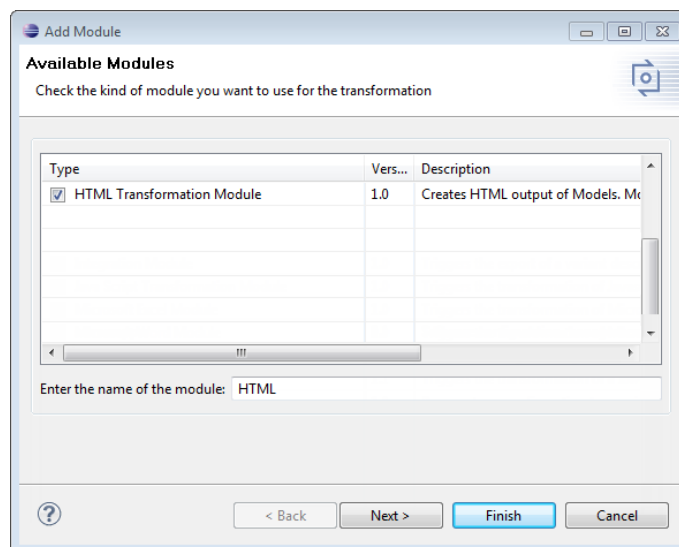
HTML Export example.

It is not possible to export a Variant Description Model using the export wizard as described above. For exporting a vdm a transformation module is used. The transformation is described in the next section.

HTML Transformation Module

For exporting a vdm to a html document the transformation module *HTML Transformation Module* is used. See below the module in the transformation module selection dialog.

Figure 6.34. HTML Transformation Module



Selection of HTML Transformation Module

The next image shows the parameter of this transformation module.

The parameter *Output* enables the user to define a different output folder, for the result of the HTML transformation.

The transformation module for HTML has three different modi, called **Result Models Tailored**, **Result Models Annotated** and **Input Models Only**. The modus is selected with the parameter *Mode*

The **Result Models Tailored** mode executes a transformation of on variant description model and will output the transformed feature and family models as html representation. Each model will generate a single html output file. The name of this file will be the name of the model suffixed with the model type. In this mode only elements part of the variant will get exported to the html.

The **Result Models Annotated** mode exports all elements defined in the input models, but it will gray out all the elements, which are not part of the transformed variant..

The **Input Models Only** mode doesn't execute a transformation but exports all input models defined in the used configuration space. Furthermore are all configuration parameters definable except the filter parameter.

Third parameter *Layout* is optional. If used it defines a tree layout, which will be used during the transformation. (the section called “Tree Editing Page”)

Fourth parameter *Stylesheet* defines wether **No Stylesheet** is used or if a **Link Stylesheet** is used, or if a **Inline stylesheet** is used.

Parameter *Stylesheet Path* is optional, but needed if **Link Stylesheet** or **Inline Stylesheet** was selected. It defines the path to the local css file or a URL to a remote css file.

The last two optional parameter allow the user to filter the input models of the configuration space. The *Model Type Filter* allows the user to filter the input models reparding their type. Additionally the paramter *Model Name Filter* allows the user to specify a regular expression, which is used to filter the models by their names.

Figure 6.35. HTML Tranformation Module Parameters

Module Parameters
Enter values for the parameters of the module

Include:

Exclude:

Additional Parameters:

Name	Type	Value
Output	ps:path	
Mode	ps:string	Result Models Tailored
Layout	ps:string	
Stylesheet	ps:string	No Stylesheet
Stylesheet Path	ps:path	
Model Type Filter	ps:string	Both
Model Name Filter	ps:string	

Buttons: Add, Remove, < Back, Next >, Finish, Cancel

Configuration of HTML Transformation Module.

Directed Graph Export

The directed graph export format generates a model graphs in the DOT language and with appropriate tools installed also images in many other image format such as JPEG, PNG, BMP. 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->Image Export), many image formats can be generated directly. The dialog shown in Figure 6.36, “Directed graph export example” 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 6.37, “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 6.36. Directed graph export example

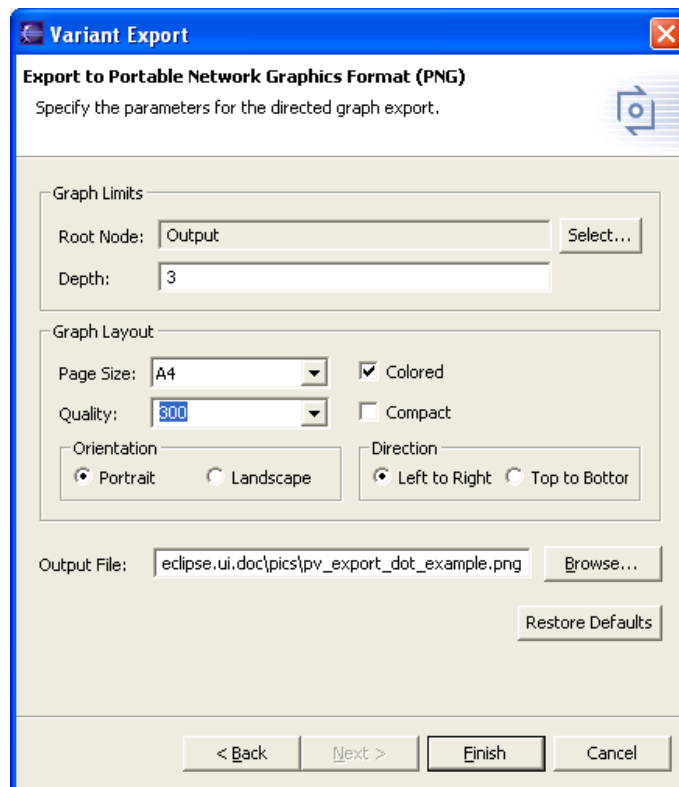
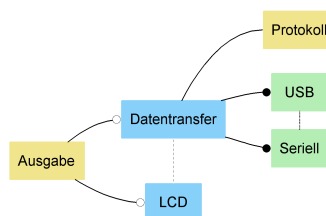


Figure 6.37. Directed graph export example (options LR direction, Colored)



6.12.2. Importing Models

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

Currently two generic imports are provided. The first one imports a Family Model from source directories. 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.

The second generic import, imports a Feature Model or a Family Model from a CSV file. While importing a few fields are directly used by pure::variants to build the model. Other fields are imported as attributes to the elements. These fields are:

Table 6.4.

Unique Name	Unique name of an element.
Unique ID	Unique Id of an element
Visible Name	Visible name of an element.
Variation Type	The variation type of an element. Possible values are: ps:mandatory, ps:optional, ps:or and ps:alternative. If no variation type is given ps:mandatory is used.
Parent Unique ID	The Unique ID of the parent element.
Parent Unique Name	The Unique Name of the parent element.
Parent Visible Name	The Visible Name of the parent element.
Parent Type	The Type of the parent element.
Class	The class of an element, most likely ps:feature for Feature Model or ps:component for Family Model.
Type	The type of an element, most likely ps:feature for Feature Model or ps:component for Family Model.

For importing a CSV to a Feature Model the field **Unique Name** is necessary. If you like to import a hierarchical model either the fields **Unique ID** and **Parent Unique ID** or **Unique Name** and **Parent Unique Name** are necessary as well. In case of importing an hierarchical model the element without Parent Unique ID will be the root element, if no Parent Unique IDs given, the first element without will be the model root.

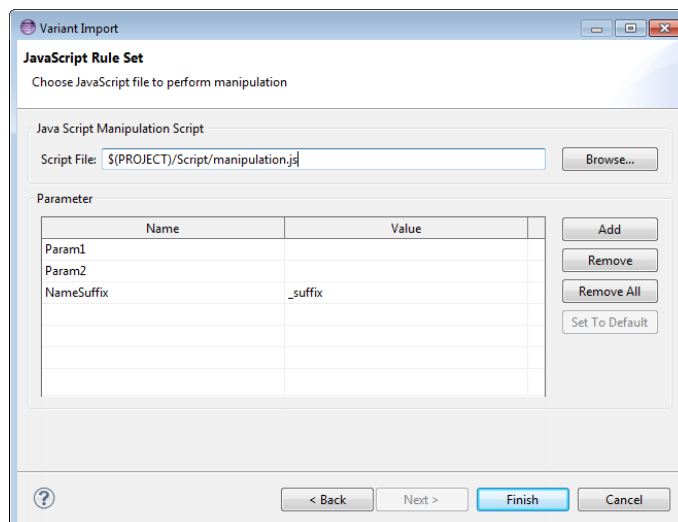
Please note, the CSV export of pure::variants exports more fields as the CSV import of pure::variants can import. Fields such as **Relations**, **Restriction** and **Constraint** are ignored by CSV import. Therefore a full roundtrip with the help of the CSV data format is not possible.

Additional imports may be available if different plug-ins are installed.

User-defined import manipulator with JavaScript

For customization of an imported pure::variants model a *JavaScript Manipulator* is provided. This manipulator is available for all importer, which support import manipulators.

Figure 6.38. JavaScript Manipulator Wizard Page



On the JavaScript Manipulator wizard page a JavaScript file needs to be given, which is performed after the import is done, to customize the resulting pure::variants model. It is allowed to use pure::variants path variables in the JavaScript path.

Additionally Parameter for the JavaScript can be defined on this page. Parameters are simple name value pairs. The JavaScript can also define parameter and default values in a comment at the top of the script. These parameters are automatically added to the parameters table, if the script is loaded.

Note

An example JavaScript is generated using the "New -> JavaScript Manipulation Script" entry from the context menu in the projects view. This script shows a basic model manipulation and how parameters are defined in a JavaScript.

6.13. External Build Support (Ant Tasks)

pure::variants provides some useful Ant task. They can be used with buildfiles inside Eclipse or in headless mode.

6.13.1. pv.import

The `pv.import` task imports a pure::variants project into the workspace. If the project is already part of the workspace nothing happens.

Example:

```
<pv.import path="C:\Projects\Weather Station"/>
<pv.import server="http://pv.server.com" name="Weather Station" revision="v2"/>
```

This task has the following attributes:

- **importreferences** if `true` the references to other projects are also imported (default is `true`)
- **path** is the absolute path to the project in the file system
- **server**, **name**, **revision** are the server URL, the name of the project, and optionally the version of a remote project to import

6.13.2. pv.evaluate

The `pv.evaluate` task performs an evaluation and stores the result in the given result model file.

Example:

```
<pv.evaluate vdm="Weather Station\Config\Indoor.vdm" vrm="Weather Station\Indoor.vrm"/>
<pv.evaluate vdm="Weather Station\Config\Outdoor.vdm" vrm="Weather Station\Outdoor.vrm">
  <property name="autoresolve" value="extended"/>
  <property name="timeout" value="120"/>
</pv.evaluate>
```

This task has the following attributes:

- **vdm** is the path to the Variant Description Model to evaluate
- **vrm** is the path to the Variant Result Model

The `pv.evaluate` task supports optional properties which influence the evaluation:

- **autoresolve** set the mode of the auto resolver. Possible values are `off`, `simple`, `extended`
- **timeout** set the maximal time used for the evaluation in seconds

6.13.3. pv.transform

The `pv.transform` task performs a transformation of a Variant Description Model or Variant Result Model.

Example:

```
<pv.transform vdm="Weather Station\Config\Indoor.vdm" name="Default" force="true">
  <property name="autoresolve" value="extended"/>
  <property name="timeout" value="120"/>
</pv.transform>
<pv.transform vrm="Weather Station\Outdoor.vrm" name="Default"/>
```

This task has the following attributes:

- **vdm** is the Variant Description Model to transform
- **vrm** is the Variant Result Model to transform
- **name** is the name of the Transformation Configuration
- **force** if `true` the transformation runs always also if the result has errors

The `pv.transform` task supports optional properties which influence the evaluation, which runs before the transformation:

- **autoresolve** set the mode of the auto resolver. Possible values are `off`, `simple`, `extended`
- **timeout** set the maximal time used for the evaluation in seconds

6.13.4. pv.inherit

The `pv.inherit` task changes the inheritance between VDMs.

Example:

```
<pv.inherit vdm="Weather Station\Config\Indoor.vdm">
  <super vdm="Weather Station\Config\Base.vdm"/>
</pv.inherit>
```

This task has the following attributes:

- **vdm** is the Variant Description Model which inherits (`pv.inherit` tag), or which is inherited (`super` tag)

6.13.5. pv.connect

The `pv.connect` task connects to a server and login as given user.

Example:

```
<pv.connect server="http://pv.server.com" user="example" pass="example"/>
```

This task has the following attributes:

- **server** is the pure::variants server to connect to
- **user** is the name of the user
- **pass** is the password for the user

6.13.6. pv.sync

The `pv.sync` task updates a model imported by a connector. The connector specific synchronization job is called to update the models data.

Example:

```
<pv.sync model="Weather Station\Sources.ccfm" />
```

This task has the following attributes:

- **model** is the model to update

6.13.7. pv.mergeselection

The `pv.mergeselection` task creates or updates a variant description model by merging all selections from the given variant description models. The following rules are applied. If an element is excluded in at least one source model the element is also excluded in the result. If an element is selected in at least one source model it is also selected in the result if not excluded by any other source model.

Example:

```
<pv.mergeselection vdm="Weather Station\Config\Merged.vdm">  
  <source vdm="Weather Station\Config\IndoorBase.vdm">  
  <source vdm="Weather Station\Config\TempOnly.vdm">  
  <source vdm="Weather Station\Config\CommUSB.vdm">  
</pv.mergeselection>
```

This task has the following attributes:

- **vdm** is the result model (pv.mergeselection tag) or the source model (source tag)

Chapter 7. Graphical User Interface

The layout and usage of the pure::variants User Interface closely follows Eclipse guidelines. See the [Workbench User Guide](#) provided with Eclipse (*Help->Help Contents*) for more information on this.

7.1. Getting Started with Eclipse

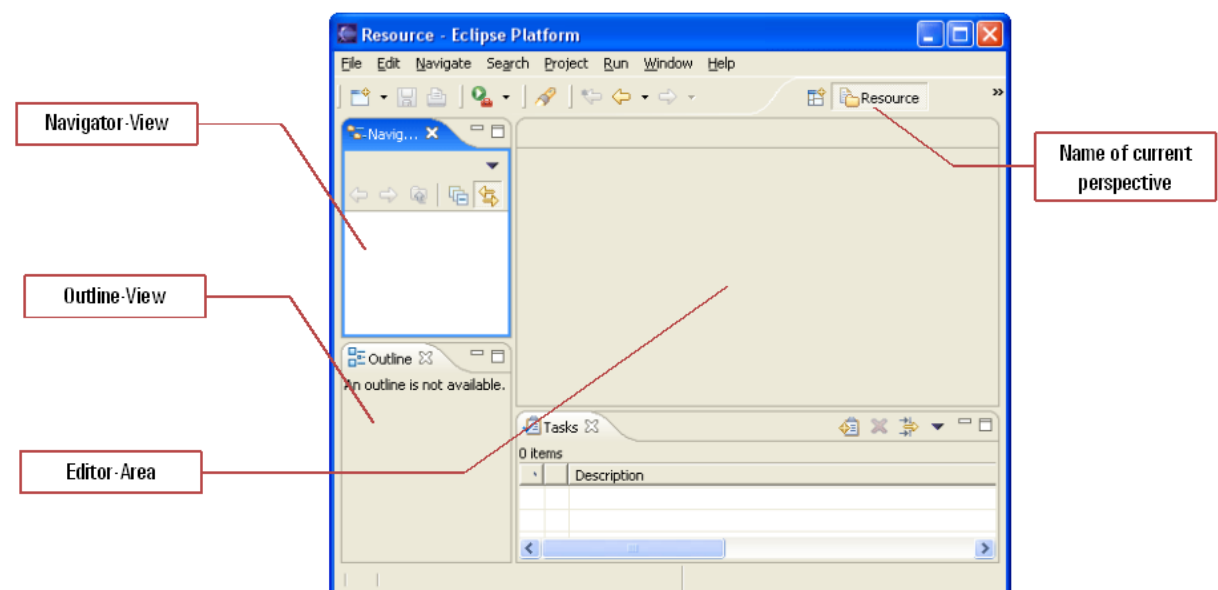
This section gives a short introduction to the elements of the Eclipse UI before introducing the pure::variants UI. Readers with Eclipse experience may skip this section.

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 7.1, “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 reopened 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 *Navigators* 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 7.1. Eclipse workbench elements

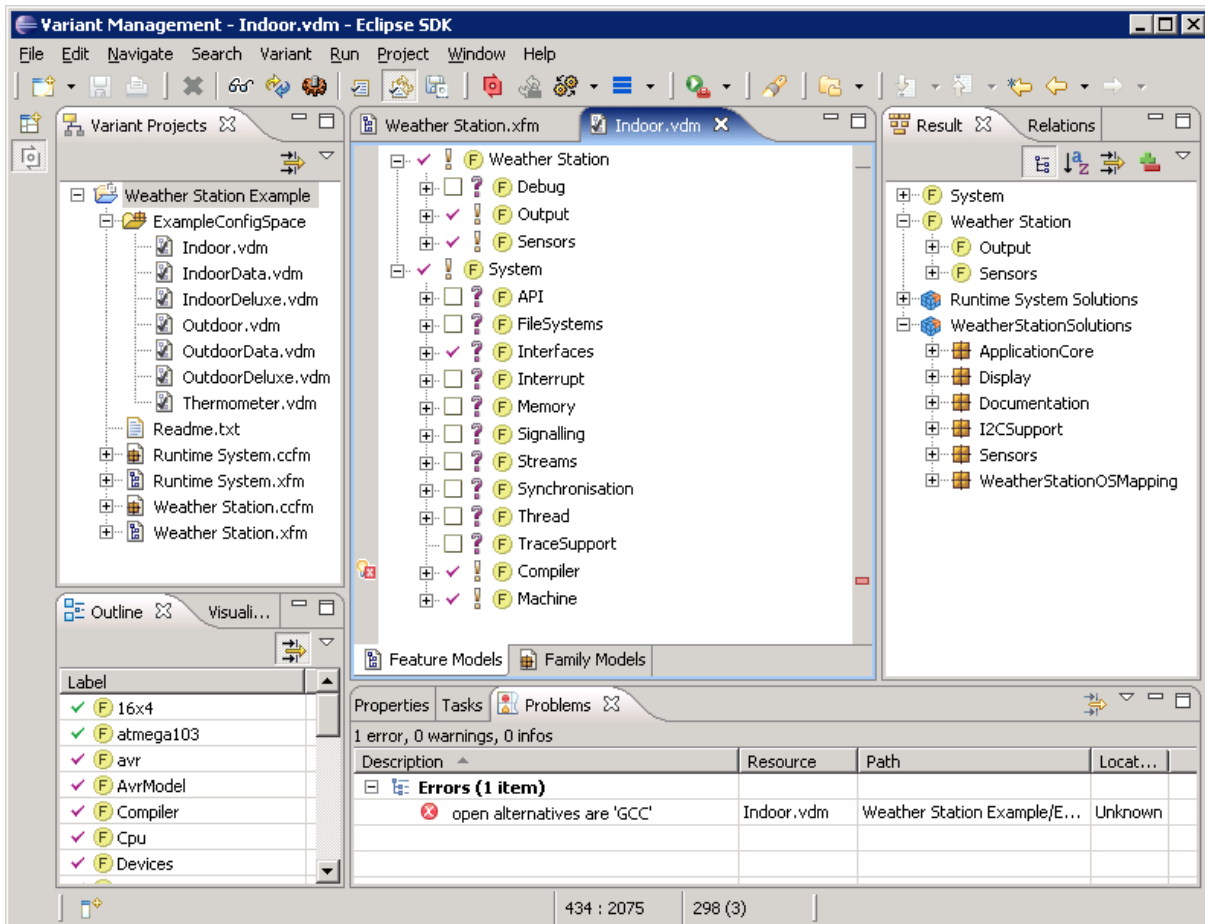


Eclipse uses *Views* to represent any kind of information. Despite their name, data in some types of view may be changed. Only one instance of a specific type of view, such as the Outline view, may be shown in the workbench at a time. All available views are accessible via **Windows->Show View->Other**.

7.2. 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 7.2, “Variant management perspective standard layout”](#) shows this perspective with a sample project.

Figure 7.2. Variant management perspective standard layout



7.3. 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.

7.3.1. 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 (see [Section 9.12, “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. The level of details shown in the tree can be changed in the "Tree Layout" sub-menu of the context menu. If an attribute is selected in the tree and the context menu is opened, this sub-menu contains the special entry "Hide Attribute: name" is shown. It is used to hide this attribute in the tree view. Hidden attributes can be made visible again with the sub-menu action *Table Layout->Change*. A dialog is opened which presents a list of all visible attributes and all invisible attributes. This list can be adapted as desired. Additionally the tree layout allows to generally show or hide "Restrictions", "Constraints", "Relations", "Attributes" and "Inherited Attributes". If attributes are set as hidden, the tables mentioned above have no effect. In addition the layouts can be given a name to store them permanently in the eclipse workspace. A named layout can be set as default layout, which can apply for only one tree layout, which then always is used for any newly opened model (see [Section 7.4.2, "Visualization View"](#) for more information on it).

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

The labels of the elements shown in the tree can be customized on the *Variant Management->Visualization* preference page.

Table Editing 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). A layout can be given a name. Named layouts are shown in, and can be restored from, the Visualization view (see [Section 7.4.2, "Visualization View"](#)). Named layouts and layout changes for each table are stored permanently in the Eclipse workspace. As for tree layouts a table layout can be set as default. Clicking on a column header sorts that column. The sort direction may be reversed with a second click on the same column header.

Tip

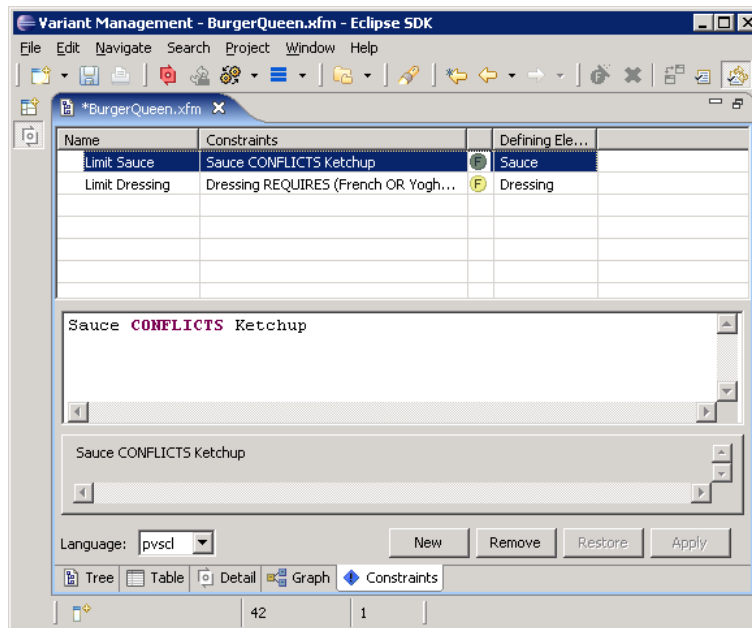
Double clicking on a column header separator adjusts the column width to match the maximal width required to completely show all cells of that column.

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.

Constraints Editing Page

The Constraints page is available in the Feature and Family Model Editor and shows all constraints in the current model. Constraints can be edited or new created on this page. It also supports to change the element defining a constraint.

[Figure 7.3, "Constraints view"](#) shows the Constraints page containing two constraints formulated in *pvSCL*. The first column in the table of the page contains the name of the constraint. The constraint expression is shown in the second column. In column three the type of the element defining the constraint is shown. The defining element itself is shown in the last column.

Figure 7.3. Constraints view

New constraints can be added by pressing button "New". The name of a constraint can be changed by double-clicking into the name field of the constraint and entering the new name in the opened cell editor. Double-clicking into the "Defining Element" column of a constraint opens an element selection dialog allowing the user to change the defining element.

Clicking on a constraint shows the constraint expression in the editor in the bottom half of the page. The kind of editor depends on the language in which the constraint is formulated (see [the section called "Advanced Expression Editor"](#) for more information about the editor). The language for the constraint expression can be changed by choosing a different language from the "Language" list button.

Changes to constraints are applied using the "Apply" button and discarded using the "Restore" button.

Graph Visualization Page

The graph visualization page is primarily intended for the graphical representation and printing of models. Although the usual model editing operations like copy, cut, and paste and the addition, editing, and deletion of model elements also are supported.

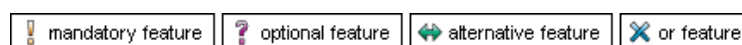
Note

The graph visualization is only available if the Graphical Editing Framework (GEF) is installed in the Eclipse running pure::variants. More information about GEF are available on the [GEF Home Page](#).

For nearly all actions on a graph that are explained in the next sections keyboard shortcuts are available listed in [Section 9.12, "Keyboard Shortcuts"](#).

Graph Elements

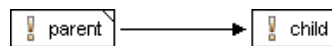
Model elements are represented in the graph as boxes containing the name of the element and an associated icon. Feature model elements are represented as shown in the next figure.



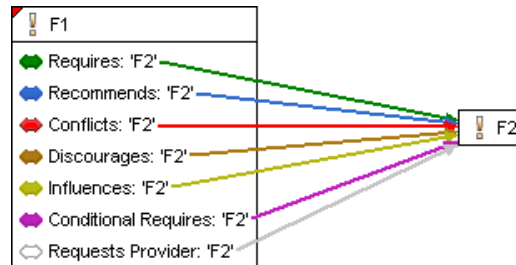
The representation of Family Model elements slightly differs for part and source elements.



Parent-child relations are visualized by arrows between the parent and child elements.



Other relations are visualized using colored connection lines between the related elements. The color of the connection line depends on the relation and matches the color that is used for this relation on the tree editing page.



If an element has children a triangle is shown in the upper right-hand corner of the element box. Depending on whether the element is collapsed or expanded a red or white corner is shown.



Graph Layout

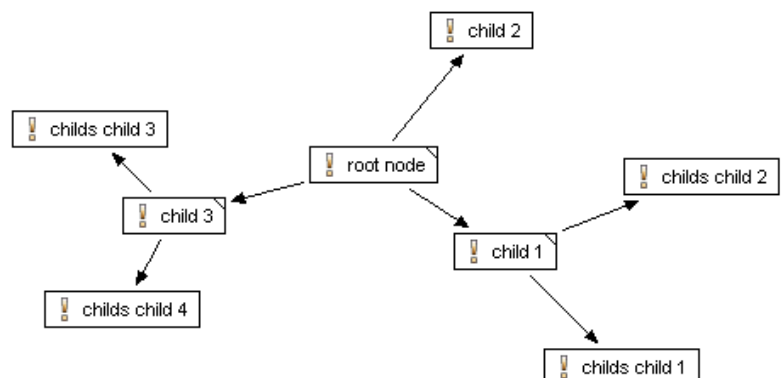
The layout of the graph can be changed in several ways. Graph elements can be moved, expanded, collapsed, hidden, and automatically aligned. The graph can be zoomed and the layout of the connections between the elements of the graph can be changed.

