pure-systems

Technical White Paper

Variant Management with pure::variants

pure-systems GmbH

Contents

1 Introduction

The pure::variants technology provides the most comprehensive and most flexible technology available for variant management. Features like complete tool support for all steps of the software development process, and excellent integration and adaptation capabilities permit to benefit from product line based software development in virtually all areas of software production. Especially the modules specifically designed for the needs of embedded software development allow a combination of maximal flexibility and resource efficiency as yet unreached.

2 pure::variants Technology

The basic technology pure::variants provides all characteristics necessary for a successful, toolsupported development and realization of product lines.

pure::variants is applicable **throughout all phases** of the development, starting from the analysis and the design, over the implementation up to the production, tests, use and maintenance.

All basic mechanisms are **independent of a concrete programming language**.

The integration of additional modules allows the flexible extension of pure::variants in order to be adaptable to specific requirements. Therefore, all modules have unrestricted access to the variant knowledge. This allows to realize **highly optimizing extension modules** that need a global overview of the overall system. With such extension modules, the production of custommade product variants can be adapted easily to the demands of the user concerning for example the use of computing time or space requirement.

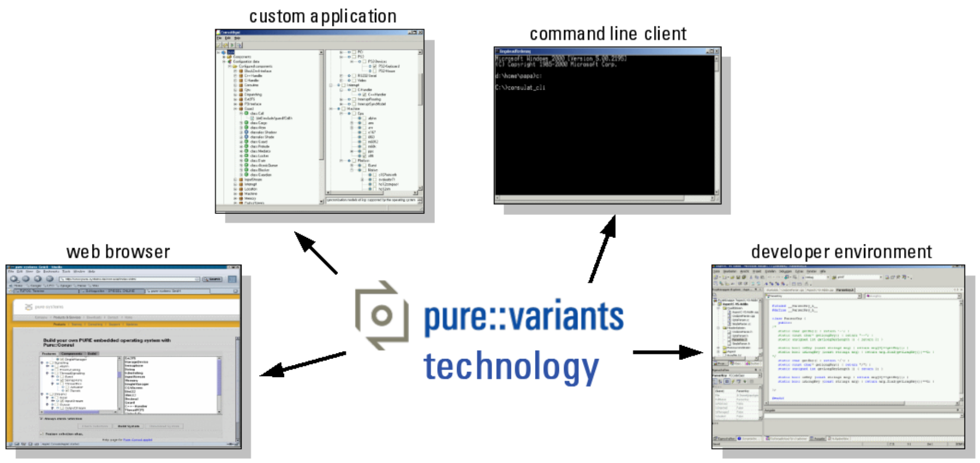

Figure 1: pure::variants integration possibilities

The **possibilities for the integration** of the pure::variants technology are outstanding. pure::variants can be integrated easily in existing development processes and environments (Figure 1). Joining pure::variants with tools for configuration management, requirements engineering, modeling and testing that are already employed by the users provides a **uniform solution**. The XML-based exchange format makes it possible to link pure::variants to arbitrary tools at any time. Due to the possibility to import existing software projects, the **initial expenditure is small**.

The flexible basic structure permits the administration and evaluation of all product-line relevant information, e.g. documentation, source code, dependencies, and test results. The user can chose among different views on this information. This allows to adapt the visualization of that information according to the organizational structure (developer, project manager, sales, customer).

The pure::variants technology is **unique** because of its generality, high integration ability and flexible expandability.

3 pure::variants-based Development with Variants

The use of the pure::variants-based tools (for products see Chapter 5) takes place in different phases of the software development process. The development of product lines is divided into two steps. First, the common assets are identified. In the second step the individual products are derived from the product line.

The principle operational sequence of such a development process is presented in the following, beginning with the identification of the common assets:

1. **Analysis of the problems and requirements:**

   Based on a content-wise problem analysis, feature models (see Section 4.1) are designed that capture the dependencies between the individual features of the product line. Feature models are easy to visualize and understand.

   Based on the experience of the practical use of feature models, the expressiveness of pure::variants feature models was extended substantially. The support of model hierarchies in particular enables different representations of the problems depending on the user (customer, developer, sales, . . .).

2. **Design of the solutions:**

   Starting from the feature model and other requirements for the software systems to be derived, the design of the software solution(s) is performed. The elements of the software solution with their relations, restrictions and requirements are integrated into the family model and hence are available for automatic processing.

   The family model is divided into several levels. The highest level is formed by the so-called components. Each component represents one or more functional features of the solutions and consists of logical parts of the software (classes, objects, functions, variables, documentation). In the next level, the physical elements of a solution are assigned to the logical elements. The physical elements are files that already exist as well as files that are to be created and actions that are to be performed based on the variant knowledge.

