pure::variants User's Guide

Parametric Technology GmbH

pure::variants User's Guide

Version 7.0.0.685 for pure::variants 7.0

Publication date 2025

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1. Introduction

1.1. What is pure::variants?

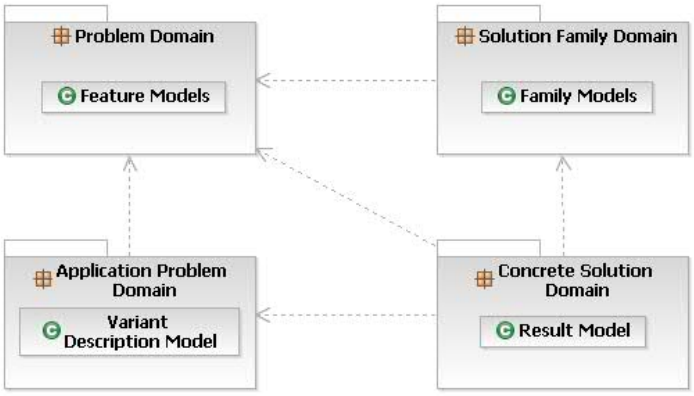
pure::variants provides a set of integrated tools to support each phase of the software product-line development process. pure::variants has also been designed as an open framework that integrates with other tools and types of data such as requirements management systems, object-oriented modeling tools, 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 .

pure::variants uses open source libraries. The list of used libraries is available here .

Chapter 2. Software and License Installation

2.1. Software Requirements

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

2.2. Software Installation

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

2.3. Obtaining and Installing a License

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

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.

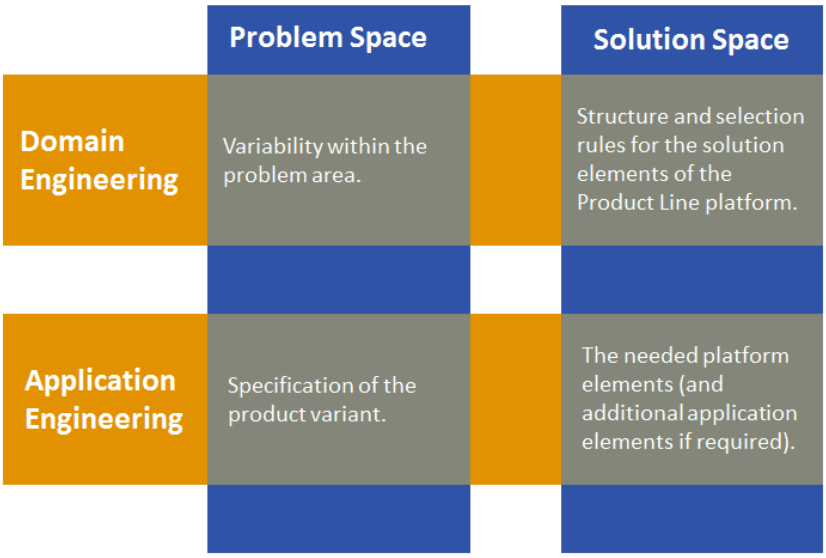
3.2. Software Product Lines

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).

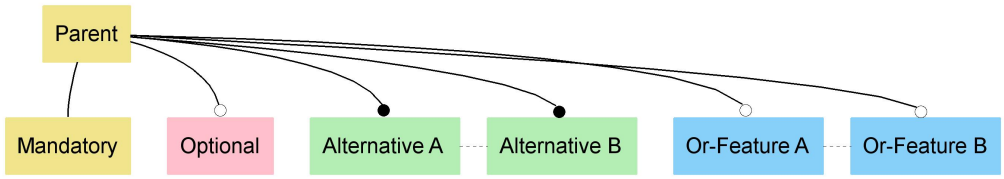
3.3. Modelling the Problem Space with Feature Models

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”.

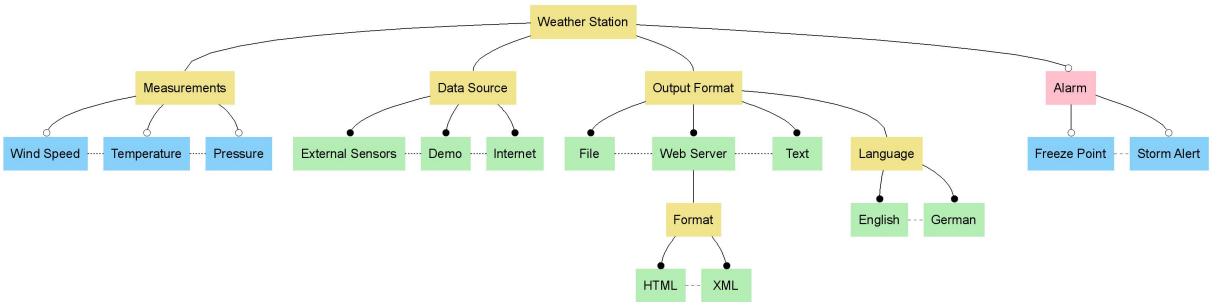
When specifying which features are to be included in a variant the following rules apply: If a parent feature is contained in a variant, all its mandatory child features must be also contained ("n from n"), any number of optional features can be included ("m from n, 0 < = m<=n"), exactly one feature 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 inFigure 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 inFigure 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.

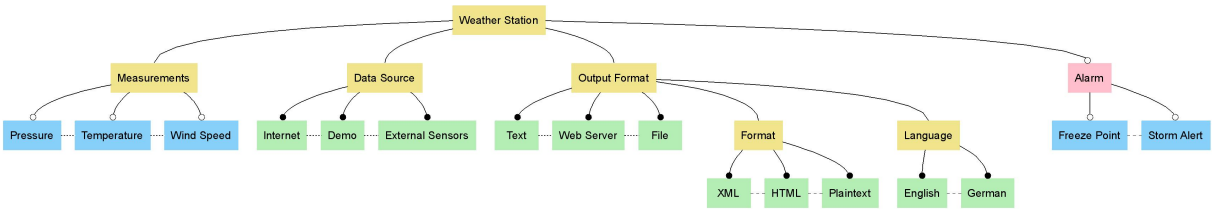
When creating a feature model it can be difficult to decide exactly how problem space variabilities are to be represented in the model. In this case it is best to discuss this further with the customer. It is usually better to base these discussions around the feature model, since such models are easier 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.

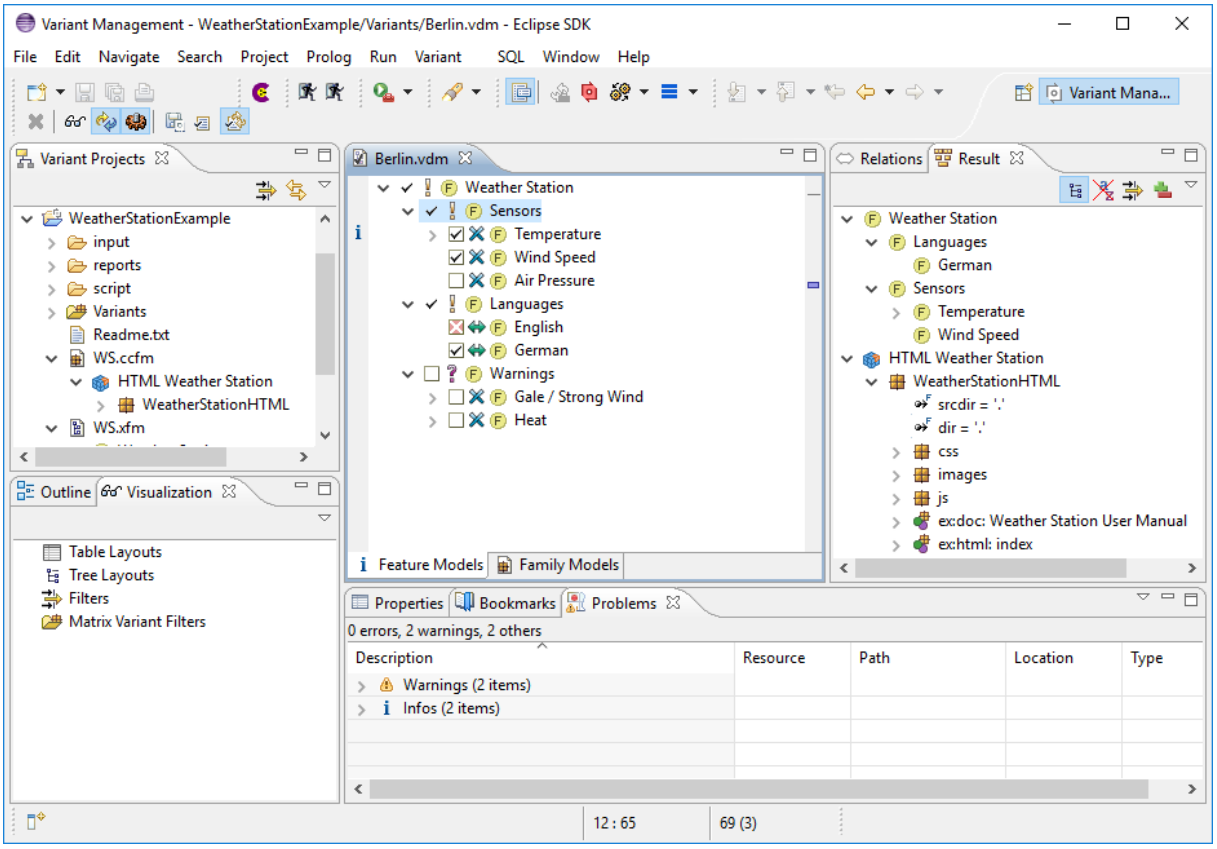
The choice of which tools to use for automating the configuration and / or production of a variant plays a substantial role in the design and implementation of the solution space. The range of variability, the complexity of relations between problem space features and solution constituents, the number 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 Languages implementation component is only included if the Languages feature has been selected from the problem space. To achieve this, a *Languages* rule is attached to the "Languages" component . Any item below “Languages” in the Family model can only be included in the solution if the corresponding Languages 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 English output, an object of type EnglishOutput is instantiated, and with German output, an GermanOutput 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 Languages 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 "Languages" 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 "Languages".

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.

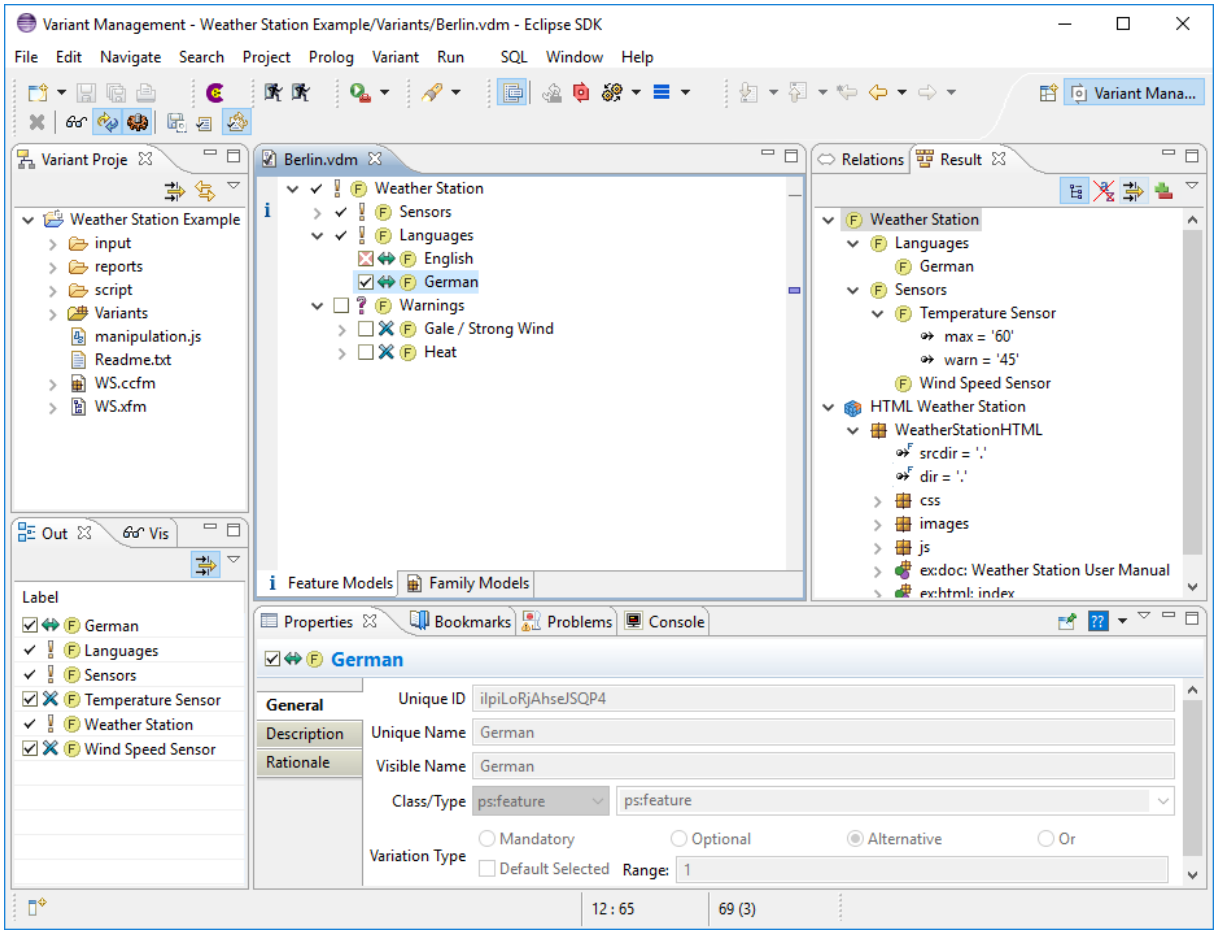
Since all these activities are done on pure::variants model level only, no "real" product has been created at this point. The last step is to execute the transformation, which interprets the models and creates an actual product variant. In pure::variants this transformation is highly configurable. In 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.

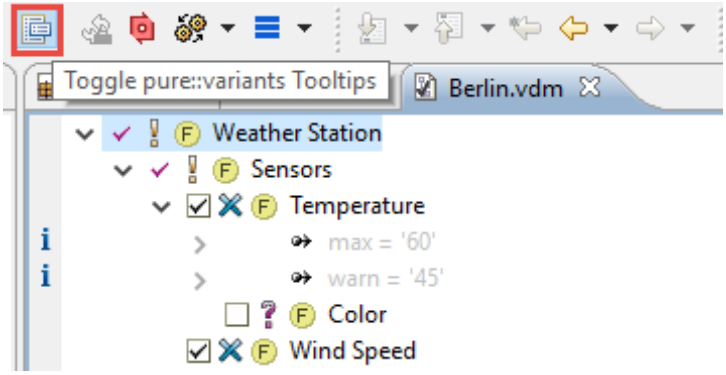
Once the standard project has been created, three editor windows will be opened automatically: one for the Feature model, one for the Family Model and one for the VDM.

To create a new project using a JavaScript template use New->Variant Project from Template. For more details about the template see the *pure::variants JavaScript Extensibility Guide* section *JavaScript Project Template* . The existing template files are shown in a table of the opening wizard. After a template is selected and the name of the project is specified it is possible to specify references projects. Finishing the wizard generates the project like specified in the JavaScript template file.

4.2. Tooltips

By default "pure::variants" shows tooltips when hovering over features, family elements or models in the project view. You can turn off the tooltips, by clicking the "Toggle pure::variants Tooltips" button in the toolbar.

Figure 4.2. Switching Tooltips on/off

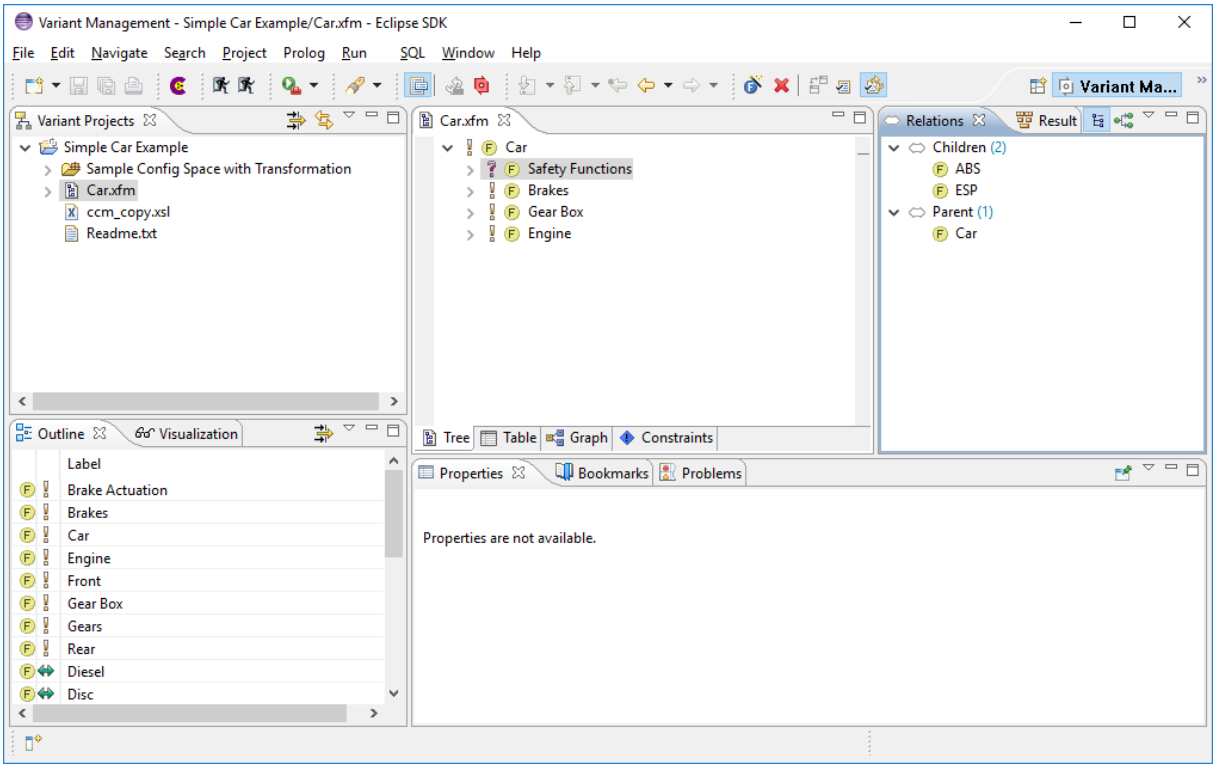


4.3. Using Feature Models

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 4.3. 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.

4.4. 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.

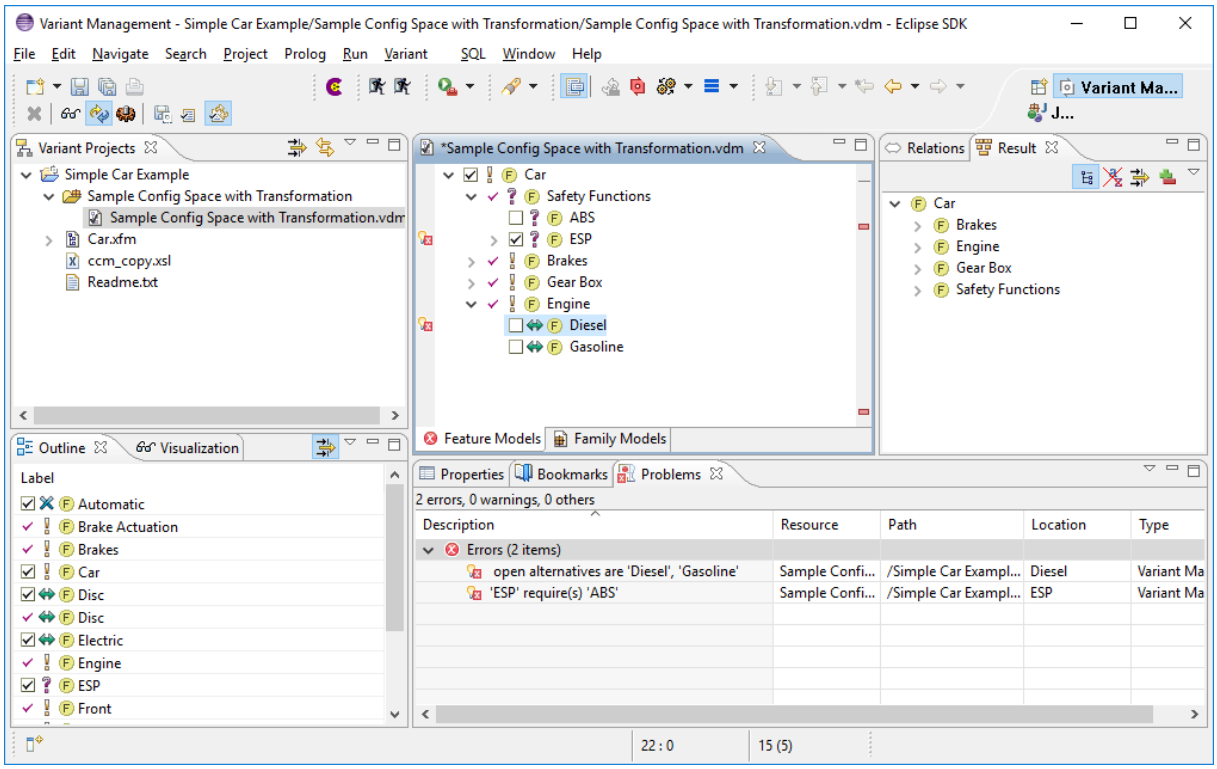
A VDM has to be created inside the Configuration Space for each configuration. This is done using the context menu of the Configuration Space.

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 4.4. VDM with a problematic selection



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

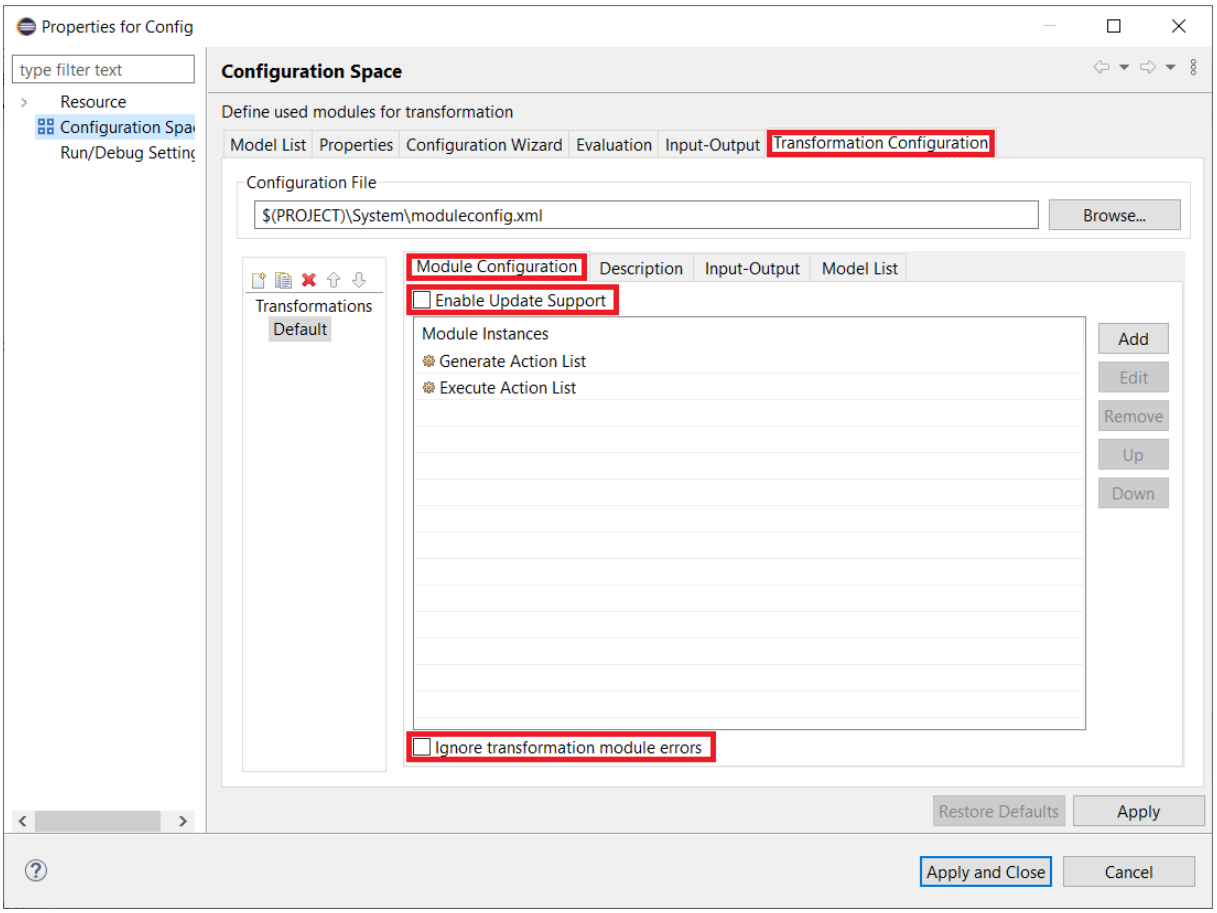
4.5. Transforming Configuration Results

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 configuration for a Configuration Space is specified in its properties dialog. The Transformation Configuration Page (Figure 4.5, “Transformation configuration in Configuration Space Properties”) of this dialog allows the creation and modification of transformation configurations. A default configuration for the standard transformation is created when the Configuration Space is created. SeeSection 6.3.1, “ Setting up a Transformation ” for more information.

Figure 4.5. Transformation configuration in Configuration Space Properties



The toolbar transformation button is used to initiate a transformation (see Figure 4.6, “Transformation button in Eclipse toolbar”). If the current feature selection is invalid a dialog is opened asking the user whether to transform anyway.

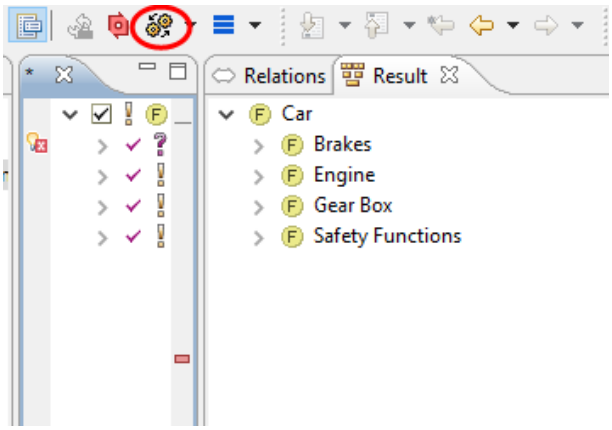
Note

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 4.6. Transformation button in Eclipse toolbar

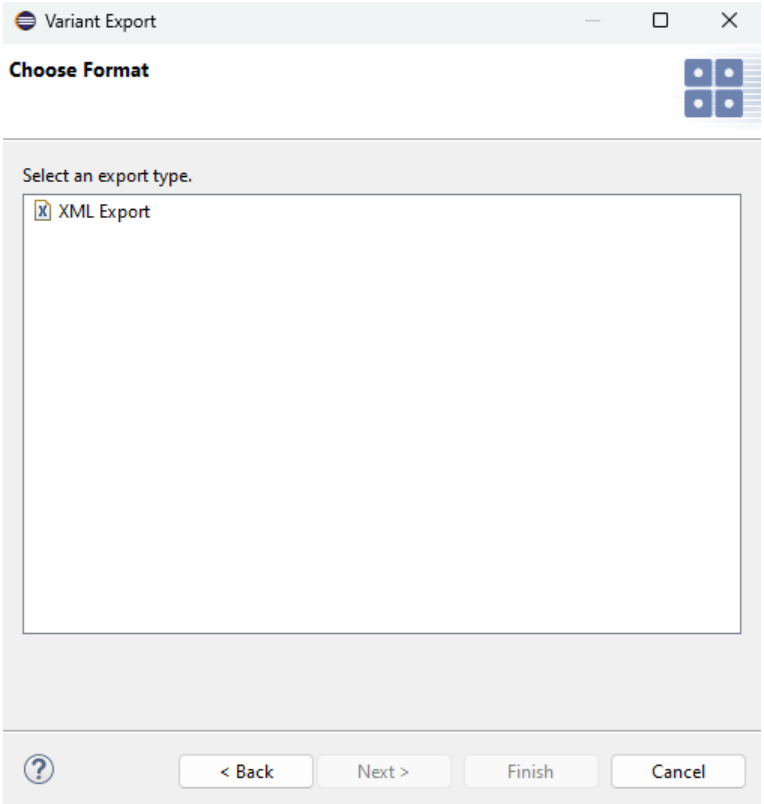


4.6. Viewing and Exporting Configuration Results

Results of a configuration can be accessed in a number of ways. The Result view (Window->Show View->Other->Variant Management->Result) allows graphical review of the concrete 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 format. 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 4.7. VDM export wizard



4.7. 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 ” .

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.

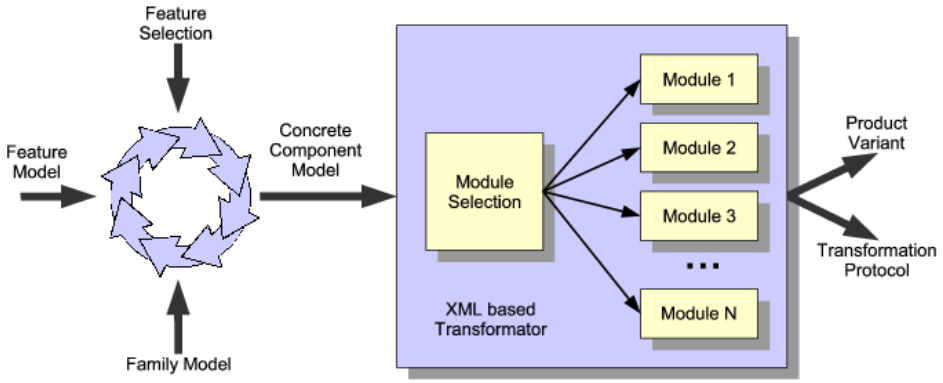
A Variant Description Model (VDM) describes the set of features of a single product, i.e., a configuration, in the product line. Taking a Feature Model and making choices where there is variability in the Feature Model creates these models. VDMs are described in more detail in Section 5.5, “ Variant Description Models ” .

pure::variants supports two modes of configurations in VDMs: In full configuration mode, which was the only mode available in pure::variants 4.0, it is assumed that the set of chosen features is complete. New in pure::variants 5.0 is the partial configuration mode, which assumes that the set of chosen features is not complete and will describe a subset of products of a product line.

The checking, whether the chosen set of features in a VDM is valid, is done in an automatic Model Evaluation. The pure::variants Model Evaluation supports both configurations modes: In full evaluation it is checked whether the current chosen set of features fulfills all relationships of the corresponding Feature and Family Models. In the partial evaluation, however, it is checked whether the current set of features or an extension of that fulfills all relationships. That is, a valid set of features can be reached by eventually selecting more features. More details about the evaluation algorithm can be found in Section 5.8, “ Variant Description Evaluation ” . Also, in the next sections, the evaluation handling for the single modeling parts is briefly described. For a better understanding this only covers the full evaluation. The differences of the evaluation in partial configuration mode is described more in detail inSection 5.8.2, “ Partial Evaluation ” .

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



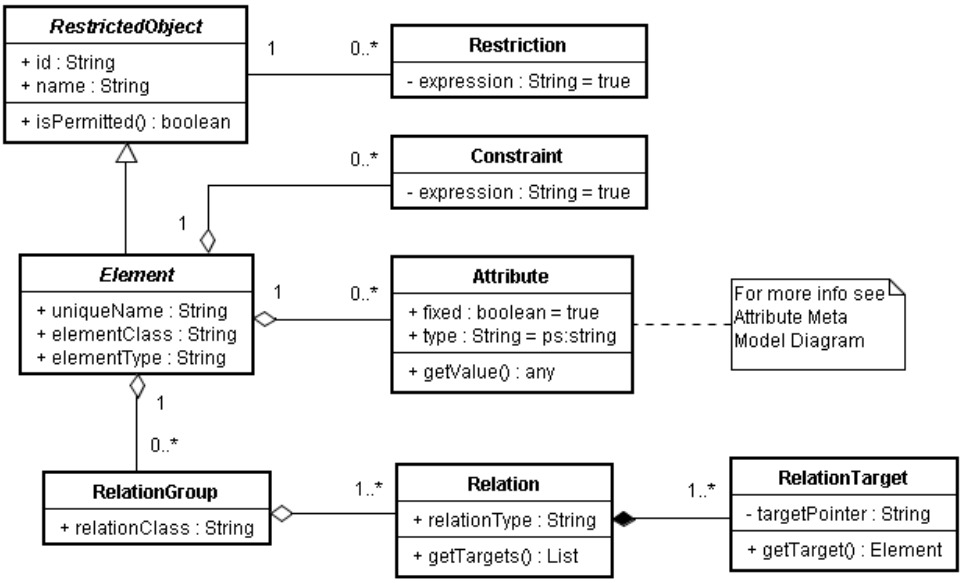
The product line is built by creating Feature and Family Models. Once these models have been created, individual products may be built by creating VDMs. Responsibility for creation of product line models and creation of product models is usually divided between different groups of users.

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

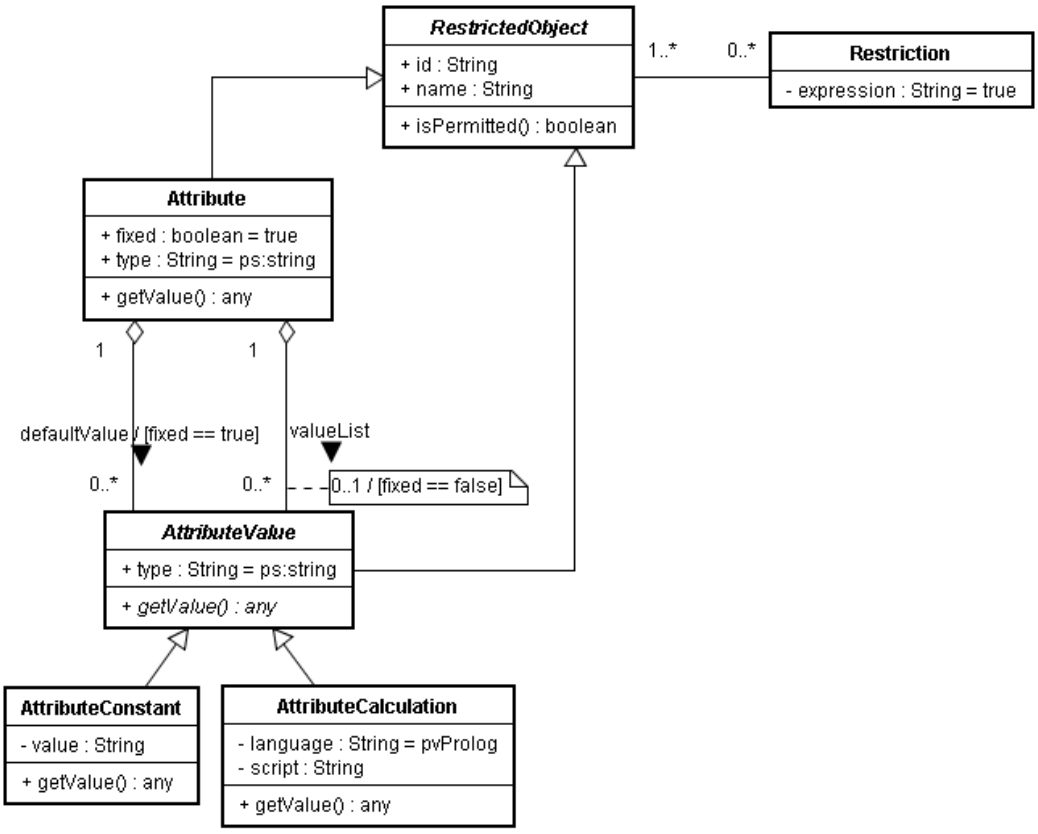
pure::variants allows arbitrary 1:n relations between model elements to be expressed. The graphical user interface provides access to the most commonly used relations. The extension interface allows additional relations to be accessed.

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[]* .

Each attribute of type *ps:integer* or *ps:float* may define a range which the attribute values have to fit in. The Syntax of a valid range is as follows.

- A number. For *ps:integer* attributes decimal numbers are allowed (e.g. 5 or -2) as well as positive hexdezimal numbers prefixed with 0x (e.g. 0x10). For*ps:float* attributes float numbers are allowed in the renage definition. (e.g. 4.56 or 2.9E2)
- An inclusive number range (e.g. [1,*] or [0,3])
- An exclusive number range (e.g. (-5,5) or (0,3))
- A mix of inclusive and exclusive bounds (e.g. (1,23])
- A set of number ranges delimited by commas (e.g. [1,2],[4,7],9)

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* 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]+)? ([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 9.7, “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'
```

5.3. Feature Models

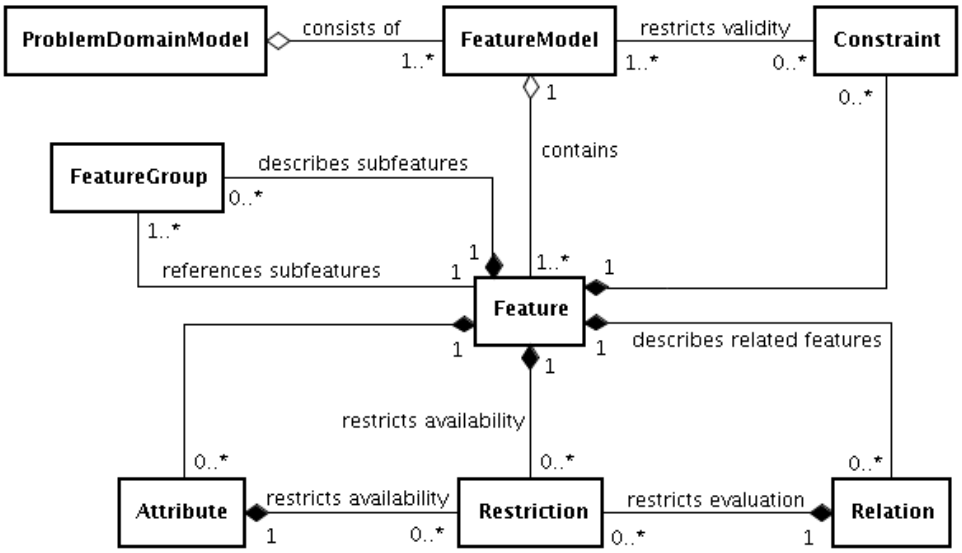
Feature Models are used to express commonalities and variabilities efficiently. A Feature Model captures *features* and their *relations* . A *feature* is a property of the problem domain that is*relevant* with respect to commonalities of, and variation between, problems from this domain. The term*relevant* 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 9.3, “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

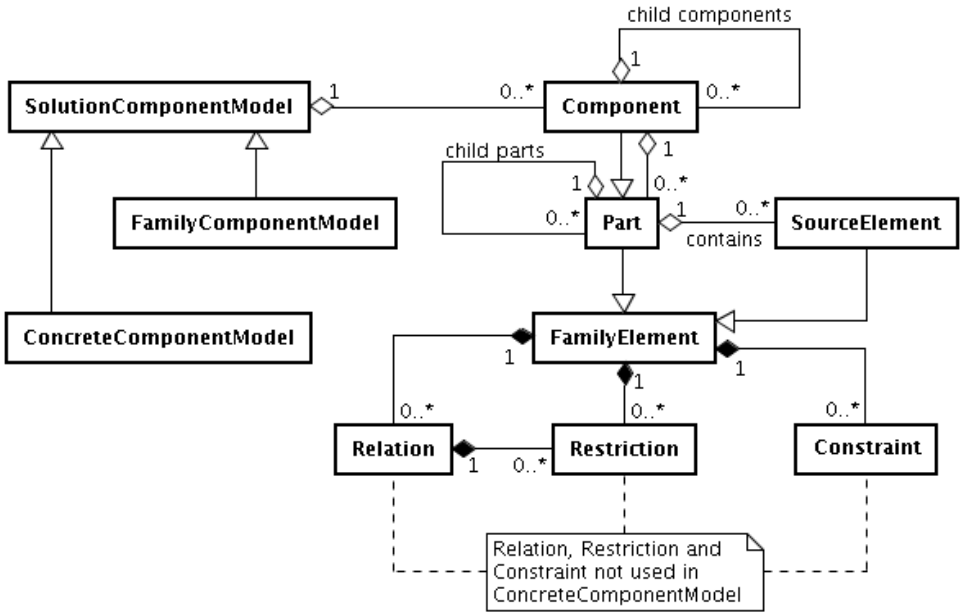
Some features of a domain cannot be easily or efficiently expressed by requiring a fixed description of the feature and allowing only inclusion or exclusion of the feature. Although for many features this is perfectly suitable. Feature attributes (i.e. element attributes in Feature Models) provide a 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,*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:

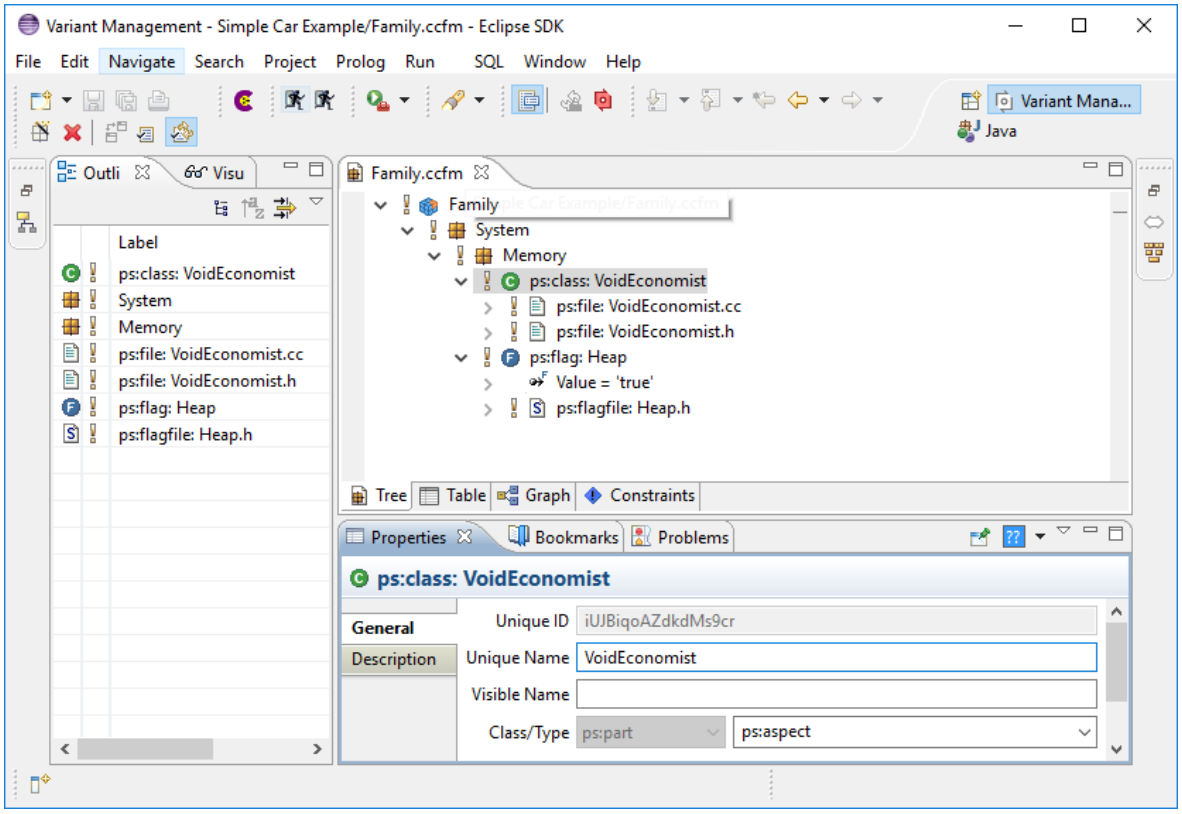
Since parts are logical elements, they need a corresponding physical representation or representations. *Source elements* realise this physical representation. A source element is an unnamed but typed element. The type of a source element is used to determine how the source code for the specified element 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.

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.

5.4.2. 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.

By default every 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. That is, a set of restrictions is evaluated as a disjunction of these restriction.

A restriction rule may contain arbitrary (pvSCL) statements. The most useful rule is `<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

```
    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(BarFoos)
```

Rule2

```
    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(BarFoos) or FoosBar
```

5.4.3. 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: Cosine
   Relation:    ps:requestsProvider = 'Cosine'

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

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

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

```
      Relation:    ps:defaultProvider = 'Cosine'
      Relation:    ps:expansionProvider = 'Cosine'

   ps:file(dir = src, file = cosine_2.cc, type = impl):
      Restriction: FixedTime and 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 inSection 7.3.4, “ Variant Description Model Editor ” . The validity of a feature selection is determined by the pure::variants model validation described inSection 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” .

pure::variants 5 introduces the independent inheritance of attributes values and selections. Now, a VDM can supply only an attribute value but still leave inheriting VDMs the choice to select or exclude the attribute's parent element. Likewise, a VDM can supply only an element selection but still leave inheriting VDMs the choice to supply values for the element's attributes. The independent inheritance mode is active for all projects created with pure::variant 5 and later. Additionally pure::variants 5 projects inherit constraints defined in a vdm. Older projects have to be converted to version 5 in order to use the independent inheritance (See Section 6.18, “Convert a pure::variants 4 project into a pure::variants 5 project”).

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. 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 relations, restrictions, and constraints.

5.8.1. Evaluation Algorithm

The input of the evaluation is a set of feature and family models and a variant description model defining the user selections/exclusions and attribute value assignments. If available, also automatic selections/exclusions created by auto resolver and extended auto resolver runs (see Section 6.1.4, “ Automatic Resolving of Selection Problems ” and Section 6.1.5, “ Automatic Selection ”) are used.

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

Figure 5.7. Model Evaluation Algorithm (Pseudo Code)

```
modelEvaluation()
{

  propagateSelectionsAndExclusions();
  foreach(current in modelRanks())
  {
    selectAndStoreFromFeatureModels(
      getFeatModelsByRank(current));
    selectAndStoreFromFamilyModels(
      getFamModelsByRank(current),class('ps:family'));
    selectAndStoreFromFamilyModels(
      getFamModelsByRank(current),class('ps:component'));
    selectAndStoreFromFamilyModels(
      getFamModelsByRank(current),class('ps:part'));
    selectAndStoreFromFamilyModels(
      getFamModelsByRank(current),class('ps:source'));
  }
  checkFeatureRestrictions(getSelectedFeatures());
  checkRelations();
  checkConstraints();
  calculateAttributeValuesForResult();
}
```

In the first step, the existing selections and exclusions are collected and used to find more trivial, logically imperative selections and exclusions. New selections are added by propagating existing selections up-tree, since a selected element always require a selected parent element. Analogously, new exclusions are added by propagating existing exclusions down-tree. Additionally, new exclusions are also added for all unselected alternatives, if at least one alternative is selected.

In the next step, the feature and family model trees are traversed to collect and add more selections and exclusions based on mandatory relations and based on the element's default-selection state (see Section 6.1.3, “ Default Element Selection State ”) in combination with restrictions. The generally applied rules are: a) If a parent element is selected, any unselected mandatory child element will also be selected. b) If a parent element is selected, an unselected child element with a set default-selection state will also be selected if the child element has either no restriction or at least one restriction, which evaluates to true. If however all restrictions evaluate to false, the child element and all its descendant elements will be excluded. Consequently, restrictions on elements, which are not selected will not be evaluated at all.

Since the evaluation of restrictions usually access the selection state of other elements, the order of adding and requesting selections need to be considered. Therefore, the traversal of the feature and family model trees is distributed by so called *ranks*. On the higher level, *model ranks* define an order of whole sets of feature and family models. Models with a higher rank (with a lower rank index number) will be traversed first. On the lower level, for a set of models with the same model rank, the order of traversal is defined by the element class. The elements of a feature model (i.e., the features) are all of class *feature* and they are traversed first. The elements of a family model are of one of four classes, which are traversed in the following order: a) *family* (the class of the family model root elements), b) *component*, c) *part*, and d) *source*.

So, the traversal is done in the following way for each model rank from higher to lower ranks: First the feature models of the current model rank are traversed in depth-first order starting with the root elements. During traversal, selections and exclusions are collected and added according to the rules defined above. The traversal stops at features, which are not selected and also cannot be selected by the mentioned rules. As soon as the traversal of all feature models of the current model rank is done, the collected and added selections will become visible for the evaluation of restrictions.

After the feature models, the family models with the current rank are traversed, beginning with the elements of the family class, followed by the component, part and source classes. A depth-first traversal is done for all elements of the same class, where again selections and exclusions are collected and added. The traversal stops again at not selectable elements. It also stops on elements of the next class. These elements are used later as a starting point for the element traversal of that class. After the traversal of the elements of each class, the new selections become again visible. So restrictions can always access safely the element selection states of previous classes and model ranks.

Warning

In restrictions, directly or indirectly accessing the selection state of features or elements of the same or lower class or of a lower model rank will always result in Boolean *false*. Make sure that element restrictions are "safe". That is, they do not contain direct or indirect references to elements for which the selection is not yet calculated.

After the traversal of the feature and family models is done, the selections are checked against the feature and family models. First for all selected features and elements the restrictions are checked. Errors are raised for each element with restrictions evaluating to false. Then tree structure relations (i.e., alternatives and or-groups) and element relations are checked. If element relations are restricted, they only need to be fulfilled, if at least one restriction evaluates to true. Again errors are raised for not fulfilled relations. The check of all constraints in all feature and family models will be done after that.

In the last step the values of all attributes of selected features or elements will be determined. This will also do the evaluation of value restrictions and of calculations. Although values of attributes of unselected features and elements are not part of the evaluation result, they can be accessed in restrictions, constraints and calculations. If an attribute value has no value in the result or if no value can be calculated, an error will be raised for this attribute.

5.8.2. Partial Evaluation

As already mentioned in the introduction of this concepts chapter, the Model Evaluation supports the two configuration modes: In the *full configuration mode* , it is assumed, that the set of selected features and elements is complete. So all features, which are not selected, are handled as excluded features. All constraints, relations, and restrictions are evaluated accordingly to this definition. It is also expected that all attributes of selected elements have a value. Therefore, missing values are handled as a configuration error.

In *partial configuration mode* , the set of selected features and elements needs not to be complete, i.e., it is partial. The currently unselected features and elements are handled as *open* decisions, which will be made later, e.g. in an inherited VDM. So for the evaluation there is a difference between excluded and still unselected features and elements. During evaluation any propositional checks are done in*three-valued logic* with the values *true* (for a selection), *false* (for an exclusion), and *open* (for an unselection).

In result, tree structure relations (e.g. alternatives), element relations, constraints, and restrictions can also evaluate to *true* , *false* , or *open* . Only tree structure relations, element relations, and constraints, which evaluate to*false* will create an error. So, no error means that the dependency is either fulfilled or potentially fulfillable. For a restriction set on a feature or element, only an error is created, if that feature or element is selected and all its restrictions evaluate to *false* . A relation with a set of restrictions will only be checked during evaluation, if at least one of the restrictions is evaluating to *true* , since otherwise the relation does not need to hold or it is open whether the relation needs to hold or not. Attributes with a set of restrictions exist in the result, except if all restrictions evaluate to *false* .

In full configuration mode, during traversal of the tree elements only restrictions a) on selected elements, and b) on unselected elements with a set default-selection state that are children of a selected element are evaluated. In partial configuration mode however, to create more exclusions, restrictions on *all* unselected elements are evaluated. If all restrictions on an unselected element evaluates to*false*, this element will be excluded. The default-selection state is not relevant in partial configuration mode. So, new selections will not be created based on that state.

The result value of an attribute can also be *open* depending on preset values, restrictions, and calculation evaluation results. Following rules apply for determining the result value of attributes if the values are set or not:

- A fixed attribute with no value creates an error (as for full configurations).
- A fixed attribute with a non-restricted value results in that value (as for full configurations).
- A non-fixed attribute with no value results in an *open* value, since the user can set this value later on.
- A non-fixed attribute with a default value result in an *open* value, since the user can overwrite the default value later on.

If the attribute values of an attribute have restrictions, the result of that attribute is determined by the evaluation result of each restriction set. So an attribute value can exist (*true*), can not exist (*false*), or can potentially exist (*open*). An attribute value without restrictions can be equated with an attibute value with a restriction always evaluating to *true* . So the next statements also apply if some or all attributes does not have any restrictions.

In full configuration mode the result value of a fixed non-collection attribute with many (restricted) attribute values is determined by finding the first attribute value with a restriction set evaluating to *true* . All previous attribute values with a false restriction will be ignored. If no attribute value remains, the attribute has no value and an error will be created for that attribute.

In partial configuration mode, the open restrictions need to be also considered. So the result depends on the first attribute value, whose restriction set does not evaluate to *false* . If the restriction set of the first of such an attribute value evaluates to*true* , this value will be the result value of the attribute. However, if it evaluates to*open* , the result value will be also*open* , since it is unknown if that attribute value is the right one, or one of its successors. As for full configuration mode, if no attribute value remains, an error will be created.

For fixed collection attributes, i.e, for list and set attributes, in full configuration mode the result value collection contains all attribute values, whose restriction set evaluates to *true* . In partial configuration mode, however, some attribute values only potentially exists. So the resulting collection value could contain optional collection members. If that is the case, so if at least one attribute value has an open restriction set, the resulting collection will be*open* .

Each attribute value is a constant or a calculation. A calculation itself can also evaluate to an *open* value. More information about how pvSCL expressions will be evaluated in partial configuration mode is described inSection 9.7, “Expression Language pvSCL” .

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 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.

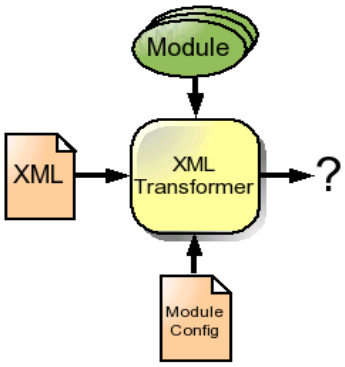
The transformation module configuration is part of the Configuration Space properties (see Section 6.3.1, “ Setting up a Transformation ”).

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, collected in the so-called Variant Result Model. In full configuration mode, the concrete model

variants contain only the selected features and elements. Successfully evaluated restrictions and constraints are removed and attribute value calculations are replaced by their calculated values. In partial configuration mode, the concrete model variants contain both the selected and open features and elements. Only the excluded features and elements are removed. In case of that single calculation results are still open, the concrete model will still contain these calculations. Only the calculations which evaluate in a non-open value will be replaced. Also in case of open restrictions on attribute values, the concrete model variants can contain more attribute values than in full configuration mode.

The type of the feature and family models is changed to signal that these models are concrete variants (see Table 5.1, “Mapping between input and concrete model types”).

Table 5.1. Mapping between input and concrete model types

Input Model Type	Concrete Model Type
ps:fm *(Feature Model)*	ps:cfm *(Concrete Feature Model)*
ps:ccfm *(Family Model)*	ps:ccm *(Concrete Family Model)*
ps:vdm *(Variant Description Model)*	ps:vdm *(Variant Description Model, identical to the input model)*

The Variant Result Model contains additional variant information and is the input of the pure::variants transformation. It has the following structure.

```
<variant>
 <locationinfo>
  <model mid="variant model ID">variant model URL</model>
  <model mid="feature model ID">feature model URL</model>
  <model mid="family model ID">family model URL</model>
 </locationinfo>
 <cm:consulmodels
  xmlns:cm="http://www.pure-systems.com/consul/model">
  <cm:consulmodel cm:id="variant model ID"
   cm:type="ps:vdm" cm:version="1.5">
   ...
  </cm:consulmodel>
  <cm:consulmodel cm:id="feature model ID"
      cm:type="ps:cfm" cm:version="1.5">
      ...
  </cm:consulmodel>
  <cm:consulmodel cm:id="family model ID"
   cm:type="ps:ccm" cm:version="1.5">
   ...
  </cm:consulmodel>
 </cm:consulmodels>
</variant>
```

The `locationinfo` subtree of this XML structure lists the URLs of the models used in the stored variant including the variant model. If the stored Variant Result Model is used for inout to a evaluation or transformation pure::variants tries to open the input models from the stored locations to complete the variant. The `cm:consulmodels` subtree contains a list of all the concrete models.

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 or automatically as part of a transformation result. See the section called “Input-Output Page” for more information.

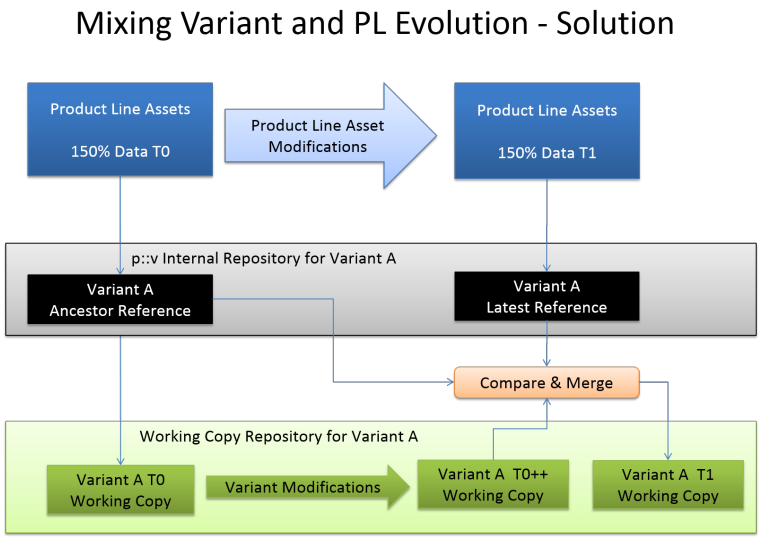
5.10. Variant Update

The Variant Update allows to merge custom changes made in a variant with a newly transformed version of that variant. Sometimes changes for a specific product need to be done after a variant was transformed. When the variant gets transformed again these changes need to be merged in order to keep both pieces of information. To do

this, certain information have to be gathered in order to keep track of who made changes where, and what needs to be merged back into the newly generated variant assets. For that purpose, pure::variants stores each transformation output in an internal repository.

With this information pure::variants is able to update changes to the latest transformation, as well as to the current customer-specific variant, by using a three-way compare. The graphic below shows this process.

Figure 5.9. General Update functionality

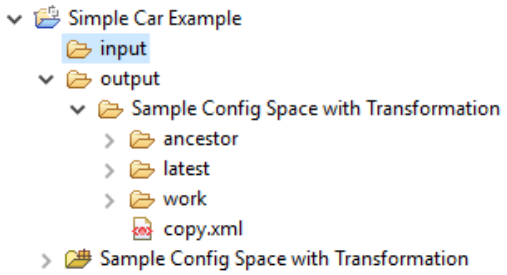


Depending on the tool, our connector either supports a file based update or a tool specific approach.

5.10.1. File based Update

If you activate the update functionality in your transformation module (see the section called “Transformation Configuration Page”), three folders will be generated into your output folder.

Figure 5.10. Folder Structure

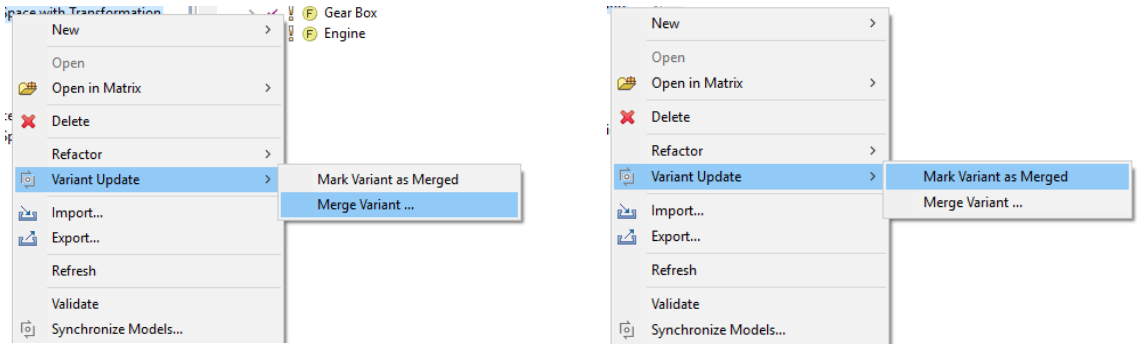


Working copy (work) : a variant created by the transformation that may be edited by the user.

Latest: a variant created by the transformation, which reflects the latest state of the product line.

Ancestor : a variant created by the transformation, which is the common ancestor of both working copy and latest reference.

After all changes have been done and the new version of the variant is generated from the product line, you can merge these changes into your local working copy as follows: Open the context menu of the variant folder you want to update. In the refactor section of the menu, you will find *Variant Update* , where *Merge Variant ...* is located. A three-way compare opens, showing the differences between the files of the respective subfolders, where you can choose which part to keep and which to take over from the product line.



When all changes are applied and saved, you can mark the variant as merged, via*Update Variant -> Mark Variant as Merged* . This will set the "latest" folder as new ancestor and the project is prepared for the next version of the product line to be transformed, so the process can continue.

