Master thesis

varBPM: A product line for creating business process model variants

Andreas Daniel Sinnhofer, BSc

Institut für Technische Informatik
Technische Universität Graz

Betreuer: Dipl.-Ing. Dr. techn. Christian Kreiner
Vorstand: Univ.-Prof. Dipl.-Inform. Dr.sc.ETH Kay Uwe Römer

26. Mai 2014, Graz

Kurzfassung

In der heutigen Industrie steigt ständig das Bedürfnis nach schnelleren Produktionszeiten bei gleichzeitiger Reduktion der Kosten. Um diesen wachsenden Bedürfnissen gerecht zu werden, werden üblicherweise Tools zur Modellierung von Geschäftsprozessen verwendet, mit denen die einzelnen Abläufe modelliert, dokumentiert und optimiert werden können.

Da sich oftmals zwischen den unterschiedlichen Geschäftsprozessen nur wenig ändert, werden neue Prozesse meistens durch kopieren bzw. klonen von bisherigen Lösungen gebildet. Das bedeutet, dass bei einer Veränderung einer dieser Vorlagen alle betroffenen Prozesse händisch verbessert werden müssen, was zu einem erheblichen Arbeitsaufwand führen kann. Dies führt in weiterer Folge auch dazu, dass eine Konfigurierung des Enterprise-Resource-Planning-Systems (ERP) mit hohem Aufwand und meistens hohen Kosten verbunden ist.

Ziel dieser Arbeit ist es, diesen Modellierungsprozess mit modernen Mitteln aus der Domänenmodellierung zu unterstützen, um eine rasche und konsistente Art der Entwicklung und Optimierung basierend auf systematischer Wiederverwendung zu gewährleisten. Das Erstellen eines kompletten Geschäftsprozesses soll nach dem 'Baukasten' Prinzip funktionieren, in dem Prozesse durch Auswahl bestimmter Features automatisch zusammengesetzt werden. Darüber hinaus soll es möglich sein, diese Features automatisiert erfassen zu können und aktuell zu halten um eine zeit- und kostenintensive Nachbearbeitung der Modelle zu vermeiden.

Stichwörter: Software Product Lines, Geschäftsprozesse, Tool-Vernetzung, Domänenmodellierung

Abstract

In the today's industry the need for faster production cycles and simultaneously decreasing production cost is steadily increasing. To solve this issue tools for modelling business processes are typically used to model, document and optimize the process steps for a specific workflow.

In most cases different business processes do only vary in some points which lead to the situation that new business processes are formed through copy or clone of previous solutions. This means that changes to a template affects many processes, where all of them need to be manually updated, which can lead to a considerable amount of work for a bigger company. Furthermore this can also result in higher costs to configure an Enterprise-Resource-Planning-System (ERP).

The goal of this thesis is to optimize the modelling process of such processes with modern techniques from domain modelling approaches - like feature oriented domain modelling approaches - to ensure a quick and consistent way of developing and optimizing this processes based on systematic reuse.
The creation of such a process should be based on a 'building block' principle, which means that business processes are automatically assembled by selecting a set of specific features. Furthermore it should be possible that this set of features is automatically updated if a domain expert designs a new feature or updates an old one. This should ensure that maintaining business processes and the according models are neither time nor cost intensive.

Keywords: Software product lines, business processes, tool integration, domain modelling

Statutory Declaration

Acknowledgements

This master thesis was kindly sponsored by Magna Cosma and was progressed at the Institute for Technical Informatics at Graz University of Technology.
I would like to thank Roman Zernig (Magna Cosma) and Peter Pühringer for their encouragement and their support. Further I want to thank my supervisor Christian Kreiner, who always had an open ear for me and with whom I had many interesting discussions on this topic. Special thanks goes to Christopher Schaffert from Intellior AG and Danilo Beuche from pure::systems.
Last but not least I want to thank my parents who enabled my education and my studies, my brother who has always supported me and Inge Siegl.

Graz, May 2014 Andreas Daniel Sinnhofer

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background and Motivation

Magna Cosma is an international company in the metal stamping and assembly industry - specialised on class-A car body panels and closure parts (e.g. doors) - with several plants all over the globe. The implemented business processes are mostly controlled by an SAP infrastructure. Although some plants are specialised on the same production parts, almost every plant develops and maintains their own business processes, which makes it really difficult to compare processes, mark bottlenecks, optimise the processes and publish the changes to other plants.

Another big issue is the documentation of the processes which is often out dated or fragmentary which means that the interchange of processes between plants is almost impossible without any guidance of domain experts, which is then a time and money consuming task. Furthermore it is a big problem if the former developer of a process is leaving the company, leaving a hole of knowledge behind which is costly to fill.

The attempt to integrate the modelling process of business processes into a software product line (SPL) seems at first sight a little strange, since a SPL is intended to systematically reuse software artefacts to automatically build variants of a similar software product, but at second glance the similarity is clear: In this particular situation, the SPL is used to systematically reuse expert knowledge in form of business process artefacts to automatically ensemble variants of a similar business process.
Through the integration of company specific extensions it is also possible to establish a generic way to develop and publish business processes throughout the whole company landscape, meaning that each partner benefits of the knowledge of all domain experts around the globe, ensuring that all implemented business processes are effective and gainful. I.e. a form of knowledge management is established throughout the whole company landscape.

1.2 Disposition

This thesis is structured in the following way:

- **Chapter 2:** Gives an overview of the research activities, covering the topics of software product lines, domain specific languages, business process management and tool integration. This chapter also covers the topic of tool evaluation for a feature oriented modelling tool and a business process modelling tool.

- **Chapter 3:** Gives an overview over the structure of this project and addresses the composition of the solution space concerning the developed type model and construction rules.

- **Chapter 4:** Deals with the implementation of the pure::variants plug-in and gives a rough idea on how the functionality could be extended.

- **Chapter 5:** Deals with the usage of the developed plug-in in combination with the BPM Tool to show the validity of this approach, highlighting some design rules and steps to be taken to ensure that this approach works best.

- **Chapter 6:** Contains a short list of further projects that could be implemented based on this work.

2 Related work

2.1 Software product lines

A SPL can be seen as a set of domain features, which are automatically assembled and configured to a whole software project just by choosing the wanted features. Instead of writing code for a whole system, the developer divides the project into small lightweight features which are implemented in so called domain artefacts. These artefacts can then be easily changed and assembled to a whole system. In classical software engineering, there is also a kind of 'SPL-thinking'. For example: If one looks at the strategy pattern (or policy pattern), where you can choose a specific behaviour of an object at runtime or compile time, making a class more reusable than the classical approach.
I have used the term 'feature', but what exactly is a feature? A definition of a feature depends strongly on the according domain and is sometimes really hard to define, but crucial to define, because the success of a SPL depends strongly on a good designed feature model (and according to this, a good designed variability model). I think Danilo Beuche defined the term in an easy to understand way as *a property of a system/component/group of components which is relevant for a specific Stakeholder.*
A simple example for features and their relationships is given in chapter 2.1.2.1.

2.1.1 Motivation

A well written discussion of product lines their motivation and their benefits can be found in [PBL05]. The key points have been identified to:

- **Development Costs:** This benefit is somehow a mixed blessing, because the initial costs of a software product line engineering (SPLE) are much higher than in classical software engineering due to the fact, that the analysis and design phase are much longer. But as stated out in [PBL05], adopting a SPL to another customer costs way less effort and money, because only new features has to be modelled and implemented instead of a whole system. According to the literature, the break-even point lays by about three systems.

- **Time to market**: As pointed out in the above benefit, it costs way less time to adopt an existing SPL to another customer, but there is also the same issue: The initial development time and respectively the initial time to market is higher than in classical software engineering.

- **Quality and Maintenance:** This is one of the main advantages of a SPLE. The product quality is much higher due to the fact that the software is tested and running on many different kind of systems. If one bug is detected, the according feature has to be modified and the patch can be easily delivered to all customers.

- **Customer Satisfaction:** Another very big advantage is that the customer gets a well suited product for his wishes, which is highly maintainable and expandable. He gets exactly what he wants and he only pays for the features he wants. And due to the fact that there are always enhancements to develop over the time, a satisfied customer will always return to your company.

2.1.2 Phases of development

According to literature ([PBL05] and [WL99]), the software product line engineering can be split up into two main parts, the ***Domain Engineering*** and the ***Application Engineering***.
The domain engineering is the procedure for defining the components, the variabilities and the commonalities of the product line and the application engineering is the procedure where the application itself is build, using some domain artefacts which were created in the domain engineering.

Each of these two processes can again be split up into four phases:

- Requirements engineering
- Design
- Realisation
- Testing

And for completing the SPLE framework some kind of project management has to control the domain engineering procedure in an economic way, defining what is inside the domain and what is outside of the domain. For illustration, the complete framework can be seen in Figure 2.1.

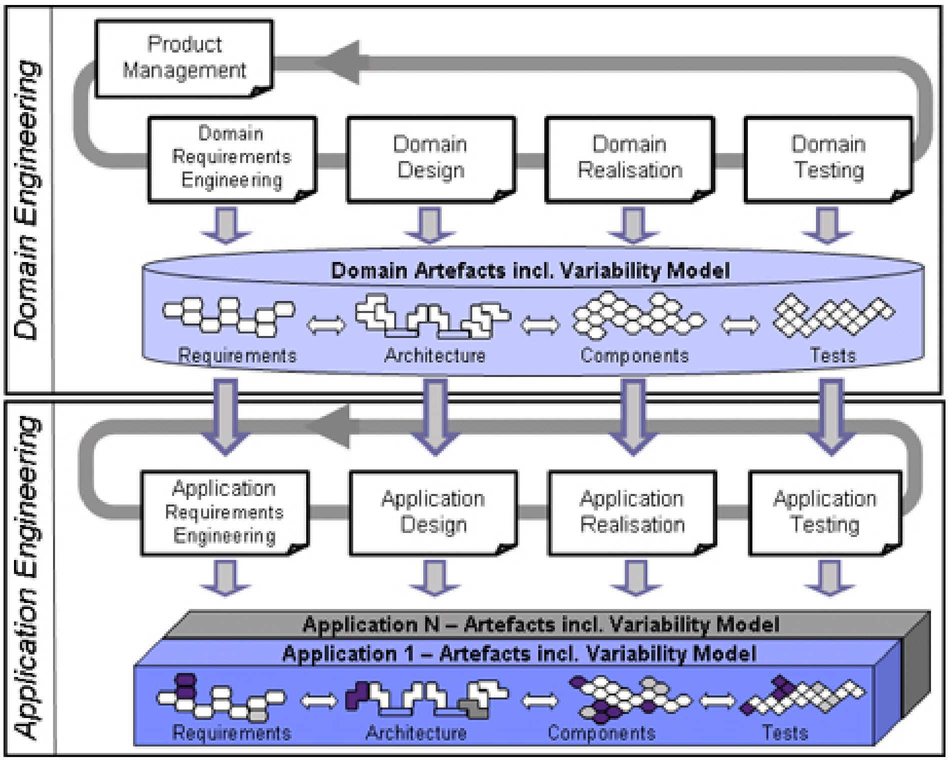


Figure 2.1: The software product line engineering framework [PBL05]

It should also be mentioned, that creating and evolving such a system is an iterative process, which is indicated in the picture with the loop back arrow.

2.1.2.1 Domain Engineering

Additional to the main goals - which where stated out before - there are some more: The set of applications, for what the product line should work, is defined and the reusable artefacts are created.

The Domain requirement phase is for defining the domain with all its components (mainly features and variation points). The heart of this analysis is the so called "Feature-oriented domain analysis" (FODA [KCH+90]). The purpose of such an analysis is to define the set of possible features and to state the possible relationships between them. There are five possible relationships between features. The 'mandatory', the 'or', the 'alternative', the 'optional' and the 'excluding' relation.

- **mandatory**: Adding a feature, adds also all its mandatory features
- **or**: These features are orthogonal to each other and at least one of it must be chosen (1 to n possible selections)
- **alternative**: Exactly one feature of a list of features must be chosen (1 out of n)
- **optional**: Somehow like the or relation, but also no feature is possible (0 to n possible selections)
- **excluding**: If one feature is included, some other features can't be chosen because their behaviour/influence on the system is incompatible

An example is shown in the Figure below.

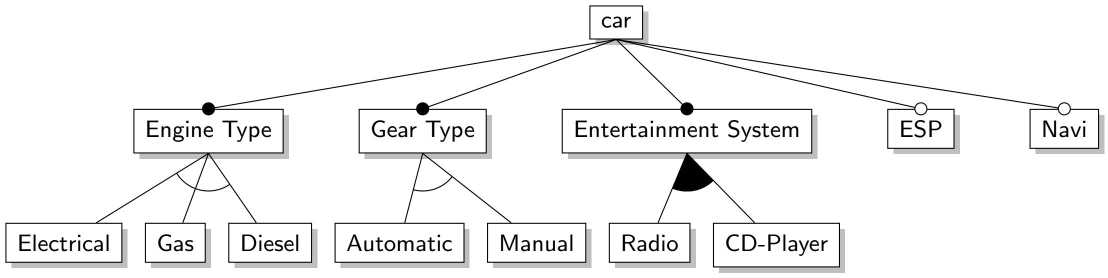


Figure 2.2: Example feature model

In this example, a car is modelled with five features whereas the "Engine Type", the "Gear Type" and the "Entertainment System" are mandatory features and the "ESP" and the "Navi" are optional features. The "Engine Type" feature is than again modelled as a set of alternative features. The same applies for the "Gear Type" feature. The "Entertainment System" feature is modelled with two additional features which are "or" related.
The result of this whole phase is the so called ***Variability Model***, which consists of feature diagrams and textual and/or model-based requirements. This deals as input for the next phase.

The domain design phase is for defining a generic reference architecture and to create a refined variability model. The reference architecture states explicitly what components are variable and which of these should be realised in the realisation phase. The architecture itself consists of different kind of views to get a more structured perspective for all kind of stakeholders. According to [PBL05] there are four important views:

- Logical View

- Development View
- Process View
- Code View

The logical view contains the requirements model, the development view deals with the software components and their interfaces, the process view states the activities during the execution of the system and the code view states the project structure in forms of source files and where they are located.

The domain realisation phase deals with the implementation of reusable artefacts (software components). Beside this output, this phase also provides a updated design and a detailed documentation on the developed components. Implementing these components differs from a traditional software engineering, due to the fact that the result consists of loosely coupled components, with an interface that should work in many contexts. Exactly this interface is the hardest part when designing and implementing such a system.

The domain testing phase is a very important phase, where all artefacts (design artefacts and software components) are tested. It is important to know that these components are only tested for their self without running in an application. Only the interface and the behaviour of each component is tested and validated. Testing these components in an running application is part of the application engineering.

2.1.2.2 Application Engineering

The main goal of the application engineering is to develop an application based on the highly reusable domain artefacts from the domain engineering.

During the application requirements phase, a requirements specification based on the domain requirements and the product feature roadmap of the according application is defined. It is also necessary to detect and evaluate the differences between the requirements of the customer and the available capabilities of the SPL. This is a very important phase of development, because the benefit (the degree of reuse) strongly depends on good application requirements.

During the application design phase, the application architecture is defined by using and configuring the reference architecture from the domain engineering and by considering some application depending adaptations. For each of the adaptations a cost-benefit analysis should be made and if the implementation effort (time and money) is too high, these changes should be rejected (of course with the agreement of the customer).

During the application realisation phase, the whole application is created. This phase differs a lot from the classical approach, due to the fact that many parts (the domain artefacts and the interfaces between them) are already implemented. It takes less time to build the whole application from them. Only new or modified features have to be implemented from scratch.

During the test phase, the complete application is tested. It is important that really all components are well tested (the reused components and the new ones) and the results of these tests can also affect the domain artefacts if a bug is detected.

2.1.3 Software product lines in action

During the software product line conference - which is held once a year - a famous and outstanding software product line is chosen and is put in the so called "product line hall of fame" which can be found here:

`http://splc.net/fame.html`

Some of the famous projects are for example the Bosch gasoline systems, Nokias Mobile Phones or Philips product line of software for television sets.

2.2 Domain specific languages

DSLs are languages which are custom made for a specific application domain. One of the most famous domain specific language is UML, where the application domain is the specification of software components (components, constraints, activities, ...) in a software project. A DSL can vary from a very small scope, which is locked to some few applications, or a very general scope with thousands of applications. For example a GUI for choosing and combining feature artefacts in a SPL has a rather small scope compared to a hardware description language such as VHDL.

2.2.1 Motivation

The key benefits are:

- **Ease of use:** Due to the fact, that a DSL is designed to only fit into one specific domain, it is very easy to learn this kind of language and to develop a new application out of it. In some cases it is even unnecessary to learn a new kind of programming language, because the DSL is intuitive and straight forward to use.
- **Time to market**: Generally it costs less time to develop a new application from a DSL, than using a general purpose language, because the language is well suited for the specific application domain (the needed number of LOCs decreases).
- **Quality and Maintenance:** This benefit is very similar to the SPLE benefit. Due to the fact that a DSL has less components to maintain (compared to a general purpose language), it costs less effort to maintain it and also the quality is in general higher.

2.2.2 Phases of development

According to [MHS05] there are four phases of development:

- Analysis
- Design
- Implementation and testing
- Deployment

Additionally to this, a ”decision” phase is mentioned, for deciding if an existing DSL should be used, a complete new one should be developed, or if a general purpose language should be used, a table of decision patterns and their description can also be found in [MHS05].

During the analysis phase, the domain is analysed using one of the domain analyse tools such as the previous mentioned ”feature oriented domain analysis” ([KCH$^+$90]) or any other tool which can be found in the literature ([FPDF98], [TTC95] or [SA98]). The output of this phase is a graphically diagram, which represents all connections and relations in the specific domain.

During the design phase, the DSL is designed with the help of the diagram of the analysis phase. Potentially a designer often tend to over-design a application, but for a DSL it is crucial to define exactly (and only) the required parts, because the more overloaded a DSL is, the less is the gained benefit. Also the way how the DSL should be created should be considered in this phase. One possible solution is to use a GUI-Tool such as MetaEdit+, or another possibility is to use an embedded approach, which means that an existing general purpose language is extended with specific abstract types and generic functions (e.g. by using template metaprogramming with C++). It is also possible to use an existing DSL and to extend it with the additional wished features (in literature it is often called the piggyback-approach).

The rest of the phases are straight forward: According to the specification, the DSL must be implemented and should be well tested so that it is safe to use and finally it is deployed so that real products and systems can be developed with it.

2.3 Business processes management

As found in the literature (e.g. [Kev01], [Mic93], [Pet07]) business process oriented companies perform better than (classically) businesses with a hierarchical or pyramid approach. The aim of such business processes is to be highly flexible to the demands of the market, to deliver high quality for low costs and to be fast in the production. To achieve these goals a form of management is needed, where processes are modelled, analysed and optimized.
The most popular definition of the term business process is defined by [Mic93] as: "*a collection of activities that takes one or more kinds of input and creates an output that is of value to the customer*".

In other words spoken, a business process is a goal oriented structure, where a sequence of tasks are executed and which produces an output that satisfies a customers need ([Gad08]). Optimizing a process is done by adjusting/reassembling the tasks in such a way, that the production costs of the output is as low as possible while the quality is as high as needed so that the customer is a satisfied one.

A business process management is often coupled with a workfow management to monitor the correct execution of a process in each process step. This approach can be seen in Figure 2.3.

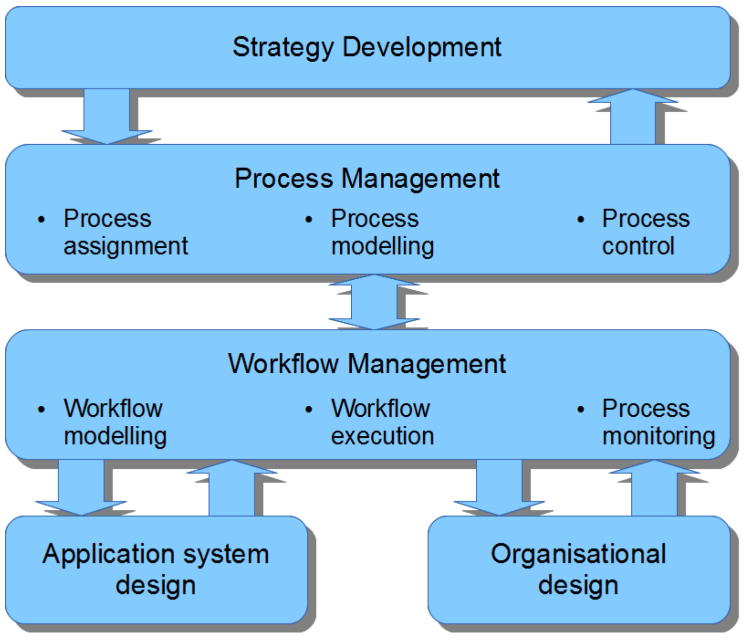


Figure 2.3: Integrated business process and workflow management (translated version from [Gad08])

Due to the supporting role of the workflow management, such systems are often called ”business process management systems” ([Gad08]).

2.3.1 Process types

As identified by [Gad08] there are three different kind of business processes:

Control process (management processes): These are processes which take the responsibility of the interaction of all other processes. For example processes which handle the business strategy development, the operational management or the operative planing.

Core process (primary processes): These are processes with a high adding value and are critical for the ability to compete. These processes covers the areas from covering customer wishes, production, and delivery or service provision.

Support process (cross section processes): These are processes with a little adding value and are typically not critical for the ability to compete. Typical examples would be (cost) accounting, reporting system or human resource management.

2.3.2 Workflow

A workflow is defined by [Öst95] as a refined business process. Based on the process design on the macroscopic level the process is split up into sub-processes until the microscopic level is reach. The microscopic level is reached when all tasks in these sub-processes are detailed enough, so that the process employees can use it as work instructions. The sequence control can be either done by an executive manager or an automatic electronic system.

