Model-Based Implementation of Meta-Variability Constructs: A Case Study using Aspects

Klaus Schmid, Holger Eichelberger
University of Hildesheim, Institute of Computer Science
Marienburger Platz 22
D-31141 Hildesheim, Germany
+49 5121 883 761/
{schmid, eichelberger}@sse.uni-hildesheim.de

Abstract

In this paper, we introduce the concept of meta-variability, i.e., variability with respect to basic variability attributes like binding time or constraints. While the main focus of this paper is on the introduction of the concept, we will also illustrate the concept by providing a case study.

The case study will feature a simple implementation environment based on aspect-oriented programming and will include an example that will exhibit some key characteristics of the envisioned production process.

1. Motivation

Product line engineering has become increasingly recognized in the last few years as a successful approach to dramatically reduce costs, reduce time-to-market and improve quality. It has also achieved significant acceptance in industry [12].

Along with the increasing recognition of product line engineering in industry, various approaches to variability modeling were proposed. In particular, feature modeling concepts are widely discussed and partially used in industry [6, 9, 10, 16], but a large range of other approaches have been proposed as well, e.g. [4, 14, 17, 19, 20].

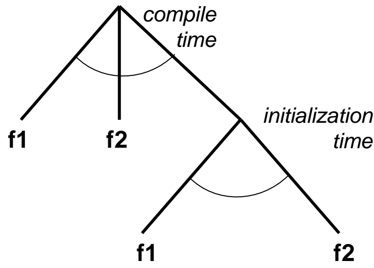
So far all these approaches share (at least) one commonality as they focus on variability as variation of specific attributes of the final product (e.g., functional or non-functional properties). Thus, they neglect the variability that may occur with respect to the characteristics of a specific variability itself. We term such variability of a variability attribute *meta-variability*.

Thus, meta-variability relates to variability that may occur with respect to production processes of subsets of the product line. For example, for some products a specific variability may be bound at compile time, while other products still support a sub-range of variability and the final binding happens during product initialization. Thus, the binding time itself may vary. Similar examples can be given for other variability attributes as well.

At first glance, the issue of meta-variability may seem very esoteric; however, it is firmly grounded in industrial practice.

A simple example, which we observed in one company, was that some variation was relevant to both high-end and low-end products. While the low-end product was produced in high volume, the high-end product was produced only in low volume. As a consequence, their production processes were different due to economic reasons. Some variations were common to these products, but for the low-end product, which also had less memory and processing power was reduced, the variation had to be bound at compile time. This way, each variant was produced independently and as this did save resources in the final variant and thus production costs, this was cost-effective.

On the other hand, for the high-end product the variability had to be bound at initialization time (prior to sales, but after shipping to country offices). The reason for this was that sales personal in the various countries could determine the final product variant in order to adapt to fluctuations in demand. As the volume of these high-end products was low, the added production costs were not relevant in comparison with the increased flexibility.



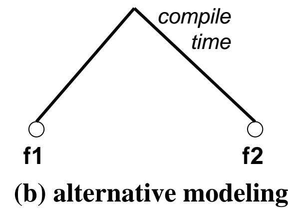


Figure 1 (a) Reification of binding time **(b) alternative modeling**

This episode shows a clear need for variation with respect to binding time. More precisely, two different binding times could be selected alternatively for the same variability.

In the following sections we will first discuss different approaches to dealing with meta-variability. In Section 3, we will then describe some basic concepts of a prototypical production environment that we used as a basis for supporting meta-variability. In Section 4, we describe a simple case study. Finally, in Section 5 we will provide our conclusions and illustrations of possible future work.

2. Dealing with Meta-Variability

To our knowledge, the problem of meta-variability has so far not been explicitly addressed. Only few approaches explicitly allow multiple binding times for a single variability. One example is [19], however, the approach does not provide a precise interpretation and semantic foundation of multiple binding times. An early case study that used this concept is described in [18].

While the problem sketched above is not uncommon in industry, so far we have not seen it being fully addressed. Instead an approach is typically taken that can be interpreted as *reification*: the meta-variability is represented as a different variability in the form shown in Figure 1a.