Two automatic graph layouts are supported, i.e. horizontal aligned and vertical aligned. Choosing "Layout Horizontal" from the context menu of the graph visualization page automatically layouts the elements of the graph from left to right. The elements are layouted from top to bottom choosing "Layout Vertical" from the context menu.

Depending on the complexity of a graph the default positioning of the connection lines between the elements of the graph may not be optimal, e.g. the lines overlap or elements are covered by lines. This may be changed by choosing one of three available docking rules for connection lines from the submenu "Select Node Orientation" of the context menu.

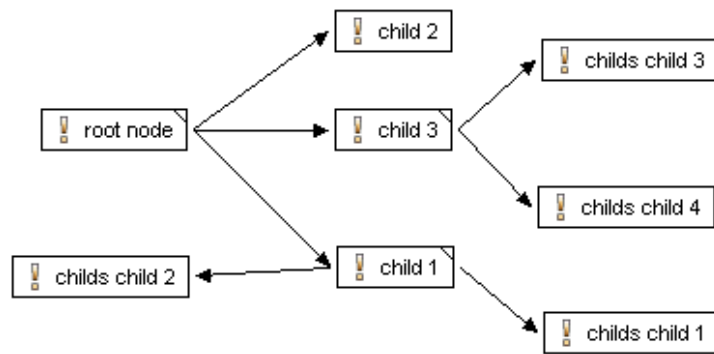
No Docking Rule

The connection lines point to the center of connected elements. Thus connection lines can appear everywhere around an element.



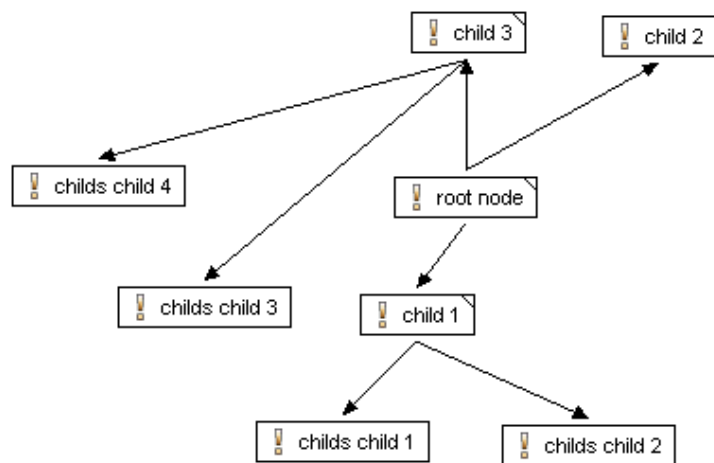
Dock Connections on Left or Right

The connection lines are positioned in the middle of the left or right side of connected elements. This is especially useful for horizontally layouted graphs.



Dock Connections on Top or Bottom

The connection lines are positioned in the middle of the top or bottom side of connected elements. This is especially useful for vertically layouted graphs.



The graph can be zoomed using the "Zoom In" and "Zoom Out" items of the context menu of the graph visualization page.

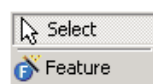
Several elements can be selected by holding down the **SHIFT** or **STRG** key while selecting further elements, or by clicking somewhere in the empty space of the graph visualization page and dragging the mouse over elements. A dashed line appears and all elements that are partially or wholly enclosed in it will be selected.

If an element has children the element can be expanded or collapsed by clicking on the triangle in the upper right-hand corner of the element's box. Another way is to use the "Collapse Element", "Expand Element", and "Expand Subtree" context menu items. In contrast to the "Expand Element" action, "Expand Subtree" expands the whole subtree of an element, not only the direct children.

To hide an element in the graph this element has to be selected and "Hide Element" has to be chosen from the context menu. Attributes, relations, and the connection lines between related elements (relations arrows) also can be hidden by choosing one of the items in the "Show In Graph" submenu of the context menu.

Elements can be moved by clicking on an element and move the mouse while keeping the mouse button pressed. This only works if the element selection tool in the tool bar is selected.

Figure 7.4. Selected Element Selection Tool



Graph Editing

Basic editing operations are available for the graph. The elements shown in the graph can be edited by choosing "Properties" from the context menu of an element. Elements can be copied, cut, pasted, and deleted using the corresponding context menu items.

New elements can be created either by choosing one of the items below the "New" context menu entry or by using the element creation tool provided in the tool bar of the graph visualization page.

Figure 7.5. Feature/Family Model Element Creation Tools



Graph Printing

Printing of a graph is performed by choosing the *File->Print* menu item. The graph is printed in the current layout.

Note

Printing is only available on Windows operating systems.

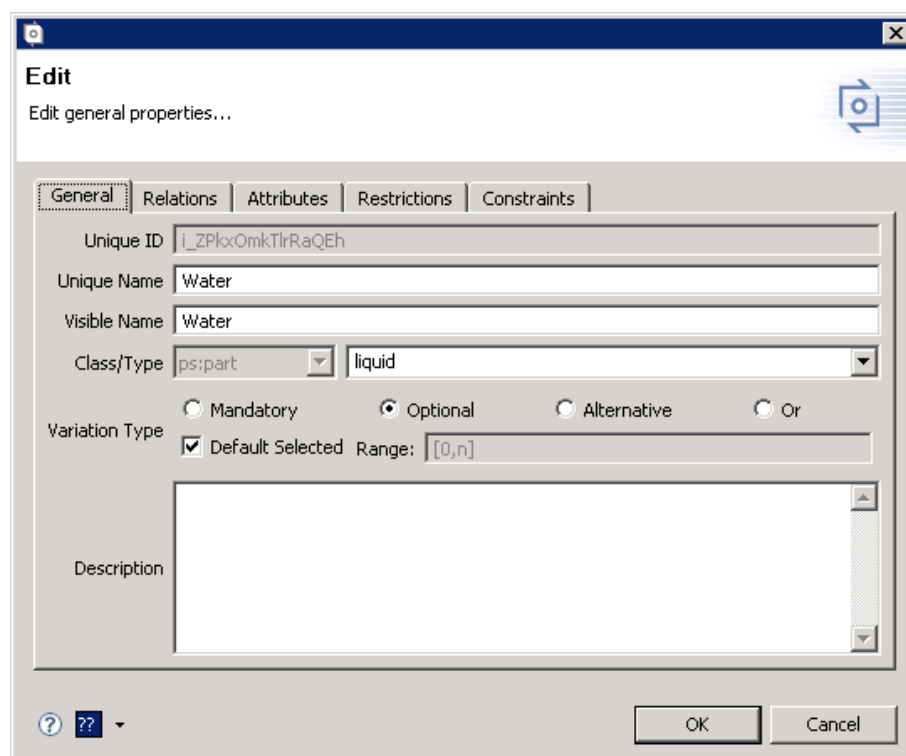
Element Properties Dialog

The properties dialog for an element contains a General, Relations, Attributes, Restrictions, and Constraints page.

General Page

This page configures the general properties of a model element. According to the model type the available element properties differ (see [Figure 7.6, "Family Model Element Properties"](#)).

Figure 7.6. Family Model Element Properties

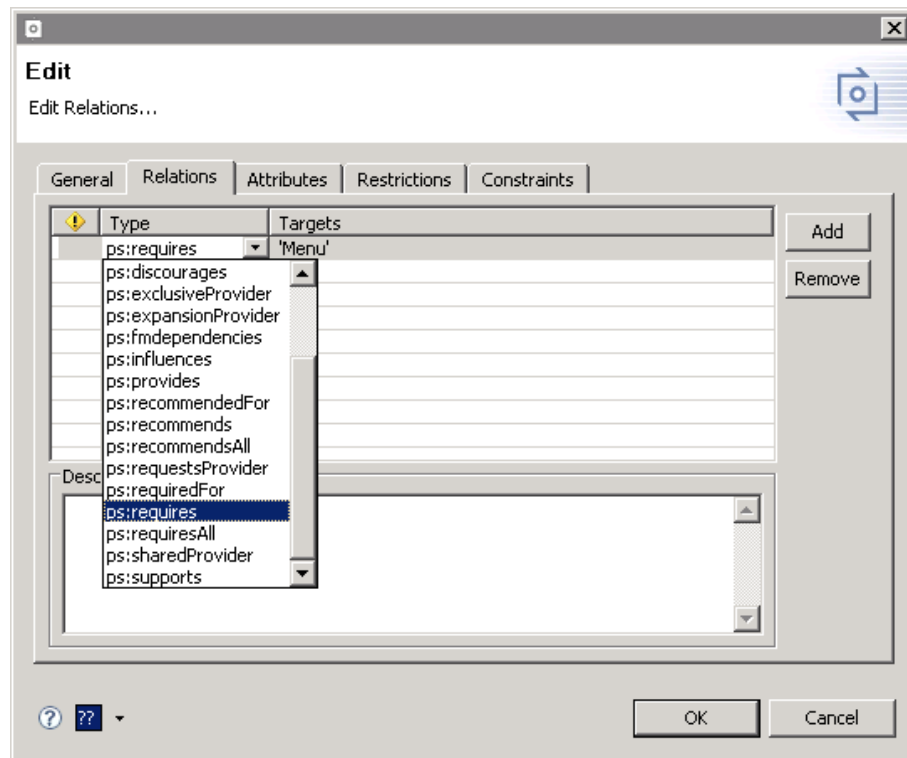


The following list describes the properties that are always available.

Unique ID	The unique identifier for the model element. This identifier is generated automatically and cannot be changed. Every Feature Model element has to have a unique identifier.
Unique Name	The 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 elements of the same model. 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. The Unique name is displayed by default (in brackets if the visible name is also displayed).
Visible Name	The 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.
Class/Type	The class and type of the model element. In feature models elements can only have class <i>ps:feature</i> . Thus the element class for features cannot be changed. Elements in Family Models can have one the following classes: <i>ps:component</i> , <i>ps:part</i> , or <i>ps:source</i> . The root element of a family model always has the class <i>ps:family</i> . The type of a model element is freely selectable.
Variation Type	The Variation type of a model element. The variation type specifies, which selection group applies to the element. One of " <i>mandatory</i> ", " <i>optional</i> ", " <i>alternative</i> " or " <i>or</i> " can be selected.
Range	For variation type <i>Or</i> it is possible to specify the number of features / family elements 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>Or</i> group.
Default Selected	This property defines the default selection state of a model element. Default selected elements are selected implicitly if the parent element is selected. To deselect this element either the parent has to be deselected or the element itself has to be excluded by the user or the auto resolver. Note, that by default the default selection state is disabled for features and enabled for family elements.
Description	The description of the model element. For formatted text editing see Section 7.5.1, "Common Properties Page" . The description field is also available on the other pages.

Relations Page

This page allows definition of additional relations between an element and other elements, such as features or components (see [Figure 7.7, "Element Relations Page"](#)). 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 [Section 5.2.3, "Element Relations"](#).

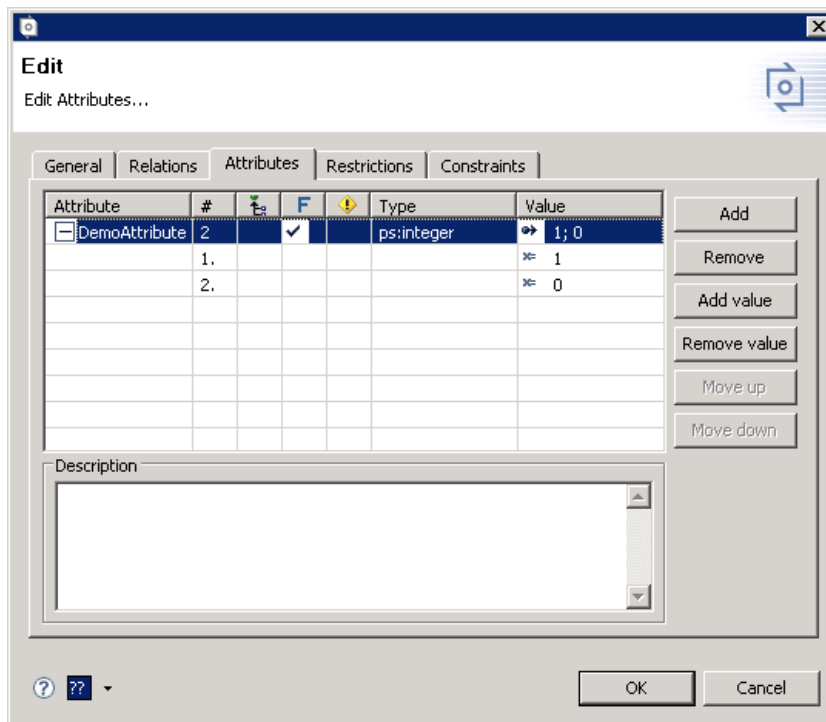
Figure 7.7. Element Relations Page


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 in [Figure 7.8, "Sample attribute definitions for a feature"](#), the attribute DemoAttribute has two value definitions (1 and 0).

Figure 7.8. 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 **F** 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 VDM. Default values can be (optionally) defined here for non-fixed attributes. These are used if no value is specified in the VDM.

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. In the appearing dialog 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 opens this editor. See [the section called “Restrictions Page”](#) 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 choosing a value from a list (combo box) of predefined values, 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 section called “Attribute Value Calculations with pvProlog”](#).

The name of an attribute can be inserted directly or chosen from a list of attributes defined for the corresponding element type in the `pure::variants` type model. When choosing an attribute from the list, the attribute type and the fixed state of the attribute are set automatically.

It is also possible to provide attributes which have a configurable collection of values as data type. Each contained value is available in a variant if the corresponding restriction holds true. To use this feature, square brackets ("[]")

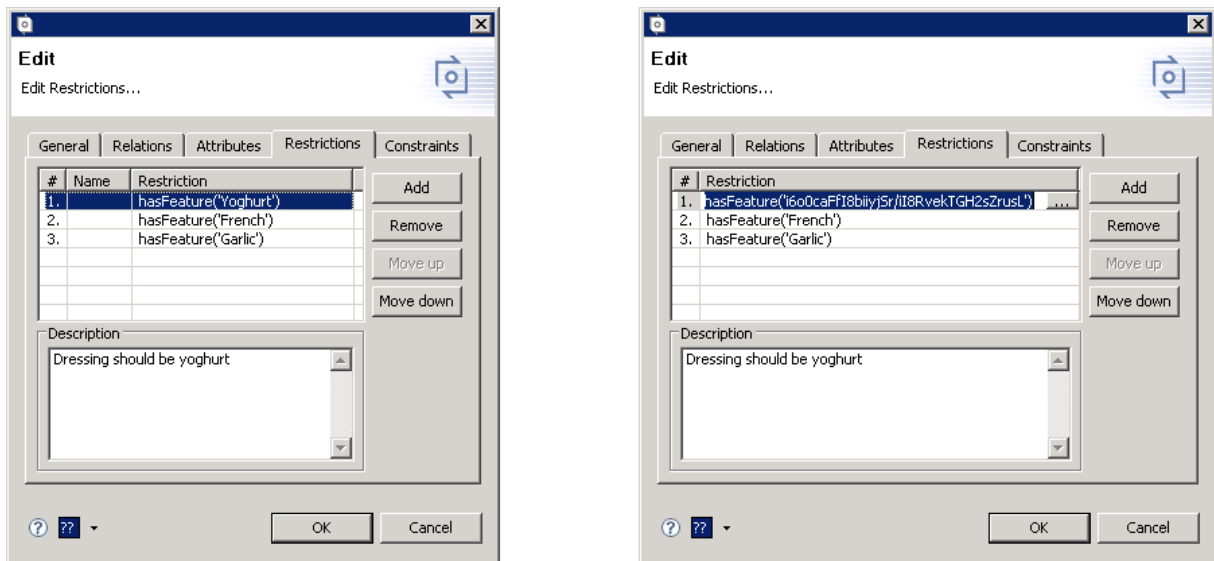
for list values or curly brackets ("{}") for set values have to be appended to the data type of the attribute in column **Type**, e.g. *ps:string{}*, *ps:boolean[]*, or *ps:integer{}*.

The use of attributes is covered further in [Section 5.2.4, “Element Attributes”](#).

Restrictions Page

The Restrictions page defines element restrictions. Any element that can have restrictions can have any number of them. A new restriction can be created using the **Add** button. An existing restriction can be removed using **Remove**. Restrictions are OR combined and evaluated in the given order. The order of the restrictions may be changed using the **Move Up** and **Move Down** buttons on the right side of the page.

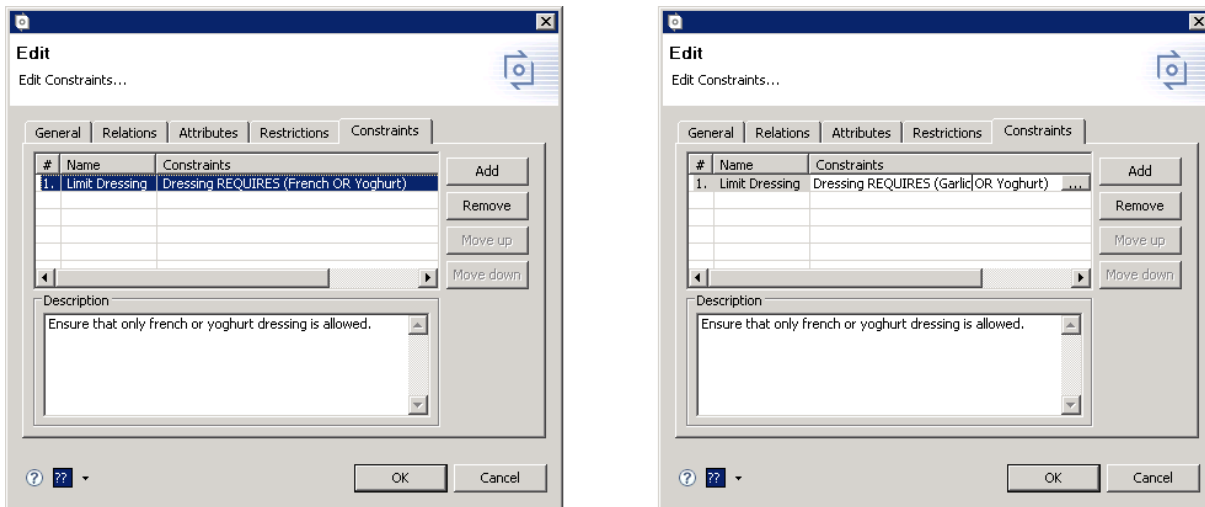
Figure 7.9. Restrictions page of element properties dialog



For each restriction a descriptive name can be specified. It has no further meaning other than a short description of what the restriction checks. A restriction can be edited in place using the cell editor (shown in the right side of figure [Figure 7.9, “Restrictions page of element properties dialog”](#)). 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 advanced editor that is more suitable for complex restrictions. This editor is described more detailed in [the section called “Advanced Expression Editor”](#).

Constraints Page

The Constraints page defines model constraints. Any element that can have constraints can have any number of them. A new constraint can be created using the **Add** button. An existing constraint can be removed using **Remove**. The order of constraints may be changed using the **Move Up** and **Move Down** buttons on the right side of the page. This has no effect on whether a constraint is evaluated or not; constraints are always evaluated.

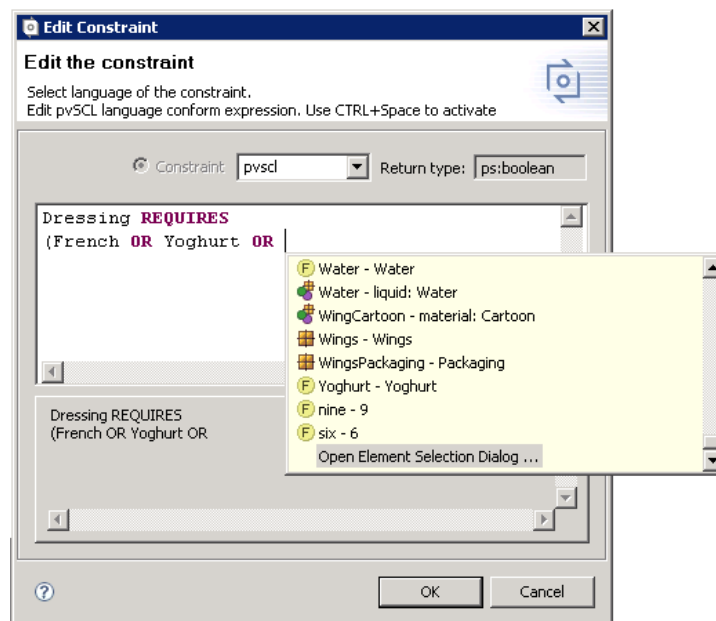
Figure 7.10. Constraints page of element properties dialog

For each constraint a descriptive name can be specified. It has no further meaning other than a short description of what the constraint checks. A constraint can be edited in place using the cell editor (shown in the right side of figure [Figure 7.10, “Constraints page of element properties dialog”](#)). The ... button opens an advanced editor dialog that is more suitable for complex constraints. This editor is described more detailed in [the section called “Advanced Expression Editor”](#).

Advanced Expression Editor

The advanced expression editor is used everywhere in pure::variants where more complex expressions may be inserted. This is for instance when writing more complex restrictions, constraints, or calculations.

Currently it supports the two languages *pvProlog* and *pvSCL*. Special editors are available for both languages. [Figure 7.11, “Advanced pvSCL expression editor”](#) shows the *pvSCL* editor editing a constraint.

Figure 7.11. Advanced pvSCL expression editor

This dialog supports syntax highlighting for *pvSCL* keywords and auto completion for identifiers. There are two forms of completion. Pressing **CTRL+SPACE** while typing in an identifier opens a list with matching model

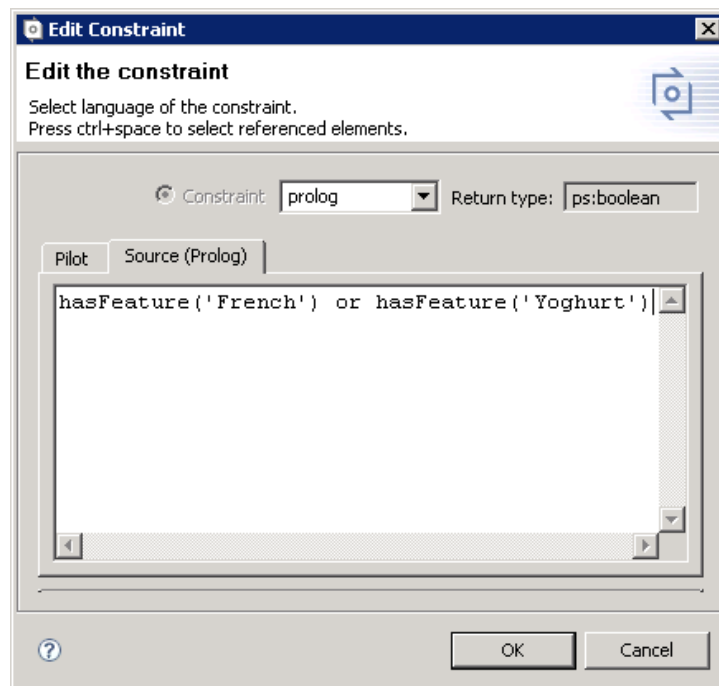
elements and *pvSCL* keywords as shown in the figure. If the user enters "<ModelName>." or "@<ModelId>/" a list with the elements of the model is opened automatically. When pressing **CTRL+SPACE** the opened list contains all kind of proposals: models, elements and operations, if there is no context information available. Therefore an typing of "" opens the list with only elements contained. When then one of the elements is selected, the full qualified name of the element is inserted into the code, i.e. "<ModelName>.<ElementName>". There is always a special entry at the end of such a list, "Open Element Selection Dialog...", which opens the Element Selection dialog supporting better element selection. This dialog is described more detailed in [the section called "Element Selection Dialog"](#).

Warning

The *pvSCL* syntax is not checked in this editor. A syntactically wrong expression will cause the model evaluation to fail.

Figure 7.12, "Advanced *pvProlog* expression editor" shows the *pvProlog* editor directly editing a constraint expression.

Figure 7.12. Advanced *pvProlog* expression editor

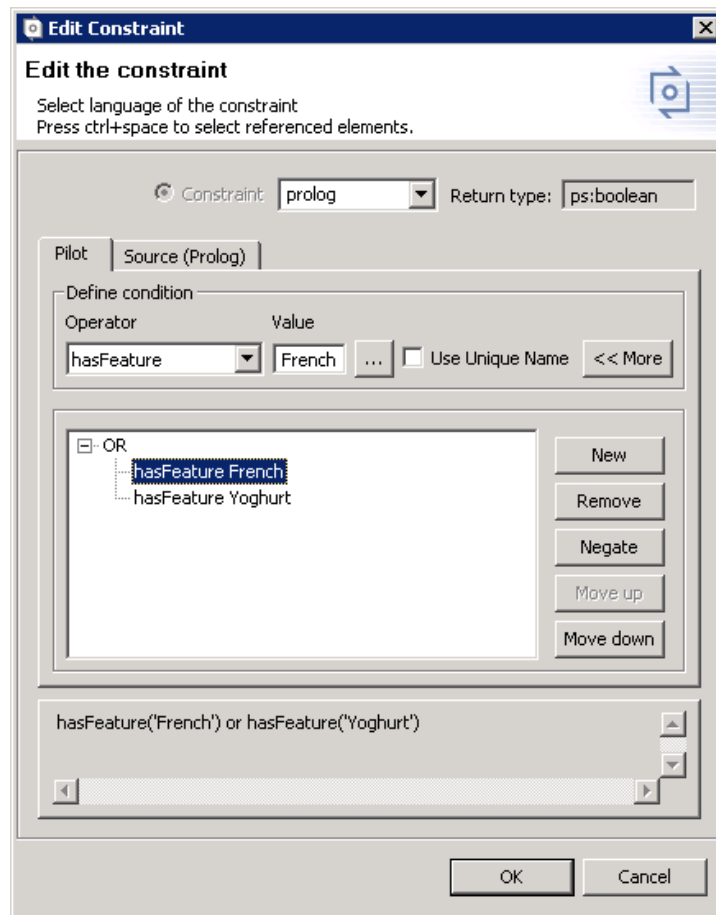


Pressing **CTRL+SPACE** in this editor opens the element selection dialog. All element identifiers selected in this dialog are inserted into the expression as quoted strings. This dialog is described more detailed in [the section called "Element Selection Dialog"](#).

Warning

The *pvProlog* syntax is not checked in this editor. A syntactically wrong expression will cause the model evaluation to fail.

Figure 7.13, "*pvProlog* expression pilot" shows the *pvProlog* editor editing a constraint in the expression pilot. In contrast to the *pvProlog* source editor the pilot always produces syntactically correct *pvProlog* code.