2.3.3 Business process modelling

In this thesis the business process modelling is seen from an analytical point of few, where business processes are modelled, analysed, optimized and eventually simulated (depends on the fact if a workflow management is used or not). As stated by [Cur92] a business process must provide information for four different categories:

- **Functional**: Describing what is going to be done

- **Behavioural**: Describing when and how the goal is achieved
- **Organisational**: Describing the where and by whom is the goal reached
- **Informational**: Describing the structure and the relationships among the data.

According to [Koc11] there exist six principles for a methodical modelling:

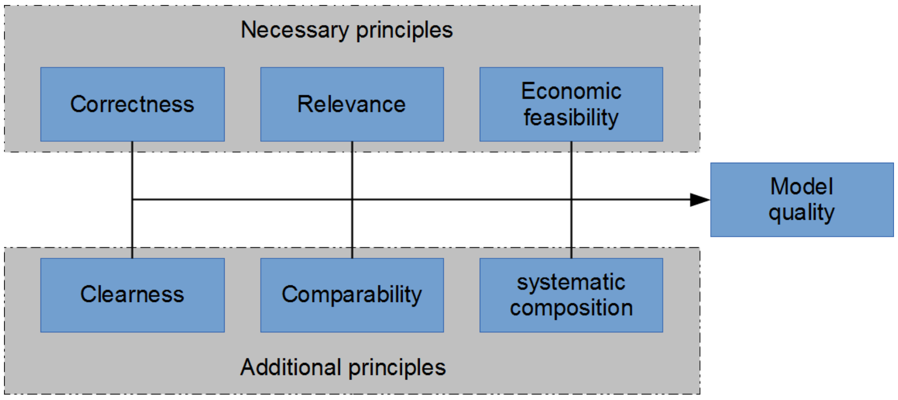


Figure 2.4: The principles for a methodical business process modelling (adapted form [Koc11])

1. **The principle of correctness**: This principle claims that the representation of the real process in a model contains all fundamental parts. This means that all involved workers/persons can derive all activities they need to do, to execute the process.

2. **The principle of relevance**: Due to the fact that a model of the "real world" can never be complete the principle of relevance was introduced to reduce the scope of the model to the according purpose. This means that a model of a process should always be concentrated on the according domain and should not try to satisfy all point of views.

3. **The principle of economic feasibility**: It should be interpreted in such a way, that the optimum level of detail is reach when all goals can be met. This means that further effort in the process does only cost money and doesn't really increase the knowledge about the process. The principle also claims that not each model needs to be developed from scratch.

4. **The principle of clearness**: This principle requires that a model is readable and coherent. This means that a model is as easy as possible and only as complex as

required. The rule of thumb says that a process model should fit into one A4 page. If the model is bigger, it should be split up into sub processes.

5. **The principle of comparability**: The aim of this principle is that models which were created with different modelling techniques can be compared to each other. This is important if different departments in a company design processes in a different way. Although this situation should be avoided, it is found very often.

6. **The principle of systematic composition**: This principle aims for the consistency of models which are developed for different views (e.g. an organisational view, a data flow view, a functional view, etc.. This means for example that processes only references data which is really present in the data model, or processes only uses organisational units which are defined in the organisational model. The principle of systematic composition ensures that each sub-model is integrated into an overall management concept so that the consistency is guaranteed.

As found in the literature, there exists a various number of modelling techniques and paradigms but since our project partner demanded a BPMN tool, only this method is further discussed.

BPMN: Business Process Modelling Notation

Sometimes also called as Business Process Model Notation, is a modelling notation for the representation of business processes and was developed by an IBM-employee (Stephan A. White) back in 2001. It was released by the Business Process Management Initiative (BPMI) in 2004 and is since June 2005 part of the Object Management Group (OMG). Since Juli 2013 it is also an ISO/IEC standard.
The focus is based on the graphically representation of the business processes but the standard also provides a direct mapping to some execution languages so that the process model can be executed. It is designed to be understandable by all kind of business stakeholders.
In version 2.0 there exists eight groups of elements[1]:

- **Activities**: Blocks which defines steps that must be executed before the control-flow can continue.

- **Conversation**: Indicates the exchange of a set of logically related messages

- **Gateways**: Splits/merges the control-flow according to some rules (e.g. exclusive gateway: according to a condition only one branch is executed. Similar to an

[1]for a complete overview see www.bpmb.de/images/BPMN2_0_Poster_DE.pdf

if-statement in computer science)

- **Choreographies**: Represents an interaction (e.g. message exchange) between two participants.
- **Swimlanes**: Represents responsibility for activities in a process
- **Events**: The standard defines a various number of events such as timer (cyclic events, timeouts, ...), errors, cancel events, etc.
- **Data**: Additional artefacts that can be linked to any of the elements.
- **Links**: Symbolizing the control flow (the sequence of execution) or relations between elements.

2.4 Tool integration

Tool integration has been identified (as stated in [NIS93]) as a crucial matter for computer based engineering processes especially for model-based development ([Gab03]). A lot of these solutions are based on an architectural approach like the one presented in [Kar99] and particularly two design patterns have been developed from [Gab03] for systems, where a model transformation is one of the key requirements.

2.4.1 Integration based on Integrated Models

The first pattern is based on the idea that all tools which should cooperate together shares a common data model for data exchange. The concept of it can be seen in Figure 2.5. The communication between the integrated models is based on any kind of middleware (e.g. CORBA is widely used). Between these tools a so called 'Integrated Model Server' is used to store the published data in a repository.

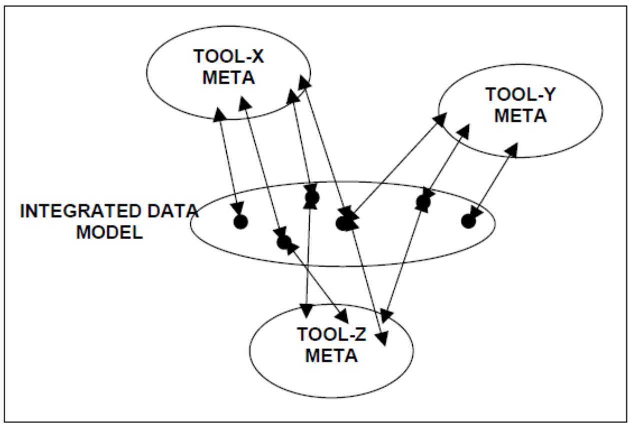


Figure 2.5: Concept of the integrated data model [Gab03]

For each tool a translator needs to be implemented, which translates the data model of the tool into the common meta-model and vice versa. So if a tool wants to publish its data, it first transforms the model into the common data model and sends it to the model server. If a tool wants to read the data, the data from the server is read and transformed into the tool specific data model. It is obvious that this approach works best if all tools have a significant similarity in the used meta-models.

2.4.2 Integration based on Process Flows

The idea of the second pattern is to implement a point to point connection between the tools, whereas the communication is message based via a backplane system. The concept of it can be seen in Figure 2.6. The backplane must be initialised with all needed meta-models, translators and workflows. A workflow contains the information about which tool publishes what kind of data or model, which tools are subscribed to this information and how these tools are sequenced.

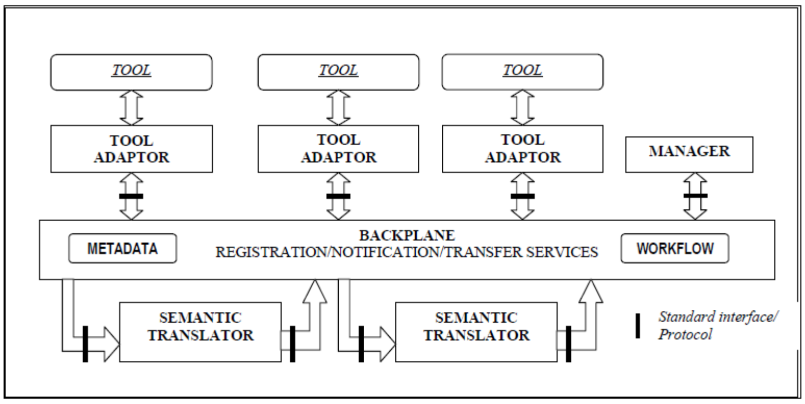


Figure 2.6: Concept of the process flow based tool integration [Gab03]

Whenever a tool wants to publish a model or data, it sends it to the backplane. The backplane then determines if a tool (or tools) is subscribed to this kind of information and if so, the data or model is transformed and the destination tool is informed that new information is available for it. The manager is used to configure the backplane and to monitor the processes. In contrast to the first pattern, typically fewer translators need to be implemented because the tools are pairwise integrated.

2.5 Tool evaluation

The most common modelling paradigms for domain modelling are feature oriented domain modelling (FODM) and domain specific modelling (DSM). It is very crucial to choose the right paradigm for the according domain, but in many cases it is very difficult to choose. For this work MADMAPS was used to determine the best suited modelling paradigm.

2.5.1 MADMAPS

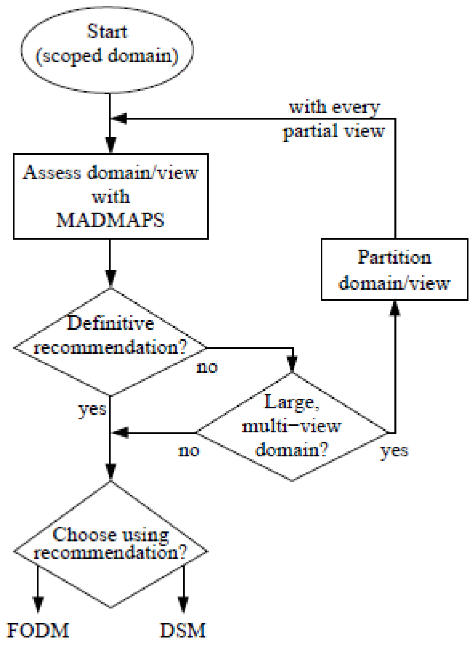


Figure 2.7: Flowchart for the domain modelling paradigm selection using MADMAPS [LWK12]

MADMAPS [LWK12] (stands for multi-attribute domain modelling approach for paradigm selection) is a tool for choosing a modelling paradigm for domain modelling. The result of such an analysis is a recommendation for a specific paradigm or the recommendation to split up the domain into some subdomains and to reapply the algorithm on these sub-domains. The flowchart of the paradigm selection with MADMAPS is shown in Figure 2.7.

For the paradigm selection, there are four different kinds of criteria and according to these criteria four different kinds of questions has to be answered. These questions are easily answerable during the early design phase of a project. The criteria are:

criterion	DSM	FODM
C1 Fixed relations $\geqslant$ variable relations	4	17
C2 Several instances of elements	31	4
C3 Different binding times/views	8	15
C4 Domain model used by non-expert	8	15

Table 1: Criteria C1 - C4 of domain modeling alternatives

The according questions and the weighting factors are:

- **Q1:** Are there more fixed relations than variable relations?
- **Q2:** Should it be possible to use several instances of an element?
- **Q3:** Should there be more than one binding time or more than one view in the domain representation?
- **Q4:** Should the domain model be used by a customer who is not a domain expert?

description	weighting factor ω
strongly disagree	-2
disagree	-1
neither agree nor disagree	0
agree	1
strongly agree	2

Table 2: Criteria weighting factors for domain modeling

The result utility value can then be calculated using the following equation:

$$u = \sum_{i=1}^{4} \omega_i \cdot C_i$$

For this project, the resulting utility values evaluates to:

Question	DSM	FODM
Q1	4	17
Q2	-31	-4
Q3	8	15
Q4	16	30
Result	-3	58

Table 3: Resulting utility factors

As result, a FODM paradigm should be used for this project to model the domain. In further only tools which support FODM are taken into account.

2.5.2 FODM: Tool candidates

2.5.2.1 Tool 1: pure::variants

pure::variants is a FODM tool which is developed from pure::systems[2] and is based on eclipse. It is either as a plug-in installation available or as a standalone eclipse application. pure::systems calls the development process as a family based development which is structured as follows:

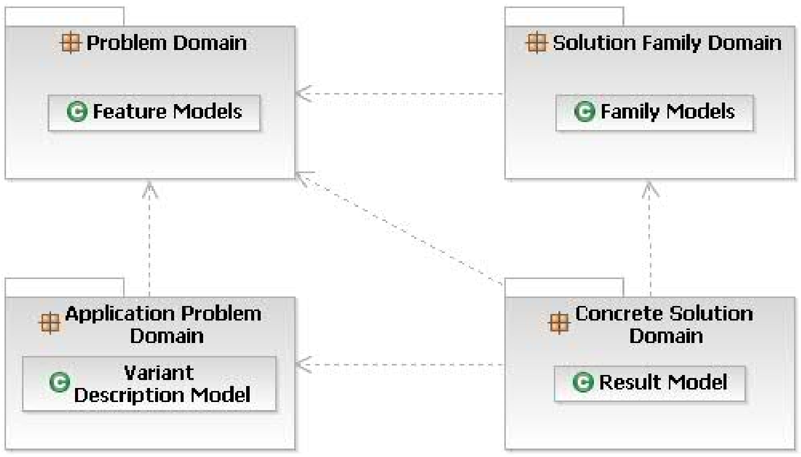


Figure 2.8: Concept of pure::variants [pur12]

[2]http://www.pure-systems.com

The left side of the picture is stated as the problem space and the right side as the solution space. As denoted in [pur12] the feature model is used to model the commonalities and the variability of the given domain. The family model describes the variable family architecture and is linked with some defined rules to the feature models. The variant description model is used to explore the problem domain and to express the problems to be solved. During this process features are selected and additional configuration information is added so that a result model is created. This result model can than be transformed to create a concrete solution.
pure::variants has an open SDK which provides ways to create any kind of model, synchronisation mechanisms for these models, validation checks (not only for models but also for the constraints and relations between them), model transformation clients, and wizards for import/export purpose. On top of that any additional eclipse plug-in (contributions to the UI, editors, ...) can be developed and used.

2.5.2.2 Tool 2: Gears

Gears[3] which is developed from BigLever Software, Inc. is a tool for product line engineering. The methodology is based on a three tiered approach and the according framework can be seen in Figure 2.9.

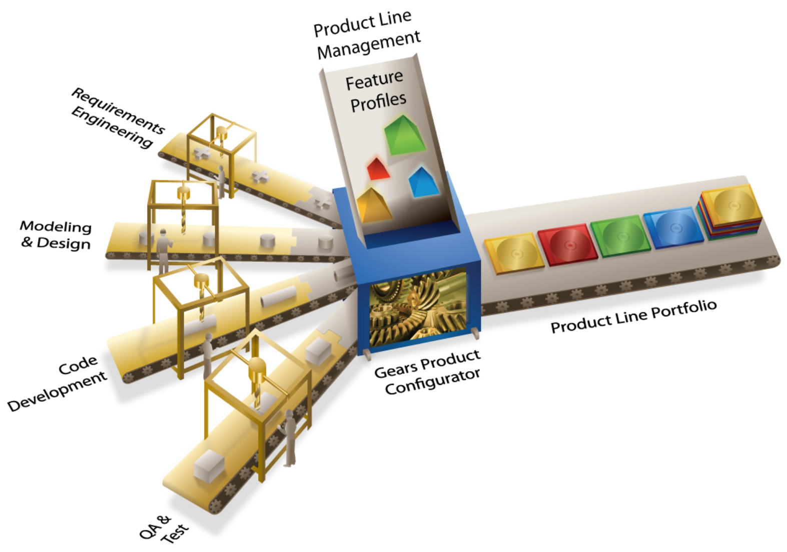


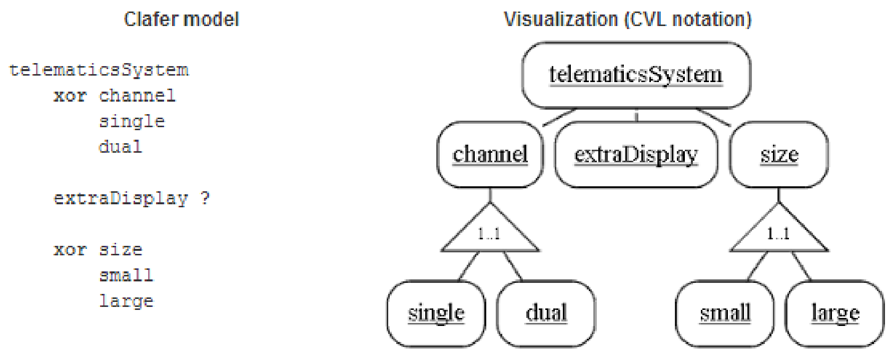
Figure 2.9: Concept of Gears [Big12]

[3]http://www.biglever.com/overview/software_product_lines.html

As they state out in [Big12], they use a single feature model that can express the whole feature diversity of the whole lifecycle of the product line. A single variation point mechanism is used to apply tools to all stages of the product line from requirements engineering over implementation and testing to change management and documentation. And finally a single automated product configurator is used to assemble and configure all parts from each stage of the development lifecycle just by pressing a button.
The Gears Framework is designed to extend the software engineering toolset which means that it does not support any type of import/export functionality due to the fact that it is integrated into the tools itself.

2.5.2.3 Tool 3: Clafer

Clafer[4] is a modeling language for lightweight modelling and is developed from the University of Waterloo. It is free to use and open source, but it is still under development. Although it is a DSM tool, it can be adopted to deal as a feature modelling tool. An example is shown in Figure 2.10.



```
telematicsSystem
    xor channel
        single
        dual

    extraDisplay ?

    xor size
        small
        large
```

Figure 2.10: Clafer feature model example [Uni13]

The tool provides also model verification and validation checks, to ensure that models are consistent. Furthermore it enables the ability to derive an example from a model and to check if this example was derived correctly. Clafer also helps to automatically complete a model according to some user defined rules.
In the current version of Clafer no import/export mechanisms are implemented and since this tool is a modelling tool, it is not natively supported to perform transformations on feature models (as mentioned before it is only possible to derive an example, e.g. an instance of the feature model where only all wished features are selected, but there is no

[4]http://gsd.uwaterloo.ca/clafer/

built in mechanism to transform this selection into a real application). But due to the fact that it is an open source project, these points can be implemented by the user.

2.5.2.4 Tool 4: XFeature

XFeature[5] is a FODM tool which is based on the product family approach and is currently developed from ETH-Zurich. Mainly it was developed for space applications, but due to its adaptable meta-model and meta-meta-model it can be used for every other domain as well. XFeature is available as a eclipse plug-in and is open source (GNU general public licence). Its architecture is based on the following concept:

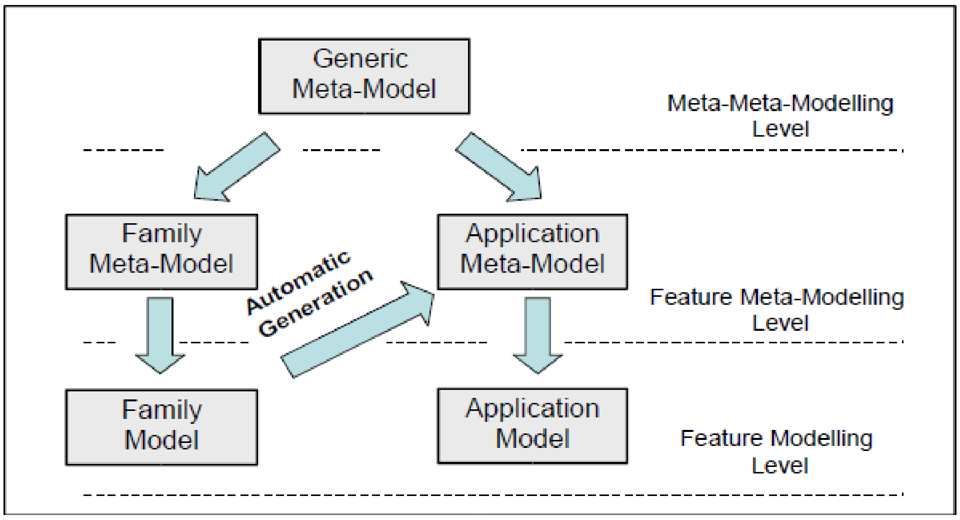


Figure 2.11: XFeature modelling architecture [Ale05]

As stated in [Ale05], the family meta-modelling level describes the available facilities of a product family, whereas the family meta-model is fixed and family-independent. It can be seen as a generic meta-model for all kind of product families. The family model on the feature modelling level is an instance of the family meta-model and deals with specific aspects of a certain product family. Finally the application meta model - which is an instance of the family model - describes the application itself.
Out of the box XFeature does not support any kind of import export functionality or any synchronisation mechanisms to synchronise a model with external sources, but due to its open source characteristics these features could be added by the user.

[5]http://www.pnp-software.com/XFeature/

2.5.3 FODM: Criteria for the Evaluation

The evaluation criteria are split into two main groups. One group deals with the performance of the FODM tool concerning the software product line capabilities and the second group deals with technical aspects like flexibility and expandability.

Weighting factors

The weighting of the different categories reflects the needs of Magna Cosma and the main topics are:

- Expandability/Flexibility of the FODM tool is highly demanded due to the fact that import/export and model synchronisation functionality are demanded for different platforms (e.g. BPM Tool, SAP connection)

- The possibility to compare models with previously created models to highlight the differences and later on analyse the impacts of changes. It is not an essential point for the success of the project, but is demanded relatively high from the project partner.

- Constraint and model checking is a very important criterion due to the fact that complex business processes are modelled with a various number of constraints between the different variations and a model transformation should only be performed if all of these constraints are sufficiently fulfilled to avoid any failure in the sequence of the process. This would lead to a disaster if later on an automatic generation of the ERP system is done.

