Domain-specific Adaptations of Product Line Variability Modeling

Deepak Dhungana, Paul Grünbacher and Rick Rabiser
Christian Doppler Laboratory for Automated Software Engineering
Johannes Kepler Universität, Linz, Austria
dhungana@ase.jku.at

Abstract. Despite its increasing popularity the widespread adoption of product line engineering is still hampered by a lack of flexible and extensible approaches that can be tailored to deal with diverse organizational specifics such as architectural styles, languages, or modeling notations. Many existing product line approaches focus on process aspects and provide general-purpose modeling approaches. In this paper we present a flexible and extensible variability modeling approach that can be adapted to domain-specific needs. The approach is supported by the meta-tool DecisionKing. The tool treats variability as a prime modeling concept and supports the domain-specific definition of dependencies between model elements. We demonstrate the feasibility of our approach with two case studies in the areas of industrial automation and service-oriented systems.

1 Introduction

Conventional single-system software engineering is often insufficient to meet the tight budget and schedule constraints faced by software industry. Companies therefore aim at understanding the relationships between similar products to exploit commonalities regarding marketing, technical, or end-user aspects. Software product line engineering (PLE) is based on creating and managing artifacts and processes such that they can be reused for building different yet related products. It has been shown that PLE can increase productivity, reliability, and quality of software development thereby also reducing cost and time-to-market [3, 4, 14, 17, 22]. This is achieved by modeling techniques for capturing the variability of reusable core assets such as requirements, architecture, code, processes, documents, or models.

Please use the following format when citing this chapter:

Dhungana, D., Grünbacher, P., Rabiser, R., 2007, in IFIP International Federation for Information Processing, Volume 244, Situational Method Engineering: Fundamentals and Experiences, eds. Ralyté, J., Brinkkemper, S., Henderson-Sellers B., (Boston Springer), pp. 238-251.

While there is a strong consensus on the benefits of PLE, it remains challenging for organizations to identify methods and techniques applicable for their particular context, to adapt these methods and techniques to address the specific needs of their domain, and to integrate them with their current practices, tools, and standards [16]. A reason for these problems lies in the inflexibility of existing product line modeling approaches and tools which often do not support the diverse needs of different organizations. A key goal of our research is thus to make our methods and tools as flexible as possible.

Variability modeling is central in PLE to capture commonalities and variability of a product line's core assets. Variability has to be understood and modeled at different levels (e.g., requirements, architecture, or implementation level) and for diverse domain-specific artifacts [7]. The traceability between variation points, i.e., decision points describing possible choices about assets' functions or qualities, and the management of variability mechanisms implementing these points are important aspects. The need for a flexible variability modeling approach becomes evident when considering the heterogeneous languages, modeling notations, or architectural styles used by different organizations. There are two important problems faced by both research and industry [7]: (1) there is a lack of integrated variability modeling approaches that work well with arbitrary and heterogeneous types of assets in the product line; (2) there is a lack of flexible and extensible tools that can be tailored to support a particular organization's needs.

In our ongoing research collaboration with Siemens VAI we are developing an approach addressing these issues. DOPLER (**D**ecision-**O**riented **P**roduct **L**ine **E**ngineering for effective **R**euse) is an approach that works with heterogeneous domain-specific artifacts while being independent of specific architectural styles, languages, or modeling notations. The approach is supported by the meta-tool DecisionKing [7] supporting the identification, design, implementation, and maintenance of a product line's assets. Unlike existing general purpose meta-tools [11, 21, 26] DecisionKing provides support for variability as a first class modeling concept. Furthermore, it adopts a rule engine to master the complexity of dependencies in the models. Organizations can also incorporate company-specific capabilities by exploiting the tool's plug-in architecture.

This paper is organized as follows: We describe our variability modeling approach and show how it allows domain-specific adaptations. We present the meta-tool DecisionKing [7] and discuss method engineering concepts used in our approach. Two case studies illustrate the benefits and feasibility of our approach in two significantly different domains: (i) Together with Siemens VAI, the world's leader in building plants for the iron, steel, and aluminum industries, we are using DOPLER to model the variability of their automation software for continuous casting in steel plants; (ii) In an ongoing research project [5, 10] we are modeling service variability by complementing the *i** modeling language [25] with variability modeling. We conclude the paper with a discussion of related work and an outlook on future work.