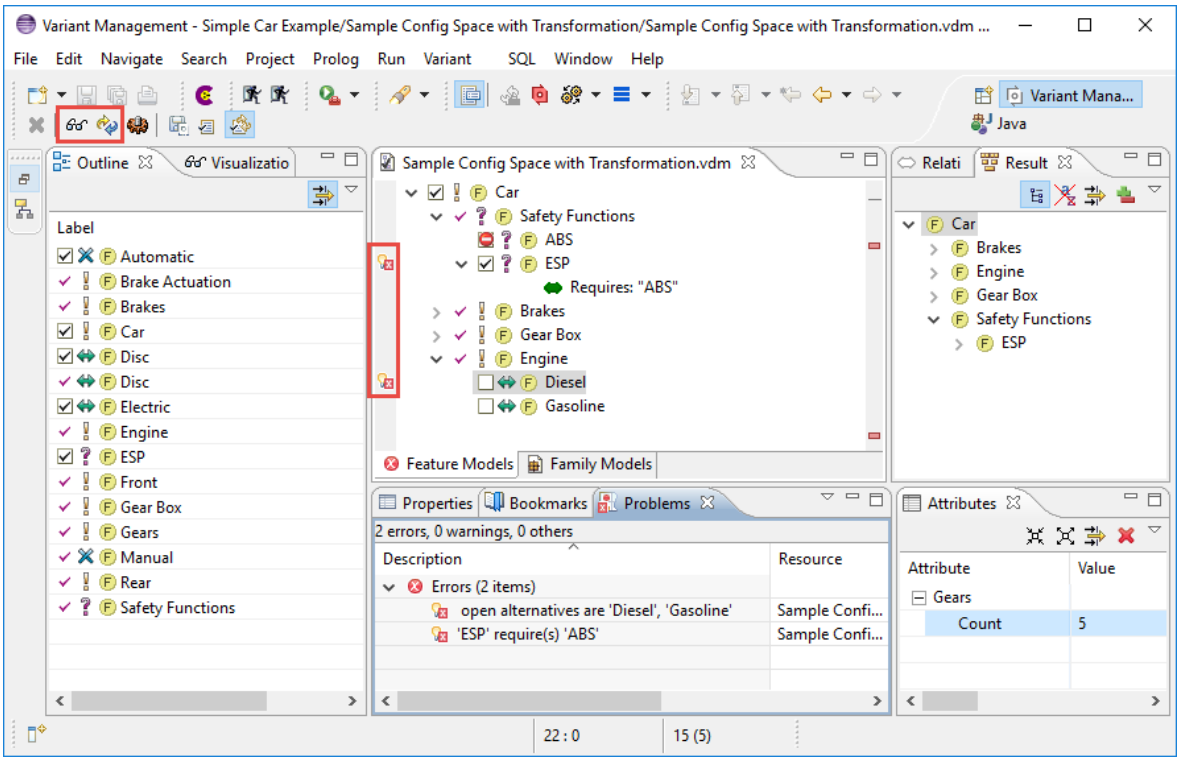
Chapter 6. Tasks

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.

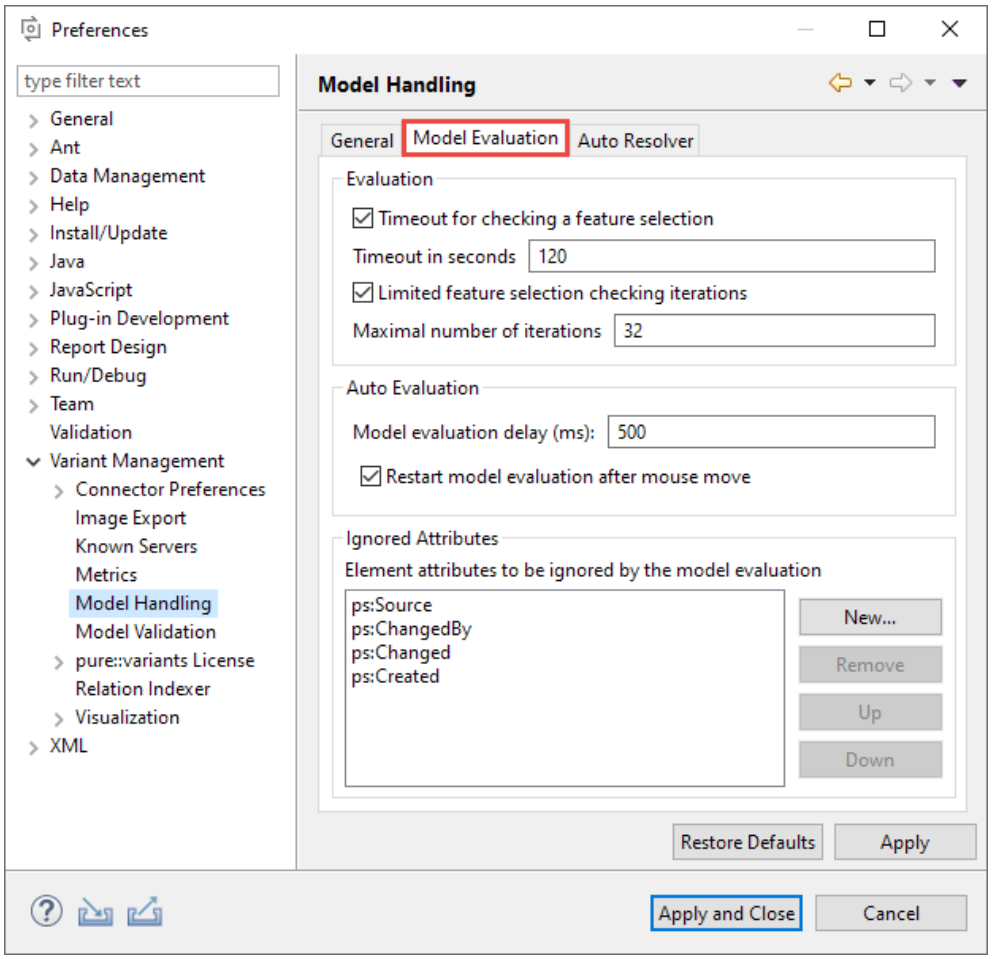
If the variant description is valid, then the result of the evaluation are the concrete variants of the models in the Configuration Space shown in the Result View (see Section 7.4.8, “ Result View ”). The concrete variants of the models are collected in the Variant Result Model, that can be saved to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

6.1.1. Configuring the Evaluation

Workspace-specific settings

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

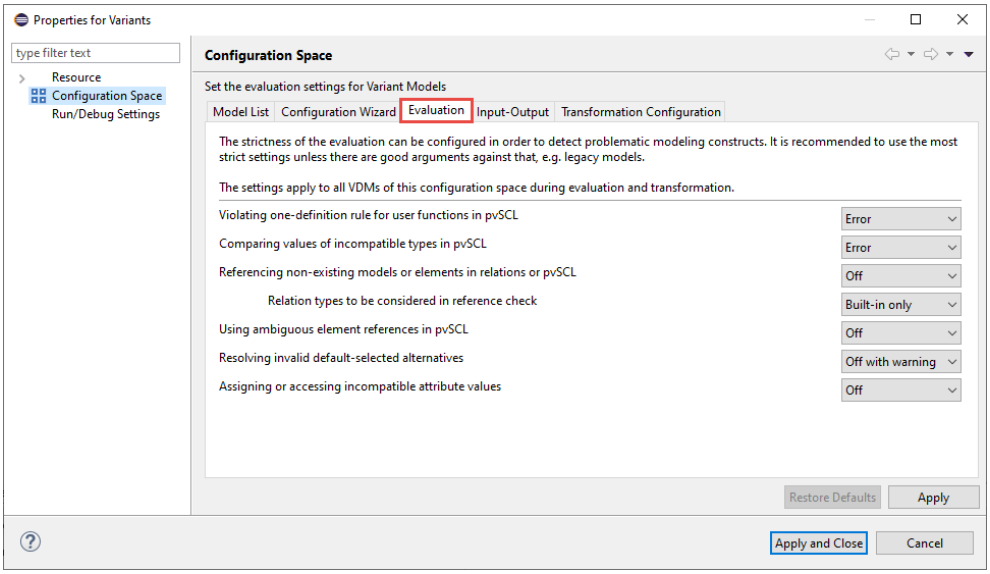
For listed attributes it is not possible to access them in restrictions and calculations during the model evaluation process. These attributes also do not become part of the Variant Result Model, i.e. the concrete models of the variant.

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

Configuration-Space-specific settings

For each configuration space, the strictness of the evaluation of the contained variant description models regarding problematic modeling constructs can be configured (see Figure 6.3, “Configuration Space Evaluation Settings Page”).

Figure 6.3. Configuration Space Evaluation Settings Page



The following problematic constructs are checked and for each of them the complain level is configurable. However, it is recommended to use the most strict settings if possible. One example for an exception of this recommendation would be the enabling of current pure::variants versions to use also legacy or baseline models containing such constructs.

Violating one-definition rule for user functions in pvSCL

pvSCL user function definitions have to follow the one-definition rule (ODR). So, multiple definitions of pvSCL functions with the same signature, i.e., same name and same number of arguments, violate this rule. For the user it is usually unclear, which of these multiple function implementations will be called in which case. Hence, to ensure the adherence to the ODR, the evaluation checks for this violation.

Comparing values of incompatible types in pvSCL

Comparing values of incompatible types in pvSCL, e.g., of values of type string and number, involves an implicit conversion of each value into a string followed by a string comparison. This could lead to potentially unexpected results for the user.

Referencing non-existing models or elements in relations or pvSCL

Referencing non-existing models or elements is a common modeling issue. Especially, references to non-existing elements are interpreted as unselected by default. So they are not easy to find.

Relation types to be considered in reference check

The scope of the non-existing references check is configurable: With option *All* all relation types are checked, whereas with option *Built-in only* the check is done for pure::variants' predefined relation types only (see Section 9.2, “Element Relation Types”).

Using ambiguous element references in pvSCL

Using several models in a configuration space can result in non-unique element names. Referencing such elements by their name is ambiguous and can lead to unexpected results, since the first occurence is used by default.

Resolving invalid default-selected alternatives

Multiple default-selected non-restricted alternatives result in an invalid configuration. In rare circumstances, this problem can be automatically re-

solved, but the automatic resolution could also lead to potentially unexpected results, e.g. invalid variants although a valid solution exists.

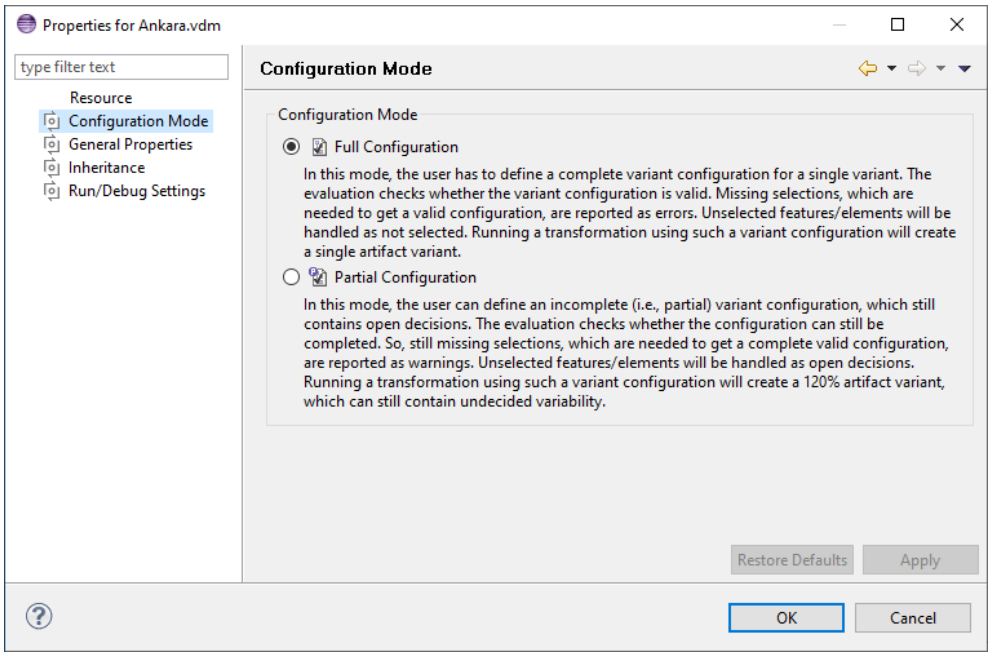
Assigning or accessing incompatible attribute values

User-defined or calculated attribute values have to match the attribute's type. Values with a wrong type can be implicitly converted to the attribute type in some cases, e.g. integer to float, number to string, and element to its Boolean selection state. However, in many cases this is not possible or not wanted.

6.1.2. Setting the VDM Configuration Mode

For pure::variant version 5 projects, the configuration mode can be set for each VDM separately during creation of a VDM or later at any time in the *Configuration Mode* page of the VDM's *Properties* dialog (see Figure 6.4, “Variant Model Configuration Mode Page”).

Figure 6.4. Variant Model Configuration Mode Page



6.1.3. Default Element Selection State

Each feature and element has a default selection state defined in Feature and Family Model. Normally Family Model elements and mandatory features are created with the state "on". All other Feature Model elements are created with the state "off". Except for mandatory features and elements, the default selection state can be changed by the user.

In full configuration mode, a feature or element with the default selection state "on" is selected automatically 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.

In partial configuration mode, the default selection state is ignored, since this state controls the default handling of unselected elements. So, unselected elements stay *open* independent of the default selection state.

6.1.4. Automatic Resolving of Selection Problems

If a feature selection is evaluated to be invalid, selection problems will occur. Such selection problems are for instance failed relations, constraints or restrictions. Certain selection problems are eligible to be resolved automatically. For example, a not yet selected feature that is required by a relation can be selected automatically.

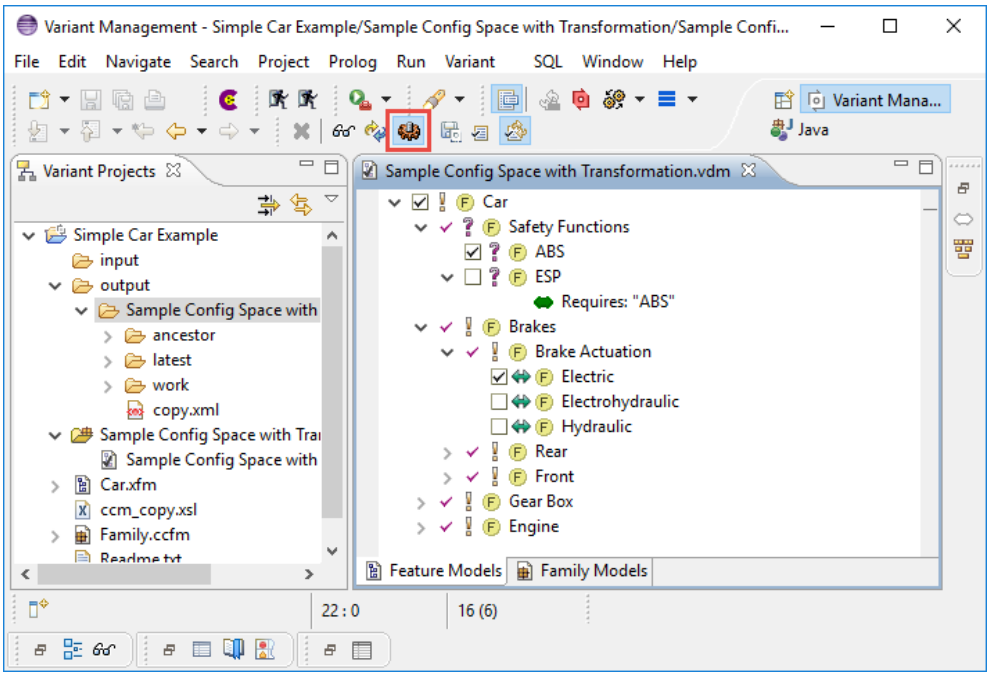
The pure::variants auto resolver component provides a set of heuristics to resolve failed relations, features selection ranges and basic propositional constraints. They are applied only in full configuration mode. In partial configuration mode the auto resolver is not executed.

Note

The auto resolver does not change the selection state of user selected or excluded features. It only adds new selections or exclusions.

As shown in Figure 6.5, “Automatically Resolved Feature Selections” , auto resolving for a VDM is enabled by clicking button in the tool bar.

Figure 6.5. Automatically Resolved Feature Selections



6.1.5. Automatic Selection

The auto resolver does only resolve selection problems locally, i.e., it considers only a single relation or constraint. It cannot consider the potentially hidden dependencies of the complete set of the evaluated feature and family models.

The pure::variants extended auto resolving component therefore uses an approach to add feature selections and exclusions, which are logically mandatory based on the whole set of feature and family models and the user selections and exclusions. It will run before evaluation, so the evaluation already checks these new automatic selections. The extended auto resolving is executed both in full and partial configuration mode. For both modes the behavior is equal.

The extended auto resolving uses a *SAT solver* based approach. It covers the propositional part of the models, i.e., the feature and family model tree structure with selection ranges, all built-in relations, and the propositional parts of constraints and restrictions in pvSCL, like Boolean operations. It does not cover attributes with their values and parts of pvSCL expressions, which are not propositional, like comparisons, arithmetical operations and model element traversal. However, non-propositional expressions do not influence the reliability of the result. The not useable parts are simply mapped to open Boolean variables.

Considering the already done user selections and exclusions, the extended auto resolving will first check, if the propositional part is satisfiable, i.e., a configuration can be reached by adding more selections, which at least fulfills all propositional dependencies of the models. If the satisfiability is given, for each unselected feature and element it will be determined, whether it has to be selected or excluded to fulfill all the propositional rules. However, if

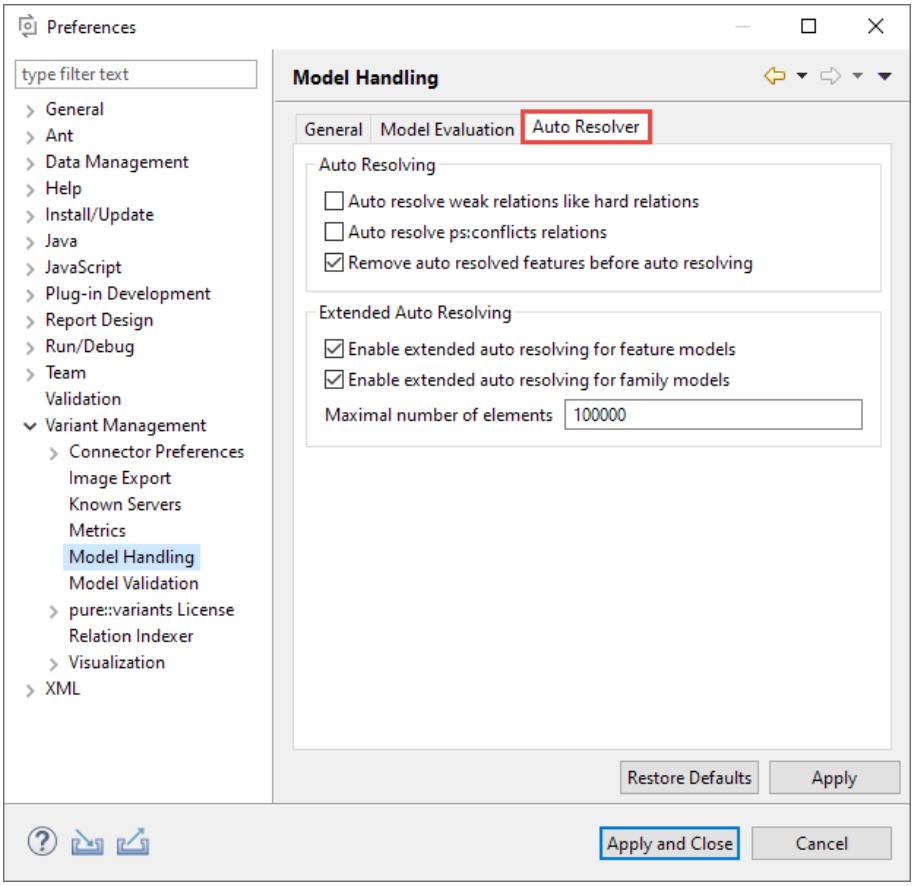
the models also contain non-propositional parts, it is still possible that a configuration, which fulfills all model dependencies can never be reached.

If the propositional part of the models is not satisfiable, i.e., there is a conflict in the models or the user selections or exclusions, the extended auto resolving cannot determine any new selections and exclusions. Then also the complete model dependencies including the non-propositional parts, cannot be fulfilled.

6.1.6. Configuring the Auto Resolver

Both auto resolving components are configured on the Auto Resolver tab of the Variant Management->Model Handling preferences page (menu *Window->Preferences* , see Figure 6.6, “Auto Resolver Preferences Page”).

Figure 6.6. 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 100,000 elements.

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 Configruation 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.

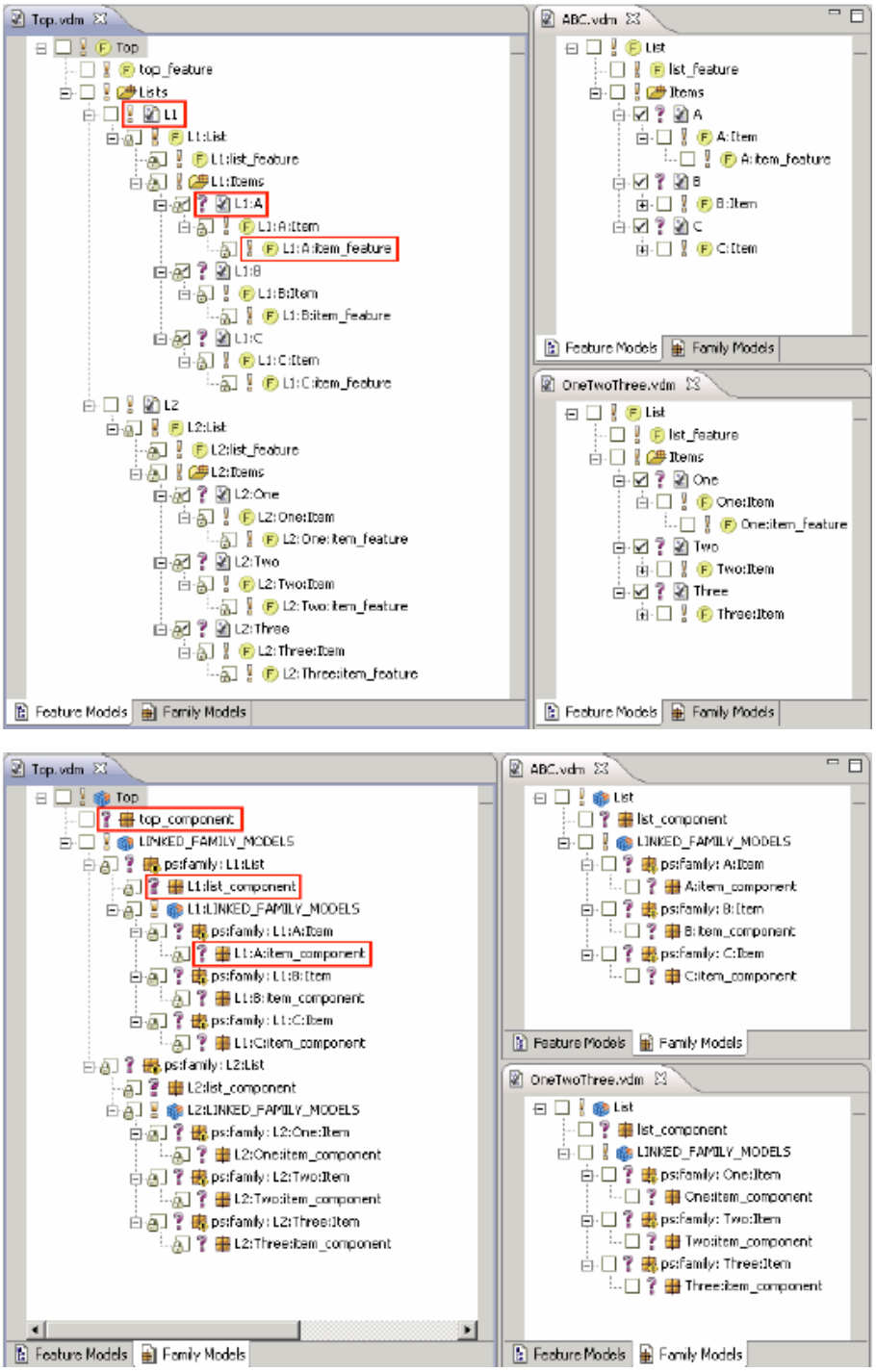
If a variant instance link is created, then the VDM Editor provides two additional actions in the context menu on the corresponding link elements, i.e. *New->Instance* and *Remove Instance* . These actions allow to create and remove the concrete instances, i.e. VDMs, of the linked Configuration Space.

Relations between the variants of a variant hierarchy can be expressed using restrictions and constraints. See Section 9.7.8, "Name and ID 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 6.7, “Unique Names in a Variant Hierarchy” and shows a hierarchy of three variants and how the unique names are prefixed in each variant.

Figure 6.7. Unique Names in a Variant Hierarchy

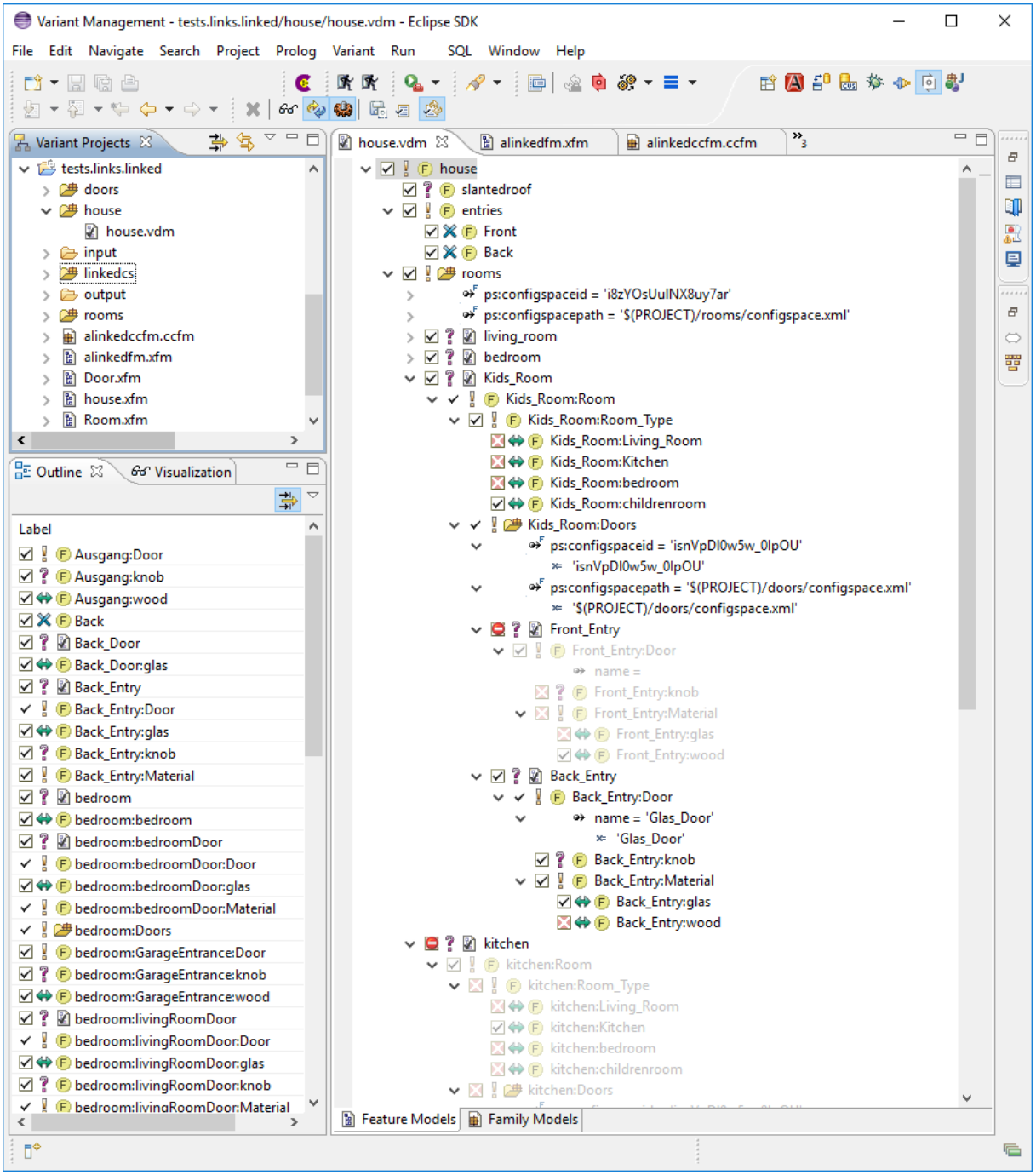


The unique IDs are prefixed in the same way except that the unique ID of the link elements is used as prefix.

Example Variant Hierarchy

Figure Figure 6.8, “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 6.8. Example Variant Hierarchy



6.2.2. Inheritance of Variant Descriptions

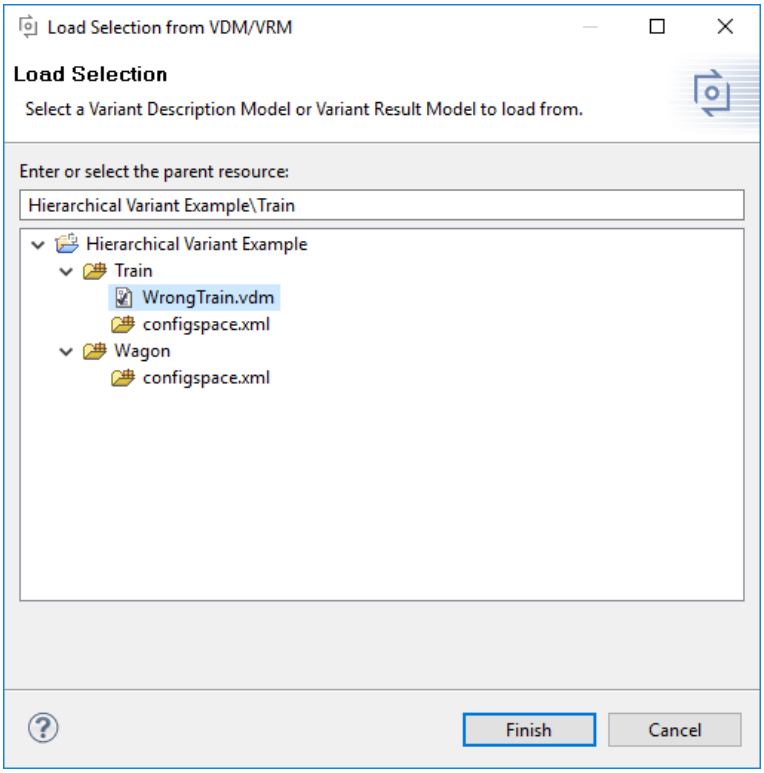
pure::variants supports sharing common feature selections/exclusions between several variant descriptions. This allows users to define the models for each VDM from which selections are to be inherited. Changes in the inherited model selection will be propagated automatically to all inheriting models. 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 inFigure 6.9, “Load Selection Dialog” .

Figure 6.9. Load Selection Dialog



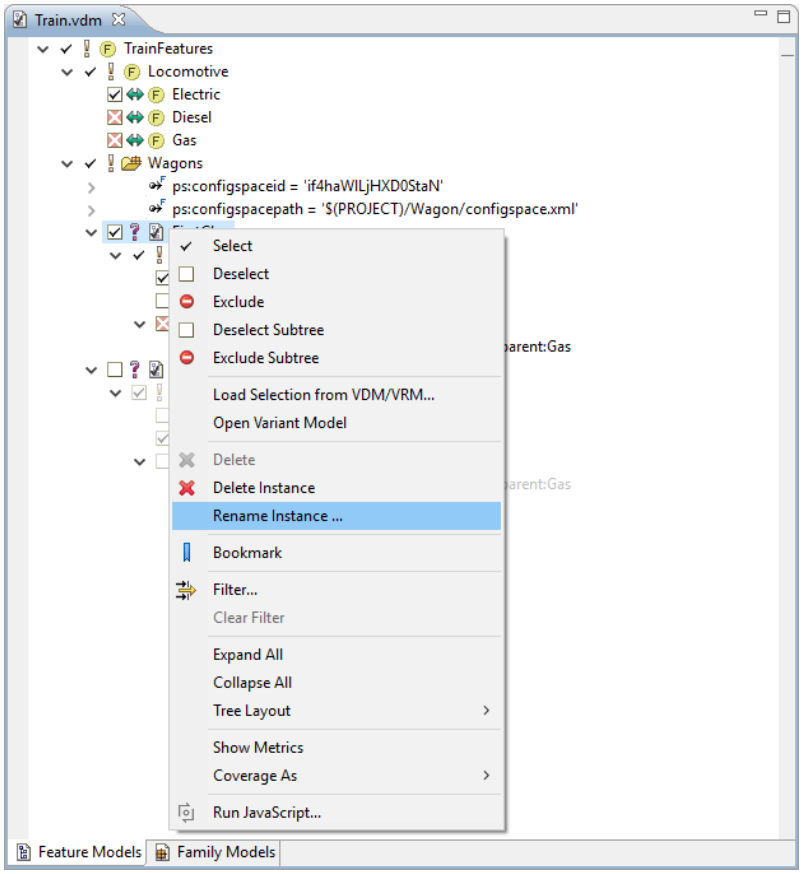
In this dialog the VDM from which to load the selection has to be selected. All selections in the currently edited VDM are overwritten with the selections from the loaded VDM.

If this action is called in an instance element the selections are changed for the selected instance only.

6.2.4. Rename Reused Variant Description Model

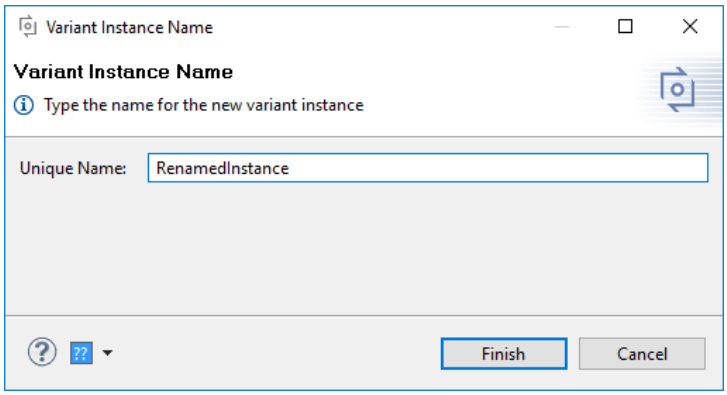
A reused Variant Description Model ("instance") can be renamed by selecting **Rename Instance ...** from the context menu as shown inFigure 6.10, “Rename Reused Variant Description Model” .

Figure 6.10. Rename Reused Variant Description Model



The opened dialog lets you choose a new name for the instance at hand and also has the option to allow a name comparison. If the option is set in two instances with the same name and the same parent, these instances will be treated as equal in comparisons. If the option is left out, the instance will be treated as unique and independent, although it might be named and positioned as another instance in another Variant Description Model.

Figure 6.11. Rename Dialog



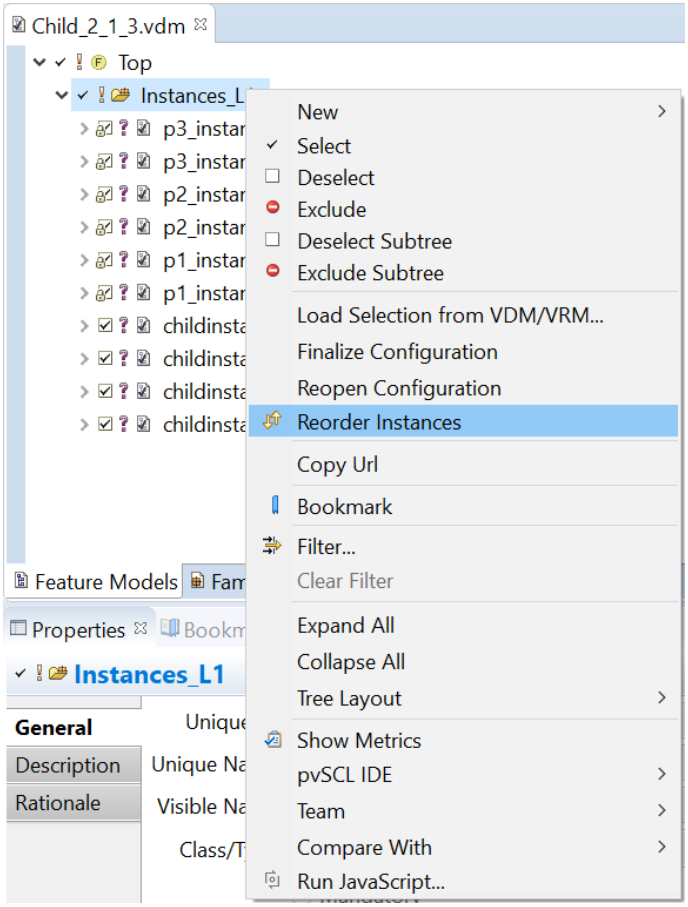
6.2.5. Reorder Reused Variant Description Models

The reused Variant Description Models ("instances") can be reorder by selecting **Reorder Instances** from the context menu as shown in Figure 6.12, “Reorder Reused Variant Description Models” . The context menu can be found by right clicking on the instance group or any instance in the variant model editor.

The reordering is only allowed for non-inherited variant instances. The order of the inherited instances is the same as in the inherited models and are grouped together in the order of the inherited models themselfs. Instance order

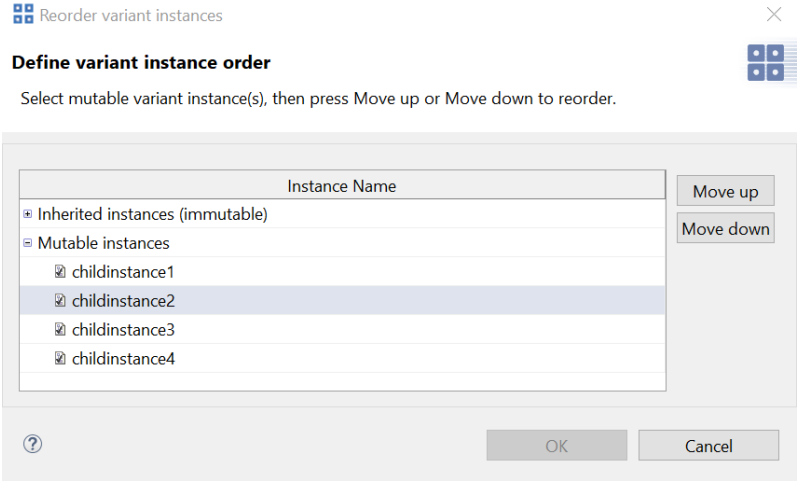
mutation is allowed only between not inherited instances. Nonetheless, it is not allowed to move an instance out of the containing group.

Figure 6.12. Reorder Reused Variant Description Models



A dialog for reorder instances will pop up as shown in Figure 6.13, “Reorder Instances Dialog”, which lists two categories of variant instances. Inherited instances are in the immutable list and non-inherited instances are in mutable list. Any or consecutive number of multiple instances can be selected and moved up or down using **Move up** or **Move down** button. To confirm the order, **OK** button can be pressed. Then *Save* the variant model to store the instance order. The categories themselves are not movable. Inherited instances are always showed at the top of the group.

Figure 6.13. Reorder Instances Dialog

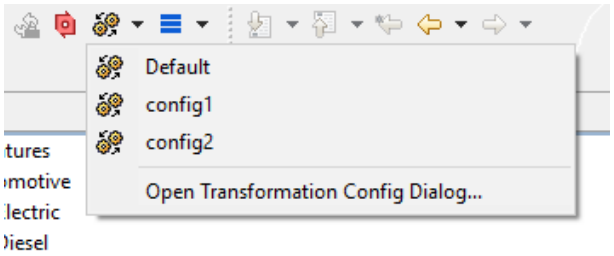


6.3. Transforming Variants

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 6.14. Multiple Transform Button



6.3.1. Setting up a Transformation

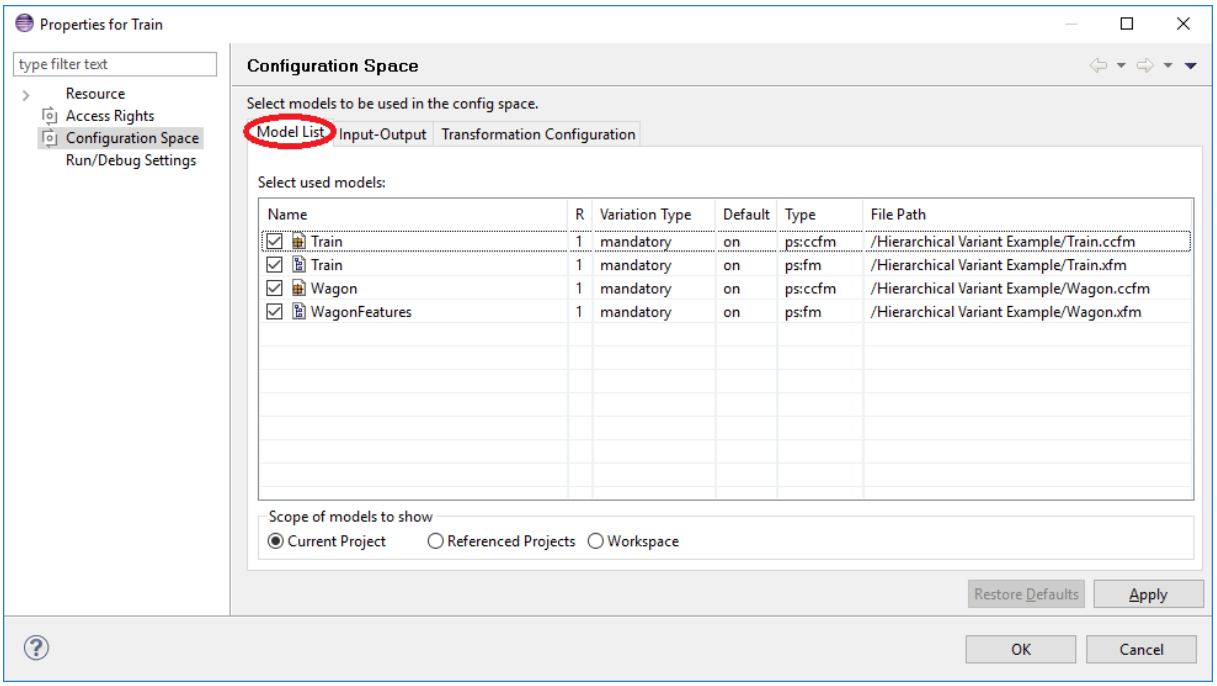
The transformation must initially be set up for a specific Configuration Space. Therefore the Configuration Space properties have to be opened from the Variant Projects view by choosing **Properties** from the context menu of the corresponding Configuration Space.

The editor is divided into six 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 6.15. 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.

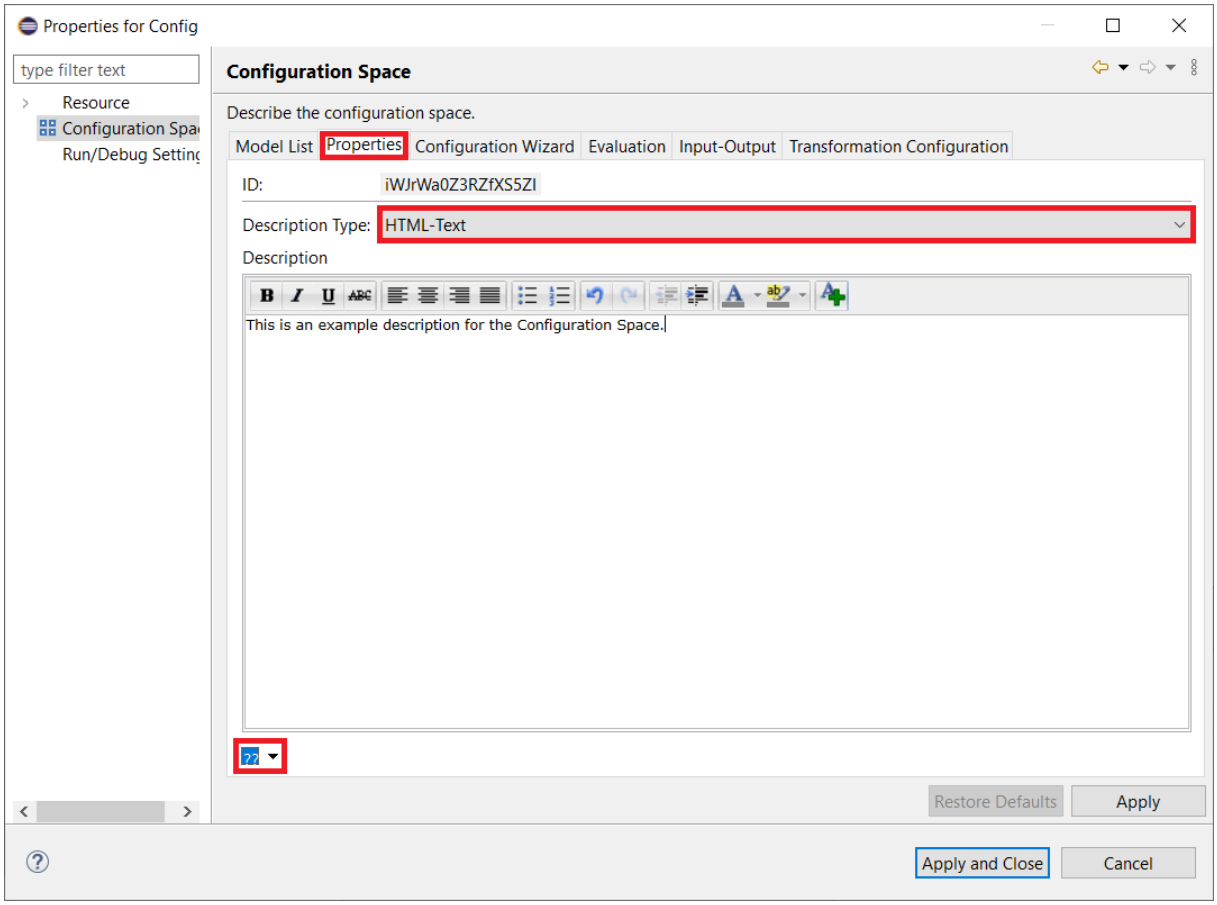
The next column ("Default") can be used to specify whether a optional model is default selected in the variants or not. This semantic is ether equal to the default selected state of pure::variants model elements.

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* .

Properties Page

This page is used to specify a description for the Configuration Space. You can also retrieve the Configuration Space ID from here. The description supports different MIME types, like plain or HTML text. The MIME type can be changed by clicking on the description type combo box. By changing the MIME type, all descriptions of that Configuration Space are getting converted. The description also supports different languages. You can switch between the languages by clicking on the dropdown on the bottom left. A Configuration Space can hold different descriptions, one for each language.

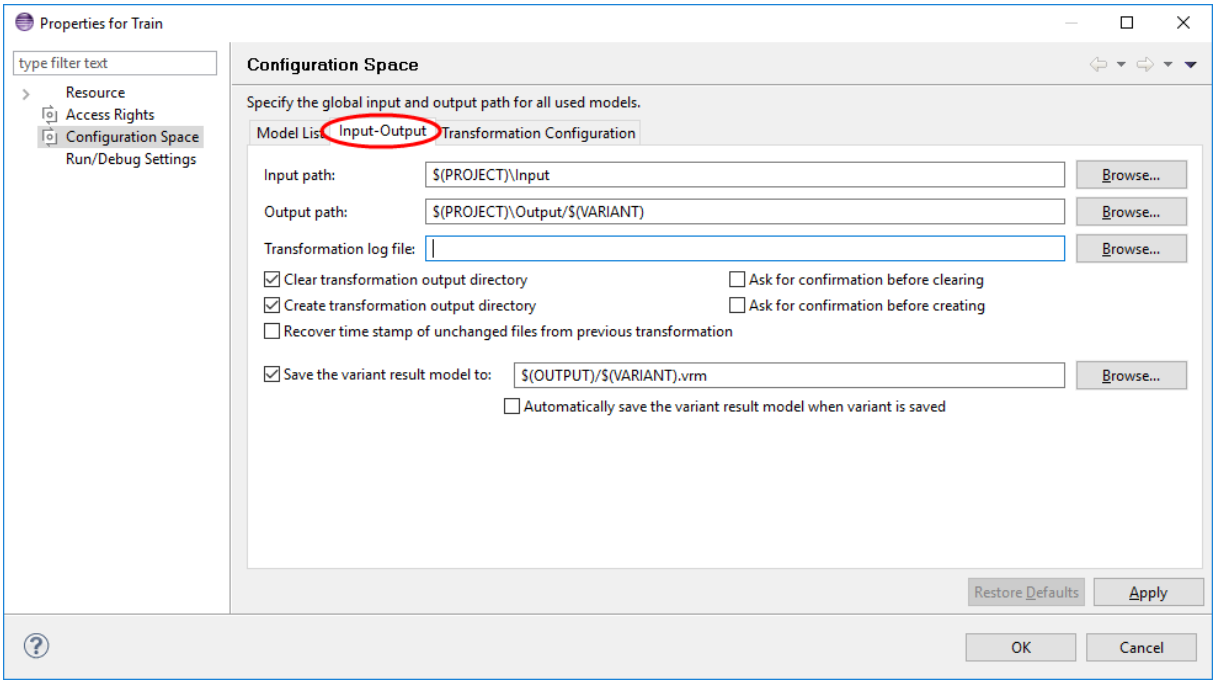
Figure 6.16. Configuration Space properties: Properties



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 6.17. 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 transformation log file is used by transformation modules to log their activities while transformation. 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 9.8, “ 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.

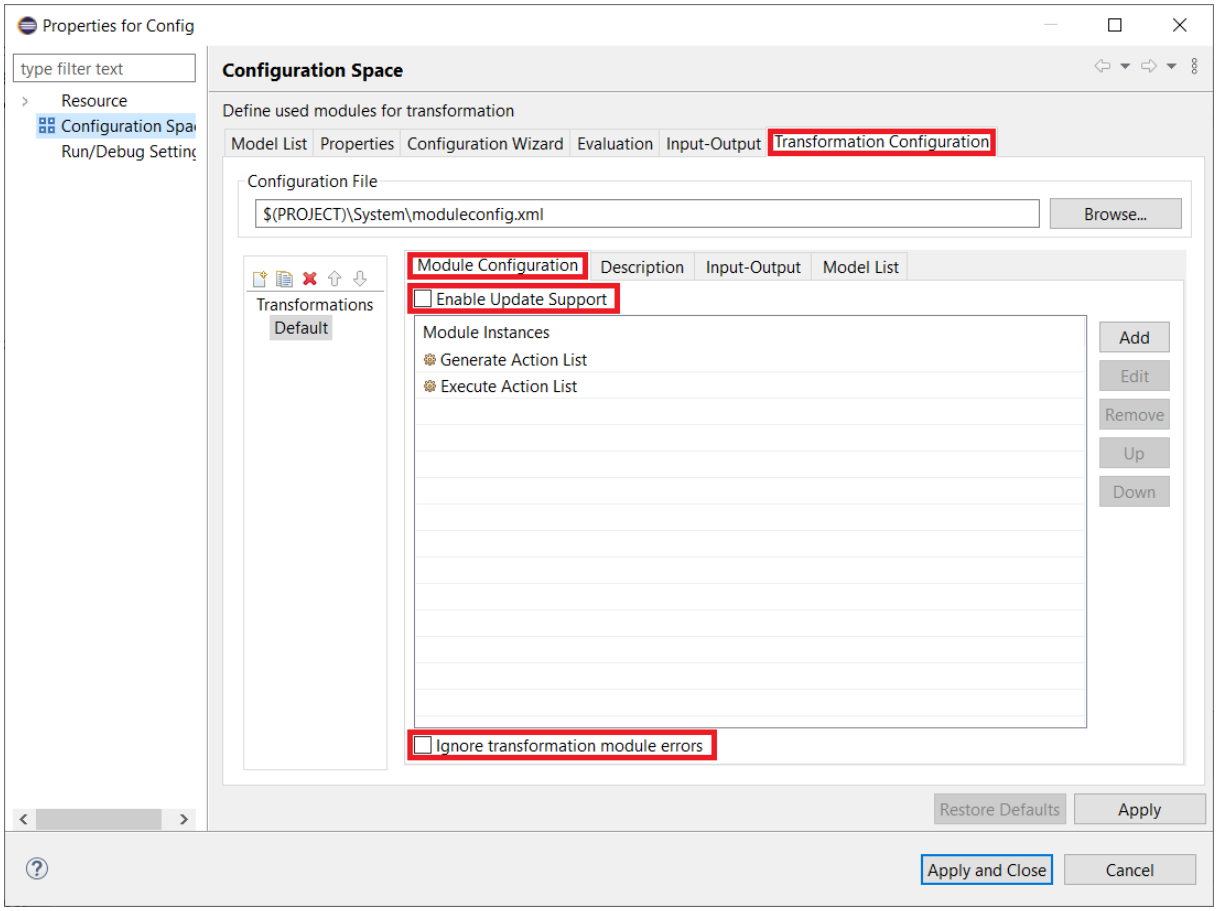
The "Save the variant..." option instructs the transformation framework to save the Variant Result Model to the given location. The Variant Result Model is the input of the transformation framework containing the concrete variants of the models in the Configuration Space.

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 6.18. 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* .

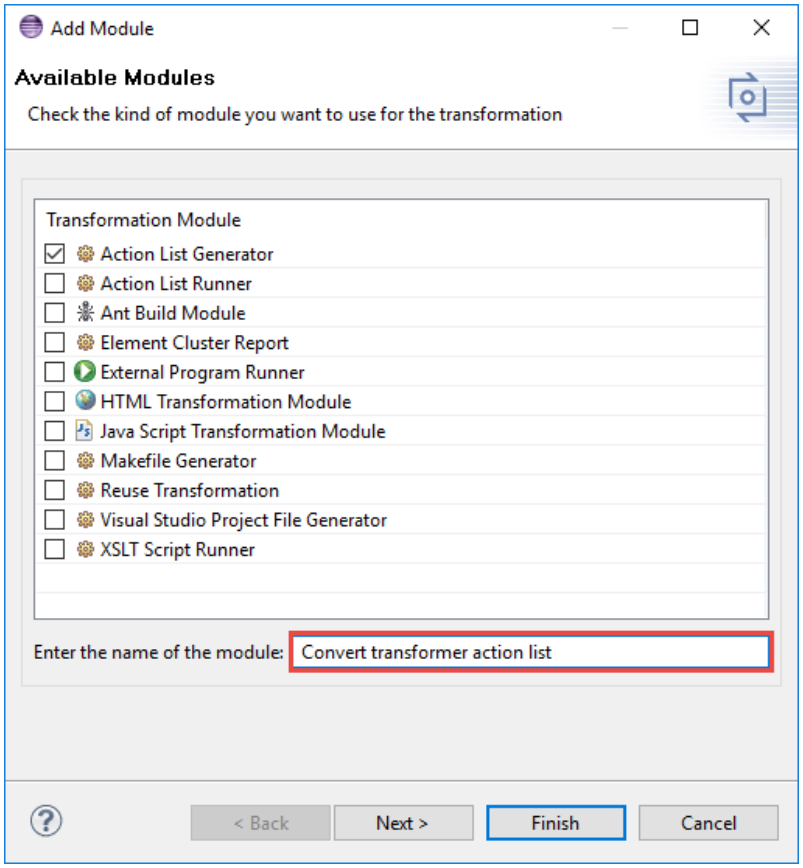
When a Transformation Configuration is selected on the left side, it can be edited with the lower right part of the Transformation Configuration Page. A Module Configuration consists of a list of configured modules. Since many modules have dependencies on other modules they must be executed in a specific 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 *Enable Update Support* button on the top of the right page is checked, the created output of transformation modules for a given variant has to support variant update scenario. In that case an already existing output for this variant may not be overwritten while transformation but can be updated afterwards with the newly created output.

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.

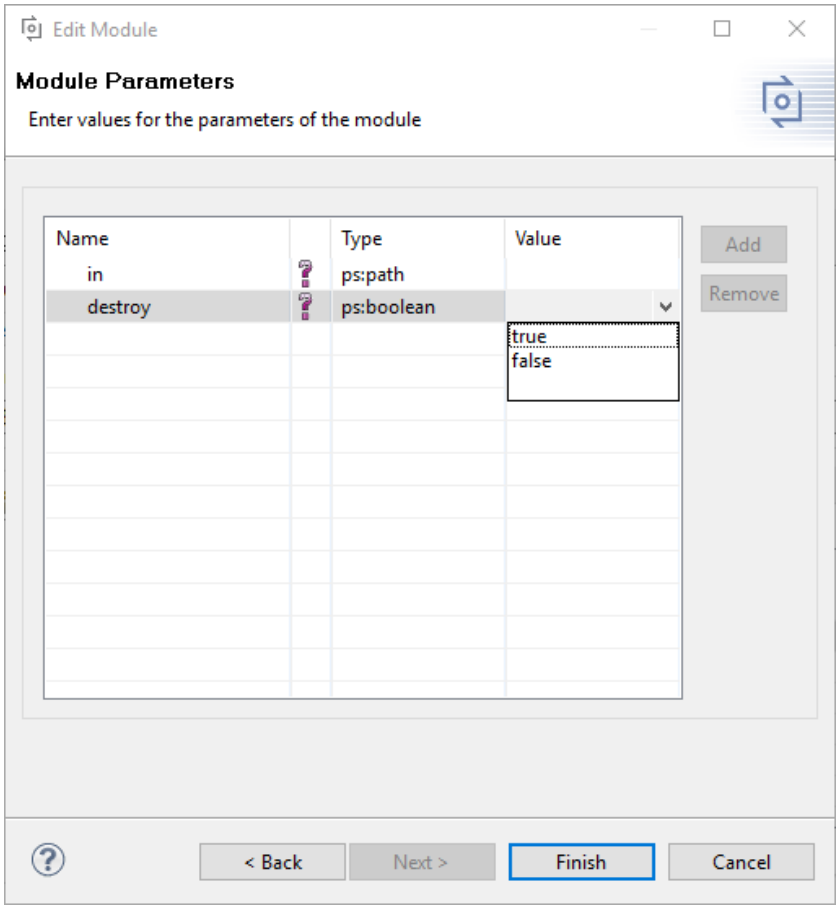
The buttons on the right side allow transformation modules to be added to or removed from the configuration, and to be edited. When adding or editing a transformation module a wizard helps to enter or change the module's configuration.

Figure 6.19. Transformation module selection dialog



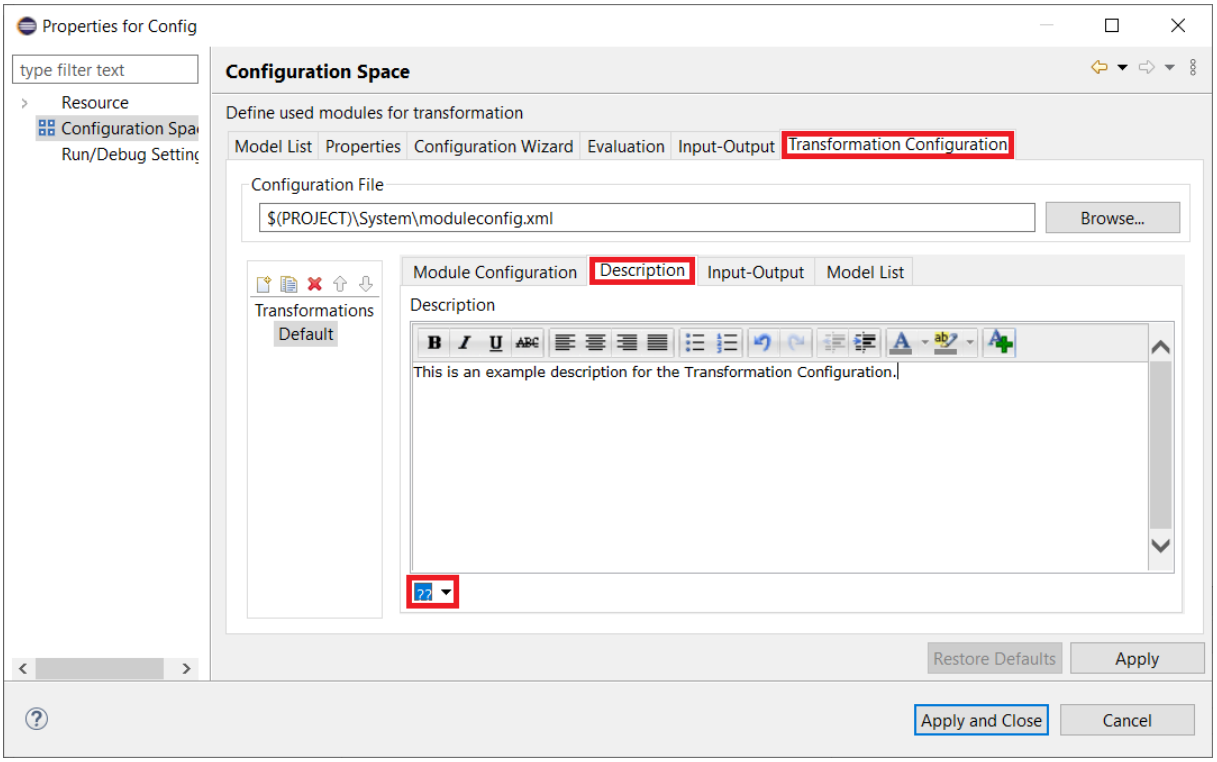
In the transformation module selection dialog a name has to be entered for the chosen transformation module. The module parameters are configured in the "Module Parameters" dialog opened when clicking on button Next.