2.5.4 FODM: Results

		Tools	pure::variants	Gears	Clafer	XFeature
	Criterion	weight	rating	rating	rating	rating
SPLE	Feature/Variability modelling	9	10	9	6	8
	Constraint checking	9	10	8	3	8
	Model comparison	8	9	4	1	1
	Feature meta-model maturity	4	3	1	1	7
Technically	Extensibility	9	10	5	9	5
	Flexibility	7	10	6	10	6
	Usability	5	5	6	6	5
	Tool stability	7	9	9	3	8
	Result	580	512	369	295	348

Table 4: FODM: Tool evaluation results

Conclusion

pure::variants is for this project the most suitable tool since it is easy expandable and quite flexible and provides the best fitting build-in functionality.

2.5.5 BPM Tool: candidates

2.5.5.1 Tool 1: Aeneis

Aeneis[6] is a tool for modelling businesses and their processes which is developed from Intellior AG. It provides the ability to perform process optimisations, document management, automated generation of documentation and many things more. A overview of the key features is given in Figure 2.12.

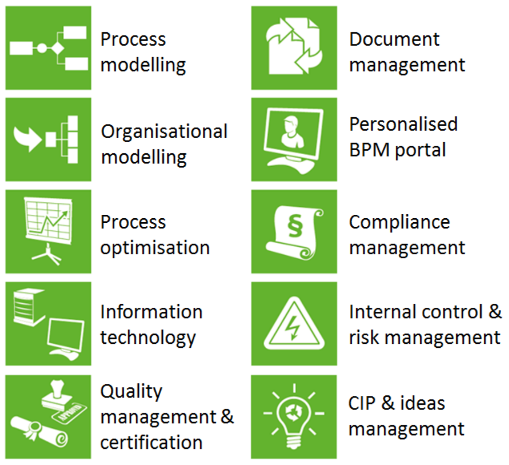


Figure 2.12: Overview of the key features of Aeneis[7]

As stated in [Int12] its meta-model is flexible adaptable which means that it is not only a BPM modelling tool but can also be used as a modelling tool for any other kind of models if the functionality is added to the meta model. It also provides a various number of import/export interfaces to some named systems, like SAP or LDAP. Furthermore the complete Aeneis database can be manipulated using a web service (XML based message exchange) or a java based (plug-in development) API. It provides a workflow engine to integrate the responsible employees into the process stages, and furthermore a version management is integrated to better analyse the impacts of changes and to undo unsuitable changes. It is also possible to link files from a file system to the processes and to automatically include these files into the automatic documentation generation.

[6]http://www.intellior.ag
[7]taken from http://www.bpm-tool.de/loesungen/unternehmensmodellierung/

2.5.5.2 Tool 2: Xpert.Ivy

Xpert.Ivy[8] is a tool to model business processes and is developed from the Swiss company Xpert Line. The concept of it can be seen in Figure 2.13. It is based on three main stages, whereas the first stage deals with the modelling and the design of the process. The second stage deals with the automatic interpretation and execution of the process with its own engine and finally the last stage deals with process controlling to monitor all process parameters and to incorporate the gained knowledge into a new design phase to model a new version to eliminate all identified bottlenecks.

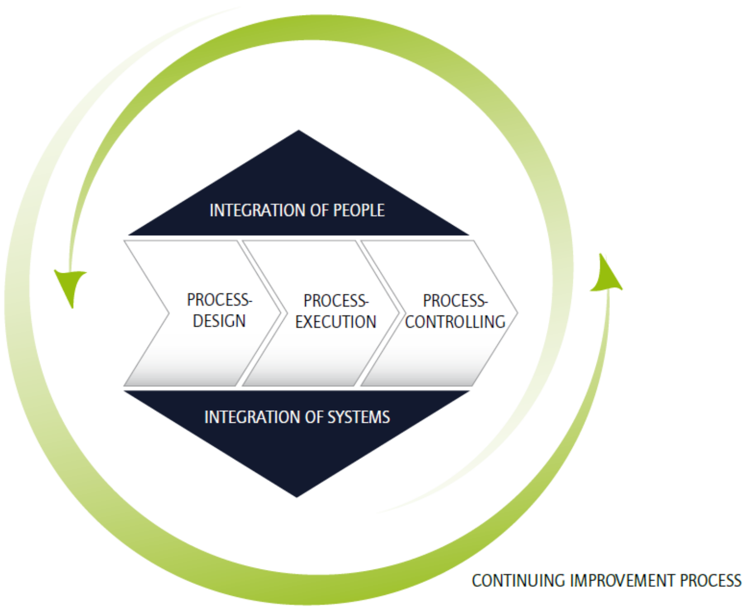


Figure 2.13: Overview of the concept of Xpert.Ivy [Xpe13]

During all three stages the integration of systems and responsible persons is supported. Like Aeneis it provides a workflow engine, a version control system, measures for process optimisation and a various number of interfaces like a Java API, a LDAP connection and connections to databases. Although Aeneis could be seen as a rival (and vice versa), both tools provides interfaces to integrate the other system in the modelling process (automatic model import and synchronisation).

[8]http://www.xpertline.ch/en/

2.5.5.3 Tool 3: Modelio

Modelio[9] is a tool for modelling UML diagrams and BPM processes. The company behind it is Modeliosoft which also provides some other products to extend the functionality of the modelling tool to create a complete business solution. In contrast to the other tools, Modelio is also available as a freeware version.
It provides a Java API to externally manipulate models and with some additional modules the possibility for automatic documentation generation, a version control system, access to the meta-model and some points more. With the documentation module it is also possible to reflect the changes in the generated documentation back to the drawn process model.

2.5.5.4 Tool 4: DHC Vision

DHC Vision[10] is a complete enterprise management system which provides solutions for process management, risk management, audit management, revision management and risk management.

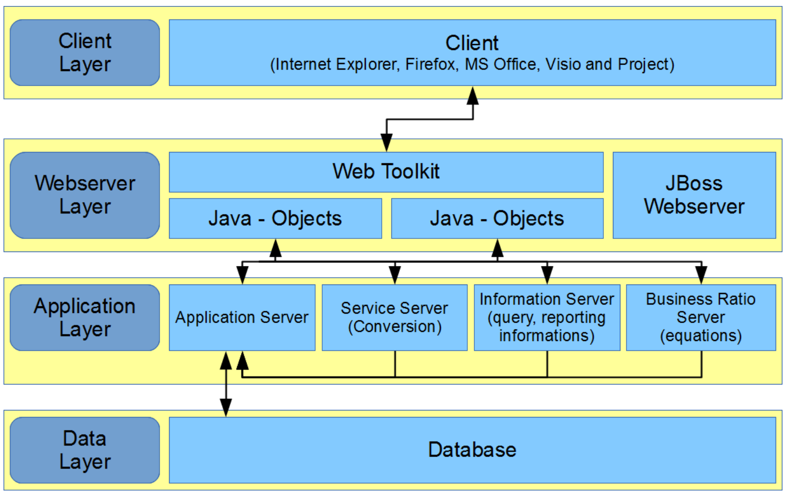

Client Layer
Client
(Internet Explorer, Firefox, MS Office, Visio and Project)
Webserver Layer
Web Toolkit
Java - Objects
Java - Objects
JBoss Webserver
Application Layer
Application Server
Service Server (Conversion)
Information Server (query, reporting informations)
Business Ratio Server (equations)
Data Layer
Database

Figure 2.14: Four layer architecture of DHC Vision

The architecture of it is based on a four layered architecture. In difference to the other tools, there is no real standalone client with a GUI. The application logic runs in the background while the presentation of the data is done over a html portal and can be

[9]http://www.modeliosoft.com
[10]http://www.dhc-gmbh.com/de/

viewed with every ordinary web browser (the official supported clients are Internet Explorer and Firefox, but every other browser should work as well).
Like Aeneis and Xpert Ivy, it provides a workflow engine, a version control system and some interfaces to manipulate the database.

2.5.6 BPM: Criteria for the Evaluation

The evaluation criteria are split into two main groups. One group deals with the performance of the BPM tool concerning the business process modelling capabilities and the second group deals with technical aspects like flexibility and expandability.

Weighting factors

The weighting of the different categories reflects the needs of Magna Cosma and the main topics are:

- Expandability/Flexibility of the BPM tool is highly demanded due to the fact that it should cooperate with the chosen FODM tool and should provide freely adoptable attribute lists and user defined objects
- The possibility to compare models with previously created models to highlight the differences and later on analyse the impacts of changes. It is not an essential point for the success of the project, but is demanded relatively high from the project partner.
- A workflow engine or a similar facility
- Automatic document generation for a various number of documents (e.g. reports, user guides, etc.

2.5.7 BPM: Results

		Tools	Aeneis	Xpert.Ivy	Modelio	DHC Vision
	Criterion	weight	rating	rating	rating	rating
BPM	BPMN 2.0	9	10	10	10	10
	User defined attributes/objects	8	10	8	8	9
	Workflow engine/approval process	8	9	10	4	9
	Model comparison	8	9	8	6	9
Technically	Extensibility	9	10	8	5	9
	Flexibility	7	9	9	6	9
	Usability	6	7	7	5	7
	Tool stability	7	10	10	10	10
	Document generation	8	9	8	9	10
	Result	700	651	609	503	642

Table 5: BPM: Tool evaluation results

Conclusion

Expect Modelio, all analysed tools are fully-fledged and highly capable for being used as a business process modelling tool. Especially the difference in functionality between the DHC Vision Tool and Aeneis is more or less negligible. Due to this head-to-head result, it was chosen to implement the coupling between the pure::variants application and BPM Tool in an open and easy to expand way so that the modelling tool could be easily replaced.

3 Variability modelling framework

The main goal of this work is to reduce the complexity and time consumption of modelling new process variants of an existing process, to easily switch between variants whereupon the documentation of all variants keeps consistent and easy to maintain.

3.1 Conceptual design

An overview of the complete conceptual design can be seen in Figure 3.15. This conceptual design includes some parts, which are not implemented in the current version of this project, but will probably be included in some future projects which are based on this work.

As seen in the Figure, the project can be split up into two main parts, the domain engineering and the application engineering. In the domain engineering an expert (or experts) for business processes develops and maintains specific processes and sub-processes with a BPM tool. With the guidance of this expert and based on the established artefacts, a feature model is created which can be updated automatically with new artefacts if they are added to the BPM tool. In fact updating an existing feature model should - symbolically spoken - be possible by just pressing a button and without the help of a domain expert because he already did all necessary steps in his BPM tool by modelling the process. Theoretically this update could be triggered from the SPL - tool (in this case pure::variants) or from the BPM tool after something has changed. In the current version of this project, the update mechanism is triggered from the SPL tool. The second approach may be implemented in a future project.

In the application engineering the developed feature model is used to select the wished artefacts and to transform these selections into a solution (a consistent business process). Additional constraints and restrictions help to ensure that no invalid process can be created, whereas these things are based on the knowledge of a domain expert. The selection of the artefacts and the creation of the solution is done by an expert who knows the current needs of the production.
After the transformation is done, it should be possible to add some additional input into the BPM domain and to flow these additional inputs back to the feature model, or the variant description model. Since this is not an essential part for the success of this project, this part is currently not implemented.

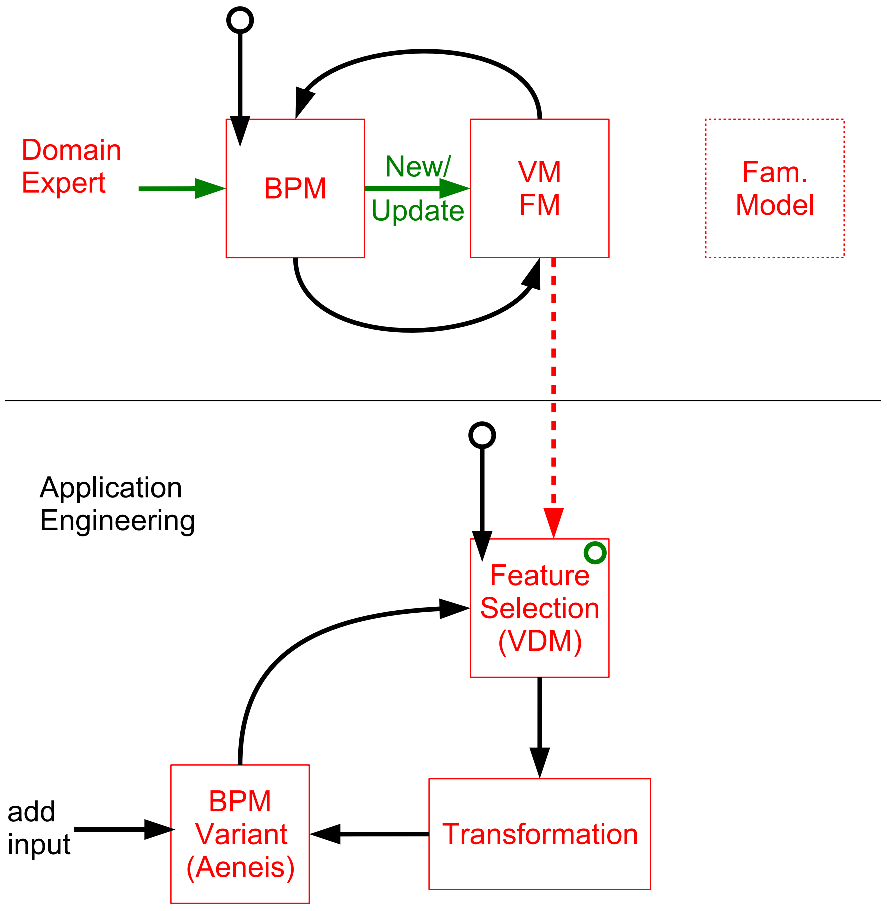


Figure 3.15: Conceptual design of the complete System

3.2 Type model

When developing a feature model, each node can be assigned to a different type so that the model is intuitive and quite easy understandable. Different types also enable the ability to define additional construction rules, constraint checks, etc. As a result a new type model was developed which contains the following types:

- **varBPM:settings**: Only one node of this type can be present in a varBPM project. It deals as root node and contains all needed information for establishing a connection to the external BPM tool.

- **varBPM:root**: A root node represents a business process, where the variability should be modelled.

- **varBPM:variationpoint**: A variationpoint can be seen as a sub-node in a process, where different variations can be linked to (e.g. a node called "logistical chain control" where "one card Kanban" or "event triggered Kanban" can be chosen).
- **varBPM:variation**: A variation is a process which is linked to a specific variation point. This process can contain variation points.

These types are also represented in the types of the nodes of the "VariabilityTree" in the implementation of the plug-in. For more information see chapter 4.3.

3.3 Construction rules

The construction of a feature model works as follows:

1. Start with a varBPM:settings node as root element
2. Add root processes where the variability should be modelled
3. For each root process do:
    a) Define the variation points
    b) Each variation point needs at least one variation which is linked to it
    c) Look at each variation and consider if this process contains variability and if so, go back to a)

The modelling is done, if each leaf has the type "varBPM:variation". A graphical interpretation of these rules can be seen in the Figure below.

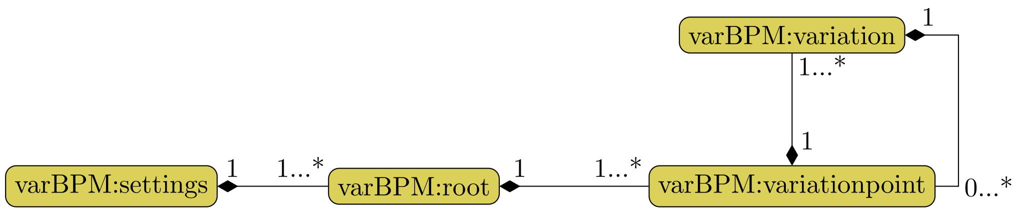


Figure 3.16: Construction rules for the feature model

3.4 Relations and Restrictions

According to user definable model checks and additional constraints that can be defined from the user, pure::variants provides a rich interface to help the modeller modelling a consistent and easy to use model. By default the varBPM plug-in automatically defines one additional restriction that is as follows:

> When assembling a solution (choosing the wished features) it is not possible to reference the same variation point with different variations.

Additional restriction can be defined by the modeller and are automatically integrated into the varBPM plug-in, which means that defined rules won't be lost after a model is updated or changed, as long as the update mechanism of this plug-in is used to perform these changes.

4 Implementation of the varBPM toolchain

In the next few chapters an overview of the developed classes of the key features is given and it is hinted where and how the functionality of these classes can be extended. First the (abstract) base classes are described followed by the concrete implementation for the varBPM plug-in using Aeneis as BPM Tool.

4.1 The varBPM PluginHandler

The "PluginHandler" class deals as main entry point for all parts of the plug-in and gives access to a default font (to ensure that every text uses the same font) and the "Client" object which deals as a connector between the tool integration parts and the variability parts of this project (see chapter 4.2 for more details). The "PluginHandler" class is automatically loaded if one of the plug-in related classes is loaded (effectively when a varBPM Model is opened, changed, checked, updated, created or transformed).

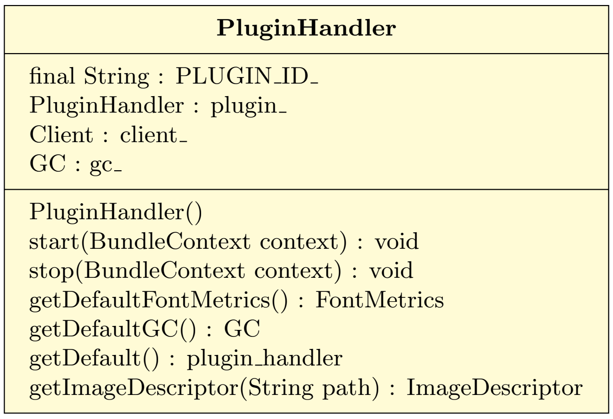

PluginHandler
final String : PLUGIN_ID_
PluginHandler : plugin_
Client : client_
GC : gc_
PluginHandler()
start(BundleContext context) : void
stop(BundleContext context) : void
getDefaultFontMetrics() : FontMetrics
getDefaultGC() : GC
getDefault() : plugin_handler
getImageDescriptor(String path) : ImageDescriptor

Figure 4.17: The plugin activator class

4.2 Tool integration

As stated in chapter 2.4, there are two ways to establish a working and expandable tool integration and for this project the "integration based on integrated models" attempt is used. The common data model of the tools is shown in Figure 4.18.
This model was primarily derived from the representation of an object in the Aeneis application, but due to its simplicity it can be used for any other tool as well and

therefore it is a good to use data model. In the current version of the varBPM plugin, the collection of values must consist of at least one value. A value stored with the key 'children' which holds all elements that are children of the according object.

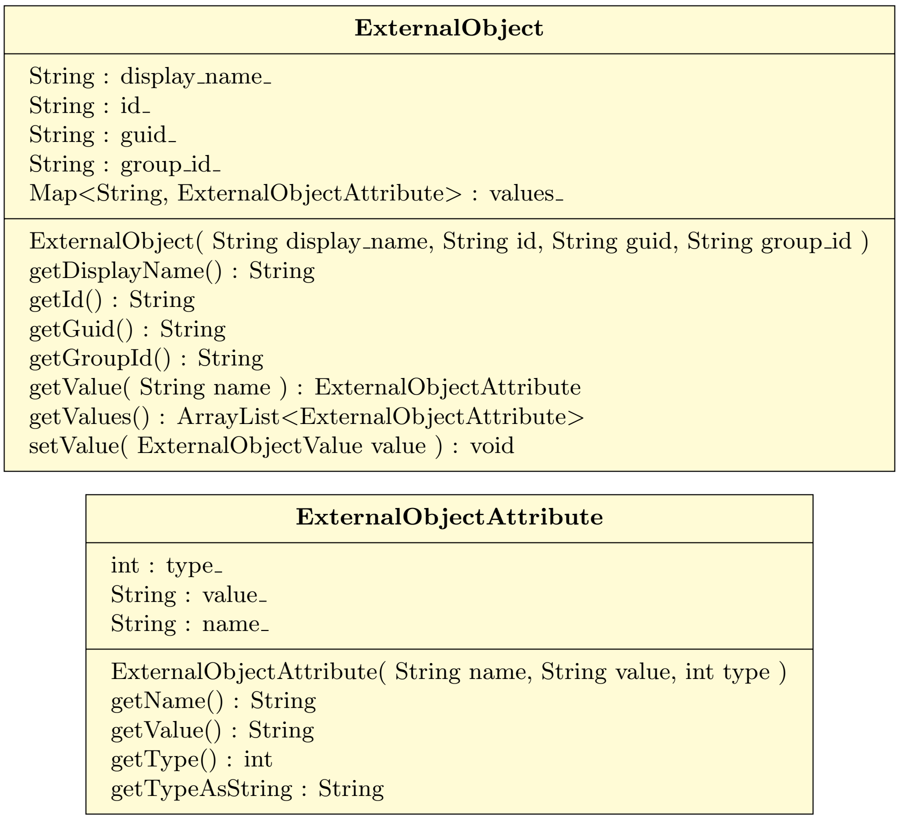

ExternalObject
String : display_name_
String : id_
String : guid_
String : group_id_
Map<String, ExternalObjectAttribute> : values_
ExternalObject(String display_name, String id, String guid, String group_id)
getDisplayName() : String
getId() : String
getGuid() : String
getGroupId() : String
getValue(String name) : ExternalObjectAttribute
getValues() : ArrayList<ExternalObjectAttribute>
setValue(ExternalObjectValue value) : void
ExternalObjectAttribute
int : type_
String : value_
String : name_
ExternalObjectAttribute(String name, String value, int type)
getName() : String
getValue() : String
getType() : int
getTypeAsString : String

Figure 4.18: Tool Integration: Java base class of the common data model

An overview of the integration mechanism can be seen in Figure 4.19. The complete communication of the varBPM logic is handled by the "Client" class. At first sight this class seems like a little bit of overhead, since the "Connector" class itself already transformed the data model of the BPM application into the common data model. The reason why this "Client" is used is that this class also deals as a broker, which means that the varBPM logic connects to an application by calling the client with an identifier for the application and a list of parameters which are necessary to establish a connection. The client then chooses the right connector class and forwards all requests to the wished connector. This reduces the effort to implement new connections, since nothing in the varBPM logic needs to be changed and furthermore the import and update wizards can be reused, including the page where the settings for the specific application are set since this page is written in a generic way, where its appearance is automatically adopted.