2 Product Line Variability Modeling

Leveraging reuse in PLE relies on documenting tacit knowledge about variability and making it explicit and manageable in models [4]. Variability models cover the product line's problem space (stakeholder needs and desired features) and its solution space (architecture and components of the technical solution). Variability models define a product line's assets with organization- and domain-specific properties and dependencies. They capture different variants of features and solution components and their valid combinations, i.e., the possible variants together with constraints and dependencies. Variability models also document fundamental system-wide decisions for the configuration and derivation of a product [8] and the rationale for these decisions.

DOPLER can deal with diverse product line assets and allows arbitrary dependency links between the assets. It relates the assets with decisions for product derivation and customization. The approach is based on a generic variability meta-model (Fig. 2) which has to be extended and adapted to organizational needs. The meta-model does not encompass every modeling element that may be relevant in certain organizations. It defines just the basic concepts to be modeled on a higher level of abstraction. Unlike a general-purpose meta-model, our approach treats variability as a prime concept by modeling decisions. Fig. 1 depicts the DOPLER modeling process encompassing domain modeling (the adaptation of the meta-model), asset modeling (the definition of the PL's assets based on the meta-model), and decision modeling (the definition of variability):

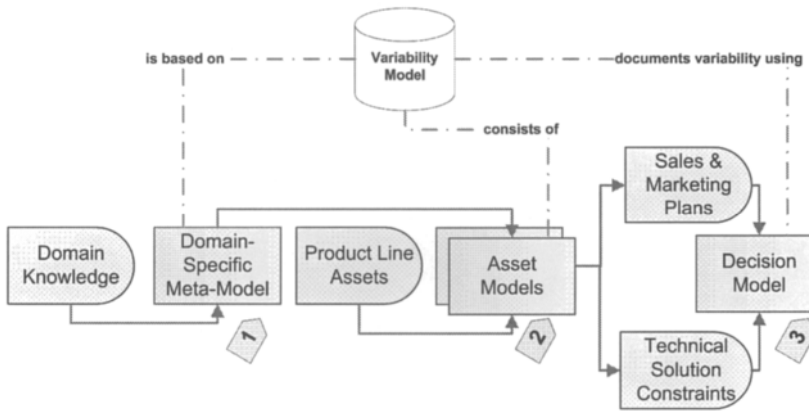

Fig. 1 DOPLER variability modeling approach [7].

(1) Domain Modeling. Managing different kinds of assets in a PL relies on the precise definition of their specific characteristics in a domain-specific meta-model. Building such a model requires knowledge about the domain and the organization's settings and specifics. The meta-model defines the types of assets to be included in the product line (e.g., Components, Services, Documents, Properties, etc.) and the possible relationships between the different asset types.

(2) Asset Modeling. An asset model is created on the basis of a domain-specific meta-model and describes the concrete reusable elements in a product line and dependencies among them. Asset models can often be created semi-automatically if product line development does not start from scratch and core assets already exist. For example, call dependencies defined in existing system configuration files can be utilized to automatically derive *requires* dependencies among software components that reflect the underlying technical restrictions (cf. Section 5.1). Modeling these dependencies is essential for later product derivation.

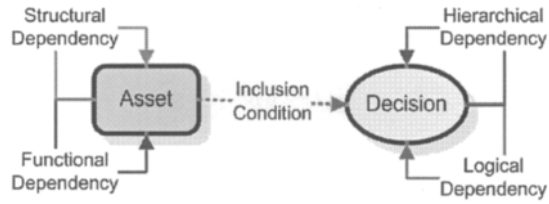

Fig. 2 Core meta-model for variability [8].

(3) Decision Modeling: Variability stemming from technical or marketing considerations is expressed using decisions to be taken when deriving products from the product line [19]. Decision models link external variability (visible to customers, sales people, or marketing staff) with internal variability (visible to engineers). A decision model is a graph where the nodes represent decisions and the edges represent relationships between them. Decisions are variables which can have special dependencies to other variables. These dependencies are expressed using a rule language. Decisions are presented to decision-takers in the form of questions. Validity conditions restrict the range of possible values. In order to link assets and decisions, assets specify an *inclusion condition* which has to be satisfied for a particular asset to be included in the final product. This expression can be composed of arbitrary decisions. Decisions and inclusion conditions also establish trace links between user demands and assets [8]. Decision models reduce modeling complexity as they represent variability at a higher level of abstraction. For instance, variability mechanism in the asset base can be changed without having to change the variation points of the system. Experience also shows that fewer decisions are necessary to reach the desired variability than adding variability specifications to all assets [8]. The core meta-model (Fig. 2) currently supports *hierarchical dependencies* specifying how the decisions are organized and *logical dependencies* specifying the known consequences of taking decisions:

Hierarchical dependencies are Boolean expressions that specify when a particular decision is visible to the user. For example, the user needs to decide if an archiving feature is required before taking more specific decisions on the type of database used for archiving. Considering the example in Fig. 3, this kind of relationship is modeled between the decision DeburrerPredecessor and Deburrer. The decision DeburrerPredecessor is visible to the user only if the value of decision Deburrer is true.

Logical dependencies specify actions that need to be executed after a decision has been taken. Typically, these are business rules that need to be checked (before and) after a decision is taken. In the example presented in Fig. 3, we can see such a

relationship between DeburrerPredecessor and MarkingPredecessor. If the user enters INPUT as the value for DeburrerPredecessor the value of the variable MarkingPredecessor is also set to INPUT. After a decision is taken, its effects are propagated automatically to all the other affected decisions in the model. This is important to guarantee the consistency of selected options and taken decisions during product derivation.

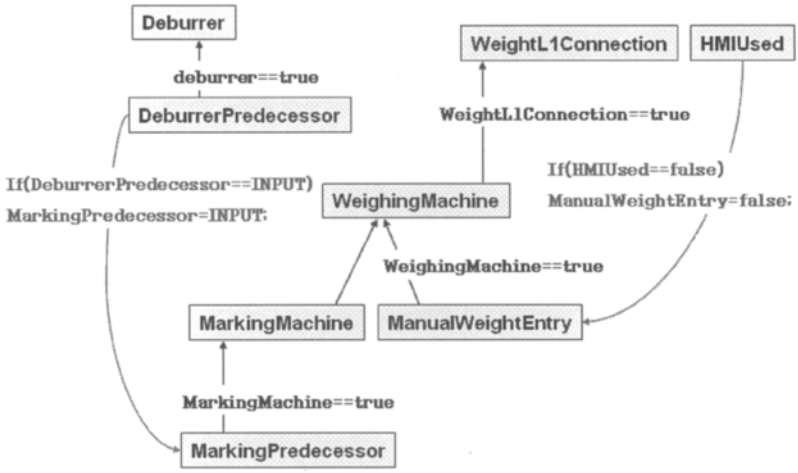

Fig. 3 Example of a Decision Model based on an existing variability model of the Siemens VAI subsystem Runout. Decision variables (nodes) are modeled with their hierarchical and logical dependencies (edges) thereby forming a graph.

3 Adopting Method Engineering Concepts

Method Engineering offers important concepts for achieving a higher level of flexibility: (i) Meta-models have proven to be useful to identify and describe the concepts of a generic method, (ii) Generic methods can be adapted to the actual situation of a project using concepts of Situational Method Engineering (SME) [15], and (iii) Meta-tools provide a automated support for such adaptations. Our approach is based on these concepts: we provide a generic meta-model, which has to be adapted to domain-specific needs. We also offer tool support through adaptations of our meta-tool DecisionKing.

Meta-model adaptation and evolution. Every domain has its own concepts, dependencies, and rules. These characteristics are defined by a meta-model specifying the attributes, dependencies, syntax, and semantics of these concepts. A meta-model defines the "language" in which domain models can be expressed and from which tools for writing domain models can be generated. While the meta-model is specified by method experts, the models are developed by domain experts using the generated domain-modeling tools. For example, in our approach the core meta-model (Fig. 2) is refined using new asset types together with attributes and relationships among them to support domain-specific concepts. The behavior of

model elements is defined by semantic classes, i.e., model element interpreters and dependency resolvers for relationships between the assets.

Meta-models can change just like other models. Variability modeling tools and techniques must be adaptable to provide an effective model-driven development cycle. We allow domain evolution via updates to the meta-model [20] thereby also adapting the variability modeling tool. This allows us to react to changing requirements of the problem domain. For instance, the introduction of new asset types as well as the modification of existing assets requires techniques for schema evolution of already existing models, automatic adaptation of tools, and methods for checking the semantic consistency of the evolved models. The evolution of the meta-model is of particular interest when introducing a new product line. In order to master the complexity, one can begin with a relatively simple meta-model which is extended as the product line evolves.