   Contrary to previous ideas from the range of feature model based techniques, the pure::variants technology **captures the problems (feature model) and the solutions (family**

model) separately and independently. This enables a **simplified re-use** of the solutions and of the feature models in new projects.

3. **Realization of the solutions:**

   Employing the possibilities of the selected programming language and tools, and using the additional possibilities of generating variants that are provided by **pure::variants** modules like AspectC++ or PatternTransformer, the solutions are realized.

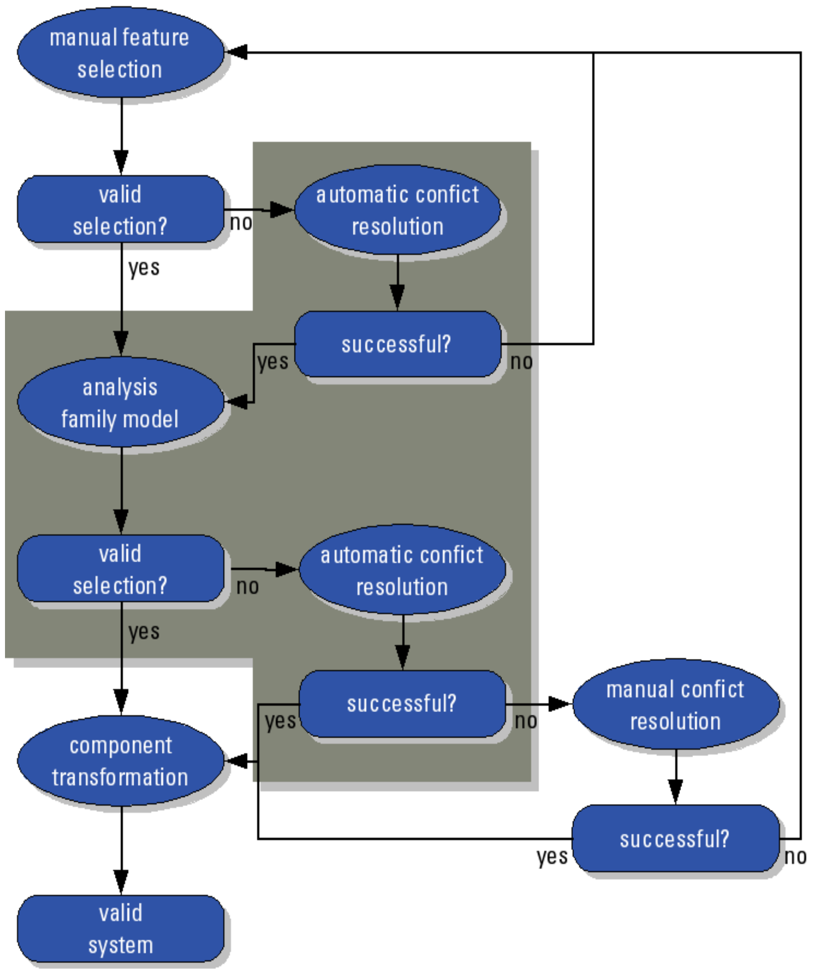

Figure 2: Employing pure::variants to create a problem solution

The **pure::variants** technology helps to collect and manage all models and information during the development process described above. Because it is based on the feature model technology, the resulting software families and product lines can be adapted easily. **pure::variants** uses this flexibility to support the automatic generation of custom-made solutions for concrete problems. Figure 2 illustrates the tool supported evaluation of feature models with **pure::variants** where the innovative aspects of that process are emphasized. The main steps of that process are the following:

1. **Determining a valid combination of features:**

   The user (customer, sales) selects the features relevant to the problem solution from the feature model. Using methods and tools from Artificial Intelligence, pure::variants checks whether the feature selection is valid (i.e. evaluates the dependency rules) and, if necessary, resolves dependency conflicts automatically. This ensures that even complex dependency structures can be efficiently transformed into a valid system.

   The result is a combination of features that describes the problem to solve as intended by the developers of the models.

2. **Selecting a suitable solution:**

   Based on the selected features, the family model is used to find a suitable solution. Using the information contained in the family model, the feature selection is analyzed by each component and its logical and physical elements. For this, methods of Artificial Intelligence are employed again. For each part it is decided whether and in which form it belongs to the solution. Problems that may arise due to further dependencies are resolved automatically if possible or handed over to the user for manual solution. The strategy for the selection of the best suitable solution is determined by the users. For this specific strategies can be realized if necessary, depending on user demands. Thereby it is possible to select a solution according to optimization parameters, e.g. according to the customer's cost model for parts of the component.