As this figure shows, reification leads to duplication of the respective variant information. In practice, this is often handled implicitly, by modeling the variation as something like Figure 1b and handling the case that both variants are selected in a special manner, by providing additional initialization code, without actually modeling this as variability. However, this actually means that variability is only partially modeled. In current industrial practice, where variability is usually not modeled at all, or at least not completely, this is not yet an issue, but if the underlying goal is to move towards more systematic (and automatic) variant-based software production processes, this becomes a major problem. Thus, we propose to deal with meta-variability explicitly and to accept it as a first class modeling element. This implies that variation of variability attributes must be modeled explicitly. This approach in turn provides the advantage that the various product instantiations can be produced automatically. We will describe this approach below and discuss it based on a case study.

3. Concepts of a Production Environment

In this paper, we provide a vision of how future software production environments that explicitly support variability can look like. We do not yet present a full-fledged product derivation environment.

We sketch a production environment that supports the automatic product derivation and instantiation of variability. The core idea is to use the variability model, together with the current variant selection to produce the binding of the variant parts. Depending on

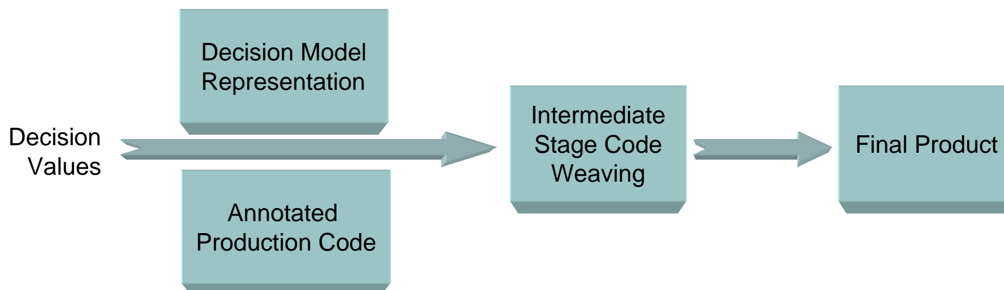


Figure 2 The Production Process

the binding time, the variant parts are bound in a different way. Thus, variation in binding time directly influences the model-based production process and leads to the production of different code. This is depicted in Figure 2.

The specific form of the production environment and of the production process depends on numerous parameters, including the type of artifacts supported, the programming languages used, etc. Here, we will focus on a particular example based on Java, which we tried to keep as simple as possible for illustration purposes. Of course, the implementation could be combined with context-oriented or feature-oriented programming [5, 13], instead of the straight-forward approach we use here. However, the key issue we address is not the implementation technique, but the need of communicating meta-variability in the product instantiation process between product developer and the final product. This goes beyond the existing approaches.

In order to evaluate the feasibility and appropriateness of this idea, we constructed a simple, prototypical production environment. The core idea of this environment is the stringent separation of functional code and the variability implementation, which we achieved in a very simple way for our example, a more sophisticated approach could include ideas from feature-oriented or context-oriented programming.

3.1 Domain Engineering

The approach to product line modeling which is used in our case study is based on decision modeling, an approach initially devised in the Reuse-Driven Software Process Guidebook [20]. The approach has been later extended in several ways, e.g., [4, 19]. Here, we will build in particular on the extensions as described in [19]. However, we believe the basic approach is not specifically influenced by the choice of variability modeling approach and could be integrated in a similar way with feature modeling or other approaches.

In accordance with the decision modeling approach, we capture the decisions that are relevant for deciding about the product characteristics. Further, we allow that for a single decision multiple appropriate binding times can be recorded. So far, we have not yet extended the modeling mechanism as far as enabling to define constraints on binding times, thus, the semantics is simply (as was initially defined in [19]) that during instantiation any of the previously specified binding times can be chosen for a concrete decision.

Besides the variability model (here the decision model) the basic information about the product line must be modeled. In general, this can happen in an arbitrary modeling language, respectively, by a combination of multiple modeling approaches. There is no specific restriction with respect to forms of modeling that can be used. In the case study, that we will discuss in Section 4 we will restrict our domain modeling actually to a Java implementation as this is a widely used and well-known implementation language. Of course, different types of artifacts may only allow for certain variability attributes. For example, a runtime adaptation of a UML model does not make too much sense.

Finally, a relation must be established between the various decisions and the artifacts that are impacted by these decisions. A high-level categorization of mechanisms to realize artifact variability was given in [20]:

- Physical Separation, i.e. to represent variant elements as physically distinct entities, e.g. as separate files.
- Target-Language specific mechanisms, e.g. templates, generics, alternatives in combination with constants, etc.
- Metaprogramming mechanisms to superimpose a language for handling variations on top of the target language.