Figure 7.13. pvProlog expression pilot

A *pvProlog* function can be inserted into the expression by pressing on button "New" or choosing Add->New from the context menu. The inserted function can be changed by choosing another function in field "Operator". The argument of the function is added by pressing on button "..." next to field "Value". If the check button "Use Unique Name" is checked, the unique name of the selected element is inserted as argument. Otherwise the id of the selected element is inserted.

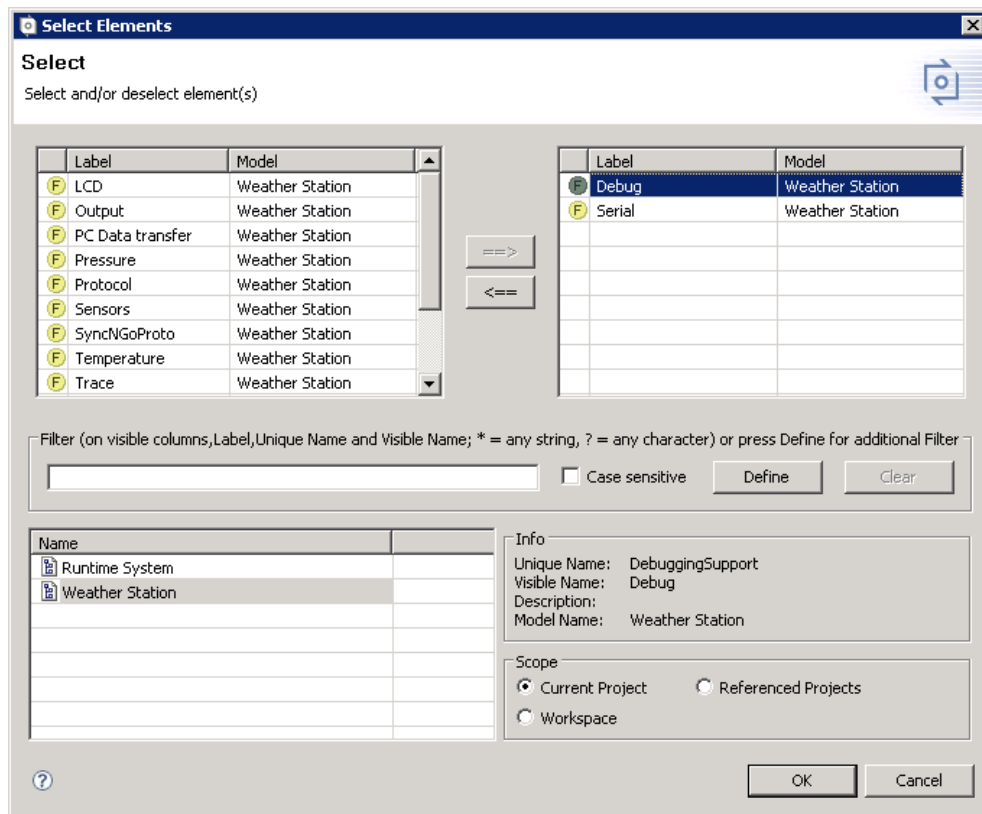
An operator can be added by choosing the corresponding operator from the Add context menu entry. To change an operator the context menu entry "Change to" is used. The "Negate" button adds a "NOT" operator on top of the selected function or operator.

A selected function or operator can be removed by pressing button "Remove". The "Move up" and "Move down" buttons are used to move operands up or down (for instance to swap operands).

The resulting *pvProlog* source code for the constructed expression is shown in the bottom half of the editor.

Element Selection Dialog

The element selection dialog (figure [Figure 7.14, "Element selection dialog"](#)) is used in most cases when a single element or a set of elements has to be selected, e.g. for choosing the relation target elements when inserting a new relation. 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 7.14. 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.

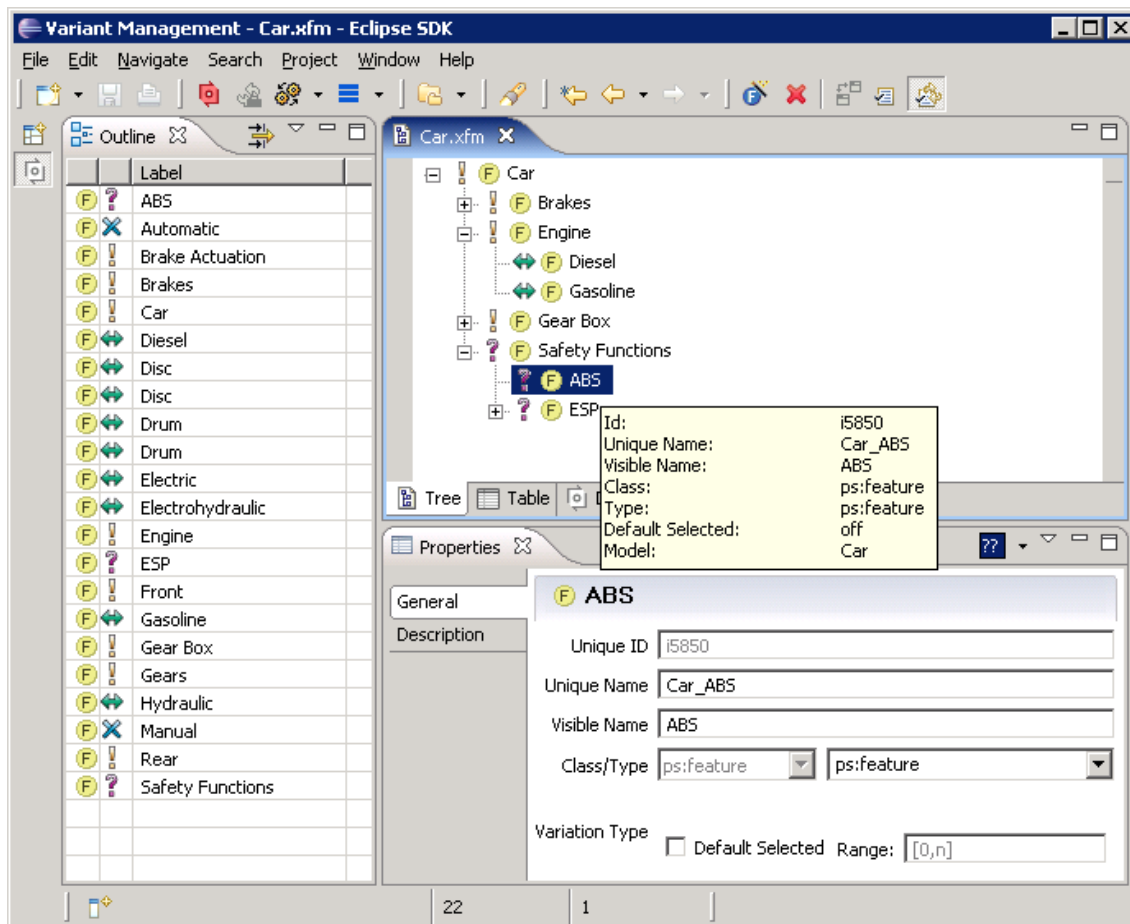
Tip

The element information shown in the left and right *Label* fields is configurable. Use *Table Layout->Change...* from the context menu to select and arrange the visible columns. See [the section called “Table Editing Page”](#) for additional information on table views.

7.3.2. 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 [Section 7.3.4, “Variant Description Model Editor”](#) and [Section 4.3, “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 9.4, “Element variation types and its icons”](#)).

Figure 7.15. Feature Model Editor with outline and property view

Some keyboard shortcuts are supported in addition to mouse gestures (see [Section 9.12, “Keyboard Shortcuts”](#)).

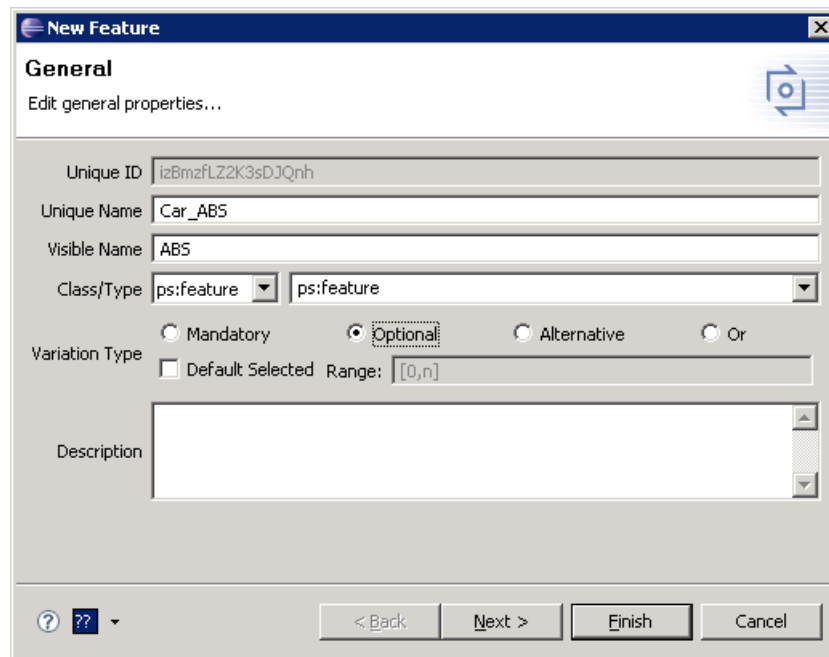
Creating and Changing Features

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 7.16, “New Feature wizard”](#)) 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**, see [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 feature's 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 (see [Section 9.12, “Keyboard Shortcuts”](#)).

Figure 7.16. New Feature wizard


New Feature
Edit general properties...

Unique ID: izBmzfLZ2K3sDJQnh

Unique Name: Car_ABS

Visible Name: ABS

Class/Type: ps:feature (selected) ps:feature

Variation Type: ☐ Mandatory ☒ Optional ☐ Alternative ☐ Or

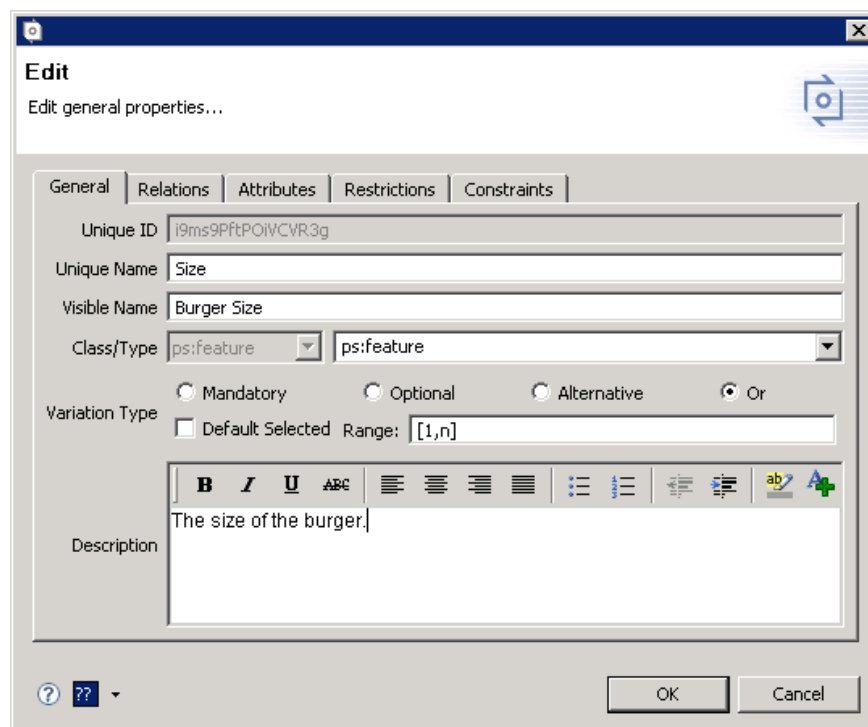
☐ Default Selected Range: [0,n]

Description:

< Back Next > Finish Cancel

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** (see [Figure 7.17](#), “Feature Model Element Properties”).

Figure 7.17. Feature Model Element Properties


Edit
Edit general properties...

General Relations Attributes Restrictions Constraints

Unique ID: i9ms9PftPOiVCVR3g

Unique Name: Size

Visible Name: Burger Size

Class/Type: ps:feature (selected) ps:feature

Variation Type: ☐ Mandatory ☐ Optional ☐ Alternative ☒ Or

☐ Default Selected Range: [1,n]

Description: The size of the burger.

OK Cancel

See [the section called “Element Properties Dialog”](#) for more information about the dialog.

7.3.3. Family Model Editor


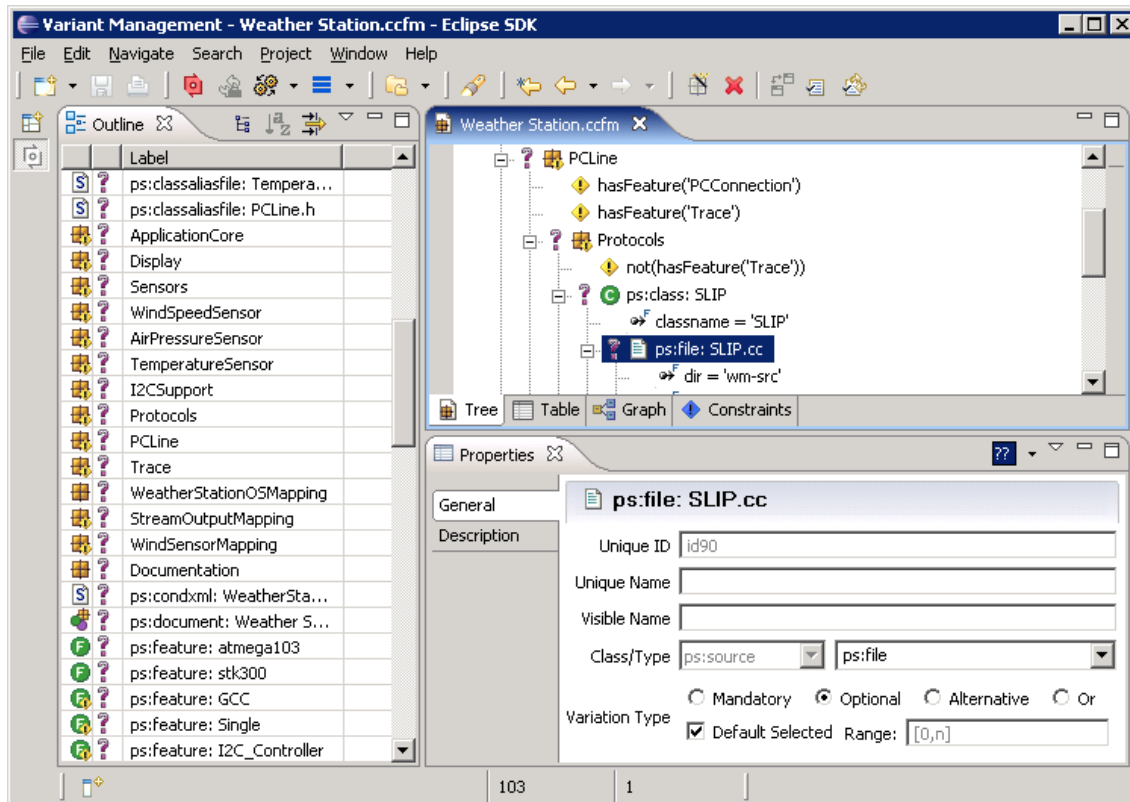
The Family Model Editor shows a tree view of the components, parts, and source elements of a solution space. Each element in the tree is shown with an icon representing the type of the element (see [Table 9.8, “Predefined part types”](#)). The element may additionally be decorated with the restriction sign  if it has associated restriction rules. For more information on Family Model concepts see [Section 5.4, “Family Models”](#).

Figure 7.18. Open Family Model Editor with outline and property view




7.3.4. Variant Description Model Editor

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


In this editor there are two tree views, one showing all feature models in the Configuration Space and another showing all family models in the Configuration Space.

Element Selection

A specific model element can be explicitly included in the configuration by marking the check box next to the element. Additional editing options are available in the context menu. For instance, there are menu entries for deselecting or excluding one or whole sub-trees of elements. It is not supported to make a selection for two elements with the same unique name of models with the same name.

Elements may also be selected automatically, e.g. by the Auto Resolver enabled by pressing button . However, the context menu allows the exclusion of an element; this prevents the Auto Resolver from selecting the element.

Each selected element is shown with an icon indicating how the selection was made. The different types of icons are documented in [Table 9.5, “Types of element selections”](#). If the user selects an element that has already been selected automatically its selection type becomes user selected and only the user can change the selection.

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

Attribute Overriding

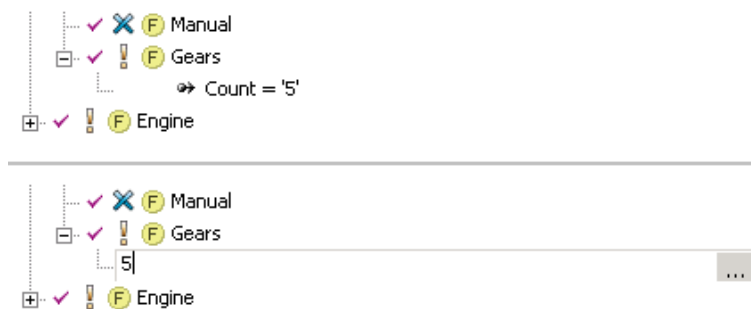
The value of non-fixed attributes is specified in the VDM. Therefore, the Variant Description Model Editor allows to change non-fixed attributes. There are three possibilities:

- with the Properties view (see [Section 7.4.6, “Properties View”](#))
- with the Attributes view (see [Section 7.4.1, “Attributes View”](#))
- with the cell editors of the Variant Description Model Editor itself

Only the first possibility will be explained in detail. The other two possibilities are similar to the first.

First make sure the VDM editor displays attributes (use context menu **Table Layout** -> **Attributes**). Next, double-click on the attribute you would like to specify a value for. A cell editor opens and a text can be entered for the attribute or pressing the ... button opens the Value editor dialog. The given value will be applied with a click somewhere else in the tree.

Figure 7.19. Specifying an attribute value in VDM with cell editor

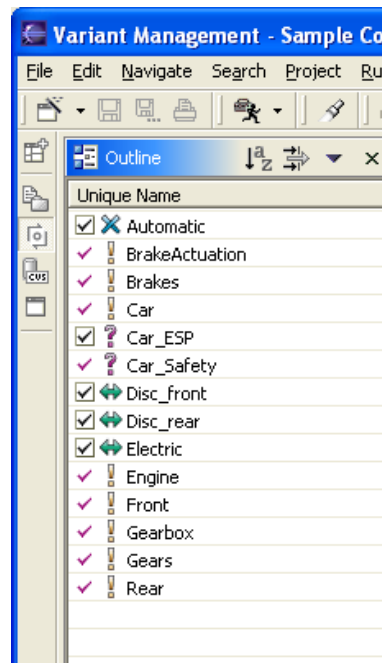


For list and set attributes a special dialog appears when editing attribute values in VDMs. The dialog contains a text field with each line representing one attribute value.

Attributes of grey color mean that there is currently no value set for the attribute and that the default value of the attribute is taken from the associated Feature or Family Model. If no value is specified in VDM for an attribute with default value then a warning will be shown, calling attention to that issue. Attributes with no value in VDM and no default value will produce an error during evaluation.

Element Selection Outline View

The outline view of the VDM shows the selected elements with their selection state. You can click on an element to navigate to it in the VDM. This view may be filtered from the views filter icon or context menu.

Figure 7.20. Outline view showing the list of available elements in a VDM

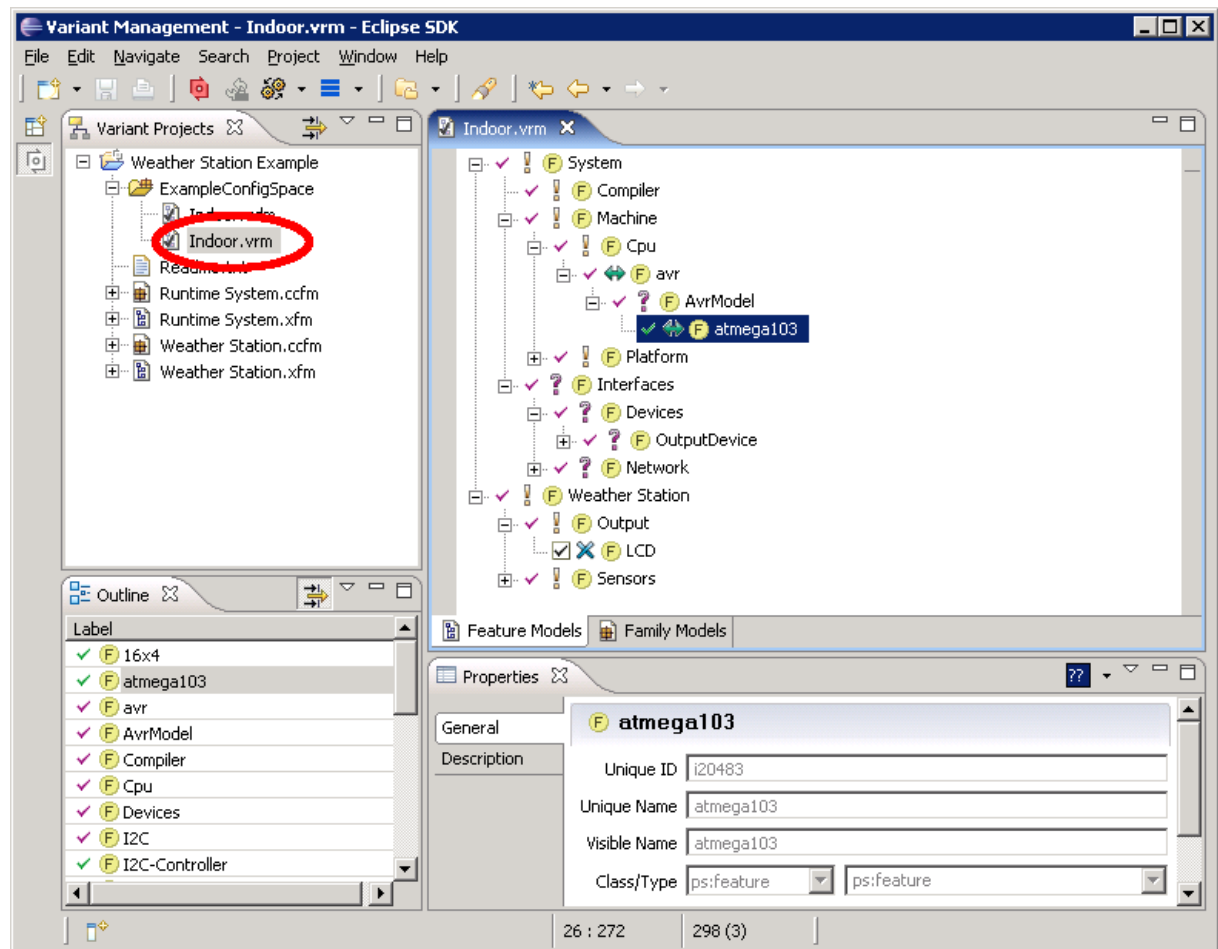
7.3.5. Variant Result Model Editor

The Variant Result Model Editor (VRM Editor) is used to view a saved Variant Result Model. To open a Variant Result Model, double-click on the corresponding file (suffix `.vrm`) in the Variant Projects View. This opens the editor in the style of the VDM Editor.

A Variant Result Model can not be changed because it already represents a concrete variant. Thus the shown element selection is read-only.

If a Variant Result Model is located below a Configuration Space folder, transformation of the Variant Result Model is possible. The required information for the transformation is taken from the Configuration Space. If no valid transformation configuration is found, the transformation will be rejected. A warning is shown if the models of the Configuration Space do not conform to the models in the Variant Result Model.

Figure 7.21, "VRM Editor with outline and properties view" shows a sample variant result model.

Figure 7.21. VRM Editor with outline and properties view

See [Section 5.9.2, “Variant Result Models”](#) for more information about Variant Result Models.

7.3.6. Model Compare Editor

The Model Compare Editor is a special editor provided by pure::variants to view and treat differences between pure::variants models. The behaviour of this editor is very similar to that of the Eclipse text compare editor. For general information about the Eclipse compare capabilities please refer to the Eclipse [Workbench User Guide](#). The *Task* section contains a subsection *Comparing resources* which explains the compare action in detail. For more information on the use of the pure::variants Model Compare Editor see [Section 6.6, “Comparing Models”](#).

7.3.7. Matrix Editor


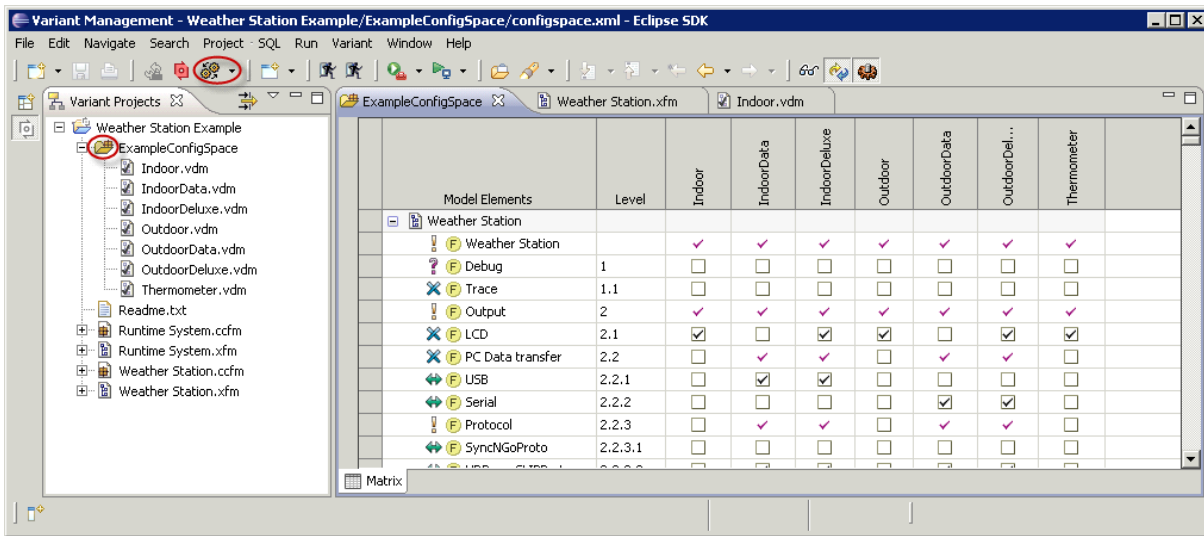
The matrix editor gives an overview of feature selections and attribute values across the variants in a configuration space. The editor is opened by double-clicking on the configuration space icon  in the *Variant Projects* view (see [Figure 7.22, “Matrix Editor of a Configuration Space”](#)). The editor 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 editor (Show elements). The general filtering mechanism can also be used to further specify which features are visible (also accessible from the context menu).