Figure 6.20. Transformation module parameters



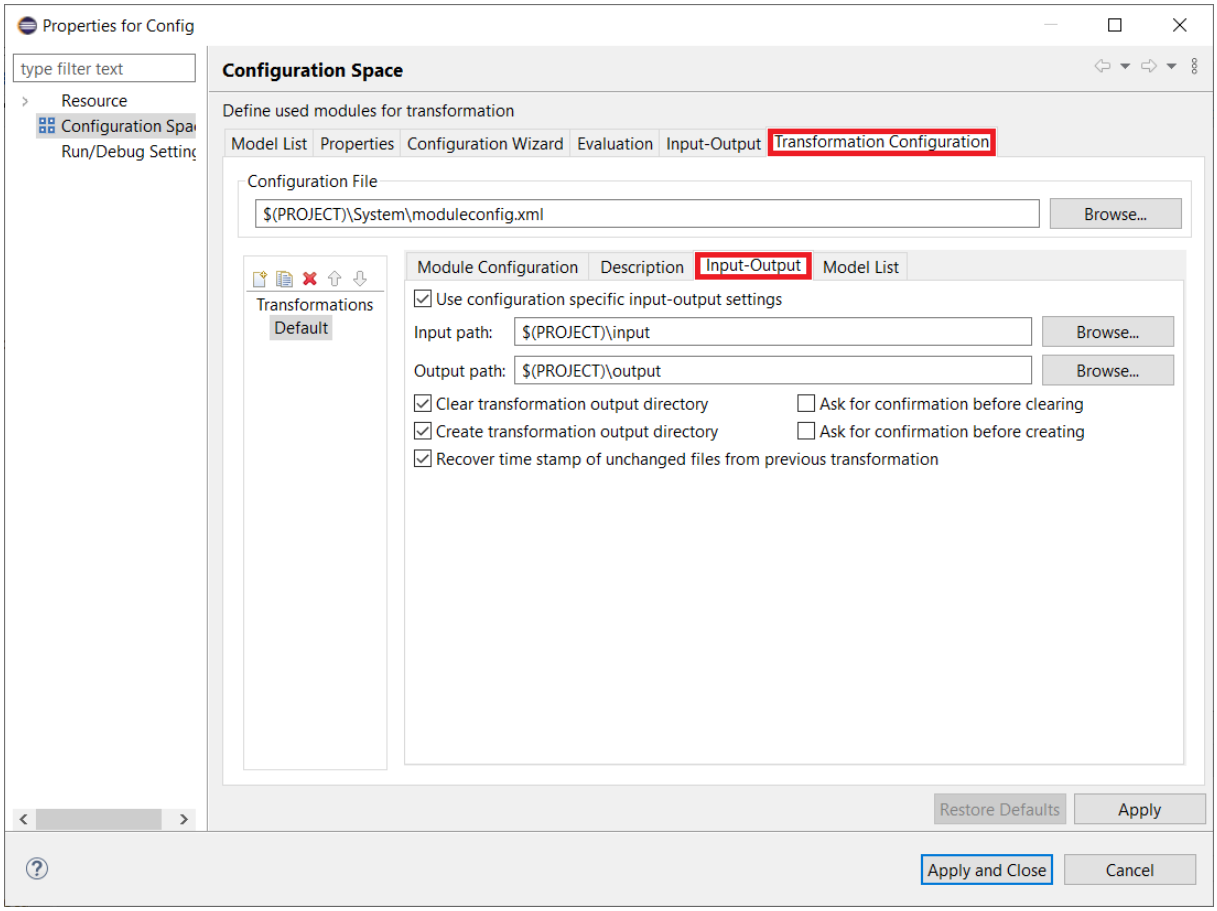
A transformation module can have mandatory and optional parameters. A module can not be added to the list of configured modules as long as there are mandatory parameters without a value. Module parameters have a name and a type. If there are values defined for a parameter, a list can be opened to choose a value from (see Figure 6.20, “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.

Figure 6.21. Configuration Space properties: Transformation Configuration



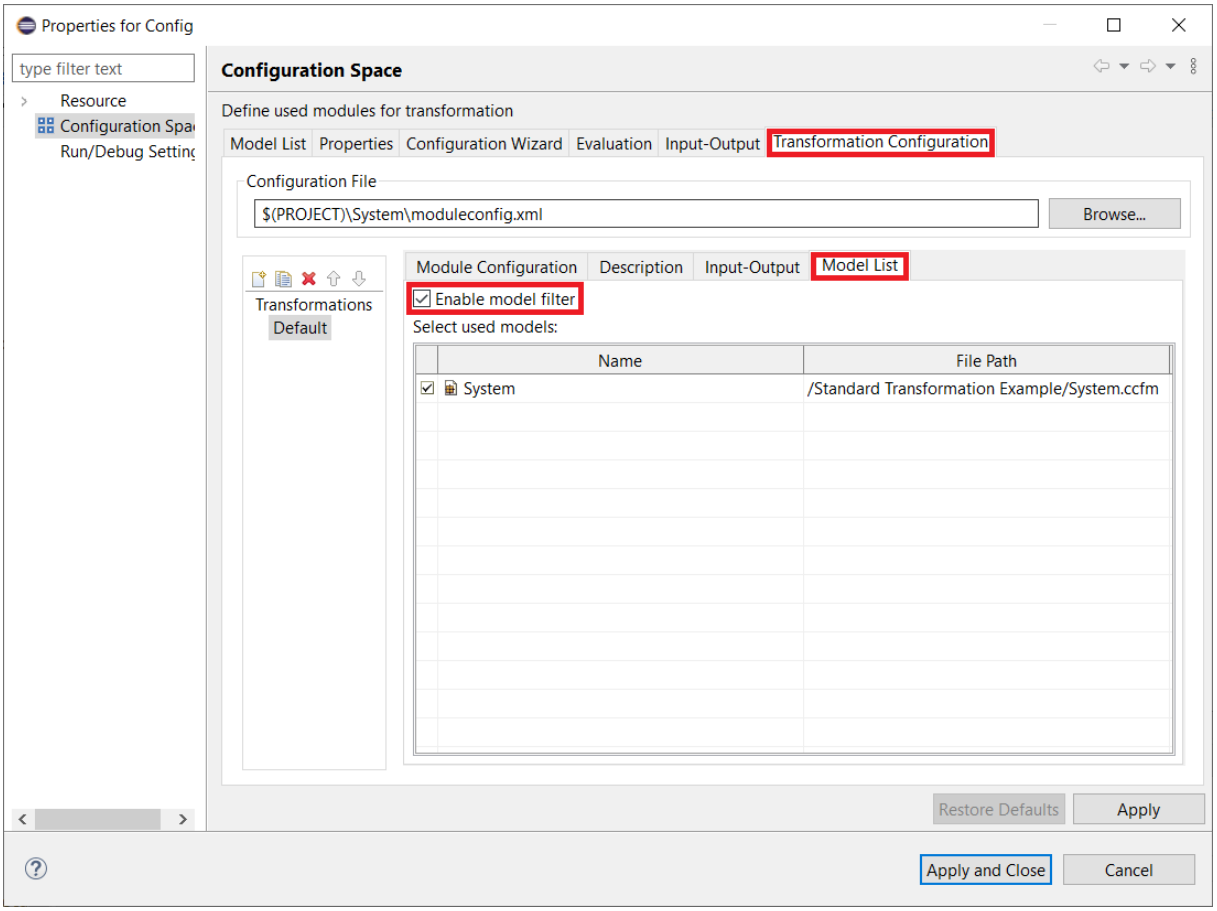
For collaboration purposes and for a better overview, it is possible to add a description to a Transformation. Therefore the Transformation description uses the same MIME type as the related Configuration Space. It is also possible to define different descriptions for different languages. You can switch the description language by clicking on the dropdown on the bottom left.

Figure 6.22. Configuration Space properties: Transformation Configuration



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 inFigure 4.5, “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*Transformation log file* 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* .

Figure 6.23. Configuration Space properties: Transformation Configuration



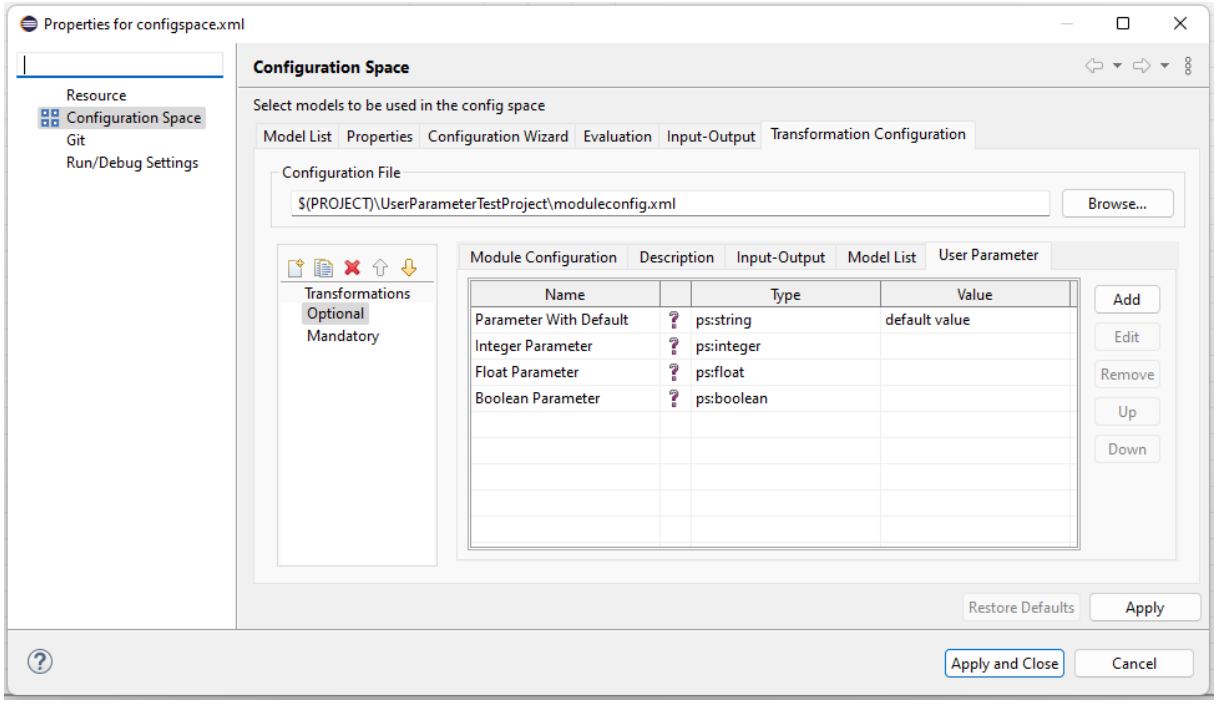
The *Model List* tab allows to specify a transformation configuration specific set of input models. The list can not contain more then the used models defined for the config space itself. It is not possible to remove feature models from the input model set. The selected input models will be processed by the defined transformation modules. The deselected input models are not known by the transformation modules and will be completely ignored during transformation. The variant evaluation will always use all input models as defined for the configuration space. The use of a transformation module configuration specific input model set can be enabled with the check button *Enable model filter*.

Note

Reducing the set of input models may have an unwanted impact in the transformation result.

Please see Section 5.9, “ Variant Transformation ” for more information on model transformation.

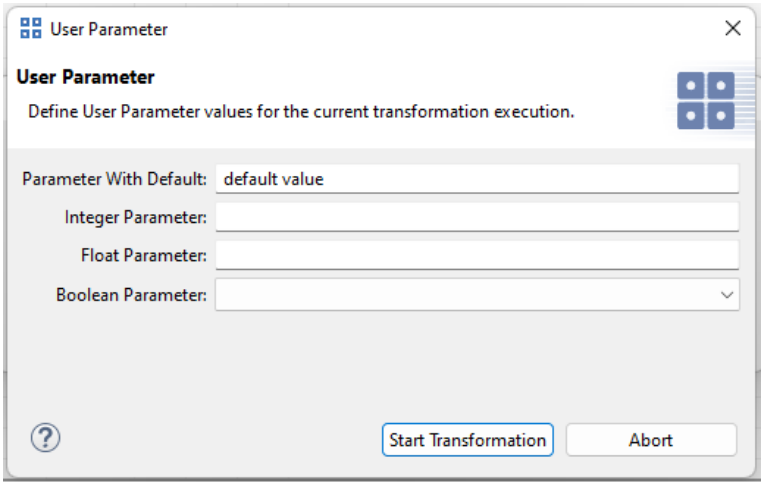
Figure 6.24. Configuration Space properties: Transformation Configuration



The *User Parameter* tab allows you to specify parameters that can be used to request user input before a transformation begins. These user parameters function like any other transformation variables, which means they can be set as values for transformation parameters and allow users to provide values for transformation parameters.

To add a parameter, use the "Add" button and specify the user parameter. A name must be given, as well as the parameter type. All other fields are optional. An optional parameter does not need to be specified by the user; it will receive an empty value if no value is given and no default value is defined.

Figure 6.25. Configuration Space properties: Transformation Configuration



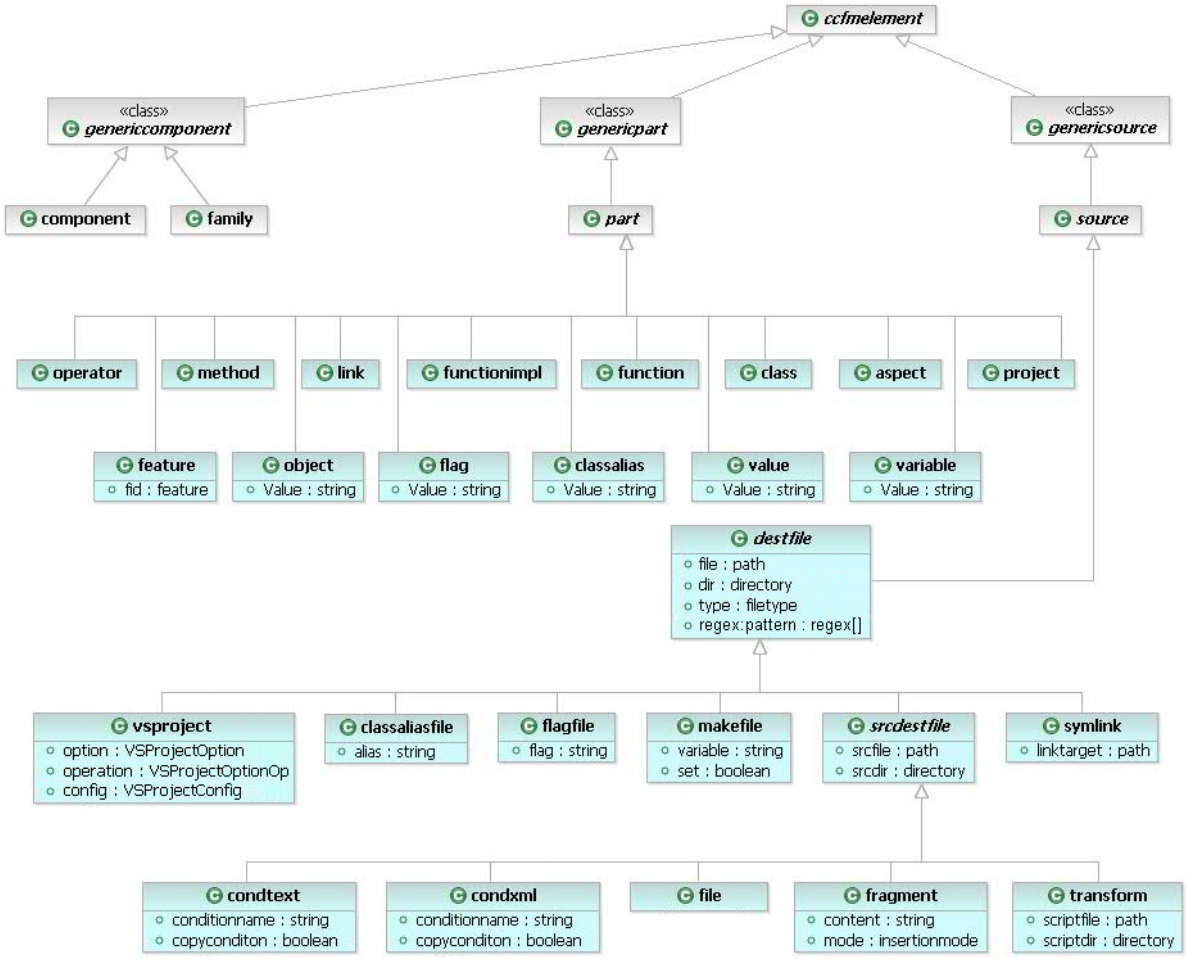
During the transformation, a dialog box appears asking the user to provide values for the parameters. Optional parameters do not need to be set. However, mandatory parameters, which are written in bold letters, must be given a value. At this point, the transformation has not started and can be aborted.

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 6.26, “The Standard Transformation Type Model”).

Figure 6.26. 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 seeSection 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 *Action List Generator* using the *Add* button. Name it for instance *Generate Standard Transformation Actionlist* .
- Add an *Action List Runner* module. Name it for instance *Execute Actionlist* . Usually there should be only one *Action List Runner* module, otherwise the action list gets executed twice.

  Note: If the standard transformation is used together with the*Makefile Generator* module to add content to one and the same file, then the*Action List Runner* module must not be placed before the*Makefile Generator* module. Otherwise all the content added to the Makefile by the*Action List Runner* module will be overwritten by the *Makefile Generator* module.

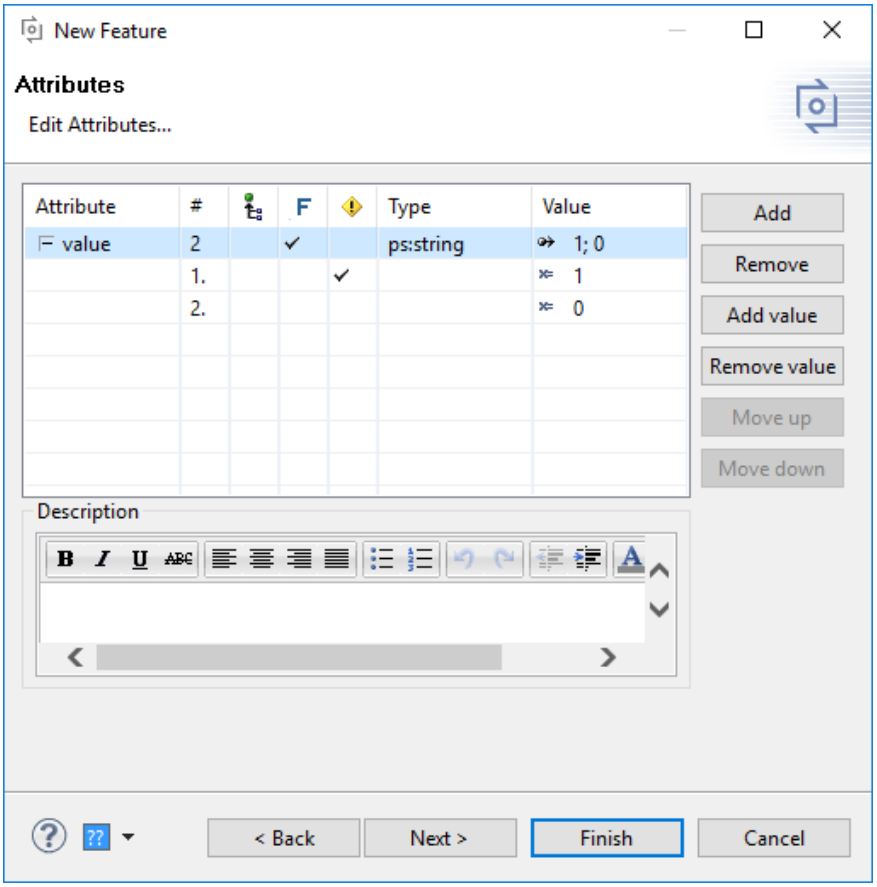
Providing Values for Part Elements

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 6.27, “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.

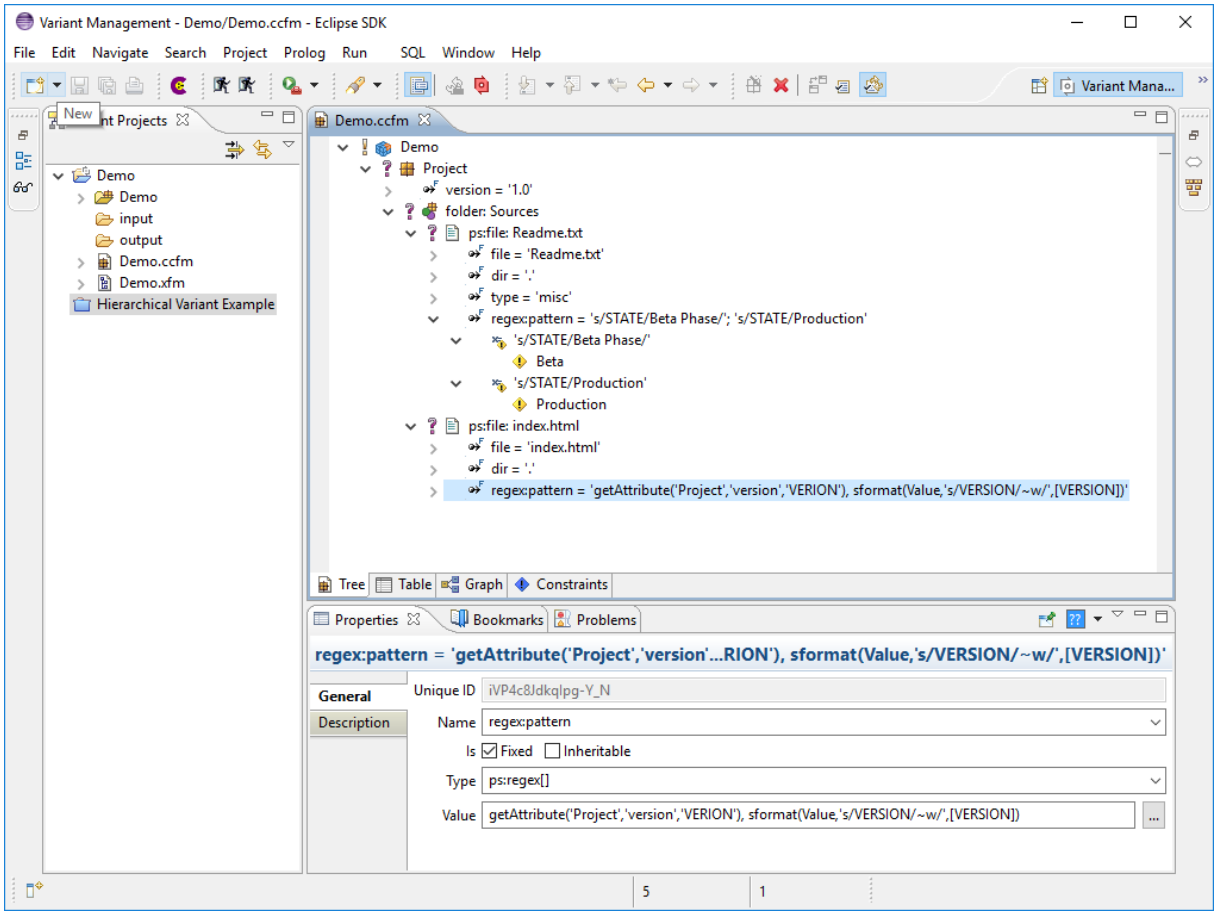
Figure 6.27. Multiple attribute definitions for Value calculation



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 6.28. 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:

```
/**
 * To set up JavaScript Transformation open configuration space properties
 * and go to "Configuration Space" -> "Transformation Configuration"
 * and add a JavaScript Transformation Module with this JavaScript.
 */

// global variables

var module = module_instance();

/**
 * Initialize this JavaScript transformation module.
 * This method is optional and does not need to be implemented.
 *
 * @param {IPVVariantModel} vdm
 *   The concrete variant description model.
 * @param {IPVModel[]} models
 *   The concrete feature and family models.
 *   This provides the full view of the current variant including all elements
 *   from instances, variant references and variant collections.
 * @param {java.util.Map<String, String>} variables
 *   The variables of the transformation configuration.
 * @param {java.util.Map<String, String>} parameter
 *   The parameter of the JavaScript transformation module.
 * @param {org.eclipse.core.runtime.IProgressMonitor} monitor
 *   The monitor for this operation
 *
 * @return {ClientTransformStatus} the status of this module method
 */
function init(vdm, models, variables, parameter, monitor) {

 var status = new ClientTransformStatus();
 status.setMessage(Constants().EMPTY_STRING);
 status.setStatus(ClientTransformStatus().OK);

 return status;
}

/**
```

```
 * Perform transformation preparation steps.
 *
 * This method is called after all modules have been initialized and before
 * any module is processed.
 *
 * This method is optional and does not need to be implemented.
 *
 * @param {org.eclipse.core.runtime.IProgressMonitor} monitor
 *   The monitor for this operation
 *
 * @return {ClientTransformStatus} the status of this module method
 */
function prepare(monitor){
 var status = new ClientTransformStatus();
 status.setMessage(Constants().EMPTY_STRING);
 status.setStatus(ClientTransformStatus().OK);

 return status;
}

/**
 * Do the work of this JavaScript transformation module
 *
 * @param {org.eclipse.core.runtime.IProgressMonitor} monitor
 *   The monitor for this operation
 *
 * @return {ClientTransformStatus} the status of this module method
 */
function work(monitor) {

 var status = new ClientTransformStatus();
 status.setMessage(Constants().EMPTY_STRING);
 status.setStatus(ClientTransformStatus().OK);

 var fo = null;

 try {
  var path = module.getVariable("OUTPUT");
  var filename = "FeatureList.txt";
  var outputfile = module.getParameter("outputfile");

  if (outputfile != null && outputfile != "") {
   fo = new java.io.FileWriter(new java.io.File(outputfile));
  } else {
   fo = new java.io.FileWriter(new java.io.File(path, filename));
  }

  var models = module.getModels();
  var steps = calculateWork(models);
  monitor.beginTask("Print Features",steps);

  for (var index = 0; index < models.length; index++) {
   // convert to pure::variants model
   var model = new IPVModel(models[index]);
   // check if model is a concrete feature model
   if (model.getType().equals(ModelConstants().CFM_TYPE) == true) {
    // convert to feature model
    var fmodel = new IPVFeatureModel(model);
    // get the root feature
    var root = fmodel.getRoot();
    // print features starting at root
    printFeatures(fo, root, monitor);
   }
  }
 } catch (e) {
  status.setMessage(e.toString());
  status.setStatus(ClientTransformStatus().ERROR);
 } finally {
   if(fo != null){
  fo.close();
   }
 }
```

```
 return status;
}

/**
 * Perform transformation post-processing steps.
 *
 * This method is called after all modules have been processed and before any
 * module is cleaned up, in reverse order. The first module on which
 * {@link #prepare(IProgressMonitor)} has been called is the last on which
 * this method is called.
 *
 * This method is optional and does not need to be implemented.
 *
 * @param {org.eclipse.core.runtime.IProgressMonitor} monitor
 *   The monitor for this operation
 *
 * @return {ClientTransformStatus} the status of this module method
 */
function postpare(monitor){
 var status = new ClientTransformStatus();
 status.setMessage(Constants().EMPTY_STRING);
 status.setStatus(ClientTransformStatus().OK);

 return status;
}

/**
 * Finalize JavaScript transformation module
 *
 * @param {org.eclipse.core.runtime.IProgressMonitor} monitor
 *   The monitor for this operation
 *
 * @return {ClientTransformStatus} the status of this module method
 */
function done(monitor) {
 var status = new ClientTransformStatus();
 status.setMessage(Constants().EMPTY_STRING);
 status.setStatus(ClientTransformStatus().OK);

 return status;
}

function calculateWork(models) {
 var total = 0;
 for (var index = 0; index < models.length; index++) {
  // convert to pure::variants model
  var model = new IPVModel(models[index]);
  // check if model is a concrete feature model
  if (model.getType().equals(ModelConstants().CFM_TYPE) == true) {
   total += model.getElementList().size();
  }
 }
 return total;
}

/**
 * Print the information of a feature to the output file
 * and do to the children.
 * @param {java.io.FileWriter} fo
 *   The file writer in order to write the information
 * @param {IPVElement} element
 *   The element to print
 * @param {org.eclipse.core.runtime.IProgressMonitor} monitor
 *   The monitor for this operation
 */
function printFeatures(fo, element, monitor) {
 monitor.subTask("Print: " + element.getName());

 // print information to file
 fo.append("Visible Name: ");
 fo.append(element.getVName());
```

```
 fo.append(Constants().NEWLINE_STRING);
 fo.append("Unique Name:  ");
 fo.append(element.getName());
 fo.append(Constants().NEWLINE_STRING);
 fo.append(Constants().NEWLINE_STRING);

 monitor.worked(1);

 // go to children
 var children = element.getChildren();
 var iterator = children.iterator();
 while (iterator.hasNext() == true && monitor.isCanceled() == false ) {
  var child = new IPVElement(iterator.next());
  printFeatures(fo, child, monitor);
 }
}
```

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

- init()

  This method is optional. Necessary work can be 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. All this informations can also be retrieved from the JavaScript transformation module using getter functions.

- prepare()

  This method is optional. It is called after all transformation modules are initialized and before any transformation module is performed.

- work()

  Does the whole transformation work.

- postpare()

  This method is optional. This method is called after all modules have been processed and before any module is cleaned up, in reverse order. The first module on which prepare has been called is the last on which this method is called.

- done()

  This method is optional. 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. The function*module_instance()* provides access to the transformation module instance, which is running the JavaScript transformation. This gives access to the transformation module API.

Evaluate PVSCL rules in a JavaScript Transformation

In general, one of the easiest ways to create variant specific assets is through the use of JavaScript transformations. It is possible to evaluate pvSCL expressions in the context of the currently transformed variant from within JavaScript transformations. We made this API as simple as possible, meaning all the cumbersome stuff of setting up the evaluator as well as putting each and every parameter correctly is hidden. You just take the expression and give it as parameter into one of two functions depending on having a rule (e.g. restriction or constraint) or a calculation.

The following two examples show the simple usage:

```
Evaluator.rule('Feature_A');

Evaluator.calculation('5*6');
```

The first line will evaluate to true or false depending on the selection state of the feature *Feature_A* and result of the second line is going to be*30* . As you see very simple. Thus you may concentrate on implementing the heart of the transformation and not fiddling around on the evaluator in order to set it up in the right manner.

Side note: If you want to have full access to the correct initialized evaluator, you can call

```
Evaluator.getDefault();
```

With the object returned by this call, you have the evaluator for the currently transformed variant in hand. See the related Java API reference in the SDK documentation for more information.

6.3.4. 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. SeeSection 9.8, “ 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 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 "Foo" then this condition always will fail if evaluated in the context of a linked variant. The correct condition would be "Link1:Foo", if linked below the link element with unique name "Link1".

6.3.5. 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 *Reuse Transformation* configuration, if defined. From Configuration Space otherwise.
Output Directory	Used from the *Reuse Transformation* configuration, if defined. From Configuration Space otherwise.

Create Output Directory	Used from the *Reuse Transformation* configuration, if defined. From Configuration Space otherwise.
Cleanup Output Directory	Used from the *Reuse Transformation* configuration, if defined. From Configuration Space otherwise.
Create Output Directory	Used from the *Reuse Transformation* configuration, if defined. From Configuration Space otherwise.
Confirm Create Output Directory	Used from the *Reuse Transformation* configuration, if defined. From Configuration Space otherwise.
Confirm Cleanup Output Directory	Used from the *Reuse Transformation* configuration, if defined. From Configuration Space otherwise.
Recover Timestamps	Used from the *Reuse Transformation* configuration, if defined. From Configuration Space otherwise.
Force Transformation	Always true, because decision was made by user before running *Reuse Transformation* 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 *Reuse Transformation* configuration.

6.3.6. 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 .

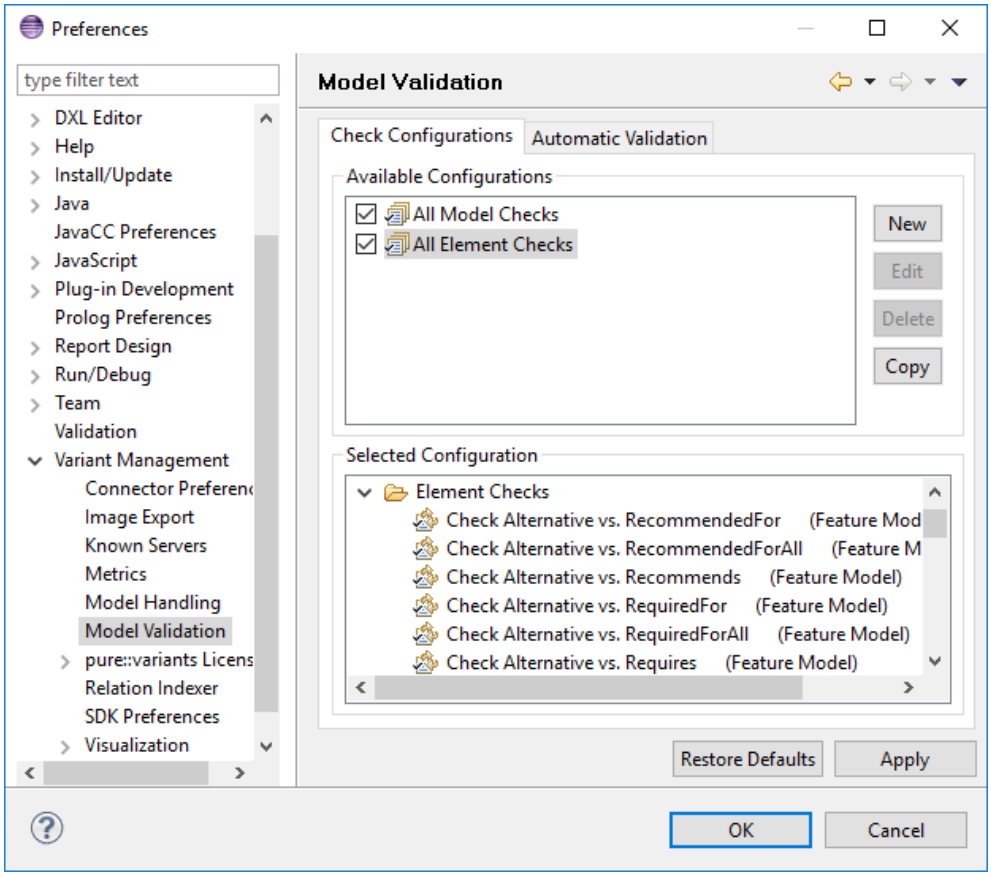
6.4.2. Model Check Framework

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 (seeFigure 6.29, “Model Validation Preferences Page”).

Figure 6.29. 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.

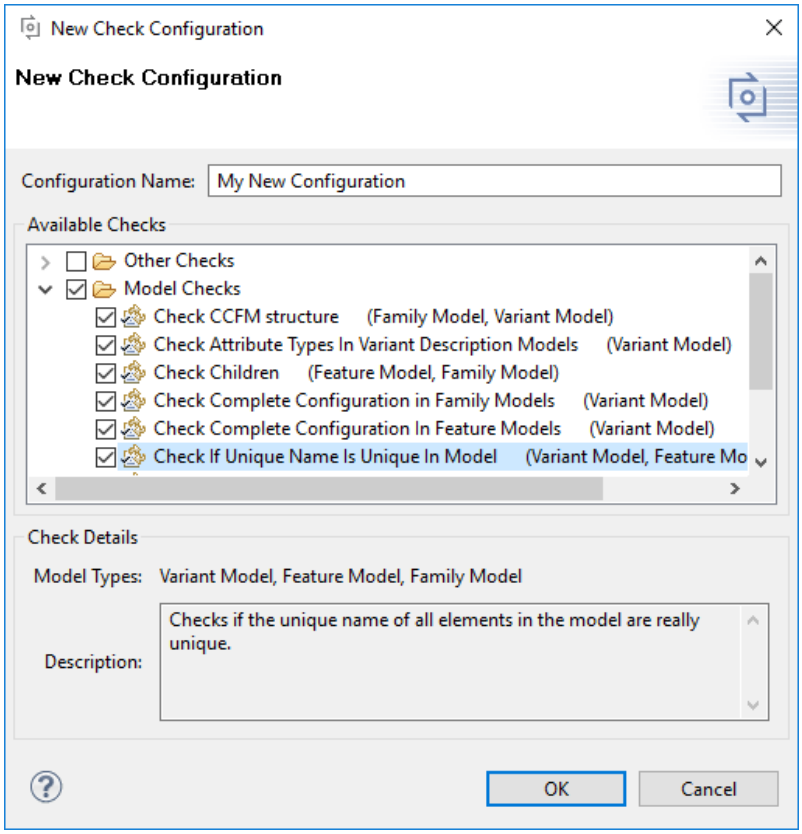
A model check configuration is activated by selecting it in the **Available Configurations** list. If more than one configuration is selected, the checks from all selected configurations are merged into one set that becomes activated.

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 (seethe 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 6.30, “New Check Configuration Dialog” .

Figure 6.30. 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.

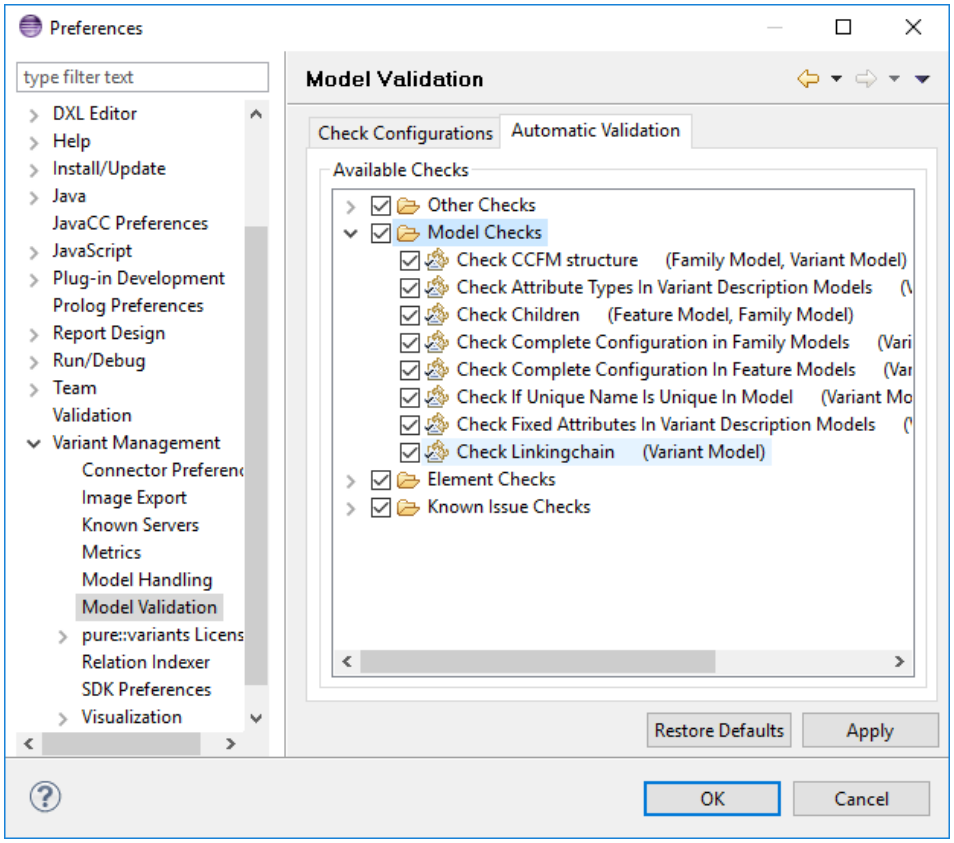
The same dialog appears for editing and copying check configurations using the **Edit** and **Copy** buttons. Only non-default configurations can be edited.

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 6.31, “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 6.31. Automatic Model Validation Preferences Page



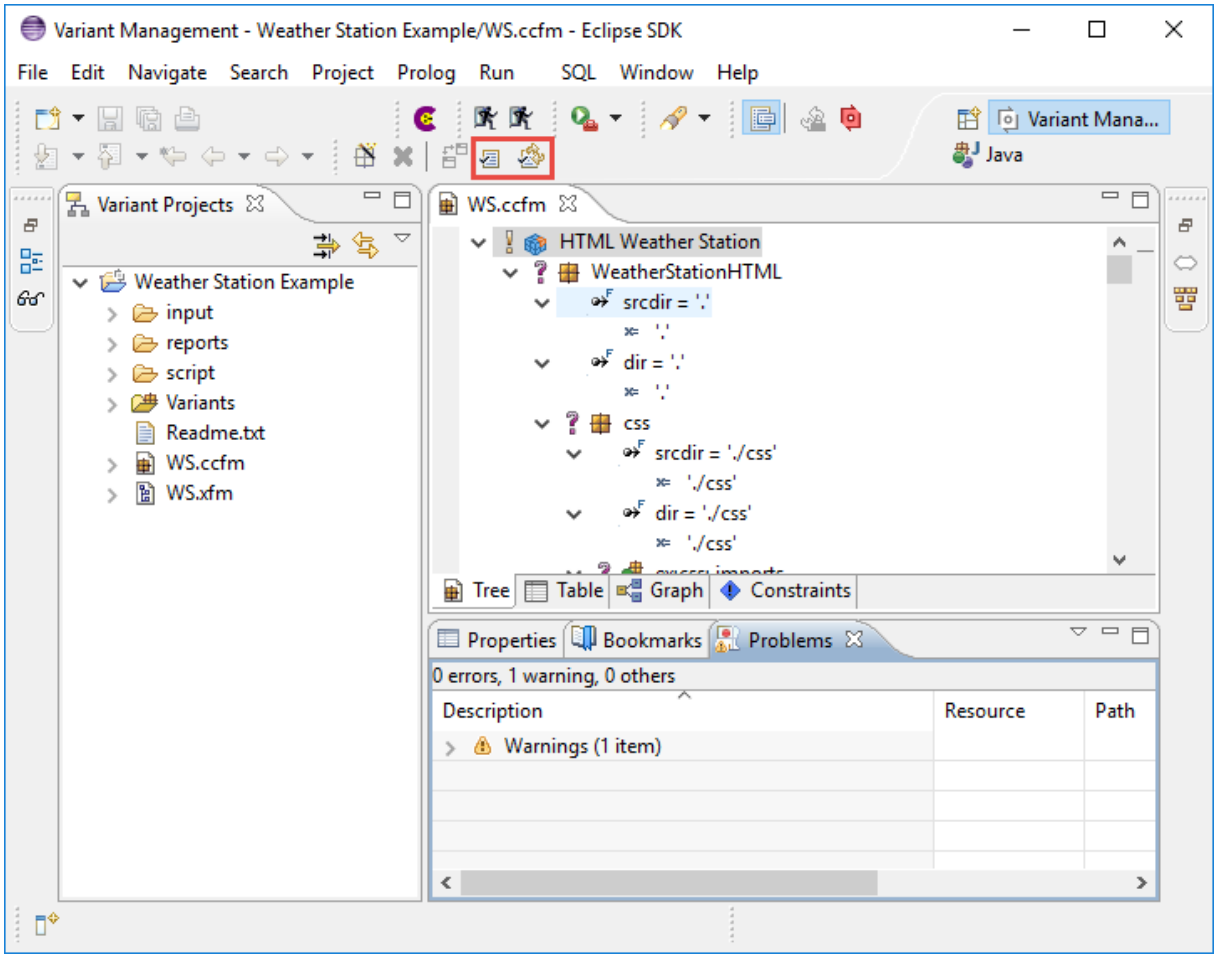
The **Available Checks** tree shows all known checks independently from the selected check configuration. Clicking on the check box of a check toggles the automatic validation state of the corresponding check. Clicking on the root of a sub-tree toggles all checks of this sub-tree.

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 6.32. 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.

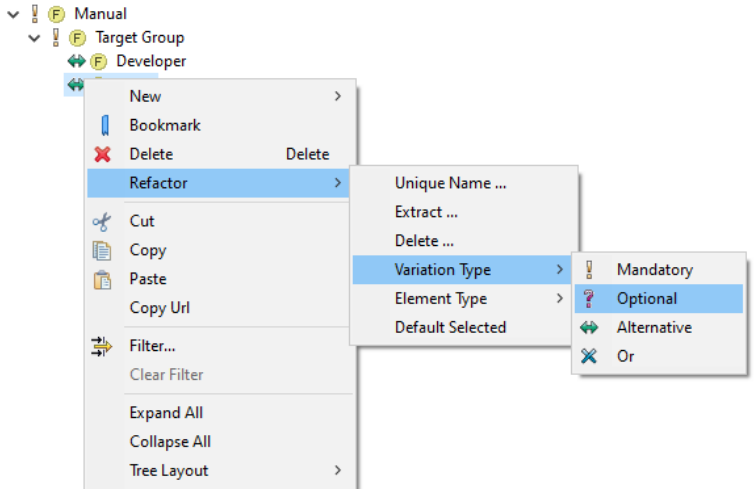
The result of a model check cycle is a list of problems found in the model. These problems are shown in the Problems view and as markers on the model. A list of quick fixes for a problem can be shown either by choosing "Quick Fix" from the context menu of the problem in the Problems view or by clicking 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 6.33, “Refactoring context menu for a feature” .

Figure 6.33. 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. Refactoring operations can include changes beyond the selected element on references in the selected scope. Therefor, refactoring operations can be long running actions. Furthermore it should be noted, that the appropriate rights have to be granted in all affected models and elements to ensure a successful operation. If access rights are missing a warning will be shown. Proceeding only on the models and elements where rights are sufficiently available, can result in references that are not updated.

The following list summarizes the available refactoring operations.

Table 6.2. Refactoring Operations

Operation on	Available Operations
Elements	Unique Name
	Extract
	Delete
	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

To extract an element or feature from a model, the target model has to exist. References to the extracted element or feature (via unique name or ID) will be updated in restrictions, constraints, and calculations with respect to the selected scope, e.g. the enclosing project, the project and all referenced projects, or the whole workspace. Depending on the scope and the amount and size of items to process, the refactoring operations can be long running actions.

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. Therefor 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

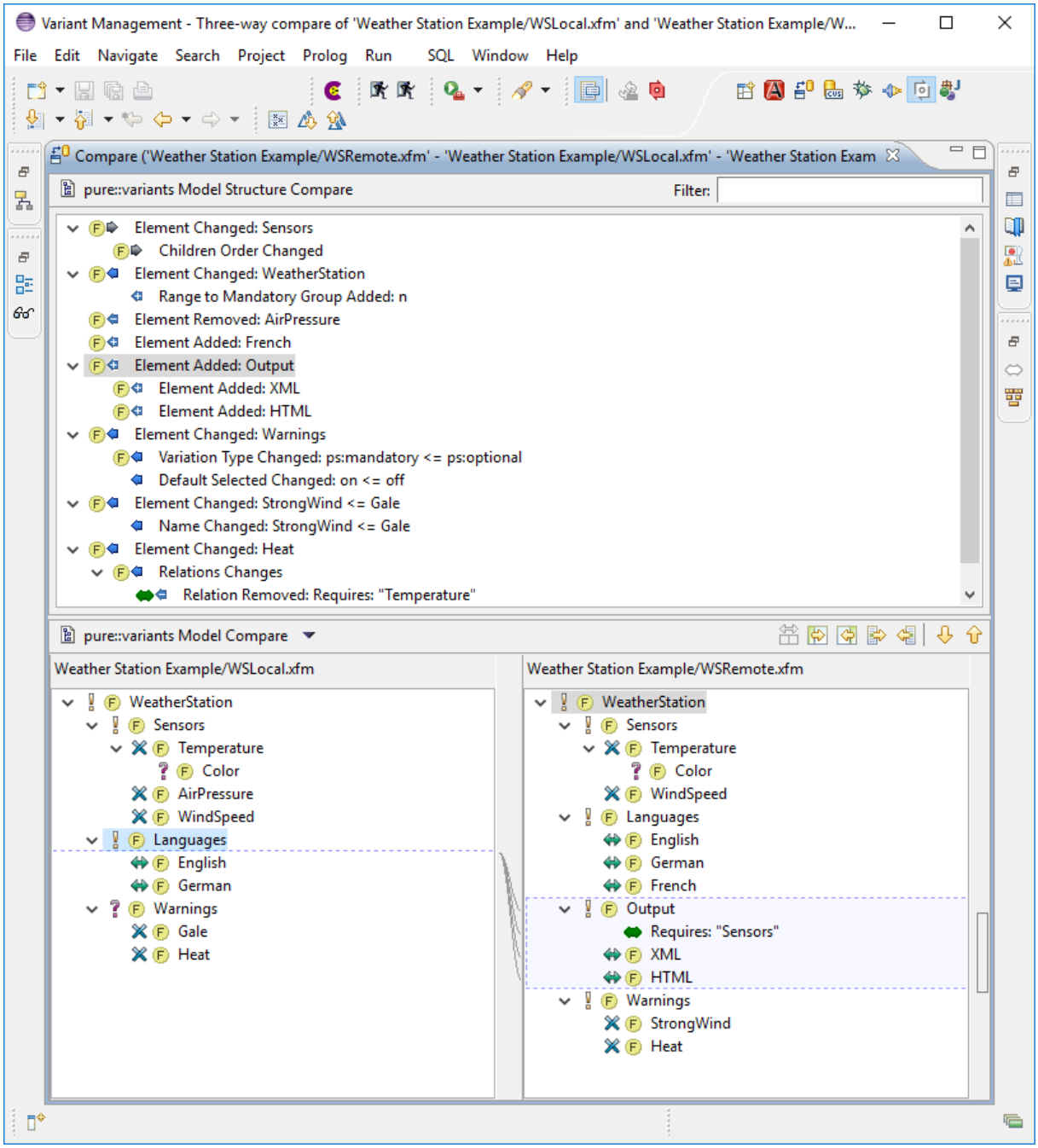
In three-way comparisons it may occur that an incoming and an outgoing patch conflict with each other. In general the model compare editor distinguishes between fatal conflicting patches and warning conflicts. In the tree viewer conflicts are red-colored. A fatal conflict is for example an element change 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 6.34, “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 6.34. Model Compare Editor



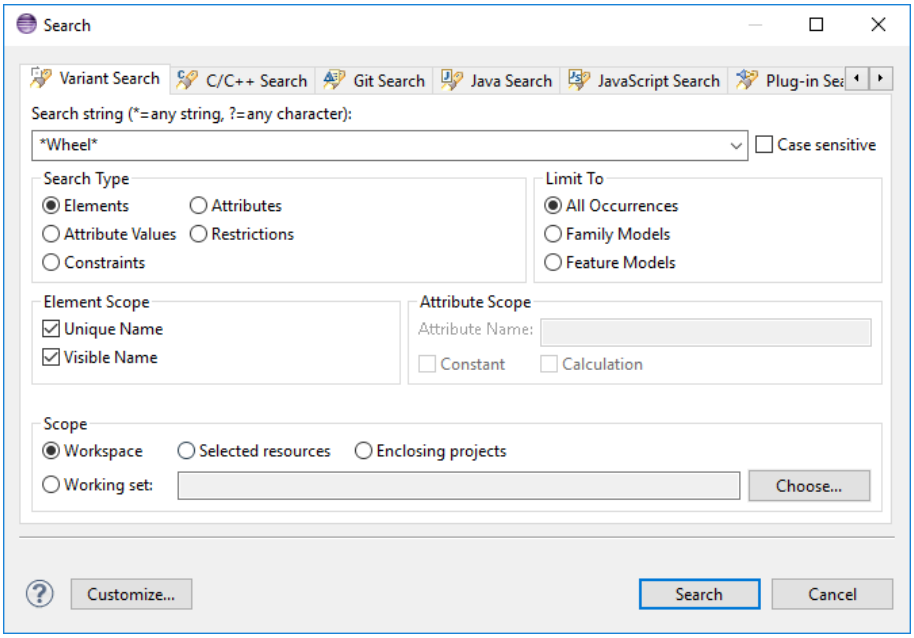
6.7. Searching in Models

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 6.35. 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 wild cards "*" and "?".

Wild card	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

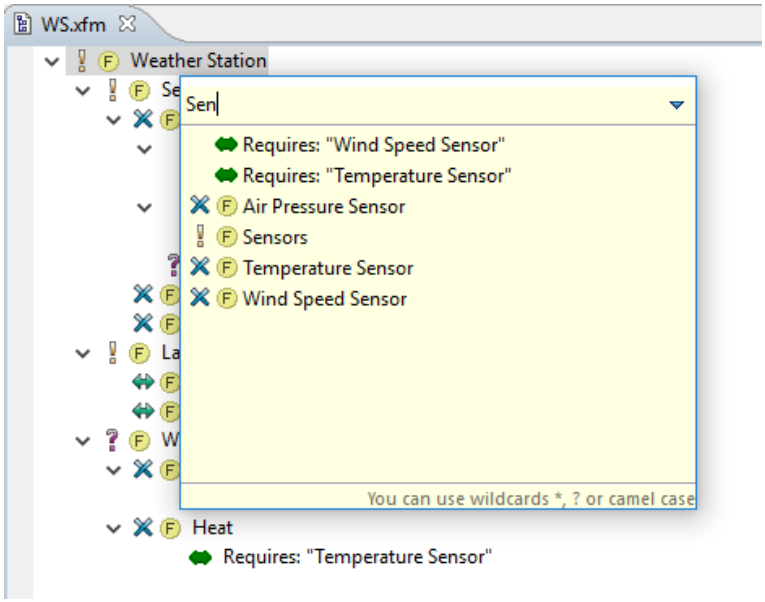
The results of the search are listed in the Variant Search view supporting a tree and table representation and a search result history. For more information about the Variant Search view see Section 7.4.3, “ Search View ” .

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**C-TRL+O** opens a small window with a sorted and filtered list of all model elements.Figure 6.36, “Quick Overview in a Feature Model” shows an example for the quick overview.

Figure 6.36. 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 wild cards 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 Camel Case notation. Camel case means that between each capital letter and the letter in front of it a* wild card 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. Analyse Models

Having a configuration space with a lot of variant description models it is very likely, that some of the variants are very similar or even equal. This section describes tasks, which enable the user to find similar variants and selection cluster within these variants.

6.8.1. Finding variant description models with similar selections

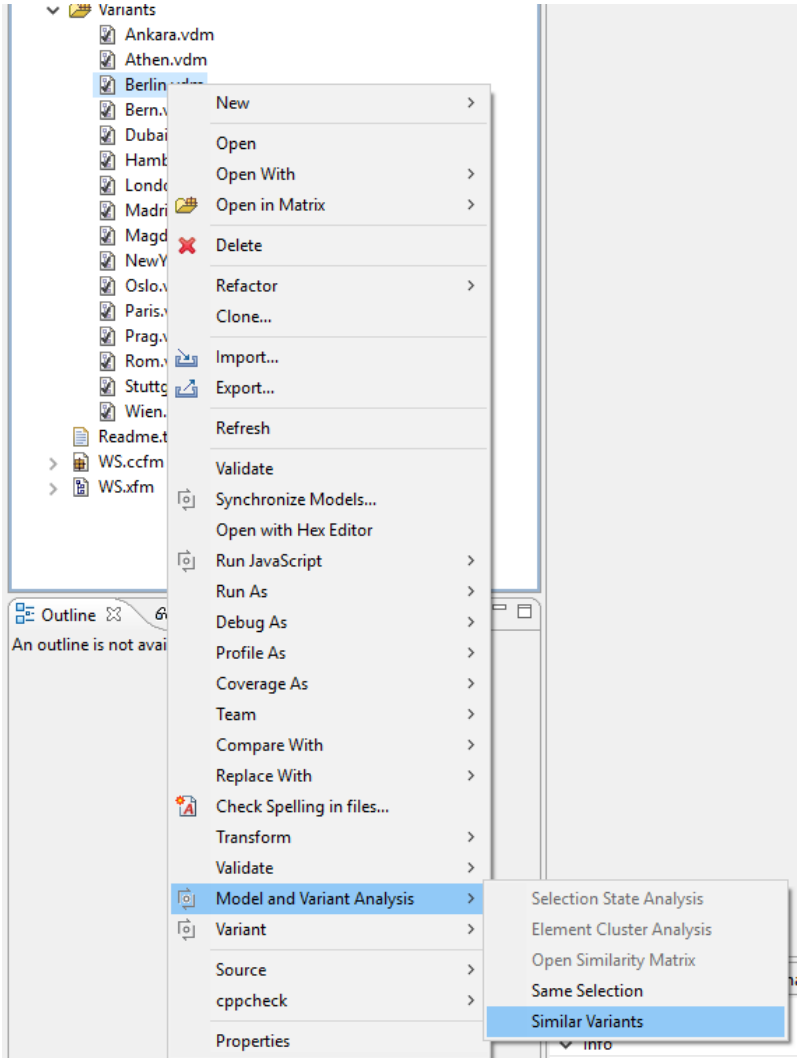
For finding variant description models with similar selections, pure::variants provides two solutions.

The first one starts with one vdm, selected by the user and calculates the similarity between this base vdm and all other vdms from the same configuration space. The second possibility is to calculate the similarity between a selection of vdms from a configuration space.

Finding variant description models similar to one base vdm

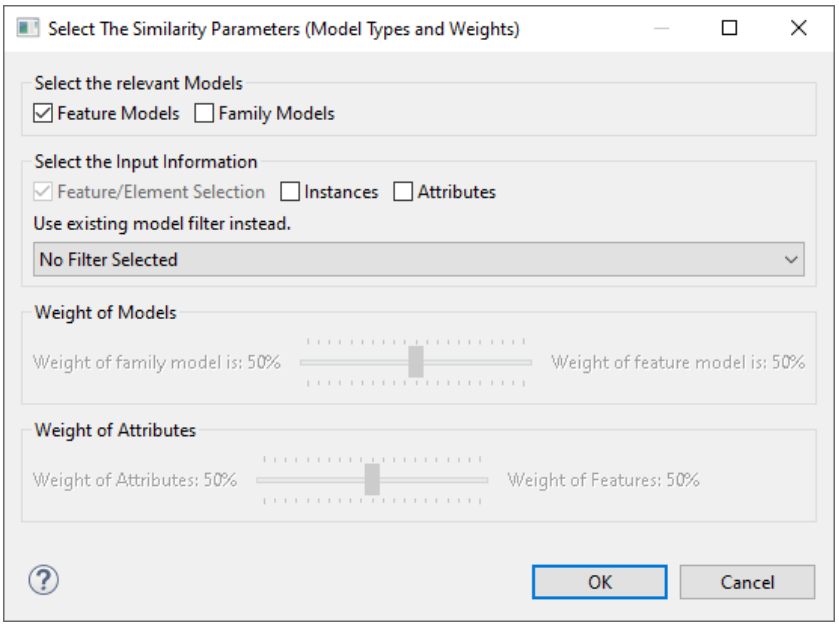
To calculate the similarity, between one vdm and the other vdms of the same configuration space, select the base vdm and start the calculation with the *Similar Variants* action in the *Model and Variant Analysis* sub menu of the context menu.

Figure 6.37.



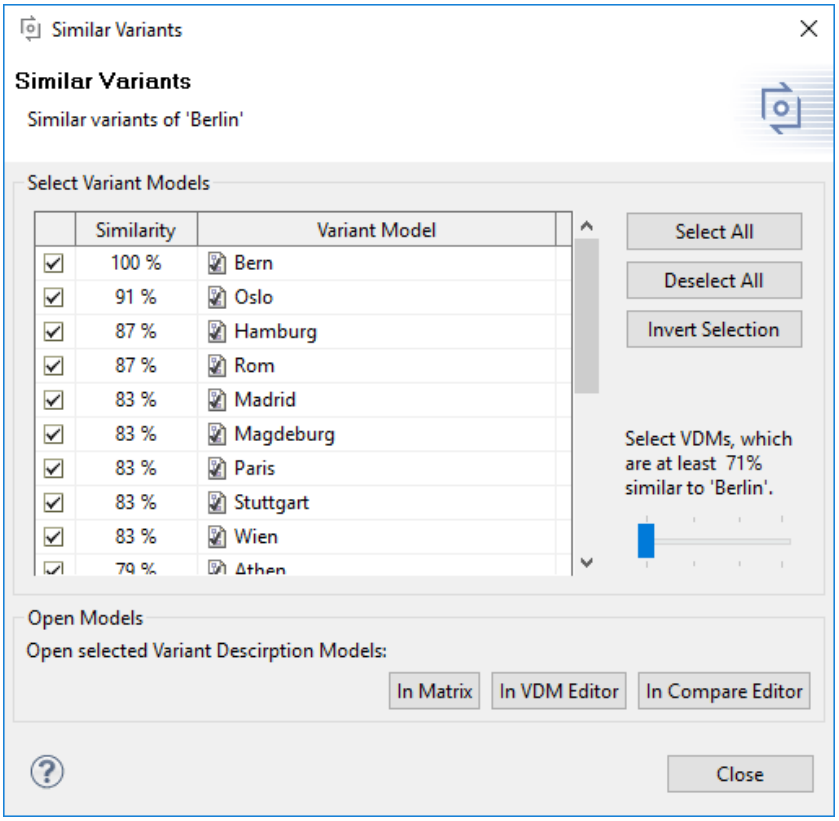
This opens a dialog, where the input data for the similarity can be configured. It allows the user to select which inout models and which inout elements shall be used for similarity calculation. Additionally it can be configured if attributes and instances shall be taken into account.

Figure 6.38. The similarity input configuration dialog



A dialog comes up, as soon as the calculation is finished. This dialog shows all variants of the configuration space and the similarity to the base vdm in percent. This dialog allows the user to select vdms for further analysis.

Figure 6.39. The similarity calculation result dialog



The selected vdms can be opened using one of the buttons in the lower part of the dialog.

- *In Matrix* opens the selected vdms in the matrix editor. An already open matrix editor is reused.
- *In VDM Editor* opens each selected vdm in a variant description model editor.