In our example, we will exclusively rely on mechanisms that are yet supported by the target language and in related IDEs, i.e. existing editing and refactoring mechanisms or well-known add-ons e.g. by using available IDE plugins and libraries.

As one interesting approach aspect-orientation lends itself to variability implementation. Thus, we decided to use AspectJ as part of the production environment. Of course different choices would have been possible as our case study does not depend on specific realization techniques. The realization of variability will be done by referring to the production code with pointcuts as defined in aspect oriented programming [11] and by the use of special purpose variables. The use of special purpose variables is a target-language specific mechanism that, in particular, relies on the target language compiler (e.g. static evaluation of constants, code elimination and inlining). Aspect oriented programming may appear as a metaprogramming mechanism (e.g. additional keywords are introduced as in AspectJ versions prior to version 5) or as a mechanism that relies on meta-information represented by constructs of the target language (support of Java annotations since AspectJ 5). Combined with physical separation and conditional packaging of the resulting binaries, aspect oriented mechanisms act in our example as a tool to easily realize binding at startup time and runtime. An alternative, probably with some more architectural effort, could be applying layers of collaborations or polymorphic selection and default objects [15].

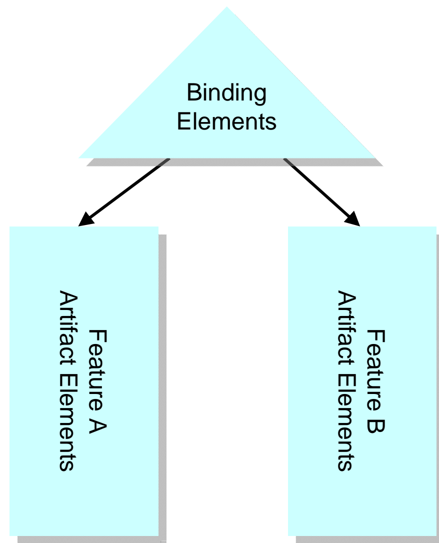


Figure 3 Instantiated Artifact Elements Model

As stated earlier, the specific form of the realization of the instantation technique depends on a number of parameters, including the programming language. Other programming languages will require explicit preprocessing mechanisms (categorized as metaprogramming in [20]) like the well known C preprocessor. As we will rely on Java in our example, such preprocessing mechanisms are substituted by the use of if-clauses with constant expressions, which are defined in Java to be equivalent to preprocessing. We combine this with target-language specific mechanisms facilities of the core language like special purpose variables and aspect oriented programming as described above.

3.2 Application Engineering

The aim of application engineering is to derive the final product. Thus, based on values for the various decisions (including the determination of the binding time), feature artifact elements from the product line model must be selected and combined (cf. Figure 3). So far, this is very much the standard approach as it has been realized in numerous other product line modeling tools (e.g., [3]). This becomes more of an issue as soon as binding times that involve runtime decisions must be addressed. In this case, the code that binds together the various feature implementations should be generated.

In our approach, we generate variability code from the decision model information and combine it with a simple runtime part provided by the production environment. Thus, if values and binding time are assigned to a decision so that full instantiation is possible at development time, the necessary artifact elements are combined. If runtime decision making is required (e.g., during start-up), this decision must be modeled on the level and in the representation of the artifact in question. In our case study, as we will only deal with Java Code, this will relate to the actual activation code.

This concept is shown in Figure 3. This figure shows two possible artifact elements. They are related by binding elements. These binding elements may take different forms, depending on the kind of decision taken and the binding time:

- In case the decision is taken to have only one of those elements (and this is valid at development time), the binding element simply needs to integrate the artifact element with the remainder of the model.
- In case the decision is taken to have both elements present at runtime (and this decision is taken at development time), the binding element must become a connector that connects both elements simultaneously.
- In case the decision is taken to have one of the elements, but the final decision which one is taken at runtime, this needs to turn into a connector that is evaluated at runtime.

Though this discussion is still rather abstract at this point, it will probably become somewhat clearer as we illustrate it in the next section based on a case study.

4. Case Study

In order to analyze the possibilities and implications of making meta-variability explicit and treating it as a first class citizen, we conducted a case study based on an existing software system, with which we were already well acquainted: the SVNControl system [2].