The "Client" class also deals as an abstraction layer, meaning that there exists only one method for getting objects from the BPM Application and internally this class tries several methods until one or more objects are found.

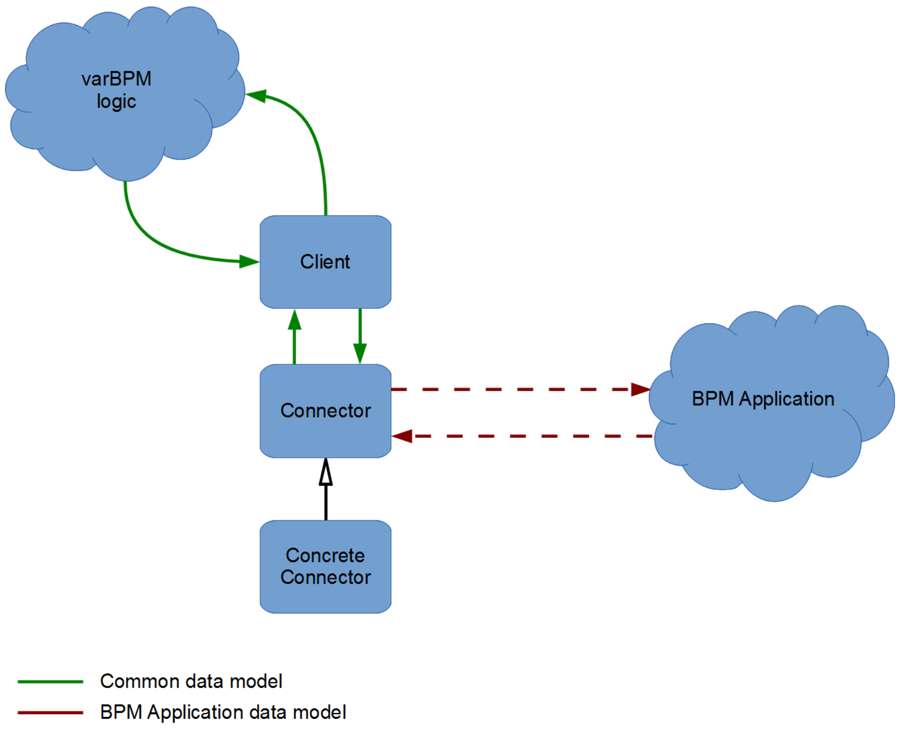


Figure 4.19: Tool Integration Overview

If a new BPM application should be connected to the plugin, a connector has to be written which connects to the according tool and performs the transformation from the particular model to the ExternalObject data type and vice versa. The base class for this kind of connector can be seen in Figure 4.20. Note that it does not matter if the real transformation between the client object and the common data model is done by the connector or by a subclass of the ExternalObject class.
During the research activities and the evaluation phase of different BPM modelling tools, I have noticed that not all tools differ between an id (user definable) and a guid (generated unique identifier; not user definable). As a result if a tool should be integrated which only supports an id (or only a guid) the according method in the Connector can just do nothing (only returning null) or show the same behaviour as the other method. The ExternalObject returns the id of the object if it does not have a guid and vice versa.

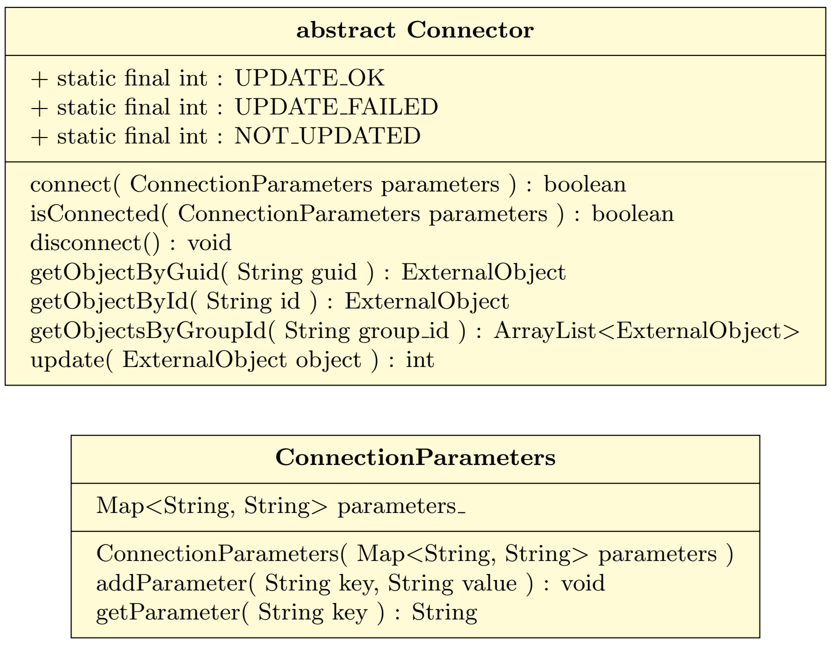

abstract Connector
+ static final int : UPDATE_OK
+ static final int : UPDATE_FAILED
+ static final int : NOT_UPDATED
connect(ConnectionParameters parameters) : boolean
isConnected(ConnectionParameters parameters) : boolean
disconnect() : void
getObjectByGuid(String guid) : ExternalObject
getObjectById(String id) : ExternalObject
getObjectsByGroupId(String group_id) : ArrayList<ExternalObject>
update(ExternalObject object) : int
ConnectionParameters
Map<String, String> parameters_
ConnectionParameters(Map<String, String> parameters)
addParameter(String key, String value) : void
getParameter(String key) : String

Figure 4.20: Tool Integration: Abstract java base class of the connector

4.2.1 Concrete implementation

For the concrete case, the connection to the Aeneis application is done by a web-service interface, which means that the communication between these tools is based on an xml based message exchange. The transformation of the object reference to the 'ExternalObject' model is done by a subclass called 'AeneisObject' and a 'AeneisObjectValue' subclass which is derived from the 'ExternalObjectValue' class.
The class diagram can be seen in Figure 4.21.

The transformation from the ExternalObject back to the native Aeneis type is partly done by the AeneisConnector. First it gets the latest version of the native Aeneis object (it is possible, that someone has added a specific attribute or has changed a value) then it updates the varBPM related attributes from the ExternalObject and delivers these changes to the Aeneis database.

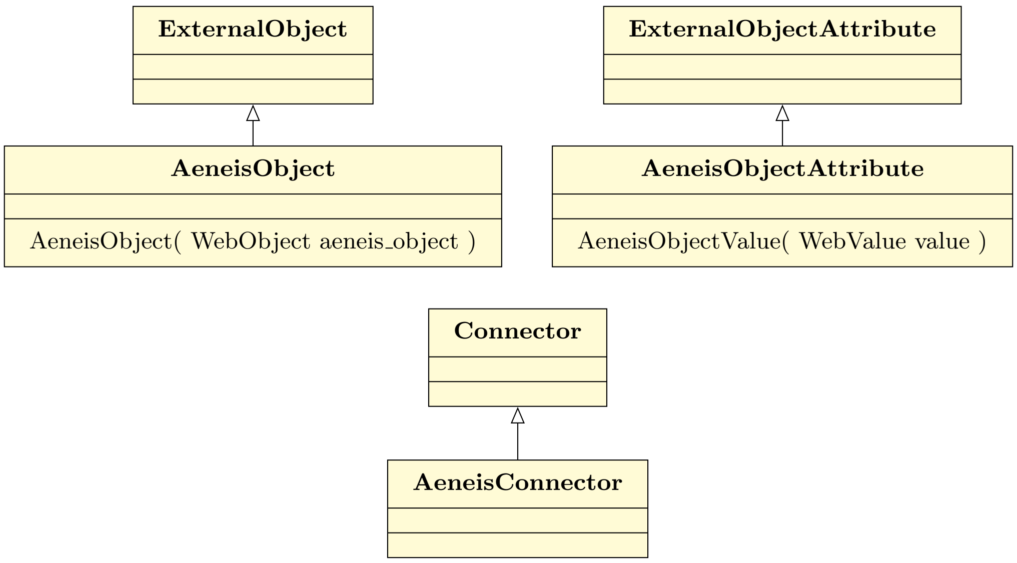


Figure 4.21: Tool Integration: Concrete Java implementation of the data model and connector

4.3 Variability

The variability of a process (or processes) is modelled using a tree representation. For simplicity the root node of this tree contains all needed information for the tool integration. It is also a little 'hack' for the pure::variants restriction that only one root node is allowed for one variability model (otherwise it would not be possible to manage more business processes in one feature model). The class diagram of all node types can be seen in Figure 4.22. Of course, there are a lot more members and methods defined, but this Figure gives a good overview over all needed types to establish a tree.

Due to the fact that in some situations it is needed to get all children of a node of a specific type, the 'getChildren' method is overloaded to automatically return a list of all elements of a specific type (an empty list is returned if there are no children of the wished type).

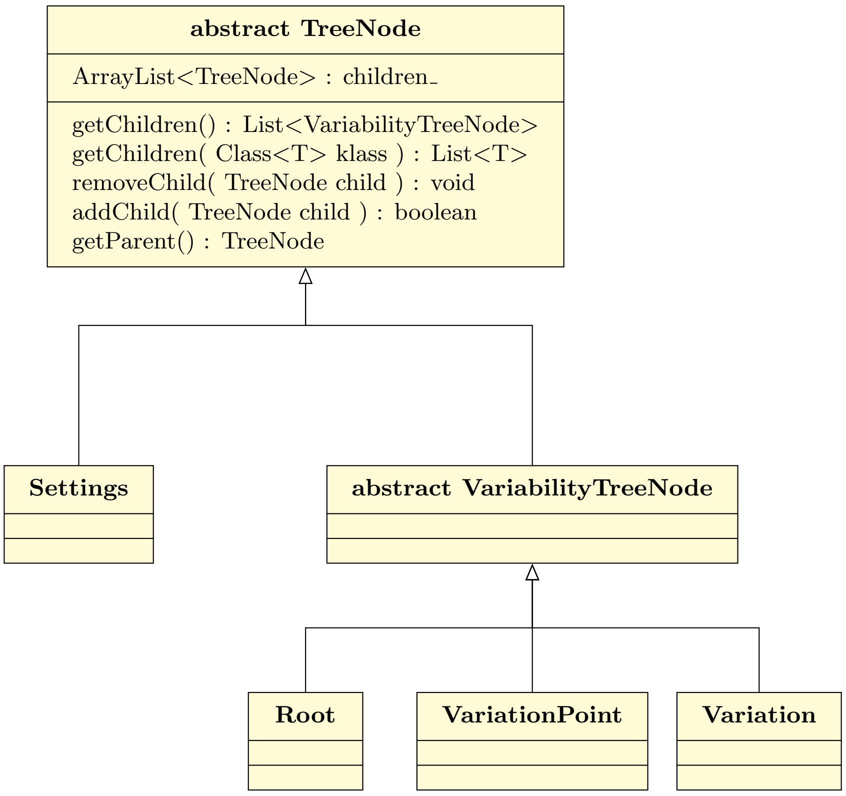

abstract TreeNode
ArrayList<TreeNode> : children_
getChildren() : List<VariabilityTreeNode>
getChildren(Class<T> klass) : List<T>
removeChild(TreeNode child) : void
addChild(TreeNode child) : boolean
getParent() : TreeNode
Settings
abstract VariabilityTreeNode
Root
VariationPoint
Variation

Figure 4.22: Variability tree node types

As one might expect, a composite pattern is used for the class hierarchy of the tree, but there are some additional rules which have to be barred in mind when manipulating the tree.

- The type of the root node is ’Settings’ and the children of this node are ’Root’ nodes.
- The children of ’Root’ nodes are ’VariationPoint’ objects
- The children of ’VariationPoint’ nodes are ’Variation’ objects
- The children of ’Variation’ nodes are ’VariationPoint’ objects

These rules are checked in the according ’addChild’ method which has to be implemented by each node to ensure these restrictions (the ’TreeNode’ base class only provides a default implementation where each type is accepted). The ’addChild’ Method also automatically checks if the given child is already in the list of children. Graphically these rules are:

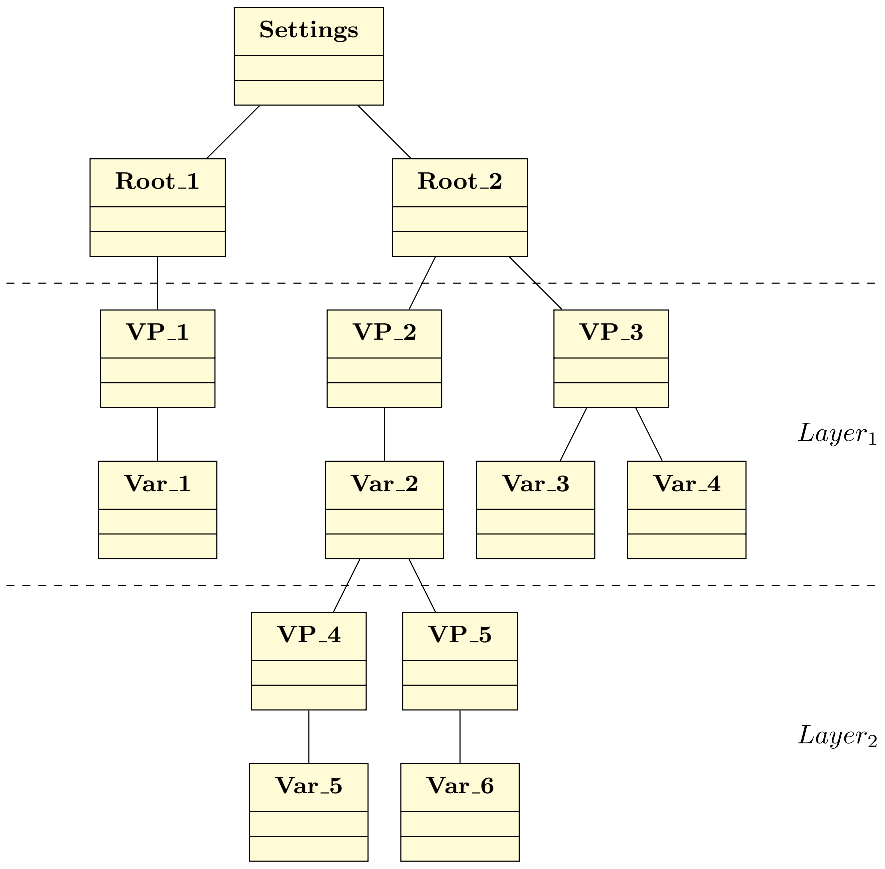
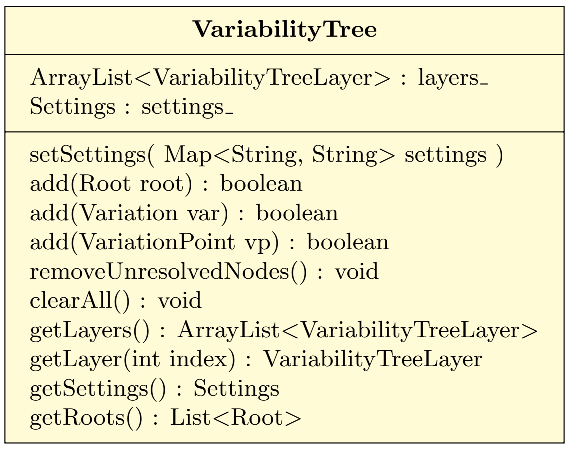


Figure 4.23: Variability tree abstract concept

The variability tree is accepted as valid if each branch of the tree ends with a node of the type 'Variation' and if all nodes have a correct reference to an 'ExternalObject' instance. As indicated in the Figure, the tree is also grouped into layers where pairs of the variation points and there according variations are grouped together. It is intended that the first layer represents the abstract model of the process and with the increasing layer number the level of detail is increasing.
In the application an additional class is used to easy add/delete nodes and access the tree. An overview is given in Figure 4.24.



```
VariabilityTree
ArrayList<VariabilityTreeLayer> : layers_
Settings : settings_
setSettings( Map<String, String> settings )
add(Root root) : boolean
add(Variation var) : boolean
add(VariationPoint vp) : boolean
removeUnresolvedNodes() : void
clearAll() : void
getLayers() : ArrayList<VariabilityTreeLayer>
getLayer(int index) : VariabilityTreeLayer
getSettings() : Settings
getRoots() : List<Root>
```

Figure 4.24: Overview of the VariabilityTree class

One might notice that there exists no direct functionality to remove a specific node from the tree. This has one reason: When a node should be deleted, all sub-nodes of it should also be deleted. When remodelling/updating a model it is possible that someone deletes accidentally a node. To recover such a accidentally deleted object, quite a lot of steps need to be done so it was decided that it is only possible to delete nodes indirectly. It is only possible when the user first marks the node to be deleted (an according method of the node is called) and then confirms the model as valid. After the model is marked as valid, the 'removeUnresolvedNodes' method is called and all nodes which are marked to be deleted or which have unresolved object references to the external tool are deleted completely out of the system.

4.4 Model validation

Creating a valid model is a vital task for a modeller and for this purpose some model checks where implemented to support the modeller during the creation of a model. The checks can be grouped into two severity categories, one with a high severity level (error) and one with a low severity (warning). As long as there is at least one error in the model it is not possible to transform the model into a solution. Warnings are hints for the modeller to indicate that something is not as expected, but this does not mean that the model is invalid and therefore a model transformation is possible. The following checks where implemented:

- **Check node type:** Checks whether the node has the right type or not. Note that the type is derived from the distance to the root node. As a result if the tree structure changes, this check needs to be updated as well. This check has an error severity with an automatic quick fix which can also be used as a multi-fix. The reason why this check has a error severity is due to the fact that some of the following checks are type based.

- **Check attributes:** According to the type of the node some attributes must be set to ensure that a model transformation is possible. This check has an error severity and provides also an automatic quick fix which can be used as a multi-fix.

- **Check double references:** In the variant description model (VDM) it is not allowed to double reference a specific variation from two different branches, except in both branches all sub nodes of this variation are set equally. This check has also an error severity and no automatic fix is provided.

- **Check if a variation is added with a group id:** If a node is added with a group id, it means that the plugin automatically searches for new variations with the same group id when the model is updated. This enables a very comfortable way to keep the models up to date when processes are nicely grouped in the according BPM tool. This check has a warning severity and an automatic quick fix which can be used as a multi-fix.

To implement these check mechanisms, two classes were written and added to the pure::variants check engine. These classes can be seen in Figure 4.25.

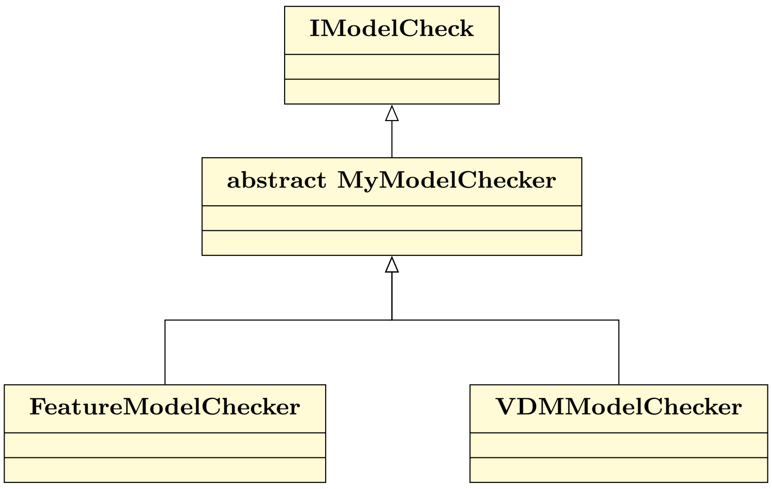


Figure 4.25: The model check engine

The abstract base class "MyModelChecker" just implements some common used methods of the two subclasses which can also be seen in the Figure. The two subclasses were added to the "com.ps.consule.eclipse.ui.checks.Checks" extension point of the pure::variants implementation and are automatically called if a feature model or a VDM model is opened, changed, or checked manually. Both of these classes first performs a model type check, to ensure, that these checks are only performed on varBPM models.

4.4.1 Feature model validation

The implemented classes for performing node checks in the feature model can be seen in Figure 4.4.1. The scope of the constructor of these classes is locked to the package to ensure that the creation of these checks is done using the 'NodeCheckerFactory'. If a new check is implemented, it needs to be registered with its type to the factory. Nothing else needs to be changed except some of the constants defined in the "NodeConstants" class if completely new types are implemented.