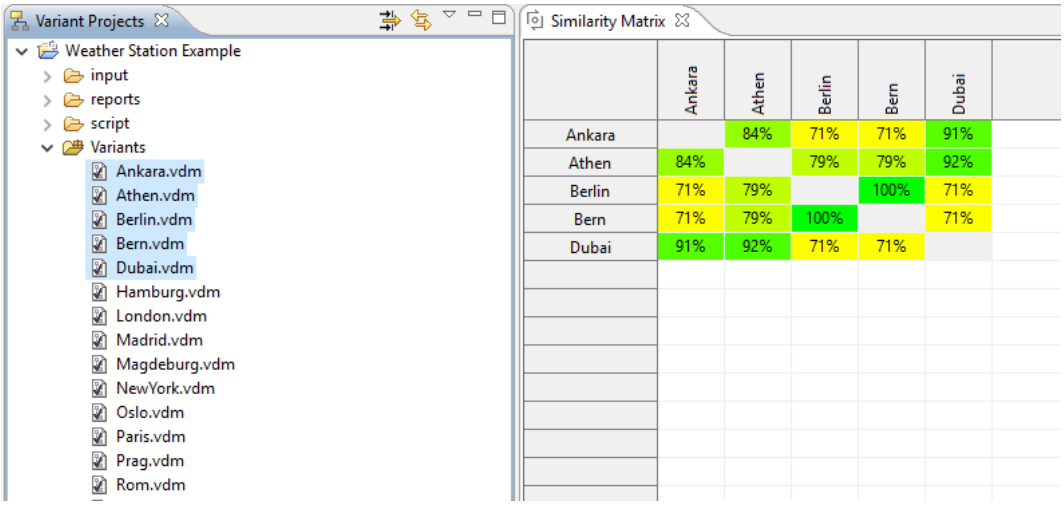
- *In Compare Editor* opens one compare editor for each selected vdm, which shows the compare result between the selected vdm and the base vdm.

Calculating similarity between multiple variant description models

To calculate the similarity between a selection of variant description models from one configuration space the action *Open Similarity Matrix* in the *Model and Variant Analysis* sub menu in the context menu is used. This action starts the calculation of the similarity between all selected vdms. It is also possible to start this action for the whole configuration space by selecting the configuration space folder. The used algorithm is the same as for the*Same Selection* and *Similar Variants* actions.

The result of the calculation is presented in the *Similarity Matrix* . Each row shows the similarity values between the vdm represented by this row and the vdms represented by the columns.

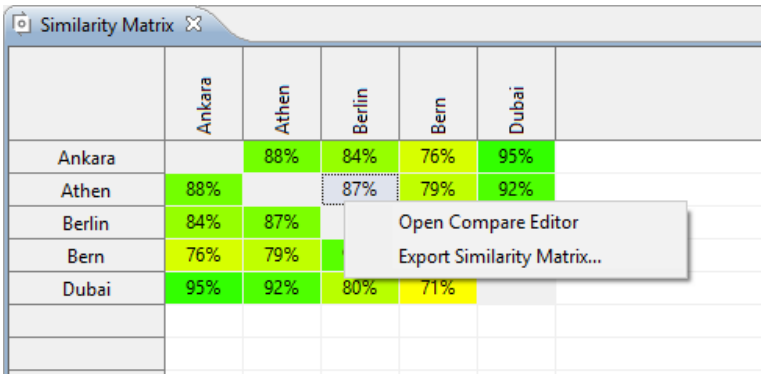
Figure 6.40. Similarity Matrix



The *Open Compare Editor* action from the context menu of one similarity value is used to have a detailed look on the differences between the corresponding vdms. This opens the vdms in a compare editor.

With the *Export Similarity Matrix...* action from the context menu the similarity matrix can be exported to a Microsoft Excel document.

Figure 6.41.

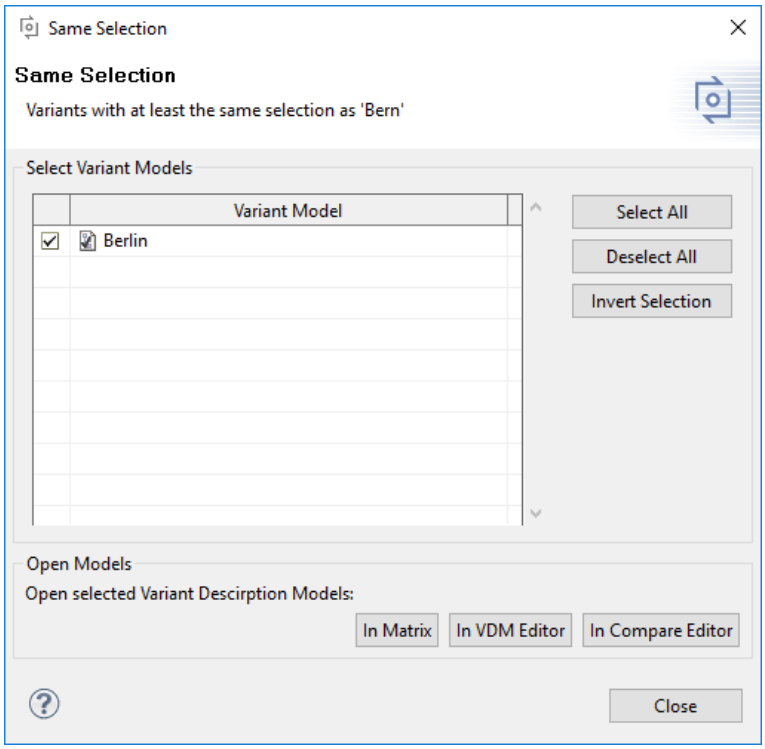


6.8.2. Finding variant description models with the same selection

The algorithm, which is used here is the same algorithm used in the *Similar Variants* analysis. The scope is just set to 100% similarity. The action is used the same way like the*Similar Variant* action. Select one base vdm and start the calculation with the*Same Selection* action in the *Model and Variant Analysis* sub menu of the context menu.

The same selection result dialog comes up, which shows all variants from the same configuration space, that have the same selections as the base vdm. This dialog allows the user to select vdms for further analysis.

Figure 6.42. The same selection result dialog



The selected vdms can be opened using one of the buttons in the lower part of the dialog.

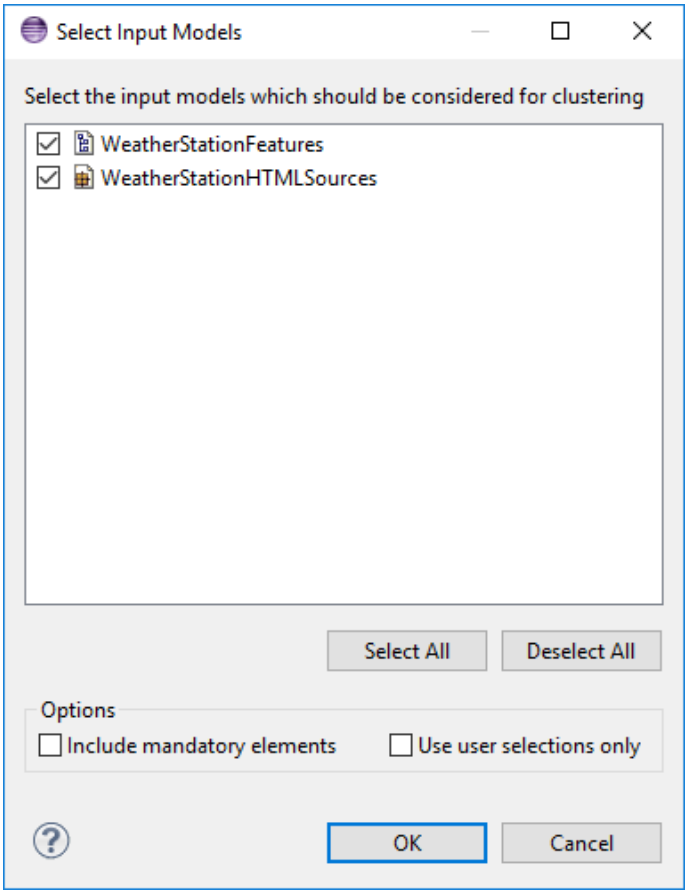
- *In Matrix* opens the selected vdms in the matrix editor. An already open matrix editor is reused.
- *In VDM Editor* opens each selected vdm in a variant description model editor.
- *In Compare Editor* opens one compare editor for each selected vdm, which shows the compare result between the selected vdm and the base vdm.

6.8.3. Find elements with the same selection states in all variant description models

To find elements, which selection state is equal in all variants pure::variants provides the action *Element Cluster Analysis* from the context menu of several selected vdms or the whole configuration space folder. Having the same selection state in all selected variants means, that an element a has the same selection state as element b for all selected variants. It does not mean, that element a and element b are selected or deselected in all checked variants. The selection state may change from variant to variant.

This action brings up the input model selection dialog. This dialog allows the user to define the scope of the analysis. Deselected input models will not contribute to the analysis. The option *Include mandatory elements* includes mandatory elements into the calculation, since this elements are automatically selected, they are ignored during the analysis by default. Option*Use suer selection only* causes the analysis to ignore all automatic selections during the calculation and just consider selection made by an user.

Figure 6.43. The same selection result dialog



The calculation result is presented in the view *Element Cluster* . Elements, which are having the same selection state in all vdms are grouped in element cluster. There are 4 types of clusters:

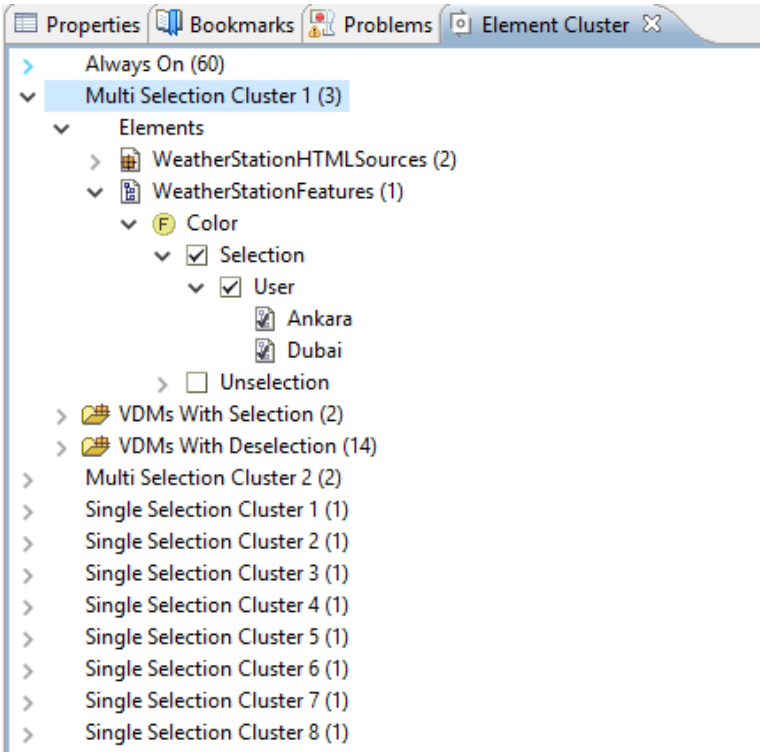
- The cluster *Allways On* lists all elements, which are selected in all considered variants.
- The cluster *Allways Off* lists all elements, which are never selected in the considered variants.
- The cluster *Multi Selection Cluster* lists variable elements. A Multi Selection Cluster contains more than one element.
- The cluster *Single Selection Cluster* lists variable elements. A Single Selection Cluster contains exactly one element.

Each cluster contains the following informations. The number of elements in that cluster, shown in brackets after the cluster name. The elements grouped by the input models and their selector. For each selector the vdm is shown.

Besides the elements for each vdm is shown, if the elements of that cluster are selected or deselected.

Hint: pure::variants navigates to the elements in the input models after double clicking on the elements in the result view.

Figure 6.44. The same selection result dialog



For further analysis the result view provides some actions. For each cluster a filter can be created using the *Create Filter for Cluster Elements* action in the context menu on a cluster tree item. This creates a filter, which can be used in all pure::variants editors to set the focus on the elements of that cluster.

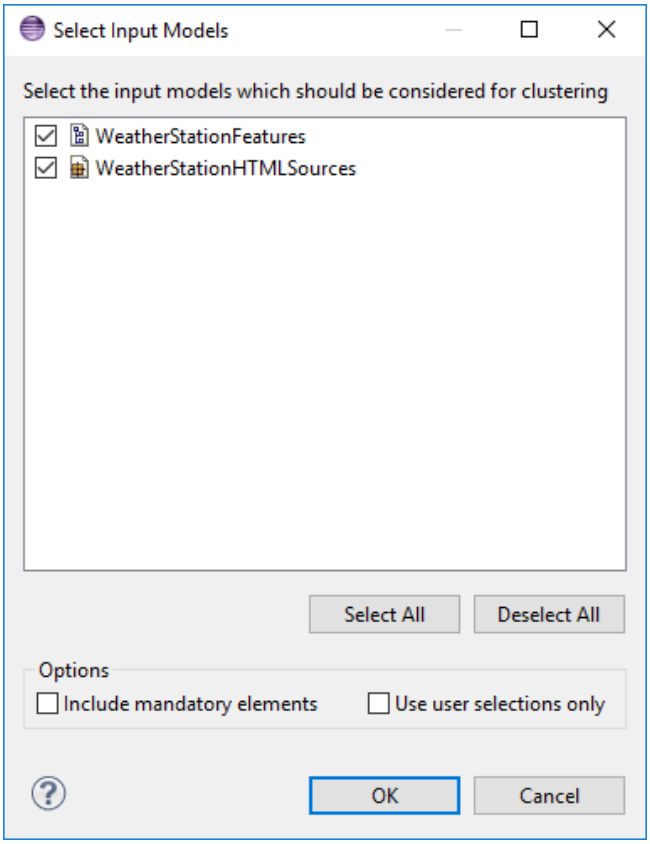
The action *Export Result to CSV* exports the result data to a csv file, which can be used for further data analysis outside pure::variants. The same output csv can be created using a transformation. The transformation module is called *Element Cluster Report* and has the same options as the dialog described above.

6.8.4. Find constant and variable elements in all variant description models

To find out which elements are variable and which elements are constant in all variants pure::variants provides the action *Selection State Analysis* from the context menu of several selected vdms or the whole configuration space folder. An element is considered to be variable, if it is at least selected in one vdm and not selected in all considered vdms.

This action brings up the input model selection dialog. This dialog allows the user to define the scope of the analysis. Deselected input models will not contribute to the analysis. The option *Include mandatory elements* includes mandatory elements into the calculation, since this elements are automatically selected, they are ignored during the analysis by default. Option*Use suer selection only* causes the analysis to ignore all automatic selections during the calculation and just consider selection made by an user.

Figure 6.45. The same selection result dialog



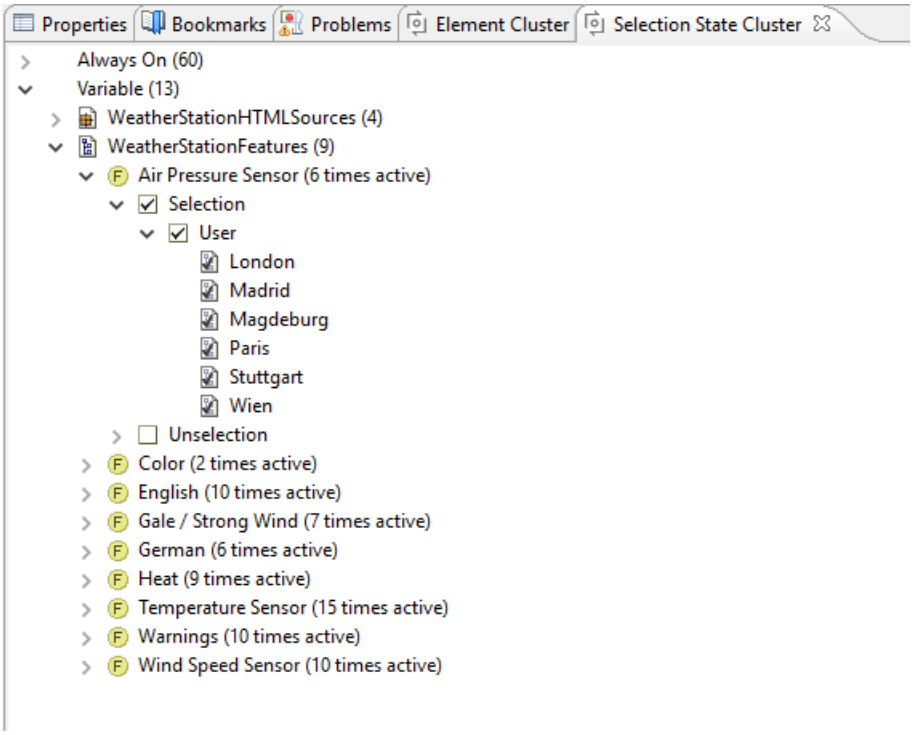
The calculation result is presented in the view *Selection State Cluster* . There are 3 types of cluster:

- The cluster *Allways On* lists all constant elements, which are selected in all considered variants.
- The cluster *Allways Off* lists all constant elements, which are never selected in the considered variants.
- The cluster *Variable* lists variable elements.

Each cluster contains the following informations. The number of elements in that cluster, shown in brackets after the cluster name. The elements grouped by the input models and their selector. For each selector the vdm is shown. In the label of the elements the number of selections is shown.

Hint: pure::variants navigates to the elements in the input models after double clicking on the elements in the result view.

Figure 6.46. The same selection result dialog



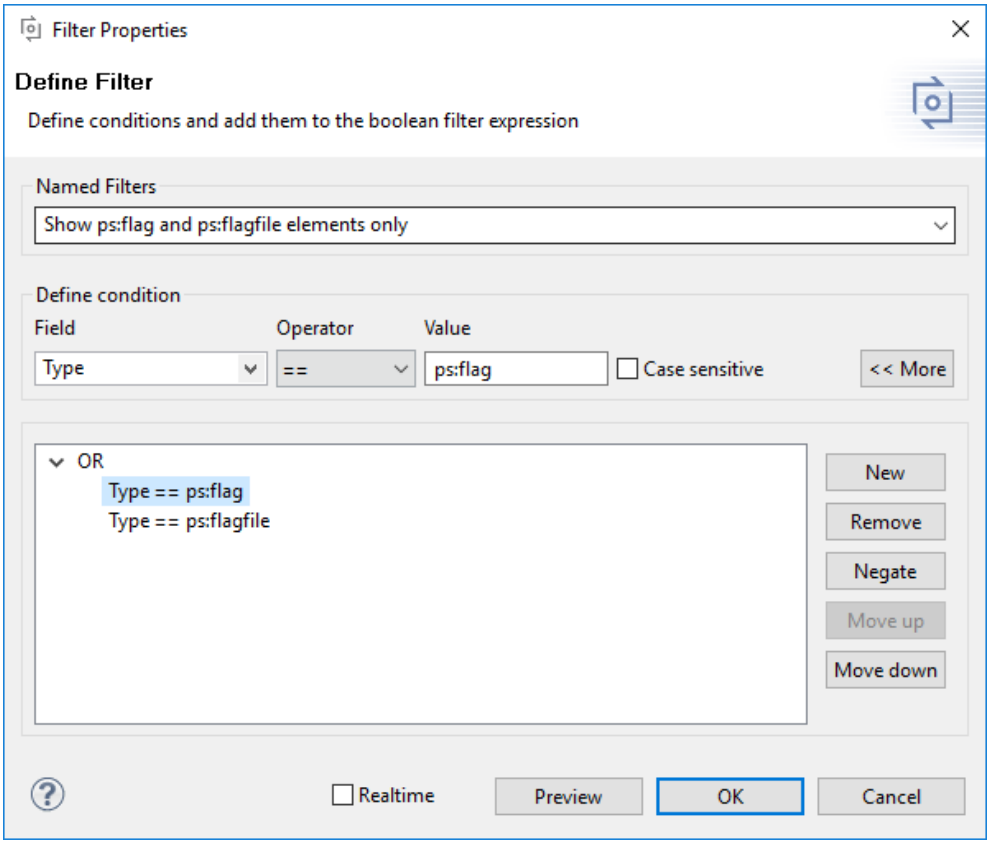
For further analysis the result view provides some actions. For each cluster a filter can be created using the *Create Filter for Cluster Elements* action in the context menu on a cluster tree item. This creates a filter, which can be used in all pure::variants editors to set the focus on the elements of that cluster.

The action *Export Result to CSV* exports the result data to a csv file, which can be used for further data analysis outside pure::variants.

6.9. 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 6.47. Filter definition dialog



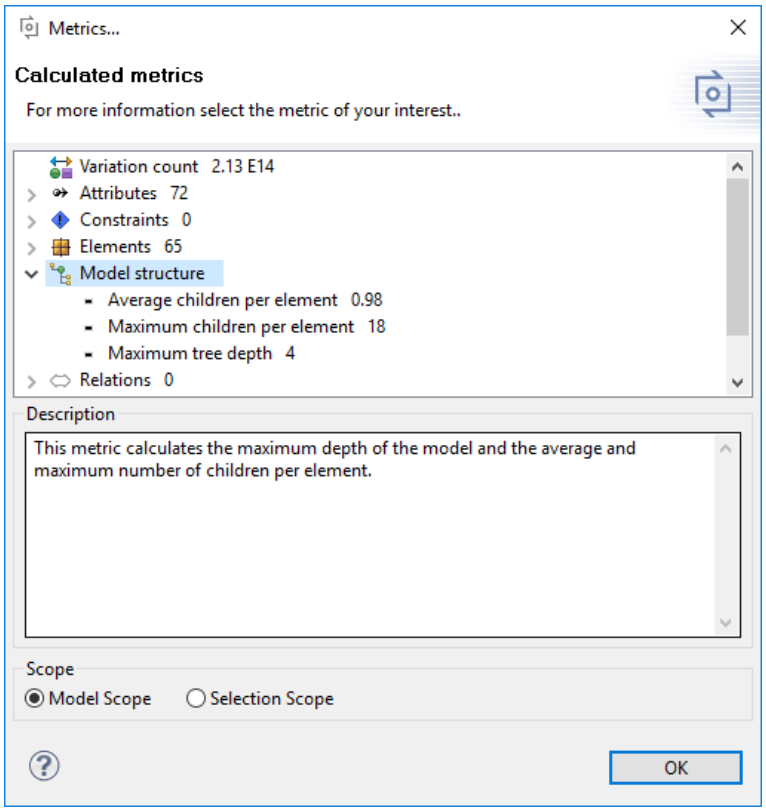
Arbitrarily complex filters based on comparison operations between feature/element properties (name, attribute values, etc.) and logical expressions (and/or/not) are supported. Comparison operations include conditions like equality and containment, regular expressions (matches) and checks for the existence of an attribute for a given element (empty/not empty). See Section 9.9, “ 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.

6.10. 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 6.48. 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 sub tree. 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.

6.11. 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 6.50, “Type Model Editor Example”).

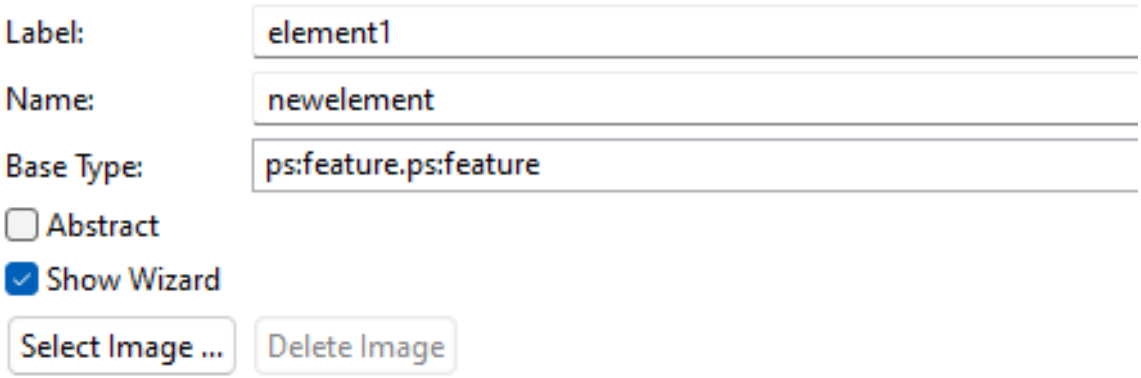
The Type Model Editor consists of two parts. The left part shows the list of types defined in the model, while the right part provides an editing area for the type selected on the left. Additionally the left part provides a context menu for adding and removing types of the type model.

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. Since 5.0.9 a custom icon can for element types can be defined. The editor allows to set and delete custom icons for element types.

The option *Show Wizard* enables or disables a specific new element wizard for the specified element type in the model editors. The new element wizard can be found in the context menu: New -> More -> the new element. We strongly recommend enabling it, if a new element type contains mandatory attributes, since the wizard will automatically create the attribute for the corresponding element type.

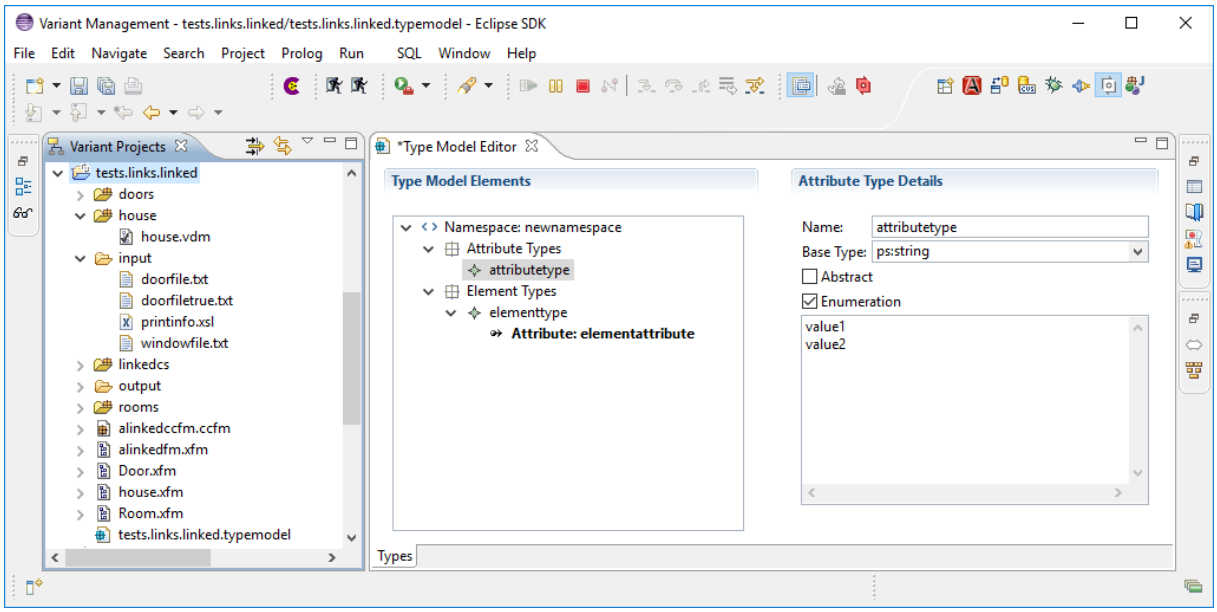
Figure 6.49. Type Model Editor Example



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 6.50. Type Model Editor Example



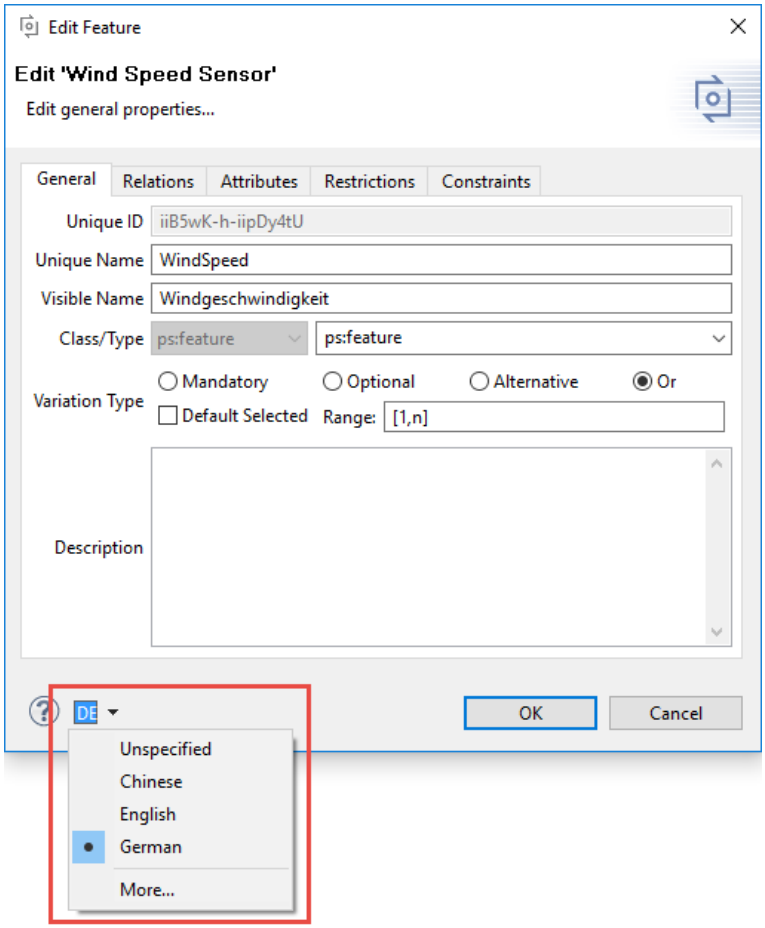
6.12. Using Multiple Languages in Models

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 inFigure 6.51, “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 chose the language in the upcoming dialog. The selected language will be added to the language list.

Figure 6.51. 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 (seeSection 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.

6.13. Importing and Exporting Models

6.13.1. 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.).

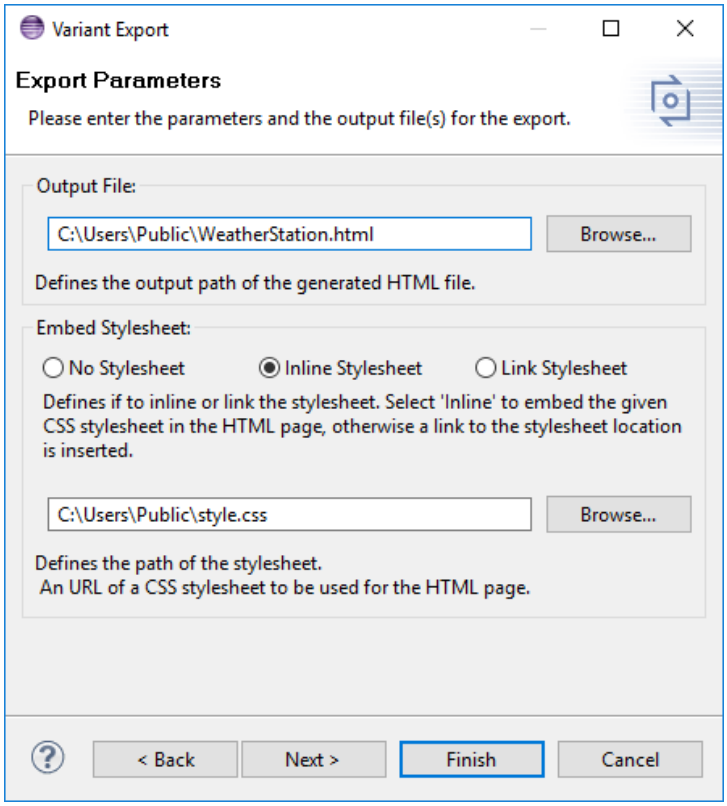
The export will generate a navigation section which represents all model elements hierarchical in a tree and the data of the elements on the right side of the generated html page. The navigation tree will help to navigate to elements quickly. The selected element in the navigation section will be shown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

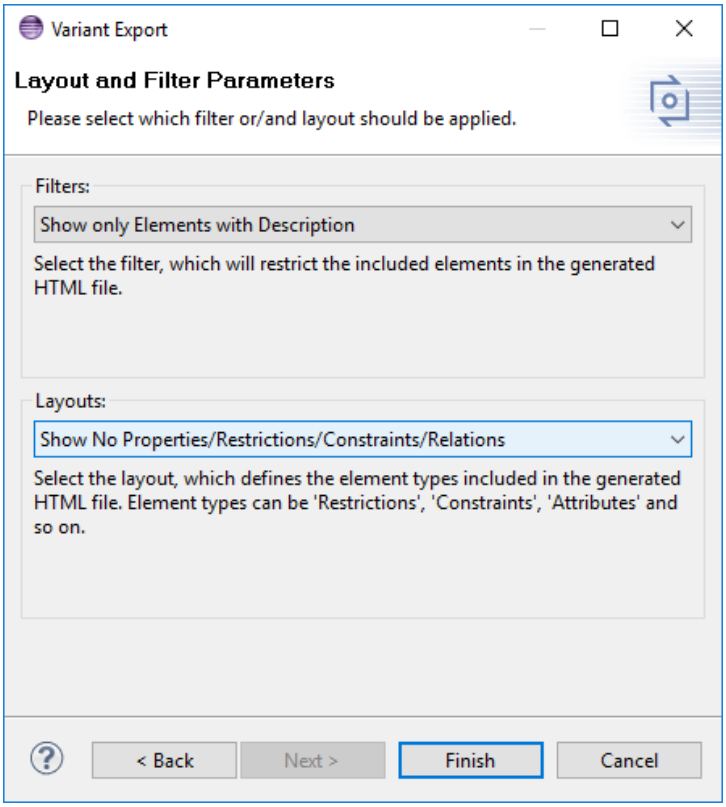
Figure 6.52. HTML Export Wizard



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 6.9, “ 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 seethe section called “ Tree Editing Page ” .

Figure 6.53. 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

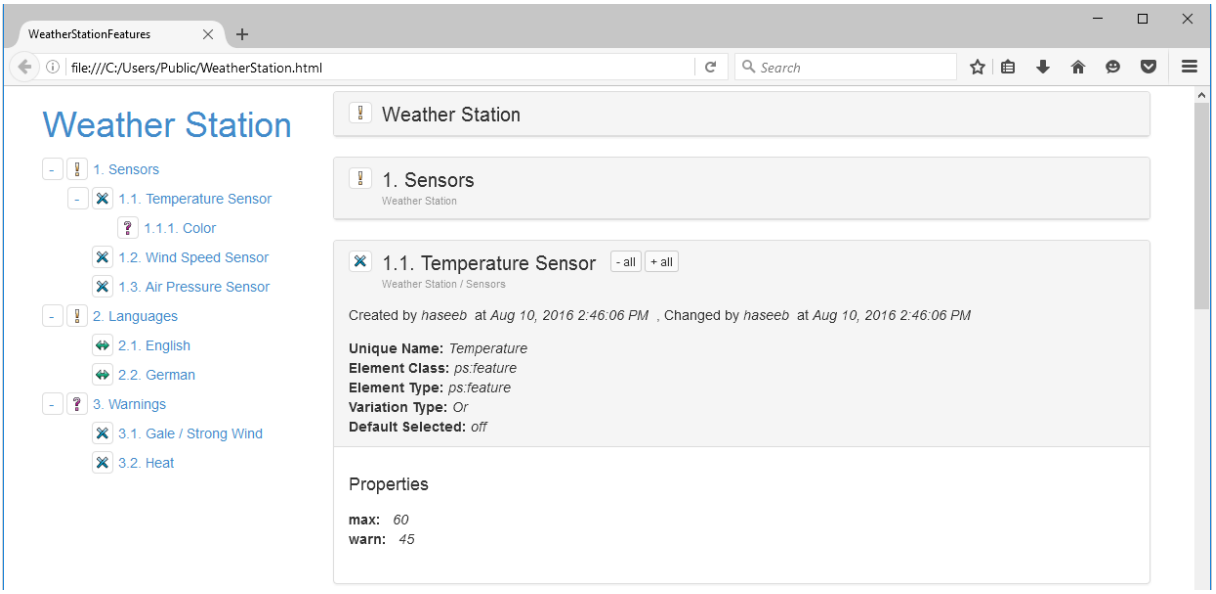
Is the html output opened in a browser the following interactions are available:

- 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 6.54. HTML Export Result



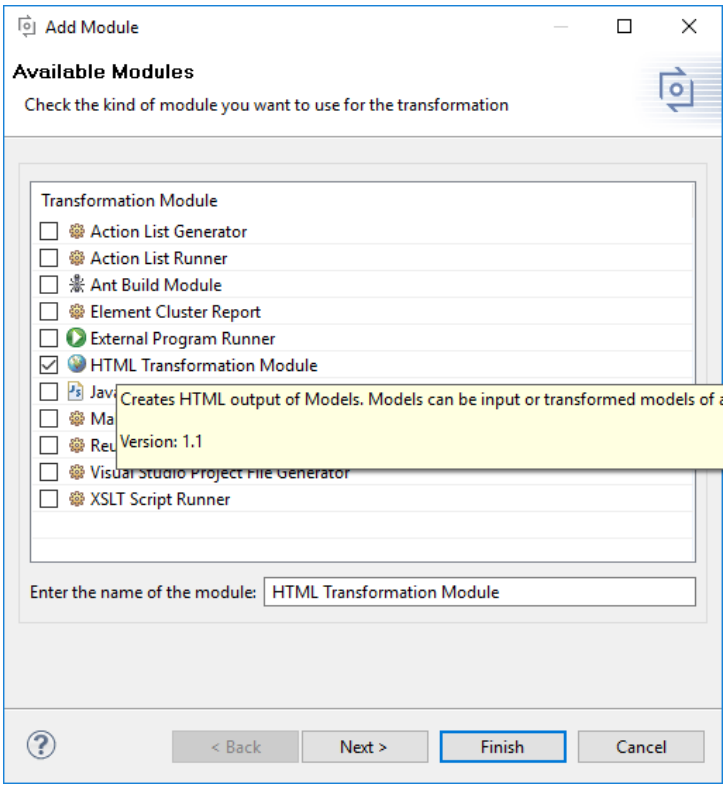
HTML Export example.

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 6.55. 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.

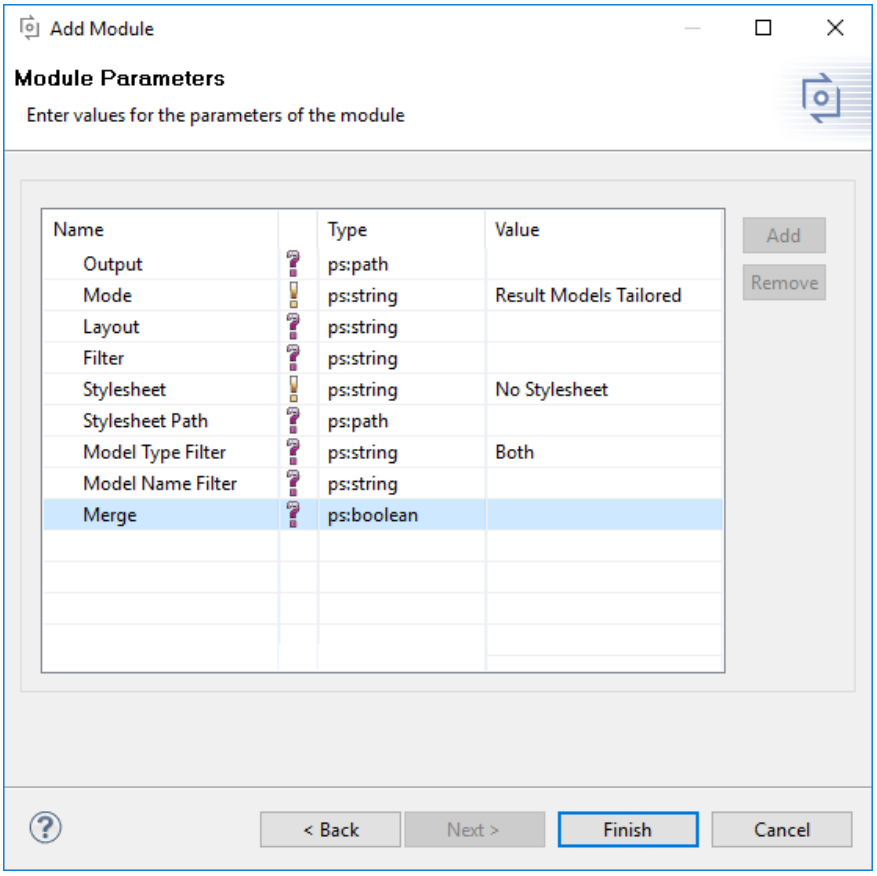
Third parameter *Layout* is optional. If used it defines a tree layout, which will be used during the transformation. (the section called “ Tree Editing Page ”)

Fourth parameter *Stylesheet* defines whether **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 regarding their type. Additionally the parameter *Model Name Filter* allows the user to specify a regular expression, which is used to filter the models by their names.

Figure 6.56. HTML Transformation Module Parameters



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 inFigure 6.57, “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 6.58, “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 6.57. Directed graph export example

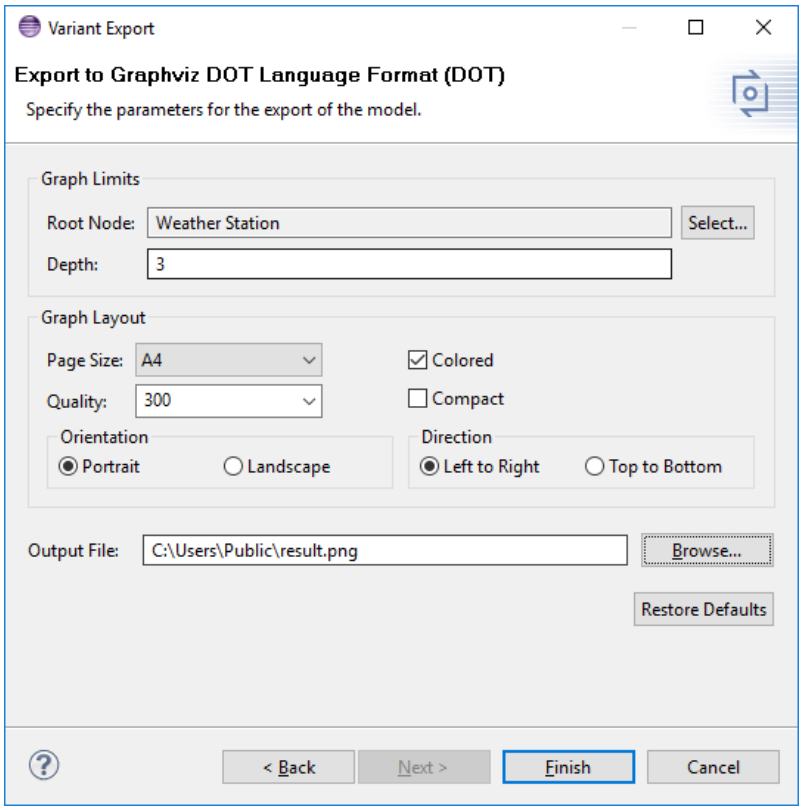
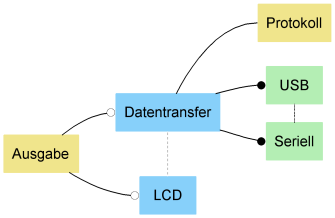


Figure 6.58. Directed graph export example (options LR direction, Colored)



6.13.2. 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.

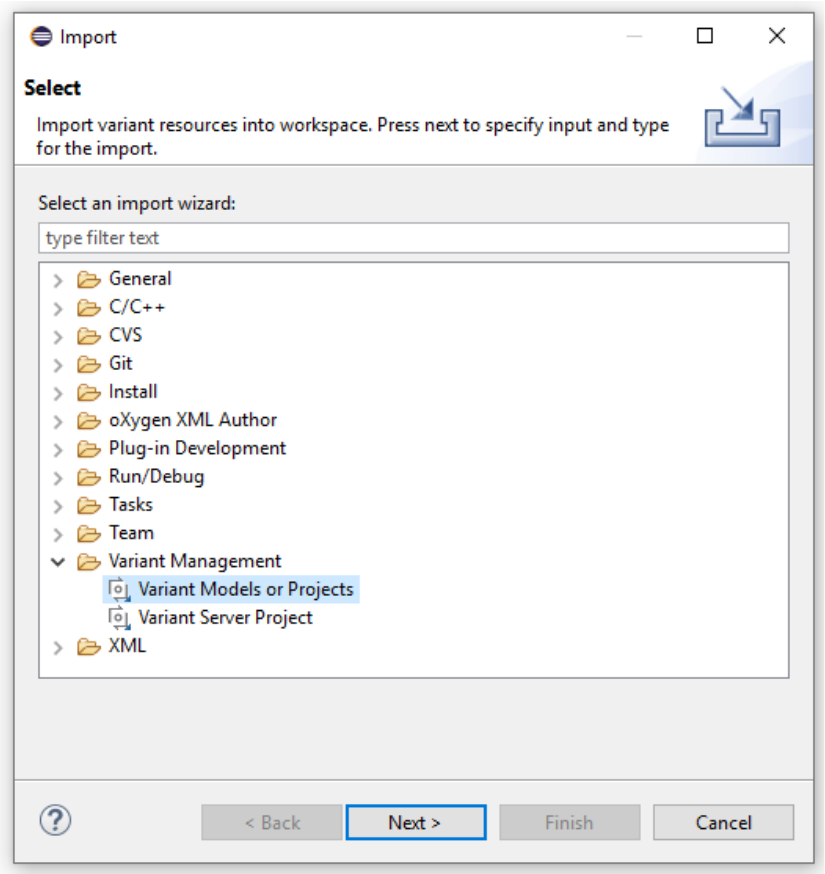
Currently there exists following generic imports which are discussed below:

- Import 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.
- Import a Feature Model from a CSV file.
- Import a Feature Model from an Excel file.

To learn more about how to import pure::variants server projets, see the documentation pure::variants Server Support Plug-In Manual. The Following steps explain how to import a Feature or a Family model from a CSV file accordingly:

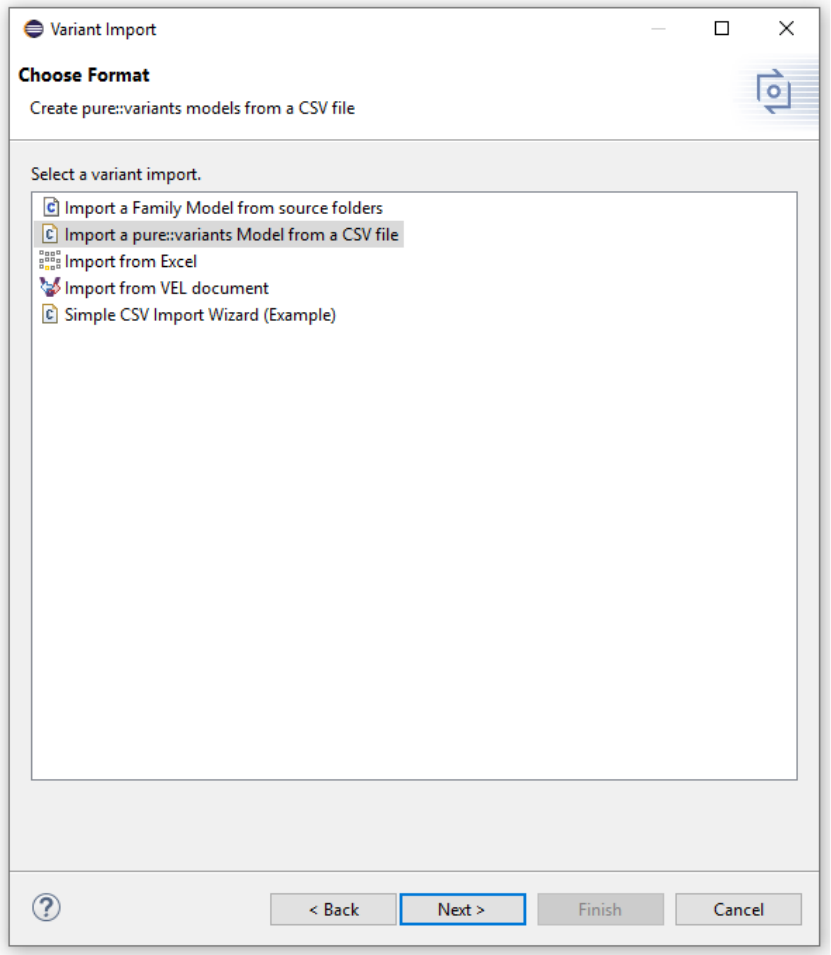
- Make sure you change the perspective to "Variant Management" or " Variant Projects view" respectively.
- Import item is provided in the Navigator and in the context menu and in the file menu
- To Open the Import Wizard dialog, right click on the file and select import from the menu option. Select "Variant Models or Projects" as shown in the below dialog. Click "next" to continue.

Figure 6.59. Import Dialog



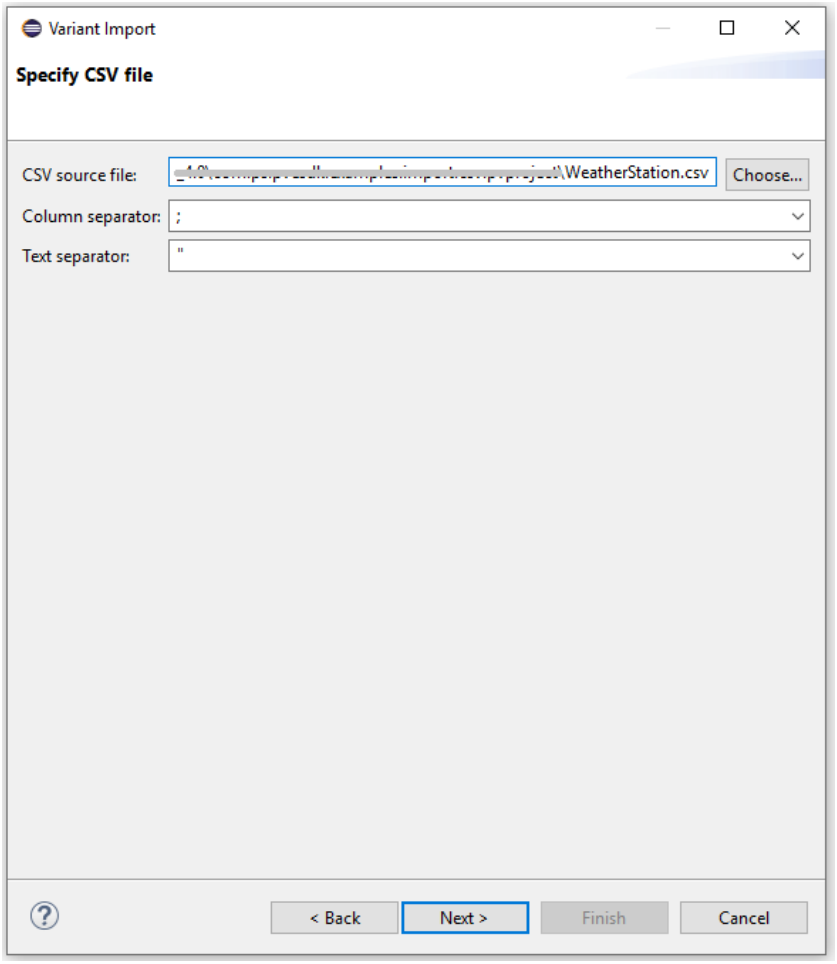
- Choose "Import a pure::variants Model from a CSV file" and click "next" to continue.

Figure 6.60. Select Variant Import Format



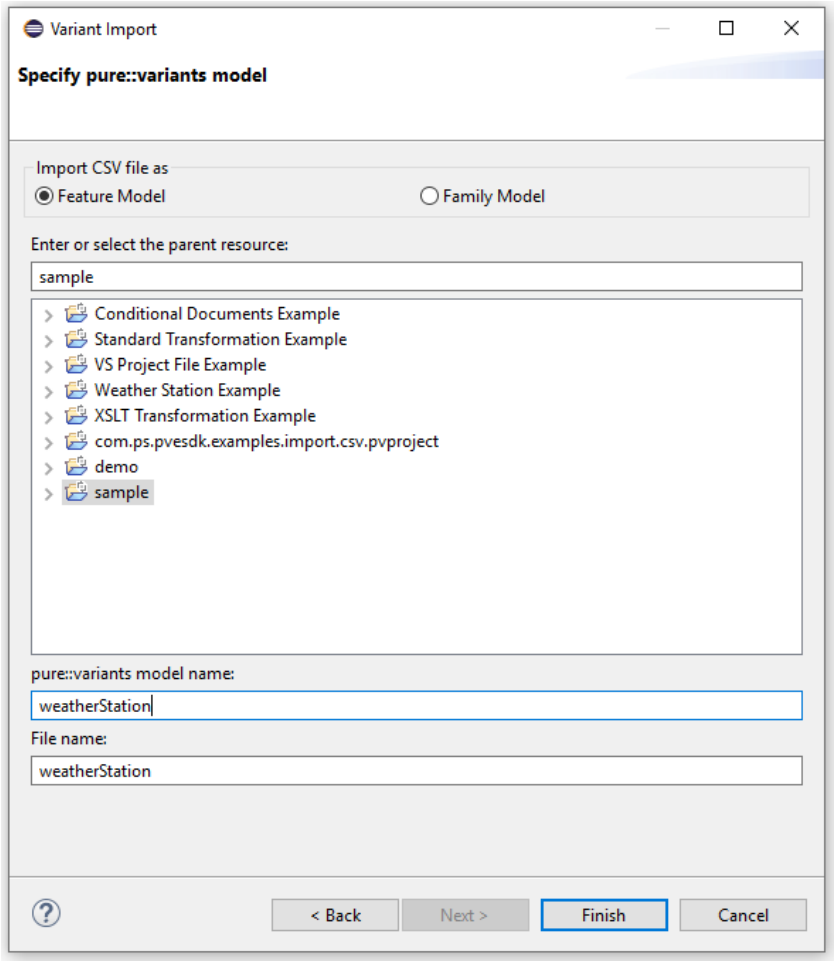
- Select the source file from your local directory and press "next"

Figure 6.61. Specify Source file



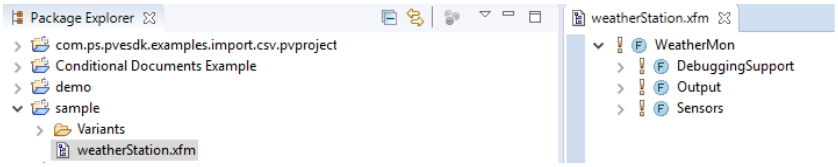
- Specify the pure::variants model, enter the model name of your choice and press "Finish"

Figure 6.62. Specify pure::variants model



- The import is completed successfully and you can now see the imported model as shown in the below figure

Figure 6.63. Imported Feature Model



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. Import Fields

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.

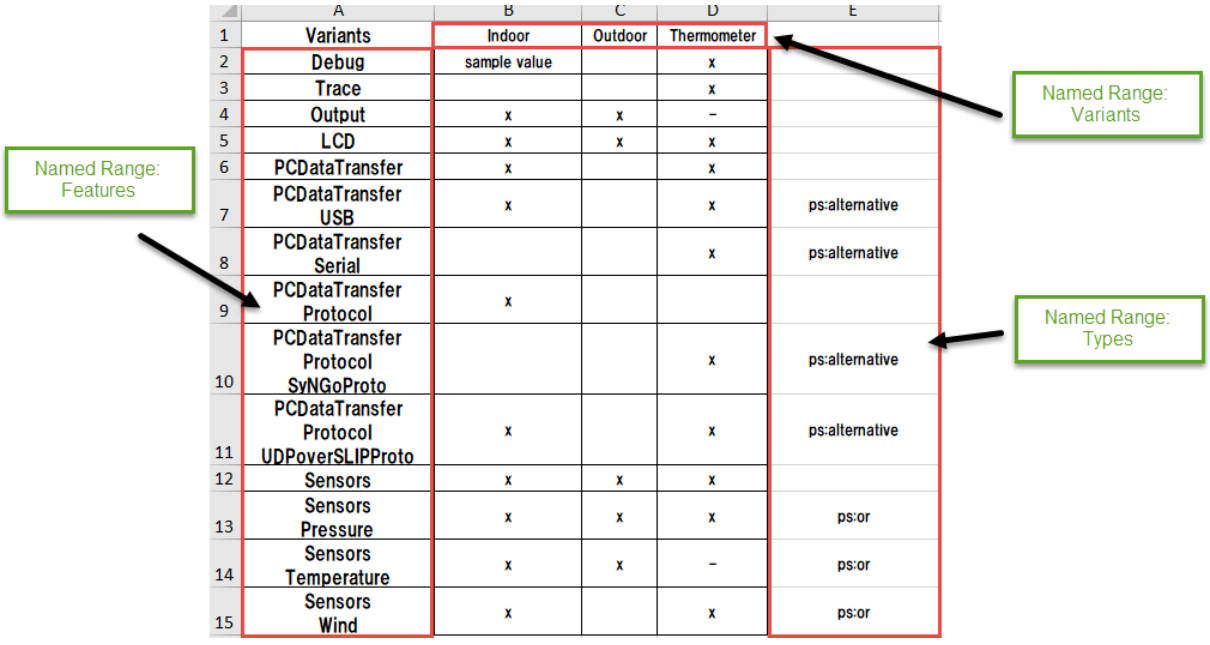
For importing a CSV to a Feature Model the field **Unique Name** is necessary. If you like to import a hierarchical model either the fields**Unique ID** and **Parent Unique ID** or **Unique Name** and **Parent Unique Name** are necessary as well. In case of importing an hierarchical model the element 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 round trip with the help of the CSV data format is not possible.

The third generic import, imports a Feature Model from an Excel file. While importing a few fields are directly used by pure::variants to build the model.

The Excel file needs a specific structure so pure::variants can interpret the information and generate models automatically from an Excel file. See below example shows this structure.

Figure 6.64. Excel File Structure



	A	B	C	D	E
1	Variants	Indoor	Outdoor	Thermometer	
2	Debug	sample value		x	
3	Trace			x	
4	Output	x	x	-	
5	LCD	x	x	x	
6	PCDataTransfer	x		x	
7	PCDataTransfer USB	x		x	ps:alternative
8	PCDataTransfer Serial			x	ps:alternative
9	PCDataTransfer Protocol	x			
10	PCDataTransfer Protocol SyNGoProto			x	ps:alternative
11	PCDataTransfer Protocol UDPoverSLIPProto	x		x	ps:alternative
12	Sensors	x	x	x	
13	Sensors Pressure	x	x	x	ps:or
14	Sensors Temperature	x	x	-	ps:or
15	Sensors Wind	x		x	ps:or

All cells named with name "Features" are used as unique names for features during the import. Names, which contains newlines become hirarchical. Meaning first name is the parent name and names after the newline are becoming childs of the first feature.

Cells names with "Variants" are considered to define variants during the import. In this example three variants will be created: Indoor, Outdoor and Thermometer. Those variants will be created in a configuration space named "Variants". The Selections are created based on regular expressions, which can configured. By Default "X" and "x" are considered as selection and "-" is considered to be an Exclusion. All other values become values of an attribute called "value". Empty cells are considered as unselection of the corresponding element.

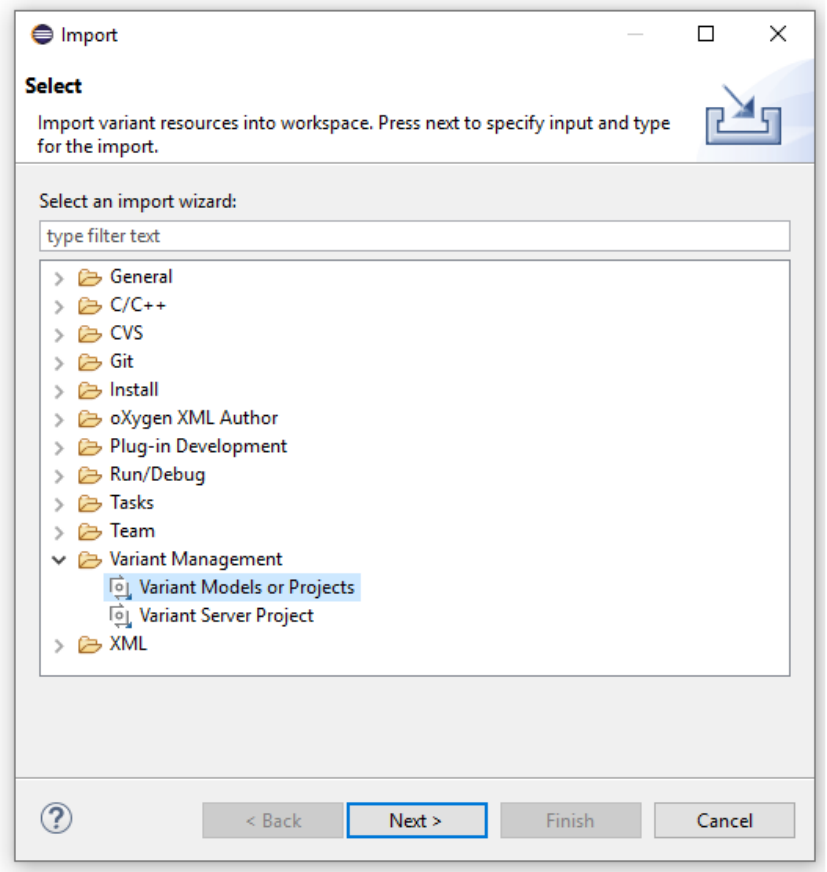
The cells of the "Types" named range define the variation types of the cerated features.

This table can be alligned vertical or horizontal and all names for the named ranges and regular expressions for the selections are configurable in the import wizard.