Figure 7.22. Matrix Editor of a Configuration Space

The Matrix Editor allows to change selection for each VDM and attribute values per VDM. As for the table the columns of the Matrix Editor can be changed via the same context menu (*Table Layout->Change...*). Only the first column which shows the Configuration Space relevant Input Models in the order as they would appear for the VDM Editor can not be (re)moved. Note for that column the manner of providing the features. The Matrix Editor supports two ways of representing the features for the Input Models: *Flat* and *Hierarchical*. In the flat manner the models are displayed as root elements and all elements flat directly as children of the models. Only the attributes of the elements are located as children of their containing element. In the hierarchical manner the elements are represented as they are in the *Variant Description Model Editor* that means, they are displayed in their normal tree hierarchy.

In addition the Matrix Editor allows to evaluate the VDMs. This is done with the *Evaluate Models* button in the editors toolbar, identical to the VDM Editor. Evaluation capability of the Matrix Editor also includes the buttons in the toolbar *Enable automatic checking...* and *Enable auto resolver....* If an evaluation is performed, only the currently visible VDMs are evaluated.

Finally it is even possible to perform transformation of the visible VDMs. Use the *Transform all models* button to perform transformation. See [Section 5.9, “Variant Transformation”](#) for detailed information.

7.4. Views

7.4.1. Attributes View

The attributes view shows for a VDM the available attributes of the associated Feature and Family Models. The user can set the value of non-fixed attributes in this view by clicking in the **Value** column of an attribute. If no value is set for an attribute then the value set in the associated Feature / Family Model is shown in grey in the **Value** column. This view may also be filtered to show only the attributes of selected features and/or where no value has been set.

Model Editors. Note that some filters may not work as expected on different models. For example a Variant Model Filter, filtering on selections will not work for a Feature Model Editor.

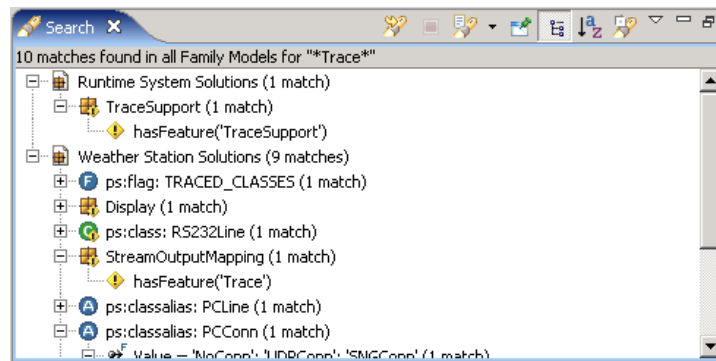
Additionally the layout and filter items may be organized within categories. Layouts or filters, created once appear at first directly below their top-level category. The view allows to create a category by choosing "Create Category..." from the context menu on a parent Category. The context menu provides an action "Move To" on an item selection, which allows to move it to any desired category.

7.4.3. Search View

Feature and Family Models can be searched using the Variant Search dialog. The Variant Search view shows the result of this search and is opened automatically when the search is started. The search results are listed in a table or in a tree representation.

The tree representation structures the search results in a simple tree. The first level of the tree lists the models containing matches. On the second level the matched elements are listed. The next levels finally list the matched attributes, attribute values, restrictions, and constraints.

Figure 7.25. Variant Search View (Tree)



Behind every element in the tree that is a root element of a sub-tree the number of matches in this sub-tree is shown. Double-clicking on an item in the tree opens the corresponding model in an editor with the corresponding match selected. The search results can be sorted alphabetically using the button "Sort by alphabet" in the tool bar of the Search view.

By pressing button "Switch to Table" the table representation of the search results is enabled. The table shows the matched model items in a flat list. Double-clicking on an item in the list opens the corresponding model in an editor with the corresponding match selected. The search results can be sorted alphabetically by clicking on the "Label" column title.

Figure 7.26. Variant Search View (Table)

Label
hasFeature('Trace')
hasFeature('TraceSupport')
hasFeature('Trace')
hasFeature('Trace') or not(hasFeature('PCConnection'))
not(hasFeature('Trace'))
hasFeature('Trace')
hasFeature('RS232Line') or hasFeature('Trace')
hasFeature('Trace') and not(hasFeature('PCConnection'))
not(hasFeature('Trace')) or hasFeature('Display')
hasFeature('Trace')

A search result history is shown when the button "Show Previous Searches" in the tool bar of the search view is pressed. With this history previous search results can be easily restored. The history can be cleared by choosing "Clear History" from the "Show Previous Searches" drop down menu. Single history entries can be removed using the "Remove" button in the Previous Searches dialog.

Note

The history for many consecutive searches with a lot of results may lead to high memory consumption. In this case clear the whole history or remove single history entries using the Previous Searches dialog.

A new search can be started by clicking on button "Start new Search".

For more information about how to search in models using the Variant Search see [Section 6.7, “Searching in Models”](#).

7.4.4. 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.

7.4.5. Problem View/Task View

pure::variants uses the standard Eclipse Problems 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.

7.4.6. Properties View

pure::variants uses the standard Eclipse Properties View. This view shows important information about the selected object and allows editing of most property values. To open the view chose menu *Window->Show View->Properties*.

Figure 7.27. Properties view for a feature

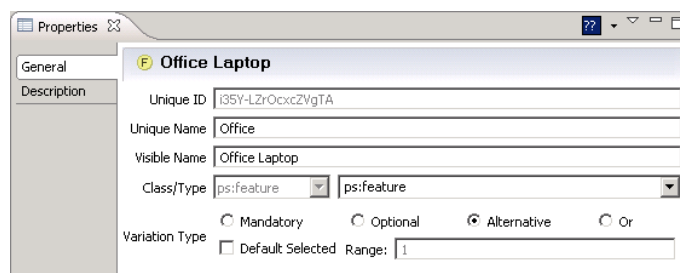
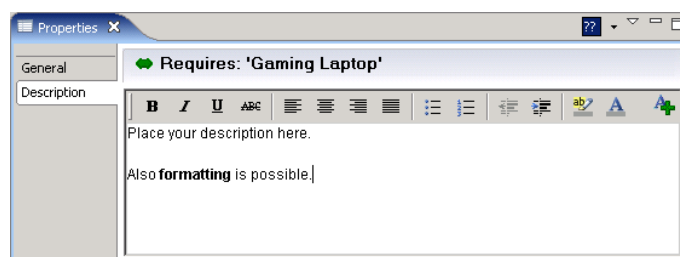


Figure 7.27, “Properties view for a feature” shows the properties view after a feature was selected in the Feature Model Editor. At the left side there are selectable tabs, each containing a set of properties that logically belong together. Usually, tabs *General* and *Description* are shown. The middle area of the properties view presents the properties for the active tab.

The properties view depends on the selection in the workbench made by the user. For instance, selecting a family element like a component allows to edit unique and visible names, whereas for a selected relation the type and the relation targets can be changed in the *General* tab. At the moment, general properties of elements, relations, attributes, attribute values and restrictions can be modified and each of them can have descriptions given in the *Description* tab (see [Figure 7.28, “Description tab in Properties view for a relation”](#)).

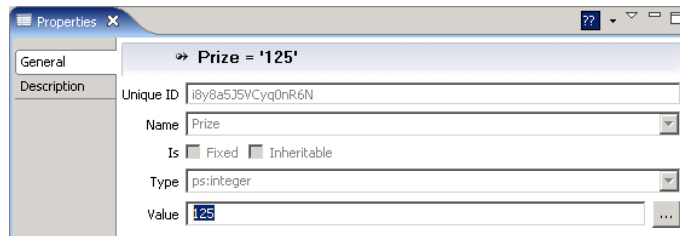
Figure 7.28. Description tab in Properties view for a relation



Properties that are edited won't be applied until the edited field loses the input focus or the **ENTER** key is pressed. That allows you to discard the current change in a text field with the **ESCAPE** key if you like.

If a VDM Editor is active in the workbench and an attribute of the variant is selected then the properties view allows to define the value of the attribute for that variant.

Figure 7.29. Properties view for a variant attribute



For the visible name of features and family elements as well as for descriptions it is possible to specify text in different languages. See [Section 6.11, “Using Multiple Languages in Models”](#) for more information about language support. For formatted text editing of descriptions see [Section 7.5.1, “Common Properties Page”](#).

7.4.7. Relations View

The Relations view shows the relations of the currently selected element (feature/component/part/source element) to other elements. The relations shown in the view are gathered from different locations. The basic locations are:

Model Structure	From the model structure, the relations view gathers information about the parent and child elements of an element.
Element Relations	From the relations defined on an element, the relations view gathers information about the elements depending on the selected element according to the defined relations. Related elements can be elements from the same model or from other models. If a relation to an element of another model cannot be resolved, it may be necessary to explicitly open the other model to let the relations view resolve the element.
Restrictions	From the restrictions defined on an element or on a relation, property, or property value of the element, the relations view gathers information about the elements referenced in these restrictions. According to the language used to formulate the restriction, i.e. pvProlog or pvSCL, the relations view shows the referenced elements below the entry "Prolog Script" or "Simple Constraint Language".
Constraints	From the constraints defined on an element, the relations view gathers information about the elements referenced in these constraints. According to the language used to formulate the constraint, i.e. pvProlog or pvSCL, the relations view shows the referenced elements below the entry "Prolog Script" or "Simple Constraint Language".
Element Properties	From the properties of an element, the relations view gathers information about mapped features. For this purpose there must be a property with the value type "ps:feature". Mapped features can be elements from the same model or from other models. If the mapped feature is an element of another model, it may be necessary to explicitly open the other model to let the relations view resolve the element.

The relations view can be extended to view other relations than the basic relations described above. Please see the `pure::variants` Extensibility Guide for more information about extending the relations view.


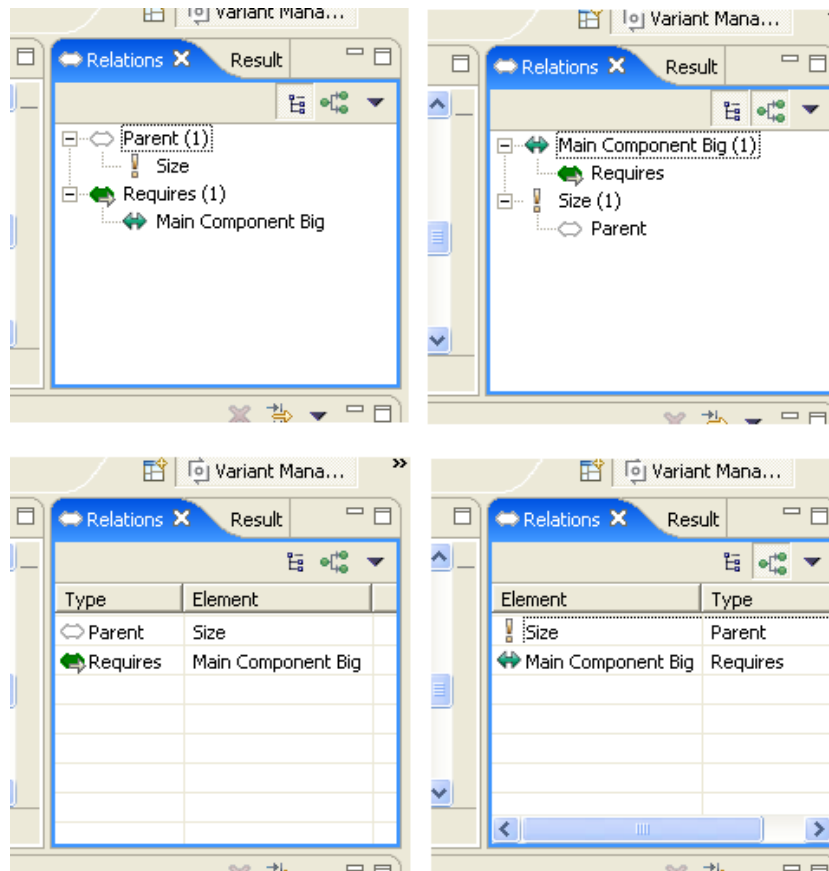
Double-clicking on a related element shown in the Relations View selects that element in the editor. The small arrow in the lower part of the relation icon shows the direction of the relation. This arrow always points from the relation source to the relation destination. For some relations the default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

Figure 7.30. 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.


Attribute values of type "ps:url" are shown as links to external documents in the relations view. A double-click on the appropriate entry opens the assigned system application for the referenced URL.

7.4.8. Result View

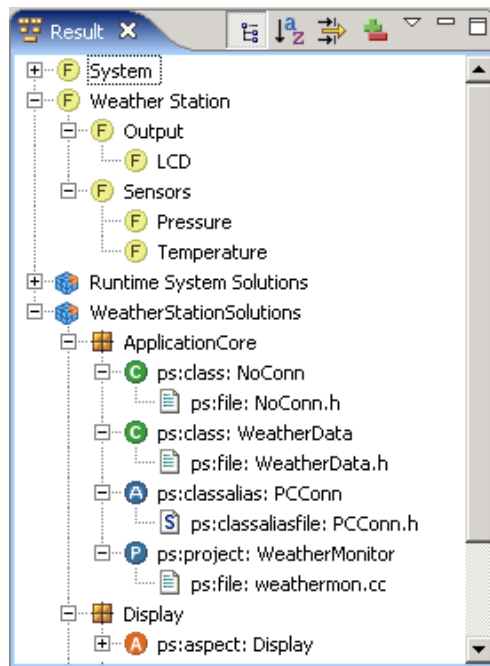
The result view shows the results of model evaluation after a selection check has been performed. It lists all selected feature and Family Model elements representing the given variant.

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.

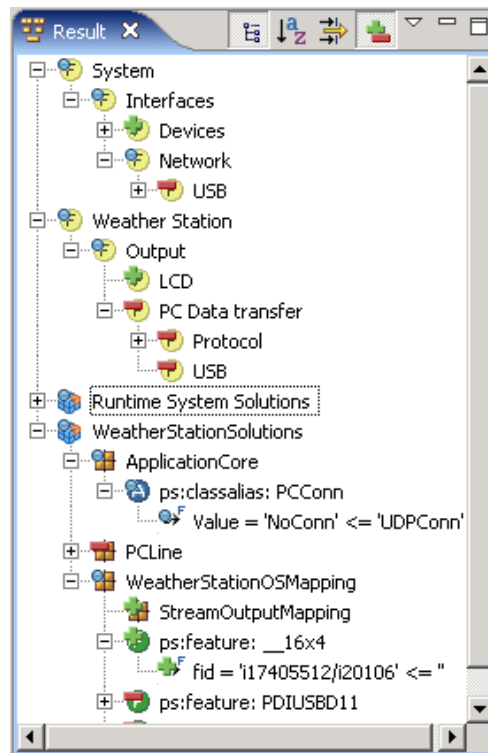
The result view displays a result corresponding to the currently selected VDM. If no VDM is selected, the result view will be empty. The result view is automatically updated whenever a VDM is evaluated.

Figure 7.31. 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 reference result.

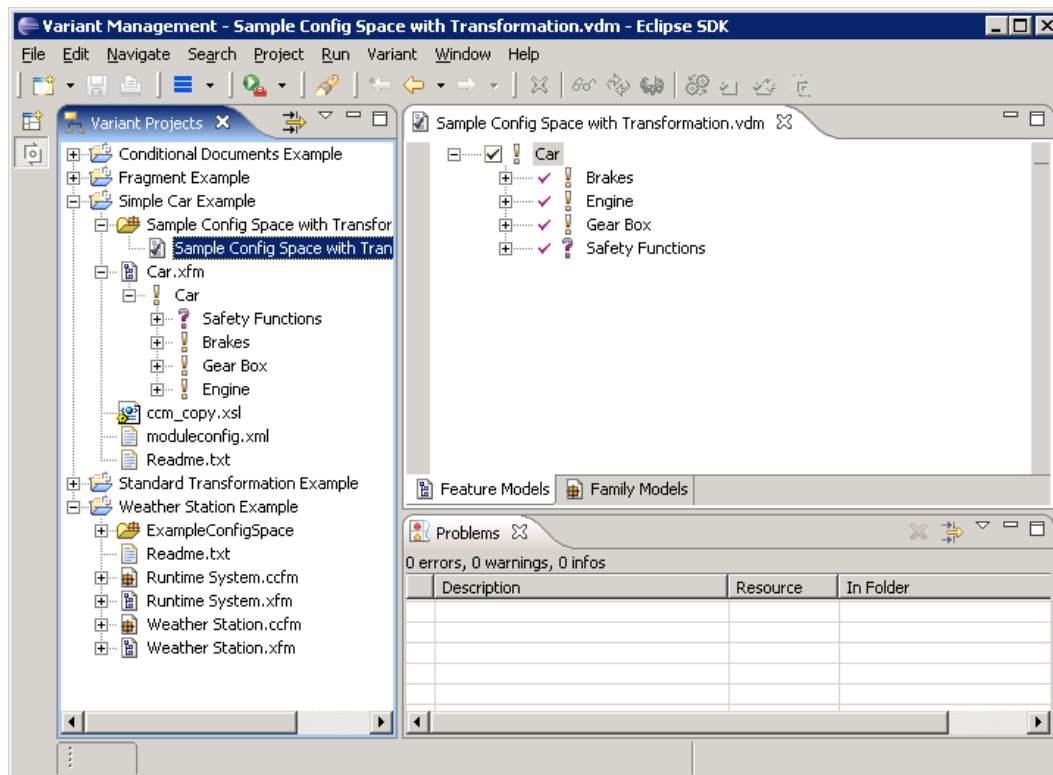
The fixed reference can be either set to the current result or can be loaded from a previously saved variant result (a .vrn file). The reference result is set from the result view menu (third button from right in toolbar). To set a fixed result as reference use *Set current result as reference*. To load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

Figure 7.32. 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).

7.4.9. Variant Projects View

The Variant Projects View (upper left part in [Figure 7.33, “The Variant Projects View”](#)) shows all variant management projects in the current workspace. Projects and folders or models in the projects can be opened in a tree-like representation. 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 7.33, “The Variant Projects View”](#) one editor is shown for a variant description model with some features selected.

Figure 7.33. The Variant Projects View

7.5. Model Properties

pure::variants models have a set of properties. Each model has at least a name. Optionally it can have an author, version, description, and a set of custom properties. Model properties are set by right-clicking on a model in the **Variant Projects** view and choosing **Properties** from the context menu. Depending on the kind of model and the registered extensions, several property pages are available.

7.5.1. Common Properties Page

The common properties are provided on the **Model** page (see [Figure 7.34, “Feature Model Properties Page”](#)).


The common properties of all models are the name, author, version, and description of the model. Additionally the description type can be changed. Available types are plain text and HTML text. Models created with a version lower than 3.0 of pure::variants usually have the plain text type. Setting to HTML text description type allows to format descriptions with styles like bold and italic or with text align like left, center and right (see again [Figure 7.34, “Feature Model Properties Page”](#)). For a full set of HTML formatting possibilities open the extended HTML description dialog by pressing the  button in the tool bar of the description field.

Figure 7.34. Feature Model Properties Page

Properties for Runtime System.xfm

type filter text

- Info
- Access Rights
- General Properties
- Model**

Model

ID: i17405512

Name: Example

Author: user

Version of Model: 1.0

Description Type: HTML-Text

Description

An example feature model.

Restore Defaults Apply

OK Cancel

7.5.2. General Properties Page

Custom model properties are defined on the **General Properties** page (see [Figure 7.35, “General Model Properties Page”](#)).

Figure 7.35. General Model Properties Page

Properties for Runtime System.xfm

type filter text

- Info
- Access Rights
- General Properties**
- Model

General Properties

Attribute	#	Type	Value
+ last_reviewd	1	ps:date	2007-01-01
+ last_reviewer	1	ps:string	user1
+ confidential	1	ps:boolean	false

Add Remove Add value Remove value Move up Move down

Description

Confidential state of the model.

Restore Defaults Apply

OK Cancel

For each property a name, type, and value has to be specified. Optionally a description can be provided.

New properties are added by clicking on button **Add** or by double-clicking in the first empty row of the table. Additional attribute values can be added by selecting the property and then clicking on button **Add value**. To remove a value select it and click on button **Remove value**. A whole property can be removed by selecting the attribute and clicking on button **Remove**.

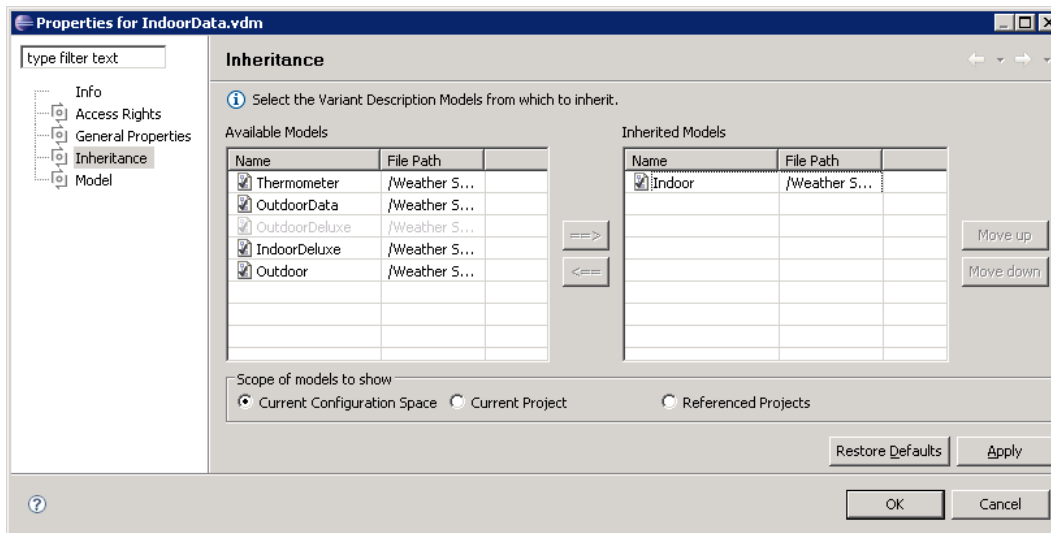
As for element attributes, model properties can also have a list type by simply adding square brackets ("[]") to the type name, e.g. *ps:string[]*, *ps:integer[]*.

Special model properties, like the name, author, version, and description of the model usually configured on other model property pages, are not shown in the **General Properties** list. To include these properties in the list, check option "Include invisible properties in list".

7.5.3. Inheritance Page

The **Inheritance** page is only available for VDMs. It is used to select the models from which a VDM inherits (see [Figure 7.36, "Variant Description Model Inheritance Page"](#)).

Figure 7.36. Variant Description Model Inheritance Page



The left table shows the models which can be inherited. To avoid inheritance cycles models inheriting from the current model are greyed out and can not be inherited. The right table shows the models from which the current model inherits.

Models can be selected from the current Configuration Space, the current project, and referenced projects. See [Section 5.7, "Inheritance of Variant Descriptions"](#) for more information on variant description model inheritance.

Chapter 8. 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 documentation 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](#) functionality for model elements, [Bugzilla](#) integration, synchronization with Borland CaliberRM, and much more.

8.1. Installation of Additional Plug-ins

Additional pure::variants plug-ins are distributed and installed in several ways:

- *Installation from an Update Site* Installation via the Eclipse update mechanism is a convenient way of installing and updating pure::variants plug-ins from an Internet site. See task "Updating features with the update manager" resp. "Updating and installing software" in the Eclipse Workbench User Guide for detailed information on the Eclipse update mechanism (menu Help -> Help Contents and then Workbench User Guide->Tasks).

The location of the site depends on the pure::variants product variant. Visit the pure-systems web site (<http://www.pure-systems.com/pv>) 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 information on how to use update sites with Eclipse 3.5.

- *Archived Update Site* pure::variants uses also the format of archived update sites, distributed as ZIP files, for offline installation into an existing Eclipse installation.

Archived update sites are available for download from the pure::variants internet update site. The location of the site depends on the pure::variants product variant. Visit the pure-systems web site (<http://www.pure-systems.com/pv>) 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 on how to use update sites with Eclipse 3.5. pure::variants archived update site file names start with *updatesite* followed by an identification of the contents of the update site. The installation process is similar to the internet update site installation.

Chapter 9. Reference

9.1. Element Attribute Types

Table 9.1. Supported Attribute Types

Attribute Type	Description	Allowed Values
<i>ps:string</i>	any kind of unspecified text	any
<i>ps:path</i>	path to a file in a file system	any
<i>ps:float</i>	floating point number	a valid floating point number
<i>ps:boolean</i>	boolean value	true and false
<i>ps:url</i>	an URL or URI	any
<i>ps:html</i>	HTML code	any
<i>ps:datetime</i>	date and time (e.g. in ISO 8601 format)	any
<i>ps:filetype</i>	file type identifier	def, impl, misc, app, undefined
<i>ps:insertionmode</i>	value type of source element type <i>ps:fragment</i>	before and after
<i>ps:element</i>	feature or family model element reference	any
<i>ps:directory</i>	path to a directory in a file system	any
<i>ps:integer</i>	integer number	a valid integer number
<i>ps:feature</i>	feature reference	a valid id of a feature
<i>ps:class</i>	<i>ps:class</i> source element reference	a valid id of a <i>ps:class</i> source element

9.2. Element Relation Types

Relations can be defined between the element containing the relation on one side and all other elements of the same or other models on the other side. In the following table the defining element is the element on which the relation is defined. EL is the list of related elements.

Table 9.2. Supported relations between elements (I)

Relation	Description
<i>ps:requires(EL)</i>	At least one element in <i>EL</i> has to be selected if the defining element is selected.
<i>ps:requiresAll(EL)</i>	All elements in <i>EL</i> have to be selected if the defining element is selected.
<i>ps:requiredFor(EL)</i>	If at least one element in <i>EL</i> is selected, then the defining element has to be selected.
<i>ps:requiredForAll(EL)</i>	If all elements in <i>EL</i> are selected, then the defining element has to be selected.
<i>ps:conditionalRequires(EL)</i>	Similar to <i>ps:requires</i> , but the relation is considered only for elements whose parent element is selected.
<i>ps:recommends(EL)</i>	Like <i>ps:requires</i> , but not treated as error if not complied.
<i>ps:recommendsAll(EL)</i>	Like <i>ps:requiresAll</i> , but not treated as error if not complied.
<i>ps:supports(EL)</i>	Like <i>ps:provides</i> , but not treated as error if not complied.
<i>ps:recommendedFor(EL)</i>	Like <i>ps:requiredFor</i> , but not treated as error if not complied.
<i>ps:recommendedForAll(EL)</i>	Like <i>ps:requiredForAll</i> , but not treated as error if not complied.