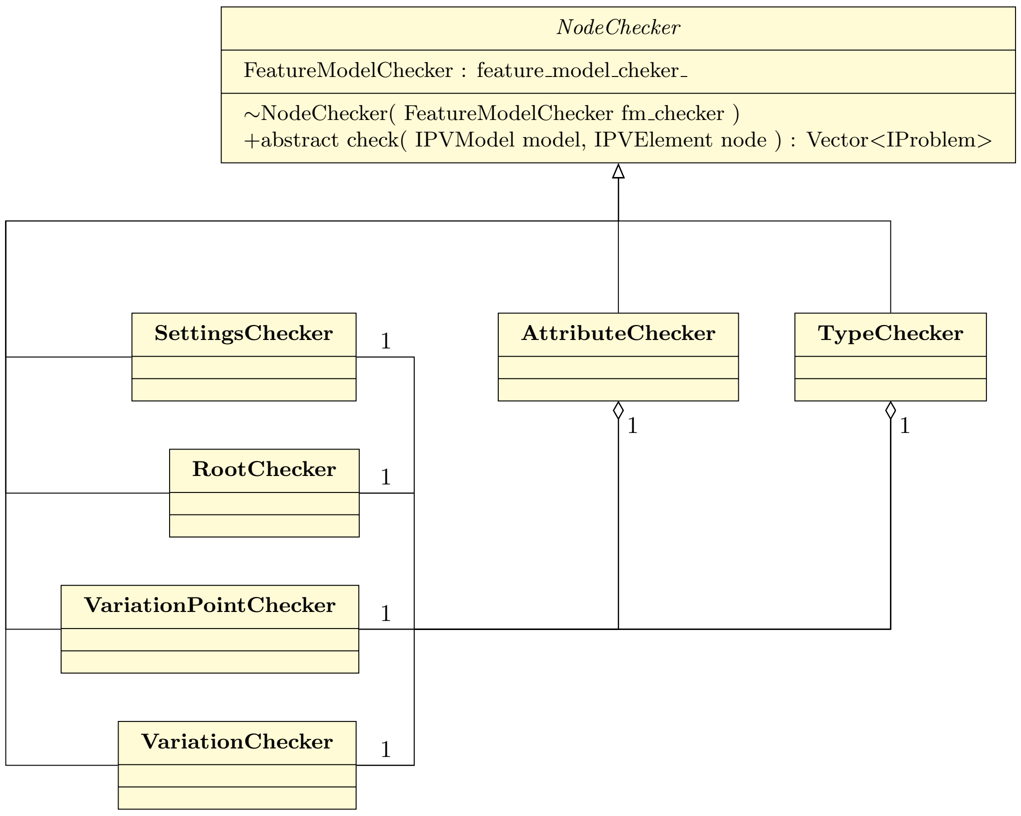

NodeChecker
FeatureModelChecker : feature_model_cheker_
~NodeChecker(FeatureModelChecker fm_checker)
+abstract check(IPVModel model, IPVElement node) : Vector<IProblem>
SettingsChecker
AttributeChecker
TypeChecker
RootChecker
VariationPointChecker
VariationChecker
1
1
1
1
1
1

Figure 4.26: Implemented classes for node checks

It would be possible to define a new eclipse extension point for node checks so that the registration of new checks can also be done in other plug-ins making the project more expandable, but this is currently not implemented.

4.4.2 VDM model validation

Currently there only exists one check which needs the information of the complete VDM model. Due to this, no additional classes were implemented.

4.4.3 Available automatic quick fixes

In the eclipse application it is possible to add quick fixes to warnings or errors which automatically try to solve the error, or to guide the user to a solution of the problem. As previously mentioned some "fixes" where developed and the following table gives a short overview over all developed quick fixes and what they are doing:

Model type	What	Severity	According quick fix	Multi-fix
ps::fm	Check node type	Error	Automatically set the type	X
ps::fm	Check attributes	Error	Opens a dialogue to change the settings	-
ps::fm	Check if a variation was added with the group id	Warning	Change the according attribute to true	X
ps::VDM	Check double references	Error	none	-

Table 6: Overview of all automatic quick fixes

'Multi fix' means, that all occurring warnings/errors can be fixed using the same quick fix. To provide an easy way to add new quick fixes a factory was developed, where the quick fixes can be registered with the type of the warning or error. If one of the "ModelChecker" classes determines a problem, they will try to obtain an according fix from the QuickFixFactory.

As mentioned earlier, another eclipse extension point could be implemented so that the registration of new fixes can be done through different plug-ins without manipulating the code.

4.5 Model transformation

To transform a model, a new class was implemented and registered at the "com.ps.consul.eclipse.core.ClientTransformModule" extension point. The implemented transformation module can be chosen in the "Transformation Config Dialog" and only works if the model is a valid varBPM model. If not, this transformation module skips all further computations. The class hierarchy can be seen in Figure 4.27.

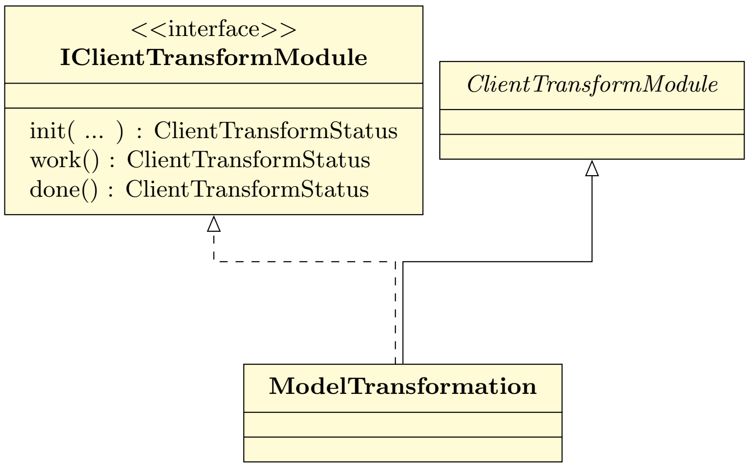

<<interface>>
IClientTransformModule
init(...) : ClientTransformStatus
work() : ClientTransformStatus
done() : ClientTransformStatus
ClientTransformModule
ModelTransformation

Figure 4.27: Class hierarchy for the transformation module

During the model transformation a VariabilityTree is built with all selected nodes and passed to the Client by calling the update - method. The abstract class "ClientTransformModule" gives access to the input models during the transformation and implements some standard implementation of some methods and is provided by pure::variants.

4.6 Model conversion

A feature model can be seen from two kind of perspectives: One is the internal feature model representation of pure::variants and the other one is the schema used by the varBPM plug-in (the VariabilityTree). It is necessary to convert between these two worlds, since the pure::variants feature model representation is used to model the variability and to derive concrete product variants and the VariabilityTree represents the concrete connection to the domain artefacts (business processes). Meaning that changes to the business process structure (update mechanism) in the according BPM modelling tool affects at first the VariabilityTree which must then reflect the changes to the pure::variants feature model. The reverse transformation occurs during the product derivation or by manually editing the feature model.

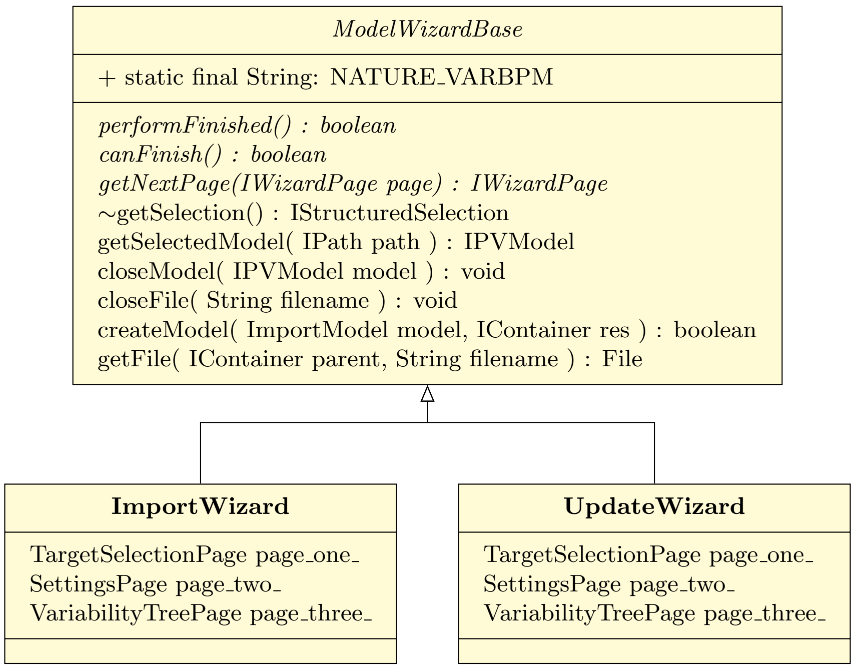
To convert between these two worlds, two conversion classes where implemented. The "ImportModel" class in combination with the ModelGenerator class from pure::variants converts the VariabilityTree into a IPVModel and the "IPVMToVariabilityTree" class implements the reverse way.
Note that these two representations are fully equivalent with regard to the content, but as mentioned before, both are optimized for specific use cases.

4.7 User interface

Various contributions to the user interface have been developed using the java.swt classes so that each contribution looks and feels like the rest of pure::variants. The first contribution that the user will notice is the menu entry "varBPM" with two entries "Import Feature Model" and "Update Feature Model". If one of those entries is pressed, the according wizard will open, which are introduced in the next sections.

4.7.1 Wizards

The class hierarchy of all wizard classes can be seen in Figure 4.28.
Due to the fact that all instances of a model needs to be closed before it can be updated (or overwritten), the base class provides two methods to solve this problem. To check if a wizard is able to be finished, a canFinish method is defined. If it returns true, the eclipse application automatically unlocks the finish button (otherwise it will lock the button). The getNextPage method should return the next page of the given one, so that the eclipse application can determine if the next button should be unlocked or not. The performFinish method is automatically called after the finish button is pressed. If this method returns true, the wizard will be closed automatically and if it returns false the wizard stays unchanged.



ModelWizardBase
+ static final String: NATURE_VARBPM
performFinished() : boolean
canFinish() : boolean
getNextPage(IWizardPage page) : IWizardPage
~getSelection() : IStructuredSelection
getSelectedModel(IPath path) : IPVModel
closeModel(IPVModel model) : void
closeFile(String filename) : void
createModel(ImportModel model, IContainer res) : boolean
getFile(IContainer parent, String filename) : File
ImportWizard
TargetSelectionPage page_one_
SettingsPage page_two_
VariabilityTreePage page_three_
UpdateWizard
TargetSelectionPage page_one_
SettingsPage page_two_
VariabilityTreePage page_three_

Figure 4.28: Class hierarchy for the wizard classes

Import wizard

For the creation of varBPM feature models, a import wizard was created and registered at the "com.ps.consul.ui.pvimport.VariantImportWizards" extension point so that the wizard is recognized by pure::variants and can be chosen like any other import wizard. After pressing the finish button a new feature model is created. If the created model was already present in the project (same file name) the user is asked if it should be replaced or if the new model should be stored with a different name. Replacing the old one will delete it completely out of the project (all constraints, restrictions and settings are lost!).

Update wizard

The update mechanism of an existing feature model can be either started by using the build in mechanism from pure::variants or by choosing the "Update Feature Model" from the varBPM menu. Of course it is also possible to manually add nodes to the model by using the feature model editor. The implemented check engine will help the modeller by

hinting what values are missing, or which of them are wrongly set, but this should only be done if the modeller exactly knows what to do. Otherwise it will be a very annoying task.
The difference between the built in update mechanism and the update wizard is, that the built in one is not as mighty. It is only possible to automatically load new variations if the previously created one where added using the group id, or to delete nodes where the object reference is missing (e.g. if the according object was delete from the BPM tool). It is not possible to set new variation points, add root elements or add variations by own will. For this purposes the update wizard must be used.
In difference to the import wizard, all previously set constraints, relations or settings won't be deleted after pressing the finish button except one of the participants was deleted so that it is obsolete.

Wizard pages

Due to the fact that the update wizard and the import wizard share a lot of commonalities, they also share the same pages. The only difference is that on the first page of the update wizard an old model must be chosen. An overview of all developed classes and the important methods is given in Figure 4.29. All classes implements the UpdateListener interface where the complete graphic is repainted and the status of the buttons (next, back, finished) are recalculated. As a result, it is very easy to couple the repainting of the page with a specific activity.
The SettingsPage class is a generic written page that can be reused for any other kind of BPM tool. Its appearance is automatically adjusted according to the selected BPM tool (via a drop down list). As a result, if a new BPM tool should be integrated all needed settings needs to be registered and the complete import/update mechanism works without any further changes. The check engine also automatically adapts its behaviour if the new variables where registered correctly. The changes are:

- A new entry has to be added to the SETTINGS_GUI constant in the WizardConstants class
- A new entry has to be added to the SETTINGS_PROPERTIES constant in the WizardConstants class.

As I previously pointed out, adding a new BPM application means no changes in the logic behind the models. With some adaptations it should be possible to add new representations of the SettingsPage via an own eclipse extension point.

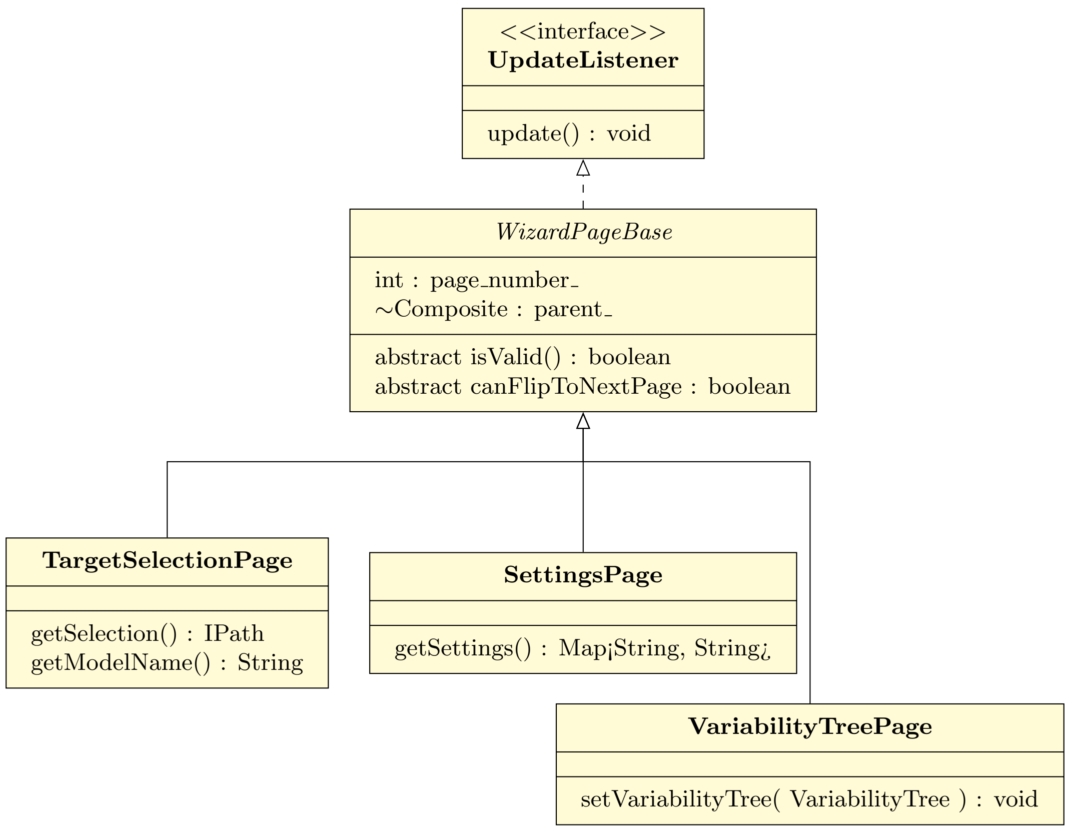


Figure 4.29: Overview of the implemented wizard pages

4.8 The command structure

Due to the fact that eclipse/pure::variants uses at least two different threads - one for the graphic and one for the application logic - there needs to be a mechanism to ensure that clicking a button executes an action in the application logic where maybe a new task for the graphical representation is formed. For this purpose a kind of command pattern was implemented, whereas all Commands implements the ActionListener, ModifyListener and SelectionListener interface. This enables the ability to add each Command to each graphic component. Packing those classes into a runnable implementation (a simple generic version is available) and passing it to the scheduler also solves the problem of invalid thread accesses.

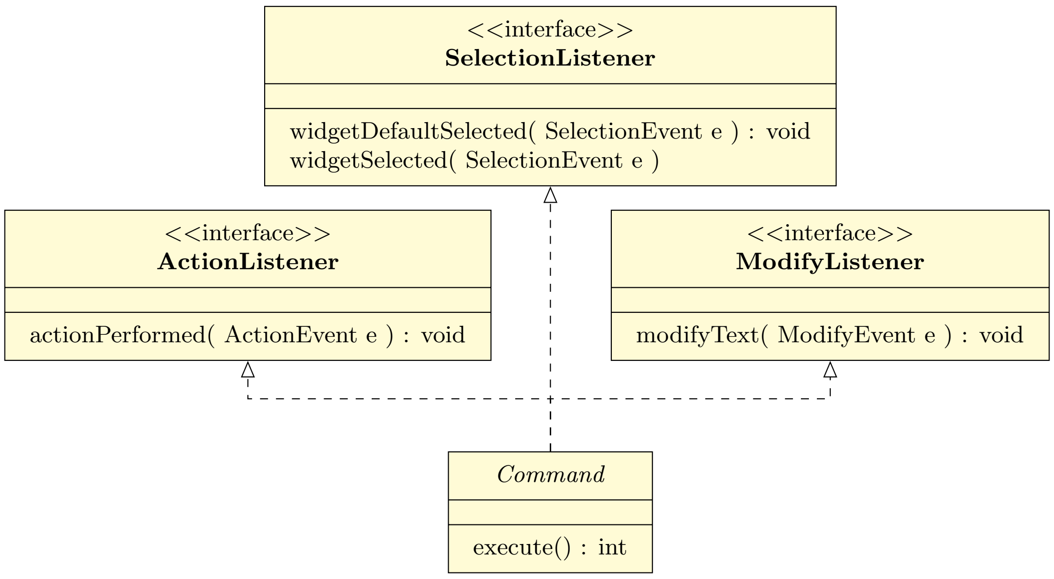

<<interface>>
SelectionListener
widgetDefaultSelected(SelectionEvent e) : void
widgetSelected(SelectionEvent e)
<<interface>>
ActionListener
actionPerformed(ActionEvent e) : void
<<interface>>
ModifyListener
modifyText(ModifyEvent e) : void
Command
execute() : int

Figure 4.30: Overview of the command structure

Implemented subclasses are:

CommandAddRoot	CommandAddVariation
CommandAddVariationPoint	CommandAdoptObjects
CommandChangeObjectReference	CommandCloseDialog
CommandConnectTo	CommandLoadObject
CommandRemoveSelectedObjectReference	CommandUpdate

Table 7: Implemented Command types

5 Using the varBPM tools in practice

5.1 Glossary

- **Root Process:** A root process is a Aeneis Process with a BPM diagram and deals as start point for the modelling process. It consists of at least one variation point and per default it does not depend on any other processes.
- **Variation point:** A variation point is a sub node in a Aeneis process (e.g. a 'task' block in the Aeneis BPM diagram) and deals as gate to the according variation which means that the BPM diagram of the chosen variation is directly linked to this object. A variation point consists of at least one variation.
- **Variation:** A variation is similar to a root process, but with the difference, that it depends on a variation point and does not necessarily consists of variation points.
- **Node:** A node is a generic name for the above terms.

5.2 Installing the plug-in

If you have an existing pure::variants installation, you have to install this plug-in manually by unzipping and copying the unzipped files into the "plugins" directory in the pure::variants installation or into the "plugins" directory in the eclipse installation if pure::variants was installed as a plug-in project and not as a standalone. The varBPM plug-in is automatically loaded after starting pure::variants (if it was already started, it has to be restarted to apply the changes).

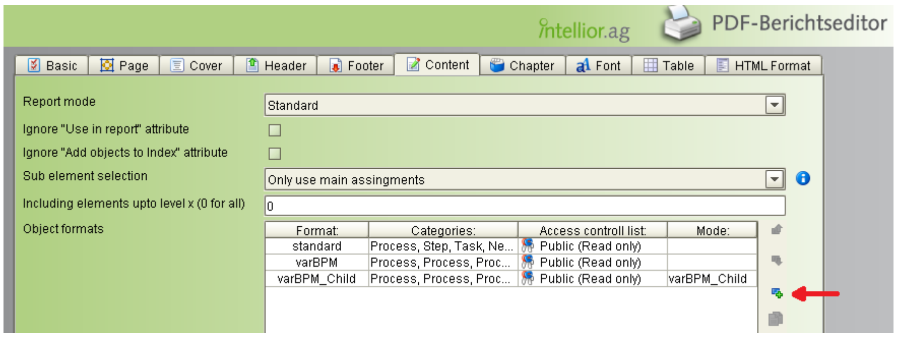
5.3 Setting up Aeneis

Depending on the wished coupling factor (see chapter 5.5.2) a few things need to be done in the Aeneis application to ensure that all linked references are evaluated if a documentation is generated. If the coupling factor was selected as low, nothing has to be done. Otherwise the following steps should be performed:

- First, one has to add an attribute called "varBPM" to each process group which should be used as a variation point. It is important that you copy this attribute to all processes so that the id of it keeps the same for each process. This attribute should be able to store references to all varBPM related process groups.

- Secondly, one has to add a user defined rule for the generation of the documentation, where the "varBPM" attribute is evaluated. The specific definition of this rule depends on the wished result, but to demonstrate the use, one very simple way is shown.