The Following steps explain how to import a feature model from an example excel file accordingly:

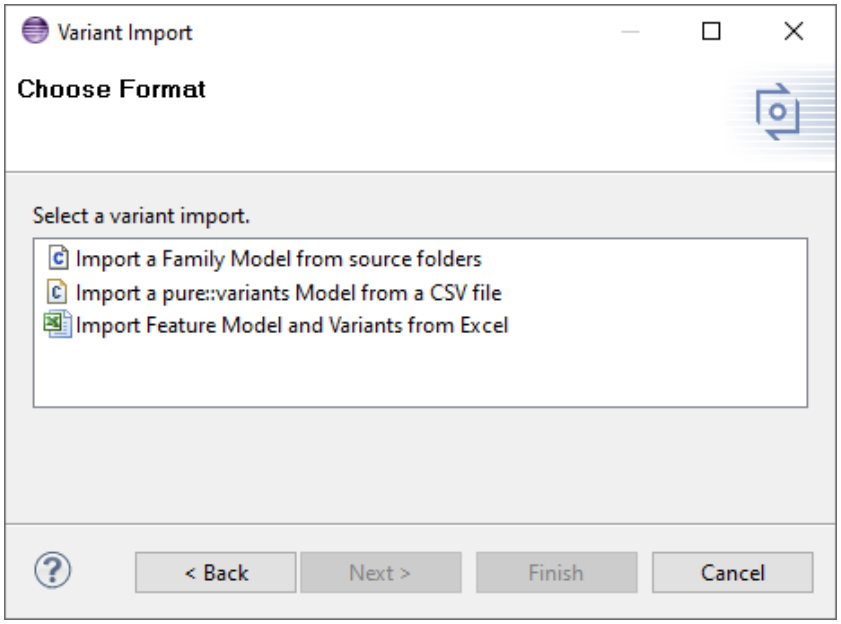
- Make sure you change the perspective to "Variant Management" or " Variant Projects view" respectively.
- Import item is provided in the Navigator and in the context menu and in the file menu
- To Open the Import Wizard dialog, right click on the file and select import from the menu option. Select "Variant Models or Projects" as shown in the below dialog. Click "Next" to continue.

Figure 6.65. Import Dialog



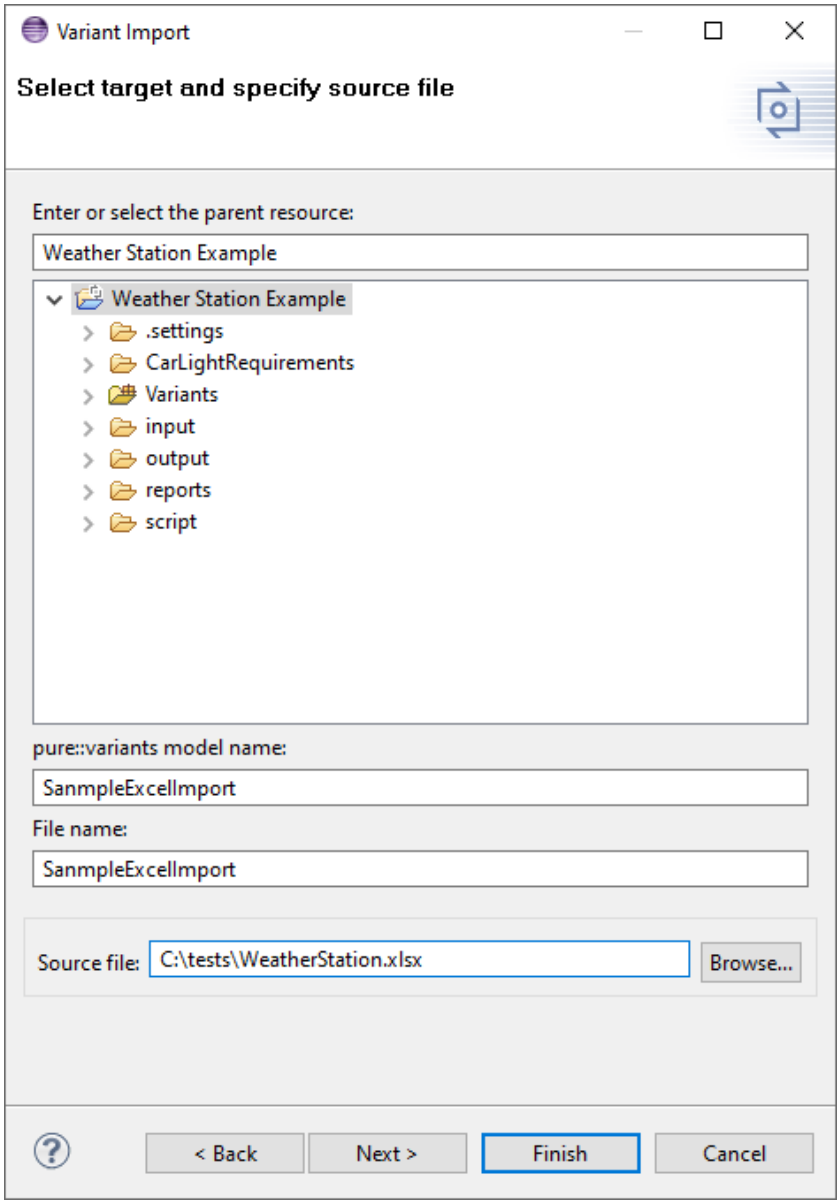
- Choose "Import Feature Model and Variants from Excel" and click "next" to continue.

Figure 6.66. Select Variant Import Format



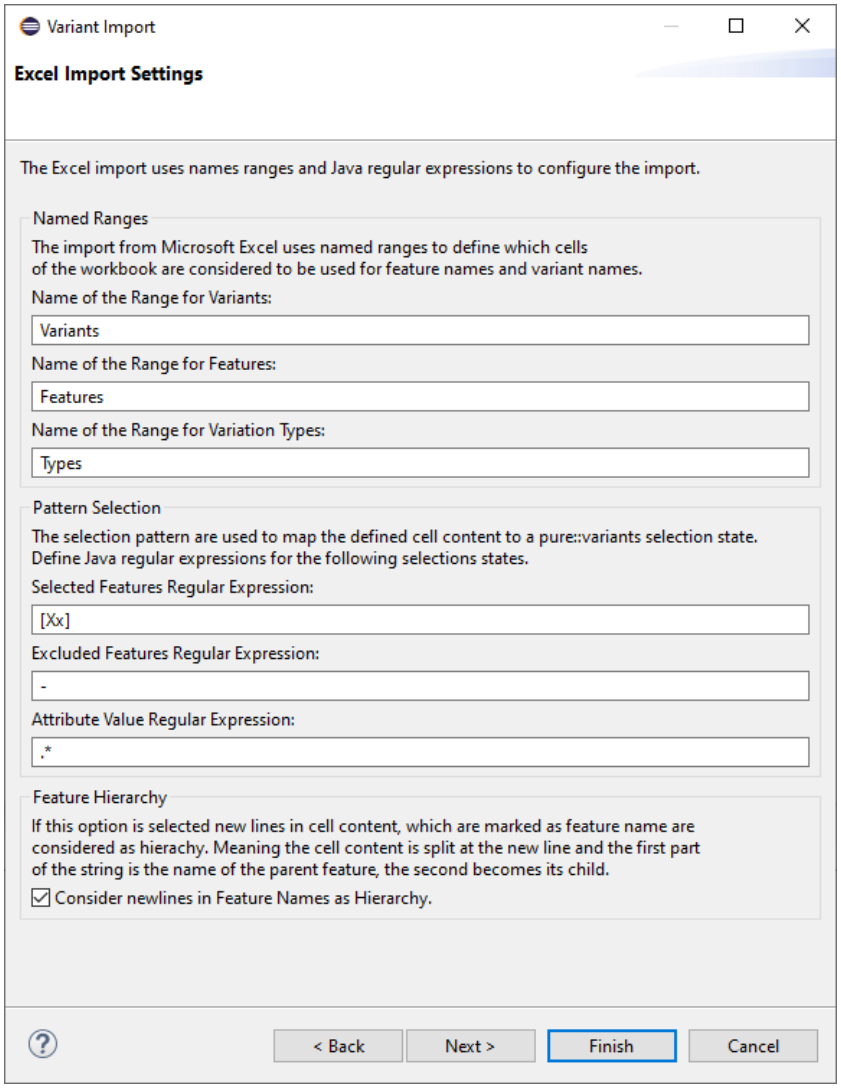
- Select a target container and fill in the "pure::variants model name" and the "File name" as follows. Also, select the source from your local directory and press "next"

Figure 6.67. Select Target and Specify Source file



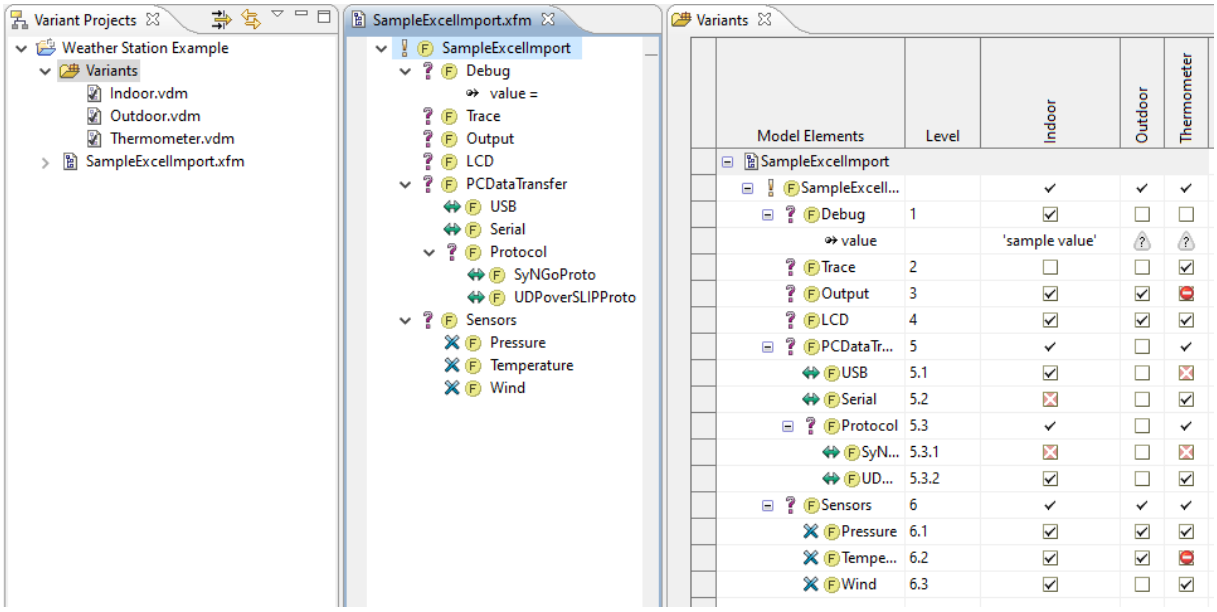
- The default expressions for the selected, excluded and the value patterns are as follows. The default named ranges are also set. Here, changes can be done as per the requirement and press "Finish"

Figure 6.68. Select Pattern for feature Selection



- The import is completed successfully and you can now see the imported feature model as shown in the below figure. If variants are defined additionally those models will created in a configuration space called "Variants". The picture below shows the import result of the sample Excel file of picture Figure 6.64, “Excel File Structure” .

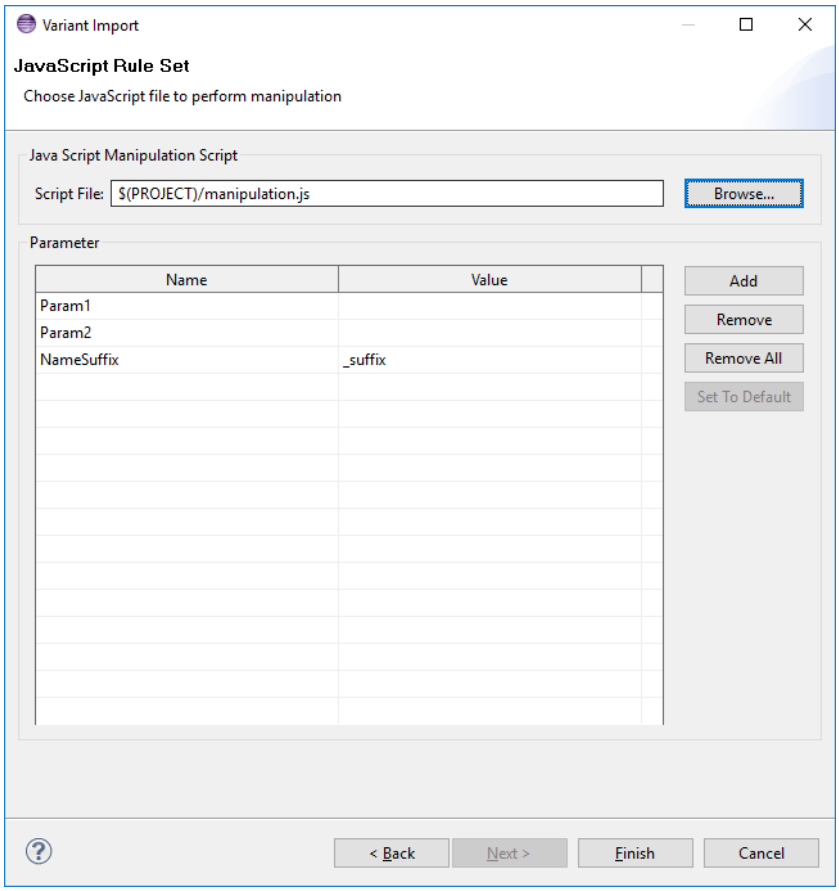
Figure 6.69. Imported Feature Model



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 6.70. 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.

6.14. External Build Support (Ant Tasks)

Eclipse comes with an integrated Ant support. This can easily be used to automate build actions. To integrate variant management actions into these build processes, pure::variants provides a number of Ant tasks. They can be used with build files inside Eclipse or in headless mode.

A simple Ant script to trigger a pure::variants transformation looks like this:

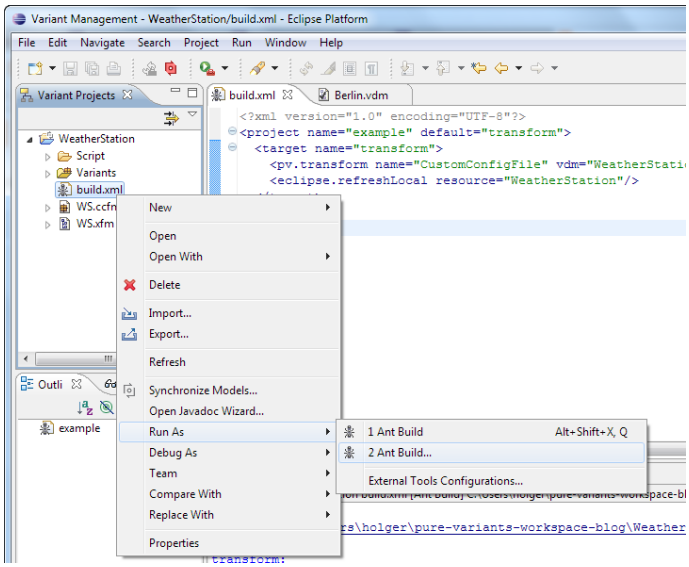
```
<?xml version="1.0" encoding="UTF-8"?>
<project name="example" default="transform">
  <target name="transform">
    <pv.import path="C:\Projects\WeatherStation"/>
    <pv.transform name="CustomConfigFile" vdm="WeatherStation/Variants/Berlin.vdm"/>
    <eclipse.refreshLocal resource="WeatherStation"/>
  </target>
</project>
```

This script runs the transformation *CustomConfigFile* on the variant description model *Berlin.vdm* in project *WeatherStation* . The transformation will generate some output in the project's directory.

First the *pv.import* task is used to import the project into the Eclipse workspace if it doesn't exist. Then the *pv.transform* task is used to start the *CustomConfigFile* transformation. And to let Eclipse reload and show the transformation results in the project directory, the Eclipse Ant task *eclipse.refreshLocal* is executed as the last build step.

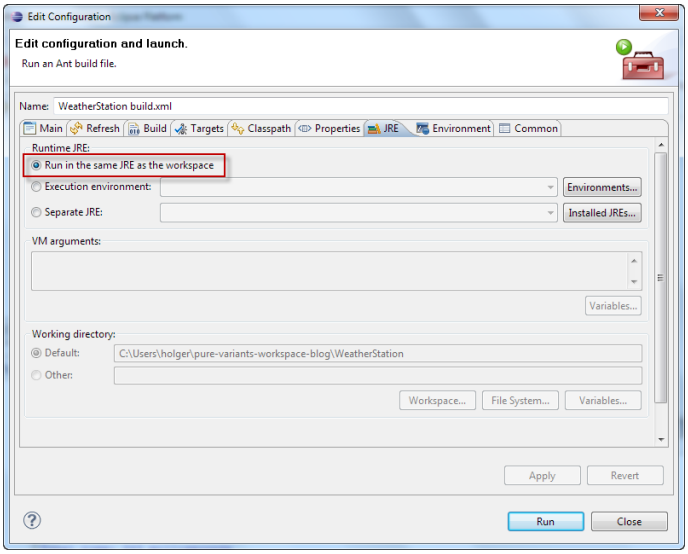
To run this Ant script, create a file *build.xml* with the above content in the project directory. Then right-click file *build.xml* and choose **Run As** -> **Ant Build…** from the context menu.

Figure 6.71. Ant Build Action



To let Ant find the pure::variants provided Ant tasks, the correct JRE needs to be selected. Switch to the JRE tab and select option **Run in the same JRE as the workspace** .

Figure 6.72. Ant Build JRE Parameter



Click **Run** to start the script execution.

The **build.xml** script can also be executed from outside of Eclipse (so-called headless mode). There are several ways to do this. Please note for Linux based operating systems the *X Window System* needs to be installed and started. Eclipse will not start if it is not able initialize GTK, which needs the X Window System to be installed and started.

You can use the Eclipse console application to run the script as follows:

```
%ECLIPSE%/eclipsec -nosplash --launcher.suppressErrors
  -application org.eclipse.ant.core.antRunner
  -data C:\workspace -buildfile build.xml -DPVLICENSE=C:\pv.de.lic
```

This command directly starts the Ant script runner of Eclipse with the Ant script **build.xml** , the path to an existing or temporary Eclipse workspace, and the definition of variable `PVLICENSE` pointing to a valid pure::variants license as arguments.

To simplify this, pure::variants comes with two batch scripts located in the **cli** sub-directory of the pure::variants installation directory.

- **runant.bat** starts the given Ant build file with the internal Eclipse Ant runner

```
runant.bat build.xml
```

- **variantscli.bat** starts the given transformation configuration on all provided VDMs

```
cd WeatherStation
variantscli.bat CustomConfigFile Variants/Berlin.vdm Variants/Paris.vdm Variants/London.vdm
```

Both scripts support the following environment variables to configure the execution.

Table 6.5. Environment Variables

Variable	Description
PVHOME	Path to the `eclipse` sub-directory in the pure::variants installation directory Example: `C:\Program Files\Parametric Technology\pv_Enterprise_6.0\eclipse`

Variable	Description
PVLIC	Path to the pure::variants license file Example: `C:\pv.de.lic`
PVJAVA	Path to Java executable Example: `C:\Java\bin\java.exe`
PVCONFIG	Name of the Eclipse configuration to use Example: `AntRun`

The `PVHOME` variable is automatically added to the **runant.bat** and **varianstcli.bat** if pure::variants has been installed using the pure::variants installer executable.

The **runant.bat** (Windows) and **runant.sh** (on Linux and Mac) scripts have the following command line parameters which must be given in the order they are listed in the following table. Optional parameters can be omitted.

Table 6.6. runant Command Line Parameters

Parameter	Description
-l	Optional parameter to enable printing the pure::variants and Eclipse logs on exit. Example: `runant.bat -l build.xml`
-t *target*	Optional parameter to run the given target of the Ant script instead of the default target. Example: `runant.bat -t "Transform and Refresh" build.xml`
-w *workspace*	Optional parameter to specify the path to an existing Eclipse workspace in which to run the Ant script. If not given, a temporary workspace directory is created, and deleted on exit. Example: `runant.bat -w C:\workspace build.xml`
antfile	The path to the Ant script to run. Example: `runant.bat "C:\Ant Scripts\script5.xml"`
vmargs	Every argument following the path to the Ant script is passed as command line option to the Java VM. Please refer to the official Java documentation for the complete list of Java command line options. Example: `runant.bat "C:\Ant Scripts\count.xml" -DFrom=1 -DTo=100` This command line runs the `print_report.xml` script with two variables `From` and `To` passed to the Java VM using option `-D` . The Ant script then could access these variables using expressions`${From}` and `${To}` . Java transformation modules and JavaScript scripts run by the Ant script could access these variables using`PVProperty.getPVProperty("From")` and `PVProperty.getPVProperty("To")` .

The **variantscli.bat** (Windows) and **variantscli.sh** (on Linux and Mac) scripts have the following command line parameters which must be given in the order they are listed in the following table. Optional parameters can be omitted.

Table 6.7. variantscli Command Line Parameters

Parameter	Description
-l	Optional parameter to enable printing the Eclipse log on exit. Example: `variantscli.bat -l Report C:\WS\Project\Variants\V5.vdm`
transformation	The name of the transformation to execute (see the section called “Transformation Configuration Page”).

Parameter	Description
	Example: `variantscli.bat Report C:\WS\Project\Variants\V5.vdm`
VDM VDM VDM ...	Every argument following the transformation name must be the path to a VDM to transform. At least one VDM must be specified. Example: `variantscli.bat Report C:\WS\Project\Variants\V5.vdm C:\WS\Project\Variants\V6.vdm C:\WS\Project\Variants\V9.vdm`

6.14.1. 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" importreferences="false" />
<pv.import server="http://pv.server.com" name="Weather Station" revision="v2"/>
<pv.import url="pvrm://pv.server.com/projects/iqRjtaATGwbmd2tGi#v2"/>
```

This task has the following attributes:

- **importreferences** if `true` the references to other projects are also imported (default is`true`) If the referenced project was stored with a revision, the referenced project is imported in that revision.
- **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
- **url** is the url of a remote project to import

6.14.2. 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"/>
  <property name="severity" value="error"/>
  <modelset>
    <include path="Weather Station\config\*.vdm" nature="com.ps.consul.nature">
      <property name="modelHeadProperty" value="value"/>
      <jsfilter script="Weather Station\antJsFilter.js" />
    </include>
    <exclude path="Weather Station\*\NotSelected.vdm" nature="com.ps.consul.nature">
      <property name="modelHeadProperty" value="value"/>
    </exclude>
  </modelset>
</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
- **continueOnError** If this property is set to *true* the task does not throw BuildExceptions, but writes problems to standard out and finishes successfully.

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

- **autoresolve** sets the mode of the auto resolver. Possible values are `off`, `simple`, `extended`
- **timeout** sets the maximal time used for the evaluation in seconds
- **severity** sets the minimal severity for problems to output. Possible values are `info`, `warning`, `error`. The default value is `info`.

Instead of using the vdm attribute for defining one variant model model for the evaluation the *modelset* can be used. It allows to define multiple variant models, which will be run in the context of the same evaluation task. This simplifies the definition of multiple models with the same evaluation settings.

For defining the set of relevant models several possibilities exist. A model is part of the model set, if it matches at least one include rule, but no exclude rule. If no include rule is defined the model must not match an exclude rule. If neither an include nor an exclude rule is defined all models of a workspace are part of the model set. If multiple options within the same include/exclude rule are used all of the defined options have to match.

- **path** is the workspace relative path to the model. The path attribute allows to use wildcards, * for any character in the same folder and ** for any character including the file separator.

  E.g. Weather Station\config*.vdm matches all Variant Description Models directly located in the config folder inside the "Weather Station" project.

  Weather Station**.vdm matches all Variant Description Models in the Weather Station project, even if they are located in subfolders.

- **nature** pure::variants uses natures to identify models imported by a specific importer from a specific external source. This nature can be used to filter the models.

  E.g. com.ps.consul.eclipse.ui.doorsng.syncronizable.nature is the nature for the IBM Doors Next Generation importer.

- **Defining model head properties** Any model property can be used to filter models. The value property is optional. If the value is not given the existance of a property with the defined name is sufficient, that a model matches the rule. If the value is given the value also has to match.

- **jsfilter** enables model filtering based on more complex criteria. The mandatory attribute **script** defines the location of the JavaScript filter script either as an absolute path or relative to the workspace. The filter script has to fulfill a particular interface. To create a template script with this interface, invoke pure::variants' project context menu and click New -> pure::variants JavaScript Script. In the wizard dialog that opens, choose *ANT ModelSet Filter* as the execution context.

6.14.3. 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"
   input="C:/somewhere/input" output="C:/somewhere/output">
  <property name="autoresolve" value="extended"/>
  <property name="timeout" value="120"/>
  <property name="severity" value="error"/>
  <inputmodelset>
    <include path="Weather Station\*.ccfm" nature="com.ps.consul.nature">
      <property name="modelHeadProperty" value="value"/>
      <jsfilter script="Weather Station\antJsFilter.js" />
    </include>
    <exclude path="Weather Station\NotSelected.ccfm" nature="com.ps.consul.nature">
      <property name="modelHeadProperty" value="value"/>
    </exclude>
  </inputmodelset>
  <modelset>
    <include path="Weather Station\config\*.vdm" nature="com.ps.consul.nature">
      <property name="modelHeadProperty" value="value"/>
```

```
    </include>
    <exclude path="Weather Station\*\NotSelected.vdm" nature="com.ps.consul.nature">
      <property name="modelHeadProperty" value="value"/>
    </exclude>
  </modelset>
</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 (default is `Default`)
- **force** if `true` the transformation runs always also if the result has errors (default is `false`)
- **continueOnError** If this property is set to *true* the task does not throw BuildExceptions, but writes problems to standard out and finishes successfully. (default is `false`)
- **input** is the input path the transformation uses. It overwrites the input path defined in the transformation configuration.
- **output** is the output path the transformation uses. It overwrites the output path defined in the transformation configuration.

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

- **autoresolve** sets the mode of the auto resolver. Possible values are `off`, `simple`, `extended` (default is `extended`)
- **timeout** sets the maximal time used for the evaluation in seconds (default is `120`)
- **severity** sets the minimal severity for problems to output. Possible values are `info`, `warning`, `error`. The default value is `info`. (default is `info`)

The `pv.transform` task supports filtering of the input model set, meaning it is possible to reduce the number of input models after evaluation for a specific Transformation Configuration. The definition of input model filtering is applied on top of the input model set defined in the used Transformation Configuration. There is no possibility to add new models into the transformation. It is not allowed to filter feature models from the input model set. Rules to filter feature models are simply ignored. To define input model filtering the *inputmodelset* tag is used. It allows to define multiple *include* and *exclude* rules.

An input model is part of the input model set, if it matches at least one include rule, but no exclude rule. If no include rule is defined the model must not match an exclude rule. If neither an include nor an exclude rule is defined all input models are part of the input model set.

For filtering the input models several possibilities exist. The options can be combined. If multiple options within the same include/exclude rule are used all of the defined options have to match.

- **path** is the workspace relative path to the input model. The path attribute allows to use wildcards, * for any character in the same folder and ** for any character including the file separator.

  E.g. Weather Station*.ccfm matches all family models directly located in the Weather Station project folder.

  Weather Station**.ccfm matches all family models in the Weather Station project, even if they are located in sub folders.

- **nature** pure::variants uses natures to identify models imported by a specific importer from a specific external source. This nature can be used to filter the input models.

  E.g. com.ps.consul.eclipse.ui.doorsng.syncronizable.nature is the nature for the IBM Doors Next Generation importer.

- **Defining model head properties** Any model property can be used to filter models. The value property is optional. If the value is not given the existance of a property with the defined name is sufficient, that a model matches the rule. If the value is given the value also has to match.

- **jsfilter** enables model filtering based on more complex criteria. The mandatory attribute **script** defines the location of the JavaScript filter script either as an absolute path or relative to the workspace. The filter script has to fulfill a particular interface. To create a template script with this interface, invoke pure::variants' project context menu and click New -> pure::variants JavaScript Script. In the wizard dialog that opens, choose *ANT ModelSet Filter* as the execution context.

Instead of using the vdm attribute for defining one Variant Description Model for the transformation the *modelset* can be used. It allows to define multiple variants description models, which will be run in the context of the same transforamtion task. This simplifies the definition of multiple variants from the same configuration space with the same transformation settings. The model set definition has the same options like the input model set definition.

6.14.4. pv.validate

The `pv.validate` task runs all available element and model checks on the given model.

Example:

```
<pv.validate model="Weather Station\WS.xfm">
  <property name="severity" value="warning"/>
  <modelset>
    <include path="Weather Station\*.ccfm" nature="com.ps.consul.nature">
      <property name="modelHeadProperty" value="value"/>
      <jsfilter script="Weather Station\antJsFilter.js" />
    </include>
    <exclude path="Weather Station\NotSelected.ccfm" nature="com.ps.consul.nature">
      <property name="modelHeadProperty" value="value"/>
    </exclude>
  </modelset>
</pv.validate>
```

This task has the following attributes:

- **model** is the model to validate

The `pv.validate` task supports optional properties which influence the output of the validation:

- **severity** set the minimal severity for problems to output. Possible values are `info`, `warning`, `error`. (default is `info`)

Instead of using the model attribute for defining one model model for the evaluation the *modelset* can be used. It allows to define multiple models, which will be run in the context of the same evaluation task. This simplifies the definition of multiple models with the same evaluation settings.

For defining the set of relevant models several possibilities exist. A model is part of the model set, if it matches at least one include rule, but no exclude rule. If no include rule is defined the model must not match an exclude rule. If neither an include nor an exclude rule is defined all models of a workspace are part of the model set. If multiple options within the same include/exclude rule are used all of the defined options have to match.

- **path** is the workspace relative path to the model. The path attribute allows to use wildcards, * for any character in the same folder and ** for any character including the file separator.

  E.g. Weather Station\config*.ccfm matches all family models directly located in the config folder inside the "Weather Station" project.

  Weather Station**.vdm matches all Variant Description Models in the "Weather Station" project, even if they are located in sub folders.

- **nature** pure::variants uses natures to identify models imported by a specific importer from a specific external source. This nature can be used to filter the models.

E.g. com.ps.consul.eclipse.ui.doorsng.syncronizable.nature is the nature for the IBM Doors Next Generation importer.

- **Defining model head properties** Any model property can be used to filter models. The value property is optional. If the value is not given the existance of a property with the defined name is sufficient, that a model matches the rule. If the value is given the value also has to match.

- **jsfilter** enables model filtering based on more complex criteria. The mandatory attribute **script** defines the location of the JavaScript filter script either as an absolute path or relative to the workspace. The filter script has to fulfill a particular interface. To create a template script with this interface, invoke pure::variants' project context menu and click New -> pure::variants JavaScript Script. In the wizard dialog that opens, choose *ANT ModelSet Filter* as the execution context.

6.14.5. 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)

6.14.6. 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

6.14.7. pv.sync

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

Example:

```
<pv.sync model="Weather Station\Sources.ccfm">
  <modelset>
    <include path="Weather Station\*.ccfm" nature="com.ps.consul.nature">
      <property name="modelHeadProperty" value="value"/>
      <jsfilter script="Weather Station\antJsFilter.js" />
    </include>
    <exclude path="Weather Station\NotSelected.ccfm" nature="com.ps.consul.nature">
      <property name="modelHeadProperty" value="value"/>
    </exclude>
  </modelset>
</pv.sync>
```

This task has the following attributes:

- **model** is the model to update

Instead of using the model attribute for defining one model for the synchronization the *modelset* can be used. It allows to define multiple pure::variants models, which will be run in the context of the same synchronization task. This simplifies the definition of multiple models from the same configuration space with the same transformation settings.

6.14.8. pv.syntaxsemanitccheck

The `pv.syntaxsemanitccheck` task checks the given configspace for semantic and syntactic errors. This is running the same checks as the **Perform Syntax and Semantic Check** action on configuration spaces.

Example:

```
<pv.syntaxsemanitccheck configspace="Weather Station\Variants" reportfile="Weather Station
\CheckResult.html" />
```

This task has the following attributes:

- **configspace** is the configuration space to be checked.
- **reportfile** the location the resulting report is stored.

6.14.9. 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)

6.14.10. pv.javascript

The `pv.javascript` task performs a given javascript in a specific context. This allows the user to automate existing javascripts. The script can be performed in the context of one model or in the context of one project. If both a project and a model is given, the model is used for the context.

Example:

```
<pv.javascript script="C:\Temp\javascript.js" project="Weather Station"
 model="$(PROJECT)\Sources.ccfm" />
```

This task has the following attributes:

- **script** is the path to the performed javascript. This path has to be absolute or relative to the used ANT workspace.
- **model** is the path to the context model. This property is optional. Variant path variables can be used here.
- **project** is the path to the context project. This property is optional. Variant path variables can be used here.

6.14.11. pv.offline

The `pv.offline` task switches the server project into offline mode. The project is selected by the name attribute. This task does nothing if the project is already offline or if the project is a local project.

Example:

```
<pv.offline name="Weather Station"/>
```

This task has the following attributes:

- **name** is the name of the project

6.14.12. pv.online

The `pv.online` task switches the server project into online mode. The task performs a "Override and update" if there are differences between the remote project and the local representation. Meaning the local data is overwritten with the current state of the project on the pure::variants Server. The project is selected by the name attribute. This task does nothing if the project is already online or if the project is a local project.

Example:

```
<pv.online name="Weather Station"/>
```

This task has the following attributes:

- **name** is the name of the project

6.14.13. pv.userrolesync

The `pv.userrolesync` task is used to synchronize users and roles of a pure::variants model server with their data sources (e.g. LDAP directory servers).

Example:

```
<project>
  <property name="server" value="http://server:1234"/>
  <pv.connect server="${server}" user="admin" pass="123"/>
  <pv.userrolesync server="${server}" username="cn=reader,dc=company,dc=com" password="456">
    <role name="Modeler"/>
    <role name="User"/>
    <role name="Tester"/>
    <user name="*"/>
  </pv.userrolesync>
</project>
```

This task has the following attributes:

- **server** is the pure::variants model server
- **username** is the name of the data source user (e.g. an LDAP bind user)
- **password** is the password of the data source user

The users and roles to synchronize are listed using **user** and **role** elements. Both elements have the attribute **name** which specifies the name or a name pattern for the users or roles to synchronize. The name can contain the special characters "*" to match any text and "?" to match a single character.

If the data source of a user or role to synchronize is a server that uses a certificate which is not trusted by pure::variants, then the synchronization with that data source server will fail. To register this certificate with pure::variants, start pure::variants and open the User Management of the pure::variants server (see "pure::variants Server Administration Manual" about details on how to do this). Then try to synchronize the same users and roles from within pure::variants. You will be asked by pure::variants to accept the certificate of the data source server permanently. After you agreed, run the Ant task again. It will not fail anymore due to an untrusted certificate.

6.14.14. pv.property

The `pv.property` task is used to define a runtime property, which can be used in several pure::variants connectors. For example runtime properties are used for defining user credentials for external tools used by some connector transformations.

Example:

```
<project>
  <pv.property name="propertyName" value="property value" />
</project>
```

This task has the following attributes:

- **name** is name of the property to set
- **value** is the value of the property to set

6.14.15. pv.about

The `pv.about` task lists the pure::variants environment including the installed pure::variants features and information about the used Eclipse and Java version.

Example:

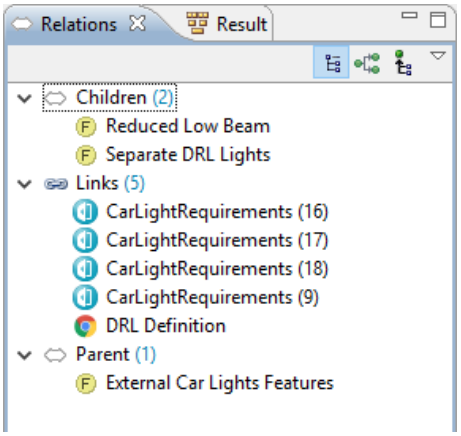
```
<pv.about/>
```

This task does not support attributes.

6.15. Linking between pure::variants and external resources

pure::variants model objects can be linked with any external resource in both directions. To create a link to an external resource inside a pure::variants object's HTML description, use the "Insert/Edit Link" ( ) action in the description editor to add the link location to your description. Alternatively you can also drag an URL and drop it directly into the editor or on the model element. All links from the description of the selected element are shown in the Relations View.

Figure 6.73. Relations View with external Links



A double click on a link results in navigating to the link's destination if a link hander is registered for the respective link type.

To support linking in the inverse direction, pure::variants model elements can also be accessed by URL links. To get a model element's URL use the "Copy URL" context menu action on that element. The URL is made available in the Clipboard and can be pasted into any other resource or application. If external applications are able to handle drop events, a simple drag of the model element with the mouse and dropping it on the external application will work too.

The pure::variants installer for Windows will setup a link handler, which allows direct navigation from external applications to the linked pure::variants model element. However, the handler only works if an instance of pure::variants is running and the linked model element is available in the currently used workspace.

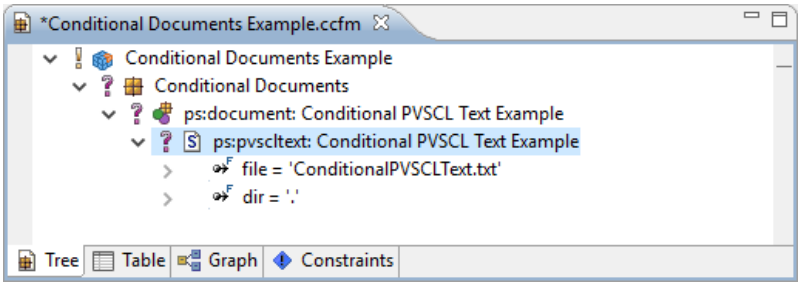
6.16. Manipulating Text Files

The pure::variants standard transformation can be used to manipulate text files based on pvSCL conditions and calculations. To achieve this, the transformation and family model needs to be set up for transforming a text file, and the file needs to be annotated with pvSCL conditions and calculations.

6.16.1. Setting Up the Transformation

For setting up the standard transformation, please refer to the section called “Setting up the Standard Transformation” . Now the text file to transform still needs to be referenced. This is done in the family model.Figure 6.74, “Family Model with ps:pvscltext transformation setup” shows an example family model referencing the input file "ConditionalPVSCLText.txt". To create the necessary family model elements, the easiest way is to use a wizard. To do this, add an element of class*ps:part* to the family model and select "New"->"PVSCL Conditional Text" from the part's context menu. Please refer toSection 9.5.5, “ ps:pvscltext ” , for details about supported attributes.

Figure 6.74. Family Model with ps:pvscltext transformation setup



6.16.2. Editing Conditions and Calculations in Text Files

Conditions and calculations are added to the text file as special statements, such as `PVSCL:IFCOND(pvSCL condi\ tion)` or `PVSCL:EVAL(pvSCL calculation)` . See Section 9.5.5, “ ps:pvscltext ” for a list of all statements and a small example text document.

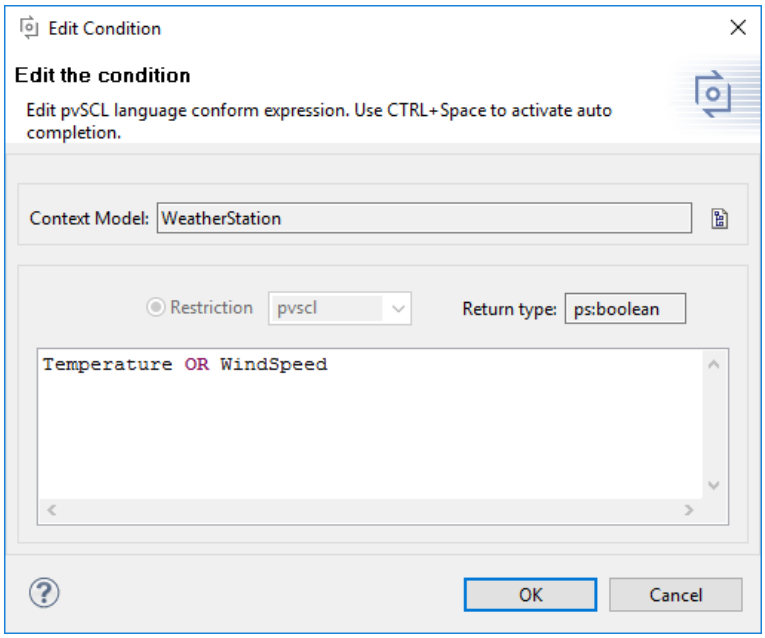
To edit conditions and calculations, use the actions "Add PVSCL Condition" ( ), "Add PVSCL Calculation" ( ), and "Edit PVSCL Condition/Calculation" ( ), which are available in the toolbar. These actions give you the same support in writing pvSCL rules as already known from editing restrictions or constraints.

For adding a condition to a section of your text, mark a section of your text and press . Now a pvSCL editor opens, in which you can write the pvSCL rule that should apply to this text section (see Figure 6.75, “Editing pvSCL conditions or calculations”). For using auto-completion, syntax highlighting, and error checks, the editor still needs to know the pure::variants project, in which context the written rule should be evaluated. Therefore, select the context model by pressing . Only feature models are allowed as context model. However, all other models of the same and referenced projects are considered automatically. After pressing "OK", the new condition is wrapped around the selected text.

Adding a calculation to your text works in a similar way. Press and use the pvSCL editor to write your rule. After closing the editor, the marked text is replaced with the calculation.

To edit an existing condition or calculation, move the text selection to a place inside the calculation or condition and press . For conditions, any place between the beginning of`PVSCL:IFCOND(` and the beginning of `PVSCL:ENDCOND` is ok. For calculations, any place between the beginning of `PVSCL:EVAL(` and the closing `)` is accepted. If a condition or calculation is found at your current text selection or caret position, the pvSCL editor opens, and you can edit your rule.

Figure 6.75. Editing pvSCL conditions or calculations



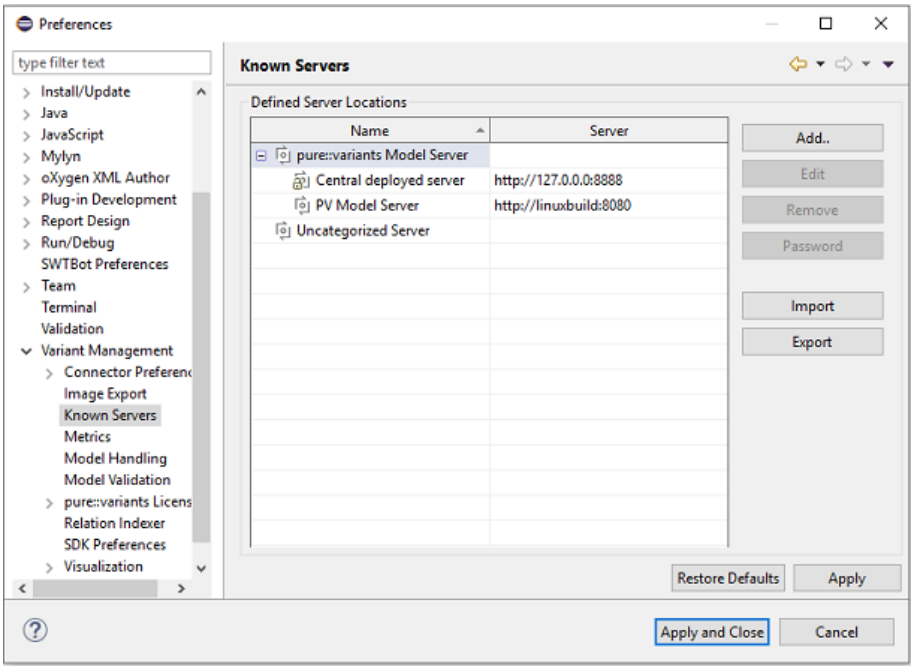
6.17. Using Known Servers Preferences

Known servers can be organized from *Window->Preferences->Variant Management->Known Servers* in pure::variants. Known servers are used by many connectors. This page provides an organized view and actions for servers of the corresponding connectors. The known server table has following components.

Category — Each existing connectors are represented as categories. Categories have unique ID and name. It is possible to see the ID by hoovering mouse pointer on any category.

Server — Servers are shown under each corresponding connector categories. Each server has a name and an URL. More information of the server can be viewed through tooltip.

Figure 6.76. Known Servers page



The following actions can be performed on the list.

- **Add:** The *Add* button only enables when a category is selected. To add a server, select any connector category, then click*Add* . A dialog box will open, enter server name in *Name* text box and server URL in *URL* text box. Press*OK* to add the server into the category.

  If the server fails to connect, a *Save anyway* dialog will open if user want to keep it.

- **Edit:** The *Edit* button enables if any server is selected. A server name or URL can be modified by clicking *Edit* button. A dialog box pops up where changes can be made.

- **Remove:** To remove a server from any category, select it from the list and press*Remove* button.

- **Password:** Password change is only available for servers which are in pure::variants Model Server category. To change the password of such server, press*Password* button.

- **Import:** Servers from a external XML file can be added to their corresponding categories. Use*Import* button to perform an import. The import XML has to have the similar structure as central deployment XML.

  An example is shown in Central deployment XML structure [124]

- **Export:** The list of all categorized servers can be exported to an XML file. By clicking*Export* , a save file dialog will open to create the XML file in prefered location.

6.17.1. Central deployment mechanism of servers

The predefined XML file with categorized servers can be stored in file named *"pv.servers.xml"* on *"C:\Program-Data\pure-variants-5"* directory for central deployment of servers with user's choices of category/categories. An example XML is shown in Central deployment XML structure [124] .

Note

To create such an xml file for central deployment we recommend using the known servers preference page. All necessary Server should be added on that page. Afterwards just export the server list and deploy the resulting XML file.

These servers can not be edited by the user and will be shown in their corresponding connector categories with "lock" () decorated icons. For example, a pure::variant model server from central deployment file would look like ().

Category IDs are mandatory for central deployment XML. pure::variants connectors of corresponding server categories are using category IDs with the following schema: We take the feature ID of the connector and replace "pure-variants" with "servercategory".

Example: For connector with ID com.ps.consul.eclipse.pure-variants.toolxyz the category ID com.ps.consul.eclipse.servercategory.toolxyz is used.

There are some categories which do not follow this schema:

Note

User can have proposal for pure::variants Floating License server in **Window->Preferences->Variant Management->pure::variants License->License Server** by adding a License category server in the central deployment file. The pure::variants Floating License Server category is not shown in Known Servers preference page but can be used from central deployment file.

Table 6.8. Table of server category IDs

Server Category Name	Server Catagory ID
pure::variants Model Server	com.ps.consul.pvserver.model

Server Category Name	Server Catagory ID
pure::variants Floating License Server	com.ps.consul.pvserver.license
Uncategorized Server	com.ps.consul.pvserver.unknown.category

The structure of the central deployment server xml file is:

```
<?xml version="1.0" encoding="UTF-8"?>
<servers>
  <server name="Name of the server 1"
     description="Description of the server 1"
     category="ID of the category"
     url="URL of the server 1" />
  <server name="Name of the server 2"
     description="Description of the server 2"
     category="ID of the category"
     url="URL of the server 2" />
  ...
</servers>
```

Name, category and url are mandatory. The description is optional.