Meta-tools and tool extensions. Meta-tools are needed to benefit from the flexibility offered by meta-modeling and meta-model evolution. Such meta-tools allow the generation of specific tools for a target environment. Recent developments in the area of software tools such as the Eclipse platform allow the development of extensible meta-tools that can be augmented with domain-specific capabilities. For instance, the plug-in approach supports a compact core that can be extended with plug-in components tailored to the users' needs to improve focus and reduce clutter by providing a customized user environment [24]. In DOPLER we used a plug-in approach to incorporate a domain-specific rule language, an off-the-shelf rule engine, a model visualization system, and domain-specific tools for semi-automatically creating initial decision models from existing assets.

4 DecisionKing: A Meta-Tool for Variability Modeling

DecisionKing can be configured to support domain-specific variability modeling with domain meta-models specifying relevant characteristics of the application domain. DecisionKing distinguishes itself from more general-purpose meta-tools like MetaEdit+ [21] or Pounamu [11, 26] by treating variability as a primary modeling concept. Also, the dependencies among model elements are not just plain trace links as they are interpreted using a rule engine. The plug-in-based architecture of the tool makes it flexible and extensible to domain-specific adaptations (cf. Fig. 4). The result of adapting DecisionKing for a particular organization is a domain-specific variability model editor for domain-specific assets. Implementing tool-extensions allows a tight integration of this editor with current practices, standards, and existing tools of the organization. Fig. 4 shows an overview of the DecisionKing's capabilities for domain-specific adaptations:

Meta-model editor. An editor allows the creation of domain-specific meta-models by specifying domain-specific asset types (e.g., components, services, data, code, settings, documents, component descriptions), their attributes (e.g., description, URL, cost), and dependencies (e.g., component *requires* component). Domain-specific behavior can be added to model elements and relationships by providing model element interpreters and dependency resolvers as domain-specific plug-ins.

The meta-model adaptation framework (cf. Fig. 4) adjusts the variability modeling editor according to the domain-specific meta-model.

Domain-specific tool extensions and plug-ins. The DecisionKing customization framework supports two types of extensions:

(i) We provide extension points for adapting the functionality of the tool. Default implementations of these capabilities can easily be replaced with domain-specific plug-ins without having to touching the tool's implementation. We have created default plug-ins of a rule language, a constraint editor, a rule engine, and a model visualizer. For example, one can provide a model viewer with domain-specific graphical layouts and symbols. Another example is the rule specification language needed to model dependencies among decisions. The language used for this purpose and choice of technology depends highly on the domain and current practices of the organization. We have experimented with different domain-specific languages for rule specification, using JBOSS[1] Rules as the rule engine. We have also tried JESS[2], where we modeled our decisions as facts of an expert shell.

(ii) A generic extension point is provided in the form of a model API which allows arbitrary tools to manipulate, use, or create models. This API has for instance been useful to develop model importers, which analyze the existing asset base to semi-automatically create asset models. The integration of existing domain-specific tools is another important aspect.

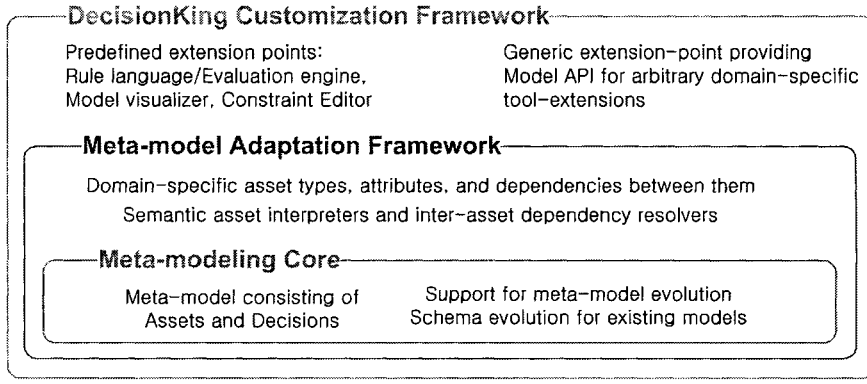

Fig. 4 Overview of DecisionKing's adaptation mechanisms.