   The result is a description of the selected solution in form of a component description.

3. **Creating the solution:**

   The customized software solution is created by a transformation process controlled by the component description. During this process all necessary transformation modules are activated and perform the conversions specified in the component description.

Not only the creation of new product lines, but the integration of existing software (re-engineering) into a product line-based development in particular is easily possible with pure::variants. For this purpose the development process does not begin with the identification of the common assets but with the (partial) automatic production of the family model based on the already existing software and with the step-by-step construction of the associated feature model by the users. The remaining steps are then similar to the process outlined above.

4 Building Variants with pure::variants

The pure::variants tool chain was designed for the development and deployment of software product lines. It supports the management and creation of the different variants of such a product line. The tools are based on the different models that are used for the description of the problem domain of the product line, for the description of the implementation and for the selection of a specific product.

Feature models play a central role hereby. They allow a uniform representation of variabilities and commonalities of the products of the entire product line. Compared to other methods based on feature models, pure::variants uses an extended version of this concept. A detailed description is given in Section 4.1.

A specific implementation of the product line is described by the *family model* (Component Family Model - CFM). It enables the mapping of the user demands to the different component implementations. The adaptation of components to a certain application is a part of this model as well. As the name suggests, this model was developed especially for the pure::variants technology, since existing modeling techniques such as UML or SDL are not suitable for this. The CFM is described in detail in Section 4.2.

The *feature selection* is used to describe an individual product. It describes the product's properties and values associated with those properties, and is used to create the final product.

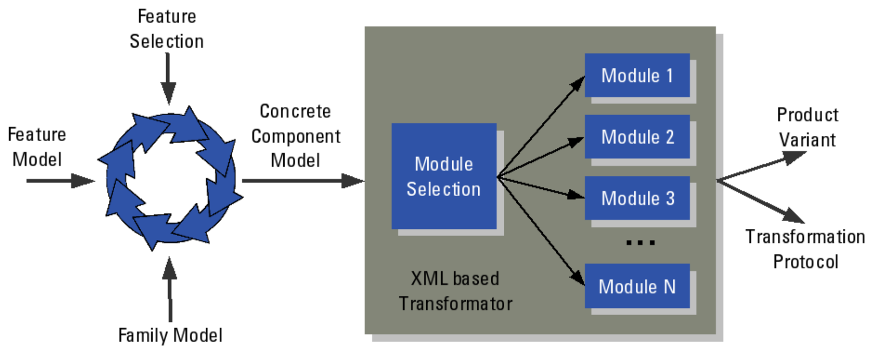

Figure 3: pure::variants transformation process

Figure 3 gives an overview of the basic process of creating variants with pure::variants. Once the different models are produces, the remaining steps are performed automatically. The developers of the product line are responsible for providing the feature model as well as the component description with CFM. The user then selects the features. Here, the user can be both a human and a tool that determines the necessary features automatically based on the application. Further processing is performed in two steps. At first pure::variants analyzes the different models. The result is a construction plan from which the customized component of the final product is derived in a second step.

The substantial difference between pure::variants and similar technologies is that the models used in pure::variants only describe what is to be done but not how it should be done. pure::variants provides only the necessary mechanisms. They can be easily extended by the user with his (own) modules according to his needs. Thus the pure::variants technology is neither fixed on a certain development process nor on a certain language or method and can be integrated into any development process. This high degree of flexibility is achieved by the combination of two very efficient languages within pure::variants.

Another important difference lies within the configuration process. New algorithms are used for the evaluation of the models and the composition of the transformation description. For this, the logical programming language Prolog, well known from Artificial Intelligence is used. Once the models are transformed into so-called Prolog facts, the selection is evaluated with respect to its validity and completeness. In the first step the correctness with respect to the feature model is examined. Then the additional requirements resulting from the component model are checked. Conflicts that result for example from implicit dependencies of features and components, are detected. The automatic resolution of such conflicts is based on heuristic procedures developed by pure-systems. Problems that cannot be solved automatically are indicated, and the configuration process is interrupted. If the selection is valid and complete, a description of the transformation in XML is generated that is then used for the creation process.

For the transformation process the XML based language XMLTrans was developed. Using this language, the adaptation and transformation process is described by defining the individual actions that have to be performed. Each action is triggered on the basis of the transformation description generated in the previous step. All actions are implemented by transformation modules. If the predefined transformations are not sufficient, new transformations can be created at any time and integrated by the users. In most cases a new transformation can be described directly in XMLTrans. Especially created modules are necessary if more complex actions are to be defined. To control the transformations, both XMLTrans and the modules can access the complete information within pure::variants (feature model, feature selection, CFM, sources, etc.). This enables transformations that are not possible with conventional technologies, because those techniques allow only limited information about individual elements of a software solution.