```
<?xml version="1.0" encoding="UTF-8"?>
<servers>
  <server name="PV Model Server"
    description="This is an example server"
    category="com.ps.consul.pvserver.model"
    url="http://127.0.0.1:4711" />
  <server name="Another Model Server"
    category="com.ps.consul.pvserver.model"
    url="http://model.server.local:8080" />
</servers>
```

Note

Eclipse restart is required to reflect the modification of central deployment file.

6.18. Convert a pure::variants 4 project into a pure::variants 5 project

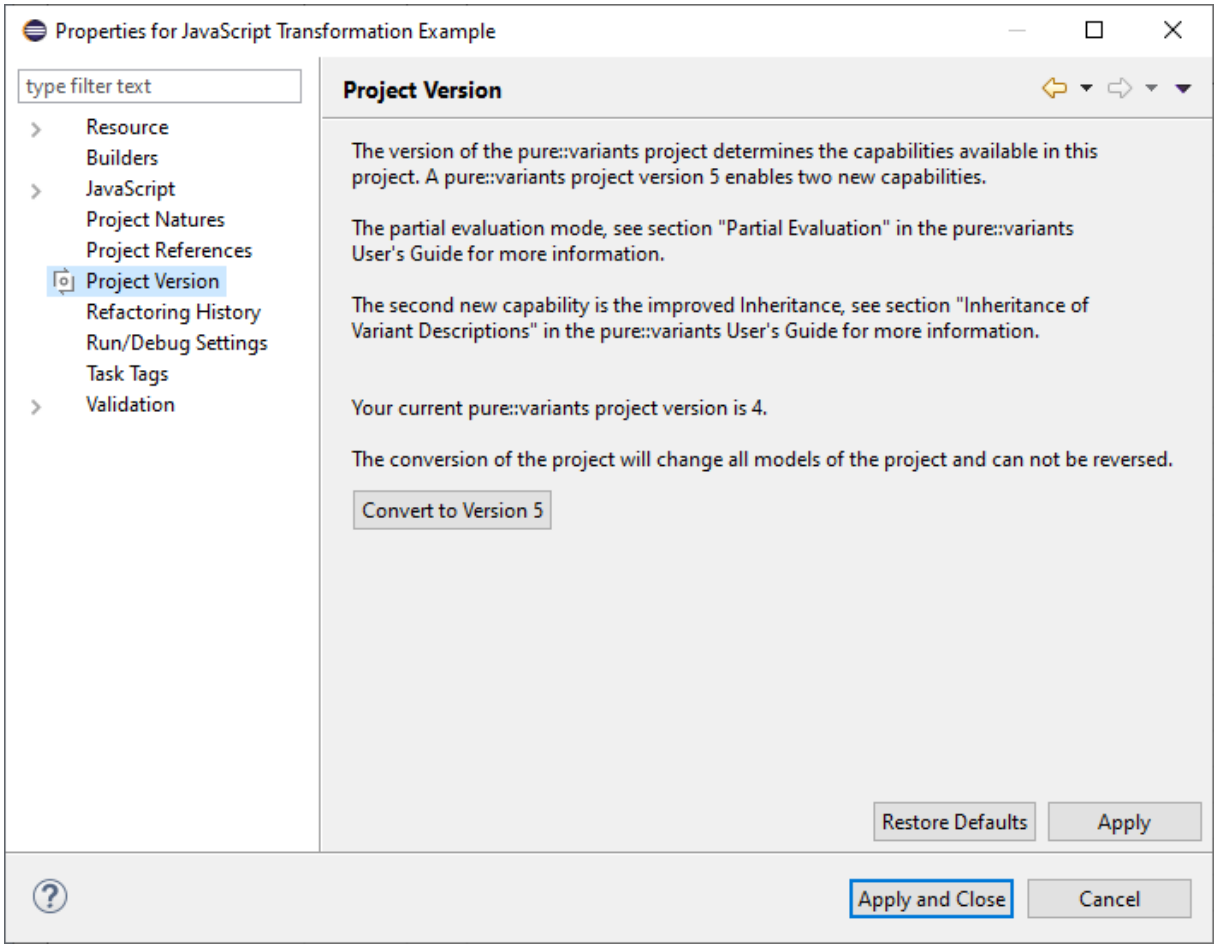
To enable some new pure::variants features a pure::variants project with version 5 is necessary. All newly created pure::variants projects are created with version 5. An existing project created with an older pure::variants release can be converted to a pure::variants project with version 5.

To convert a project open the properties dialog on the pure::variants project. Select **Project Version** in the navigation on the left side of the properties dialog. The selected page now shows the current project version along with some additional informaton. The conversion is started with the**Convert to Version 5** button at the bottom of the page.

Note

After converting a pure::variants project to version 5 it can not be converted back to version 4. The converted pure::variants project is not compatible to pure::variants releases prior to 5.0.0.

Figure 6.77. pure::variants Project Version



If the project to convert has references to other pure::variants projects in the workspace, the user will be asked if the referenced projects shall be converted as well. Chose **Yes** to convert all referenced projects together with the initial project. If the

6.19. Customizing the Variant Configuration Process

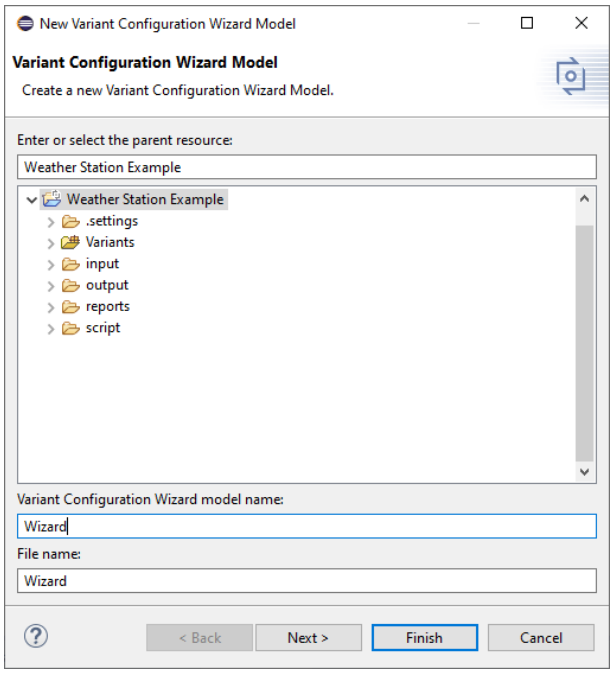
As of pure::variants 5, it is possible to customize the variant configuration process with the help of a Variant Configuration Wizard. A configuration space that is intended to use a Variant Configuration Wizard has to be configured with a **Variant Configuration Wizard Model** . See the section called “ Guided Variant Configuration ” for detailed information about the Variant Configuration Wizard.

This section describes how to create and configure the Variant Configuration Wizard Model.

6.19.1. Creating a Variant Configuration Wizard Model

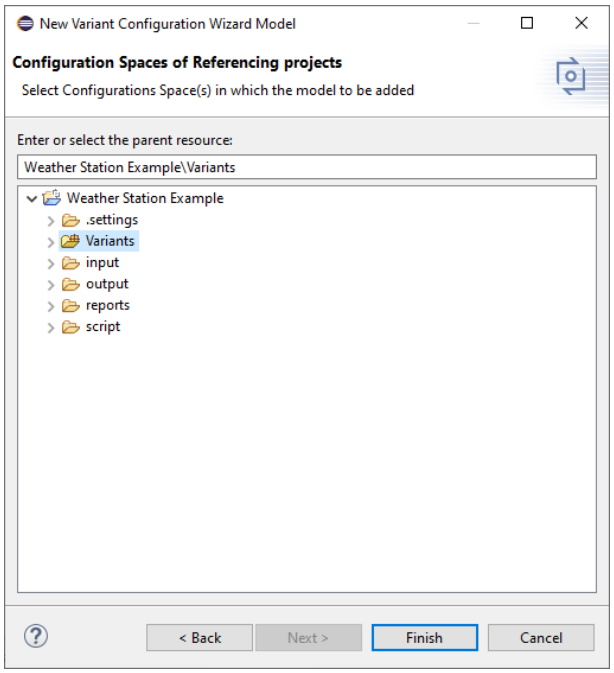
Start the **New Variant Configuration Wizard Model (VCWM)** wizard from the**New** menu of the context menu in the **Variant Projects** view. The following wizard opens. On the first page select a target container and define the name for the new Variant Configuration Wizard Model.

Figure 6.78. New Variant Configuration Model



The wizard can be finished now and the new Variant Configuration Wizard Model will be created. Using the **Next >** button instead switches to the next page where it is possible to select the configuration spaces to which the new model will be added automatically. The page lists configuration spaces from the target project and all projects which reference the target project. The configuration spaces to which the Variant Configuration Wizard Model is added can also be changed later on. See the following section for more information on how to do that.

Figure 6.79. Add the new Variant Configuration Model to Configuration Spaces

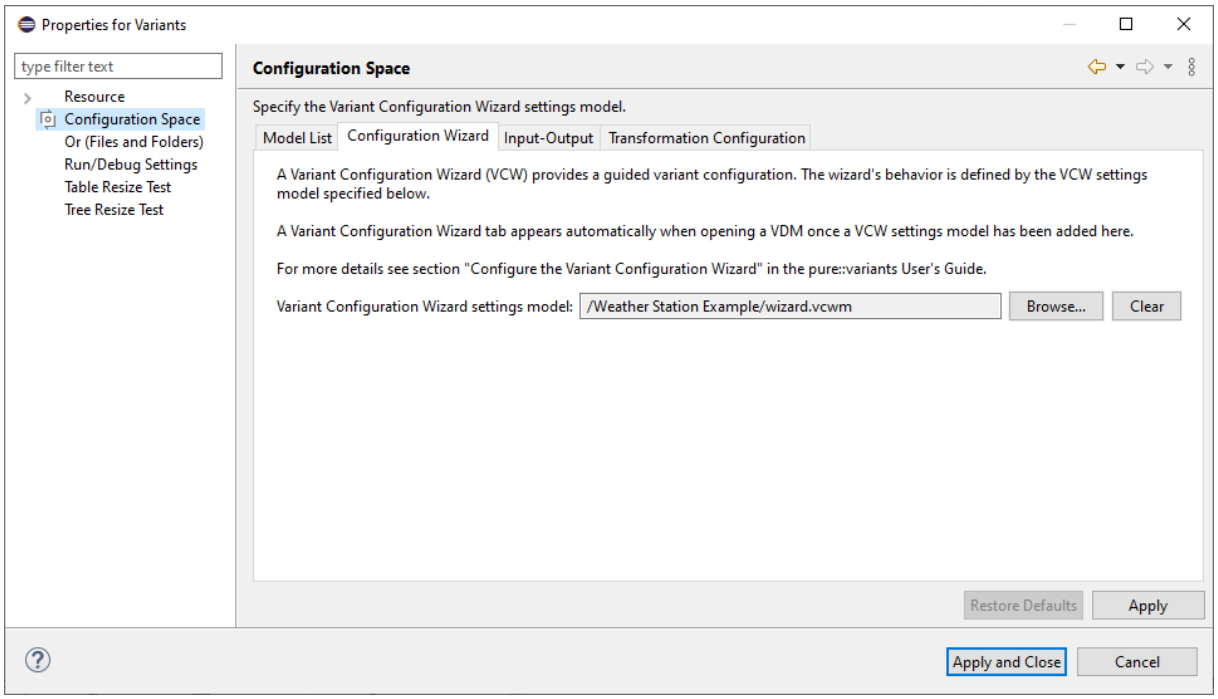


Adding the Variant Configuration Wizard Model to a Configuration Space

To add a Variant Configuration Wizard model to a configuration space, open the properties of the configuration space and navigate to the **Configuration Wizard** tab. The Variant Configuration Model is selected with the

Browse ... button. The **Clear** button allows the user to remove a Variant Configuration Model from the confguration space.

Figure 6.80. Add a Variant Configuration Model to a Configuration Space



6.19.2. Configure a Variant Configuration Wizard Model

The Variant Confiuration Wizard Model provides options to configure the Variant Configuration Wizard. These options are configured in the Variant Configuration Wizard Model editor. The editor is divided into 3 sections, the **General Wizard Setting** section at the top, the **Start Page Section** in the middle and the**Finish Page Section** at the bottom of the editor.

The **General Wizard Settings** section defines the configuration step wizard pages that will be displayed on the left side of the wizard. The pages can be defined in two ways. The first way is to select the**Root of Wizard Pages** using the **Set** button. All direct children of this element are automatically added as configuration pages. The resulting wizard pages are listed in the table below. Sorting or removing them is not possible. The **Clear** button removes the selected root element. If a root element is already defined, clicking**Clear** will discard all previously defined **Wizard Pages** .

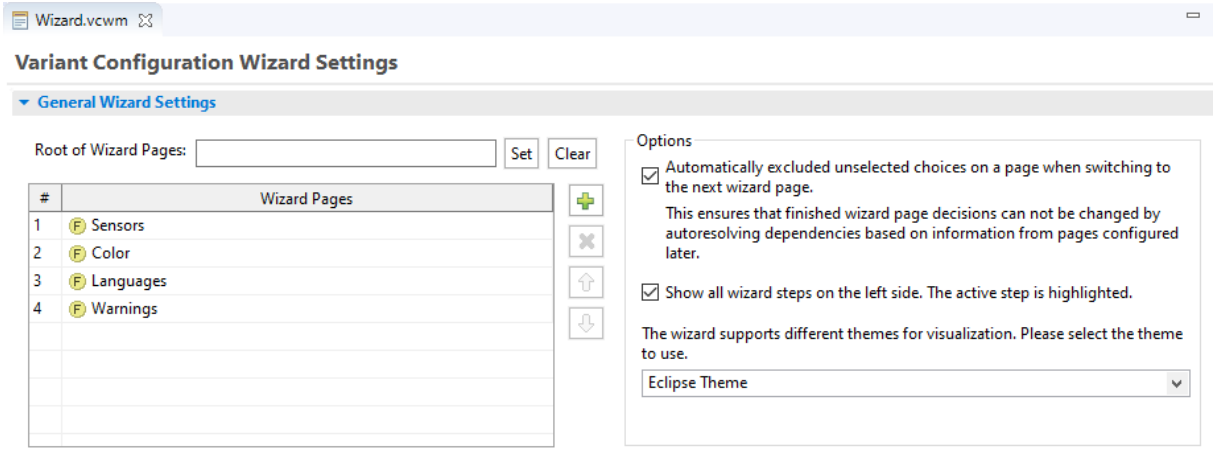
The second way to define the wizard pages is to select the defining elements manually. Use the plus icon next to the table to add wizard pages. Each entry in the table will be one wizard page in the wizard. The order in the table is also the order of the pages in the wizard. The table allows to sort the entries using the up and down buttons as well as to remove elements using the **x** button.

The right side defines general options for the wizard's behavior. If the first option is enabled the wizard automatically excludes all non-selected elements when the user switches from one wizard page to the next. This prevents the auto resolver from automatically selecting elements (based on rule knowledge from the input models) whose configuration was already completed in a previous configuration step.

The second option in the right half of the **General Wizard Settings** section enables or disables the navigation part of the Variant Configuration Wizard. If this option is selected, the resulting wizard will show the list of configuration steps. Otherwise, the configuration steps will be hidden.

Finally, if more than one wizard theme is available, then the desired theme can be selected with the combo box of the third option.

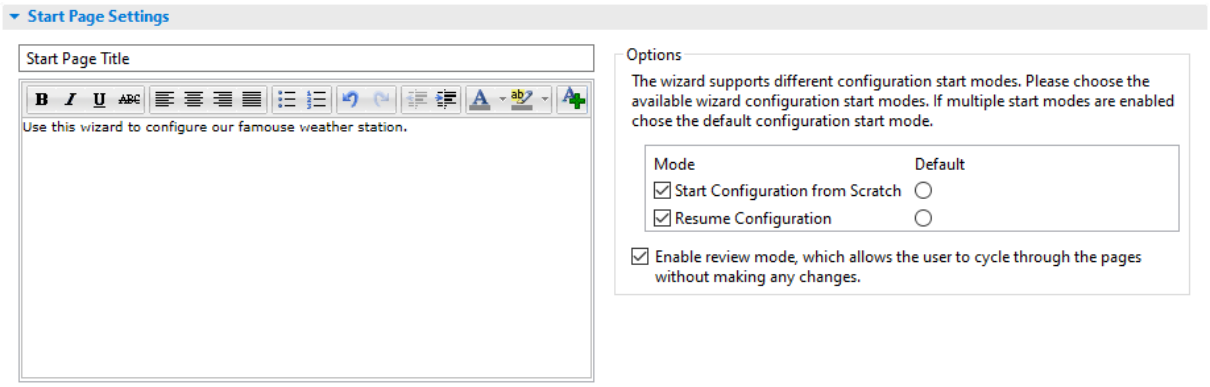
Figure 6.81. VCWM Editor General Settings Section



The **Start Page Settings** section defines the user message and the page title on the left side. The right side of the section defines which start page options will be available on the start page and, if more than one mode is selected, the default mode needs to be specified.

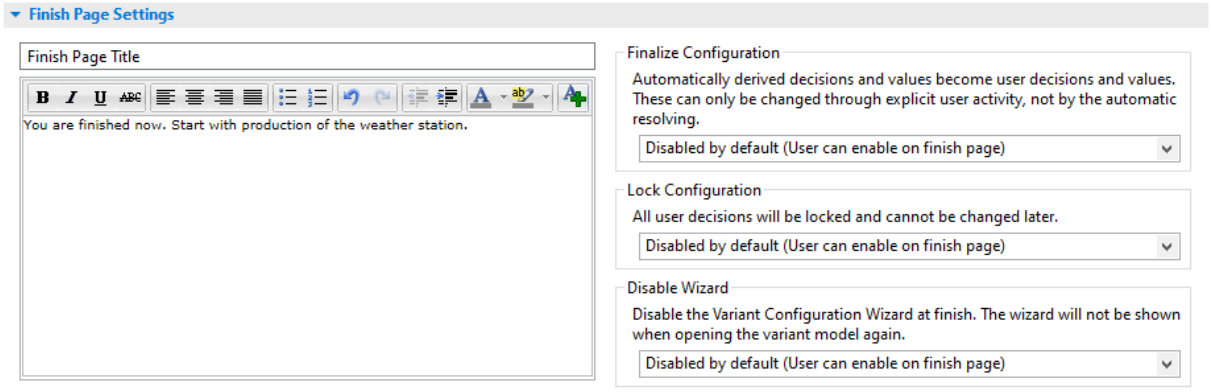
Selecting **Enable review mode** enables the review mode. This is the default start mode of the wizard, but this can be changed by the user, if at least one other start mode is available.

Figure 6.82. VCWM Editor Start Page Section



The **Finish Page Settings** section defines the user message and the title of the final wizard page, as shown in the list of configuration steps on the left side. The right side of the**Finish Page Settings** section defines which finishing options will be presented to the user on the finish page and which one is the default. Three options are available.**Finalize Configuration** , **Lock Configuration** and **Disable Wizard** . The meaning of each of these options is explained on the right side of the Finish Page Settings section (seeFigure 6.83, “VCWM Editor Finish Page Section”). The combo boxes below each option allow to choose if and how an option is presented on the wizard's finish page. The choices are **Disabled by default** , which means that the option is disabled on the finish page but the user can select it,**Enabled by default** , which means that the option is enabled on the finish page but the user can deselect it, **Always disabled** , which means that the option is disabled and not shown on the finish page and, finally,**Always enabled** , which means that the option is enabled and not shown on the finish page.

Figure 6.83. VCWM Editor Finish Page Section



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

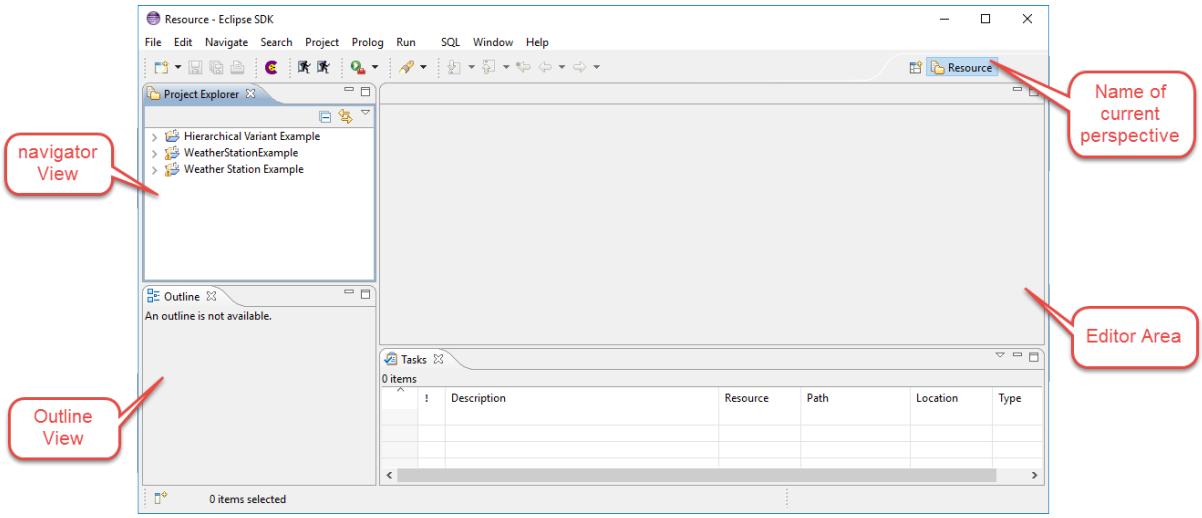
This section gives a short introduction to the elements of the Eclipse UI before introducing the pure::variants UI. Readers with Eclipse experience may skip this section.

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*Navigator* view or by using a context menu. When there are several suitable editors for a given resource type the context menu allows the desired one to be chosen. The figure below shows some of the main User Interface elements:

Figure 7.1. Eclipse workbench elements

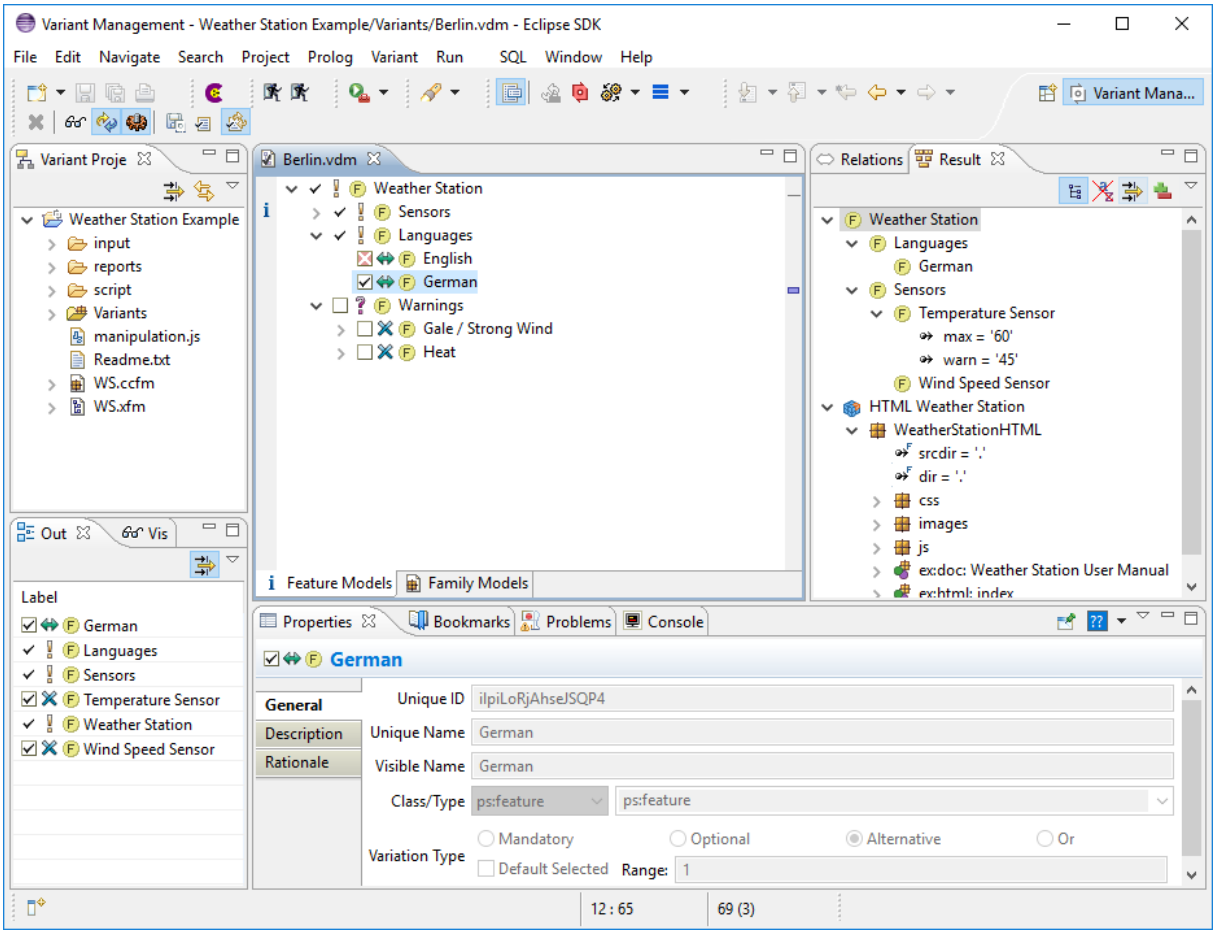


Eclipse uses *Views* to represent any kind of information. Despite their name, data in some types of view may be changed. Only one instance of a specific type of view, such as the Outline view, may be shown in the workbench at a time. All available views are accessible via Windows->Show View->Other.

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 (seeSection 9.10, “ 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.

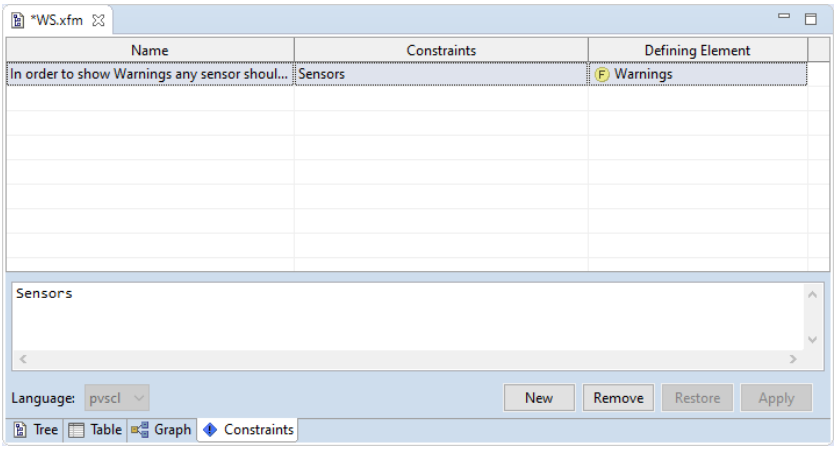
Most cells in table views are directly editable. A single-click into a cell selects the row; a second click opens the cell editor for the selected cell. The context menu for a row permits addition of new elements or deletion of the row. A double-click on a row starts a property dialog for the element 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. For pure::variant 5 projects constraint can also be added to variant models Constraints can be edited or new created on this page. It also supports to change the element defining a constraint. The defining element is not available for variant models.

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 .

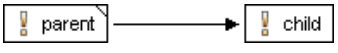
For nearly all actions on a graph that are explained in the next sections keyboard shortcuts are available listed in Section 9.10, “ Keyboard Shortcuts ” .

Graph Elements

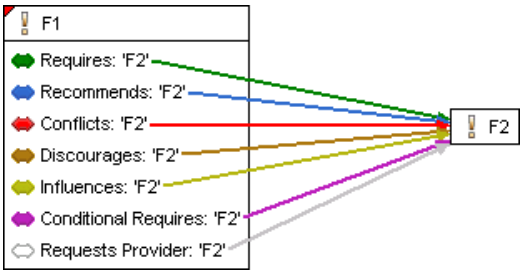
Model elements are represented in the graph as boxes containing the name of the element and an associated icon. Feature model elements are represented as shown in the next figure.

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

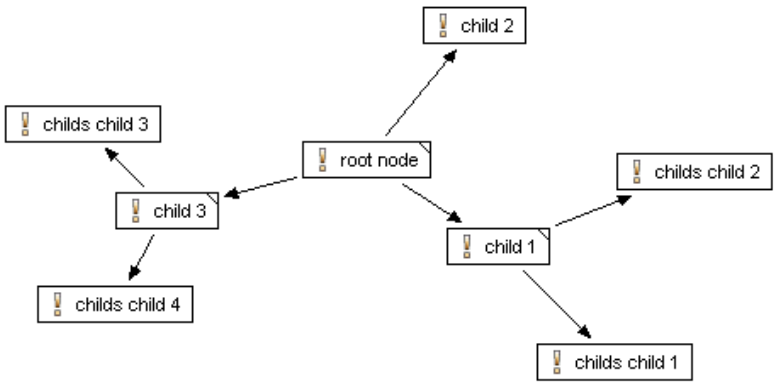
The layout of the graph can be changed in several ways. Graph elements can be moved, expanded, collapsed, hidden, and automatically aligned. The graph can be zoomed and the layout of the connections between the elements of the graph can be changed.

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.

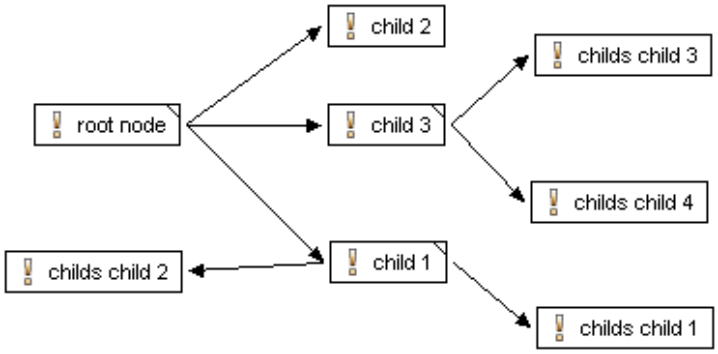
No Docking Rule

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



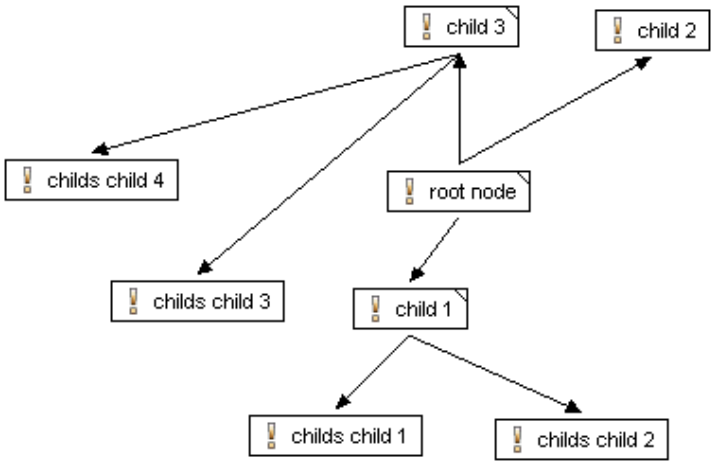
Dock Connections on Left or Right

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.

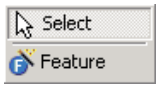
Several elements can be selected by holding down the **SHIFT** or **STRG** key while selecting further elements, or by clicking somewhere in the empty space of the graph visualization page and dragging the mouse over elements. A dashed line appears and all elements that are partially or wholly enclosed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

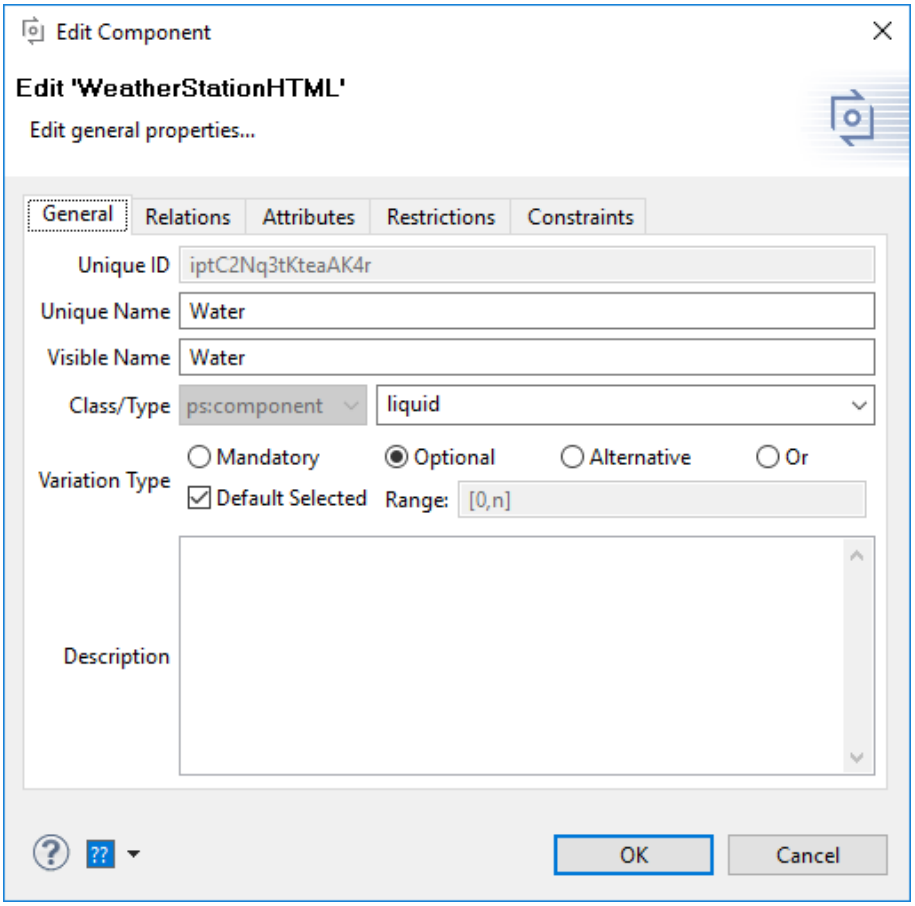
Element Properties Dialog

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 *ps:feature* . Thus the element class for features cannot be changed. Elements in Family Models can have one the following classes: *ps:component* , *ps:part* , or *ps:source* . The root element of a family model always has the class*ps:family* . 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 *"mandatory"* , *"optional"* , *"alternative"* or *"or"* can be selected.

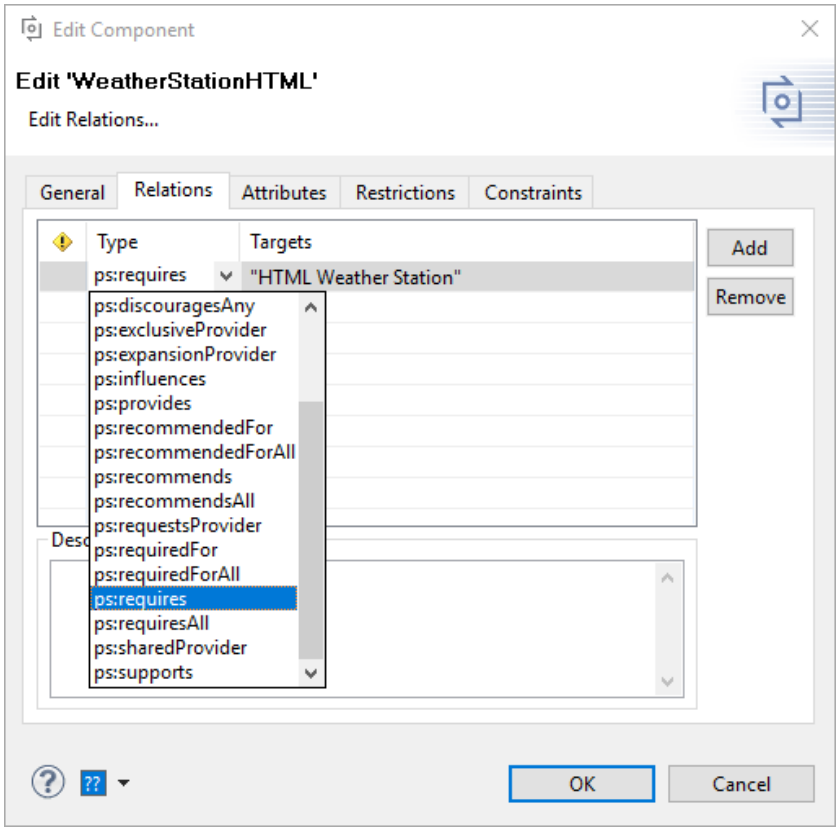
Range	For variation type *Or* it is possible to specify the number of features / family elements that have to be selected in a valid configuration in terms of a range expression. These range expressions can either be a number, e.g. 2, or an inclusive number range given in square brackets, e.g. [1,3], 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 *Or* group.
Default Selected	This property defines the default selection state of a model element. Default selected elements are selected automatically 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. For defining a new custom relation type the name of the new type can be entered into the text filed into the **Type** column instead of selection on predefined relation from the dropdown list.

More information on element relations can be found in Section 5.2.3, " Element Relations " .

Figure 7.7. Element Relations Page



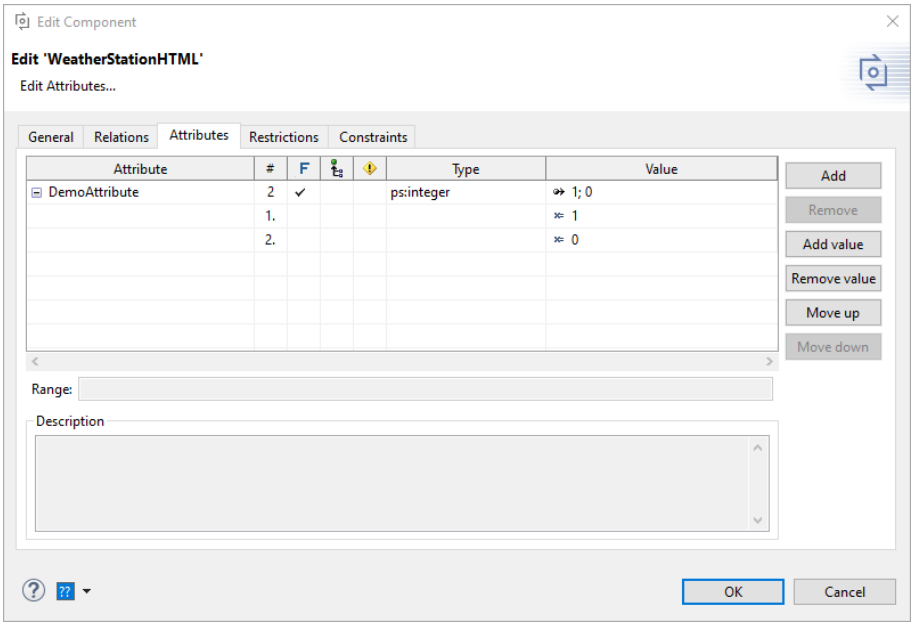
Attributes Page

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



Each attribute of type *ps:integer* or *ps:float* may define a range which the attribute values have to fit in. This range can be defined in the "Attribute page" of an element while creating the attribute or theSection 7.4.6, “ Properties View ” after selecting the corresponding attribute. The syntax of the ranges is described in Section 5.2.4, “ Element Attributes ”

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. Seethe 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 *pvSCL* language to provide more complex value definitions. More information on calculating attribute values is given inthe section called “ Attribute Value Calculations with pvSCL ” .

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.

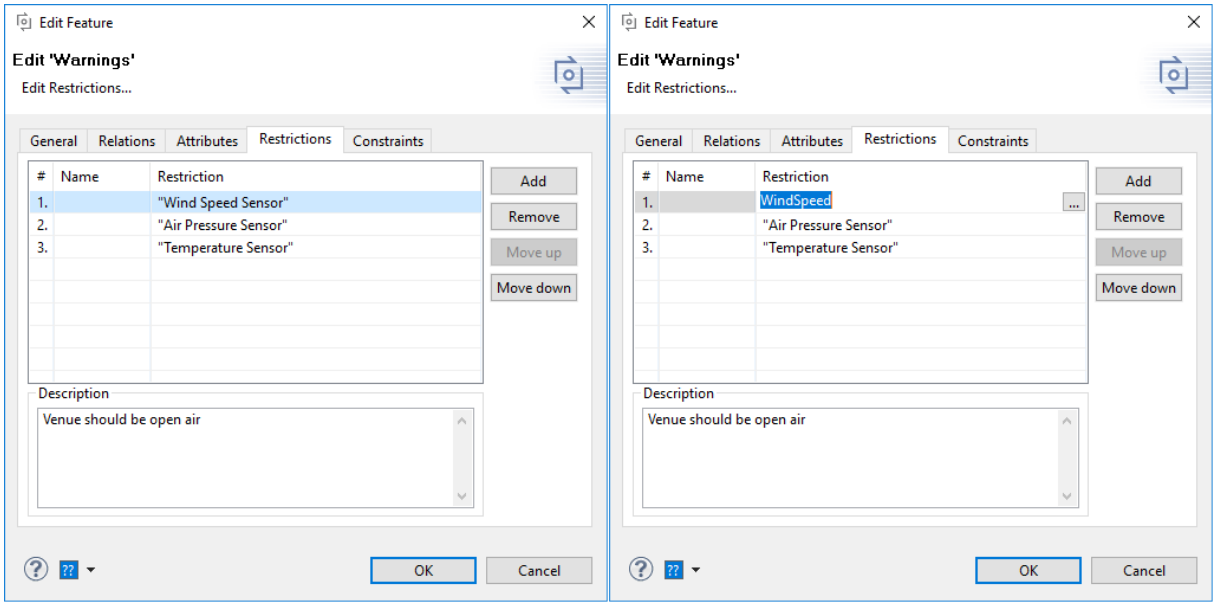
It is also possible to provide attributes which have a configurable collection of values as data type. Each contained value is available in a variant if the corresponding restriction holds true. To use this feature, square brackets ("[]") for list values or curly brackets ("{}") for set values have to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



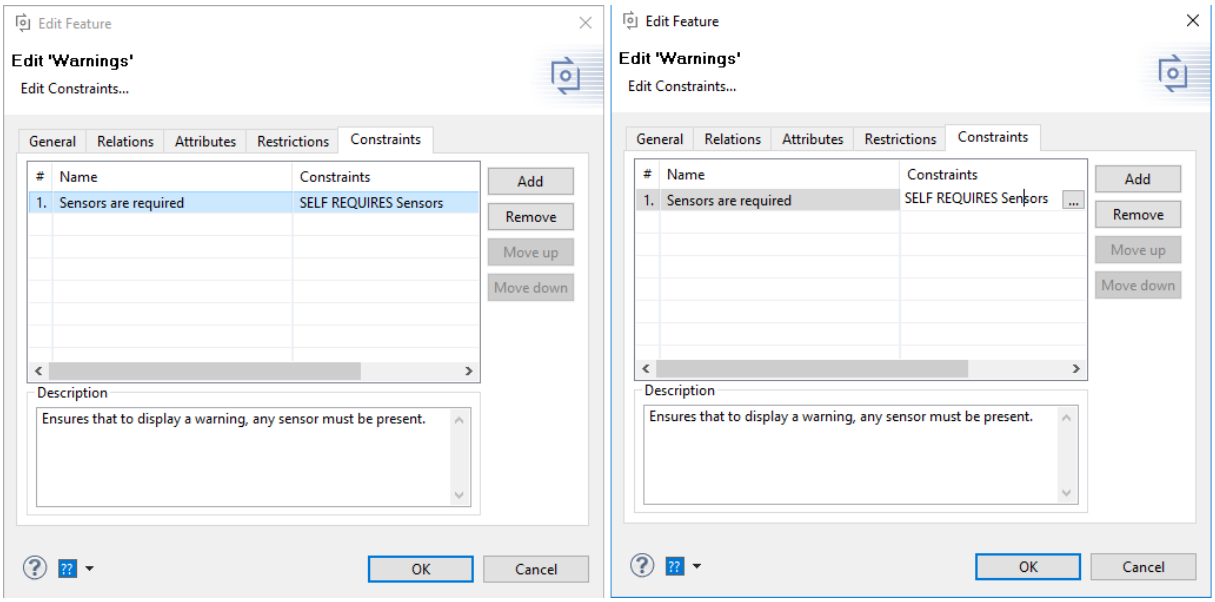
For each restriction a descriptive name can be specified. It has no further meaning other than a short description of what the restriction checks. A restriction can be edited in place using the cell editor (shown in the right side of figure Figure 7.9, “Restrictions page of element properties dialog”). 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 Visible names (e.g. 'Wind Speed Sensor) when available. When the editor is opened the unique name is shown (e.g. 'WindSpeed'), 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 inthe 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



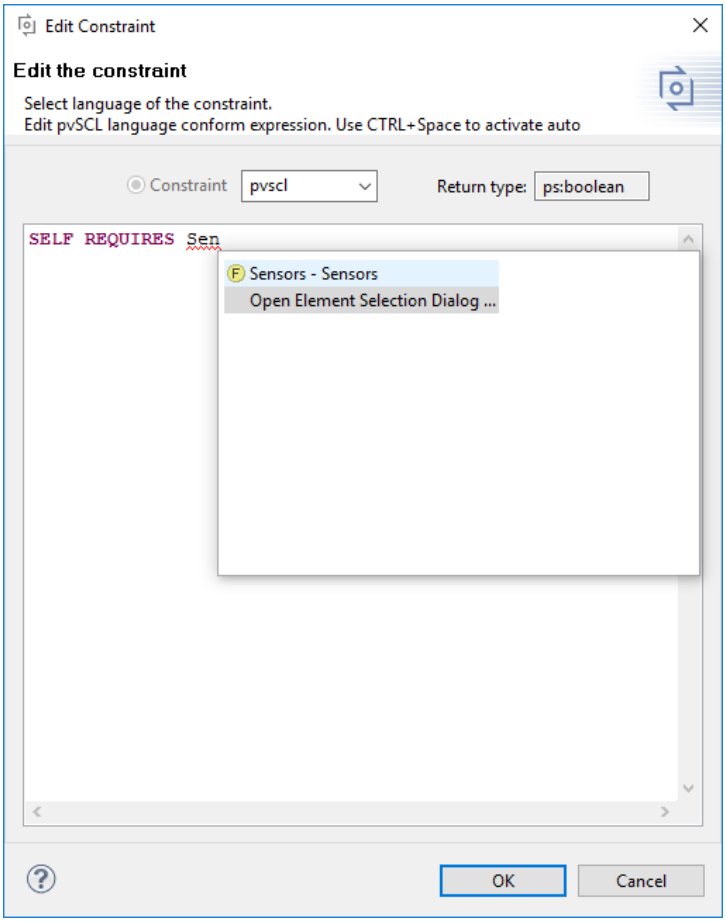
For each constraint a descriptive name can be specified. It has no further meaning other than a short description of what the constraint checks. A constraint can be edited in place using the cell editor (shown in the right side of figure Figure 7.10, “Constraints page of element properties dialog”). The ... button opens an advanced editor dialog that is more suitable for complex constraints. This editor is described more detailed inthe 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 *pvSCL* language. A special editor is available for the pvSCL language.Figure 7.11, “Advanced pvSCL expression editor” shows the *pvSCL* editor editing a constraint.

Figure 7.11. Advanced pvSCL expression editor

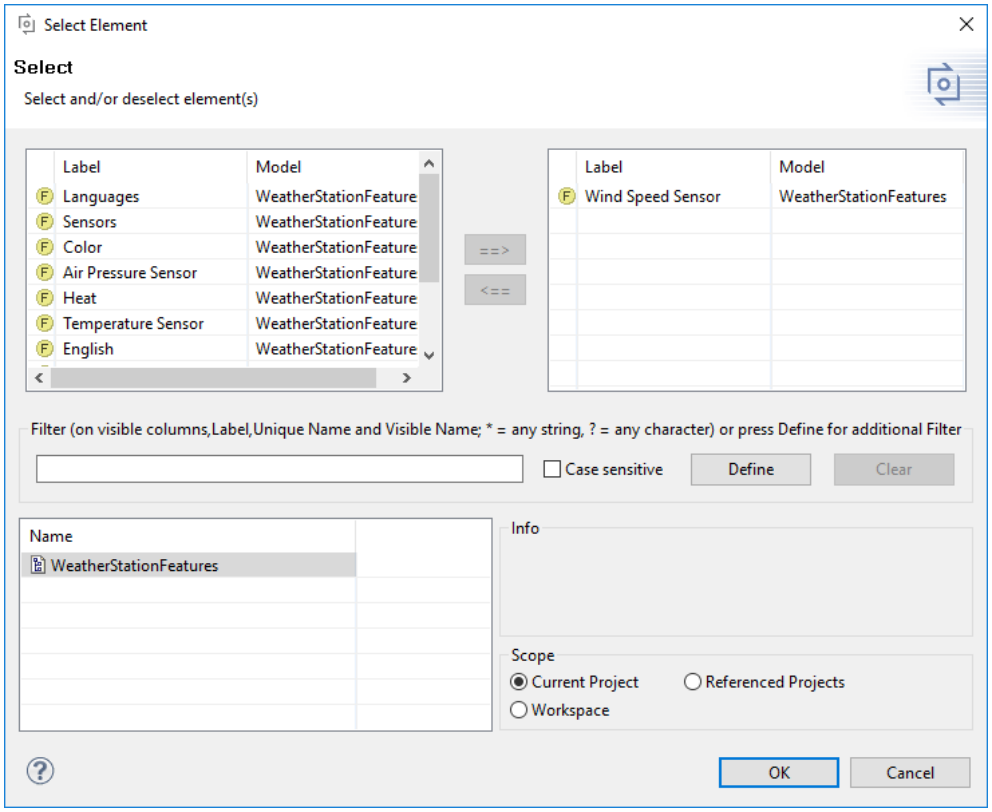


This dialog supports syntax highlighting for *pvSCL* keywords and auto completion for identifiers. There are two forms of completion. Pressing **CTRL+SPACE** while typing in an identifier opens a list with matching model elements and*pvSCL* keywords as shown in the figure. If the user enters "<ModelName>." 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 ” .

Element Selection Dialog

The element selection dialog (figure Figure 7.12, “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 7.12. 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

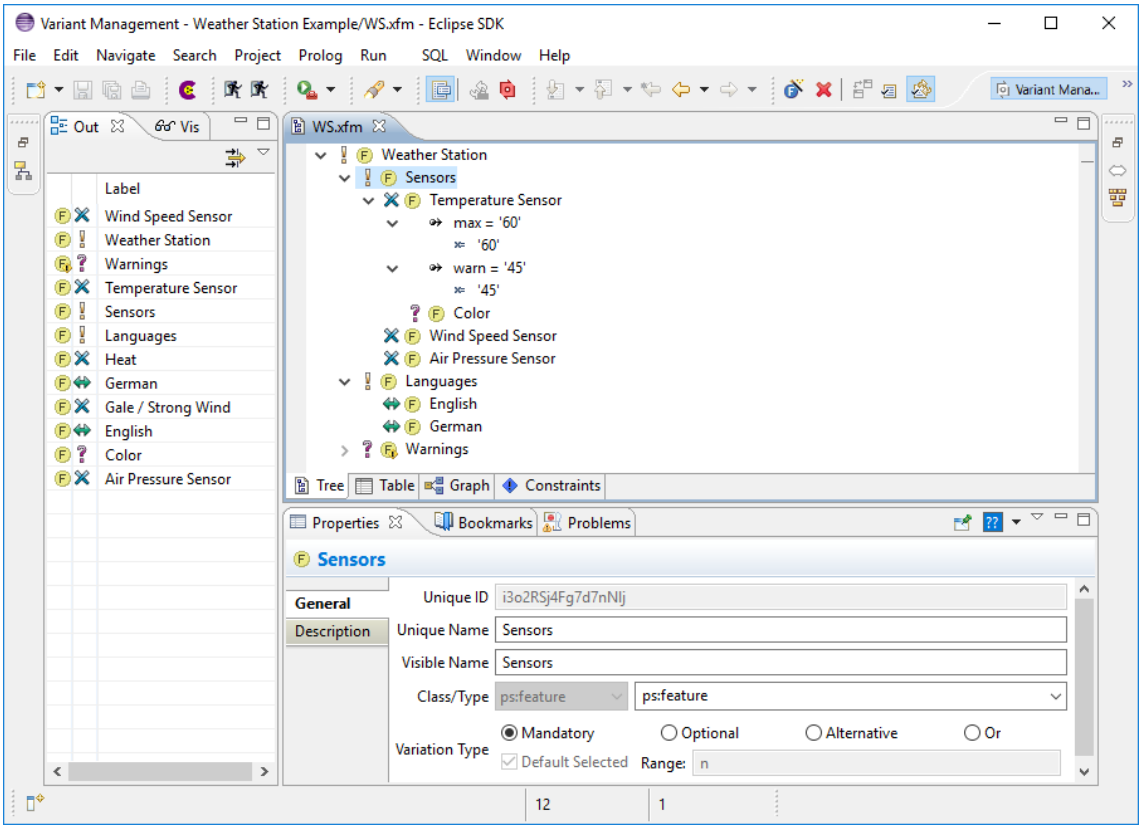
The element information shown in the left and right Label fields is configurable. Use *Table Layout->Change...* from the context menu to select and arrange the visible columns. Seethe section called " Table Editing Page " for additional information on table views.

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 4.4, "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 9.3, "Element variation types and its icons").

Figure 7.13. Feature Model Editor with outline and property view



Some keyboard shortcuts are supported in addition to mouse gestures (see Section 9.10, “ Keyboard Shortcuts ”).

Creating and Changing Features

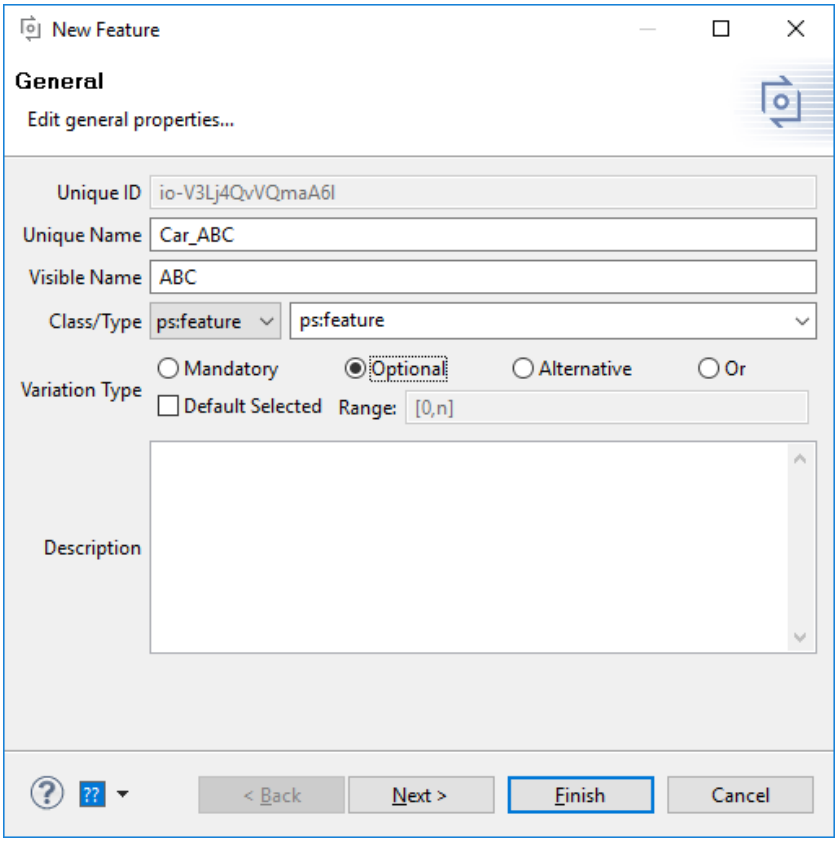
Whenever a new Feature Model is created, a root feature of the same name is automatically created and associated with the model.

Additional sub-features may be added to an existing feature using the **New** context menu item. This opens the New Feature wizard (seeFigure 7.14, “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 (seeSection 9.10, “ Keyboard Shortcuts ”).

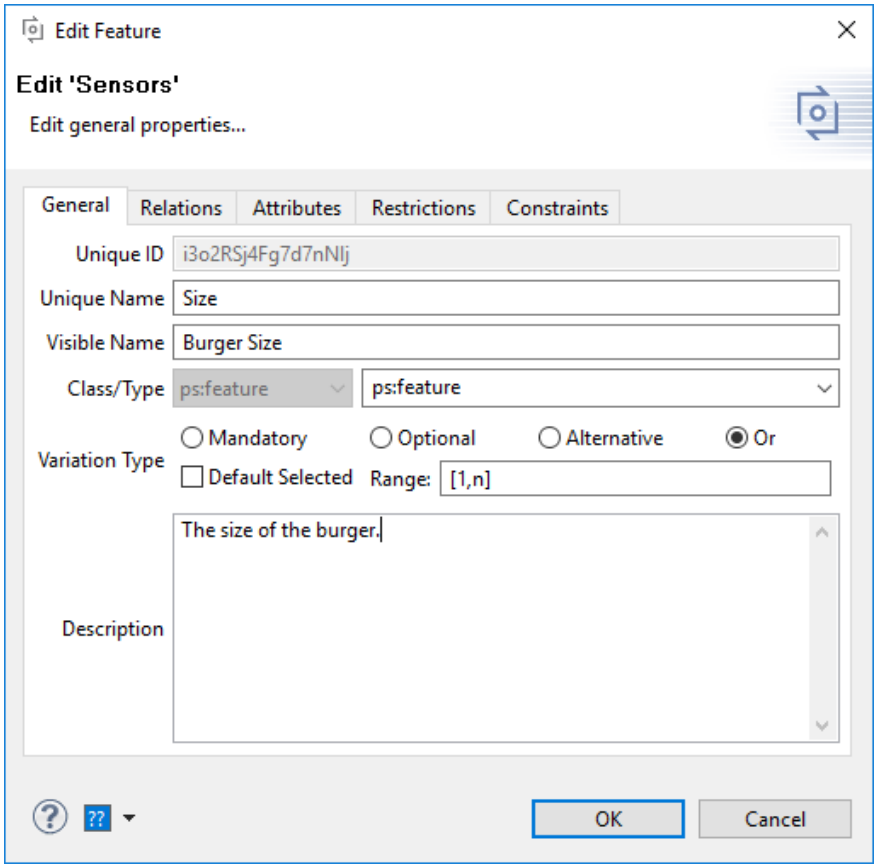
Figure 7.14. New Feature wizard



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 7.15, “Feature Model Element Properties”).

Figure 7.15. Feature Model Element Properties

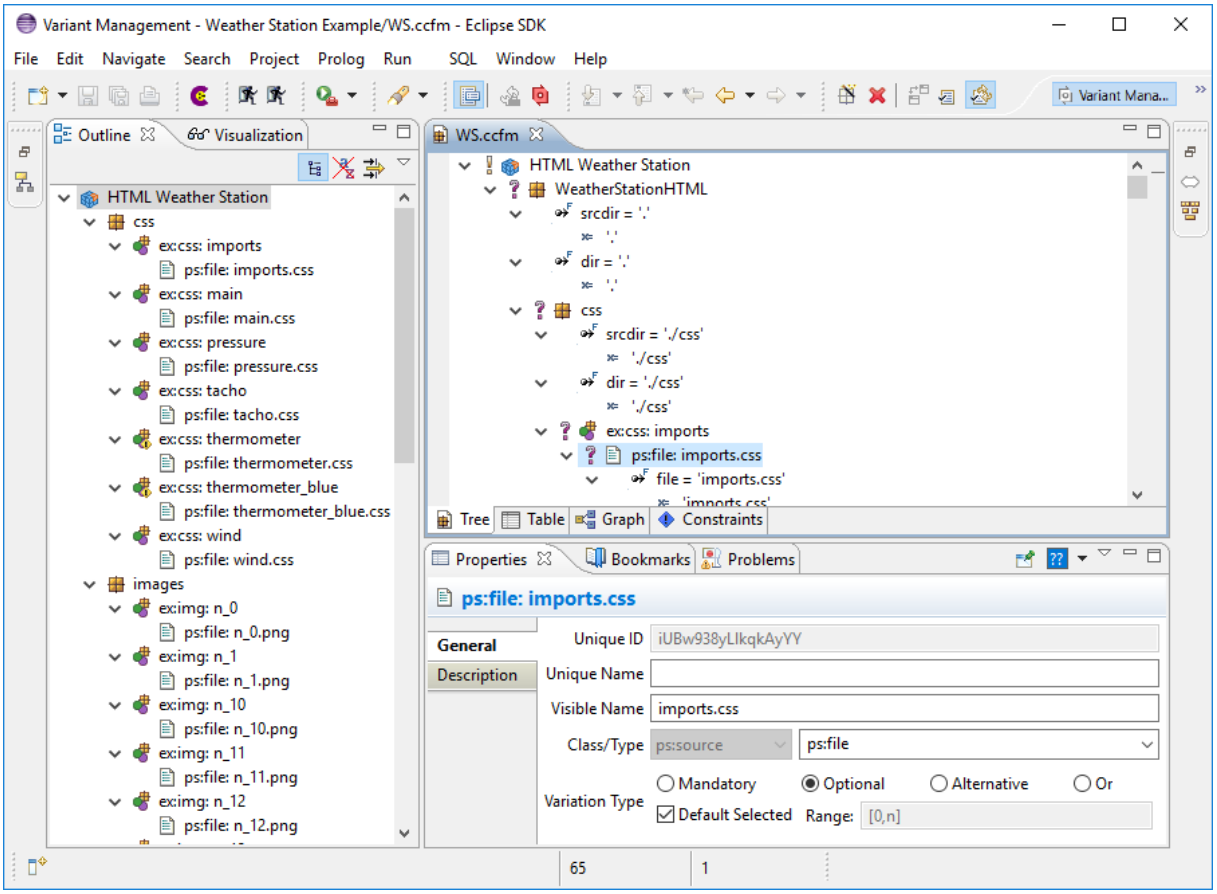


See the section called “ Element Properties Dialog ” for more information about the dialog.

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 9.6, “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 seeSection 5.4, “ Family Models ” .

Figure 7.16. 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.

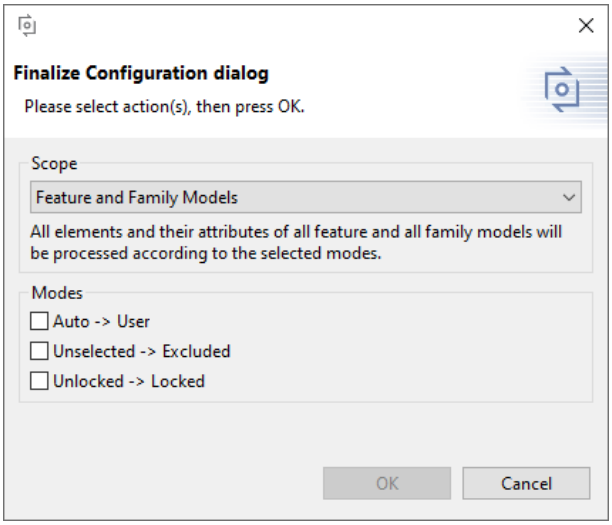
Each selected element is shown with an icon indicating how the selection was made. The different types of icons are documented in Table 9.4, “Types of element selections” . If the user selects an element that has already been selected automatically its selection type becomes user selected and only the 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.

Since automatically calculated selections may be changed during evaluation by the auto resolver to make the selectgions valid the Variant Description Model editor provides an action to make the current selection explicit.

Meaning the current automatic calculated selection can be changed to explicit user selections to prevent the auto resolver from changing them. This is done with the *Finalize Configuration* from the editors context menu. This action opens a new dialog whioch allows the user to select which selections will be changed to explicit selections.

Figure 7.17. Finalize Configuration Dialog



First the scope allows the user to selected wether the feature or family models or both shall be considered. The modes allow the user to select wether auto selections shall be converted into user selection and if unselected elements shall be excluded. Additionally the converted selections can be locked, so the user can not change them by accident.

The *Reopen Configuration* action reverts the finalization.

Guided Variant Configuration

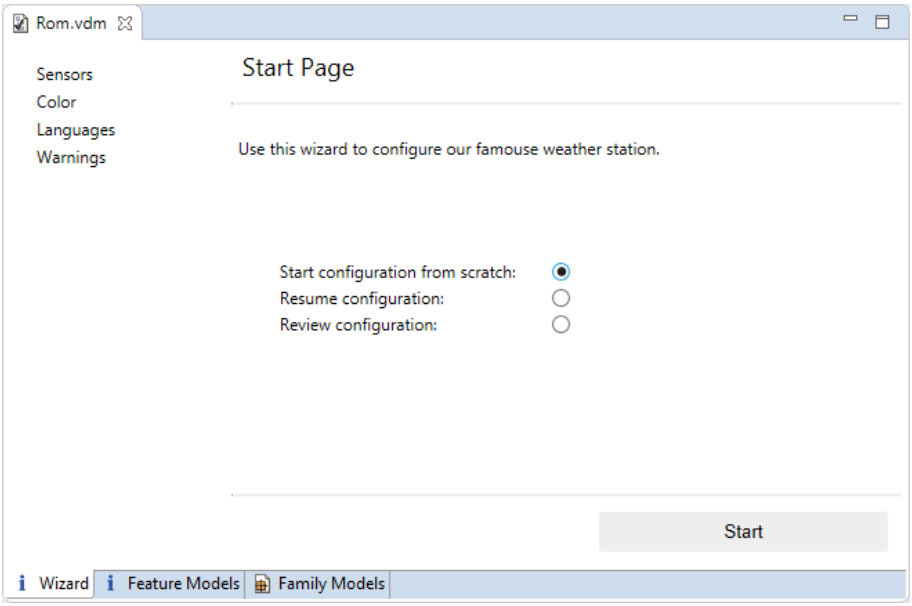
In addition to configuring variants in the **Variant Description ModelEditor** , pure::variants offers the possibility to create Variant Configuration Wizards that guide the user through the configuration process. The Variant Configuration Wizard is available in the**Variant Description Model Editor** and as part of the **Model Viewer** in the **pure::variants Web Components** .

If a configuration space is configured to use a **Variant Configuration Wizard Model** the **Variant Description Model** editor shows an additonal editor viewer named**Wizard** . See Section 6.19, “Customizing the Variant Configuration Process” for detailed information on how to configure the Variant Configuration Wizard.

The wizard is divided into two areas. The left area lists the configurations steps that the wizard provides. The bigger area on the right is the configuration area. It allows the user to make selections and also displays the start and finish page of the wizard.

When the configuration wizard is launched, the start page of the wizard displays startup options. (See Figure 7.18, “Variant Configuration Wizard Start Page”) Depending on the configuration of the**Variant Configuration Wizard** the start page lists the following startup options.**Start configuration from scratch** discards all previous selections and resets the variant model to its initial, unconfigured state. **Resume configuration** resumes the configuration at the point where the user left the configuration the last time.**Review configuration** allows the user to view the configuration without being able to change any selection. This is the only mode which allows the user to navigate through the pages without changing selections.

Figure 7.18. Variant Configuration Wizard Start Page

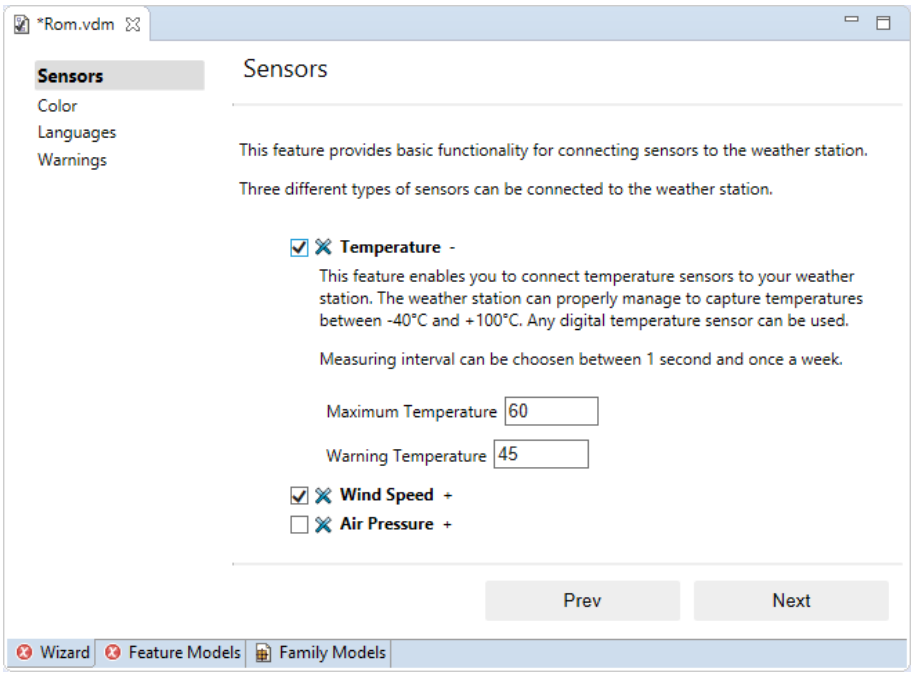


After clicking the **Start** button, the user is guided through the configuration process step by step. Each configuration step is displayed on a single page in the wizard, and this page lists all the configuration items that are necessary to complete the corresponding configuration step. (SeeFigure 7.19, “Variant Configuration Wizard Step Page”) If a configuration item has an associated description, this description is shown below the item. In addition to individual configuration items, a configuration step itself can also have a description. This description is shown at the top of the page.

In this example, shown in Figure 7.19, “Variant Configuration Wizard Step Page” an or group is shown, which means that at least one element has to be selected. Selecting elements may change the content of the step page. Since selecting **Temperature** requires configuring the values of the attribute **Maximum Temperature** and **Warning Temperature** those two attributes automatically become visible on the page.

The buttons **Prev** and **Next** allow page navigation. **Next** is avilable only after all items in the current configuration step have been configured. Using the**Prev** button resets all configuration decisions that have been made on the current page and navigates back to the previous page.

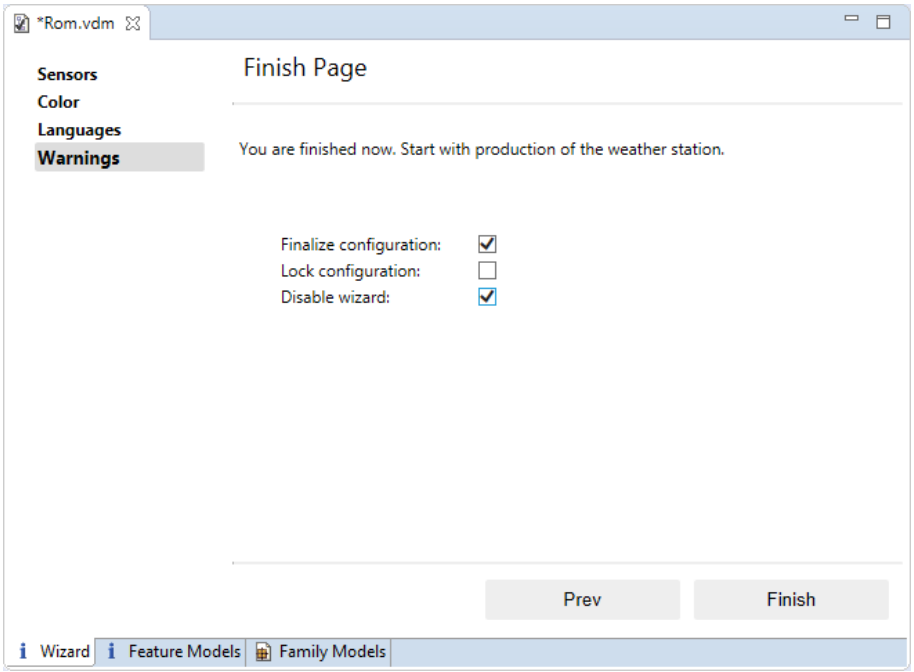
Figure 7.19. Variant Configuration Wizard Step Page



After all configuration steps are done, the finish Page is shown (See Figure 7.20, "Variant Configuration Wizard Finish Page"). The finish page lists the following options: **Finalize configuration** automatically converts derived selections and values into user selections and values. The effect of this conversion is that all configuration decisions made in the wizard, even those that were computed by the auto resolver, are treated as if they were made manually by the user. As such, the auto resolver will not change these decisions accidentally if the variant model is reopended later on. The only possibility to revise these configuration choices is through explicit user interaction.**Lock configuration** locks all user selections so they can not be changed later.**Disable wizard** disables the wizard for the currently configured Variant Description Model. This means the wizard is not shown, if the Variant Description Model is opened again.

Pressing the **Finish** button performs the selected actions and saves the Variant Description Model.

Figure 7.20. Variant Configuration Wizard Finish Page



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:

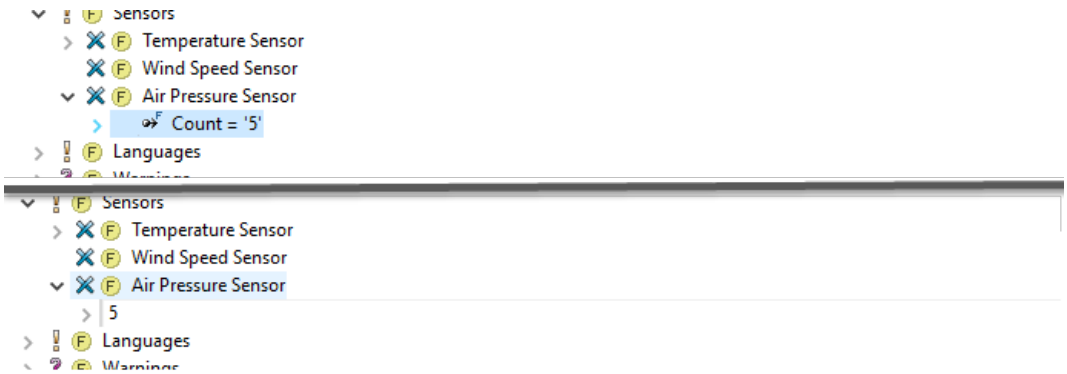
- with the Properties view (see Section 7.4.6, “ Properties View ”)
- with the Attributes view (see Section 7.4.1, “ Attributes View ”)
- with the cell editors of the Variant Description Model Editor itself

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.

Alternatively, values can be added to a non-fixed/editable attribute of a VDM or other models by right-click on it and navigating to **New** ->**Attribute value**. This action will provide relevant dialogs to input values. By pressing OK in the dialogs, the value can be stored in the attribute.

Figure 7.21. Specifying an attribute value in VDM with cell editor



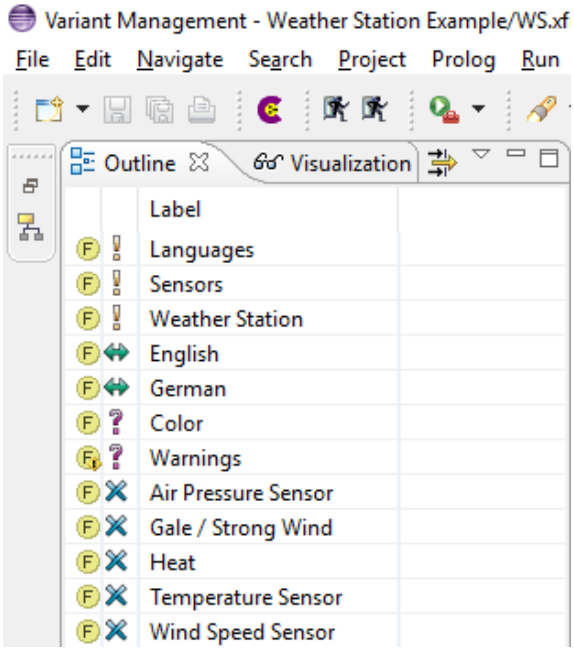
For list and set attributes a special dialog appears when editing attribute values in VDMs. The table represents the values and provides possibility to add (using Add value button), edit (by double clicking the table cells), remove (one or multi select) or re-arrange values.

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



7.3.5. Variant Result Model Editor

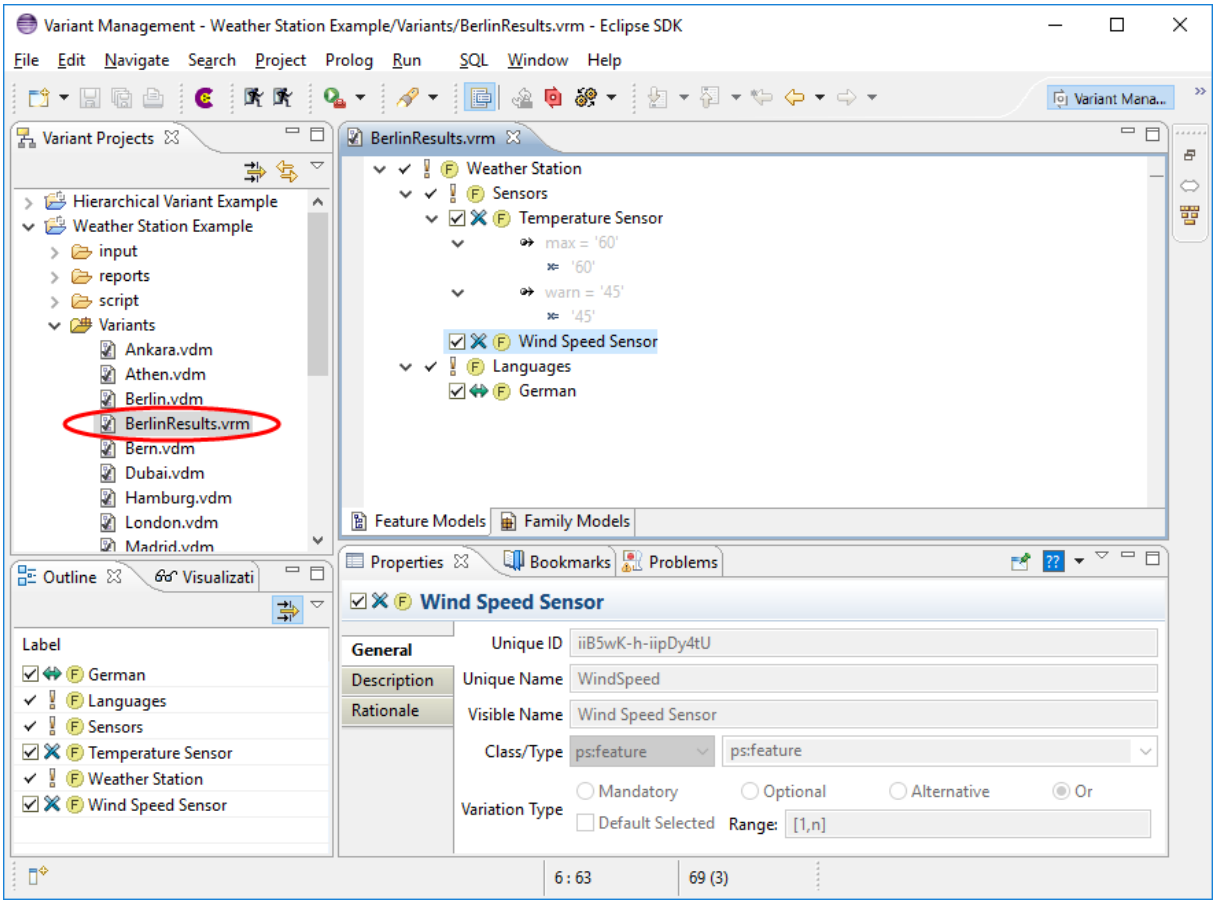
The Variant Result Model Editor (VRM Editor) is used to view a saved Variant Result Model. To open a Variant Result Model, double-click on the corresponding file (suffix `.vrm`) in the Variant Projects View. This opens the editor in the style of the VDM Editor.

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 7.23, “VRM Editor with outline and properties view” shows a sample variant result model.

Figure 7.23. VRM Editor with outline and properties view



See Section 5.9.2, " Variant Result Models " for more information about Variant Result Models.

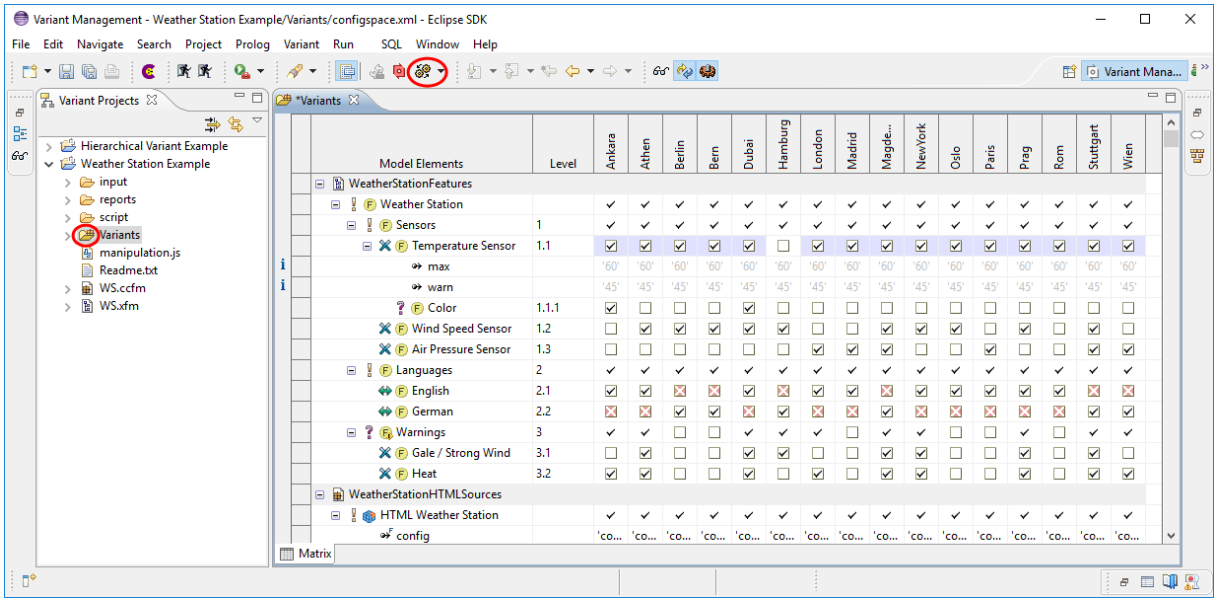
7.3.6. Model Compare Editor

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 7.24, "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 7.24. Matrix Editor of a Configuration Space



The Matrix Editor allows to change selections 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...)* . 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. The Input Model Values column, which is visible by default, can be shown but not moved, since its supposed to show the values of attributes as they are defined in the input model and needs those next to it. Additionally the table layout allows the user to define the VDMs visible in the matrix. This selection can be stored in a*Matrix Variant Filter* . Those filters can be used to open the matrix on the VDMs only matching the filter as well as starting transformation and evaluation on the same filter matching VDMs.