Relation	Description
<i>ps:conflicts(EL)</i>	If all element in <i>EL</i> are selected, then the defining element must not be selected.
<i>ps:conflictsAny(EL)</i>	If any element in <i>EL</i> is selected, then the defining element must not be selected.
<i>ps:discourages(EL)</i>	Like <i>ps:conflicts</i> , but not treated as error if not complied.
<i>ps:discouragesAny(EL)</i>	Like <i>ps:conflictsAny</i> , but not treated as error if not complied.
<i>ps:influences(EL)</i>	The elements in <i>EL</i> are influenced in some way by the selection of the defining element. The interpretation of the influence is up to the user.
<i>ps:provides(EL)</i>	The "inverse" relation to <i>ps:requires</i> . For all selected elements in <i>EL</i> at least one defining element has to be selected.

Table 9.3. Supported Relations between Elements (II)

Relation	Description	Use for	Partner relation
<i>ps:exclusiveProvider(id)</i>	In a valid configuration at most one <i>exclusiveProvider</i> or one set of <i>sharedProvider</i> for a given id is allowed. Thus, the relation defines a mutual exclusion relation between elements.	Concurrent implementations for an abstract concept.	<i>ps:requestsProvider</i>
<i>ps:sharedProvider(id)</i>	In a valid configuration at most one <i>exclusiveProvider</i> or one set of <i>sharedProvider</i> for a given id is allowed. Thus, the relation defines a mutual exclusion relation between elements.	Shared implementations for an abstract concept.	<i>ps:requestsProvider</i>
<i>ps:requestsProvider(id)</i>	In a valid configuration for each <i>requestsProvider</i> with the given id there must be an <i>exclusiveProvider</i> or any number of <i>sharedProvider</i> with the same id. There may be any number of <i>requestsProvider</i> relations for the same id.	Request existence of an abstract concept.	<i>ps:exclusiveProvider</i>
<i>ps:expansionProvider(id)</i>	In a valid configuration at most one <i>expansionProvider</i> for a given id is allowed. Thus, the relation defines a mutual exclusion relation between elements.	Provides mechanism for implementing variation points with default solution.	<i>ps:defaultProvider</i>
<i>ps:defaultProvider(id)</i>	If an element marked as <i>expansionProvider</i> is additionally marked as <i>defaultProvider</i> for the same given id and there is more than one possible element claiming to be an <i>expansionProvider</i> for this id, then all <i>defaultProvider</i> are excluded. If there are more than one <i>defaultProvider</i> selected and no non- <i>defaultProvider</i> selected, one <i>defaultProvider</i> must be chosen manually.	Provides mechanism for implementing variation points with default solution.	<i>ps:expansionProvider</i>

9.3. Element Variation Types

Table 9.4. Element variation types and its icons

Short name	Variation Type	Description	Icon
mandatory	<i>ps:mandatory</i>	A mandatory element is implicitly selected if its parent element is selected.	
optional	<i>ps:optional</i>	Optional elements are selected independently.	
alternative	<i>ps:alternative</i>	Alternative elements are organized in groups. Exactly one element has to be selected from a group if the parent element is selected (although this can be changed using range expressions). <i>pure::variants</i> allows only one <i>ps:alternative</i> group for the same parent element.	
or	<i>ps:or</i>	Or elements are organized in groups. At least one element has to be selected from a group if the parent element is selected (although this can be changed using range expressions). <i>pure::variants</i> allows only one <i>ps:or</i> group for the same parent element.	

9.4. Element Selection Types








Table 9.5. Types of element selections

Type	Description	Icon
User	Explicitly selected by the user. Auto resolver will never change the selection state of a user selected element.	
Auto resolved	An element selected by the auto resolver to correct problems in the element selection. Auto resolver may change the state of an auto resolved element but does not deselect these elements when the user changes an element 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 element selection.	
Implicit	All elements from the root to any selected element and mandatory elements below a selected element are implicitly selected if not selected otherwise.	
Excluded	The user may exclude an element from the selection process (via a context menu). When the selection of an excluded or any child element of an excluded element is required, an error message is shown.	
Auto Excluded	An element excluded by the auto resolver to correct conflicts. When the selection of an excluded or any child element of an excluded element is required, an error message is shown.	
Non-Selectable	For a specific element selection the auto resolver may recognize elements as non-selectable. This means, selection of these elements always results in an invalid element selection. For other element selections these elements may not non-selectable.	

9.5. Predefined Source Element Types

Table 9.6. Predefined source element types

Source Type	Description	Icon
<i>ps:dir</i>	Maps directly to a directory.	
<i>ps:file</i>	Maps directly to a file.	
<i>ps:fragment</i>	Represents a file fragment to be appended to another file.	

Source Type	Description	Icon
<i>ps:transform</i>	Describes an XSLT script transformation of a document.	
<i>ps:condxml</i>	Maps directly to an XML document containing variation points (conditional parts).	
<i>ps:condtext</i>	Maps directly to a text document containing variation points (conditional parts).	
<i>ps:flagfile</i>	Represents a file that can hold flags such as a C/C++ header file containing preprocessor defines.	
<i>ps:makefile</i>	Represents a make (build) file such as GNU make files containing make file variables.	
<i>ps:classaliasfile</i>	Represents a file containing an alias e.g. for a C++ class that can be concurrently used in the same place in the class hierarchy.	
<i>ps:symlink</i>	Maps directly to a symbolic link to a file.	

The following sections provide detailed descriptions of the family model source element types that are relevant for the standard transformation (see [Section 6.3.2, “Standard Transformation”](#)).

All file-related source element types derived from element type `ps:destfile` specify the location of a file using the two attributes `dir` and `file`. Using the standard transformation the corresponding file is copied from `<ConfigSpaceInputDir>/<dir>/<file>` to `<ConfigSpaceOutputDir>/<dir>/<file>`. Source element types derived from `ps:srcdestfile` optionally can specify a different source file location using the attributes `srcdir` and `srcfile`. If one or both of these attributes are not used, the values from `dir` and `file` are used instead. The source file location is relative to the `<ConfigSpaceInputDir>`.

Every description has the following form:

9.5.1. aSourceElementType

Attributes: `attributeName1` [*typeName1*]
 `attributeName2?` [*typeName2*]

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.

9.5.2. ps:dir

Attributes: `dir` [*ps:directory*]
 `srcdir?` [*ps:directory*]

This source element type is used to copy a directory from the source location to the destination location. All included subdirectories will also be copied. The optional attribute `srcdir` is used for directories that are located in a different place in the source hierarchy and/or have a different name.

9.5.3. ps:file

Attributes: `dir` [*ps:directory*]
 `file` [*ps:path*]
 `type` [*ps:filetype*]
 `srcdir?` [*ps:directory*]
 `srcfile?` [*ps:path*]

This source element type is used for files that are used without modification. The source file is copied from the source location to the destination location. The optional attributes `srcdir` and `srcfile` are used for files that are located in a different place in the source hierarchy and/or have a different source file name.

The value of attribute `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
impl	This type is used for files containing an implementation, e.g. .cc or .cpp files
def	This type is used for files containing declarations, e.g. C++ header files. In the context of <i>ps:classalias</i> calculations this information is used to determine the include files required for a given class.
misc	This type is used for any file that does not fit into the other categories.
app	This type is used for the main application file.
undefined	This type is for files for which no special meaning and/or action is defined.

9.5.4. ps:fragment

Attributes:

- dir** [*ps:directory*]
- file** [*ps:path*]
- type** [*ps:filetype*]
- srcdir?** [*ps:directory*]
- srcfile?** [*ps:path*]
- mode** [*ps:insertionmode*]
- content?** [*ps:string*]

This source element type is used to append text or another file 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*.

9.5.5. ps:transform

Attributes:

- dir** [*ps:directory*]
- file** [*ps:path*]
- type** [*ps:filetype*]
- srcdir?** [*ps:directory*]
- srcfile?** [*ps:path*]
- scriptdir** [*ps:directory*]
- scriptfile** [*ps:path*]
- [scriptparameters]?** [*ps:string*]

The source element type is used to transform a document using an XSLT script and to save the transformation output to a file. The document to transform is searched in `<ConfigSpaceInputDir>/<srcdir>/<srcfile>`. The transformation output is written to `<ConfigSpaceOutputDir>/<dir>/<file>`. `<ConfigSpaceInputDir>/<script>\<dir>/<scriptfile>` specifies the location of the XSLT script to use. Any other attributes are interpreted as script parameters and are accessible as global script parameters in the XSLT script initialized with the corresponding attribute values.

9.5.6. ps:condxml

Attributes:

- dir** [*ps:directory*]
- file** [*ps:path*]
- type** [*ps:filetype*]
- srcdir?** [*ps:directory*]
- srcfile?** [*ps:path*]
- conditionname?** [*ps:string*]
- copycondition?** [*ps:boolean*]

This source element type is used to copy an XML document and optionally to save the copy to a file. Special conditional attributes on the nodes of the XML document are dynamically evaluated to decide whether this node (and its subnodes) are copied into the result document. The name of the evaluated condition attribute is specified

using the attribute `conditionname` and defaults to *condition*. If the attribute `copycondition` is not set to *false*, the condition attribute is copied into the target document as well.

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.

The condition itself has to be a valid XPath expression and may use the XSLT extension functions defined in the following namespaces. Calls to these functions have to be prefixed by the given namespace prefix followed by a colon (":"), e.g. `pv:hasFeature('F')`.

Table 9.7. Registered XSLT Extensions

Namespace Prefix	Namespace
pv	http://www.pure-systems.com/purevariants
pvpath	http://www.pure-systems.com/path
pvstring	http://www.pure-systems.com/string
xmlts	http://www.pure-systems.com/xmlts
dynamic	http://exslt.org/dynamic
math	http://exslt.org/math
sets	http://exslt.org/sets
strings	http://exslt.org/strings
datetime	http://exslt.org/dates-and-times
common	http://exslt.org/common
crypto	http://exslt.org/crypto

For a description of the pure::variants XSLT extension functions see [Table 9.17, “Extension functions providing model information”](#). For a description of the EXSLT extension functions see <http://www.exslt.org>.

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

Example 9.1. 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>
```

A special XML node is supported for calculating and inserting the value of an XPath expression. The name of this node is `pv:value-of` (namespace "pv" is defined as "<http://www.pure-systems.com/purevariants>"). The expression to evaluate has to be given in the attribute `select`. The `pv:value-of` node is replaced by the calculated value in the result document.

Example 9.2. Example use of pv:value-of

Source document:

```
<?xml version='1.0'?>
<version xmlns:pv="http://www.pure-systems.com/purevariants">
  <pv:value-of select="pv:getAttributeValue('Version','ps:feature','version')"/>
</version>
```

Result document:

```
<?xml version='1.0'?>
<version xmlns:pv="http://www.pure-systems.com/purevariants">
  1.0
</version>
```

9.5.7. ps:condtext

Attributes:

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

This source element type is used to copy a text document and optionally to save the copy to a file. Special statements in the text document are evaluated to decide which parts of the text document are copied into the result document, or to insert additional text.

The statements (macro-like calls) that can be used in the text document are listed in the following table.

Macro	Description
PV:IFCOND(<i>condition</i>) PV:IFCONDLN(<i>condition</i>)	Open a new conditional text block. The text in the block is included in the resulting text output if the given condition evaluates to true. The opened conditional text block has to be closed by a PV:ENDCOND call.
PV:ELSEIFCOND(<i>condition</i>) PV:ELSEIFCONDLN(<i>condition</i>)	This macro can be used after a PV:IFCOND or PV:ELSEIFCOND call. If the condition of the preceding PV:IFCOND or PV:ELSEIFCOND is failed, the condition of this PV:ELSEIFCOND is checked. If it evaluates to true, the enclosed text is included in the resulting text output.
PV:ELSECOND PV:ELSECONDLN	This macro can be used after a PV:IFCOND or PV:ELSEIFCOND call. If the condition of the preceding PV:IFCOND or PV:ELSEIFCOND is failed, the enclosed text is included in the resulting text output.
PV:ENDCOND PV:ENDCONDLN	Close a conditional text block. This macro is allowed after a PV:IFCOND, PV:ELSEIFCOND, or PV:ENDCOND call.
PV:EVAL(<i>expression</i>) PV:EVALLN(<i>expression</i>)	Evaluate the given expression and insert the expression value into the result document.

These macros can occur everywhere in the text document and are directly matched, i.e. independently of the surrounding text. The conditions of PV:IFCOND and PV:ELSEIFCOND and the expression of PV:EVAL are the same as the conditions described for source element type *ps:condxml* (see [Section 9.5.6, “ps:condxml”](#) for details).

Conditional text blocks can be nested. That means, that a PV:IFCOND block can contain another PV:IFCOND block defining a nested conditional text block that is evaluated only if the surrounding text block is included in the resulting text output.

For each macro a version with suffix LN exists, i.e. PV:IFCONDLN, PV:ELSEIFCONDLN, PV:ELSECONDLN, PV:ENDCONDLN, and PV:EVALLN. These macros affect the whole line and are only allowed if there is no other macro call in the same line. All characters before and behind such a macro call are removed from the line. It is allowed to mix macros with and without suffix LN, e.g. PV:IFCONDLN can be followed by PV:ENDCOND and PV:IFCOND by PV:ENDCONDLN.

In the example document given below after processing with an *ps:condtext* transformation, the resulting text document only contains an introductory chapter if the corresponding feature `WithIntroduction` is selected.

Example 9.3. A sample conditional document for use with the *ps:condtext* transformation

```
PV:IFCOND(pv:hasFeature('WithIntroduction'))
  This text is in the resulting text output
  if feature WithIntroduction is selected.
PV:ELSECOND
  This text is in the resulting text output
  if feature WithIntroduction is not selected.
PV:ENDCOND
  This text is always in the resulting text output.
```

9.5.8. ps:flagfile

Attributes: **dir** [*ps:directory*]
 file [*ps:path*]
 type [*ps:filetype*]
 flag [*ps:string*]

This source element type is used to generate C/C++-Header files containing `#define <flag> <flagValue>` statements. The `<flagValue>` part of these statements 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 “Providing Values for 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 9.4. Generated code for a *ps:flagfile* for flag "DEFAULT" with value "1"

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

9.5.9. ps:makefile

Attributes: **dir** [*ps:directory*]
 file [*ps:path*]
 type [*ps:filetype*]
 variable [*ps:string*]
 set? [*ps:boolean*]
 makesystem? [*ps:makesystemtype*]

This source element type is used to generate *makefile* variables using a `<variable> += '<varValue>'` statement. The `<varValue>` part of the statement 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 “Providing Values for Part Elements”](#) for more details. The attribute `set` defines if the variable is set to the value (true) or if the variable is extended by the value (false). The generated code is compatible with the `gmake` system. To generate code for a different make system the attribute `makesystem` can be used. The same file location can be used for more than one *ps:makefile* element to include multiple makefile variables in a single file.

Example 9.5. Generated code for a *ps:makefile* for variable "CXX_OPTFLAGS" with value "-O6"

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

9.5.10. ps:classaliasfile

Attributes: **dir** [*ps:directory*]


```

file [ps:path]
type [ps:filetype]
alias [ps:string]

```

This source element type is 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*. A `typedef aliasValue alias;` statement is generated by the standard transformation for this element type. `aliasValue` is the value of the attribute *Value* of the parent part element. Furthermore, in the standard transformation the Variant Result 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 with the value 'def'. If the alias name contains a namespace prefix, corresponding namespace blocks are generated around the `typedef` statement.

Example 9.6. Generated code for a *ps:classalias* for alias "io::net::PConn" with aliased class "NoConn"

```

#ifndef __PConn_include__
#define __PConn_include__
#include "C:\Weather Station Example\output\usr\wm-src\NoConn.h"
namespace io {
namespace net {
typedef NoConn PConn;
}
}
#endif __PConn_include__

```

9.5.11. ps:symlink

Attributes:

```

dir [ps:directory]
file [ps:path]
type [ps:filetype]
linktarget [ps:string]

```







This source element type is used to create a symbolic link to a file or directory named `<linktarget>`.









Note

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

9.6. Predefined Part Element Types

Table 9.8. Predefined part types

Part type	Description	Icon
<i>ps:class</i>	Maps directly to a class in an object-oriented programming language.	
<i>ps:classalias</i>	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.	
<i>ps:object</i>	Maps directly to an object in an object-oriented programming language.	
<i>ps:variable</i>	Describes a configuration variable name, usually evaluated in make files. The variable can have a value assigned.	
<i>ps:flag</i>	A synonym for <i>ps:variable</i> . 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. <i>ps:flag</i> is usually used in conjunction with the <i>flagfile</i> source element, which generates a C++-preprocessor <code>#define <flagName> <flagValue></code> statement in the specified file.	
<i>ps:project</i>	<i>ps:project</i> can be used as the part type for anything that does not fit into other part types.	

Part type	Description	Icon
<i>ps:aspect</i>	Maps directly to an aspect in an aspect-oriented language (e.g. AspectJ or AspectC++).	
<i>ps:feature</i>	Maps directly to a feature in a Feature Model.	
<i>ps:value</i>	General abstraction of a value.	
<i>ps:method</i>	Maps directly to a method of a class in an object-oriented programming language.	
<i>ps:function</i>	Describes the declaration of a function.	
<i>ps:functionimpl</i>	Describes the implementation of a function.	
<i>ps:operator</i>	Maps directly to a programming language operator or operator function.	
<i>ps:link</i>	General abstraction for a link. This could be for instance a www link or file system link.	

The following sections provide detailed descriptions of the family model part element types that are relevant for the standard transformation (see [Section 6.3.2, “Standard Transformation”](#)).

Every description has the following form:

9.6.1. aPartElementType

Attributes: **attributeName1** [*typeName1*]
 attributeName2? [*typeName2*]

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

9.6.2. ps:classalias

Attributes: **value** [*ps:string*]

A class alias is an abstract place holder for variant specific type instantiations. It allows to use concepts similar to interface inheritance with virtual methods in C++ without any overhead. The corresponding source element `ps:classaliasfile` can be used to generate the required C++ code. The unique name of the `ps:classalias` element represents the class name to be used when creating or referencing to objects implementing this abstract interface.

The values of attribute `value` must evaluate to unique names of `ps:class` elements. The value calculated during evaluation is used to locate the implementation class for the abstract class alias.

For more information and an example see [Section 9.5.10, “ps:classaliasfile”](#).

9.6.3. ps:class

Attributes: **classname?** [*ps:string*]

A class represents a class in the architecture. It can be used in conjunction with `ps:classalias`.

The value of the optional attribute `classname` represents the fully qualified name of the class (e.g. `std::string`) to be used when generating code using the standard transformation. Otherwise the unique name of the element is used for this purpose.

For more information and an example on using `ps:class` together with `ps:classalias` see [Section 9.5.10, “ps:classaliasfile”](#).

9.6.4. ps:flag

Attributes: **value** [*ps:string*]

A flag represents any kind of named value, e.g. a C/C++ preprocessor constant. For the standard transformation the value of attribute `value` is evaluated by `ps:flagfile` resp. `ps:makefile` source elements to generate C/C++ specific preprocessor definitions resp. *make* file variables.

For more information about the `ps:flagfile` and `ps:makefile` source element types see [Section 9.5.8, “ps:flagfile”](#) and [Section 9.5.9, “ps:makefile”](#).

9.6.5. ps:variable

Attributes: **value** [*ps:string*]

A variable represents any kind of named value, e.g. a *make* file or programming language variable. For the standard transformation the value of attribute `value` is evaluated by `ps:flagfile` resp. `ps:makefile` source elements to generate C/C++ specific preprocessor definitions resp. *make* file variables.

For more information about the `ps:flagfile` and `ps:makefile` source element types see [Section 9.5.8, “ps:flagfile”](#) and [Section 9.5.9, “ps:makefile”](#).

9.6.6. ps:feature

Attributes: **fid** [*ps:feature*]

This special part type is used to define features which have to be present if the part element is selected. If `pure::variants` detects a selected part of type `ps:feature`, the current feature selection must contain the feature with the id given as value of the attribute `fid`. Otherwise the result is not considered to be valid. The selection problem Auto Resolver (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.

9.7. Expression Language pvProlog

The `pure::variants` expression 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 and constraints. If more complicated computations have to be done, the full power of the underlying Prolog engine can be used. See <http://www.swi-prolog.org> for more information on SWI-Prolog syntax and semantics.

Table 9.9. pvProlog Syntax (EBNF notation)

Expr	= Func UnaryOpExpr OpExpr '(' Expr ')'
OpExpr	= Expr BinOp Expr
UnaryOpExpr	= UnaryOp '(' Expr ')'
BinOp	= 'xor' 'equiv' 'and' 'implies' 'or'
UnaryOp	= 'not'
Func	= FuncName '(' Args ')'
Args	= Argument Args ',' Argument
Argument	= String Number
String	= '''[.]*'''
Number	= ('+' '-')? ['0'-'9']+ ('.' ['0'-'9'])+ ?