4.1 pure::variants Feature Models

The feature model contains the variabilities and commonalities of the product line. A feature represents a property of the system visible for the user. This is further characterized by the associated feature description. The effects of the selection of a feature must be recognizable from the feature description.

Within the model the individual features are represented by an acyclic graph. The nodes correspond to the features. The edges describe the basic relations (mandatory, optional, alternative, or) between the individual features. Figure 4 shows a simple feature model.

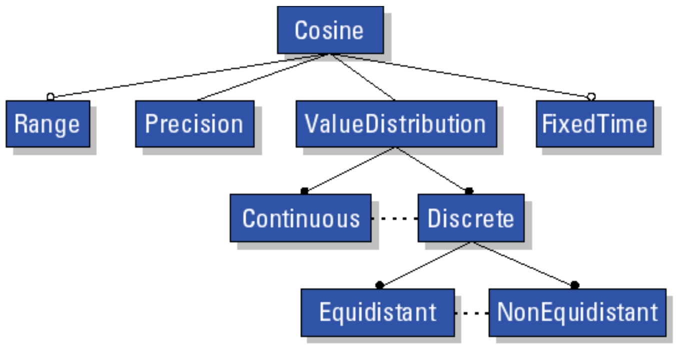

Figure 4: Feature Model

Features with value assignments

For the description of complex problems it is necessary that values of different types can be assigned to features. The pure::variants feature models supports this by a pair of type/values that can be assigned to each feature. The value assigned to a feature can be used both for the variant building process and during the transformation.

Features with restrictions

In addition to the basic relations described above, the pure::variants model allows the description of more complex restrictions between arbitrary features. They are expressed by logical rules and represent cross connections within the graph. Dependencies of features can be easily described by this flexible mechanism such as conflicts between feature combinations or the implicit inclusion of another feature.

Hierarchies of feature models

In contrast to other technologies based on feature models, pure::variants permits the use of an arbitrary number of feature models that are connected by mappings. All other known technologies use only one single feature model. The use of multiple feature models allows the consistent integration of different views on the variabilities and commonalities of a product line. Thus, for example models with different degree of detail are possible (e.g. customer: small degree of detail, sales: middle degree of detail, development: maximum degree of detail). Due to the hierarchical connection of the different models, the consistency of the feature combinations is maintained at any rate. The combination of different (parts of) product lines to a new product line can also be simplified by this. For this, a top-level model is added to the existing hierarchy that provides a mapping to the existing models of the product lines that are to be integrated.

4.2 pure::variants Family Model

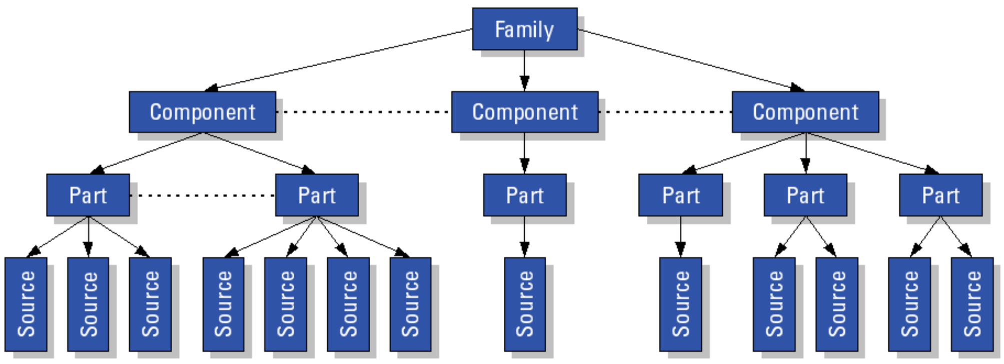

Figure 5: Family Model

The pure::variants component model (Component Family Model) describes the internal structure of the individual components of a product line and their dependencies on the features. Using this description the different components are bound to the features. The term component is used in a broad sense in this context. In the sense of **pure::variants** a component encapsulates a set of configurable functionalities. Figure 5 gives an overview of the hierarchical structure of the component model.

In the following section the elements of the CFM are briefly presented:

Component: The basic elements of the model are the components. Each component has a unique name and consist of at least one part that is further defined by its source elements.

A component encapsulates one or more functionalities that realize features from the feature model of the product line.

Part: A part is a named and typed element that belongs to exactly one component. It is defined by at least one source element.