Open the report editor and open the tab "content". A page with three input boxes will appear, press the add button of the first one as can be seen in Figure 5.31 and add two new entries. The workflow steps are as follows:

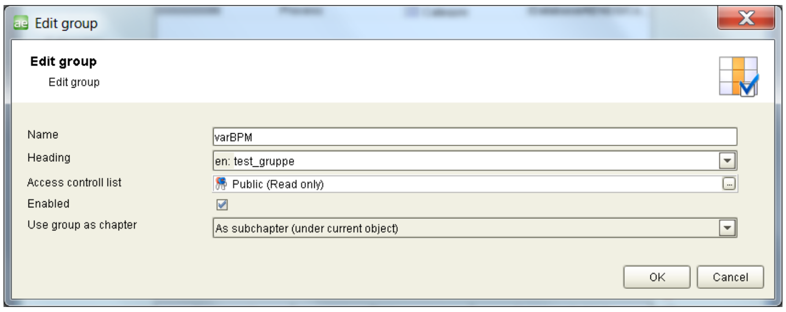


Report mode	Standard
Ignore "Use in report" attribute	☐
Ignore "Add objects to Index" attribute	☐
Sub element selection	Only use main assingments
Including elements upto level x (0 for all)	0
Object formats	

Format:	Categories:	Access controll list:	Mode:
standard	Process, Step, Task, Ne...	Public (Read only)	
varBPM	Process, Process, Proc...	Public (Read only)	
varBPM_Child	Process, Process, Proc...	Public (Read only)	varBPM_Child

Figure 5.31: Setting up Aeneis: Step 1, adding an object format to the report editor

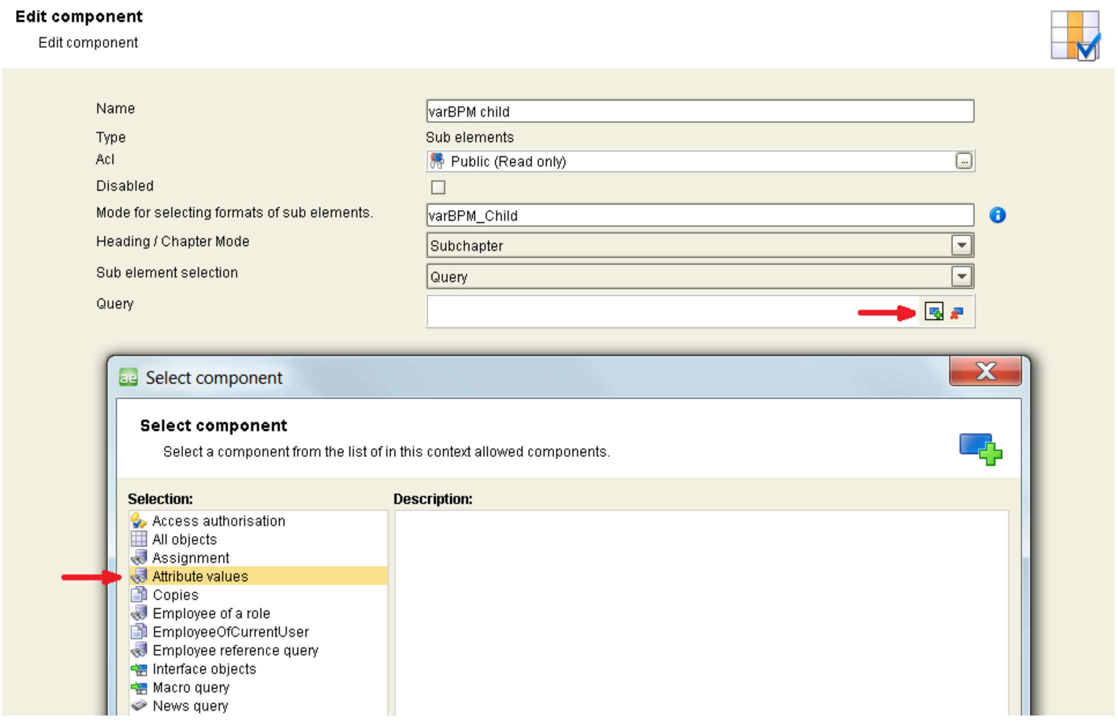
A new window will open with three input boxes. One can choose any name for the format, but the "Mode" field must be empty! In the first box (Categories), add all varBPM related process groups (in Figure 5.35 you will notice three different processes). In the second input box (Component groups), first copy the "overview" entry from the standard format into it and then add a new entry (you can choose the name and the heading as you wish; see Figure 5.32).



Name	varBPM
Heading	en: test_gruppe
Access controll list	Public (Read only)
Enabled	☑
Use group as chapter	As subchapter (under current object)

Figure 5.32: Setting up Aeneis: Step 2, adding a new "Component group"

After this is done select the newly created entry and add two more entries in the last input box (Components), whereas the second one (in Figure 5.35 called childs) can be copied from the standard format. The type of the first entry (in the Figure called varBPM child) must be ”Sub elements”. Enter any string for the ”Name” and the ”Mode for selecting formats of sub elements” field, but remember the later one (in the example varBPM_Child). Next Choose ”Query” from the ”Sub element selection” drop down list and click the add button. In the new Window select ”Attribute values” and click on the OK button (see Figure 5.33).


Edit component
Edit component

Name	varBPM child
Type	Sub elements
Acl	Public (Read only)
Disabled	☐
Mode for selecting formats of sub elements.	varBPM_Child
Heading / Chapter Mode	Subchapter
Sub element selection	Query
Query	

Select component

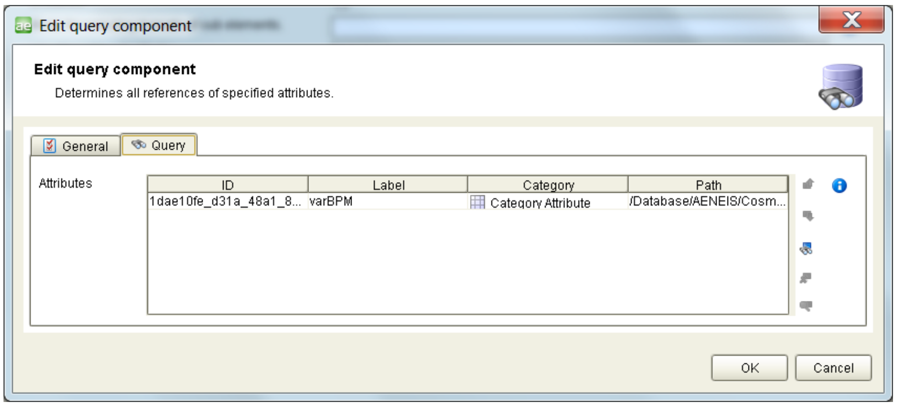
Select a component from the list of in this context allowed components.

Selection:
- Access authorisation
- All objects
- Assignment
- Attribute values
- Copies
- Employee of a role
- EmployeeOfCurrentUser
- Employee reference query
- Interface objects
- Macro query
- News query

Description:

Figure 5.33: Setting up Aeneis: Step 3, adding a component

Fill in the general settings (if needed) and switch to the ”Query” tab and click onto the ”Add reference(s)” button. A list with a lot of entries will appear where the varBPM attribute needs to be chosen (Select as category one of the related processes so that the list is much shorter). If you have copied the varBPM attribute to each process you only need to add it once (if not, you need to add it for each process).
After this is done close the edit object format dialog and create the second format. Now fill in the same string into the ”Mode” field as the one entered into the ”Mode for selecting formats of sub elements” field. The entries of the ”Component Group” input box are the same as in the standard format.



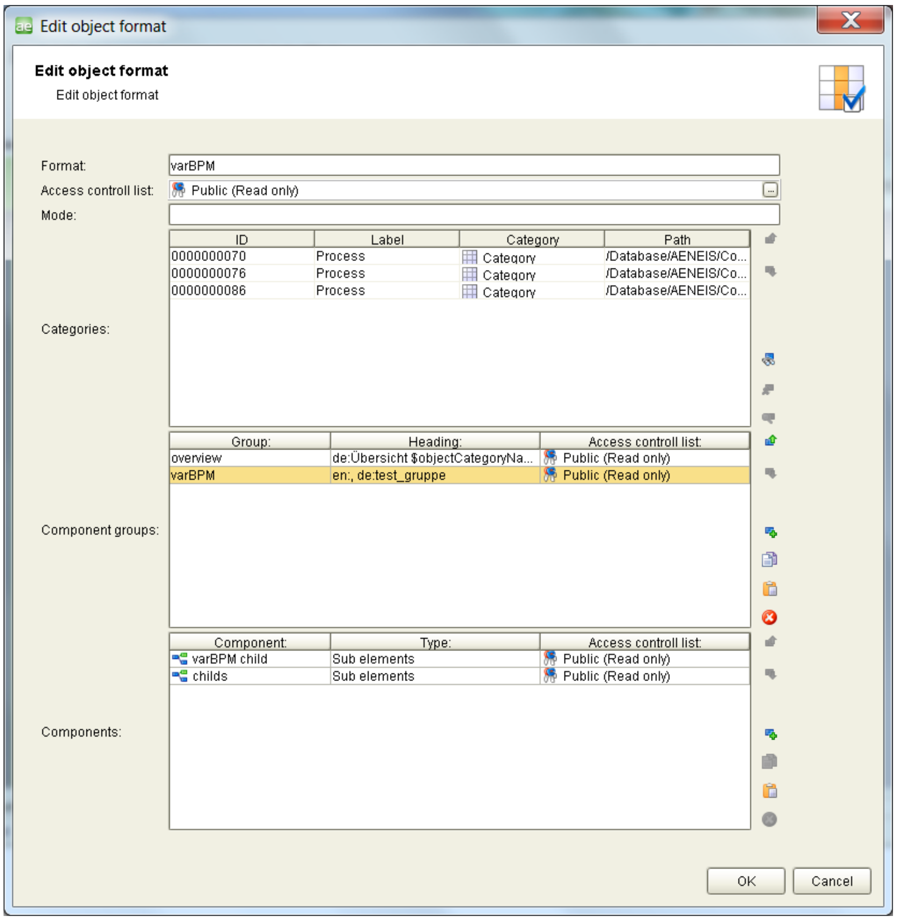
Edit query component

Determines all references of specified attributes.

Attributes

ID	Label	Category	Path
1dae10fe_d31a_48a1_8...	varBPM	▦ Category Attribute	/Database/AENEIS/Cosm...

Figure 5.34: Setting up Aeneis: Step 4, setting up the query



Edit object format

Edit object format

Format:	varBPM
Access controll list:	Public (Read only)
Mode:	

Categories:

ID	Label	Category	Path
0000000070	Process	▦ Category	/Database/AENEIS/Co...
0000000076	Process	▦ Category	/Database/AENEIS/Co...
0000000086	Process	▦ Category	/Database/AENEIS/Co...

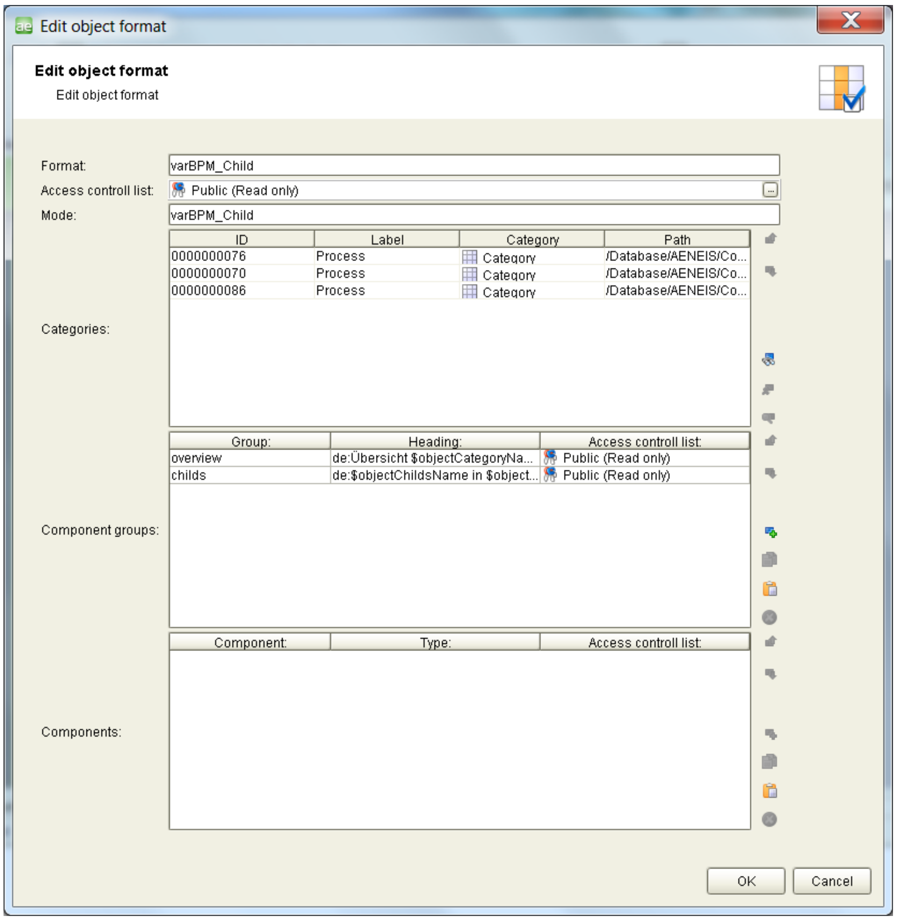
Component groups:

Group:	Heading:	Access controll list:
overview	de:Übersicht $objectCategoryNa...	Public (Read only)
varBPM	en:, de:test_gruppe	Public (Read only)

Components:

Component:	Type:	Access controll list:
varBPM child	Sub elements	Public (Read only)
childs	Sub elements	Public (Read only)

Figure 5.35: Setting up Aeneis: Step 5, final result of the object format


Edit object format

Edit object format

Edit object format

Format: varBPM_Child

Access controll list: Public (Read only)

Mode: varBPM_Child

Categories:

ID	Label	Category	Path
0000000076	Process	Category	/Database/AENEIS/Co...
0000000070	Process	Category	/Database/AENEIS/Co...
0000000086	Process	Category	/Database/AENEIS/Co...

Component groups:

Group:	Heading:	Access controll list:
overview	de:Übersicht $objectCategoryNa...	Public (Read only)
childs	de:$objectChildsName in $object...	Public (Read only)

Components:

Component:	Type:	Access controll list:

OK Cancel

Figure 5.36: Setting up Aeneis: Step 6, final result of the sub element object format

After this is done Aeneis is set up for a high coupling factor and ready to use.

5.4 Using the varBPM pure::variants plug-in

5.4.1 Contributions to the user interface

As stated in chapter 4.7, the varBPM plug-in provides a new menu entry which is called "varBPM" with two menu items "Import Feature Model" and "Update Feature Model":

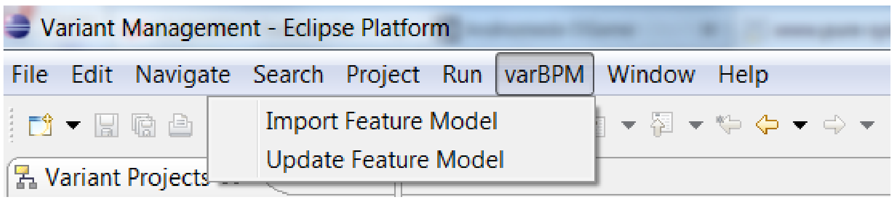
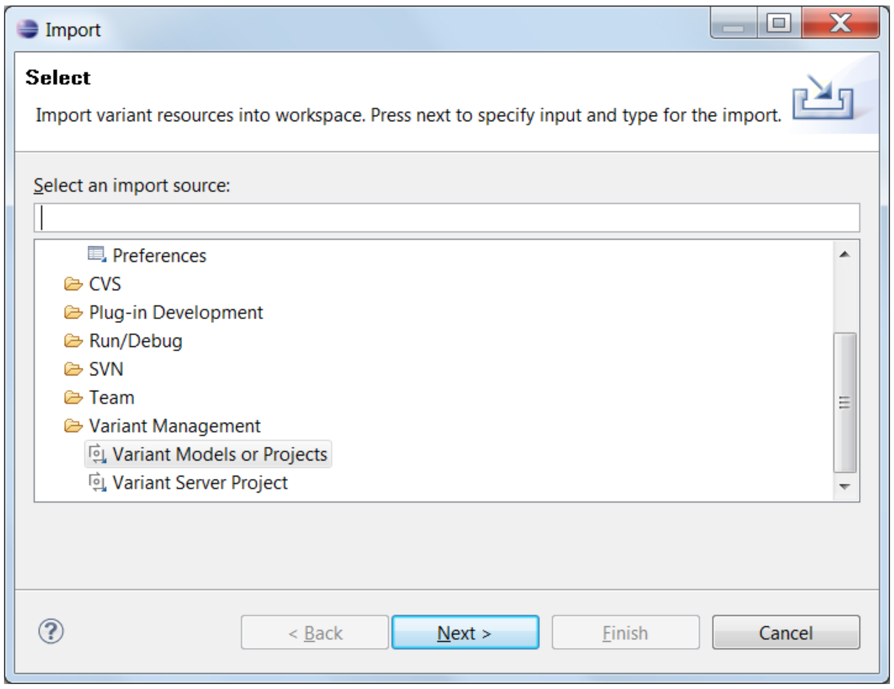

Variant Management - Eclipse Platform
File Edit Navigate Search Project Run varBPM Window Help
Import Feature Model
Update Feature Model
Variant Projects

Figure 5.37: Menu contribution of the varBPM plug-in

The import functionality can also be started over the import dialog which is opened over "File → Import". In the new opened window select "Variant Models or Projects" under the "Variant Management" folder. After clicking on the next button, choose "Import new Model from Aeneis":

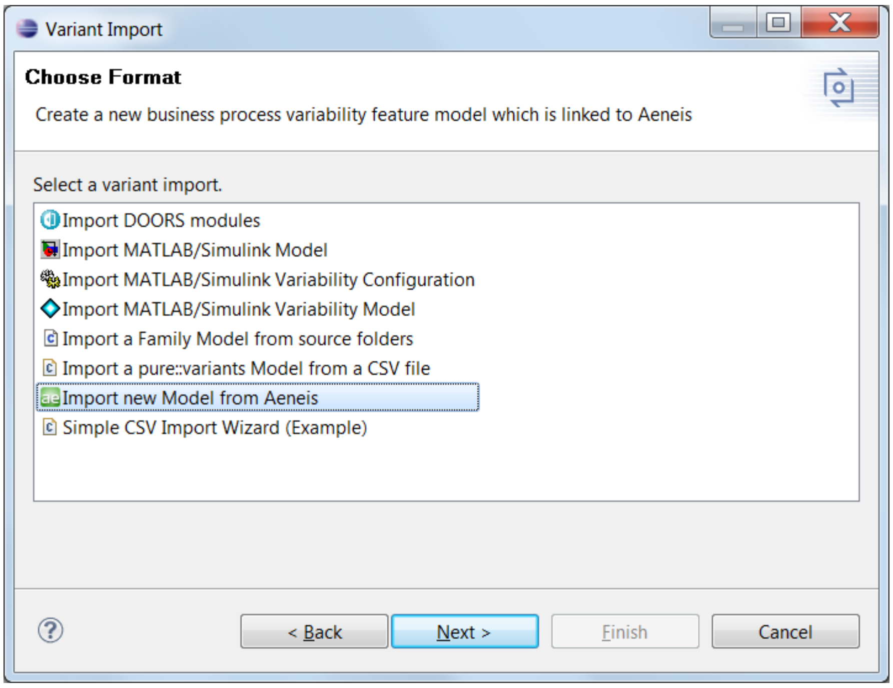

Import

Select

Import variant resources into workspace. Press next to specify input and type for the import.

Select an import source:

- Preferences
- CVS
- Plug-in Development
- Run/Debug
- SVN
- Team
- Variant Management
  - Variant Models or Projects
  - Variant Server Project


Variant Import

Choose Format

Create a new business process variability feature model which is linked to Aeneis

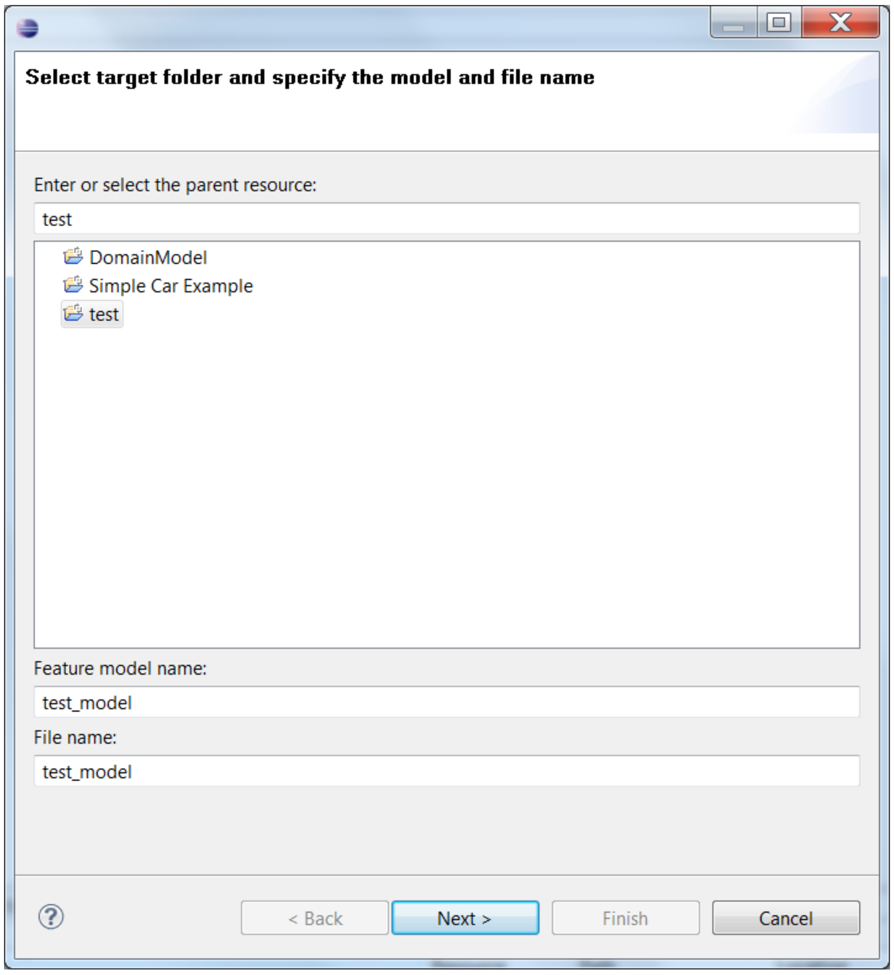
Select a variant import.