4.1 Prototype Realization of the Production Environment

The prototypical production environment contains

- the domain modeling view, an Eclipse plugin, which maintains the decision definition table from domain engineering (see section 4.3). The editor allows creating, editing and deleting domain decisions. In particular, for each domain decision the allowed binding times, i.e. the binding range, and a value range (currently boolean values, arbitrary integer ranges and arbitrary enumerations are supported) constraining the instantiation of the decision can be specified. The domain definition table is stored as a file in XMI format.
- the product derivation view, also part of the Eclipse plugin used to specify the values of con-

crete decisions (as described in section 4.4). The derivation view allows to provide values to non-instantiated decisions, and the editing and deleting of the concrete decisions. The concrete decision values are stored as a file in XMI format, which is linked to the domain definition table file.

- the code generator (as described in section 4.3) and build management support, i.e. appropriate ANT [1] tasks to
  - Configure and run the code generator by specifying the product decisions file and the names of the classes to be generated.
  - Optionally execute a Java specific C-style preprocessor based on the values of the product decisions. Currently, the preprocessor is intended to prepare the environment for other target languages.
  - Clean up empty class files that result from the execution of a preprocessor.
- the runtime core to be included into the final product if decisions are left to the user and must be made e.g. during startup or runtime. The runtime core contains a default mechanism for user decision making and the implementation of value range types in order to validate the user input.

The individual parts of the production environment will be described along with the case study in the next section.

4.2 The Base System

Our case study is based on the SVNControl system, which provides a network based management interface to the subversion system. The system was initially developed at the University of Hildesheim, but has been released as Open Source [2]. Currently, it is still under development and has been taken up by organizations like UBS, GDV (association of German insurances), and many others.

SVNControl is a remote administration tool with graphical user interface for the version management system Subversion. SVNControl supports the administration on repository level (e.g. to create, rename or delete repositories consistently), user or group level, access permission level and on scripting level, i.e. to take control over several hooks controlling valid check-ins etc.

Basic administrative functions are relevant to all users and should therefore be treated as commonalities in a product line. However, some more advanced features like scheduling of permissions as well as scripting and hooks are candidates for a special distribution for advanced administrators. Based on discussions with users, also the entire user management is in question in some environments, because often in organizations, users and user attributes like groups are centrally administered, e.g. by LDAP or ActiveDirectory and therefore, this functionality should not be available.

While currently the product is built without variability, we decided it makes a good case study, as a need for variability can be clearly identified and the code is well known to the second author.

4.3 Modeling Variability

Following the brief introduction of SVNControl in the section above, we will now discuss how we represented the variability using our prototypical production environment in terms of the decision model approach as described in [19]. Due to space limitations, we will restrict ourselves to the variabilities for the scheduling and the hook functionality.

The definition of a decision consists of:

- A unique name used to reference the decision.
- The relevancy specifies the circumstances under which the definition is meaningful.
- A textual description of the decision.
- A range to define or restrict the values that the decision can take. The cardinality defines how many values the decision (seen as a set) may have.
- Constraints among values of the various decision variables
- Binding times: Define a range of points in time, which describe when the decision can be bound to a concrete value.

As mentioned above, we will discuss two decisions in this case study. They refer to the capability of the resulting product to administrate

- permissions using a scheduler.
- the hook scripting mechanism.

Figure 4 depicts the decision modeling view of our production environment showing those decisions for SVNControl. The definitions of the decisions can be maintained in the domain decision table in the upper left part. In particular, the user can specify a value range (e.g. boolean values) and an individual binding time range for each decision (in the lower part). In this case study we will neither consider the relevance nor the constraints of the decisions.

When the domain decision model is specified, the information on the decisions must be transformed into source code related information. Therefore, depending on the binding time range of the individual decisions, the code generator of our prototypical production environment will produce a set of constants for each avail-

able binding time. We will now discuss how the relation between the decisions and the variant artifacts can be realized for Java as target language.

In Java, compile time decisions can simply be represented as constant values to be evaluated as expressions in alternatives (if-statements), because, according to the Java Language Specification [8], the compiler will evaluate the constants during compile time and inline or exclude the source code influenced by the alternative. Consequently, the if-statement in Java in combination with constants acts like a preprocessor statement in other languages. Of course, it would be clearer to have an explicit preprocessing step, but this is the way Java is defined. This confusion of preprocessing-IF and runtime-IF can be considered a shortcoming of the Java-language definition.