5 Case Studies

To demonstrate the feasibility of our approach, we present two case studies from two different contexts. The goal of the case studies was to validate the generic meta-model and to gain experience with method engineering concepts (cf. Section 3) in practical settings. The case studies were also instrumental to demonstrate the usefulness and usability of our tools in different contexts. We describe the meta-

[1] http://www.jboss.com/products/rules

[2] http://herzberg.ca.sandia.gov/jess/

model adaptations and domain-specific extensions of our tools developed for the case study contexts, as well as key experiences gained.

5.1 Case study 1: Industrial automation

Siemens VAI[3] is the world's leading engineering and plant-building company for the iron, steel, and aluminum industries. In an ongoing research project, we are modeling the variability of their software product line for process automation, optimization, supervision, and material tracking of continuous casting in steel plants.

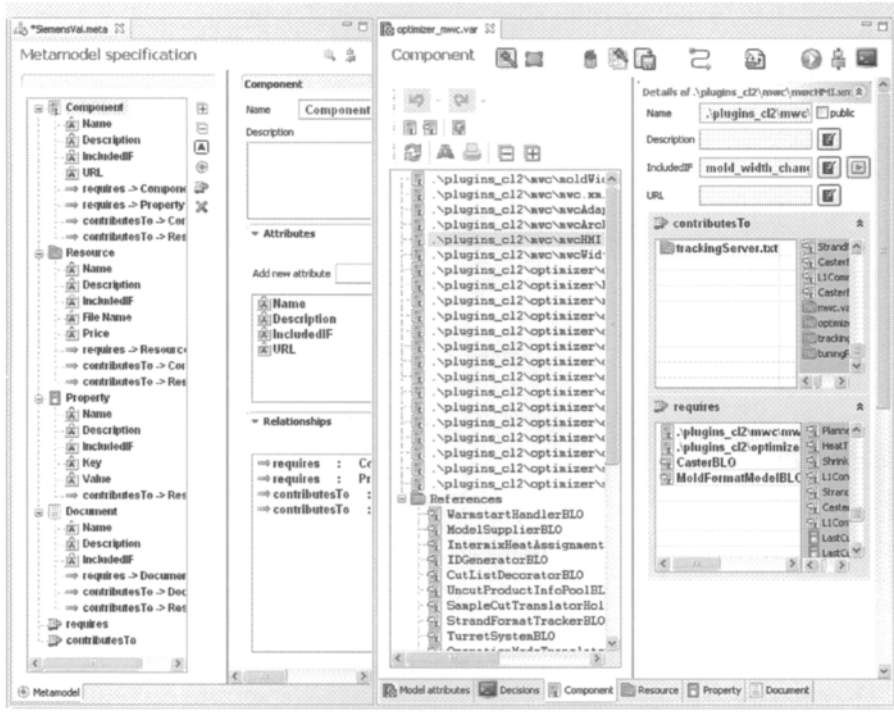

Fig. 5 DecisionKing's Meta-Model Editor (left) and Variability Model Editor (right). The variability model on the right is based on the meta-model on the left.

Meta-model adaptation. In various workshops conducted with the engineers and sales experts of Siemens VAI, we identified the types of core assets to be reused in the product line: *Components* (specified using Spring[4] XML files), *Properties* (configuration parameters for components), *Resources* (legacy hard- or software elements, configuration files, etc), and *Documents* (e.g., descriptions of components, notes, fragments of end user documentation, etc). We also identified the functional dependency *requires* between assets. E.g., a software component may rely on another component to function properly (similar modeling capabilities are available

[3] http://www.industry.siemens.com/metals/en/

[4] http://www.springframework.org/

in architecture description languages such as xADL [6]). A domain-specific resolver for the relationship *requires* adds all components required by a certain component as soon as the parent is added to the final system (i.e., by taking a decision during product derivation). Information about the deployment structure of the system is modeled using the relationship *contributesTo (e.g.,* a component contributes to the sub-system it belongs to).

Domain-specific extensions and plug-ins. We developed a tree-based graphical viewer for Siemens VAI variability models based on GEF viewers[5] which is seamlessly integrated in the modeling environment of DecisionKing. In order to represent the relationships between the decisions needed to derive a product, we have implemented a default rule language with Java-like syntax that includes a simple interpreter as part of the rule engine. As already mentioned, Siemens VAI's software components are described using Spring XML. To expedite the modeling process and to ensure consistency of the models with the technical solution we developed a model importer extension capable of analyzing existing component descriptions and creating an initial asset model based on these descriptions. This model importer extension is also capable of suggesting decisions if two Spring XML describe two different implementations of the same interface. The user can decide whether to contribute the decision to the decision model.