Parts can be logical elements of a programming language, e.g. classes, functions or objects. But any other key element of the internal or external structure of a component can be described by a part as well. These are for example interfaces, specifications and graphics. **pure::variants** provides a number of predefined types (class, object, flag, classalias, variable). The introduction of other types according to the needs of the user is possible at any time by user-defined extensions.

Source (source element): The physical representation of a component is described by its source(s). A source element is a typed entry. The type is used to determine which transformation modules must be selected during the transformation to create the part. A simple action of a transformation module is for instance the copying of a file to a specified destination. It is also possible to generate new files as is used by the transformation modules for the creation of a "classalias" or a "flag". Arbitrary new transformation modules with own actions can be integrated by the introduction of a new source element type.

Restrictions can be specified for all elements of the component model. The restrictions are used to connect a component, part or source to a feature or a combination of features. With this it is possible to express specific characteristics of the implementation (e.g. the incompatibility of two components in certain feature combinations).

4.3 Modules

The possibilities offered by the **pure::variants** technology can be flexibly extended by different modules. They can be used for different purposes. On the one hand, transformation modules extend the executable actions during the creation of the product. On the other hand, integrating modules provide links to external processes and tools.

The following areas are already covered by such modules:

Modelling and Programming Languages

Standard Transformation: Provides typical family model elements and transformation patterns for commonly used programming languages C/C++ and Java. The Standard Transformation can be extended with customer/project specific transformations.

Connector for Simulink: Enables configuration and management of variabilities for Matlab/Simulink models in family models.

Integration into Development Processes

Transformer für Software Configuration Management Systems: The Transformer links existing configuration management system into the transformation process. The integration of variant management and configuration management in a single solution enables an efficient collaboration during development and deployment of complex systems in larger teams.

Synchronizer for Requirements Engineering: Classic requirements management tools are extend with the ability to manage variability information in feature models. **pure::variants** feature models are directly link with requirements in the external requirements management tool. By this variants of requirement documents can be created quickly. The created feature models can be used in the later steps of the software development for system configuration purposes.

5 Products

Different products for different employment scenarios are derived from the basic technology.

5.1 Developer Edition

The **pure::variants** **Developer Edition** supports the software developers during design, development and use of product lines. It enables the users to use the **pure::variants** technology for the realization of their own software projects. It is available as an independent tool or as a Plug-In for smooth integration into integrated development environments (e.g. Eclipse or Rational Software Architect). The Developer Edition complements other development tools such as CASE Tools, editors, compilers and debuggers.

5.2 Integration Edition

The **pure::variants** **Integration Edition** enables the integration of the **pure::variants** technology into customer products. An example could be the configuration of an embedded operating system by a customer with a configuration utility that is provided by the manufacturer of that operating system based one the Integration Edition.

By using the Integration Edition, customers do not need to develop the variant management all by themselves. Due to the Integration Edition's universal design, all necessary basic functions are already present and tested. Only the user interface and the transformation process has to be adapted. This results in advantages in quality and shorter development times.

5.3 Server Edition

The **pure::variants** **Server Edition** enables a centralized management of software product lines, for example in the development department. It also allows the sale of customized configurations over the Internet without the necessity of a local installation of pure::variants at the developer/customer. A substantial benefit of the use of the Servers Edition is the centralized management of the variants in an organization similar to software configuration management that is usually centralized as well. The central maintenance results in reduced personnel costs. Basing sales on the Server Edition allows the creation of customized solutions by the customers themselves without risking disclosure of proprietary knowledge about the overall structure of the software solution. This may result in reduced costs on both sides. On the one hand, the customer can determine exactly what he really needs (and has to pay for). On the other hand, the use of a server solution is rather economical.

5.4 Extension Modules

The pure::variants **Modules** extend the different pure::variants editions by additional functionalities. The modules cover different areas and tasks of the software development processes. Examples for modules are the integration of a software configuration management solution for version management or the extension of different programming languages to enhance support for creating variants.

6 Conclusion

Innovation cycles that become shorter and shorter rise the requirements of today's software systems concerning their longevity more and more. Substantial basis for long-lived software systems is their adaptability to unexpected or not anticipated (customer) requirements. A promising solution are product lines that support software development not only for one product but for a class of products. Because of the initial identification of common assets, synergies can be exploited that do no come to fruition in classical technologies of software design. For a successful use of software product lines, tools are necessary that support the entire process beginning with the design up to the deployment. The efficient management and realization of building variants within the common assets represents a particular technological challenge. The pure-systems GmbH supports this process with its variant management tools based on the pure::variants technology.