```
FuncName = ['a'-'z']['a'-'z' 'A'-'Z' '0'-'9' ' _']*
```

9.7.1. Element References

Most of the *pvProlog* functions require a model element reference as argument. These element references have to be put in single quotes and have the following format.

Table 9.10. Element references

Format	Meaning
ElementName ModelName.ElementName	Refers to the element with the unique name <code>ElementName</code> . If no model name is given, then the element name is resolved first in the model containing the <i>pvProlog</i> expression, then in all other models of the Configuration Space. Otherwise it is resolved in the model with the name <code>ModelName</code> .
ElementId ModelId/ElementId	Refers to the element with the unique id <code>ElementId</code> . If no model id is given, then the element is resolved first in the model containing the <i>pvProlog</i> expression, then in all other models of the configuration space. Otherwise it is resolved in the model with the id <code>ModelId</code> .
:ElementName :ModelName.ElementName	Full qualified element reference. The element reference is resolved in the context of the top parent model.
ElementName:ElementName ModelName.ElementName:ElementName ElementName:ModelName.ElementName ModelName.ElementName: ModelName.ElementName	Qualified element name reference. The element reference is resolved in the context of the model containing the element reference.
ElementId:ElementId ModelId/ElementId:ElementId	Qualified element id reference. The element reference is resolved first in the context of the top parent model. If it could not be resolved, it is resolved in the context of the model containing the element reference.
ElementName[0] ModelName.ElementName[3] ElementName[42]:ElementName ElementName[1]:ModelName.ElementName ModelName.ElementName[10]: ModelName.ElementName	Anonymous context change. This syntax allows to reference elements in the n-th linked model below a Link Element. This is useful especially to reference elements in variant collections.
parent:ElementName parent:ModelName.ElementName parent:parent:ElementName parent:parent:ModelName.ElementName	Parent context access. In a linked model, <code>parent</code> resolves to the parent context. The parent context is the context of the parent model, i.e. the model that contains the link. If a model is not linked by another model, then <code>parent</code> is ignored. <code>parent</code> can only occur at the beginning of a qualified element reference or directly after another <code>parent</code> qualifier, and has to be followed by the context operator ":".

Table 9.11. Examples

Example	Meaning
<code>hasFeature('Product1')</code>	Check if element 'Product1' is selected.
<code>hasFeature('Products.Product2')</code>	Check if element 'Product1' of model 'Products' is selected.

Example	Meaning
<code>hasFeature('Network.Computer:Office.Monitor')</code>	Check if feature Monitor is selected in the linked model Office.
<code>hasFeature('Network.Computer[42]:Monitor')</code>	Check if feature Monitor is selected in the 42th linked Computer model of model Network.
<code>hasFeature('Networks.Network[0]:Computer[42]:Monitor')</code>	Check if feature Monitor is selected in the 42th linked Computer model of the first linked Network model of model Networks.
<code>hasFeature('parent:parent:MonitorSupport')</code>	Check if feature MonitorSupport is selected in the parent model of the parent model of the model containing the element reference.

9.7.2. Logical Operators

Table 9.12. Logical operators in pvProlog

Name/Symbol	Association	Type	Description
<code>xor</code>	right	binary	logical exclusive or
<code>equiv</code>	none	binary	<code>not(A xor B)</code>
<code>and</code>	left	binary	logical and
<code>or</code>	left	binary	logical or
<code>implies</code>	left	binary	logical implication

9.7.3. Supported Functions

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).

Table 9.13. Logical functions in pvProlog

Function	Description
<code>true</code>	Always true
<code>false</code>	Always false
<code>not (EXP)</code>	True if EXP is false

Table 9.14. Functions for value calculations, restrictions, and constraints in pvProlog

Function	Description
<code>isElement(EID)</code>	True if the element with id <code>EID</code> is found in any model.
<code>isFamilyModelElement(EID)</code>	True if the element with id <code>EID</code> is found in a family model.
<code>isFeatureModelElement(EID)</code>	True if the element with id <code>EID</code> is found in a Feature Model.
<code>hasAttribute(AID)</code> <code>hasAttribute(EN, AN)</code> <code>hasAttribute(ET, EN, AN)</code> <code>hasAttribute(EC, ET, EN, AN)</code>	<p>These methods check the existence of a definition for the specified attribute. Attribute is identified by its id (<code>AID</code>), by the symbolic name of its associated element and its symbolic name (<code>EN, AN</code>) or similarly by additionally specifying the element type <code>ET</code>. To ensure correct operation of <code>hasAttribute</code> variants using symbolic names, symbolic element names <code>EN</code> must be unique inside the Configuration Space or inside the element space of the Configuration Space <code>[(ET, EN), (EC, ET, EN)]</code> and the symbolic attribute name <code>AN</code> must be unique inside the attribute space of the element.</p> <p>Note: Due to the evaluation algorithm it is not supported to use one of these functions in a restriction of an attribute to check the existence of the same attribute. Instead the existence of the attribute should be checked in a restriction on the associated element.</p>
<code>getAttribute(AID, VALUE)</code> <code>getAttribute(EN, AN, VALUE)</code> <code>getAttribute(ET, EN, AN, VALUE)</code> <code>getAttribute</code> <code>(EC, ET, EN, AN, VALUE)</code>	<p>These methods get or check the existence and value of the specified attribute. Attribute is identified by its id (<code>AID</code>), by the symbolic name of its associated element and its symbolic name (<code>EN, AN</code>), or similar by additionally specifying the element type <code>ET</code>. When <code>VALUE</code> is a constant, <code>getAttribute</code> checks that the attribute has the specified value. If <code>VALUE</code> is a variable, then subsequent rules can access the attributes value using the specified variable name. To ensure correct operation of <code>hasAttribute</code> variants using symbolic names, symbolic element names <code>EN</code> must be unique inside the Configuration Space or inside the element space of the Configuration Space <code>[(ET, EN), (EC, ET, EN)]</code> and the symbolic attribute name (<code>AN</code>) must be unique inside the attribute space of the element.</p> <p>Note: Due to the evaluation algorithm it is not supported to use one of these functions in a restriction of an attribute to check the existence and value of the same attribute. Instead the existence and value of the attribute should be checked in a restriction on the associated element.</p>
<code>getAttributeName(AID, ANAME)</code>	<code>ANAME</code> is unified with the attribute name of the attribute specified with <code>AID</code> .
<code>getAttributeType(AID, ATYPE)</code>	<code>ATYPE</code> is unified with the meta-model attribute type of the attribute specified with <code>AID</code> .
<code>getAttributeId(EN, AN, AID)</code>	<code>AID</code> is unified with the ID of the attribute with the name <code>AN</code> on the element with the unique name <code>EN</code> .
<code>isTrue(VALUE)</code>	<p>If <code>VALUE</code> is equal to the internal representation of the <code>true</code> value for an attribute of type <i>ps:boolean</i>, it will evaluate to true.</p> <p>Example usage in a restriction:</p> <pre>getContext(EID) and getAttribute(EID, 'ABoolean', BV) and isTrue(BV)</pre>
<code>isFalse(VALUE)</code>	If <code>VALUE</code> is not equal to the internal representation of the <code>true</code> value for an attribute of type <i>ps:boolean</i> , it will evaluate to true.
<code>getContext(EID)</code> <code>getSelf(SELF)</code> <code>getContext(EID, SELF)</code>	<p>These methods can be used to determine the restriction/calculation context. <code>EID</code> 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. <code>SELF</code> is the unique id of the calculation/restriction itself.</p>

Function	Description
	<p>Example: Access the attribute X of the same element in a calculation:</p> <pre>getContext(EID), getAttribute(EID, 'X', XValue)</pre>
<pre>getVariantContext(VCID, VCN) getVariantContextId(VCID) getVariantContextName(VCN)</pre>	<p>These methods are used to get the ID and name of the current variant context element. This element marks a context change in the evaluation of a model that links other models. Each linked model is evaluated in its own unique context.</p>
<pre>isVariant(VN)</pre>	<p>True if the currently evaluated variants unique id or name (VDM name, set in model properties) equals VN.</p> <p>Example usage in a restriction:</p> <pre>isVariant('MyVariant')</pre> <p>Example usage in a calculation getting the variants unique id:</p> <pre>isVariant(Value)</pre>
<pre>getVariantId(MID)</pre>	MID is unified with the unique id of the VDM (.vdm) currently being evaluated.
<pre>getModelList(MIDL)</pre>	MIDL is unified with the list of all models currently being evaluated. This gives access to ids of the Feature Models, Family Models and VDMs in the current configuration space
<pre>getElementModel(EID, MID) getElementModel(MID)</pre>	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.
<pre>getElementChildren(EID, CEIDS)</pre>	CEIDS is unified with the list of children of the element specified with EID or an empty list if no children exist.
<pre>getElementParents(EID, PARIDS)</pre>	PARIDS is unified with the list of parents of the element specified by EID or an empty list if no parents exist.
<pre>getElementRoot(EID, ROOTID)</pre>	ROOTID is the root element for the element specified by EID. For elements with several root elements only one is chosen.
<pre>getElementName(EID, ENAME)</pre>	ENAME is unified with the unique name of the element specified with EID.
<pre>getElementVisibleName(EID, ENAME)</pre>	ENAME is unified with the visible name of the element specified with EID.
<pre>getElementClass(EID, ECLASS)</pre>	ECLASS is unified with the type model element class of the element specified with EID. The standard meta model uses the classes <i>ps:feature</i> , <i>ps:component</i> , <i>ps:part</i> and <i>ps:source</i> .
<pre>getElementType(EID, ETYPE)</pre>	ETYPE is unified with the type model element type of the element specified with EID.
<pre>getMatchingElements(MatchExpr, MEIDS) getMatchingElements(CTXID, MatchExpr, MEIDS)</pre>	<p>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 <code>getContext</code>) unless CTXID is used to specify a different context.</p> <p>Match expressions are explained below.</p> <p>Example: Put all features below the current element with unique names starting with <i>FEA_X</i> in a list:</p> <pre>getMatchingElements('**FEA_X*', LIST)</pre>
<pre>getMatchingAttributes(MatchExpr, EID, AIDS)</pre>	AIDS is unified with all attributes of the element specified with the unique id EID which match with the pattern in MatchExpr. The match pattern is the same as for <code>getMatchingElements</code> , but it must not contain dot characters.

Function	Description
	Match expressions are explained below.
<code>subnodeCount</code> <code>(ECLASS, ENAME, COUNT)</code> <code>subnodeCount</code> <code>(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:</p> <pre>subnodeCount('ps:component', 'X', 3)</pre>
<code>subfeatureCount(FNAME, COUNT)</code>	COUNT is bound to the number of selected child features of feature FNAME. Convenience method for <code>subnodeCount('ps:feature', _, FNAME, COUNT)</code> .
<code>singleSubfeature(FNAME)</code>	True if feature FNAME has just a single child. Convenience method for <code>subnodeCount('ps:feature', _, FNAME, 1)</code> .
<code>alternativeChild(FN, FN2)</code>	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.
<code>userMessage(TYPE, STRING,</code> <code>RELATEDEIDS, CONTEXTEID)</code>	<p>Issues a problem message to be shown, for example, in the Eclipse problems view. TYPE is one of { 'error', 'warning', 'info' }. 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.</p> <p>Example:</p> <pre>userMessage('error', 'Something happened', [REID1, REID2], MYEID)</pre>
<code>userMessage</code> <code>(TYPE, STRING, RELATEDEIDS)</code>	Issues a problem message as above but automatically sets the current element to be the context element.
<code>warningMsg</code> <code>(STRING, RELATEDEIDS)</code> <code>errorMsg(STRING, RELATEDEIDS)</code> <code>infoMsg(STRING, RELATEDEIDS)</code>	Convenience methods for <code>userMessage</code> , sets TYPE automatically.
<code>warningMsg(STRING)</code> <code>errorMsg(STRING)</code> <code>infoMsg(STRING)</code>	<p>Convenience methods for <code>userMessage</code>, set TYPE automatically and uses empty RELATEDEIDS list.</p> <p>Example:</p> <pre>errorMsg('An unknown error occurred')</pre>

9.7.4. Additional Functions for Variant Evaluation

Table 9.15. Additional functions available for variant evaluation

Function	Description
<code>hasElement(EID)</code> <code>has(EID)</code>	<p>True if the element EID is in the variant. Fails silently otherwise.</p> <p>If <code>hasElement</code> 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 models the element has to be in Feature Models of the same rank or in any model of higher rank.</p>
<code>hasFeature(FN)</code>	True if the feature FN is found in the current set of selected features. Fails silently otherwise.

Function	Description
	see <code>hasElement</code>
<code>hasComponent(CN)</code> <code>hasPart(PN)</code> <code>hasSource(SN)</code>	<p>True if the component/part/source <code>xN</code> is found in the current set of selected components in the current component configuration. Fails silently otherwise.</p> <p>See <code>hasElement</code>. <code>hasPart</code> may also refer to components from the same Family Model. <code>hasSource</code> may also refer to parts from the same model.</p>
<code>getAllSelectedChildren(EID,IDL)</code>	<p>Binds <code>IDL</code> to contain all selected children and children of children below and not including <code>EID</code>.</p> <p><code>EID</code> must be an element of a model with the same or higher rank when this rule is used in attribute calculations. <code>EID</code> must be an element of a model with higher rank when used in restrictions. In Family Model restrictions <code>EID</code> can also be an element of a model with the same rank.</p>
<code>getAllChildren(EID,IDL)</code>	Binds <code>IDL</code> to contain all children and children of children below and not including <code>EID</code> .
<code>getMatchingSelectedElements(MatchExpr,MEIDS)</code> <code>getMatchingSelectedElements(CTXID, MatchExpr,MEIDS)</code>	Similar to <code>getMatchingElement</code> described above, but the list is unified only with the elements which are in the current configuration.
<code>sumSelectedSubtreeAttributes(EID,AN,Value)</code>	<p>Calculates the numerical sum of all attributes with the name <code>AN</code> for all selected elements below element with id <code>EID</code> not including the elements attributes itself.</p> <p>see <code>getAllSelectedChildren</code></p>
<code>checkMin(EN,AN,Minimum)</code>	<p>Checks if the value of attribute <code>AN</code> of element <code>EN</code> is equal or greater than <code>Minimum</code>. <code>Minimum</code> has to be a number or the name of an attribute of <code>EN</code> with a number as value.</p> <p>Examples:</p> <pre>checkMin('Car','Wheels',4) checkMin('Car','Wheels','MinNumWheels')</pre>
<code>checkMax(EN,AN,Maximum)</code>	<p>Checks if the value of attribute <code>AN</code> of element <code>EN</code> is equal or less than <code>Maximum</code>. <code>Maximum</code> has to be a number or the name of an attribute of <code>EN</code> with a number as value.</p> <p>Examples:</p> <pre>checkMax('Hand','Fingers',10) checkMax('Hand','Fingers','MaxNumFingers')</pre>
<code>checkRange(EN,AN,Minimum,Maximum)</code>	<p>Checks if the value of attribute <code>AN</code> of element <code>EN</code> is equal or greater than <code>Minimum</code> and equal or less than <code>Maximum</code>. <code>Minimum</code> and <code>Maximum</code> have to be numbers or names of attributes of <code>EN</code> with a number as value.</p> <p>Examples:</p> <pre>checkRange('Car','Speed',0,130) checkRange('Car','Speed',0,'MaxSpeed')</pre>

Additional functions can be specified using the `ps:codelib:prolog` model property (see [Section 9.7.8, “User-Defined Prolog Functions”](#)).

9.7.5. 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 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 sub hierarchies 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.

9.7.6. Accessing Model Attributes

Information stored in model meta attributes can be accessed using the *pvProlog* function `getAttribute`. The table below lists the available attributes and their meaning.

Table 9.16. Meta-Model attributes in pvProlog

Attribute Name	Description
name	The descriptive name of the model.
date	The creation date of the 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).
dir	The absolute path leading to the model directory.

To access a model meta attribute, `getAttribute` has to be called with the ID of model and the name of the attribute. The attribute value is written in the third argument of `getAttribute`. For example, to get the name of the currently processed VDM write the following (the name will be stored in variable `NAME`):

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

9.7.7. Advanced pvProlog Examples

This section demonstrates the use of some useful Prolog functions for handling strings, numbers and lists.

Check if VAR is bound to nothing, a string, a number, an integer number, or a floating point number.

```
var(VAR)
string(VAR)
number(VAR)
integer(VAR)
float(VAR)
```

Convert string STR to a number. The number is stored in N.

```
atom_number(STR,N)
```

Get the length of string STR and store it in LEN.

```
string_length(STR,LEN)
```

Concatenate two strings STR1 and STR2 and store the result in STR.

```
string_concat(STR1,STR2,STR)
```

Concatenate the strings STR1, STR2, and STR3, separated by a slash. Store the result in STR.

```
concat_atom([STR1,STR2,STR3], '/',STR)
```

Split string STR at every slash in the string. The list of substrings is stored in L.

```
concat_atom(L, '/',STR)
```

Check if string STR contains a slash.

```
sub_string(STR,_,_, '/')
```

Check if string STR starts with "abc".

```
sub_string(STR,0,_, 'abc')
```

Check if string STR ends with "xyz".

```
sub_string(STR,_,_,0, 'xyz')
```

Get the first occurrence of a slash in string STR and store its start index in IDX.

```
sub_string(STR,IDX,_,_, '/')
```

Cut the first three characters from string STR1 and store the resulting string in STR.

```
sub_string(STR1,3,_,0,STR)
```

Split the element target TAR into the element ID part and the model ID part. Store the element ID in EID and the model ID in MID.

```
sub_string(TAR,IDX1,_,_, '/'),
sub_string(TAR,0,IDX1,_,MID),
IDX0 is IDX1 + 1,
sub_string(TAR,IDX0,_,0,EID)
```

Construct a new string STR using the given pattern and arguments. The arguments are inserted into the string at the positions marked by "~w" in the order they are given.

```
getAttribute('Product','version',VERSION),
getAttribute('Product','date',DATE),
getAttribute('Product','author',AUTHOR),
sprintf(STR,'Product version ~w. Created on ~w by ~w.',[VERSION,DATE,AUTHOR])
```

Check if variable `L` is bound to a list.

```
is_list(L)
```

Get the length of list `L` and store it in `LEN`.

```
length(L,LEN)
```

Check if item `E` is member of list `L`.

```
member(E,L)
```

Get the third member of list `L`, beginning at index 0, and store it in `E`.

```
nth0(2,L,E)
```

Get the last item of list `L` and store it in `LAST`.

```
last(L,LAST)
```

Append list `L2` to list `L1` and store the resulting list in `L`. Append "c" to "a" and "b", resulting in a list containing "a", "b", and "c".

```
append(L1,L2,L)
append(['a','b'],['c'],ABC)
```

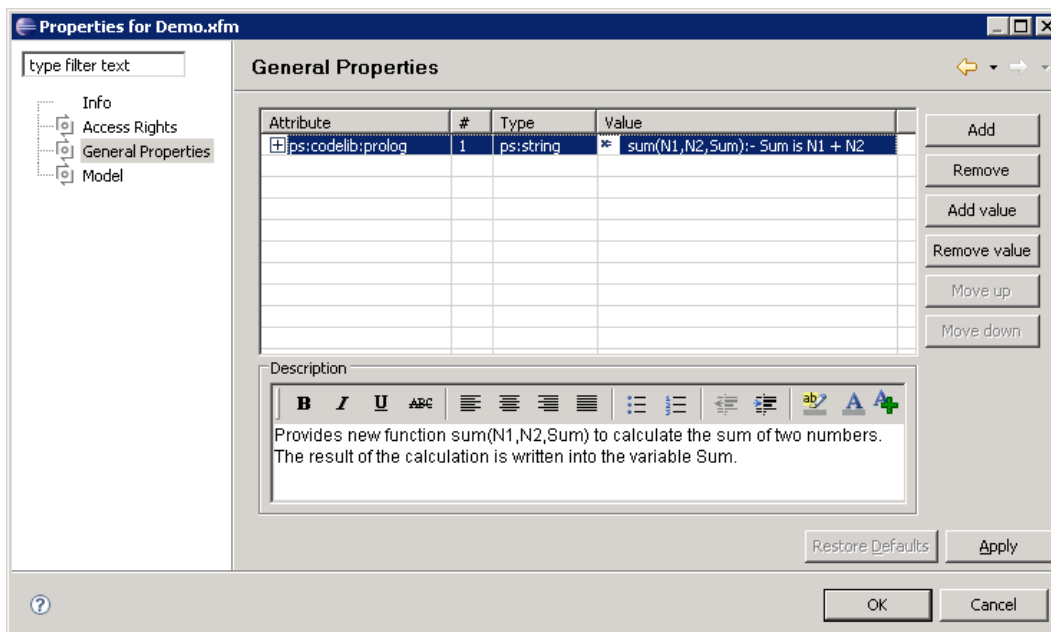
Sort list `L` and store the sorted list in `SORTED`. Duplicates are removed.

```
sort(L,SORTED)
```

9.7.8. User-Defined Prolog Functions

For complex restrictions and calculations it may be useful to provide additional functions, e.g. to simplify the expressions or to share code. For the expression language *pvProlog* a code library can be defined for each model. This is done on the **General Properties** page of a model by adding the property `ps:codelib:prolog` with the code as its value (see [Figure 9.1, "Prolog Code Library Model Property"](#)).

Figure 9.1. Prolog Code Library Model Property



Each model in a Configuration Space, including variant description models, can define code libraries. Code defined in one model is also available in all other models of the same configuration space. When defining the same function in more than one model, these function has to be marked as "multifile". If for instance the function `sum(N1, N2, Sum)` is defined in several models the corresponding code libraries have to contain the following line:

```
:- multifile sum/3.
```

Means the function `sum` with three arguments is defined in multiple models.

9.8. Expression Language pvSCL

The pure::variants expression language *pvSCL* is a simple language to express constraints, restrictions and calculations. It provides logical and relational operators to build simple but also complex boolean expressions. The direct element reference and attribute access syntax makes *pvSCL* expressions more compact than *pvProlog* expressions.

9.8.1. Comments

Expressions can be commented. A comment is started with a slash immediately followed by a star. The comment itself can span multiple lines. It is ended with a star immediately followed by a slash. Comments are ignored when an expression is evaluated.

Syntax `/* comment text */`

Examples `A /* The first character in the alphabet. */ OR
Z /* The last character in the alphabet.*/`

9.8.2. Boolean Values

Expressions can resolve to a boolean value, i.e. TRUE or FALSE. An expression is said to fail if its boolean value is FALSE, and to succeed otherwise. Boolean values have type *ps:boolean*.

Syntax `TRUE
FALSE`

Examples `NOT(TRUE = FALSE)`

9.8.3. Numbers

Numbers can either be decimal and hexadecimal integers, or floating point numbers. Hexadecimal integers are introduced by 0x or 0X followed by digits and / or characters between a and f. Floating point numbers contain a decimal point and / or positive or negative exponent.

Integers have type *ps:integer*, and floating point numbers have type *ps:float*.

Examples `100
10e2
150e-3
0xFF00
1.5
5.5E+3`

9.8.4. Arithmetics

Numbers can be negated, added up, subtracted, multiplied, and divided. If at least one operand of an arithmetic operation has floating point type, the result also will have floating point type.

Arithmetic operators have a higher precedence than comparison operators and a lower precedence than conditionals. Addition and subtraction have a lower precedence than multiplication and division. That means, $2*3+3*2$ is calculated as $(2*3)+(3*2)=6$ instead of $2*(3+3)*2=24$.

Syntax

```
expr + expr
expr - expr
expr * expr
expr / expr
-expr
```

Examples

```
5 * 5 + 2 * 5 * 6 + 6 * 6
-(8 * 10) + (10 * 8)
-0xFF / 5
```

9.8.5. Strings

Strings are sequences of characters and escape sequences enclosed in single quotation marks. The allowed characters are those of the Unicode character set. Strings have type *ps:string*.

Following escape sequences are supported.

Escape Sequence	Meaning
\n	New line
\t	Horizontal tabulator
\b	Backspace
\r	Carriage return
\f	Form feed
\'	Single quotation mark
\"	Quotation mark
\\	Backslash
\0 - \777	Octal character code
\u0000 - \uffff	Unicode character code

Strings can be concatenated with other strings and numbers using the plus operator. The result is a new string containing the source strings and numbers in the order they were concatenated.

Syntax

```
'characters including escape sequences'
```

Examples

```
'Hello'
'10\44' = '10$'
'10\u20AC' = '10€'
'Line ' + 1 + '\n' + 'Line ' + 2
```

9.8.6. Collections

Collections are lists or sets of values of the same type. Lists may contain one and the same value twice, whereas sets only contain unique values. The type of lists either is *ps:list* or the value type followed by *[]*, e.g. *ps:string[]* for a list of strings. The type of sets either is *ps:set* or the value type followed by *{}*, e.g. *ps:integer{}* for a set of integers.

Collection literals have list type. Their items are constructed from the values of any expressions, particularly nested collections, and must have the same type.

Syntax

```
{ expr, expr, ... }
```

Examples

```
{ 'spring', 'summer', 'autumn', 'winter' }
{ 1, 2, 3 }
```

9.8.7. Value Comparison

Expressions can be compared based on their values. For this purpose the expressions are evaluated to their values first, and then the comparison operator is applied to the values resulting in TRUE or FALSE.

Two numbers are compared based on their numeric values, two strings lexically, two collections item by item, and two booleans by their boolean values. All other operand types and type combinations cause the operands to be converted to strings and then compared lexically.

Following comparison operators are supported.

Operator	Meaning
=	Yields TRUE if both operands have the same value.
<>	Yields TRUE if the operands have different values.
>	Yields TRUE if the left operand's value is greater than the right operand's value.
<	Yields TRUE if the left operand's value is less than the right operand's value.
>=	Yields TRUE if the left operand's value is greater than or equals the right operand's value.
<=	Yields TRUE if the left operand's value is less than or equals the right operand's value.

Comparison operators have a lower precedence than arithmetic operators but a higher precedence than logical operators.

Syntax

```

expr = expr
expr <> expr
expr > expr
expr < expr
expr >= expr
expr <= expr

```

9.8.8. SELF and CONTEXT

The keywords *SELF* and *CONTEXT* are context dependent name references. The type of *SELF* and *CONTEXT* is *ps:model* if a model is referenced, *ps:element* for an element, *ps:relation* for a relation, *ps:attribute* for an attribute, and *ps:constant* for an attribute value.

Model Object	SELF	CONTEXT
Constraint	Element containing the constraint	Model containing the constraint
Restriction on element	Element containing the restriction	Element containing the restriction
Restriction on relation	Relation containing the restriction	Element containing the relation
Restriction on attribute	Attribute containing the restriction	Element containing the attribute
Restriction on attribute value	Attribute value containing the restriction	Element containing the attribute value
Attribute value calculation	Attribute value being calculated	Element containing the attribute value

Syntax

```

SELF
CONTEXT

```

Examples

```

SELF AND SELF->value = 5
CONTEXT IMPLIES SELF <> 0

```

9.8.9. Name and ID References

Models, elements, and attributes can be referenced by their unique identifiers. Models can also be referenced by their names, and elements by their unique names, optionally prefixed by the name of the model containing

the element. For a referenced model the result type is *ps:model*, for an element *ps:element*, and for an attribute *ps:attribute*.

Elements can be referenced across linked variants, i.e. variant collections, instances, and references, by means of a path name. Path names navigate to elements in another variant along the variant elements in a variant hierarchy. Variant elements are elements with type *ps:variant* representing the root element of a linked variant.

Path Name Element	Description
variant-name:name	Relative path name
:name	Absolute path name
parent:name	Parent variant navigation
variant-collection-or-instance-name[3]:name	Anonymous variant navigation for variant collections and instances

A name is resolved as follows.

1. If *name* or *model-name* equals "context", "CONTEXT", "self", or "SELF"
 - resolves to the context dependent name reference CONTEXT or SELF
2. If *name* is the name of a visible local variable, iterator or accumulator
 - resolves to the local variable, iterator or accumulator
3. If *name* is the unique name of an element
 - resolves to the element
4. If *element-name* is the unique name of an element in model *model-name*
 - resolves to the element
5. If *name* is the name of a model
 - resolves to the model
6. If it is an absolute *path-name*
 - resolve name without the leading : to an element or model
7. If it is a *path-name* with parent variant navigation
 - resolve name in the context of the parent variant of the current variant to an element
8. If it is a *path-name* with anonymous variant navigation
 - resolve name in the context of the specified variant to an element
9. Otherwise it is a relative name
 - resolve as full qualified name to an element or model

Syntax

```
@id
name
model-name.element-name
path-name
```

Examples

```
@isdkd
Frontdoor
Doors.Backdoor
Residence:Frontdoor:Color->value = 'white'
DoubleLock IMPLIES parent:parent:Manson
```



```
House.Doors[1] AND House.Doors[1]:Type->number = '113a'
```

9.8.10. Element Existence Check

Elements can be referenced independently of their selection, i.e. existence, in the current variant.

To check the existence of an element explicitly, meta-attribute *pv:Selected* can be called on that element returning TRUE if it is selected and thus exists in the variant.

But there are also contexts in which the existence of elements is checked implicitly. These contexts are:

- Constraint (final value)
- Condition of a conditional
- Operand of operator NOT
- Left and right operand of operator XOR
- Left operand of operators AND and OR
- Right operand of operators AND and OR if left operand resolves to FALSE
- Left and right operand of operator EQUALS
- Left operand of operators IMPLIES, REQUIRES, RECOMMENDS, CONFLICTS and DISCOURAGES
- Right operand of operators IMPLIES, REQUIRES, RECOMMENDS, CONFLICTS and DISCOURAGES if left operand resolves to FALSE

Examples

```
Black OR White
IF Winter THEN Snow->pv:Selected ELSE Sunshine->pv:Selected ENDIF
Diesel RECOMMENDS ParticleFilter
NOT(High) IMPLIES Low
```

9.8.11. Attribute Access

Attributes and meta-attributes can be accessed using the call operator. The left operand of the call operator is the context of the call, the right operand the attribute or meta-attribute to call. It is an error if there is no attribute or meta-attribute with the given name for the context of a call.

If the context has model or element type, ordinary model and element attributes can be accessed. The result type is *ps:attribute*.

The value of an attribute is automatically accessed in all contexts a value is required, e.g. operand of a logical, relational, arithmetic, or comparison operator. Meta-attribute *pv:Get* can be used to access an attribute value explicitly. For an attribute with collection type a specific value can be accessed by specifying the index of the value as argument to the call (function call syntax).

The context types meta-attributes can be called on depend on the implementation of a meta-attribute. Meta-attributes may accept an argument list (function call syntax). The result of calling a meta-attribute also depends on its implementation. A meta-attribute with the same name as an ordinary attribute of a model or element hides that attribute.

Syntax

```
context-expr -> attr-name
context-expr -> attr-name(index-expr)
context-expr -> meta-attr-name
context-expr -> meta-attr-name(expr, expr, ...)
```

Examples

```
product->version > 3
seasons->names = { 'spring', 'summer', 'autumn', 'winter' }
seasons->names(1) = 'summer' AND seasons->names(2) = 'autumn'
seasons->names->pv:Size = 4
seasons->names->pv:Get(3) = 'winter'
```

9.8.12. Relations

Expressions can be set in relation to each other. For this purpose the expressions are evaluated to their boolean values. It is an error if this conversion is not possible. The relational operator is then applied to the boolean values resulting in TRUE or FALSE.

Following relational operators are supported.

Operator	Meaning
REQUIRES	If the first operand is FALSE then the second operand will not be evaluated.
IMPLIES	Like REQUIRES.
CONFLICTS	If the first operand is FALSE then the second operand will not be evaluated.
RECOMMENDS	Like REQUIRES but always yields TRUE.
DISCOURAGES	Like CONFLICTS but always yields TRUE.
EQUALS	Yields TRUE if both operands either are TRUE or FALSE.

Relational operators have a lower precedence than conditionals, and logical and arithmetic operators.

Syntax

```
expr IMPLIES expr
expr REQUIRES expr
expr CONFLICTS expr
expr RECOMMENDS expr
expr DISCOURAGES expr
expr EQUALS expr
```

Examples

```
car REQUIRES wheels
legs->number = 4 CONFLICTS human
```

9.8.13. Logical Combinations

Expressions can be logically combined. For this purpose the expressions are evaluated to their boolean values. It is an error if this conversion is not possible. The logical operator is then applied to the boolean values resulting in TRUE or FALSE.

Following logical operators are supported.

Operator	Meaning
AND	Binary operator that yields TRUE if both operands are TRUE.
OR	Binary operator that yields TRUE if at least one operand is TRUE. If the first operand is TRUE then the second operand will not be evaluated.
XOR	Binary operator that yields TRUE if exactly one operand is TRUE.
NOT	Unary operator that yields TRUE if the operand is FALSE.

Logical operators have a lower precedence than comparison operators but a higher precedence than relational operators.

Syntax