- Import DOORS modules
- Import MATLAB/Simulink Model
- Import MATLAB/Simulink Variability Configuration
- Import MATLAB/Simulink Variability Model
- Import a Family Model from source folders
- Import a pure::variants Model from a CSV file
- Import new Model from Aeneis
- Simple CSV Import Wizard (Example)

Figure 5.38: Import wizard contribution of the varBPM Plugin

5.4.2 Creating a new feature model

First off all make sure that the Aeneis application is started, because the communication of these two tools is performed online. According to the purchased licence of Aeneis you may have to close the database you want to manipulate, otherwise this plug-in will complain about a problem with the connection. Just try closing the database if pure::variants complains about it.
As mentioned before, there are two ways to start the import model dialog. Just choose one of it and proceed. On the first page, the user is asked about the destination of this model, the name of it and the name of the file. You can choose a different name for the model and the file, but the same name is recommended (if you first enter the model name, the input is also added to the file name).


Select target folder and specify the model and file name

Enter or select the parent resource:

test

- DomainModel
- Simple Car Example
- test

Feature model name:

test_model

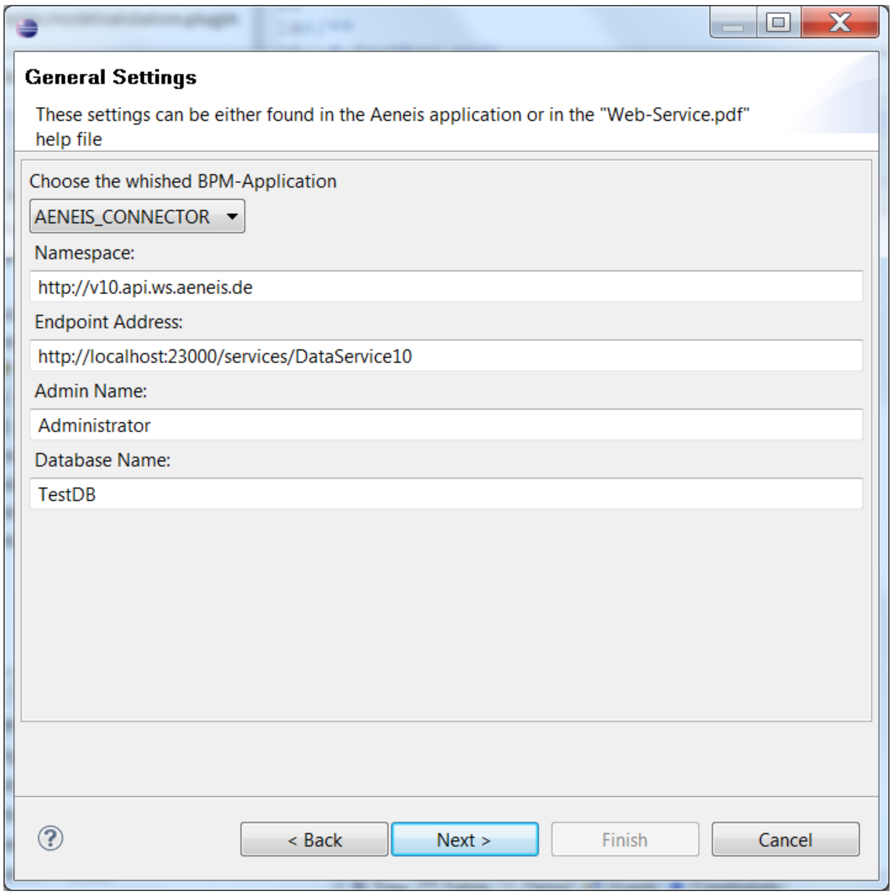
File name:

test_model

Figure 5.39: Example: Import wizard first page

A small note: the model name has to be OCL valid (the first character must be a letter or a underscore, no spaces are allowed, ...), but the Wizard will display a error message at the top of it, if something is wrong.

On the next page, the user is asked to enter some general settings of the Aeneis application. The drop down list does only contain one entry at the moment and can be ignored. Due to the fact that the connection is based on a web service, the user has to specify the namespace and the endpoint address of the server. There are already some default values entered, but according to the documentation from Aeneis, the namespace setting is subject to change in future releases. The name of the administrator of this database must also be specified which is in almost every situation the name "Administrator" (this is the reason why it is entered as default value). At last the name of the database has to be set.



General Settings

These settings can be either found in the Aeneis application or in the "Web-Service.pdf" help file

Choose the whished BPM-Application

AENEIS_CONNECTOR

Namespace:

http://v10.api.ws.aeneis.de

Endpoint Address:

http://localhost:23000/services/DataService10

Admin Name:

Administrator

Database Name:

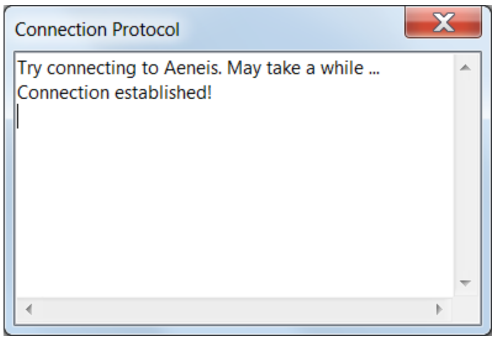
TestDB

Figure 5.40: Example: Import wizard second page

After clicking on the 'Next' button a dialogue will appear telling you something about the connection process. If everything was fine, the next page is displayed and the dialogue can be closed. Otherwise an error message is displayed complaining about the connection. This can have the following reasons:

- The Aeneis application is not started.
- The Aeneis application does not allow multiply logins to the same database and someone is locked in.
- One or more of the settings were wrong.

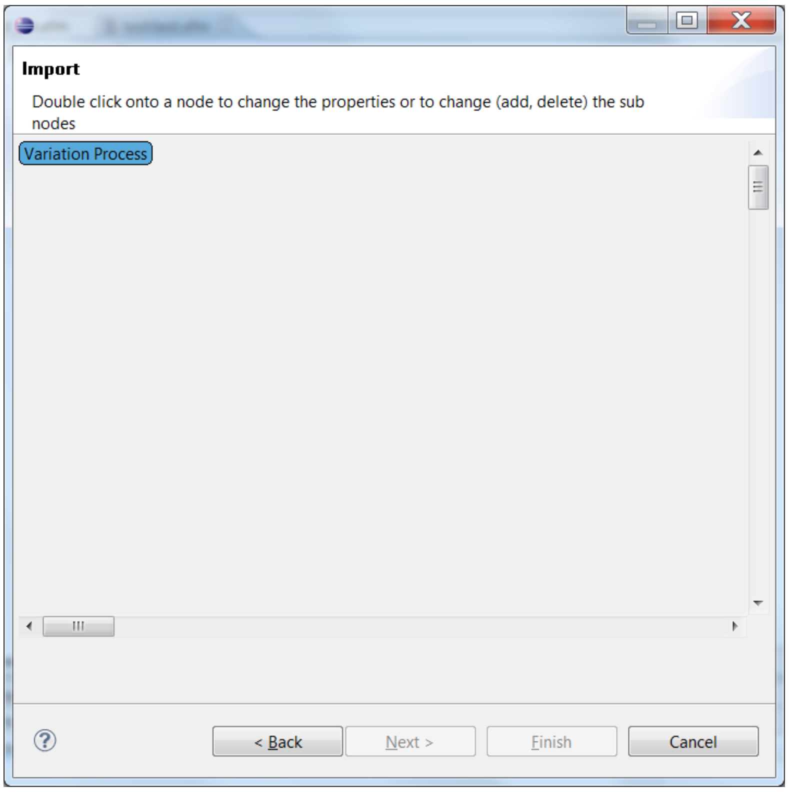
- A connection problem if Aeneis is running on a server/different machine.



Try connecting to Aeneis. May take a while ...
Connection established!

Figure 5.41: Example: Connection protocol dialogue

The next page is the heart of the import process where the variability of the business process is modelled.


Import

Double click onto a node to change the properties or to change (add, delete) the sub nodes

Variation Process

Figure 5.42: Example: Import wizard third page

The variability of the process is visualized as a tree as one can see in Figure 5.43. The structure of this tree was discussed in chapter 4.3.

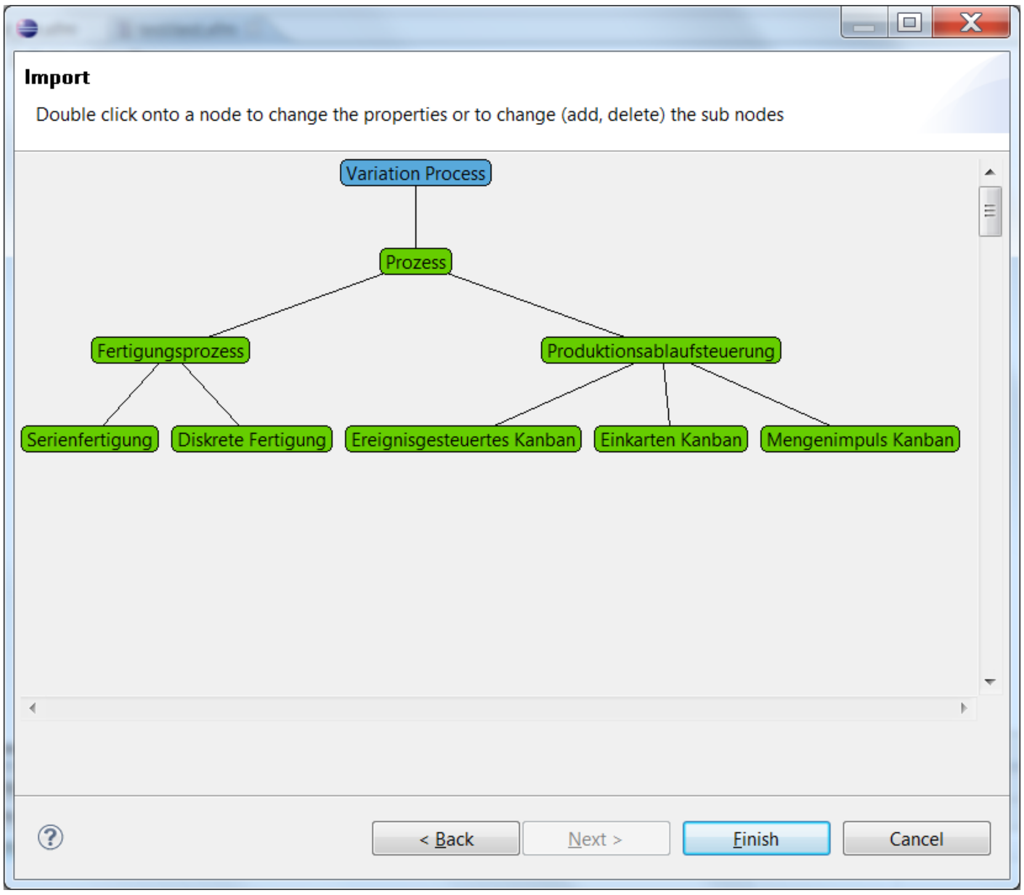
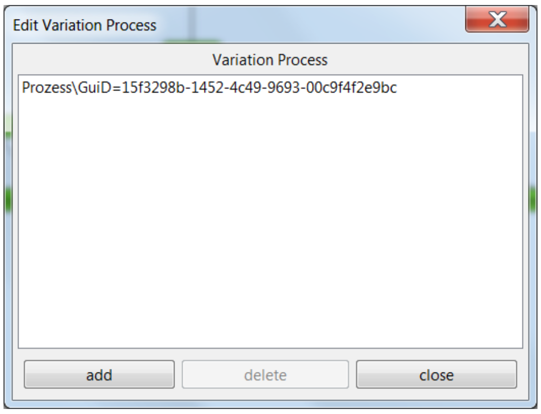

Import
Double click onto a node to change the properties or to change (add, delete) the sub nodes
Variation Process
Prozess
Fertigungsprozess
Produktionsablaufsteuerung
Serienfertigung
Diskrete Fertigung
Ereignisgesteuertes Kanban
Einkarten Kanban
Mengenimpuls Kanban
< Back
Next >
Finish
Cancel

Figure 5.43: Example: Import wizard third page with sample process structure

Due to the fact that in big projects this tree can be very big, it is at every node possible to right click on it and set this node to the root element of the view. As a result, only this branch will be displayed (right click anywhere and choosing "Restore view" will reset the view).
To edit a node, or to add new children to a node, one has to double click onto it and a new dialogue will appear (see Figure 5.44) where some values of the linked Aeneis object can be seen and a list of all children nodes. Double clicking on the node "Variation Process" does only show a list of children, because there is no according Aeneis object. This node deals only as a start point and as a little "hack" for some pure::variants restrictions.

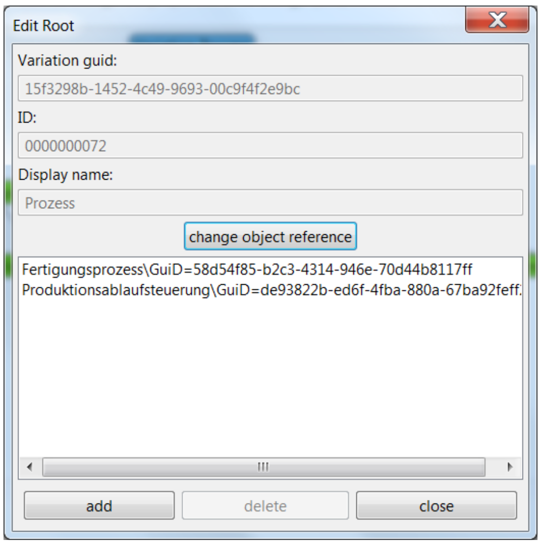


Edit Variation Process

Variation Process

Prozess\GuiD=15f3298b-1452-4c49-9693-00c9f4f2e9bc

(a) Edit variation process



Edit Root

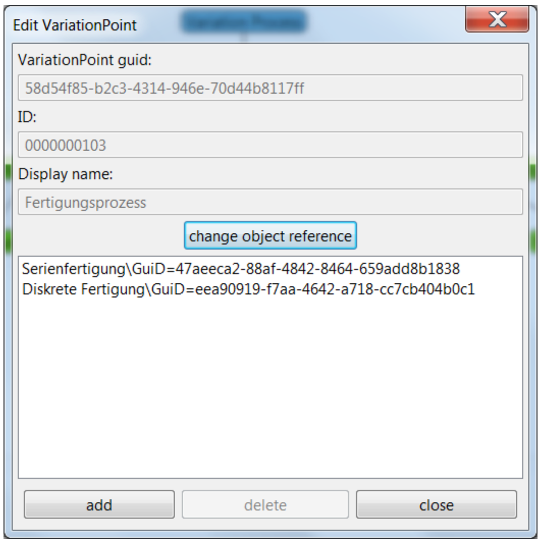
Variation guid: 15f3298b-1452-4c49-9693-00c9f4f2e9bc

ID: 0000000072

Display name: Prozess

Fertigungsprozess\GuiD=58d54f85-b2c3-4314-946e-70d44b8117ff
Produktionsablaufsteuerung\GuiD=de93822b-ed6f-4fba-880a-67ba92feff

(b) Edit root process



Edit VariationPoint

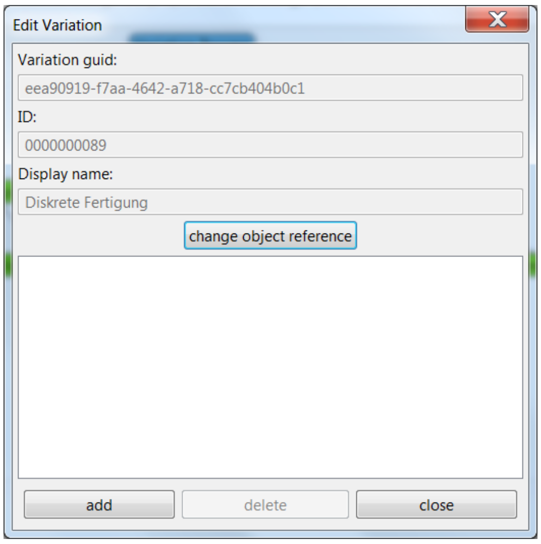
VariationPoint guid: 58d54f85-b2c3-4314-946e-70d44b8117ff

ID: 0000000103

Display name: Fertigungsprozess

Serienfertigung\GuiD=47aeeca2-88af-4842-8464-659add8b1838
Diskrete Fertigung\GuiD=eea90919-f7aa-4642-a718-cc7cb404b0c1

(c) edit variation point



Edit Variation

Variation guid: eea90919-f7aa-4642-a718-cc7cb404b0c1

ID: 0000000089

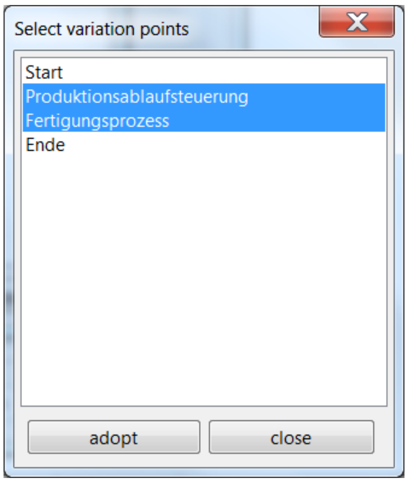
Display name: Diskrete Fertigung

(d) edit variation

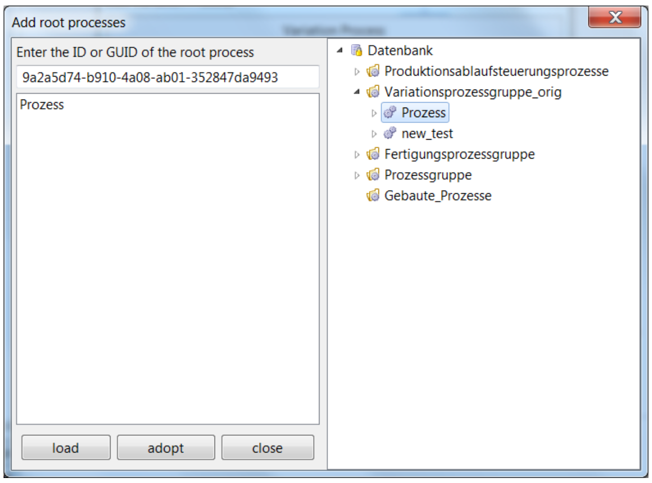
Figure 5.44: Example: Edit dialogues

If the add button in the "Edit Root" or "Edit Variation" dialog is pressed, the user will see a list of all possible nodes that can be set as a VariationPoint (multiple selections are allowed; hold the strg button while clicking onto the according elements).
If the add button is pressed in the dialogs, a new window will open asking the user to enter the ID or the GUID of the wished object to load it. It is also possible to choose the wished element from a tree viewer representing the database of the Aeneis application,

but since this is an online tool, choosing a process from the list may take a while if the database is huge. When adding Variations it is also possible to enter the group ID of the objects to load a bunch of objects at once. If not all of the displayed objects should be adopted, just select the elements you need and press the adopt button.



(a) Dialog: Select variation points



9a2a5d74-b910-4a08-ab01-352847da9493

(b) Dialog: Load object reference

Figure 5.45: Example: Load object dialogues

As mentioned earlier deleting an already added object is only indirectly possible. If the delete button is pressed, the node is coloured red, indicating that this node (and all sub elements of it) will be deleted after the finish button is pressed. If the deletion of this node should be undone double click onto it and reload the object reference by clicking onto the "Change Object reference" button.
The finish button can be pressed if the created tree is valid, which means that:

- There is at least one root process
- There is no root process without a child (except it is marked to be deleted)
- There is no variation point without a child (except it is marked to be deleted)

After finish was pressed, a new feature model is created with the following default behaviour:

- The root node contains all Aeneis settings which were entered on page two.
- All root processes are added as mandatory features which means that all of these have to be selected if you want to create a new result model in the configuration

space.

- All variation points are also added as a mandatory feature.
- All variations are added as an alternative feature, which means that only one of the variations can be selected.
- No additional restrictions or constraints

If this default behaviour is not the expected one, you can add any additional restrictions or attributes of the feature by editing the feature model with the built in pure::variants editor. There are only few things to bear in mind:

- Do not delete any of the automatically created attributes.
- Do not change the class/type fields.

5.4.3 Update an existing feature model

There are two ways to update a feature model. The first one is to use the built in model compare editor from pure::variants by clicking on the following button:

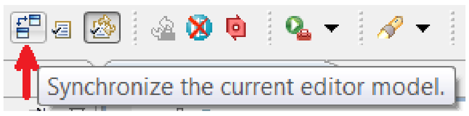


Figure 5.46: pure::variants synchronise button

Note: This button is only visible if the model is opened using the editor in pure::variants (double click on it). On the opened window you will then see two models. On the left side, there is the old original model and on the other side, there is a new one, where all elements with a missing Aeneis object reference are delete and where new variations are loaded. New variations can only be found, if a variation was added using the group id (The "added_with_group_id" member in the according feature is set to true). The manipulation is done like in any "diff" tool: For each difference one can decide to use the left model (the original one) or to use the right model.
As already mentioned in chapter 4.7.1, this method is not very mighty because it is not possible to add new nodes or delete old nodes, but it is a very comfortable way to load new variations of a group or to remove unresolved references.
The second way - and the more powerful way - is to use the update wizard which can be

started over the menu "varBPM → Update Feature Model". This wizard is very similar to the import wizard. The only difference is that on the first page, the according feature model has to be chosen. On the third page (the tree page) new nodes are coloured green, old nodes are coloured blue and nodes which are marked to be deleted are coloured red (if the Aeneis object is missing, or the user manually has marked it to be deleted).

5.4.4 Model validation

As I have discussed in chapter 4.4, there exists two types of model validation checks. One check is applied to a feature model and one to the variant model. Per default these checks are activated. To change this setting go to "Window → Preferences → Variant Management → Tab: Automatic Validation". The varBPM check engine should not cost much CPU usage if no varBPM model is opened, because the check classes always checks, if the nature of the model is the varBPM nature or not. And if not, the checks won't do anything.