For example, if the decision is made at compile time that the scheduling functionality should be available, the code generator will produce a constant class containing a constant named according to the identification of the decision and initialized with the given default value as follows:

```
/**
 * Configuration constant for the
 * decision "Can user or group …?".
 */
 public static final boolean
     OPT_SCHEDULES = false;
```

The production code itself may now contain appropriate alternatives depending on the value of the compile time decision, e.g. in our case study code to display the related GUI elements in the case that the decision value is `true`.

So far, this was standard implementation of a compile time variability. However, the interesting part is that the production environment can handle also startup and runtime binding for the same decisions.

For these decisions more information must be taken into account in order to construct an appropriate decision-making mechanism at startup time or runtime. Taking the domain decision table as input, the model-based generator will produce object constants (enum values) that will carry additional information to be provided to the runtime decision-making mechanism. Even if constants are generated, the related concrete decision values may change during runtime of the program, e.g. using a dialog which initializes itself according to a set of these enum constants. The following source code fragment depicts one of the produced enum constants showing also some additional information from the decision model like the description and the value range:

```
/**
 * Configuration constant for the
 * decision "Can user or group …?".
 */
OPT_SCHEDULES("Can user or group…?"
     ,BooleanValueRange.
     BOOLEAN_RANGE, …),
```

Only two more steps are needed to make startup or runtime decisions work: The decision-making dialog must be called at an appropriate point of time in the production code of SVNControl and SVNControl itself must be able to react when a certain decision is made.

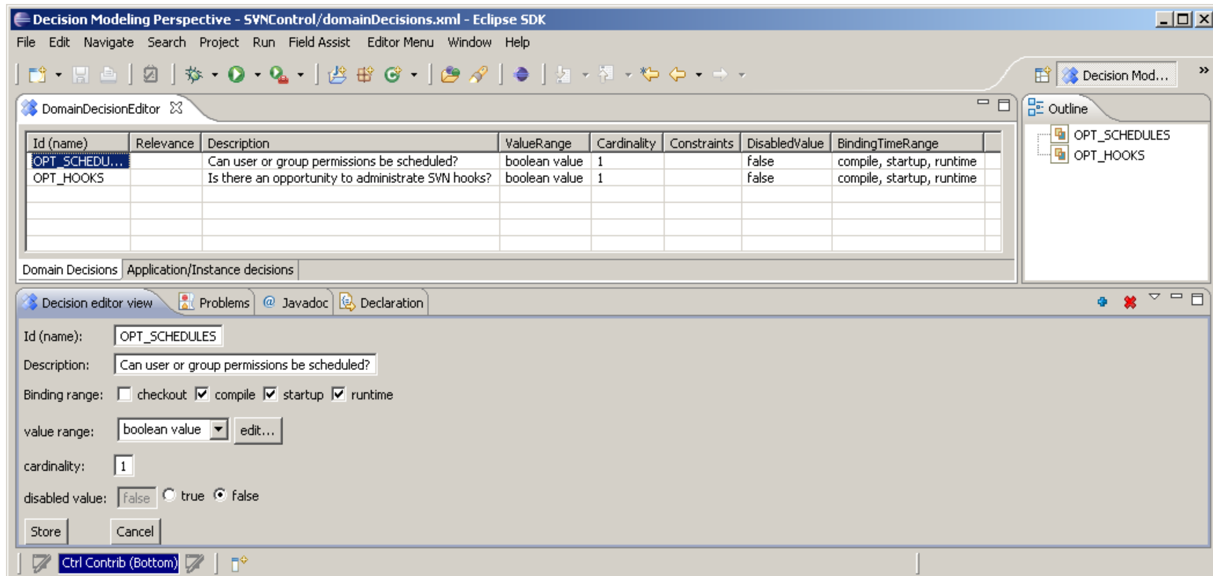


Figure 4 The decision modeling view of our production environment.

To address both issues, we resort to using AspectJ pointcuts to introduce startup calling code and to insert the necessary code fragments from runtime variability into the code. Other approaches (e.g., preprocessing) could have been used as well for this task, however, we use AspectJ as it provides rather clear and well-known mechanisms. In addition, we used a legacy system as a basis for our case study. Thus, it was a key requirement to use technologies that enable the introduction of variability with as little restructuring of architecture and code as possible.