```
expr AND expr
expr OR expr
expr XOR expr
```

```
NOT(expr)
```

Examples

```
be OR NOT(be)
cabriolet XOR sunroof
```

9.8.14. Conditionals

Conditionals allow to evaluate alternative expressions depending on the boolean value of a condition. If *boolean-condition-expr* evaluates to TRUE, expression *consequence-expr* is evaluated to determine the result of the conditional expression. Otherwise, expression *alternative-expr* is evaluated. It is an error if *boolean-condition-expr* can not be evaluated to TRUE or FALSE.

Conditionals can occur everywhere expressions are allowed. This means in particular that conditionals can be nested. Conditionals have a higher precedence than relational, logical, arithmetic and compare operators.

Syntax

```
IF condition-expr THEN consequence-expr ELSE alternative-expr ENDIF
```

Examples

```
IF summer THEN
  weather->temperature >= 25
ELSE
  IF winter THEN
    weather->temperature <= 5
  ELSE
    weather->temperature > 5 AND weather->temperature < 25
  ENDIF
ENDIF
```

9.8.15. Variable Declarations

The LET keyword declares at least one variable with name *var-name* and initializes it with the value of expression *init-expr*. The variable is visible only in the expression following keyword IN, and in the *init-expr* of subsequent variable declarators.

Variable declarations can occur everywhere expressions are allowed. To avoid name conflicts it is recommended to use own namespaces for the variable names (e.g. *my:var-name* instead of *var-name*).

The result of a variable declaration is the value of the expression following keyword IN.

Syntax

```
LET var-name = init-expr, var-name = init-expr, ... IN expr
```

Examples

```
LET
  doors = car->frontDoors + car->rearDoors,
  cabrio = (doors = 2),
  limousine = (doors = 4)
IN
  cabrio OR limousine
```

9.8.16. Function Calls

A function call executes the built-in or user-defined function *fct-name* with the given argument list and returns the value calculated by the function. It is an error if the function does not exist.

Syntax

```
fct-name(expr, expr, ...)
```

Examples

```
average(accounts, 'income') > average(accounts, 'outgoings')
```

9.8.17. Iterators

Iterators are special functions able to iterate collections. For each collection item expression *expr* is evaluated. The current collection item is accessible in the expression using iterator variable *iter-name*, which is visible there only. The value of an iterator function call depends on the implementation of that function.

Syntax

```
fct-name(iter-name | expr)
```

Examples

```
accounts->pv:Children()->
  pv:ForAll(account | account->balanced = TRUE)
```

9.8.18. Accumulators

Accumulators are special functions able to iterate collections. For each collection item expression *expr* is evaluated and its value is assigned to the accumulator variable *acc-name*. The initial value of accumulator variable *acc-name* is the value of expression *acc-init-expr*. The current collection item is accessible in the expression using iterator variable *iter-name*. Both variables, *iter-name* and *acc-name*, are visible in expression *expr* only.

The value of an accumulator function call is the final value of the accumulator variable.

Syntax

```
fct-name(iter-name; acc-name = acc-init-expr | expr)
```

Examples

```
accounts->pv:Children()->
  pv:Iterate(account; sum = 0 | sum + account->deposit) > 0
```

9.8.19. Function Definitions

The DEF keyword defines a function with name *fct-name* and the given parameter list. The parameter names are visible only in expression *fct-body-expr*. The result of calling a so defined function is the value of the *fct-body-expr* calculated for the given argument list.

Function definitions can only occur before any other pvSCL expression and evaluate to TRUE if not followed by an expression. To avoid name conflicts it is recommended to use own namespaces for the function and parameter names (e.g. *my:fct-name* instead of *fct-name*, and *my:param-name* instead of *param-name*).

If not in a pvSCL code library, a so defined function is visible only in the constraint, restriction or calculation containing the function definition.

Syntax

```
DEF fct-name(param-name,param-name,...) = fct-body-expr ;
DEF fct-name(param-name,param-name,...) = fct-body-expr ;
...
expr
```

Examples

```
DEF min(x,y) = (IF x <= y THEN x ELSE y ENDIF);
DEF max(x,y) = (IF x >= y THEN x ELSE y ENDIF);
max(spare->x,spare->y) <= min(part->x,part->y)
```

9.8.20. Function Library

pv:Abs()

Get the absolute value of the context which must be a number.

Examples

```
(outside->temp - inside->temp)->pv:Abs() > 10
```

pv:AllChildren()

Get all children of the context which must be either a model, element, or attribute. Fail otherwise. All children of a model are the elements of the model, of an element are the elements of the sub-tree with this element as root (excluding this element), and of an attribute its attribute values.

Examples

```
self->pv:AllChildren()->
  pv:ForAll(child | NOT(child->pv:Selected) )
```

pv:Append(expr)

Append the value of *expr* to the context which must be a collection. It is an error if the type of the value is not compatible to the item type of the collection.

Examples `{1,2,3}->pv:Append(4) = {1,2,3,4}`

pv:AsSet()

Convert the context to a set. It is an error if the context does not have collection type. If the context has list type, all duplicate items of the list are removed.

Examples `{1,1,2,3}->pv:AsSet = {1,2,3}`

pv:Attribute(name)

Get the attribute with the given name. Fails if the context does neither have model nor element type, or no attribute with the name exists.

Examples `self->pv:Attribute('speed') = 100`

pv:Characters()

Get the characters of the context string as list.

Examples `'Text'->pv:Characters() = {'T','e','x','t'}`

pv:Child(index)

Get the child of the context with the given index. Fails if the context does neither have model, element, nor attribute type, or the index is invalid. The child of a model is an element, of an element an element, and of an attribute an attribute value.

Examples `self->pv:Child(0)->pv:Selected`

pv:Children()

Get the direct children of the context which must be either a model, element, or attribute. Fail otherwise. The children of a model is a list containing the root element of the model, of an element its child elements, and of an attribute its attribute values.

Examples `alternatives->pv:Children()->pv:Size() >
alternatives->ps:SelectedChildren()->pv:Size()`

pv:ChildrenByState(state), pv:ChildrenByState(state,selector)

Get all children of the context element with the given selection state and optionally given selector, as *ps:element[]*. Fails if the context does not have element type.

Examples `features->pv:ChildrenByState('ps:excluded','ps:user')->
pv:ForAll(element |
pv:Inform('User excluded element ' + element->pv:Name()))`

pv:Class()

Get the class of the context, as *ps:string*, which must be a configuration space, model, element, relation, attribute, or attribute value. Fails otherwise. The class of a configuration space is *ps:configspace*, of a model *ps:model*, of an element the element class, of a relation the relation class, of an attribute *ps:attribute*, and of an attribute value the type of the attribute value.

Examples `context->pv:Class() = 'ps:model'`

```
IMPLIES self->pv:Class() = 'ps:element'
```

pv:Collect(iterator)

Iterate the context collection and evaluate the iterator expression for each element of the collection. Return a new collection with all the evaluation results. The return type is *ps:list*.

Examples

```
products->pv:Children()->
  pv:Collect(p | IF p->stocked THEN 1 ELSE 0 ENDIF)->
  pv:Sum() > 50
```

pv:Element(name-or-id)

Get the element with the given unique name or identifier. If called on a model only elements in that model are considered. It is an error if the element does not exist or the function is called on anything else than a model.

Examples

```
Model->pv:Element('winter')->pv:Selected() = true
```

pv:DefaultSelected()

Check if the context element is selected by default. Fails if the context does not have element type.

Examples

```
radio->pv:DefaultSelected() AND speakers->number = 2
```

pv:Fail(message)

Show an error message. Always returns TRUE. Let the model evaluation fail.

Examples

```
doors->number = 2 OR
doors->number = 4 OR
pv:Fail('Invalid number of doors [' + doors->number + ''])
```

pv:Floor()

Get the largest (closest to positive infinity) integer value that is less than or equal to the context floating point number and is equal to a mathematical integer. Fails if the context does not have floating point number type. The return type is *ps:integer*.

Examples

```
3.5->pv:Floor = 3
```

pv:ForAll(iterator)

Iterate the context collection and evaluate the iterator expression for all items. Return FALSE if at least for one item the expression evaluates to FALSE.

Examples

```
bugs->pv:Children()->
  pv:ForAll(bug | bug->state = 'fixed')
```

pv:Get(), pv:Get(index)

Get the value of an attribute if the context is an attribute or attribute value, or return the input value. If an index is given and the context is an attribute, return the attribute value at that index, or fail if the index is invalid.

Examples

```
seasons->order->pv:Get(2) = 'autumn'
```

pv:HasAttribute(name)

Return TRUE if the attribute with the given name exists on the context model or element, FALSE otherwise. Fails if the context does not have model or element type.

Examples `self->pv:HasAttribute('speed') = true`

pv:HasElement(name-or-id)

Return TRUE if the element with the given name or identifier exists, FALSE otherwise. If called on a model only elements in that model are considered. It is an error if the function is called on anything else than a model.

Examples `Model->pv:HasElement('seasons') = true`

pv:HasModel(name-or-id)

Return TRUE if the model with the given name or identifier exists, FALSE otherwise.

Examples `pv:HasModel('Weather') = true`

pv:ID()

Get the unique identifier of the context, as *ps:string*, which must be a model, element, attribute, constant, or relation, or fail otherwise.

Examples `context->pv:ID() <> ''`

pv:IndexOf(sub-string)

Return the index (starting at 0) of the first occurrence of the given sub-string within the context string, or -1 if the sub-string was not found. It is an error if the context does not have string type. The resulting index has type *ps:integer*.

Examples `'Hello World'->pv:IndexOf('World') = 6`

pv:Inform(message)

Show an informational message. Always returns TRUE.

Examples `sportedition AND NOT(rearspoiler) RECOMMENDS
pv:Inform('Rear spoiler recommended for sport edition')`

pv:IsContainer()

Return TRUE if the context is a container, i.e. a collection like lists or sets.

Examples `self->pv:IsContainer() RECOMMENDS self->pv:Size() > 1`

pv:IsFixed()

Return TRUE if the context attribute has a fixed value. Fails if the context does not have attribute type.

Examples `self->pv:IsFixed() = TRUE`

pv:IsInheritable()

Return TRUE if the context attribute is inheritable. Fails if the context does not have attribute type.

Examples `self->pv:IsInheritable() = FALSE`

pv:IsKindOf(type)

Return TRUE if the type of the context object is the same as the type given as argument, or a type derived from it.

Examples `seasons->pv:IsKindOf('ps:feature') = TRUE`

pv:Item(index)

Get the item with the given index (starting at 0) of the context collection or the character with the given index of a string. Fail if the context does not have collection or string type, or the index is invalid.

Examples `seasons->pv:Children()->
pv:Item(0)->pv:Name() = 'spring'`

pv:Iterate(accumulator)

Iterate the context collection and return the value accumulated by evaluating the iterator expression for each element of the collection. The return type is that of the accumulated value.

Examples `pv:Inform('Current price is ' +
products->pv:SelectedChildren()->
pv:Iterate(product; price = 0 | price + product->price) + '$')`

pv:Max()

Return the maximal number of the context collection of numbers, or fail if the context is not a number collection. The return type is *ps:integer* or *ps:float* depending on the type of the collection. The maximal value of an empty collection is 0.

Examples `{1,2,3,4}->pv:Max() = 4`

pv:Min()

Return the minimal number of the context collection of numbers, or fail if the context is not a number collection. The return type is *ps:integer* or *ps:float* depending on the type of the collection. The minimal value of an empty collection is 0.

Examples `{1,2,3,4}->pv:Min() = 1`

pv:Model(), pv:Model(name-or-id)

Get the model, as *ps:model*, containing the context element, or the model with the given name or identifier if not called on an element. It is an error if the function is called on anything else than an element or configuration space.

Examples `NOT(context->pv:Model()->pv:RootElement())
IMPLIES pv:Fail('Root element of model ' +
context->pv:Model()->pv:Name() + ' must be selected')`

pv:Models(), pv:Models(type)

Get the models of a configuration space as *ps:model[]*. Optionally accepts a model type as argument to get only the models of a specific type. Fails if the context does not have configuration space type.

Examples `context->pv:Parent()->pv:Models()->pv:Size() > 0
pv:Models()->pv:Size() > 0`

pv:Name()

Get the name of the context, as *ps:string*, which must be a model, element, or attribute, or fail otherwise.

Examples `self->pv:SelectionState() = 'ps:nonselectable' IMPLIES`


```
pv:Warn('Feature ' + self->pv:Name() + ' is now non-selectable!')
```

pv:Parent()

Get the parent of the context, or fail if the context is not a model, element, relation, attribute, or attribute value. The parent of a model is the corresponding configuration space, of an element its parent element, or the corresponding model if it is the root element, of a relation the element on which the relation is defined, of an attribute the element on which the attribute is defined, and of an attribute value the attribute containing the value.

Examples

```
summer->pv:Parent()->pv:Name() = 'seasons'
```

pv:Prepend(expr)

Prepend the value of *expr* to the context which must be a collection. It is an error if the type of the value is not compatible to the item type of the collection.

Examples

```
{1,2,3}->pv:Prepend(4) = {4,1,2,3}
```

pv:Relations(), pv:Relations(type)

Get the relations of class *ps:dependencies* defined on the context element, as *ps:relation[]*. Optionally accepts the relation type as argument to get only relations of the given type. Fails if the context does not have element type.

Examples

```
specialedition->pv:Relations('my:extras')->
  pv:ForAll(r | re->pv:Targets()->pv:Size() <> 0)
```

pv:RootElement()

Get the root element of the context model, as *ps:element*. Fails if the context does not have model type.

Examples

```
context->pv:RootElement()->pv:Selected() = TRUE
```

pv:Round()

Return the closest integer of the context floating point number, or fail if the context does not have floating point number type. The context number is rounded to an integer by adding 0.5 and taking the floor of the result. The return type is *ps:integer*.

Examples

```
3.5->pv:Round() = 4
```

pv:Select(iterator)

Iterate the context collection and add all the collection items to the result list for which the iterator expression evaluates to TRUE. The return type is the type of the context collection.

Examples

```
customers->
  pv:Select(customer | customer->balanced = FALSE)->
  pv:ForAll(customer |
    pv:Inform('Send customer ' + customer->id + ' a reminder'))
```

pv:Selected()

Return TRUE if the context element or attribute exists in the variant, FALSE otherwise. Fails if the context does not have element or attribute type.

Examples

```
self EQUALS self->pv:Selected()
```

pv:SelectedChildren(), pv:SelectedChildren(type)

Get all children in the sub-tree of the context element that exist in the variant, as *ps:element[]*. Optionally accepts an element type as argument to get only child elements with the given type. Fails if the context does not have element type.

Examples

```
parts->pv:SelectedChildren('my:engine')->
pv:Size() = 1
```

pv:SelectionState()

Get the selection state of the context element, as *ps:string*. Fails if the context does not have element type. The selection state is one of *ps:selected*, *ps:excluded*, *ps:unselected*, or *ps:nonselectable*.

Examples

```
airbags->pv:SelectionState() = 'ps:excluded'
REQUIRES speed->max < 30
```

pv:Selector()

Get the selector of the context element, as *ps:string*. Fails if the context does not have element type. The selector is *ps:user* for user selections, *ps:auto* for selections caused by the auto resolver, *ps:mapped* for selections caused by mapped features, *ps:implicit* for implicitly selected elements, *ps:inherited* for inherited selections, or *none* for elements that neither are explicitly or implicitly selected nor excluded.

Examples

```
self IMPLIES self->pv:Selector() = 'ps:user'
OR pv:Inform('Feature ' + self->pv:Name() +
' was added automatically')
```

pv:Size()

Get the number of attribute values for attribute types, collection items for collection types, or characters for string types as *ps:integer*. For any other context type, 1 is returned.

Examples

```
seasons->pv:Children()->pv:Size() = 4 AND
seasons->pv:SelectedChildren()->pv:Size() = 1
```

pv:SubString(begin), pv:SubString(begin,end)

Return a new string, as *ps:string*, that is a sub-string of the context string. The sub-string begins at the specified *begin* index and extends to the *end-1* index or end of the context string. It is an error if the context does not have string type.

Examples

```
'Hello World'->pv:SubString(6) = 'World'
'smiles'->pv:SubString(1,5) = 'mile'
```

pv:Sum()

Return the sum of all numbers in the context collection, or fail if the context is not a number collection. The return type is *ps:integer* or *ps:float* depending on the type of the collection. The sum of an empty collection is 0.

Examples

```
{1,2,3,4}->pv:Sum() = 10
```

pv:Target(index)

Get the relation target with the given index of the context relation, as *ps:element*. Fails if the context does not have relation type.

Examples

```
self->pv:Target(0) XOR self->pv:Target(1)
```

pv:Targets()

Get the relation targets of the context relation, as *ps:element[]*. Fails if the context does not have relation type.

Examples

```
self->pv:Type() = 'ps:discourages' AND  
self->pv:Targets()->pv:ForAll(element |  
pv:Warn('You better deselect element ' + element->pv:Name()))
```

pv:ToFloat()

Convert the context number to a floating point number. Fails if the context does not have number type. The return type is *ps:float*.

Examples

```
1->pv:ToFloat() = 1.0
```

pv:ToLowerCase()

Convert all characters of the context string to lower case. Fails if the context does not have string type. The return type is *ps:string*.

Examples

```
'Hello'->pv:ToLowerCase() = 'hello'
```

pv:ToString()

Return a string representation of the context object. The return type is *ps:string*.

Examples

```
6->pv:ToString() = '6'
```

pv:ToUpperCase()

Convert all characters of the context string to upper case. Fails if the context does not have string type. The return type is *ps:string*.

Examples

```
'Hello'->pv:ToUpperCase() = 'HELLO'
```

pv:Type()

Get the type of the context as *ps:string*.

Examples

```
summer->pv:Type() = 'my:season'
```

pv:VariationType()

Get the variation type of the context element or attribute, as *ps:string*. Fails if the context does not have element or attribute type. The variation type of attributes always is *ps:mandatory*, and of elements *ps:mandatory*, *ps:optional*, *ps:or*, or *ps:alternative*.

Examples

```
summer->pv:VariationType() = 'ps:alternative'
```

pv:VName(), pv:VName(language)

Get the visible name of the context, as *ps:string*, which must be an element, or fail otherwise. Optionally the language can be specified.

If no language is give the visible name with no specified language will be returned. If no such visible name exists any other visible name will be returned. If no visible name is defined for the element an empty string is returned. If a language is specified the visible name in the given language will be returned if available. If no such visible name exists the function falls back to the version without given language.

Examples

```
self->pv:SelectionState() = 'ps:nonselectable' IMPLIES
```

```
pv:Warn('Feature ' + self->pv:VName() + ' is now non-selectable!')
```

pv:Warn(message)

Show a warning message. Always returns TRUE.

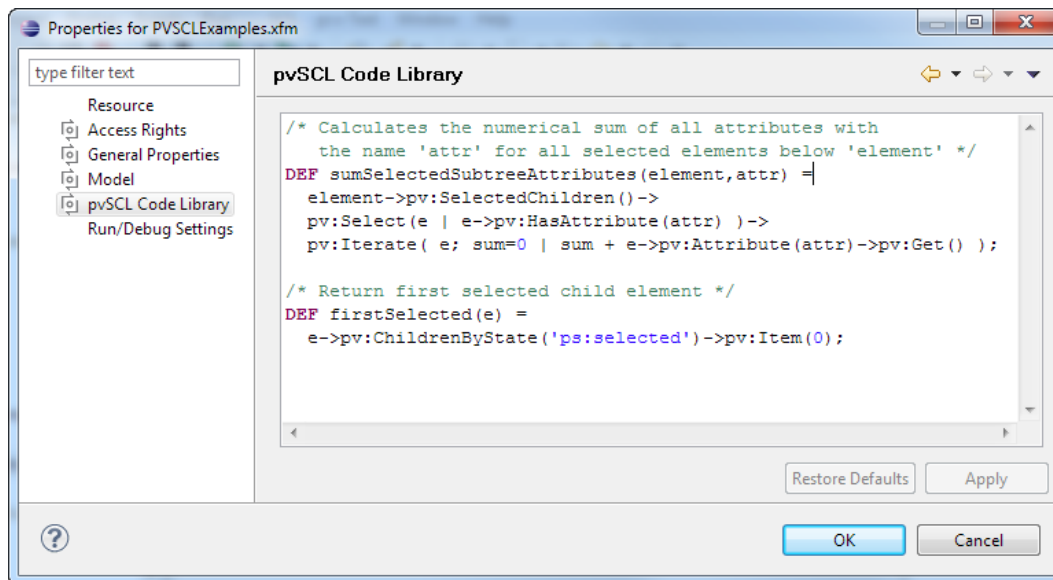
Examples

```
car->wheels > 4 IMPLIES
pv:Warn('Too many wheels (' + car->wheels + ') configured')
```

9.8.21. User-Defined pvSCL Functions

For complex restrictions and calculations it may be useful to provide additional functions, e.g. to simplify the expressions or to share code. For the expression language *pvSCL* a code library can be defined in each model. This is done by entering the code into the *pvSCL Code Library* properties page of a model (see [Figure 9.2, “pvSCL Code Library Model Property Page”](#)).

Figure 9.2. pvSCL Code Library Model Property Page



Each feature or family model in a Configuration Space can define code libraries. Code defined in one model is also available in all other models of the same configuration space. Defining the same function in more than one model, will redefine the function. Since there is no explicit model loading order the used version of the function may differ.

9.9. XSLT Extension Functions

Several extension functions are available when using the XSLT processor integrated in the pure::variants XML Transformation System for model transformations and model exports. These extension functions are defined in own namespaces. Before they can be used in an XSLT script, the corresponding namespaces have to be included using the "xmlns" stylesheet attribute:

```
<?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"
  extension-element-prefixes="pv">

  ...any script content...