To store the currently opened VDMs in a filter use the *Create Matrix Variant Filter* action from the context mennu of the Matrix Editor. The second possibility to create such a filter is to select a number of VDMs in the projects view and use the*Create Matrix Variant Filter* action from the context menu of the project view. A dialog comes up to define a name for the new filter.

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. The result of the evaluation will be visible in different ways depending on the type of the object the cell represents. A Restriction will show its evaluation state. A not evaluated Restriction will be shown as , whereas a possitively or negatively evaluated Restriction will show , or respectively. A Constraint will always show a , since it will produce an error, if the condition is not met. If no value for an attribute can be calculated, a is shown in the corresonding cell to indicate that the attribute has no value at all under the current configuration.

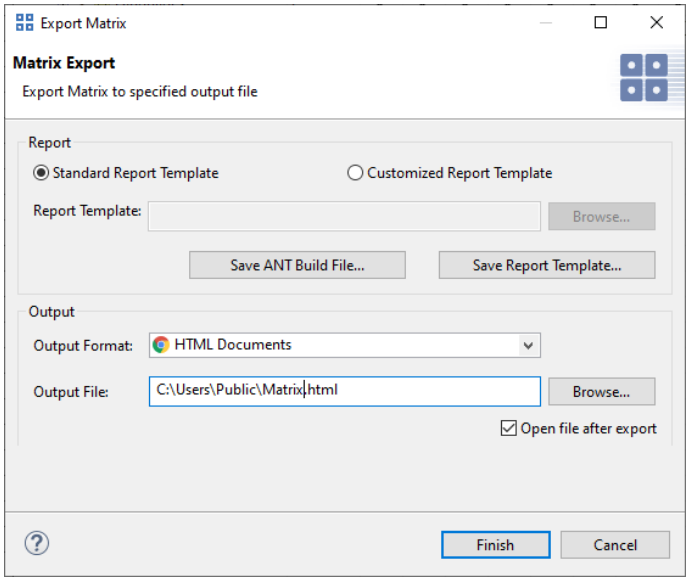
Lastly it is even possible to perform transformation of the visible VDMs. Use the *Transform all models* button to perform transformation. See Section 5.9, “ Variant Transformation ” for detailed information.

The Matrix shown in the editor can be exported to various output formats using the *Export Matrix...* action from the context menu. The dialog, which opens, allows the user to chose the output format and location for the export.

Note

The *Export Matrix...* action is available only, if the *pure::variants - Connector for Reporting with BIRT* is installed.

Figure 7.25. Export Matrix Dialog



The action export the visible content of the matrix editor, it does even take the expansion state of a element into acount. Meaning collapsed elements and attributes will not be visible in the export result. As well as filtered elements and varaints not opened in the editor.

In the dialog a custom report template can be selected. As a starting point we recommend to use the stadard report template, which can be saved with the *Save Report Template...*. After customizing the report template, it can be used for future matrix exports.

> **Note**
>
> The template contains a table, which is named "Matrix". This table is the entry point for the matrix exporter. The table can be modified, but there has to be a variant column, which defines the the layout for the columns inserted by the exporter. Which column is the variant column is defined with a user property on the table. Name of the property is "VariantColumn", type is integer. The value is the index of the variant column. The index is 0 based, so third column has index 2. This column is replaced by the exporter with the necessary variant columns.

The *Save ANT Build File...* button generates an ANT Build file, which can be used to run the report generation in head less mode automatically by any build system. See Section 6.14, “External Build Support (Ant Tasks)”

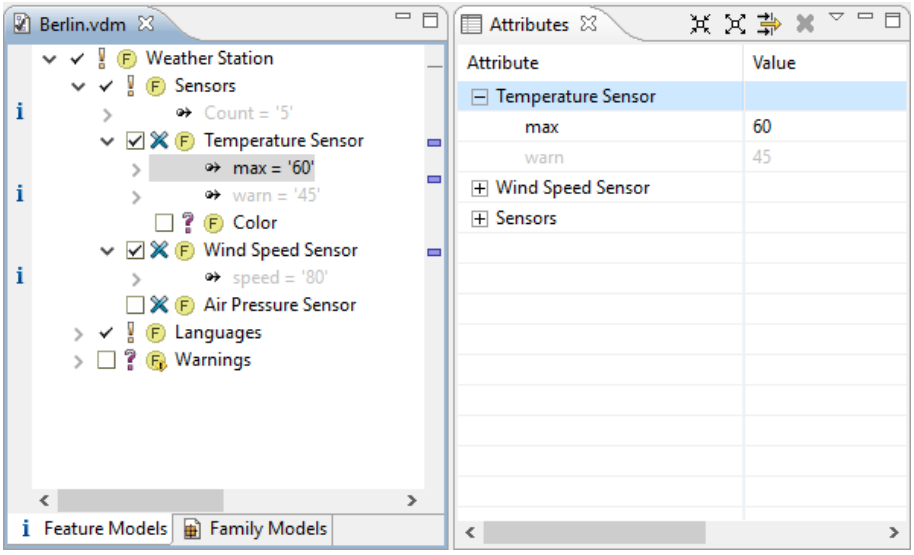
The generated build file uses the same tree and table layout like currently configured in the confguration space editor, including model item visibility and expansion state for the shown model data. Elements not visible due to an applied filter will not be visible in the report if you run the generated ANT build file.

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.

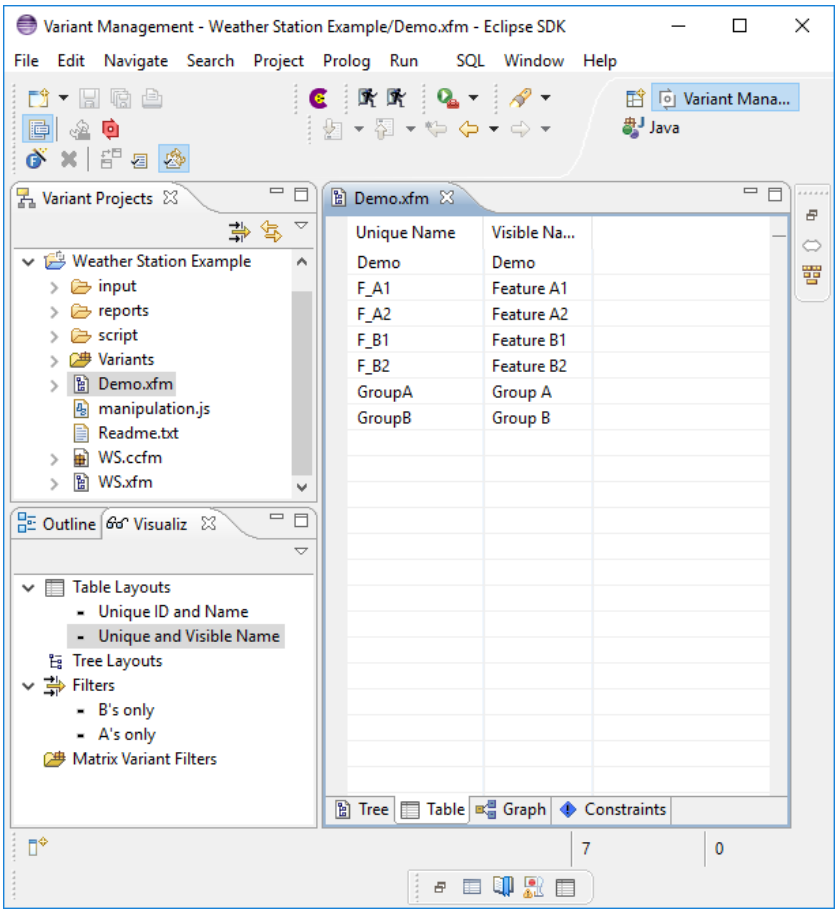
Figure 7.26. Attributes view (right) showing the attributes for the VDM



7.4.2. Visualization View

The model editors and most of the views support named layouts and filters. The Visualization view shows all named layouts and named filters defined in the current Eclipse workspace (see Figure 7.27, “Visualization view (left) showing 2 named filters and 2 named layouts”).

Figure 7.27. Visualization view (left) showing 2 named filters and 2 named layouts



When the Visualization view is opened, the first level of layouts and filters is expanded. To expand or collapse the visualizations manually use the "Expand.." and "Collapse.." buttons in the tool bar of the view. Additional filters and layouts may be imported from a file by choosing "Import" from the context menu. To export all visualizations listed in the Visualization view choose "Export" from the context menu. Exported visualizations are stored in a file which can be imported into another Eclipse installation or shared in the project's team repository. Visualizations can be applied either by double clicking on the name of the visualization or by choosing "Apply Item" from the context menu of a visualization. Other actions on visualizations are Delete and Rename by choosing the corresponding context menu entries.

Three top-level categories are available in the visualization view. These are *Filters* , *Table Layouts* and *Tree Layouts* . The corresponding items can only be created in the editors. See the section called “ Table Editing Page ” , the section called “ Tree Editing Page ” and Section 6.9, “ Filtering Models ” for information on it. Tree Layouts can only be applied to Editors Tree Viewers, Table Layouts to Editors Table Viewers and Filters to all pure::variants 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.

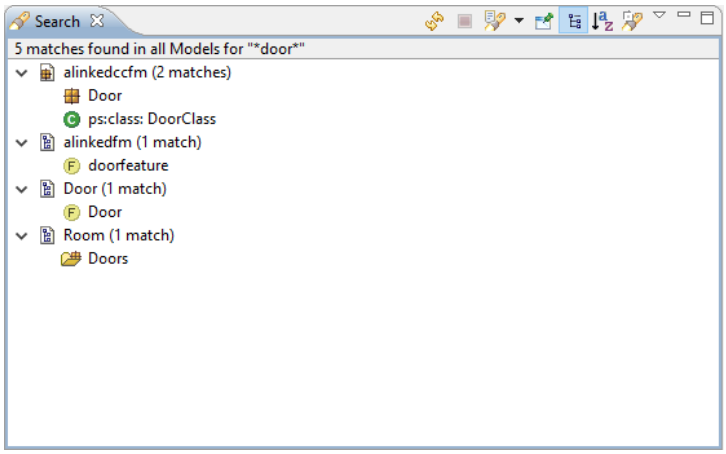
Additionally the layout and filter items may be organized within categories. Layouts or filters, created once appear at first directly below their top-level category. The view allows to create a category by choosing "Create Category..." from the context menu on a parent Category. The context menu provides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

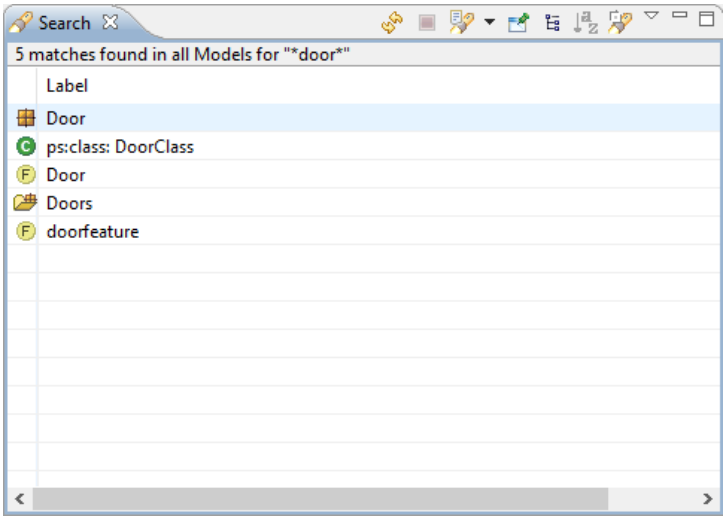
Figure 7.28. Variant Search View (Tree)



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 seach 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 7.29. Variant Search View (Table)



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 7.30. Properties view for a feature

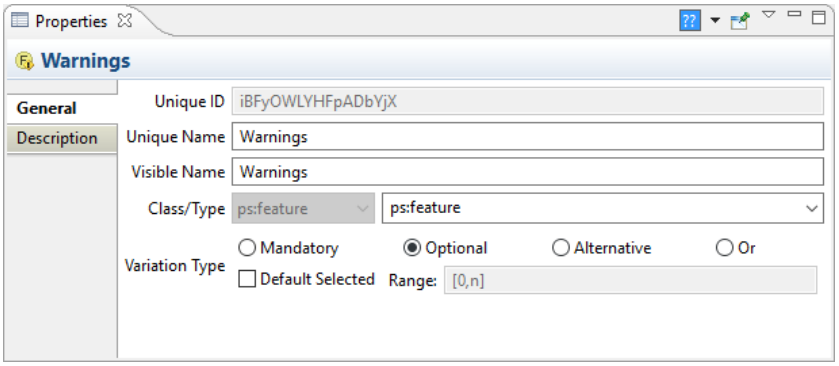
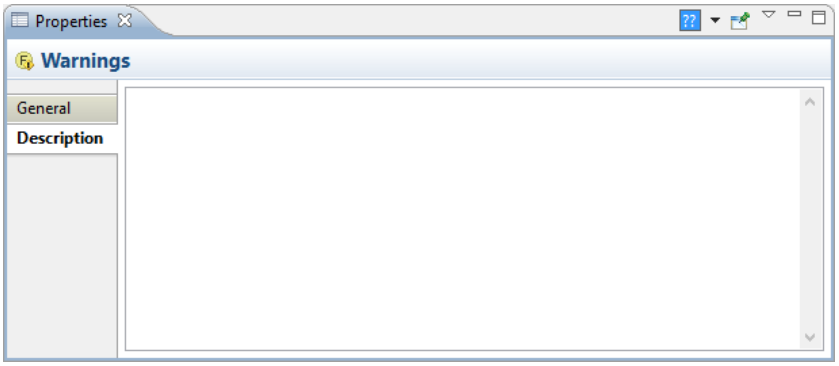


Figure 7.30, “Properties view for a feature” shows the properties view after a feature was selected in the Feature Model Editor. At the left side there are selectable tabs, each containing a set of properties that logically belong together. Usually, tabs*General* and *Description* are shown. The middle 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 7.31, “Description tab in Properties view for a relation”).

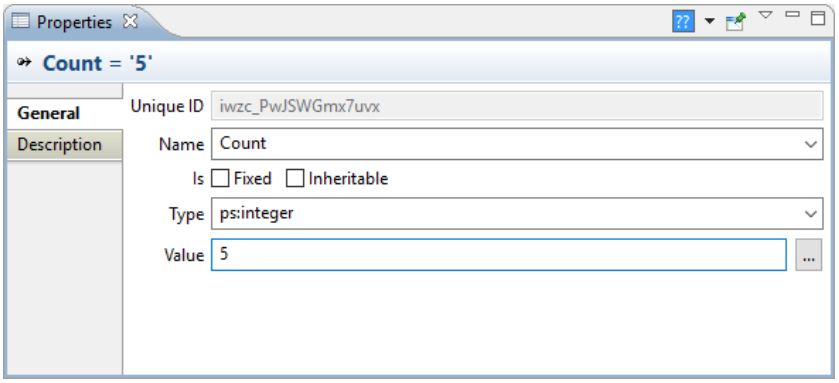
Figure 7.31. Description tab in Properties view for a relation



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 7.32. 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 6.12, “ Using Multiple Languages in Models ” for more information about language support. For formatted text editing of descriptions seeSection 7.5.1, “ Common Properties Page ” .

7.4.7. Relations View

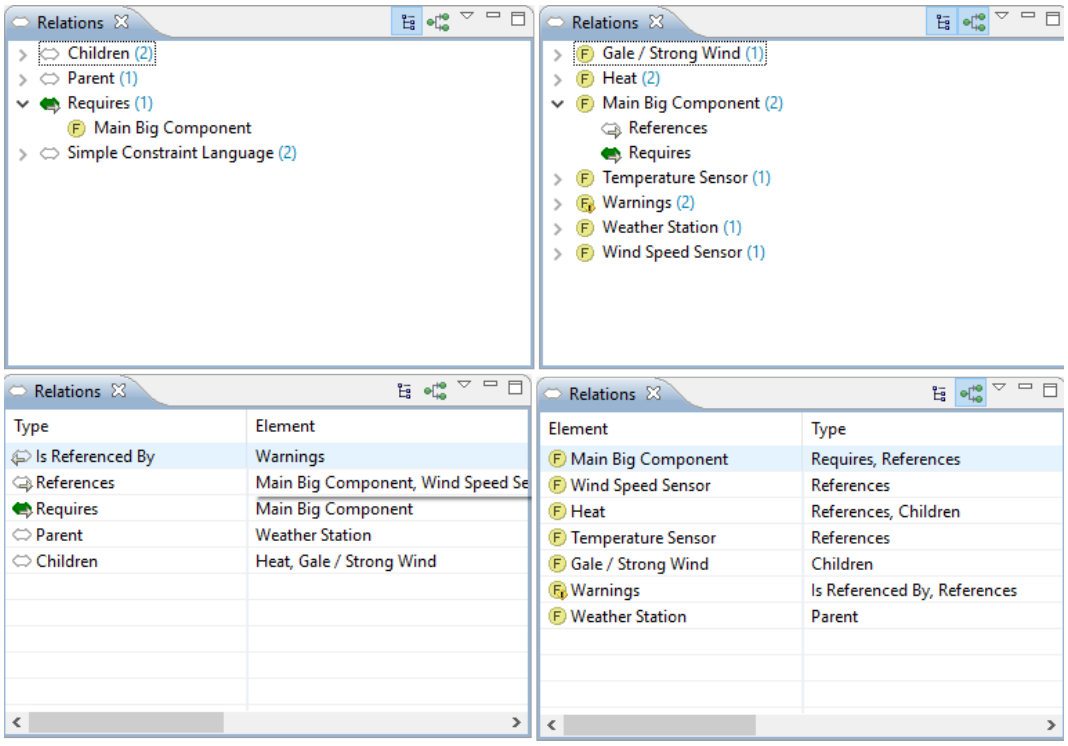
The Relations view shows the relations of the currently selected element (feature/component/part/source element) to other elements. The relations shown in the view are gathered from different locations. The basic locations are:

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, pvSCL, the relations view shows the referenced elements below the entry "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, pvSCL, the relations view shows the referenced elements below the entry "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 7.33. 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. In full configuration mode, it lists all selected Feature and Family Model elements representing the given variant. In partial configuration mode, it lists both selected and open Feature and Family Model elements of the given variant.

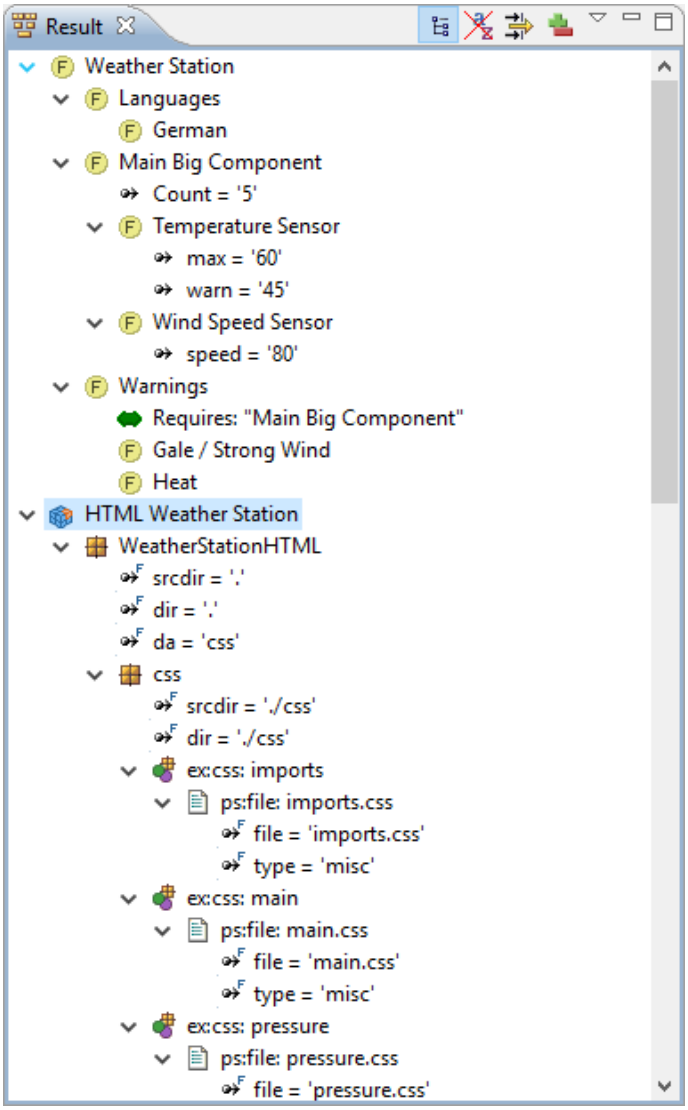
The result view also provides a special operation mode where, instead of a result, the difference (delta) between two results are shown, similar to the model compare capability for Feature and Family Models.

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 7.34. Result View

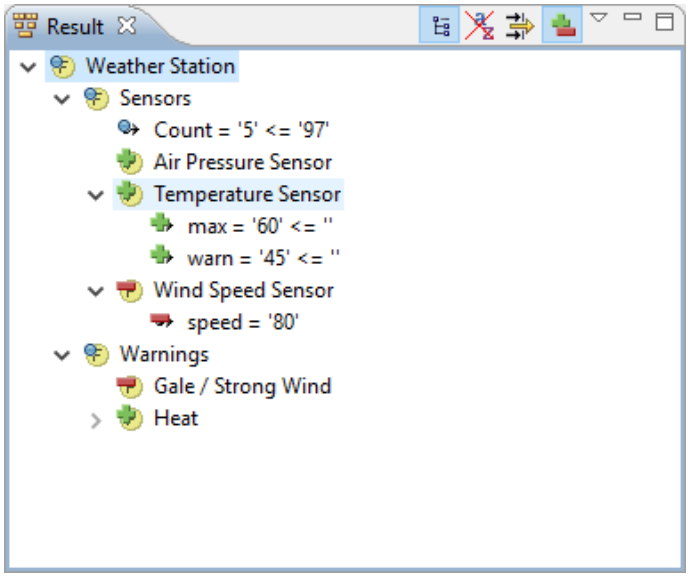


Result Delta Mode

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

Figure 7.35. 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).

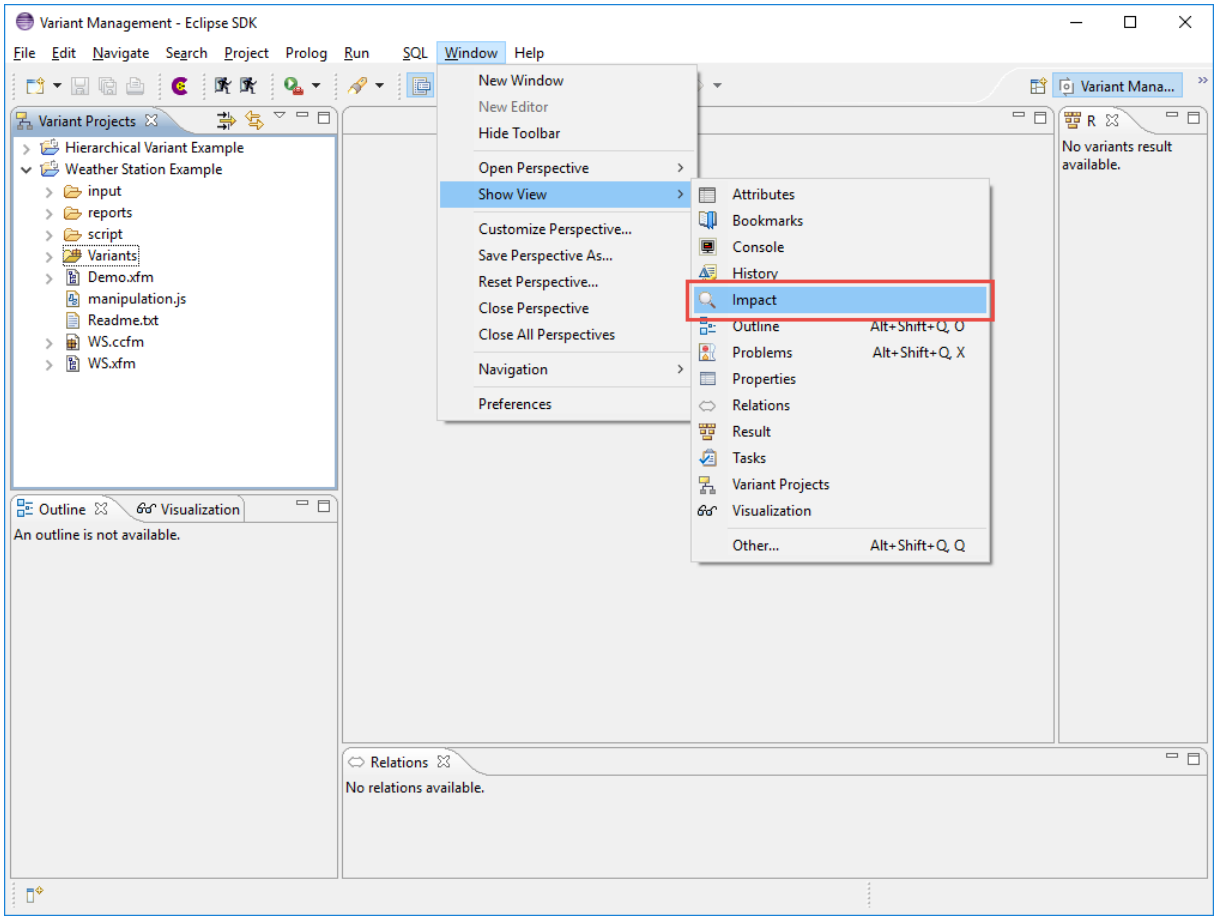
7.4.9. Impact View

Variant description models are used to configure variation points in pure::variants. These vdms are connected to a configuration space, which lists all input models. Variation points can be either feature-based or manually configured. The feature-based configuration is used to automatically configure variation points based on selections of features. The feature-based configuration depends directly on the inout models of the configuration space. Changing the input models may have an impact on the existing variation point configurations.

The impact view shows possible impacts on the variant configurations while changing the input models.

To open the **Impact View** use the *Show View -> Impact View* action form the *Window* menu.

Figure 7.36. Open Impact View



After opening the view work on the input models can be started. The impact analysis is disabled by default and needs to be explicitly enabled for each input model, which shall be considered by the impact analysis. To enable the analysis open an input model and click on the *Enable Imput Analysis* button ( ). The initial analysis is performed now for all variant models, which use the input model. The impact view shows the state of each variant model after the analysis is done. There are 5 different states:

- the variant is currently analysed by the impact analysis
- the variant is valid and not changed by the input model changes
- the variant is valid but changed by the input model changes
- the variant is invalid
- the variant is deactivated

The impact is calculated automatically for every change on the input models, for which the impact analysis is activated. With the button the automatic calculation can be paused, if the user is performing a lot of changes and can be resumed after the changes are done. Reseting the impact analysis is triggered with the button. This removes the current analysis result and starts a fresh calculation of the impact. The result is the same as enabling the impact claculation the first time on the current state of the input models.

Figure 7.37. Impact Calculation Result

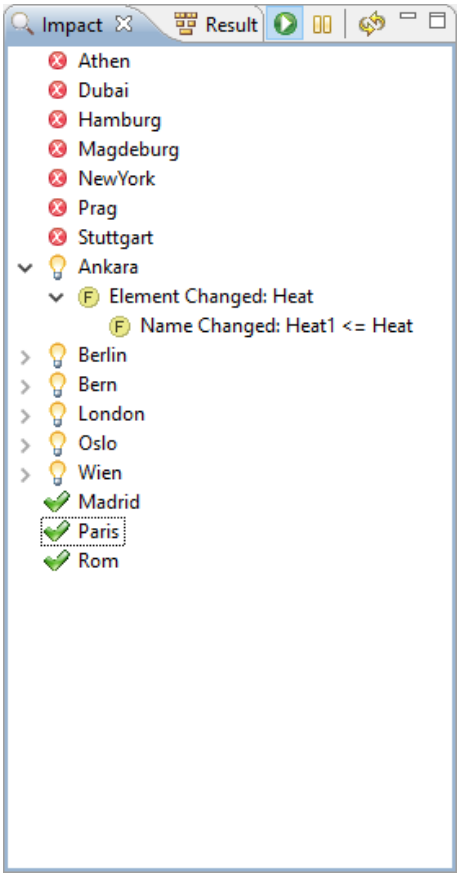
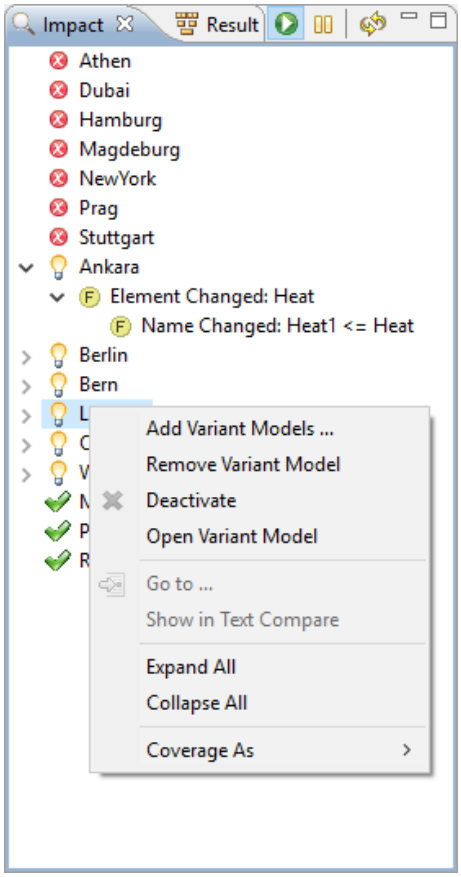


Figure Figure 7.37, “Impact Calculation Result” shows an example result for the impact analysis. If there are problems the result contains details on the problem. The same applies to changes in the variant description models. For changes it is possible to see more details with the action*Show in Text Compare* from the context menu of one change.

The context menu of the impact view allows the user to change the scope of the impact analysis. The Actions in sub menu *Add Variant Models* allows the user to add additional variant models to the analysis. *Manually From Workspace...* lets the user chose the variants to be added to impact analysis from all variant models currently imported to the workspace.*Automatically All Related From Workspace* does check all variant models currently imported to the workspace and adds all variants using the input model currently used in the impact view. *Related from Server Projects* lets the user decide which variants to import from all related server projects.

Remove Variant Model removes the selected models from the impact analysis. *Deactivate* deactivates the analysis for the selected variant models. This action keeps them in the impact view and just ignores them during calculation.

Figure 7.38. Impact View Context Menu



Navigation from the information shown in the impact view to the corresponding elements or models is enabled using the *Go to* ... action from the context menu or by just double clicking the elements or models in the impact view.

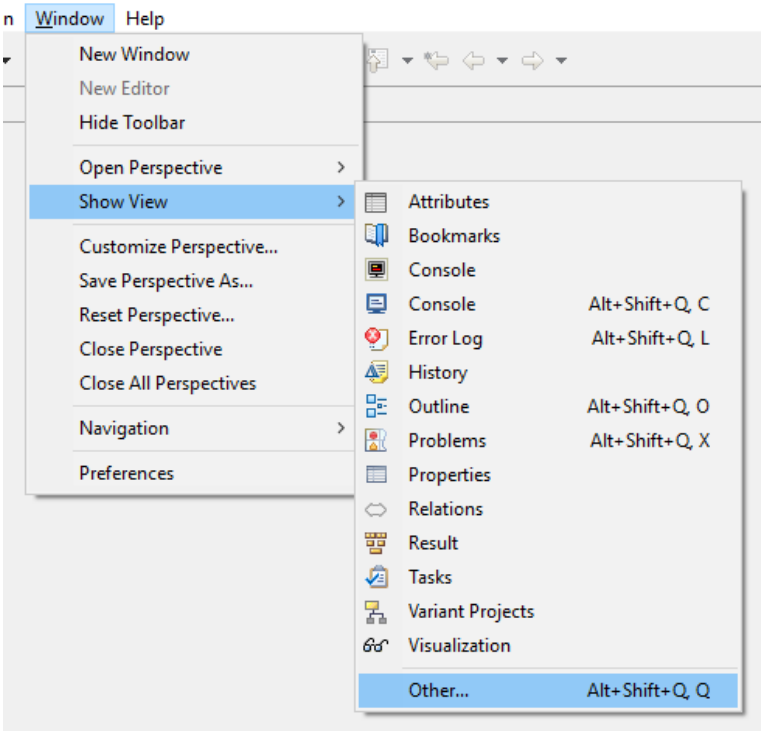
7.4.10. pvSCL IDE

Writing complex pvSCL rules in the modal Code Editor dialog is notvery comfortable since it is not possible to look at your feature models until you closed the dialog. To avoid that you can use the pvSCL IDE view to prepare the pvSCL rules and than just copy them to the Code Editor after you have finished them.

Essentially, the pvSCL IDE is a live expression evaluator which can be used to successfully developed large and complex expressions with it.

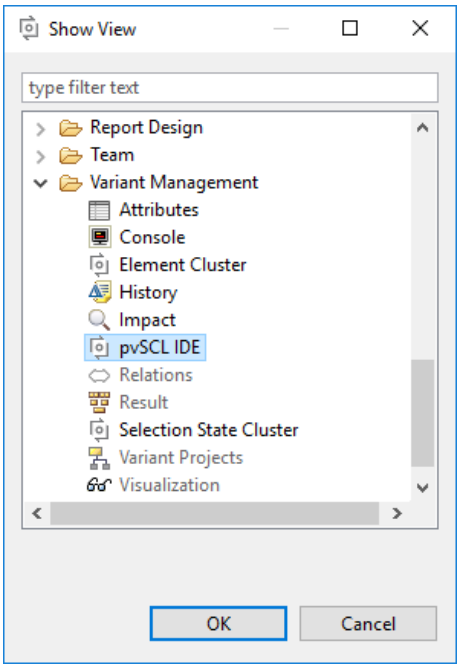
It is used in three steps.

Figure 7.39. Open pvSCL IDE View



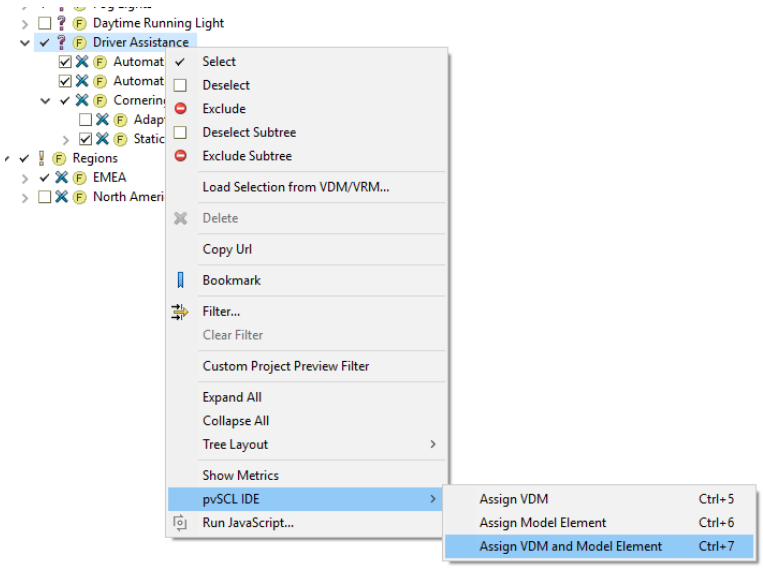
Step 1: Open the pvSCL IDE view. Go to *Window -> Show View -> Other* and chose *pvSCL IDE* in the opening dialog. After ending this dialog the pvSCL IDE view opens.

Figure 7.40. Open pvSCL IDE View



Step 2: Select an element as expression context in a variant model. This initializes the evaluation ontext for the pvSCL IDE. This usually should be the element, which will be the parent of the constraint, restriction or calculation.

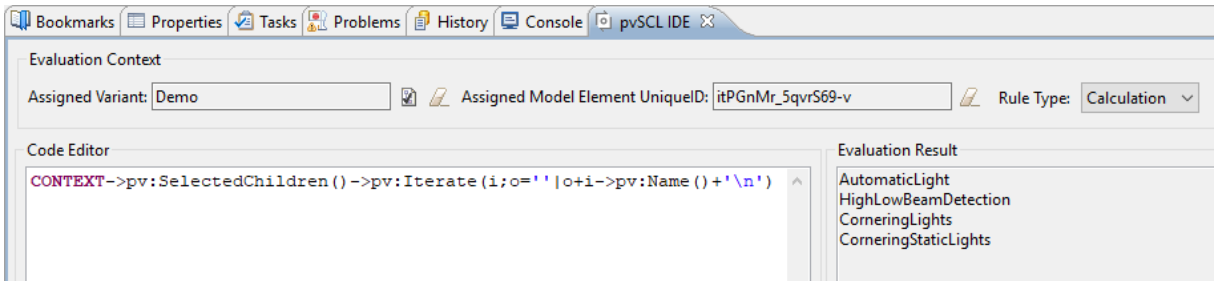
Figure 7.41. Assign context element to pvSCL IDE



To do so, right-click on the element, select pvSCL IDE --> Assign VDM and Model Element. Alternatively you can press [Ctrl]+[7].

Step 3: Enter your expression in the Code Editor part of the pvSCL IDE.

Figure 7.42. The pvSCL IDE View



Enter the text of your pvSCL expression. You can use auto-completion using [Ctrl]+[Space], as usual. You also have on-the-fly syntax and error highlighting. The expression in the Code Editor is evaluated on the variant model immediately as you type. The result appears on the right side instantly.

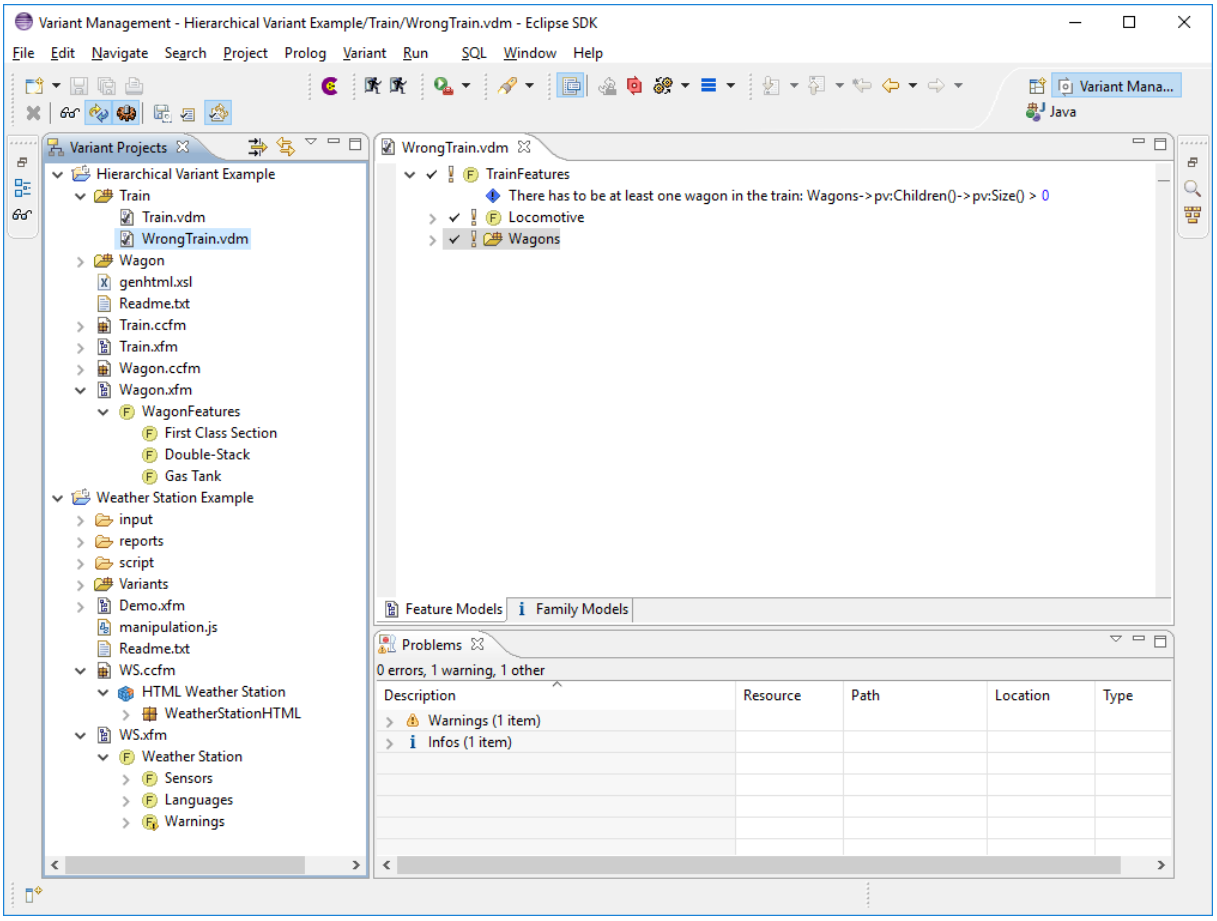
Make sure you use the rigth rule type setting. The rule type constraint and restriction have a result of type *ps:boolean* only. Calculations on the pther hande can also have results of other types.

If the evaluation of your expression would generate error, warning or information marker the Evaluation Result control will indicate that with a small marker at the top left corner of the control.

7.4.11. Variant Projects View

The Variant Projects View (upper left part in Figure 7.43, “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. InFigure 7.43, “The Variant Projects View” one editor is shown for a variant description model with some features selected.

Figure 7.43. The Variant Projects View



7.5. Model Properties

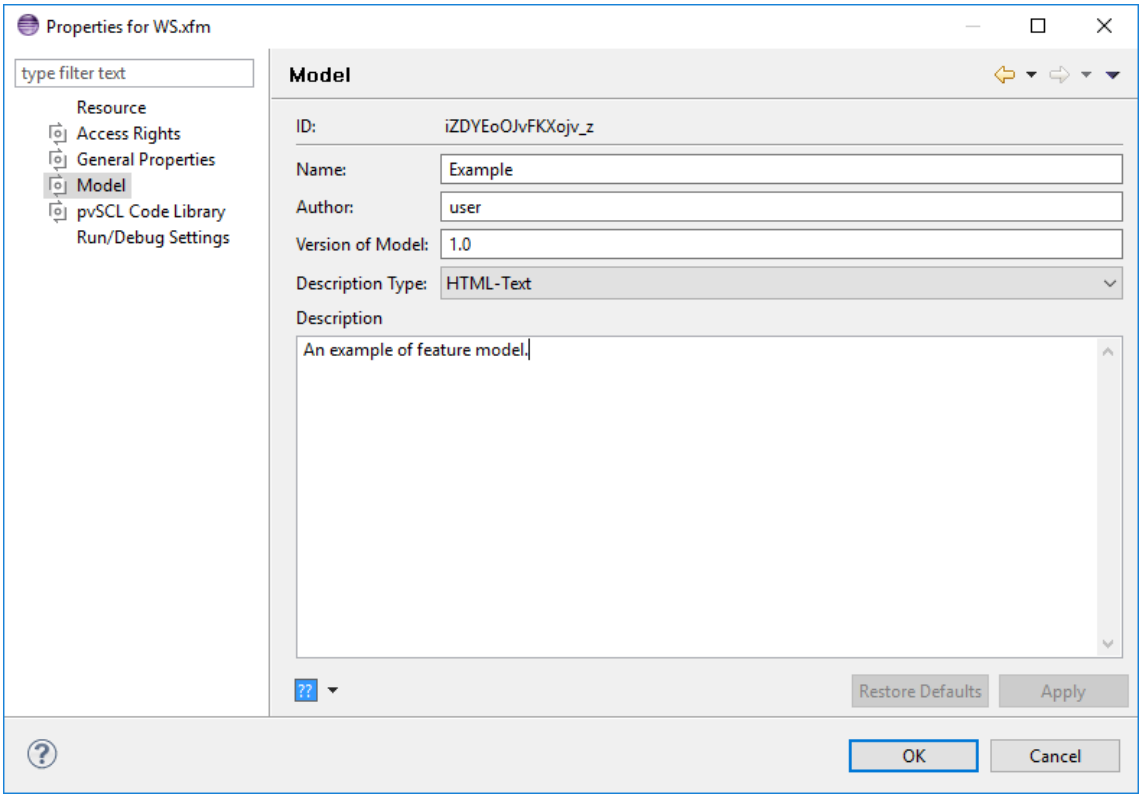
pure::variants models have a set of properties. Each model has at least a name. Optionally it can have an author, version, description, and a set of custom properties. Model properties are set by right-clicking on a model in the **Variant Projects** view and choosing **Properties** from the context menu. 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 7.44, “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 7.44, “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.

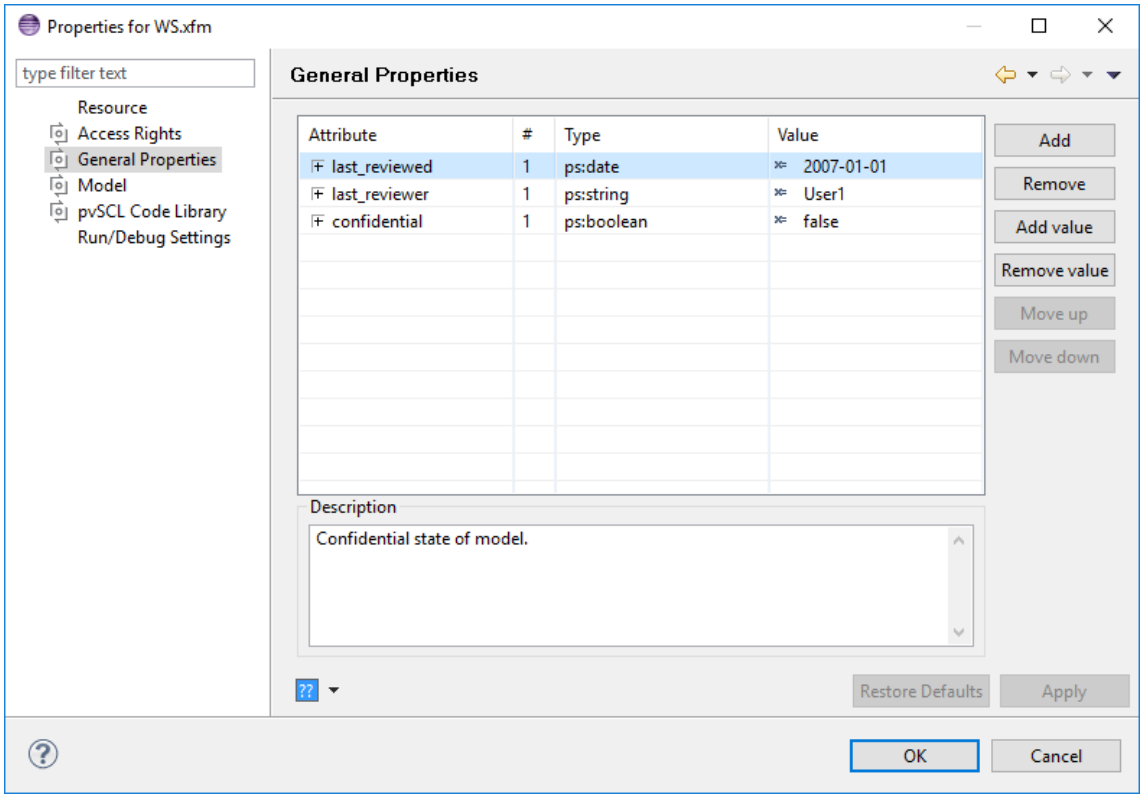
Figure 7.44. Feature Model Properties Page



7.5.2. General Properties Page

Custom model properties are defined on the **General Properties** page (see Figure 7.45, “General Model Properties Page”).

Figure 7.45. General Model Properties Page



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** .

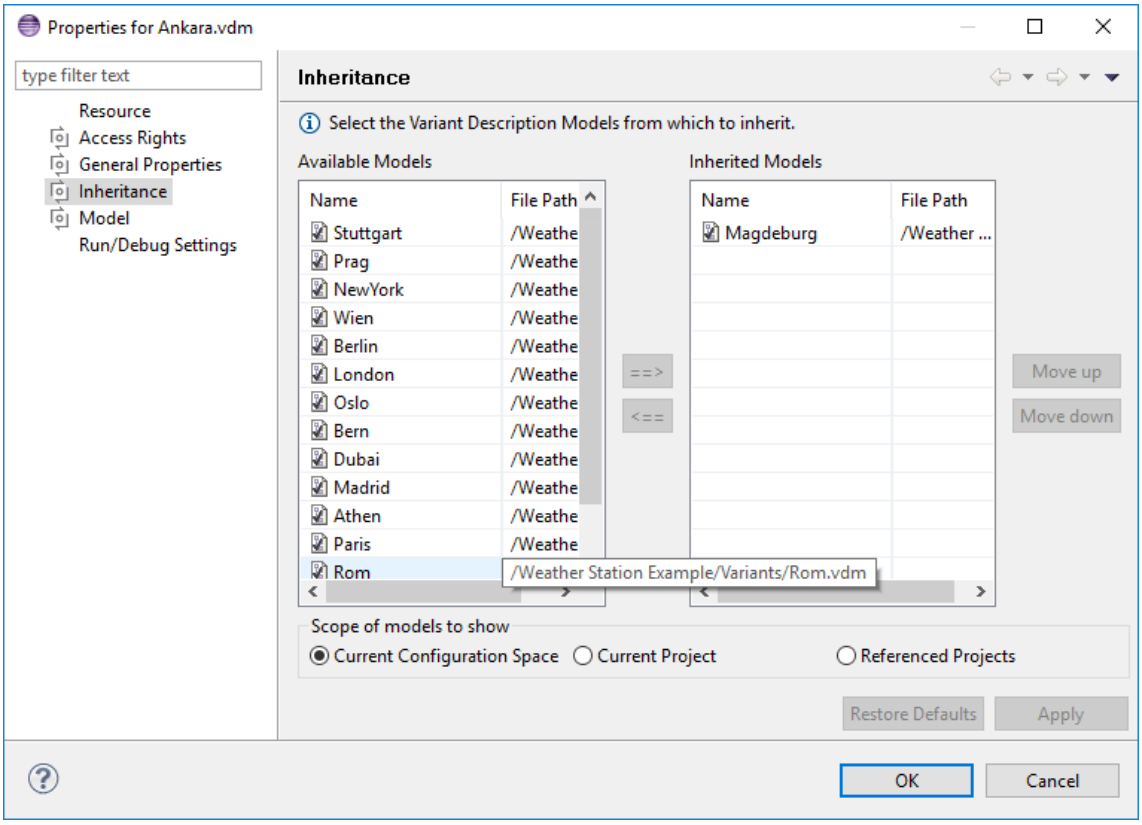
As for element attributes, model properties can also have a list type by simply adding square brackets ("[]") to the type name, e.g. *ps:string[]* , *ps:integer[]* .

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 (seeFigure 7.46, “Variant Description Model Inheritance Page”).

Figure 7.46. 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 Extensions

The features offered by pure::variants may be further extended by the incorporation of additional software extensions. An extensions may just contribute to the Graphical User Interface or it may extend or provide other functionality. For instance an extensions could add a new editor tab for model editors or a new view. The online version of this user guide contains documentation for additional extensions. Printable documentation for the additional extensions is distributed with the extensions and can be accessed from the online documentation via a hyperlink.

Currently available extensions can be found on our web site (https://www.pure-systems.com/purevariants/pure-variants-connectors)

8.1. Installation of Additional pure::variants Extensions

Additional pure::variants extensions 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 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 Parametric Technology web site (https://www.pure-systems.com) or read your registration e-mail to find out which site is relevant for the version of the software your are using. Open the page in your browser to get information on how to use update sites with Eclipse.

- *Archived Update Site* pure::variants uses also 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 Parametric Technology web site (https://www.pure-systems.com) or read your registration e-mail to find out which site is relevant for the version of the software your are using. Open the page in your browser to get additional information on how to use update sites with Eclipse. 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
ps:boolean	boolean value	`true` and `false`
ps:integer	integer number	a valid integer number of format `('0x' [0-9a-fA-F]+) | ([+-]? [0-9]+)`
ps:float	floating point number	a valid floating point number of format `[+-]? [0-9]+ ('.' [0-9]+)? ([eE] [+-]? [0-9]+)?`
ps:string	any kind of unspecific text	any
ps:path	path to a file in a file system	any
ps:directory	path to a directory in a file system	any
ps:url	a URL or URI	any
ps:html	HTML code	any
ps:datetime	date and time (e.g. in IS0 8601 format)	any
ps:version	a version string (with wildcards)	a string of format `[0-9]+ ('.' [*0-9]+ ('.' [*0-9]+ ('.' [*0-9a-zA-Z_-]+)?)?)?`
ps:filetype	file type identifier	`def` , `impl` , `misc` , `app` , `undefined`
ps:insertionmode	value type of source element type *ps:fragment*	`before` and `after`
ps:element	feature or family model element reference	a valid ID of an element
ps:feature	feature reference	a valid ID of a feature
ps:class	*ps:class* source element reference	a valid ID of a *ps:class* 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, showing the supported element relations, the defining element D is the element on which the relation is defined, and EL is the list of related elements $E_1 ... E_n$.

Note

Users can use their own custom relation types, which are ignored during evaluation. Some of the supported and thus evaluated relation types exist only since a specific release version of pure::variants. In previous versions they are treated as custom relations and are not evaluated at all. So, using models containing such relations in previous versions of pure::variants, which does not support them yet, can lead to invalid variant configurations. To avoid this source of error, the user is responsible to ensure that these models are not used in previous versions. One possibility for that is to add a pvSCL version guard constraint in each feature or family model, which uses such a new relation. See the section called “pv:PVVersion()” for that.

Table 9.2. Supported relations between elements (I)

Relation	Description	Logical equivalent
ps:requires(EL)	At least one element in *EL* has to be selected if the defining element is selected.	*D* implies (E_1 or ... or E_n)
ps:requiresAll(EL)	All elements in *EL* have to be selected if the defining element is selected.	*D* implies (E_1 and ... and E_n)
ps:requiredFor(EL)	If at least one element in *EL* is selected, then the defining element has to be selected.	(E_1 or ... or E_n) implies *D*
ps:requiredForAll(EL)	If all elements in *EL* are selected, then the defining element has to be selected.	(E_1 and ... and E_n) implies *D*
ps:conditionalRequires(EL)	Similar to *ps:requires* , but the relation is considered only for elements whose parent element is selected.	*D* implies ((parentSel(E_1) implies E_1) or ... or (parentSel(E_n) implies E_n)) , where parentSel(*root*) = *true*
ps:recommends(EL)	Like *ps:requires* , but not treated as error if not complied.	*D* implies (E_1 or ... or E_n)
ps:recommendsAll(EL)	Like *ps:requiresAll* , but not treated as error if not complied.	*D* implies (E_1 and ... and E_n)
ps:recommendedFor(EL)	Like *ps:requiredFor* , but not treated as error if not complied.	(E_1 or ... or E_n) implies *D*
ps:recommendedForAll(EL)	Like *ps:requiredForAll* , but not treated as error if not complied.	(E_1 and ... and E_n) implies *D*
ps:equalsAny(EL) (available since pure::variants 4.0.7)	If the defining element is selected, at least one element in *EL* has to be selected. If the defining element is not selected, none of the elements in*EL* may be selected.	*D* equals (E_1 or ... or E_n)
ps:equalsAll(EL) (available since pure::variants 4.0.7)	If the defining element is selected, all elements in *EL* have to be selected. If the defining element is not selected, not all of the elements in*EL* may be selected.	*D* equals (E_1 and ... and E_n)
ps:conflicts(EL)	If all element in EL are selected, then the defining element must not be selected.	(E_1 and ... and E_n) implies not(*D*)
ps:conflictsAny(EL)	If any element in EL is selected, then the defining element must not be selected.	(E_1 or ... or E_n) implies not(*D*)
ps:discourages(EL)	Like *ps:conflicts* , but not treated as error if not complied.	(E_1 and ... and E_n) implies not(*D*)
ps:discouragesAny(EL)	Like *ps:conflictsAny* , but not treated as error if not complied.	(E_1 or ... or E_n) implies not(*D*)
ps:influences(EL)	The elements in *EL* are influenced in some way by the selection of the defining element. The interpretation of the influence is up to the user.	
ps:provides(EL)	The "inverse" relation to *ps:requires* . For all selected elements in *EL* at least one defining element has to be selected.	*E* implies (D_1 or ... or D_n)

Relation	Description	Logical equivalent
ps:supports(EL)	Like *ps:provides* , but not treated as error if not complied.	*E* implies (D_1 or ... or D_n)

9.3. Element Variation Types

Table 9.3. Element variation types and its icons

Short name	Variation Type	Description	Icon
mandatory	*ps:mandatory*	A mandatory element is automatically selected if its parent element is selected.	
optional	*ps:optional*	Optional elements are selected independently.	
alternative	*ps:alternative*	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). pure::variants allows only one *ps:alternative* group for the same parent element.	
or	*ps:or*	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). pure::variants allows only one *ps:or* group for the same parent element.	

9.4. Element Selection Types

Table 9.4. 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.	
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 9.5. Predefined source element types

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

Source Type	Description	Icon
ps:pvsclxml	Maps directly to an XML document containing variation points (conditional parts) using pvSCL expressions.	
ps:pvscltext	Maps directly to a text document containing variation points (conditional parts) using pvSCL expressions.	
ps:flagfile	Represents a file that can hold flags such as a C/C++ header file containing preprocessor defines.	
ps:makefile	Represents a make (build) file such as GNU make files containing make file variables.	
ps:classaliasfile	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.	
ps:symlink	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 fromps: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>` .

9.5.1. 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 copied. The optional attribute `srcdir` ist used for directories that are located in a different place in the source hierarchy and/or have a different name.

9.5.2. ps:file

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

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 optional attribute `srcurl` is used to specify a source directory with an url. This supports basic authentification. If this srcurl property is set srcdir is ignored. The standard transformation supports the protocls http and https.

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 *ps:classalias* calculations this information is used to determine the include files required for a given class.

Type	Description
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.

9.5.3. ps:fragment

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

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. If taken from a string, attribute `encoding` can be used to specify the character encoding of the string content. 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` .

9.5.4. ps:pvsclxml

Attributes:
dir *[ps:directory]*
file *[ps:path]*
type *[ps:filetype]*
srcdir? *[ps:directory]*
srcfile? *[ps:path]*
conditionname? *[ps:string]*
copycondition? *[ps:boolean]*
valuesubstitution? *[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 *pv:condition* . If the attribute `copycondition` is not set to*false* , the condition attribute is copied into the target document as well. If the attribute`valuesubstitution` is set to *true* , the content of all attribute values of the XML document will be handled asSection 9.5.5, “ ps:pvscltext ” .

The condition itself has to be a valid pvSCL expression. For details on writing pvSCL expressions, see Section 9.7, “Expression Language pvSCL” .

In the example document given below after processing with an *ps:pvsclxml* 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:pvsclxml transformation