To start the check engine one can either use the automatic model validation feature or press the validate model button manually:

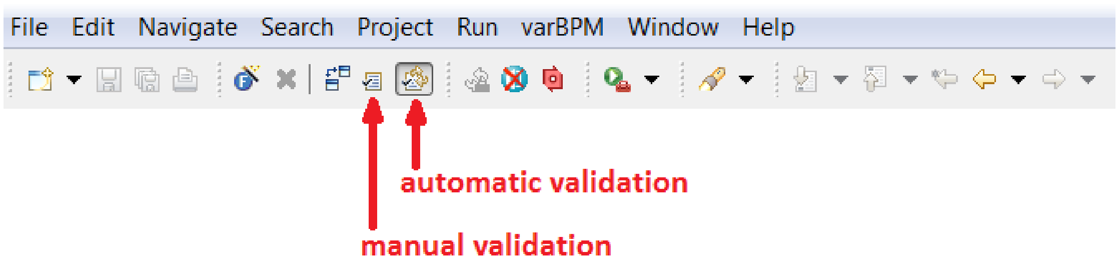
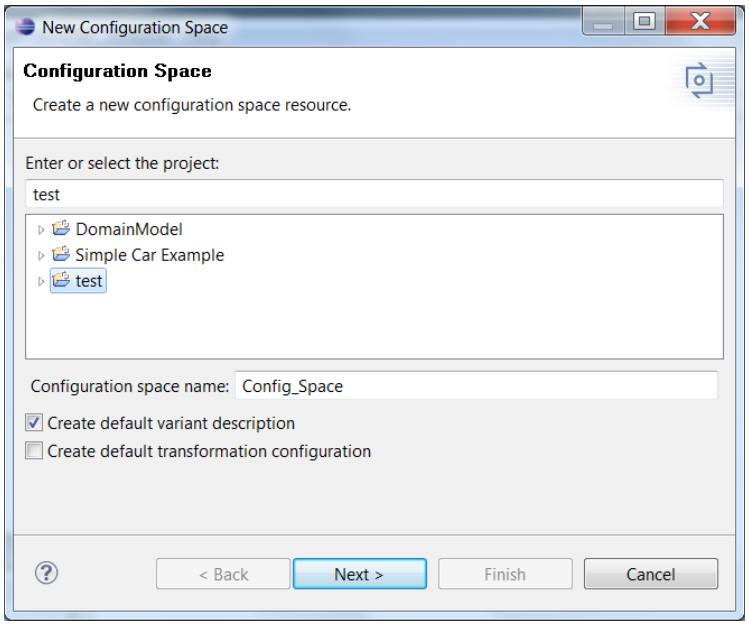


Figure 5.47: pure::variants model validation buttons

5.4.5 Model Transformation

Before a model can be transformed, a configuration space needs to be created. For this right click onto the project and choose "New → Configuration Space". A new window will appear where the name of the configuration space needs to be entered and the "create default transformation configuration" check box needs to be unchecked. Also check if the right project is selected and if not, select the right one before clicking on the next button. On the next page, check if the wished feature model is checked and press the finish button to close the wizard. After this is done, a transformation configuration needs to be added to the variant description model (.VDM file).


New Configuration Space

Configuration Space

Create a new configuration space resource.

Enter or select the project:

test

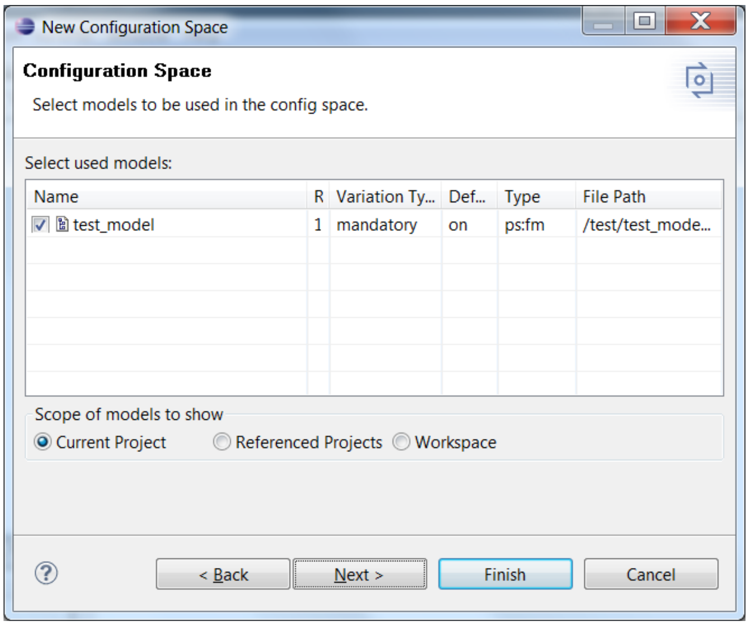
- DomainModel
- Simple Car Example
- test

Configuration space name: Config_Space

[✓] Create default variant description

[ ] Create default transformation configuration

Figure 5.48: Setting up a configuration space: Step 1


New Configuration Space

Configuration Space

Select models to be used in the config space.

Select used models:

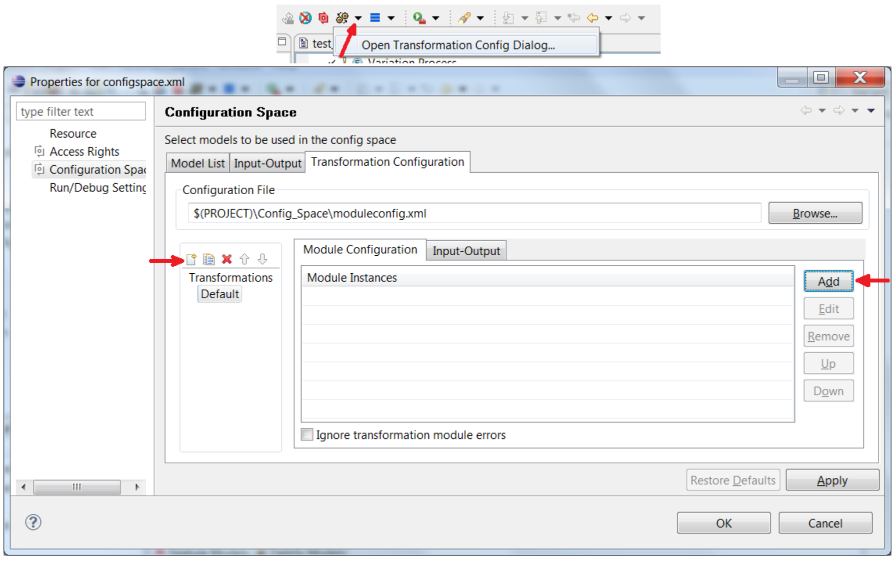
Name	R	Variation Ty...	Def...	Type	File Path
✓ test_model	1	mandatory	on	ps:fm	/test/test_mode...

Scope of models to show

[● Current Project] ○ Referenced Projects ○ Workspace

Figure 5.49: Setting up a configuration space: Step 2

For this, open the transformation configuration dialog. In the new Window click the "New" button on the left side and afterwards the "add" Button on the right side.

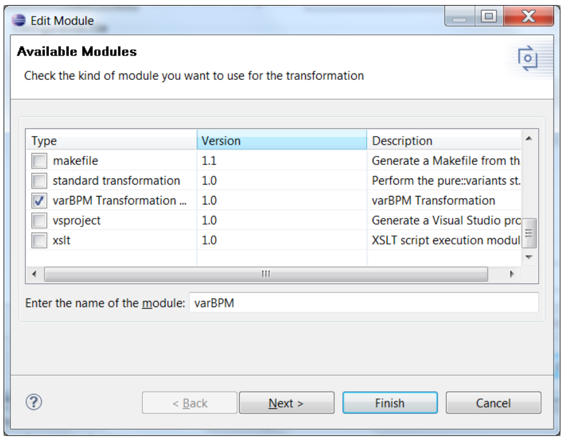


Select models to be used in the config space

Configuration File: $(PROJECT)\Config_Space\moduleconfig.xml

☐ Ignore transformation module errors

Figure 5.50: Adding a transformation configuration



Available Modules

Check the kind of module you want to use for the transformation

Type	Version	Description
☐ makefile	1.1	Generate a Makefile from th
☐ standard transformation	1.0	Perform the pure::variants st.
✓ varBPM Transformation ...	1.0	varBPM Transformation
☐ vsproject	1.0	Generate a Visual Studio prc
☐ xslt	1.0	XSLT script execution modul

Enter the name of the module: varBPM

Figure 5.51: Adding a transformation configuration

A new window will appear where a name for the transformation needs to be specified and "varBPM Transformation" needs to be selected from the list of available transformations.

5.5 Design rules in Aeneis

There are a few bunches of design rules which should be considered when designing and modelling a process in Aeneis to ensure, that the automatic model update mechanism in pure::variants works as expected.

5.5.1 Structure

The first and maybe the most important rule is, to define own process groups for classes of variations and to add these variations in pure::variants with the guid of the group. The advantage of this approach is that new variations are known to pure::variants after creating a new process in this group and after starting one of the update processes of the feature model.

5.5.2 Coupling factor

Depending on how tight you want to couple the objects (only link the according BPMN diagram to the variation point or directly link the complete object to it) one has to add a new schema attribute, named "varBPM" to the template of the process category which is designed to store links to this and other process categories.
If one can answer the following questions with 'no' a loosely coupled approach can be used, if one of the question is answered with 'yes' the tight coupling should be used (a mixed version is also possible if the tight coupling is only needed for some process categories):

1. Must the parent knows all attributes of the according child/children?

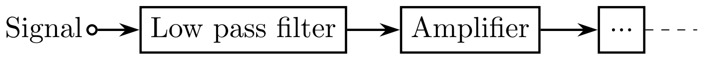
2. Is it possible, that one of the children also has children?

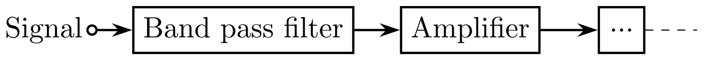
The disadvantage of the low coupling factor is, that it is not possible to iterate through the children of a child during the generation of the documentation (or maybe we were not aware that this is indeed possible). In fact it only works if the VariabilityTree has a depth of one.

5.5.3 Commonalities

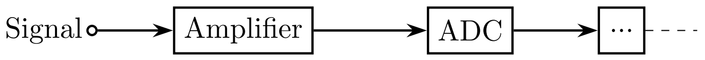
This is more an advice to ensure that the model is more reusable and more flexible to changes (refactoring can be a very long and frustrating task): Try defining the variation

points in an abstract way (e.g group some blocks together to one variation point). For example consider the following problem (it is not really a business process problem, or a meaningful example, but the idea behind the advice should be clear):
You are doing some signal processing stuff using an analogue circuit in two different ways:

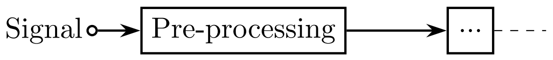




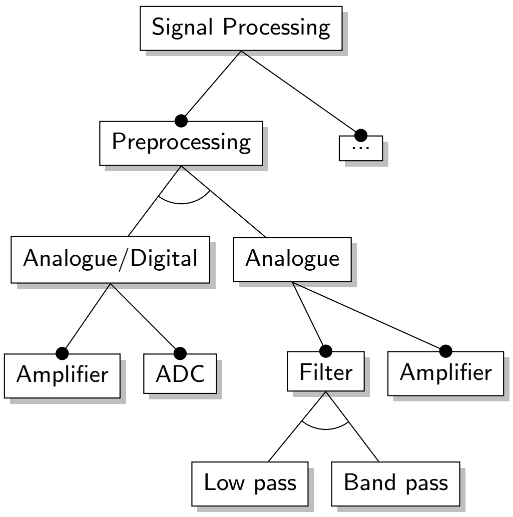
At first sight, one probably think choosing the first block as a variation point calling it filter would be a good idea, but if someday the manager comes around telling you that there should be a third variation using a half analogue (an amplifier which also deals as a band limiting circuit) and half digital approach:



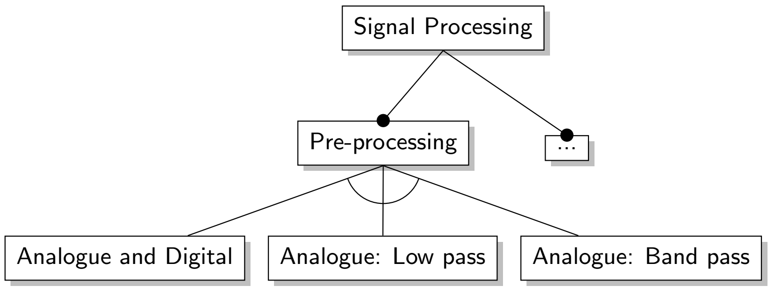
Then you will have to change the feature model a lot. But if you use a top level approach like this:



Only little things need to be changed.
Pre-processing is a variation point whereas the user can choose between an analogue approach or a digital approach. If an analogue approach is used, one can also choose which type of filter should be used. The feature model would probably look like this:



Even if the boss wishes to add a new pure digital variation for, only slightly changes has to be performed: Just adding a new variation to the existing variation point "Preprocessing" which hopefully works automatically just by updating the varBPM model after creating the according Aeneis object and adding some digital variations to it.
The trade-off of this approach is that the complexity of the system increases with each abstraction layer and the higher the number of layers the harder it is to imagine the complete system. For the above example, the feature model could probably be reorganised in such a way:



Meaning that you still have some of the advantages of the first approach but now using fewer layers. On the other hand adding some new variations possible takes longer (e.g adding an analogue variation with a high pass filter, a mixed analogue and digital approach, etc.).

6 Future work

As discussed in the early stages of the project, an automatic SAP customisation could be implemented, to not only optimize the design process of business processes but also the implementation and usage of such. This could be done by integrating the project from Andrea Leitner [And09] (respectively an adopted version) into this project or by using the connection to the SAP-Solution-Manager over the Aeneis application (providing the needed information over the varBPM pure::variants plug-in). Whereas the first approach would be more reusable since it would be independent from the used BPM modelling tool.

Another point which could be quite interesting is the implementation of an 'assembly' mode, meaning that it is not only possible to link the chosen references to the objects but also to create a new instance of the process meaning that the variation is directly added as a child to the process. An advantage of such an approach would be, that it would be more easy to use different variants of the same process at a time, but it would be more difficult to stay up to date which process needs to be updated. Another advantage is that plant dependant differences could be integrated in a smoother way, since the structure of the business process is more flexible, because it is not bound to a fixed structure.
A basic API was already implemented for the implementation of such a mode, but with no real implementation behind it.

A Detailed FODM tool evaluation criteria

Nr.	Criterion	Definition
1	Feature and Variability Modeling	• Support global constraints • Support different abstraction levels • Help to model FODA-like concepts (feature decomposition, feature type, cardinalities, dependency links, ...)
2	Constraint checking	• Support validation checking for PL model and meta-model • Check consistency of product model and PL model • Rule checking
3	Model comparison	• Compare different products • Compare different versions of a product
4	Feature meta - model maturity	• Allow to define a PL meta-model • The tool should be unambiguous • Support product line evolution
5	Extensibility	• To integrate existing platforms/products into the PL

6	Flexibility	• Changes should be possible in each development phase
7	Usability	• Intuitive usage • Easy installation and maintenance • Offer high accessibility of functions, zoom, views, ...
8	Tool stability	• Stability of API functions • Compatibility between versions (migration from old models)

Table 8: Criteria to rate FODM tools, based on [Len00], [DSF07], [DRGN07] and [And09]

B Detailed BPM evaluation criteria

Nr.	Criterion	Definition
1	BPMN compliance	• The tool should use and support the BPMN version 2.0 and all its features (swimlanes, gateways, events, ...)
2	User defined attributes / objects	• In addition to the BPMN standard, various kinds of attributes (text, pictures, files, ...) for each object or each object category should be definable • Ability to extend the existing objects with user defined objects and symbols
3	Workflow engine and approval process	• The concept of a workflow and the link of tasks to responsible persons should be possible • The approval of processes or new versions of processes is done by responsible persons and not from everyone • Right management for each user
4	Model comparison	• Compare different processes • Compare different versions of a process
5	Extensibility	• To integrate existing platforms / products into the Tool • Rich API to manipulate the database and models

6	Flexibility	• Changes should be possible in each development phase
7	Usability	• Intuitive usage • Easy installation and maintenance • Offer high accessibility of functions, zoom, views, ...
8	Tool stability	• Stability of API functions • Compatibility between versions (migration from old models)

Table 9: Criteria to rate BPM tools, derived from the project

References

[Ale05] ALESSANDRO PASETTI: Technical Note on a CONCEPT FOR THE XFEATURE TOOL. (2005). – Online available: `http://www.pnp-software.com/XFeature/pdf/XFeatureToolConcept.pdf`

[And09] ANDREA LEITNER: A software product line for a business process oriented IT landscape. (2009)

[Big12] BIGLEVER SOFTWARE INC.: Product Line Engineering Solutions for Systems and Software. (2012). – Online available: `http://www.biglever.com/extras/BigLever_Solution_Brochure.pdf`

[Cur92] CURTIS, BILL AND KELLNER, MARC I. AND OVER, JIM: Process Modeling. In: *Communications of the ACM* (1992)

[DRGN07] DHUNGANA, Deepak ; RABISER, Rick ; GRÜNBACHER, Paul ; NEUMAYER, Thomas: Integrated tool support for software product line engineering. In: *ASE*, ACM, 2007. – ISBN 978–1–59593–882–4, 533-534

[DSF07] DJEBBI, Olfa ; SALINESI, Camille ; FANMUY, Gauthier: Industry Survey of Product Lines Management Tools: Requirements, Qualities and Open Issues. In: *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*, 2007, 301–306

[FPDF98] FRAKES, William ; PRIETO-DIAZ, Ruben ; FOX, Christopher: DARE: Domain analysis and reuse environment. In: *Ann. Softw. Eng.* 5 (1998), Januar, S. 125–141. – ISSN 1022–7091

[Gab03] GABOR KARSAI, ANDREAS LANG, SANDEEP NEEMA: Tool Integration Patterns. (2003)

[Gad08] GADATSCH, Andreas: *Grundkurs Geschäftsprozessmanagement: Methoden und Werkzeuge für die IT Praxis: Eine Einführung für Studenten und Praktiker.* 5. Auflage. 2008

[Int12] INTELLIOR AG: Think Big Start Small. (2012). – Online available: `http://www.intellior.ag/fileadmin/Downloadcenter/Processmanagement_Software_BPM-Tool_Aeneis_Broschuere_EN.pdf`

[Kar99] KARLSEN, Einar W.: The UniForM WorkBench A Higher Order Tool Integration Framework. Version: 1999. `http://dx.doi.org/10.1007/3-540-48257-1_17`. In: *Applied Formal Methods — FM-Trends 98.* Springer

Berlin Heidelberg, 1999 (Lecture Notes in Computer Science). – ISBN 978–3–540–66462–8, 266-280

[KCH+90] Kang, Kyo C. ; Cohen, Sholom G. ; Hess, James A. ; Novak, William E. ; Peterson, A. S.: *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* 1990

[Kev01] Kevin P. McCormack, William C. Johnson: *Business Process Orientation - Gaining the E-Business Competitive Advantage.* 2001

[Koc11] Koch, Susanne: *Einführung in das Management von Geschäftsprozessen - Six Sigma, Kaizen und TQM.* Springer, 2011

[Len00] Len Bass, Paul Clements and Patrick Donohoe: Fourth Product Line Practice Workshop Report. (2000). – Online available: `http://www.sei.cmu.edu/pub/documents/00.reports/pdf/00tr002.pdf`

[LWK12] Leitner, Andrea ; Weiss, Reinhold ; Kreiner, Christian: MADMAPS - Simple and systematic assessment of modeling concepts for software product line engineering. (2012)

[MHS05] Mernik, Marjan ; Heering, Jan ; Sloane, Anthony M.: When and how to develop domain-specific languages. In: *ACM Comput. Surv.* 37 (2005), Dezember, Nr. 4, S. 316–344. – ISSN 0360–0300

[Mic93] Michael Hammer and James Champy: *Reengineering the Corporation - A Manifesto For Business Revolution.* 1993

[NIS93] NIST ISEE Working Group and the ECMA TC33 Task Group: Reference Model for Frameworks of Software Engineering Environments. (1993)

[Öst95] Österle, Hubert: *Business Engineering. Prozess- und Systementwicklung: Band 1: Entwurfstechniken.* Springer, 1995 (Business engineering : Prozess- und Systementwicklung). – ISBN 9783540600480

[PBL05] Pohl, Klaus ; Böckl, Günther ; Linden, Frank van d.: *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer, 2005

[Pet07] Peter Willaert, Joachim Van den Bergh, Jurgen Willems, Dirk Deschoolmeester: *The Process-Oriented Organisation: A Holistic View - Developing a Framework for Business Process Orientation Maturity.* 2007

[pur12] pure::systems: pure::variants User's Guide. (2012)

[SA98] Simos, M. ; Anthony, J.: Weaving the Model Web: A Multi-Modeling Approach to Concepts and Features in Domain Engineering. In: *Proceedings of the 5th International Conference on Software Reuse.* Washington, DC, USA : IEEE Computer Society, 1998 (ICSR '98). – ISBN 0–8186–8377–5, S. 94–

[TTC95] Taylor, Richard N. ; Tracz, Will ; Coglianese, Lou: Software development using domain-specific software architectures: CDRl A011-a curriculum module in the SEI style. In: *SIGSOFT Softw. Eng. Notes* 20 (1995), Dezember, Nr. 5, S. 27–38. – ISSN 0163–5948

[Uni13] University of Waterloo: Clafer Lightweight Modeling Language: Examples. (2013). – Online available: `http://www.clafer.org/p/examples.html`

[WL99] Weiss, David M. ; Lai, Chi Tau R.: *Software product-line engineering: a family-based software development process.* Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1999. – ISBN 0–201–69438–7

[Xpe13] Xpert Line: IVY Business Process Management for You. (2013). – Online available: `http://www.xpertline.ch/pdf/en/prospekt_xpertivy_xpertline%20en.pdf`