</xsl:stylesheet>
```

After including the namespace, the extension functions can be used in XPath expressions using the defined namespace prefix, e.g. `pv:hasFeature('F')`.

The following extension functions are defined in the namespace "http://www.pure-systems.com/purevariants" and provide access to the pure::variants model information.

Table 9.17. Extension functions providing model information

Function	Description
<code>nodeset models()</code>	Get all input models known to the transformer, i.e. the opened variant description model, and all Feature and Family Models of the Configuration Space without any modifications. See Section 5.9.2, “Variant Result Models” for more information about the transformation input. Note: In the pure::variants Server Edition this function returns an empty set. Access to the input models of the transformation is not supported in the pure::variants Server Edition.
<code>nodeset model-by-id(string)</code>	Get all variant Result Models known to the transformer having the given id. The Result Models are derived from the models of the Configuration Space describing a single concrete solution drawn from the solution family. See Section 5.9.2, “Variant Result Models” for more information about the transformation input.
<code>nodeset model-by-name(string)</code>	Get all Variant Result Models known to the transformer having the given name. The Variant Result Models are derived from the models of the Configuration Space describing a single concrete solution drawn from the solution family. See Section 5.9.2, “Variant Result Models” for more information about the transformation input.
<code>nodeset model-by-type(string)</code>	Get all Variant Result Models known to the transformer having the given type. The Variant Result Models are derived from the models of the Configuration Space describing a single concrete solution drawn from the solution family. Valid types are <i>ps:vdm</i> , <i>ps:cfm</i> , and <i>ps:ccm</i> . See Section 5.9.2, “Variant Result Models” for more information about the transformation input.
<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>nodeset getChildrenTargets(string id)</code>	Return the full qualified ids of the children elements of the element with the given id.
<code>nodeset getChildrenTargets(nodeset element)</code>	Return the full qualified ids of the children elements of the given element.
<code>nodeset getChildrenTargets(string ename,string eclass,string etype?)</code>	Return the full qualified ids of the children elements of 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.

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

The following extension functions are defined in the namespace "http://www.pure-systems.com/xmlts" and provide basic information about the current transformation.

Table 9.18. Extension functions providing transformation information

Function	Description
<code>string os()</code>	Get the target system type. This is either the string "win32", "macosx", or "linux" (default).
<code>string version()</code>	Get the transformation system version.
<code>string input-path()</code>	Get the transformation input path.
<code>string output-path()</code>	Get the transformation output path.
<code>string generate-id()</code>	Generate an unique identifier.
<code>nodeset current()</code>	Get the node currently being transformed.

Function	Description
<code>nodeset entry-points()</code>	Get the transformation entry point list, i.e. a list of full qualified element IDs. Transformation modules can use this list to identify sub-trees of the input models that are to be transformed.
<code>boolean below-entry-point(string id)</code>	Return <i>true</i> if the given full qualified element ID denotes an element below a transformation entry point. Transformation modules can use this function to identify sub-trees of the input models that are to be transformed.
<code>nodeset exit-points()</code>	Get the transformation exit point list, i.e. a list of full qualified element IDs. Transformation modules can use this list to identify sub-trees of the input models that are to be ignored.
<code>boolean above-exit-point(string id)</code>	Return <i>true</i> if the given full qualified element ID denotes an element above a transformation exit point. Transformation modules can use this function to identify sub-trees of the input models that are to be ignored.
<code>nodeset results-for(nodeset nodes?)</code>	Get the transformation module results for the given nodes. If no argument is given, then the results for the context node are returned.
<code>nodeset log(string message,number level?)</code>	Add a logging message that is shown in the Console View. The first parameter is the message and the second the logging level (0-9). It is recommend to use a logging level between 4 (default) and 8 (detailed tracing). Returns the empty nodeset.
<code>nodeset info(string message,string id?,nodeset related?)</code>	Add an info message that is shown in the Problems View resp. as marker on a model element. The first parameter is the message. All other parameters are optional. The second is the ID of the context element of the info (used to place the marker), and the third is a set of IDs of related model elements. Returns the empty nodeset.
<code>nodeset warning(string message,string id?,nodeset related?)</code>	Add a warning message that is shown in the Problems View resp. as marker on a model element. The first parameter is the message. All other parameters are optional. The second is the ID of the context element of the info (used to place the marker), and the third is a set of IDs of related model elements. Returns the empty nodeset.
<code>nodeset error(string message,string id?,nodeset related?)</code>	<p>Add an error message that is shown in the Problems View resp. as marker on a model element. The first parameter is the message. All other parameters are optional. The second is the ID of the context element of the info (used to place the marker), and the third is a set of IDs of related model elements. Returns the empty nodeset.</p> <p>Note</p> <p>Error messages may abort the XSLT script execution and the whole transformation.</p>

Table 9.19. Extension elements for logging and user messages

Element	Description
<code><log level="0-9">message</log></code>	Add a logging message that is shown in the Console View. The optional attribute "level" specifies the logging level (0-9). It is recommend to use a logging level between 4 (default) and 8 (detailed tracing).
<code><info context="element id" related="nodeset">message</info></code>	Add an info message that is shown in the Problems View resp. as marker on a model element. The optional attribute "context" specifies the ID of the context element of the info (used to place the marker). The optional attribute "related" specifies a set of IDs of related model elements.

Element	Description
<pre><warning context="element id" related="nodeset">message</ warning></pre>	<p>Add a warning message that is shown in the Problems View resp. as marker on a model element. The optional attribute "context" specifies the ID of the context element of the info (used to place the marker). The optional attribute "related" specifies a set of IDs of related model elements.</p>
<pre><error context="element id" related="nodeset">message</error></pre>	<p>Add an error message that is shown in the Problems View resp. as marker on a model element. The optional attribute "context" specifies the ID of the context element of the info (used to place the marker). The optional attribute "related" specifies a set of IDs of related model elements.</p> <p>Note</p> <p>Error messages may abort the XSLT script execution and the whole transformation.</p>

The following extension functions are defined in the namespace "http://www.pure-systems.com/path" and provide additional file operations.

Table 9.20. Extension functions providing file operations

Function	Description
<code>string normalize(string path)</code>	Normalized the given path for the current target platform.
<code>string dirname(string path)</code>	Get the directory part of the given path.
<code>string filename(string path)</code>	Get the file part of the given path.
<code>string basename(string path)</code>	Strip the file extension from the given path.
<code>string extension(string path)</code>	Get the file extension from the given path.
<code>string absolute(string path)</code>	Make the given path absolute (i.e. full path).
<code>string add-part(string path,string part)</code>	Add the given part to the path using the platform specific path delimiter.
<code>number size(string file)</code>	Get the size (in bytes) of the given file.
<code>number mtime(string path)</code>	Get the modification time of the given file or directory.
<code>string cwd()</code>	Get the current working directory.
<code>string tempdir()</code>	Get the directory for temporary files.
<code>string delimiter()</code>	Get the path delimiter of the target platform.
<code>boolean exists(string path)</code>	Return <i>true</i> if the given file or directory exists.
<code>boolean is-dir(string path)</code>	Return <i>true</i> if the given path points to a directory.
<code>boolean is-file(string path)</code>	Return <i>true</i> if the given path points to a file.
<code>boolean is-absolute(string path)</code>	Return <i>true</i> if the given path is absolute (i.e. full path).
<code>string to-uri(string path)</code>	Get the file URI build from the given path (i.e. <code>file:///...</code>).
<code>string read-file(string uri)</code>	Read a file from a given URI and return its content as string.

The following extension functions are defined in the namespace "http://www.pure-systems.com/string" and provide additional string operations.

Table 9.21. Extension functions providing string operations

Function	Description
<code>nodeset parse(string xml)</code>	Parse the given string as XML and return the resulting node set.
<code>boolean matches(string str,string pattern)</code>	Match the regular expression pattern against the given string. Return <i>true</i> if the pattern matches.
<code>nodeset match(string str,string pattern)</code>	Match the regular expression pattern against the given string and return the set of sub-matches.
<code>string submatch(string str,string pattern,number n)</code>	Match the regular expression pattern against the given string and return the n-th sub-match.
<code>string replace(string str,string pattern,string replacement,number n?)</code>	Replace the matches in the given string with the replacement string using the regular expression match pattern. The optional fourth parameter specifies the maximal number of replacements. 0 means all, 1 means to replace only the first, 2 means to replace the first 2 matches etc. Returns the resulting string.
<code>string expand(string str)</code>	Expand variables in the given string and return the expanded string. Variables are recognized by the following pattern: <code>\$(VARIABLENAME)</code> . See Section 9.10, “Predefined Variables” for the list of supported variables.

Further information about XSLT extension functions is available in the external document *XML Transformation System*.

9.10. Predefined Variables

There are several places in `pure::variants` where variables are supported. That are for instance the transformation input and output paths as well as in the parameters of transformation modules. The following pattern is used for accessing variables: `$(VARIABLENAME)`.

Table 9.22. Supported Variables

Variable	Description
INPUT	Transformation input directory.
OUTPUT	Transformation output directory.
CONFIGSPACE	Path to the Configuration Space folder.
PROJECT	Path the folder of the current project.
PROJECT:name	Path to the folder of the project with the given name.
WORKSPACE	Path the workspace folder.
MODULEBASE	Path the transformation module base folder.
VARIANT	Name of the current variant, i.e. the name of the VDM currently being evaluated resp. transformed.
VARIANTSPATH	Name of the currently being evaluated resp. transformed VDM prefixed by the names of the parent VDMs. The names are separated by a slash. If a VDM is not linked, then the value of <code>VARIANTSPATH</code> is identical to the value of <code>VARIANT</code> .

9.11. Regular Expressions

Regular expressions are used to match patterns against strings.

9.11.1. Characters

Within a pattern, all characters except `.`, `|`, `(`, `)`, `[`, `{`, `+`, `\`, `^`, `$`, `*`, and `?` match themselves. If you want to match one of these special characters literally, precede it with a backslash.

Patterns for matching single characters:

<code>x</code>	Matches the character <code>x</code> .
<code>\</code>	Matches nothing, but quotes the following character.
<code>\\</code>	Matches the backslash character.
<code>\On</code>	Matches the character with octal value <code>On</code> ($0 \leq n \leq 7$).
<code>\Onn</code>	Matches the character with octal value <code>Onn</code> ($0 \leq n \leq 7$).
<code>\Omnn</code>	Matches the character with octal value <code>Omnn</code> ($0 \leq m \leq 3$, $0 \leq n \leq 7$).
<code>\xhh</code>	Matches the character with hexadecimal value <code>0xhh</code> .
<code>\uhhhh</code>	Matches the character with hexadecimal value <code>0xhhhh</code> .
<code>\t</code>	Matches the tab character (<code>"\u0009"</code>).
<code>\n</code>	Matches the newline (line feed) character (<code>"\u000A"</code>).
<code>\r</code>	Matches the carriage-return character (<code>"\u000D"</code>).
<code>\f</code>	Matches the form-feed character (<code>"\u000C"</code>).
<code>\a</code>	Matches the alert (bell) character (<code>"\u0007"</code>).
<code>\e</code>	Matches the escape character (<code>"\u001B"</code>).
<code>\cx</code>	Matches the control character corresponding to <code>x</code> .

To match a character from a set of characters the following character classes are supported. A character class is a set of characters between brackets. The significance of the special regular expression characters `.`, `|`, `(`, `)`, `[`, `{`, `+`, `^`, `$`, `*`, and `?` is turned off inside the brackets. However, normal string substitution still occurs, so (for example) `\b` represents a backspace character and `\n` a newline. To include the literal characters `]` and `-` within a character class, they must appear at the start.

<code>[abc]</code>	Matches the characters <code>a</code> , <code>b</code> , or <code>c</code> .
<code>[^abc]</code>	Matches any character except <code>a</code> , <code>b</code> , or <code>c</code> (negation).
<code>[a-zA-Z]</code>	Matches the characters <code>a</code> through <code>z</code> or <code>A</code> through <code>Z</code> , inclusive (range).
<code>[a-d[m-p]]</code>	Matches the characters <code>a</code> through <code>d</code> , or <code>m</code> through <code>p</code> : <code>[a-dm-p]</code> (union).
<code>[a-z&&[def]]</code>	Matches the characters <code>d</code> , <code>e</code> , or <code>f</code> (intersection).
<code>[a-z&&[^bc]]</code>	Matches the characters <code>a</code> through <code>z</code> , except for <code>b</code> and <code>c</code> : <code>[ad-z]</code> (subtraction).
<code>[a-z&&[^m-p]]</code>	Matches the characters <code>a</code> through <code>z</code> , and not <code>m</code> through <code>p</code> : <code>[a-lq-z]</code> (subtraction).

Predefined character classes:

<code>.</code>	Matches any character.
<code>\d</code>	Matches a digit: <code>[0-9]</code> .

`\D` Matches a non-digit: `[^0-9]`.

`\s` Matches a whitespace character: `[\t\n\x0B\f\r]`.

`\S` Matches a non-whitespace character: `[^\s]`.

`\w` Matches a word character: `[a-zA-Z_0-9]`.

`\W` Matches a non-word character: `[^\w]`.

POSIX character classes (US-ASCII):

`\p{Lower}` Matches a lower-case alphabetic character: `[a-z]`.

`\p{Upper}` Matches an upper-case alphabetic character: `[A-Z]`.

`\p{ASCII}` Matches all ASCII characters: `[\x00-\x7F]`.

`\p{Alpha}` Matches an alphabetic character: `[\p{Lower}\p{Upper}]`.

`\p{Digit}` Matches a decimal digit: `[0-9]`.

`\p{Alnum}` Matches an alphanumeric character: `[\p{Alpha}\p{Digit}]`.

`\p{Punct}` Matches a punctuation character: one of `!"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~`

`\p{Graph}` Matches a visible character: `[\p{Alnum}\p{Punct}]`.

`\p{Print}` Matches a printable character: `[\p{Graph}]`.

`\p{Print}` Matches a space or a tab: `[\t]`.

`\p{Cntrl}` Matches a control character: `[\x00-\x1F\x7F]`.

`\p{XDigit}` Matches a hexadecimal digit: `[0-9a-fA-F]`.

`\p{Space}` Matches a whitespace character: `[\t\n\x0B\f\r]`.

Classes for Unicode blocks and categories:

`\p{InGreek}` Matches a character in the Greek block (simple block).

`\p{Lu}` Matches an uppercase letter (simple category).

`\p{Sc}` Matches a currency symbol.

`\P{InGreek}` Matches any character except one in the Greek block (negation).

`[\p{L}&&[^\p{Lu}]]` Matches any letter except an uppercase letter (subtraction).

9.11.2. Character Sequences

Character sequences are matched by string the characters together.

`XY` Matches X followed by Y.

The following constructs are used to easily match character sequences containing special characters.

`\Q` Quotes all characters until `\E`.

`\E` Ends quoting started by `\Q`.

9.11.3. Repetition

Repetition modifiers allow to match multiple occurrences of a pattern.

X?	Matches X once or not at all.
X*	Matches X zero or more times.
X+	Matches X one or more times.
X{n}	Matches X exactly n times.
X{n,}	Matches X at least n times.
X{n,m}	Matches X at least n but not more than m times.

These patterns are greedy, i.e. they will match as much of a string as they can. This behavior can be altered to let them match the minimum by adding a question mark suffix to the repetition modifier.

9.11.4. Alternation

An unescaped vertical bar "|" matches either the regular expression that precedes it or the regular expression that follows it.

X|Y Matches either X or Y.

9.11.5. Grouping

Parentheses are used to group terms within a regular expression. Everything within the group is treated as a single regular expression.

(X) Matches X.

9.11.6. Boundaries

The following boundaries can be specified.

^	Matches the beginning of a line.
\$	Matches the end of a line.
\b	Matches a word boundary.
\B	Matches a non-word boundary.
\A	Matches the beginning of the string.
\G	Matches the end of the previous match.
\Z	Matches the end of the string but for the final terminator (e.g newline), if any.
\z	Matches the end of the string.

9.11.7. Back References

Back references allow to use part of the current match later in that match, i.e. to look for various forms of repetition.

\n Whatever the n-th group matched.

9.12. Keyboard Shortcuts

Some of the following keyboard shortcuts may not be supported on all operating systems.

Table 9.23. Common Keyboard Shortcuts

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

Table 9.24. Model Editor Keyboard Shortcuts

Key	Action
ENTER	Show properties dialog
DEL / ENTF	Delete selected elements
Up/Down cursor keys	Navigate tree
Left/Right cursor keys	Collapse or expand subtree
CTRL+O	Open Quick-Outline

Table 9.25. Graph Editor Keyboard Shortcuts

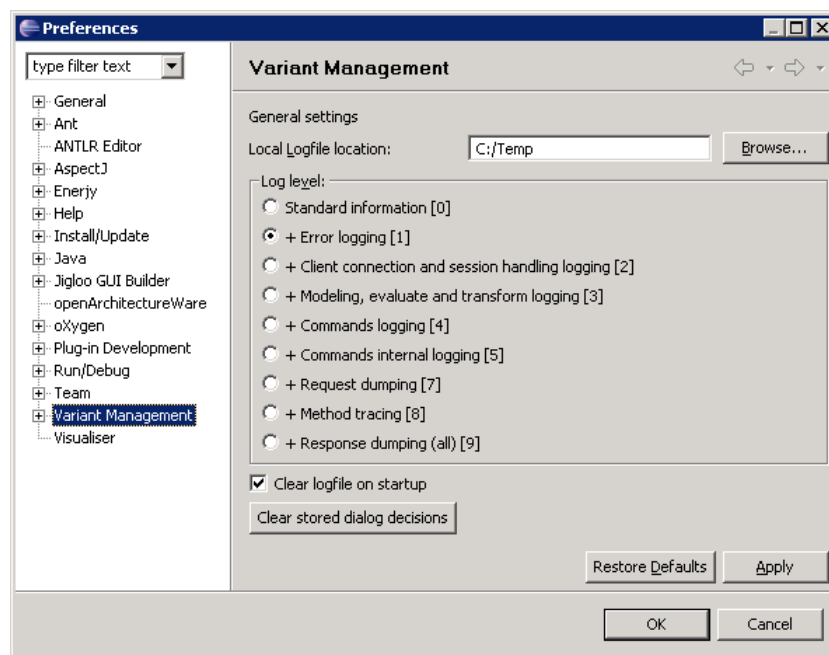
Key	Action
CTRL+P	Print graph
CTRL+=	Zoom in
CTRL+-	Zoom out
CTRL+ALT+A	Show relation arrows in graph
CTRL+ALT+X	Expand complete subtrees of selected elements
ALT+X	Expand one level of selected elements
ALT+C	Collapse selected elements
ALT+H	Layout graph horizontal
ALT+V	Layout graph vertical
ALT+DEL	Hide selected elements

Chapter 10. Appendices

10.1. Software Configuration

pure::variants may be configured from the configuration page (located in Window->Preferences->Variant Management). 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 10.1. The configuration dialog of pure::variants



10.2. User Interface Advanced Concepts

10.2.1. 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.

10.3. Glossary

Configuration Space

The Configuration Space describes the set of Input Models for creating product variants. It also defines the transformation of variants.

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.
Family Model	This model type is used to describe how the products in a product line will be assembled or generated from pre-specified components. Each component in a 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. Family models are described in more detail in Section 5.4, “Family Models” .
Family Model Editor	The editor for Family Models. See Section 7.3.3, “Family Model Editor” for a detailed description.
Matrix Editor	The editor for Configuration Spaces. See Section 7.3.7, “Matrix Editor” for a detailed description.
Feature Model	This model type is used to 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 Section 5.3, “Feature Models” .
Feature Model Editor	The editor for Feature Models. See Section 7.3.2, “Feature Model Editor” for a detailed description.
HTML	Hyper Text Markup Language.
Input Model	Input Models are the Feature and Family Models of a Configuration Space. They are added to a Configuration Space using the Configuration Space properties dialog. See Figure 6.9, “Configuration Space properties: Model Selection” for more information.
Link Element	Elements in models that represent links to VDMs or Configuration Spaces to create a variant hierarchy. See Section 6.2.1, “Hierarchical Variant Composition” for a detailed description.
Model Rank	The model rank is a positive integer that is used to control the order in which the models of a Configuration Space are evaluated. Models are evaluated from higher to lower ranks, i.e. models with rank 1 (highest) are evaluated before models with rank 2 or lower. The rank of a model is specific to a Configuration Space 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.
pvSCL	pure::variants Simple Constraint Language. A simple language to express constraints, restrictions and calculations.
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)
Variant Description Model	This model type is used to describe the set of features of a single product in the product line. Taking the Input Models of a Configuration Space and making choices where there is variability in the Input Models creates these models. VDMs are described in more detail in Section 5.5, “Variant Description Models” .
Variant Result Model	This model is the result of evaluating the input models of a Configuration Space according to a given element selection (VDM). It represents a specific variant of the input models and is used as the input for the transformation. See Section 5.9.2, “Variant Result Models” for a detailed description.
VDM	Abbreviation of Variant Description Model.
VDM Editor	The editor for the pure::variants Variant Description Model. See Section 7.3.4, “Variant Description Model Editor” for detailed information about it.
VRM Editor	The editor for Variant Result Models. See Section 7.3.5, “Variant Result Model Editor” for a detailed description.
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 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 .

Index

A

Attribute

- Calculation, 24, 24
- Element, 22
- Feature, 26
- Hide, 85
- List Attribute, 23, 23
- Set Attribute, 23, 23
- Value, 24
- Value Types, 23, 117
 - ps:boolean, 117
 - ps:class, 117
 - ps:datetime, 117
 - ps:directory, 117
 - ps:element, 117
 - ps:feature, 117
 - ps:filetype, 117
 - ps:float, 117
 - ps:html, 117
 - ps:insertionmode, 117
 - ps:integer, 117
 - ps:path, 117
 - ps:string, 117
 - ps:url, 117

Attribute Overriding

- Variant Description Model, 101

Attributes

- Editor, 91
- View, 104

Auto Resolver

- Variant Description Model, 39

C

Calculations

- Editor, 94
- pvProlog, 127

Compare

- Model, 63
- Models, 103

Configuration Space

- Transformation, 44

Constraints

- Editor, 94
- Editor Pages, 85
- Model, 22

D

Default Selected

- Element Properties, 38, 90

Dialog

- Element Selection, 96

E

Editor

Attributes, 91

Calculations, 94

Common Pages, 84

Configuration Space, 44

Constraints, 94

Family Model, 100

Feature Model, 97

Filter, 68

Metrics, 69

Quick Overview, 68

Relations, 90

Restrictions, 94

Variant Description Model, 100

Variant Result Model, 102

Editor Pages

Constraints, 85

Graph, 86

Table, 85

Tree, 84

Element

Attribute, 22

Calculation, 24, 24

Constraints, 22

Default Selection State, 38

Restrictions, 22

Selection Dialog, 96

Variation Types, 119

Element Properties

Attributes Page, 91

Constraints Page, 93

Dialog, 89

General Page, 89

Relations Page, 90

Restrictions Page, 93

Element Selection

Variant Description Model, 100

Element Variation Types

Alternative, 119

Mandatory, 119

Optional, 119

Or, 119

Evaluation, 37

Prolog Code Library, 136

pvSCL Code Library, 152

Variant Description Model, 31

Export

Model, 72

Expression Editor, 94

F

Family Model, 26

Editor, 100

Element Variation Types, 119

Part Element Types, 125

ps:class, 126

ps:classalias, 126

ps:feature, 127

ps:flag, 126

- ps:variable, 127
- Restrictions, 28
- Source Element Types, 119
 - ps:classaliasfile, 124
 - ps:condtext, 123
 - ps:condxml, 121
 - ps:dir, 120
 - ps:file, 120
 - ps:flagfile, 124
 - ps:fragment, 121
 - ps:makefile, 124
 - ps:symlink, 125
 - ps:transform, 121
- Feature
 - Attributes, 26
 - Constraints, 22
 - Relations, 22
 - Restrictions, 22
- Feature Model, 25
 - Editor, 97
 - Element Variation Types, 119
- Features
 - Matrix Editor, 103
- Filter
 - Model, 68

G

- Graph Visualization
 - Editor Pages, 86

H

- Hierarchical Variant Composition, 30, 41

I

- Import
 - Model, 77

K

- Keyboard Shortcuts, 160

L

- Language Support, 71
- List Attribute, 23, 23

M

- Metrics
 - Model, 69
- Model
 - Common Properties, 112
 - Compare, 63, 103
 - Constraints, 22
 - Export, 72
 - Family, 26
 - Feature, 25
 - Filter, 68
 - General Properties, 113
 - Import, 77

- Meta Attributes, 134
- Metrics, 69
- Prolog Code Library, 136
- Properties, 112
- pvSCL Code Library, 152
- Search, 65
- Validation, 58
- Variant Description, 30
- Variant Result, 34
- Model Meta Attributes
 - author, 134
 - date, 134
 - dir, 134
 - file, 134
 - name, 134
 - time, 134
 - version, 134
- Multiple
 - Transformation, 57

O

- Outline
 - View, 107
- Outline View
 - Variant Description Model, 101

P

- Problems
 - View, 107
- Projects
 - View, 111
- Properties
 - View, 107
- pvProlog
 - advanced examples, 134
 - Calculations, 127
 - Code Library, 136
 - element references, 128
 - parent, 128
 - Restrictions, 127
- pvProlog Functions, 129
 - alternativeChild, 132
 - checkMax, 133
 - checkMin, 133
 - checkRange, 133
 - errorMsg, 132
 - false, 129
 - getAllChildren, 133
 - getAllSelectedChildren, 133
 - getAttribute, 130
 - getAttributeId, 130
 - getAttributeName, 130
 - getAttributeType, 130
 - getContext, 130
 - getElementChildren, 131
 - getElementClass, 131
 - getElementModel, 131

getElementName, 131
 getElementParents, 131
 getElementRoot, 131
 getElementType, 131
 getElementVisibleName, 131
 getMatchingAttributes, 131
 getMatchingElements, 131
 getMatchingSelectedElements, 133
 getModelList, 131
 getSelf, 130
 getVariantContext, 131
 getVariantContextId, 131
 getVariantContextName, 131
 getVariantId, 131
 has, 132
 hasAttribute, 130
 hasComponent, 133
 hasElement, 132
 hasFeature, 132
 hasPart, 133
 hasSource, 133
 infoMsg, 132
 isElement, 130
 isFalse, 130
 isFamilyModelElement, 130
 isFeatureModelElement, 130
 isTrue, 130
 isVariant, 131
 not, 129
 singleSubfeature, 132
 subfeatureCount, 132
 subnodeCount, 132
 sumSelectedSubtreeAttributes, 133
 true, 129
 userMessage, 132
 warningMsg, 132
 pvProlog Operators, 129
 and, 129
 equiv, 129
 implies, 129
 or, 129
 xor, 129
 pvSCL
 Code Library, 152
 pvSCL Functions
 pv:Abs, 144
 pv:AllChildren, 144
 pv:Append(expr), 144
 pv:AsSet, 145
 pv:Attribute(name), 145
 pv:Characters(), 145
 pv:Child(index), 145
 pv:Children, 145
 pv:ChildrenByState(state),
 pv:ChildrenByState(state,selector), 145
 pv:Class, 145
 pv:Collect(iterator), 146
 pv:DefaultSelected, 146
 pv:Element(name-or-id), 146
 pv:Fail(message), 146
 pv:Floor, 146
 pv:ForAll(iterator), 146
 pv:Get, pv:Get(index), 146
 pv:HasAttribute(name), 146
 pv:HasElement(name-or-id), 147
 pv:HasModel(name-or-id), 147
 pv:ID, 147
 pv:IndexOf(sub-string), 147
 pv:Inform(message), 147
 pv:IsContainer, 147
 pv:IsFixed, 147
 pv:IsInheritable, 147
 pv:IsKindOf(type), 147
 pv:Item(index), 148
 pv:Iterate(accumulator), 148
 pv:Max, 148
 pv:Min, 148
 pv:Model, pv:Model(name-or-id), 148
 pv:Models, pv:Models(type), 148
 pv:Name, 148
 pv:Parent, 149
 pv:Prepend(expr), 149
 pv:Relations, pv:Relations(type), 149
 pv:RootElement, 149
 pv:Round, 149
 pv:Select(iterator), 149
 pv:Selected, 149
 pv:SelectedChildren, pv:SelectedChildren(type),
 149
 pv:SelectionState, 150
 pv:Selector, 150
 pv:Size, 150
 pv:SubString(begin), pv:SubString(begin,end), 150
 pv:Sum, 150
 pv:Target(index), 150
 pv:Targets, 150
 pv:ToFloat, 151
 pv:ToLowerCase, 151
 pv:ToString, 151
 pv:ToUpperCase, 151
 pv:Type, 151
 pv:VariationType, 151
 pv:VName, 151
 pv:Warn(message), 152

R

Refactoring, 62
 Regular Expressions, 157
 Relation Types
 ps:conditionalRequires, 117
 ps:conflicts, 118
 ps:conflictsAny, 118
 ps:defaultProvider, 118
 ps:discourages, 118
 ps:discouragesAny, 118
 ps:exclusiveProvider, 118

- ps:expansionProvider, 118
- ps:influences, 118
- ps:provides, 118
- ps:recommendedFor, 117
- ps:recommendedForAll, 117
- ps:recommends, 117
- ps:recommendsAll, 117
- ps:requestsProvider, 118
- ps:requiredFor, 117
- ps:requiredForAll, 117
- ps:requires, 117
- ps:requiresAll, 117
- ps:sharedProvider, 118
- ps:supports, 117

Relations

- Editor, 90
- Feature, 22
- View, 108

Restrictions

- Editor, 94
- Element, 22
- Family Model, 28
- pvProlog, 127

Result

- Delta Mode, 110
- View, 109

S

Search, 65

- Model, 65
- Quick Overview, 68
- View, 106

Set Attribute, 23, 23

T

Tasks

- View, 107

Transformation, 44

- JavaScript, 52
- Regular Expression, 50
- Standard Transformation, 48
- Variant Description Model, 33
- Variant Result Model, 34
- XSLT Extension Functions, 152
- XSLT Transformation, 53

Type Model, 70

V

Validation

- Models, 58

Variables, 157

- CONFIGSPACE, 157
- INPUT, 157
- MODULEBASE, 157
- OUTPUT, 157
- PROJECT, 157
- VARIANT, 157

VARIANTSPATH, 157

WORKSPACE, 157

Variant

- Matrix Editor, 103

Variant Description Model, 30

- Auto Resolver, 39
- Editor, 100
- Evaluation, 31
- Inheritance, 30, 114
- Load Selection, 44
- Outline, 101
- Selection Types, 119
 - Auto, 119
 - Auto Excluded, 119
 - Excluded, 119
 - Implicit, 119
 - Mapped, 119
 - Non-Selectable, 119
 - User, 119

- Transformation, 33

Variant Projects

- View, 111

Variant Result Model

- Editor, 102
- Transformation, 34

Views

- Attributes, 104
- Matrix Editor, 103
- Outline, 107
- Problems, 107
- Properties, 107
- Relations, 108
- Result, 109
- Search, 106
- Tasks, 107
- Variant Projects, 111
- Visualization, 105

Visualization

- View, 105

X

XSLT Elements

- error, 156
- info, 155
- log, 155
- warning, 156

XSLT Extension Functions, 152

XSLT Functions

- above-exit-point, 155
- absolute, 156
- add-part, 156
- basename, 156
- below-entry-point, 155
- current, 154
- cwd, 156
- delimiter, 156
- dirname, 156
- entry-points, 155

error, 155
exists, 156
exit-points, 155
expand, 157
extension, 156
filename, 156
generate-id, 154
getAttribute, 154
getAttributeValue, 154
getChildrenTargets, 153
getElement, 153
hasAttribute, 153
hasAttributeValue, 154
hasComponent, 153
hasElement, 153
hasFeature, 153
hasPart, 153
hasSource, 153
info, 155
input-path, 154
is-absolute, 156
is-dir, 156
is-file, 156
log, 155
match, 157
matches, 157
model-by-id, 153
model-by-name, 153
model-by-type, 153
models, 153
mtime, 156
normalize, 156
os, 154
output-path, 154
parse, 157
read-file, 156
replace, 157
results-for, 155
size, 156
submatch, 157
tempdir, 156
to-uri, 156
version, 154
warning, 155