The examples shown in this case study are given in the old style of AspectJ – AspectJ 5 facilitates Java annotations so that no proprietary keywords are necessary anymore. From a strict viewpoint, using the old style of AspectJ implies the use of a metaprogramming approach that is outside the target language. We will ignore this minor issue here, because the example could easily be refactored to new style and the well-known notation simplifies reading this paper.

Using AspectJ, the Java compiler as well as the runtime environment of the final product will be enhanced by code weaving facilities as indicated in Figure 2. The following example shows the pointcut related to the startup time decision for permission schedules in SVNControl. `StartupConfiguration` is assumed to be the enum class produced by the code generator for startup time decisions. Method parameters are not shown to keep the example simple.

```
aspect Startup_Schedules {

 pointcut myClass():
  within(MainWindow);

 pointcut myMethod(): myClass() &&
  execution(void MainWindow.
  initializeDynamicElements());

 before(): myMethod() {
  if (StartupConfiguration.
   OPT_SCHEDULES.getBooleanValue()) {
    // activate the scheduler UI
  }
}
```

The code artifacts to be injected for runtime decisions look similar. A runtime related aspect defines the call to the decision-making mechanism, e.g. as an action of a special menu item. To notify SVNControl about changes of the runtime decision value, the pointcut may register an observer [7] in the runtime core of our production environment. The concrete application may then react appropriately when a value is changed, i.e. in the case of SVNControl by enabling or disabling GUI parts related to the decision.

Due to the architecture of SVNControl, all variabilities sketched in this paper can be realized in a similar way as presented in this section.

By providing information on all binding times of each decision specified in the domain decision table and by separating the binding time related code into several aspects, we gain the flexibility to relate the realizing artifacts to decisions at development time and to postpone the decision on the concrete binding times until product derivation time and finally to smoothly switch among the available binding opportunities while product derivation time. In the next section, the mechanisms related to product derivation will be discussed.

4.3 Deriving the Products

Based on the results of the variability modeling and the domain engineering, i.e. the domain decision table, the initially generated constant sets for the supported binding times and the (implemented) pointcuts, now our production environment can be used to derive concrete products by instantiating the decisions and to build the individual products.

Using the product derivation view shown in Figure 5, the product engineer can now determine the concrete (initial) value, the binding time of all decisions previously specified in the domain decision table and the related code artifacts. Then, by executing the model-based code generator, the constant sets in the production code are (re)generated and the concrete values are stored in the production code of SVNControl. In particular, this step is important for the compile time decisions, because it determines the concrete values of the constants. Thereby, additional build information for the following build steps is gathered in order to give the current decision values control over the binary packaging process and, therefore, to influence which classes or pointcuts will be present in the final product. The next build step removes existing binaries from previous builds and calls the compiler with respect to the relevant pointcuts. Finally, based on the generated packaging information, only the binaries related to the selected decisions will be assembled together into executable Java archives.

4.4 Results

We have applied our prototypical production environment to introduce and realize compile-time, startup-time and runtime decisions in the context of the SVNControl application. Based on the decision model maintained by the domain and application modeling view of the Eclipse plugin, the generated constant classes, the runtime part and the build support, it was easy to realize the discussed decisions and therefore the intended variability. In particular, with little overhead also binding times postponed to application decisions can be realized easily.

The code produced by the model-based code generator is easy to read and fits to usual source code conventions. Beside tests whether the intended binding time for the decisions is realized and functional, we were also interested, whether the proper binary parts appear in the packaged result and whether the binaries related to disabled binding times disappear. Therefore, all tests were carried out after rebuilding and repackaging SVNControl. The intended functionality was fully functional and only the binary parts selected by the binding times in concrete decisions appeared in the packaged application, i.e. unintended binary fragments were completely absent. Thus, the production process that has been set actually achieved its underlying goals.

5. Summary and Outlook

In this paper, we argued for the importance of meta-variability. This concept describes the variation of variation attributes (as opposed to the mere variation of product characteristics). Meta-Variability, especially in its form of binding time variability is actually relevant, but has so far not been dealt with.