```
<?xml version='1.0'?>
<text xmlns:pv="http://www.pure-systems.com/purevariants">
  <chapter pv:condition="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 pvSCL expression. The name of this node is `pv:eval` (namespace "pv" is defined as "http://www.pure-systems.com/purevariants"). The`pv:eval` node is replaced by the calculated value in the result document.

Example 9.2. Example use of pv:eval

```
Source document:

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

Result document:

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

9.5.5. ps:pvscltext

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

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. If no text document encoding is given, then UTF-8 encoding is assumed.

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

Statement	Description
`PVSCL:IFCOND(` *condition* `)`	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 `PVSCL:ENDCOND` call.
`PVSCL:ELSEIFCOND(` *condition* `)`	This macro can be used after a `PVSCL:IFCOND` or `PVSCL:ELSEIFCOND` call. If the condition of the preceding `PVSCL:IFCOND` or `PVSCL:ELSEIFCOND` is failed, the condition of this `PVSCL:ELSEIFCOND` is checked. If it evaluates to true, the enclosed text is included in the resulting text output.
`PVSCL:ELSECOND`	This macro can be used after a `PVSCL:IFCOND` or `PVSCL:ELSEIFCOND` call. If the condition of the preceding `PVSCL:IFCOND` or `PVSCL:ELSEIFCOND` is failed, the enclosed text is included in the resulting text output.
`PVSCL:ENDCOND`	Close a conditional text block. This macro is allowed after a `PVSCL:IFCOND` , `PVSCL:ELSEIFCOND` , or `PVSCL:ENDCOND` call.
`PVSCL:EVAL(` *expression* `)`	Evaluate the given pvSCL expression and insert the value of the expression into the result document.

These statements can occur everywhere in the text document and are directly matched, i.e. independently of the surrounding text. The conditions of `PVSCL:IFCOND` and `PVSCL:ELSEIFCOND` and the expression of`PVSCL:EVAL` are the same as the conditions described for source element type*ps:pvsclxml* (see Section 9.5.4, “ ps:pvsclxml ” for details), except for a list of comma-separated flags that can follow the pvSCL code. Following flags are supported.

Flag	Description
`LINE`	Clear or remove the line containing the pvSCL conditional text statement.

Flag	Description

	Example for a multi-line if-statement utilizing flag `LINE` :
	`/* PVSCL:IFCOND(Temperature,LINE) */`
	`initializeSensor("temperature",PVSCL:EVAL(Temperature->max));`
	`/* PVSCL:ELSECOND */`
	`disableSensor("temperature");`
	`/* PVSCL:ENDCOND */`
	Result if feature Temperature is selected:
	`initializeSensor("temperature",60);`
	Example for a single-line if-statement utilizing flag `LINE` :
	`//PVSCL:IFCOND(WindSpeed,LINE)updateSensor("wind");PVSCL:ENDCOND`
	Result if feature WindSpeed is selected:
	`updateSensor("wind");`

`BLANKS`	Clear the line containing the pvSCL conditional text statement. In contrast to the flag `LINE` does `BLANKS` replace each character with a whitespace. This ensures the location of the parts remaining in the file is the same as in the input document.
	Example for a multi-line if-statement utilizing flag `BLANKS` :
	`/* PVSCL:IFCOND(Temperature,BLANKS) */`
	`initializeSensor("temperature",PVSCL:EVAL(Temperature->max));`
	`/* PVSCL:ELSECOND */`
	`disableSensor("temperature");`
	`/* PVSCL:ENDCOND */`
	Result if feature Temperature is selected:
	`/*                                  */`
	`initializeSensor("temperature",60);`
	`/*`
	`                */`
	Example for a single-line if-statement utilizing flag `BLANKS` :
	`//PVSCL:IFCOND(WindSpeed,BLANKS)updateSensor("wind");PVSCL:ENDCOND`
	Result if feature WindSpeed is selected:
	`//                              updateSensor("wind");`

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

In the example document given below after processing with an *ps:pvscltext* 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:pvscltext transformation

```
PVSCL:IFCOND(WithIntroduction)
    This text is in the resulting text output
    if feature WithIntroduction is selected.
PVSCL:ELSECOND
```

```
    This text is in the resulting text output
    if feature WithIntroduction is not selected.
PVSCL:ENDCOND
    This text is always in the resulting text output.
```

9.5.6. ps:flagfile

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

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
```

9.5.7. ps:makefile

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

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"
```

9.5.8. ps:classaliasfile

Attributes:
dir *[ps:directory]*
file *[ps:path]*
type *[ps:filetype]*
alias *[ps:string]*
includebasedir? *[ps:directory]*
encoding? *[ps:encoding]*

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 className alias;` statement is generated by the standard transformation for this element type.`className` is the name of the class referenced by the parent *ps:classalias* part element. Furthermore, in the standard transformation `#include` directives are generated for each of the referenced class' *ps:file* source elements that have a *type* attribute with the value 'def'. The optional attribute `includebasedir` defines how the `#include` directives referencing the class header files will be generated. If this attribute is missing or it has an empty value, the generated `#include` directives will reference the class header file by absolute file paths. Otherwise the value will be used as the base directory path. In that case the generated `#include` directives will reference the class header files by a file paths relative to that base directory. 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::PCConn" with aliased class "NoConn"

```
#ifndef __PCConn_include__
#define __PCConn_include__
#include "C:/Weather Station Example/output/usr/wm-src/NoConn.h"
namespace io {
namespace net {
typedef NoConn PCConn;
}
}
#endif __PCConn_include__
```

Example 9.7. Generated code for a ps:classalias for alias "io::net::PCConn" with aliased class "NoConn" with includebasedir set to "usr/wm-src"

```
#ifndef __PCConn_include__
#define __PCConn_include__
#include "NoConn.h"
namespace io {
namespace net {
typedef NoConn PCConn;
}
}
#endif __PCConn_include__
```

9.5.9. 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 9.6. Predefined part types

Part type	Description	Icon
ps:class	Maps directly to a class in an object-oriented programming language.	

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

9.6.1. ps:classalias

Attributes: **`class`** *[ps:class]*
`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 class or class alias to be aliased can be either referenced by the attribute `class` or the attribute `Value`.

Using attribute `class` the class or class alias element is directly referenced. The referenced element has to be of part type`ps:class` or `ps:classalias`. Alternatively, using the attribute `Value` the class or class alias element can be referenced by its unique name.

For more information and an example see Section 9.5.8, “ ps:classaliasfile ” .

9.6.2. 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 9.5.8, “ ps:classaliasfile ” .

9.6.3. 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 9.5.6, “ ps:flagfile ” and Section 9.5.7, “ ps:makefile ” .

9.6.4. 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 9.5.6, “ ps:flagfile ” and Section 9.5.7, “ ps:makefile ” .

9.6.5. 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 evaluation tries to satisfy feature selections expected by `ps:feature` part elements. This functionality does not depend on the use of any specific transformation modules.

This type is deprecated since pure::variants version 7.0.0. Please contact pure::variants team for replacement options.

9.7. 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 language is based on a simple object model. An object has an identity, attributes (data) and functions which can be applied to it. Some functions can be used without an explicit object context. Objects represent either simple data items such as numbers, or collections of objects; or in many cases they represent pure::variants model items such as elements or models.

Both full and partial configuration mode is fully supported when evaluating pvSCL expressions. See also Section 5.8.2, “ Partial Evaluation ” for details about model evaluation in these modes. In partial evaluation, calculations are done also with a special *open* value. So, the result of a constraint, restriction, or calculation can be also *open*.

9.7.1. How to read this reference

The reference use the term *context* to denote the object to which an operator or function is applied to. This term is not to be confused with the keywords`context/CONTEXT`, which deliver a special object, see details below.

9.7.2. 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.*/
```

9.7.3. 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)
```

9.7.4. 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
```

9.7.5. 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
```

9.7.6. 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.

In partial evaluation, if the result of a calculation is a collection with at least one *open* member, instead of this incomplete collection only the *open* value will be returned.

Syntax

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

Examples

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

9.7.7. 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
```

9.7.8. 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

Path Name Element	Description
: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'
```

9.7.9. Element Selection State Check

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

To check the selection state of a given element, the meta-attribute *pv:Selected* can be called on that element. Depending on the configuration mode and selection state following values will be returned:

Selection state	Full evaluation	Partial evaluation
Selected	TRUE	TRUE
Excluded	FALSE	FALSE
Unselected	FALSE	*open*

Applying Boolean operations on element references enforce an implicit conversion to the Boolean selection state. So an explicit call of *pv:Selected* on element references is not necessary in following use cases:

- A constraint or restriction with a single element reference or a single expression resulting in an element reference
- 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
```

9.7.10. 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).

In full configuration mode, an error is created, if the accessed attribute has no value. In partial configuration mode, instead an *open* value is returned.

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.

Since meta-attributes (built-in and user-defined) and attributes use the same calling syntax, the calling precedence of meta-attribute and attribute calls needs to be considered:

- The built-in meta-attributes (see Section 9.7.23, “Function Library”) will override all attribute calls with the same name. However, it is generally not recommended to name attributes as same as built-in meta-attributes.
- A user-defined function in the meta-attribute syntax (see Section 9.7.17, “Function Definitions”) will override attribute calls with the same name and the same number of arguments counted in the meta-attribute syntax.

That is, a user-defined function with one argument (in the corresponding meta-attribute syntax called with zero arguments) will override an attribute call without arguments. Analogous, a user-defined function with two arguments (in the meta-attribute syntax called with one argument) will override an attribute call with index argument.

To access such hidden attributes, the meta-attribute *pv:Attribute* has to be used instead.

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'
```

9.7.11. 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.

In partial evaluation, logical operations are applied using *three-valued logic*. So, Boolean *open* values are supported as operands. The result can then be also *open*.

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
```

9.7.12. 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.

In partial evaluation, relation operations are applied using *three-valued logic*. So, Boolean *open* values are supported as operands. The result can then be also *open*.

Following relational operators are supported:

Operator	Meaning
REQUIRES	Evaluates to TRUE, iff a) both operands evaluate to TRUE or b) the left operand evaluates to FALSE. In the latter case, the right operand will not be evaluated.
IMPLIES	Same as REQUIRES.
CONFLICTS	Evaluates to TRUE, iff a) the left operand evaluates to TRUE and the right operand evaluates to FALSE or b) the left operand evaluates to FALSE. In the latter case, the right operand will not be evaluated.
RECOMMENDS	Like REQUIRES but always yields TRUE.
DISCOURAGES	Like CONFLICTS but always yields TRUE.
EQUALS	Evaluates to TRUE, iff either both operands evaluate to TRUE or both operands evaluate to 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
```

9.7.13. 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. If the condition evaluates to FALSE, expression*alternative-expr* is evaluated instead. In partial evaluation, if the condition is *open*, both *consequence-expr* and *alternative-expr* are evaluated. If both result values are equal, that equal value with be the result of the conditional. Otherwise the result is an *open* value. It is an error if*boolean-condition-expr* cannot be evaluated to a Boolean value.

Conditionals can occur everywhere where 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
```

9.7.14. 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. In partial evaluation, if one if the operands is *open*, the result of the comparison will be also *open*.

Beginning with pure::variants 5.0.0, in general values of different base types are not comparable. A comparison of such value combinations will create an error. Exceptions are a) the number types (*ps:float* and *ps:integer* are comparable) and b) versions (type *ps:version*), which also can be compared with strings (type *ps:string*).

Two numbers are compared based on their numeric values, two strings lexically, two collections item by item, two booleans by their boolean values, and model and element references by their ID.

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.

The types *ps:boolean*, *ps:element*, and *ps:model* do not have a natural order. Thus, beginning with pure::variants 5.0.0 any order comparison of such values will create an error.

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
```

9.7.15. Arithmetics

Numbers can be negated, added, 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. Division by zero and floating point overflows create errors.

In partial evaluation, if one of the operands is *open*, the result will usually also be *open*. Exceptions are: Multiplication of *open* by zero and division of zero by *open* results both in zero.

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=18.

Syntax

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

Examples

```
5 * 5 + 2 * 5 * 6 + 6 * 6 = 121
-(8 * 10) + (10 * 8) = 0
-0xFF / 5 = -51
-5->pv:Abs() = -5
(-5)->pv:Abs() = 5
```

9.7.16. 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
```

9.7.17. Function Definitions

The DEF keyword defines a function with name *fct-name* and the given parameter list (see syntax below). Multiple functions with the same name can be defined, if they have different numbers of parameters. Defining multiple functions with the same name and same number of arguments are not allowed (one-definition rule (ODR)). Using the same function name as for built-in functions is also not allowed. The parameters of the definition are only accessible in the function body (*fct-body-expr*). The result of calling such a function is the value of the *fct-body-expr* calculated for the given argument list.

Since pure::variants 5.0.0, such functions can also be called using meta-attribute syntax if they have at least one parameter. In this case, the context on which the function is called is assigned to the first parameter of the function. The arguments of the function call are assigned to the remaining parameters of the function.

Function definitions are only allowed at the beginning of a pvSCL expression. pvSCL expressions which contain only function definitions evaluate to TRUE. To avoid name conflicts, it is recommended to use own name spaces for the function and parameter names (e.g. *my:fct-name* instead of *fct-name*, and *my:param-name* instead of *param-name*). To avoid future name conflicts it is recommended not to use the *pv* name space for function names.

If not defined in a pvSCL code library, such a 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 plus(x,y) = x + y;
plus(plus(plus(1,2),3),4) = 10 // function syntax
AND
1->plus(2)->plus(3)->plus(4) = 10 // meta-attribute syntax
```

9.7.18. 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.

Since pure::variants 5.0.0, functions can also be called using meta-attribute syntax if they have at least one parameter. In this case, the context on which the function is called is assigned to the first parameter of the function. The arguments of the function call are assigned to the remaining parameters of the function.

Syntax

```
fct-name(expr1, expr2, ...) // function syntax
```

```
is equivalent to

expr1->fct-name(expr2, ...) // meta-attribute syntax
```

Examples

```
average(accounts,'income') > average(accounts,'outgoings') // function syntax
accounts->average('income') > accounts->average('outgoings') // meta-attribute
 syntax
```

9.7.19. 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)
```

9.7.20. 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
```

9.7.21. Error Handling

During evaluation of pvSCL expressions, using wrong syntax, wrong input types or invalid values will create evaluation errors and the evaluation of that expression is canceled. In partial evaluation, the usage of *open* values can hide such errors. An example is getting an item of a collection by using function *pv:Item(n)*, when *n* is *open*. If *n* evaluates to a concrete number in future configurations, the function will return either the *n*th item or cancels with an index-out-of-range error. Since it cannot be known beforehand, the partial evaluation returns not only *open*, but also sets a potential-error flag for the evaluation of that pvSCL expression. Even if the evaluation of the complete expression results in a constant value, like in

```
collection->pv:Item(Feature->openattr) = 2 OR SelectedFeature
```

which will return either TRUE or create an error, the partial evaluation will always return *open* if that potential-error flag is set.

Errors, warnings, and information markers can also be created using functions *pv:Fail*, *pv:Warn*, and *pv:Inform*, respectively. Usually they are applied in expressions like

```
condition OR pv:Fail('Error: Condition is not fulfilled.')
```

So if *condition* evaluates to FALSE, the right operand of OR, *pv:Fail*, is executed and an error marker is created. If the *condition* evaluates to TRUE, the shortcut applies and *pv:Fail* is not executed. However, if in partial evaluation *condition* evaluates to *open*, the right operand of OR also needs to be executed. So the execution of *pv:Fail* actually needs to create an error marker, although it is not clear, if the condition is fulfilled or not. To avoid this, operand expressions, which needs to be only executed because a shortcut could not be applied because of an *open* operand, will be executed in a special mode, where *pv:Fail*, *pv:Warn*, and *pv:Inform* will not create any markers.

9.7.22. Limitations

Depth of syntax tree

The depth of the syntax tree of the parsed pvSCL expression is limited to 512 levels by default. If a pvSCL expression exceeds this limit, a parsing error will be created and the expression will not be processed further.

An example for pvSCL expressions, which could raise this limit, are sequences of operations of a large number of operands without parentheses like

```
Feature_1 OR Feature_2 OR ... OR Feature_520
```

Usually grouping the operations by parentheses will reduce the number of syntax tree levels of these kind of pvSCL expressions. Another example are deep nested expressions like

```
IF Feature_1 THEN ..
ELSE IF Feature_2 THEN ..
...
ELSE IF Feature_520 THEN ..
ELSE ..
ENDIF ... ENDIF
```

The default limit can be overridden by setting the environment variable PV_PVSCL_MAX_AST_LEVELS to a numeric value greater than zero, interpreted as the new number of maximum levels. The value of this variable will be considered by the pure::variants Desktop Client, the pure::variants Web Client and all pure::variants integrations. It will also be considered by the pure::variants Model Server, if the variable is set during the launch of the server process.

Note

Increasing the limit above the default limit can lead to stack overflows and crashes of the pure::variants Desktop Client, the pure::variants Web Client, the pure::variants integrations, and the pure::variants Model Server. So the changing of the limit should only be done if it is really needed and is subject to the user's own risk.

Depth of recursive operation calls

The depth of recursive operation calls of an executed pvSCL expression is limited to 512 operations by default. Usually, pvSCL expressions with recursive function calls can exceed this limit. Example:

```
DEF sum(x) = IF x = 0 THEN 0 ELSE sum(x-1) + x ENDIF;
sum(1000)
```

In that example, the execution will be canceled and the error “pvSCL call depth limit of *limit* operations reached. It may be an endless recursion.” will be created.

In partial evaluation, the limit can also be reached because of an endless recursion created by an open termination condition. Example:

```
DEF sum(x) = IF x = 0 THEN 0 ELSE sum(x-1) + x ENDIF;
sum(Feature->openAttr)
```

Because of the open condition *x = 0*, this results in an endless recursion of calls of the function *sum* with open argument *x*. If the operation depth limit is reached, the recursion is canceled and the result of the function will be an *open* value.

The default limit can be overridden by setting the environment variable or Java system property PV_PVSCL_MAX_OP_CALL_DEPTH to a numeric value greater than zero, interpreted as the new number of maximum operations. The value of this variable will be considered by the pure::variants Desktop Client, the pure::variants Web Client and all pure::variants integrations.

Note

Increasing the limit above the default limit can lead to stack overflows and crashes of the pure::variants Desktop Client, the pure::variants Web Client and all pure::variants integrations. This can be prevented by also increasing the thread stack size of the Java VM (e.g. by setting the JVM argument *-Xss*). The changing of the limit should only be done if it is really needed and is subject to the user's own risk.

9.7.23. Function Library

In partial evaluation all functions can process *open* values as context and as each of their arguments. Depending on the functionality the return value can be also *open*.

pv:Abs()

Get the absolute value of the context which must be a number.

Examples

```
10->pv:Abs() = 10
(-10)->pv:Abs() = 10
(-2.5)->pv:Abs() = 2.5
```

pv:Acos()

Return the trigonometric arc cosine of the context number. The result value has type *ps:float*. This function must only be called on numbers between -1 and 1.

Examples

```
0->pv:Acos() = 1.5707963267948966
0.2->pv:Acos() = 1.369438406004566
1->pv:Acos() = 0.0
(-1)->pv:Acos() = 3.141592653589793
```

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. If the context collection is a set, then the item only is appended if not already contained in the set.

Examples

```
{}->pv:Append(1) = {1}
{1,2,3}->pv:Append(2) = {1,2,3,2}
{1,2,3}->pv:AsSet()->pv:Append(2) = {1,2,3}->pv:AsSet()
```

pv:AppendAll(collection)

Append all elements of the argument collection to the context collection. It is an error if the types of both collection don't match. If the context collection is a set, then only items from the argument collection are appended if not already contained in the set.

Examples

```
{}->pv:AppendAll({1,2,3}) = {1,2,3}
{1,2,3}->pv:AppendAll({1,3,5}) = {1,2,3,1,3,5}
{1,2,3}->pv:AsSet()->pv:AppendAll({1,3,5}) = {1,2,3,5}->pv:AsSet()
```

pv:Asin()

Return the trigonometric arc sine of the context number. The result value has type *ps:float*. This function must only be called on numbers between -1 and 1.

Examples

```
0->pv:Asin() = 0
0.2->pv:Asin() = 0.2013579207903308
1->pv:Asin() = 1.5707963267948966
(-1)->pv:Asin() = -1.5707963267948966
```

pv:AsList()

Convert the context to a list. It is an error if the context does not have collection type.

Examples

```
{1,1,2,3}->pv:AsList() = {1,1,2,3}
```

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:Atan()

Return the trigonometric arc tangent of the context number. The result value has type *ps:float*.

Examples

```
0->pv:Atan() = 0
0.2->pv:Atan() = 0.19739555984988078
100->pv:Atan() = 1.5607966601082315
(-100)->pv:Atan() = -1.5607966601082315
```

pv:Attribute(name)

Get the attribute with the given non-empty name. Calling this function with an empty name creates an error. 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:Attributes(), pv:Attributes('type')

Get all attributes of the context, optionally with the non-empty (exact) type. Calling this function with an empty type creates an error. Fails if the context does neither have model nor element type, or if no attribute (with given type) exists.

Examples

```
self->pv:Attributes('ps:integer')
```

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. Fails 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->pv:SubTree(false)->pv:Select(e|e->pv:Selected())->pv:Size()
```

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:Contains(expr)

Check whether the evaluation result of expression *expr* is contained in the context, which must be a collection.

Examples

```
{1,2,3}->pv:Contains(3) = true
```

pv:ContainsAll(collection)

Check whether each value of *collection* is contained in the context, which must be a collection.

Examples

```
{1,2,3}->pv:ContainsAll({1,2}) = true
```

pv:Cos()

Return the trigonometric cosine of the context number. The result value has type *ps:float*.

Examples

```
0->pv:Cos() = 1
0.2->pv:Cos() = 0.9800665778412416
100->pv:Cos() = 0.8623188722876839
(-100)->pv:Cos() = 0.8623188722876839
```

pv:Date()

Returns the date of the given date time value. If the date time is timezoned, the date in GMT time zone is returned. The result type is *ps:date*. If the given date time value is timezoned, the resulting date is also timezoned.

Examples

```
pv:EvaluationDateTime()->pv:Date()->pv:ToString()
'2020-02-28T10:24:42'->pv:ToDateTime()->pv:Date()->pv:ToString() = '2020-02-28'
'2020-02-28T10:24:42+01:00'->pv:ToDateTime()->pv:Date()->pv:ToString() =
 '2020-02-28Z'
```

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:Element(name-or-id)

Get the element with the given non-empty unique name or identifier. Calling the function with an empty unique name or identifier creates an error. If called on a model, only elements in that model will be 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:EvaluationDateTime()

Returns the date and time when the current evaluation has started. The result type is *ps:datetime*.

Examples

```
pv:EvaluationDateTime()->pv:ToString()
```

pv:EvaluationIsPartial()

Returns *true* if the current evaluation is executed in partial configuration mode. Returns *false* if the current evaluation is executed in full configuration mode. The result type is *ps:boolean*.

Examples

```
IF pv:EvaluationIsPartial() THEN 'fixed' ELSE 'open' ENDIF
```

pv:ExclusionHint(message,element), pv:ExclusionHint(message,element,force)

If the given element is not excluded in a full configuration or selected in a partial configuration, a warning or error message will be created. The severity of the message will be defined by the Boolean force argument: If the force argument is missing or TRUE, an error message will be created. If the force argument is FALSE, only a warning message is created. This function will always return TRUE. If activated, the auto resolver will try to resolve warning and error messages created by this function by excluding the given element if possible.

Examples

```
product->price < 100 IMPLIES
  pv:ExclusionHint('Because of the low price, feature \'luxury\' could be
 excluded.', luxury, false)
```

pv:Exp()

Return the Euler's number *e* raised to the power of the context number (exponent). The result value has type *ps:float*.

Examples

```
1->pv:Exp() = 2.718281828459045
(-1.2)->pv:Exp() = 0.30119421191220214
```

pv:Fail(message), pv:Fail(message,element)

Show an error message at the context element or the given element. Always returns TRUE. Lets the model evaluation fail.

Examples

```
doors->number = 2 OR
  doors->number = 4 OR
  pv:Fail('Invalid number of doors [' + doors->number + ']', doors)
```

pv:Flatten()

Flatten the context, which has to be a collection.

Examples

```
LET
list1 = {1,2,3,4},
list2 = {{0},list1,{5}}
IN
list2->pv:Flatten()->pv:ToString()
```

pv:Floor()

Get the largest integer value not greater than the context number (round downwards towards negative infinity). Fails if the context does not have number type. The return type is *ps:integer*.

Examples

```
3.1->pv:Floor() = 3
3.5->pv:Floor() = 3
3.9->pv:Floor() = 3
(-3.1)->pv:Floor() = -4
(-3.5)->pv:Floor() = -4
(-3.9)->pv:Floor() = -4
```

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, otherwise return TRUE.

Examples

```
bugs->pv:Children()->
  pv:ForAll(bug | bug->state = 'fixed')
```

pv:ForAny(iterator)

Iterate the context collection and evaluate the iterator expression for all items. Return TRUE if at least for one item the expression evaluates to TRUE, otherwise return FALSE.

Examples

```
bugs->pv:Children()->
  pv:ForAny(bug | bug->state <> 'fixed')
```

pv:Format(format)

Return a formatted string representation of the context number. Fails if the context does not have number type. The return type is *ps:string*.

The argument is a C-printf-like format specifier string. The supported strings are shown in Table 9.7, “Supported format specifiers”. The output is not localized.

Table 9.7. Supported format specifiers

Context number type	Format specifier	Meaning
ps:integer	%d	Decimal integer
ps:integer	%x	Hexadecimal integer with lower-case letters
ps:integer	%X	Hexadecimal integer with upper-case letters
ps:integer	%o	Octal integer
ps:float	%e	Scientific (exponential) notation with six digits after the decimal point. Uses a lower-case letter 'e' as the exponent symbol.
ps:float	%.*n*e	Scientific (exponential) notation with *n* digits after the decimal point. Uses a lower-case letter 'e' as the exponent symbol.
ps:float	%E	Same as %e, but with upper-case letter 'E'
ps:float	%.*n*E	Same as %.*n*e, but with upper-case letter 'E'
ps:float	%f	Decimal (non-exponential) notation with six digits after the decimal point.
ps:float	%.*n*f	Decimal (non-exponential) notation with *n* digits after the decimal point.

Due to limited precision of *ps:float* values (they are internally represented in double-precision floating-point format), each *ps:float* value can be represented by at most 17 significant decimal digits. So, formatting a *ps:float* value with more digits would pretend a higher precision in the output compared to the input. Additionally, the exact formatted output in this excessive precision range depends on the runtime libraries of the used operation system. So, the resulting string can be different across operation systems. In result, it is recommended to use the format specifiers %.*n*e and %.*n*f only with *n* <= 16 and *n* <= 17-*i* respectively, where *i* is the number of significant digits before the decimal point in non-exponential notation.

Examples

```
51966->pv:Format('%x') = 'cafe'
3.14159265->pv:Format('%f') = '3.141593'
3.14159265->pv:Format('%f') = '3.141593'
6.62607015e-34->pv:Format('%e') = '6.626070e-34'
6.62607015e-34->pv:Format('%.0e') = '7e-34'
```

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)

Returns TRUE if the attribute with the given non-empty name exists on the context model or element, FALSE otherwise. Calling this function with an empty name creates an error. Fails if the context does not have model or element type.

Examples

```
self->pv:HasAttribute('speed') = true
```

pv:HasElement(name-or-id)

Returns TRUE if the element with the given non-empty name or identifier exists, FALSE otherwise. Calling this function with an empty name or identifier creates an error. If called on a model, only elements in that model will be 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)

Returns TRUE if the model with the given non-empty name or identifier exists, FALSE otherwise. Calling this function with an empty name or identifier creates an error.

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 fails otherwise.

Examples

```
context->pv:ID() <> ''
```

pv:IndexOf(string-or-collection)

Return the index (starting at 0) of the first occurrence of the given non-empty sub-string or collection item within the context, or -1 if the given item was not found. Calling this function on a string with an empty string argument creates an error. It is also an error if the context does not have string or collection type. The resulting index has type *ps:integer*.

Examples

```
'Hello World'->pv:IndexOf('World') = 6
{1,2,3}->pv:IndexOf(2) = 1
{1,2,3}->pv:IndexOf(4) = -1
```

pv:Inform(message), pv:Inform(message,element)

Show an informational message at the context element or the given element. Always returns TRUE.

Examples

```
sportedition AND NOT(rearspoiler) RECOMMENDS
  pv:Inform('Rear spoiler recommended for sport edition', rearspoiler)
```

pv:Insert(index,item)

Insert the given item into the context collection before the item at the given index. It is an error if the type of the item is not compatible to the item type of the context collection. Using this function with index 0 is the same as calling *pv:Prepend(item)* on the collection. And using this function with the size of the context collection as index is the same as calling *pv:Append(item)* on the collection. If the context collection is a set, then the item is only inserted at the given index if not already contained in the set.

Examples

```
{}->pv:Insert(0,4) = {4}
{1,2,3}->pv:Insert(3,4) = {1,2,3,4}
{1,2,3}->pv:Insert(0,4) = {4,1,2,3}
{1,2,3}->pv:Insert(1,4) = {1,4,2,3}
{1,2,3}->pv:AsSet()->pv:Insert(0,3) = {1,2,3}->pv:AsSet()
{1,2,3}->pv:AsSet()->pv:Insert(0,4) = {4,1,2,3}->pv:AsSet()
```

pv:InsertAll(index,collection)

Insert the given items into the context collection before the item at the given index. It is an error if the type of the argument collection is not compatible to the type of the context collection. Using this function with index 0 is the same as calling *pv:PrependAll(collection)* on the collection. And using this function with the size of the context collection as index is the same as calling *pv:AppendAll(collection)* on the collection. If the context collection is a set, then only items from the argument collection are inserted at the given index if not already contained in the set.

Examples

```
{}->pv:InsertAll(0,{1,2}) = {1,2}
{1,2,3}->pv:InsertAll(3,{4,5}) = {1,2,3,4,5}
{1,2,3}->pv:InsertAll(0,{-1,0}) = {-1,0,1,2,3}
{1,2,3}->pv:InsertAll(1,{1,1,2}) = {1,1,1,2,2,3}
{1,2,3}->pv:AsSet()->pv:InsertAll(3,{1,2,3,4,5}) = {1,2,3,4,5}->pv:AsSet()
```

pv:IsContainer()

Return TRUE if the context is a container, i.e. a collection like list or set.

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)

Returns TRUE if the type of the context object is the same as the non-empty type given as argument, or a type derived from it. Calling this function with an empty type creates an error.

The type of the context object needs to be defined in a type model. Otherwise it will always return FALSE.

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:SubTree(false)->pv:Select(e|e->pv:Selected())->
    pv:Iterate(product; price = 0 | price + product->price) + '$')
```

pv:LastIndexOf(string-or-collection)

Return the index (starting at 0) of the last occurrence of the given non-empty sub-string or collection item within the context, or -1 if the given item was not found. Calling this function on a string with an empty string argument creates an error. It is also an error if the context does not have string or collection type. The resulting index has type *ps:integer*.

Examples

```
'Hello World, Hello World'->pv:IndexOf('World') = 19
{1,2,3,2}->pv:IndexOf(2) = 3
{1,2,3,2}->pv:IndexOf(4) = -1
```

pv:Log()

Return the natural logarithm (base *e*) of the context number. The result value has type *ps:float*. This function must not be called on zero and negative numbers.

Examples

```
1->pv:Log() = 0
0.2->pv:Log() = -1.6094379124341003
```

pv:Log10()

Return the common logarithm (base 10) of the context number. The result value has type *ps:float*. This function must not be called on zero and negative numbers.

Examples

```
100->pv:Log10() = 2
0.2->pv:Log10() = -0.6989700043360187
```

pv:Max(), pv:Max(number)

If called on a number collection and no arguments, it returns the greatest number of the collection. If called on a single number and one number argument (both either *ps:integer* or *ps:float*), it returns the greater of the context number and the argument number. The return type is *ps:integer* or *ps:float* depending on the type of the context. The result for an empty collection is 0.

Examples

```
{1,2,3,4}->pv:Max() = 4
2->pv:Max(4) = 4
```

pv:Min(), pv:Min(number)

If called on a number collection and no arguments, it returns the smallest number of the collection. If called on a single number and one number argument (both either *ps:integer* or *ps:float*), it returns the smaller of the context number and the argument number. The return type is *ps:integer* or *ps:float* depending on the type of the context. The result for an empty collection is 0.

Examples

```
{1,2,3,4}->pv:Min() = 1
2->pv:Min(4) = 2
```

pv:Mod(divisor)

Return the remainder of the devision of the context integer number (dividend) with the argument integer number (divisor, modulo operation). The return type is *ps:integer*. If the dividend is a negative number, then the remainder also is negative. The divisor must not be zero. A negative divisor is treated as if it were positive.

Examples

```
5->pv:Mod(3) = 2
5->pv:Mod(-3) = 2
(-5)->pv:Mod(3) = -2
(-5)->pv:Mod(-3) = -2
```

pv:Model(), pv:Model(name-or-id)

Get the model, as *ps:model*, containing the context element, or the model with the given non-empty 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, or if it is called with an empty model name or identifier.

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 all models of a configuration space as *ps:model[]* collection. Optionally accepts a non-empty model type as argument to get only the models of a specific type. The parameter type is of ps:string type. See Table 5.1, “Mapping between input and concrete model types” for the list of applicable type names. Calling the function with an empty model type string creates an error. If applied on an object, call fails if the object is not of configuration space type ('ps:configspace').

Examples

```
pv:Models('ps:fm')->pv:Size() > 1
/* applicable everywhere, number of feature models more than 1 */

context->pv:Parent()->pv:Models()->pv:Size() > 1
/* this form only in constraints*/
/* context of constraint is a model, parent is config space */
```

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:Pow(exponent)

Return the value of the context number (base) raised to the power of the argument number (exponent). If both, the base and the exponent, are integers, then the result value has type *ps:integer*. Otherwise the result value has type *ps:float*. If the base is negative, then the exponent has to be an integer.

Examples

```
2->pv:Pow(8) = 256
3.14->pv:Pow(2) = 9.8596
```

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

```
{}->pv:Prepend(1) = {1}
{1,2,3}->pv:Prepend(2) = {2,1,2,3}
{1,2,3}->pv:AsSet()->pv:Prepend(2) = {1,2,3}->pv:AsSet()
```

pv:PrependAll(collection)

Prepend the values of *collection* to the context, which must be a collection. It is an error if the types of collections are not compatible.

Examples

```
{}->pv:PrependAll({1,2,3}) = {1,2,3}
{1,2,3}->pv:PrependAll({1,3,5}) = {1,3,5,1,2,3}
{1,2,3}->pv:AsSet()->pv:PrependAll({1,3,5}) = {5,1,2,3}->pv:AsSet()
```

pv:PVVersion()

Get the current version of pure::variants as *ps:version*. The result contains the complete version string, e.g. '4.0.7.685'. To check against specific versions, comparison operators can be used.

Examples

```
pv:PVVersion() >= '4.0.7.*' /* at least version 4.0.7 */
pv:PVVersion() = '4.0.*' /* any service release of the 4.0 branch */
pv:PVVersion() < '4.*' /* any release before version 4.x */
```

pv:Relations(), pv:Relations(type)

Get the relations of class *ps:dependencies* defined on the context element, as *ps:relation[]*. Optionally accepts a non-empty relation type as argument to get only relations of the given type. Calling this function with an empty relation type creates an error. 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:Remove(item), pv:Remove(begin,end)

If called with an *item* as the single argument, then a new collection is returned containing all the items from the context collection except of the given item. If called with an index range instead, then the resulting collection contains all the items from the context collection except the items with index *begin* up to index *end-1*.

Examples

```
{'a','b','c','b','a'}->pv:Remove('b') = {'a','c','a'}
{'a','b','c','b','a'}->pv:Remove(0,2) = {'c','b','a'}
{'a','b','c','b','a'}->pv:Remove(3,5) = {'a','b','c'}
```

pv:RemoveAll(collection)

If the context, which must be a collection, contains an element from the given collection, this element is removed from the context.

Examples

```
{1,2,3,2,1}->pv:RemoveAll({1,3}) = {2,2}
```

pv:RetainAll(collection)

If an element of the given collection is not contained in the context, which has to be a collection, it will be removed from the context.

Examples

```
{1,2,3,2,1}->pv:RetainAll({2,3}) = {2,3,2}
```

pv:Reverse()

Reverses the context, which has to be a collection.

Examples

```
{1,2,3,4,5}->pv:Reverse() = {5,4,3,2,1}
```

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()

Get the integer value nearest to the context number. Positive context numbers are rounded up towards positive infinity if the fractional part is equal to or greater than 0.5, and rounded downwards towards negative infinity otherwise. Negative context numbers are rounded downwards towards negative infinity if the fractional part is equal to or greater than 0.5, and rounded up towards positive infinity otherwise. Fails if the context does not have number type. The return type is *ps:integer*.

Examples

```
3.1->pv:Round() = 3
3.5->pv:Round() = 4
3.9->pv:Round() = 4
(-3.1)->pv:Round() = -3
(-3.5)->pv:Round() = -4
(-3.9)->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:SelectionHint(message,element), pv:SelectionHint(message,element,force)

If the given element is not selected in a full configuration or excluded in a partial configuration, a warning or error message will be created. The severity of the message will be defined by the Boolean force argument: If the force argument is missing or TRUE, an error message will be created. If the force argument is FALSE, only a warning message is created. This function will always return TRUE. If activated, the auto resolver will try to resolve warning and error messages created by this function by selecting the given element if possible.

Examples

```
product->price > 1000 IMPLIES
  pv:SelectionHint('Because of the high price, feature \'luxury\' could be
 selected.', luxury, false)
```

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 all other selections, or *none* for elements that neither are explicitly or automatically selected nor excluded.

Examples

```
self IMPLIES self->pv:Selector() = 'ps:user'
  OR pv:Inform('Feature ' + self->pv:Name() +
    ' was added automatically')
```

pv:Sequence(end), pv:Sequence(begin,end), pv:Sequence(begin,end,increment)

Generate the finite sequence of all integers, beginning with integer *begin* and increased by integer *increment* each, which are less than (so excluding) integer *end*. If the function signatures without *begin* and *increment* are called, the values 0 and 1 are used, respectively. The resulting type is *integer[]* An *increment* of 0 (zero) is not allowed and creates an error. This function can be used as input of collection iteration functions to iterate across more than one collection at a time.

Examples

```
pv:Sequence(3) = {0, 1, 2}
pv:Sequence(1,5) = {1, 2, 3, 4}
pv:Sequence(0,10,2) = {0, 2, 4, 8}
LET list1 = {'A','B','C'}, list2 = {'1','2','3'} IN
  pv:Sequence(list1->pv:Size())->
  pv:Collect(i | list1->pv:Item(i) + list2->pv:Item(i)) = {'A1','B2','B3'}
```

pv:Sin()

Return the trigonometric sine of the context number. The result value has type *ps:float*.

Examples

```
0->pv:Sin() = 0
0.2->pv:Sin() = 0.19866933079506122
100->pv:Sin() = -0.5063656411097588
(-100)->pv:Sin() = 0.5063656411097588
```

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:SubTree(false)->pv:Select(e|e->pv:Selected())->pv:Size() = 1
```

pv:Sort()

Sort the items of the context collection in ascending order. Numbers are sorted by value and precede strings. Strings are sorted alphabetically where upper-case characters precede lower-case characters. Collections are sorted by their elements.

Examples

```
{1,3,2}->pv:Sort() = {1,2,3}
{'c','C','b'}->pv:Sort() = {'C','b','c'}
{1.6,-1.0,0.3}->pv:Sort() = {-1.0,0.3,1.6}
{ {3,1}, {1,3}, {2,1}, {1,2} }->pv:Sort() = { {1,2}, {1,3}, {2,1}, {3,1} }
{ {{3,1},{1,3}}, {{2,1},{1,2}} }->pv:Sort() = { {{2,1},{1,2}}, {{3,1},{1,3}} }
```

pv:Sqrt()

Return the square root of the context number. The result value has type *ps:float*. This function must not be called on negative numbers.

Examples

```
9->pv:Sqrt() = 3
1.2->pv:Sqrt() = 1.0954451150103321
```

pv:StringSplit(delimiter)

Splits the context string around occurrences of the string *delimiter* in the context string and returns the result value as list of string (*ps:string[]*). It is an error if the context or *delimiter* does not have string type and if the *delimiter* is an empty string.

Examples

```
'Hello World'->pv:StringSplit(' ') = {'Hello','World'}
```

pv:StringReplace(search,replace)

Replace the *first* occurrence of the string *search* in the context string with the string *replace* and return the result. It is an error if context, *search*, or *replace* does not have string type and if the *search* string is an empty string.

Examples

```
'Hello World, Hello World'->pv:StringReplace('World','All') =
  'Hello All, Hello World'
```

pv:StringReplaceAll(search,replace)

Replace *all* occurrences of the string *search* in the context string with the string *replace* and return the result. It is an error if context, *search*, or *replace* does not have string type and if the *search* string is an empty string.

Examples

```
'Hello World, Hello World'->pv:StringReplace('World','All') =
  'Hello All, Hello All'
```

pv:SubList(begin), pv:SubList(begin,end)

Return a new collection that is a sub-collection of the context collection. The sub-collection begins at the specified *begin* index and extends to the *end-1* index or end of the context collection. It is an error if the context does not have collection type.

Examples

```
{1,2,3,4,5}->pv:SubList(0) = {1,2,3,4,5}
{1,2,3,4,5}->pv:SubList(2) = {3,4,5}
{1,2,3,4,5}->pv:SubList(5) = {}
{1,2,3,4,5}->pv:SubList(1,4) = {2,3,4}
{1,2,3,4,5}->pv:SubList(0,0) = {}
{1,2,3,4,5}->pv:SubList(0,1) = {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:SubTree()

Get all elements of a model, or just a sub-tree. If the context is a model, then all elements of that model are returned. If the context is an element and the function is called without an argument or with *true* as argument, then the sub-tree with this element as root is returned. If called on an element with argument *false*, then the context element will not be part of the result.

Examples

```
model->pv:SubTree->pv:ForAll(e|not(e->pv:Selected))
element->pv:SubTree(false)->pv:Select(e|e->pv:Selected)->pv:Size > 0
```

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:Tan()

Return the trigonometric tangent of the context number. The result value has type *ps:float*.

Examples

```
0->pv:Tan() = 0
0.2->pv:Tan() = 0.2027100355086725
100->pv:Tan() = -0.5872139151569291
(-100)->pv:Tan() = 0.5872139151569291
```

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:Time()

Returns the time of the given date time value. The result type is *ps:time*. If the given date time value is timezoned, the resulting time is also timezoned.

Examples

```
pv:EvaluationDateTime()->pv:Time()->ToString()
'2020-02-28T10:24:42'->pv:ToDateTime()->pv:Time()->pv:ToString() = '10:24:42.000'
'2020-02-28T10:24:42+01:00'->pv:ToDateTime()->pv:Time()->pv:ToString() =
 '09:24:42.000Z'
```

pv:ToDate()

Converts the context string containing a date in XML Schema date format with or without time zone into a date value of type *ps:date*. The supported format is: `'-'?[0-9]{4,}'-'[0-1][0-9]'-'[0-3][0-9]('Z'|('+'|'-')[0-2][0-9]':'[0-9][0-9])?` It has to be an existing date in the Gregorian calendar and the time zone, if given, has to be in range +14:00 to -14:00. During conversion the eventually existing time zone is normalized to so-called recoverable time zone, which has the range +12:00 to -11:59. It fails, if the date format is invalid, or the given date does not exist.

Examples

```
'2020-02-28->pv:ToDate()->pv:ToString() = '2020-02-28'
'2020-02-28Z'->pv:ToDate()->pv:ToString() = '2020-02-28Z'
'2020-02-28+01:00'->pv:ToDate()->pv:ToString() = '2020-02-27Z'
'-0050-07-13'->pv:ToDate()->pv:ToString() = '-0050-07-13'
```

pv:ToDateTime()

Converts the context string containing a date and time in XML Schema dateTime format with or without time zone into a date time value of type *ps:datetime*. The supported format is: `'-'?[0-9]{4,}'-'[0-1][0-9]'-'[0-3][0-9]'T'[0-2][0-9]':'[0-5][0-9]':'[0-5][0-9]('.'[0-9]+)?('Z'|('+'|'-')[0-2][0-9]':'[0-9][0-9])?` It has to be an existing date in the Gregorian calendar and the time zone, if given, has to be in range +14:00 to -14:00. During conversion the time is rounded to millisecond precision and, if a time zone is given, the time is normalized to GMT. It fails, if the date time format is invalid, or the given date or time does not exist.

Examples

```
'2020-02-28T12:34:56'->pv:ToDateTime()->pv:ToString() = '2020-02-28T12:34:56.000'
'2020-02-28T12:34:56.25Z'->pv:ToDateTime()->pv:ToString() =
 '2020-02-28T12:34:56.250Z'
'2020-02-28T00:02:42.123+01:00'->pv:ToDateTime()->pv:ToString() =
 '2020-02-27T23:02:42.123Z'
```

pv:ToFloat()

Convert the context number or string to a floating point number. If the context number is an integer, which exceeds the range of valid float number, an overflow error is created. If the context number is already a float, it will be returned as it is. If the context is a string, it will be parsed as a float value. Valid float strings have to follow the grammar:

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

If the string is not a valid float string, an error will be created. The return type is *ps:float*.

Examples

```
1->pv:ToFloat() = 1.0
'-2.3e42'->pv:ToFloat() = -2.3e42
```

pv:ToInteger(), pv:ToInteger(radix)

Convert the context number or string to an integer number. If the context number is a float, it will be truncated (i.e., round towards zero). If the context number is already an integer, it will be returned as it is. If the context is a string, it will be parsed as an integer value with radix *radix* (or radix 10 if omitted). The value of *radix* has to be in the range [2, 36], where for radices 2 to 10, numerical digits '0' to '9', and for the radices 11 to 36 additionally the letter digits 'A' to 'Z' (or 'a' to 'z') are used. If the string is not a valid integer string, an error will be created. The return type is *ps:integer*.

Examples

```
1.9->pv:ToInteger() = 1
'-123'->pv:ToInteger() = -123
'CAFE'->pv:ToInteger(16) = 51966
```

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(), pv:ToString(delimiter), pv:ToString(delimiter,last delimiter)

Return a string representation of the context object. The return type is *ps:string*.

If a delimiter is given, then the context object needs to be a collection. Instead of just converting the collection to a string, the collection items are listed each separated from the other using the given delimiter. If additionally a last delimiter is given, then this delimiter is inserted between the last item in the collection and its predecessor.

Examples
```
6->pv:ToString() = '6'
{1,2,3}->pv:ToString = '{1,2,3}'
{1,2,3}->pv:ToString('+') = '1+2+3'
{1,2,3}->pv:ToString(', ',', and ') = '1, 2, and 3'
{{100,-100},{30,75},{10}}->pv:ToString(', ',', and ') = '{100,-100}, {30,75}, and
 {10}'
```

pv:ToTime()

Converts the context string containing a time in XML Schema time format with or without time zone into a time value of type *ps:time*. The supported format is: `[0-2][0-9]':'[0-5][0-9]':'[0-5][0-9]('.'[0-9]+)?('Z'|('+'|'-')[0-2][0-9]':'[0-9][0-9])?` The time zone, if given, has to be in range +14:00 to -14:00. During conversion the time is rounded to millisecond precision and, if a time zone is given, the time is normalized to GMT. It fails, if the time format is invalid, or the given time does not exist.

Examples
```
'12:34:56'->pv:ToTime()->pv:ToString() = '12:34:56.000'
'12:34:56.25Z'->pv:ToTime()->pv:ToString() = '12:34:56.250Z'
'00:02:42.123+01:00'->pv:ToTime()->pv:ToString() = '23:02:42.123Z'
```

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:Truncate()

Convert the context number into an integer by truncating the fractional digits. Fails if the context does not have number type. The return type is *ps:integer*.

Examples
```
3->pv:Truncate() = 3
1.9->pv:Truncate() = 1
(-2.6)->pv:Truncate() = -2
```

pv:Type()

Get the type of the context as *ps:string*.

Examples
```
'hello'->pv:Type() = 'ps:string'
42->pv:Type() = 'ps:integer'
FeatureA->pv:Type() = 'ps:feature'
```

pv:VariantName()

Get the name of the currently evaluated variant as *ps:string*.

Examples
```
pv:VariantName() = pv:Models('ps:vdm')->pv:Item(0)->pv:Name()
```

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 non-empty language identifier can be specified. Calling the function with an empty language identifier creates an error.

If no language is given, 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 will be 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), pv:Warn(message,element)

Show a warning message at the context element or the given element. Always returns TRUE.

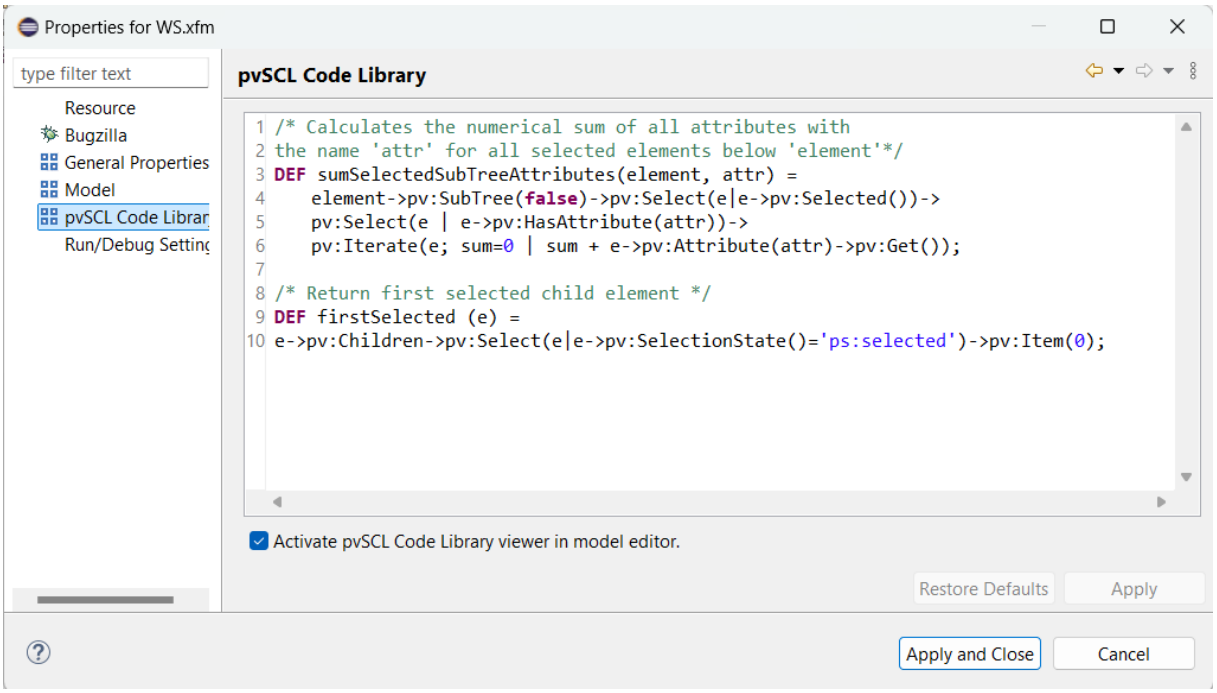
Examples

```
car->wheels > 4 IMPLIES
  pv:Warn('Too many wheels (' + car->wheels + ') configured', car)
```

9.7.24. 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 (seeFigure 9.1, “pvSCL Code Library Model Property Page”).

Figure 9.1. 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.

9.8. 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 9.8. Supported Variables

Variable	Description
`CONFIGSPACE`	Path to the Configuration Space folder.
`CONFIGSPACE_NAME`	Name of the Configuration Space.
`ENV:variable`	The content of the environment variable with the given name.
`INPUT`	Transformation input directory.
`MODULEBASE`	Path to the transformation module base folder.
`OUTPUT`	Transformation output directory.
`PROJECT`	Path to the folder of the current project.
`PROJECT:name`	Path to the folder of the project with the given name.
`QUALIFIER`	The actual time stamp in the form yyyyMMddHHmmss, e.g. 20190101143045.
`TRANSFORMLOG`	Path to the transformation log file.
`TRANSFORMATION`	The name of the transformation configuration which triggered the current transformation.
`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 `VARIANTSPATH` is identical to the value of `VARIANT` .
`WORKSPACE`	Path to the workspace folder.

9.9. Regular Expressions

Regular expressions are used to match patterns against strings.

9.9.1. 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:

x Matches the character x.

\ Matches nothing, but quotes the following character.

\\ Matches the backslash character.

\0n Matches the character with octal value 0n (0 <= n <= 7).

\0nn Matches the character with octal value 0nn (0 <= n <= 7).

\0mnn Matches the character with octal value 0mnn (0 <= m <= 3, 0 <= n <= 7).

\xhh Matches the character with hexadecimal value 0xhh.

\uhhhh Matches the character with hexadecimal value 0xhhhh.

\t Matches the tab character ('\u0009').

\n Matches the newline (line feed) character ('\u000A').

\r Matches the carriage-return character ('\u000D').

\f Matches the form-feed character ('\u000C').

\a Matches the alert (bell) character ('\u0007').

\e Matches the escape character ('\u001B').

\cx Matches the control character corresponding to x.

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.

[abc] Matches the characters a, b, or c.

[^abc] Matches any character except a, b, or c (negation).

[a-zA-Z] Matches the characters a through z or A through Z, inclusive (range).

[a-d[m-p]] Matches the characters a through d, or m through p: [a-dm-p] (union).

[a-z&&[def]] Matches the characters d, e, or f (intersection).

[a-z&&[^bc]] Matches the characters a through z, except for b and c: [ad-z] (subtraction).

[a-z&&[^m-p]] Matches the characters a through z, and not m through p: [a-lq-z] (subtraction).

Predefined character classes:

. Matches any character.

\d Matches a digit: [0-9].

\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).

9.9.2. 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.

9.9.3. 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.

9.9.4. 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.

9.9.5. 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.

9.9.6. 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.

9.9.7. 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.

9.10. Keyboard Shortcuts

Some of the following keyboard shortcuts may not be supported on all operating systems.

Table 9.9. 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 9.10. Model Editor Keyboard Shortcuts

Key	Action
ENTER	Show properties dialog

Key	Action
DEL / ENTF	Delete selected elements
Up/Down cursor keys	Navigate tree
Left/Right cursor keys	Collapse or expand subtree
CTRL+O	Open Quick-Outline
CTRL+INSERT	Create New Feature / Element (Feature Model Editor, Family Model Editor)
Space	Select / Unselect Features (Variant Model Editor / Matrix)
SHIFT+Space	Exclude / Unselect Features (Variant Model Editor / Matrix)
CTRL+1	Evaluate Variant Description Model / Matrix
CTRL+2	Validate Model (Feature Model Editor, Family Model Editor, Variant Model Editor)
CTRL+T	Run last used Transformation (Variant Model Editor / Matrix)

Table 9.11. 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

9.11. Naming Restrictions

There are different naming restrictions for different types of objects, which will be declared in this section.

9.11.1. Project Name

The project name follows the OS-specific rules for directory naming.

Apart from that limitation, there are no characters especially forbidden.

9.11.2. Folder Name

The folder name follows the OS-specific rules for directory naming.

Apart from that limitation, there are no characters especially forbidden.

9.11.3. Config Space Name

The config space name follows the OS-specific rules for directory naming.

Apart from that the name has to begin with a letter or underline ('_').

- the following character is especially forbidden: (':')

- any character which is not a letter or digit except for underline ('_').

9.11.4. Model Name

The model name follows the OS-specific rules for directory naming.

Apart from that the name has to begin with a letter or underline ('_').

- the following character is especially forbidden: (':')
- any character which is not a letter or digit except for underline ('_').

9.11.5. Revision Name

The revision name consists of

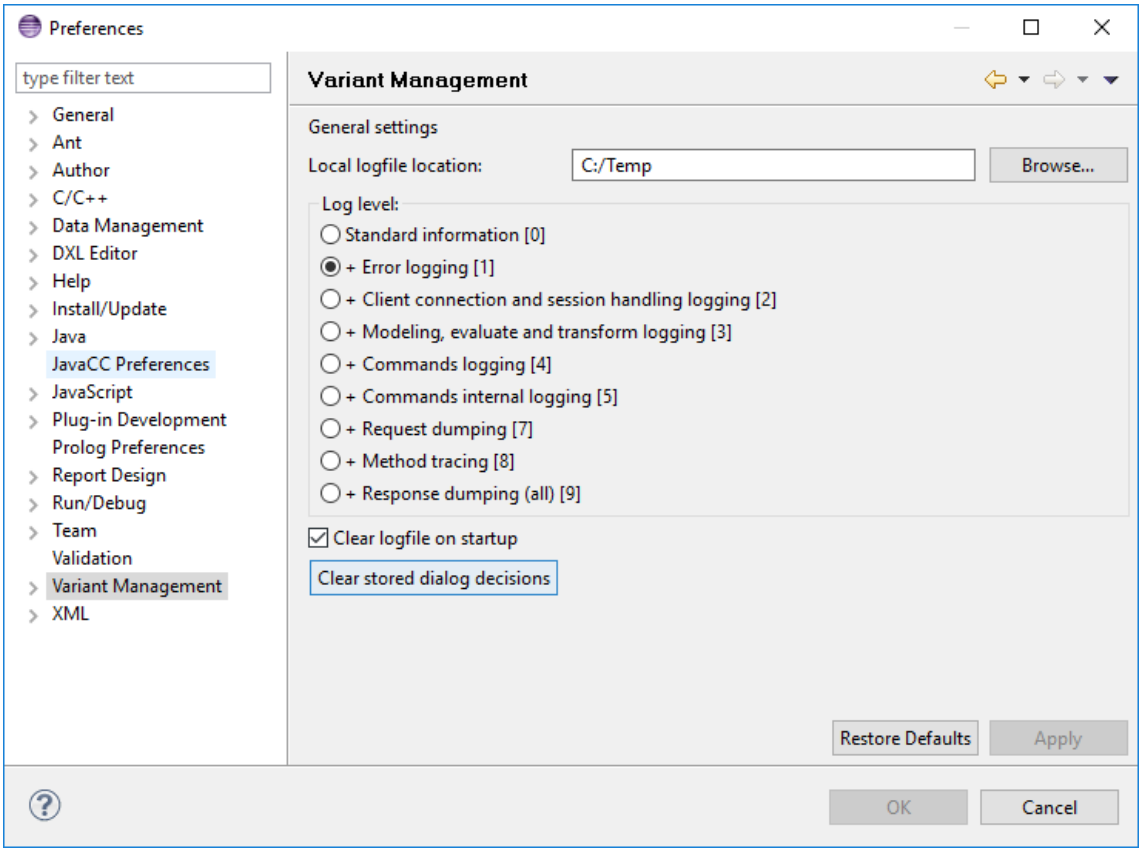
- non-ASCII characters or
- ASCII characters like digits, letters and the following: ? / - . _ ~ ! $ & ' ( ) * + =
- Following ASCII characters are especially forbidden: # , : ; @ | and space (' ')

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. Parametric Technology 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 syntaxt.
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 6.15, “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 .
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

G

H

I

K

L

M

N

O

P

R

S

T

U

V