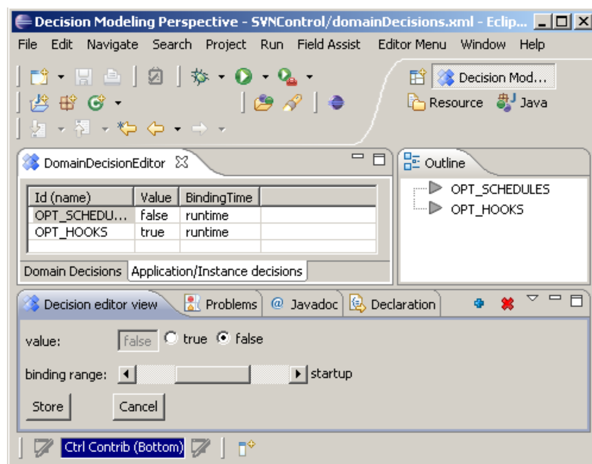


Figure 5 The product derivation view of our production environment.

We showed that by setting up an adequate production environment for variant code, we are able to deal with binding time variability very easily and in a canonical manner. We presented a case study of a system that is currently in daily use, introduced the variabilities artificially in order to have a "clean" test-bed.

The prototypical production environment showed that systematic support of binding time variability is possible.

In the future, we will study also other forms of meta-variability (e.g., the variability of constraints) and will aim to enhance our production environment along these lines. Furthermore, specific editors are planned to also support arbitrary artifacts.

We will further study possibilities of integrating our model-based generation with our forms of model-based code generation, in order to arrive at a seamless integration with target-language independent model-based development.

References

[1] Project homepage ANT, 2007. Online available at: http://ant.apache.org/.

[2] Project homepage SVNControl, 2007. Online available at: http://svncontrol.tirgirs.org/.

[3] M. Antkiewicz and K. Czarnecki. FeaturePlugin: Feature modeling plug-in for Eclipse. In *OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop*, 2004.

[4] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel. *Component-based Product Line Engineering with UML*. Addison-Wesley, 2002.

[5] P. Costanza, R. Hirschfeld, and W. De Meuter. Efficient layer activation for switching context-dependent behavior. In D. Lightfoot and C. Szyperski, editors, *JMLC 2006*, volume 4228 of *Lecture Notes in Computer Science*, Berlin / Heidelberg, 2006. SpringerVerlag Inc.

[6] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Techniques, and Applications*. Addison-Wesley, 1999.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 2000.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, 3. edition*. Addison-Wesley, 2005.

[9] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21 ESD-90-TR-222, Software Engineering Institute Carnegie Mellon University, 1990.

[10] Kyo C. Kang, Jaejoon Lee, and Patrick Donohoe. Feature-Oriented Product Line Engineering. *IEEE Software*, 19(4):58–65, 2002.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference*

on Object-Oriented Programming, volume 1241 of *Lecture Notes in Computer Science*. SpringerVerlag Inc., 1997.

[12] F. van der Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action - The Best Industrial Practice in Product Line Engineering*. Springer, 2007. http://www.spl-book.net/.

[13] M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 127–136, New York, NY 10036, USA, 2004. ACM Press.

[14] D. Muthig. *A Light-weight Approach Facilitating an Evolutionary Transition Towards Software Product Lines*. PhD thesis, Fraunhofer-Institut für Experimentelles Software Engineering IESE, Kaiserslautern, University of Kaiserslautern, Germany, 2002.

[15] D. Muthig and T. Patzke. Generic implementation of product line components. In *Proceedings of the Net.ObjectDays (NODE'02), Erfurt, Germany*, October 2002.

[16] I. Pashov, M. Riebisch, and I. Philippow. Supporting Architectural Restructuring by Analyzing Feature Models. In *Proceedings 8th European Conf. On Software Maintenance and Reengineering, Tampere, Finland, March 24-26, 2004*, pages 25–33, 2004.

[17] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. SpringerVerlag Inc., August 2005.

[18] K. Schmid, U. Becker-Kornstaedt, P. Knauber, and F. Bernauer. Introducing a software modeling concept in a medium-sized company. In *International Conference on Software Engineering (ICSE'22)*, New York, NY 10036, USA, 2000. ACM Press.

[19] K. Schmid and I. John. A Customizable Approach To Full-Life Cycle Variability Management. *Science of Computer Programming*, 53(3):259–284, 2004.

[20] Software Productivity Consortium Services Corporation, Technical Report SPC-92019-CMC. *Reuse-Driven Software Processes Guidebook, Version 02.00.03*, November 1993.