Experiences. Despite its simplicity, the meta modeling core provided a good match to describe the variability for the different asset types. A key to accelerate the modeling process are automatic importers. Support for domain evolution turned out to be essential because the characteristics of the problem domain needed to stabilize in the initial stages of product line adoption. We were able to adapt our modeling paradigm to these often-changing requirements. The concepts of domain evolution are important for organizations introducing product lines. It allows them to start with a simple domain-model and adapting it over time as new modeling aspects are needed (cf. Section 4).

5.2 Case study 2: Multi-Stakeholder distributed Systems

Multi-stakeholder distributed systems (MSDS) are distributed systems in which subsets of the nodes are designed, owned, or operated by distinct stakeholders [12]. MSDS are quickly gaining importance in today's networked world as, e.g., shown in the field of service-oriented computing. We have been using the *i** language [25] to model a service-oriented multi-stakeholder distributed system in the travel domain to validate the usefulness of *i** for that purpose. A major goal of the project was to enhance *i** with capabilities for variability modeling in the context of our MSDS framework [5].

Meta-model adaptation. We identified four asset types in our framework relevant to variability modeling: goals, service types, services, and service instances. The element *Goal* in DecisionKing's meta-model maps to the element "actor goal" in *i**. Different *Service types* contribute to fulfilling these goals. Available services realizing a service type are modeled as a *Service*. Finally, available runtime implementations of services can be modeled as *Service instances*. We also identified

[5] http://www.eclipse.org/gef/

two kinds of relationships between the assets: A *requires* relationship is used whenever the selection of a service leads to the selection of another service. This can be the result of logical dependencies between goals, conceptual relationships between service types, relationships between services, or functional dependencies between service instances. The *contributesTo* relationship is used to capture structural dependencies between assets of different levels. Service instances for example contribute to services. Services contribute to service types which themselves contribute to goals. It is however also possible that a goal is split up into sub-goals. Such compositional relationships between goals can also be modeled using the *contributesTo* relationship.

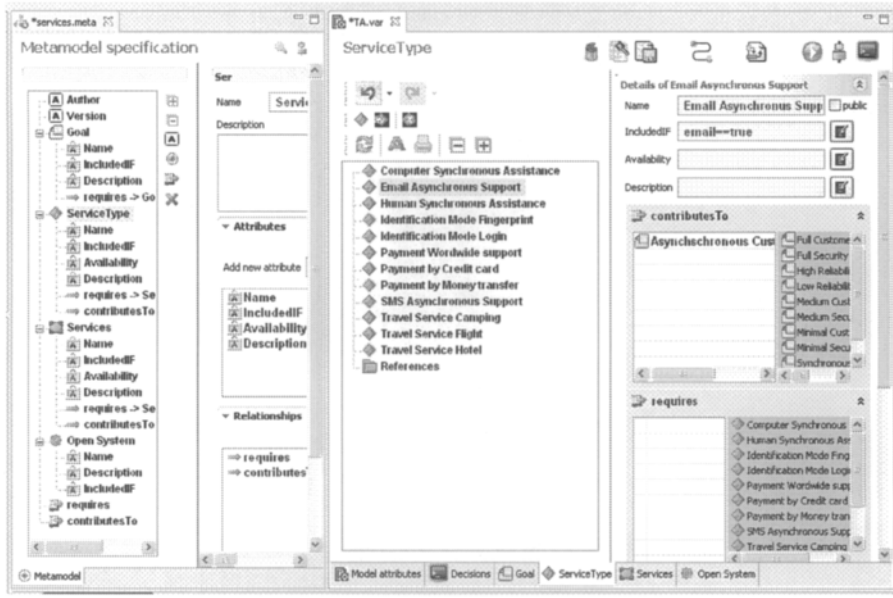

Fig. 6 DecisionKing's Meta-Model Editor (left) and Variability Model Editor (right). The variability model on the right is based on the meta-model for service-oriented systems on the left.

Domain-specific extensions and plug-ins. The dependencies among decisions were expressed using a domain-specific language; the rules were transformed to JBOSS rules using a rule-converter. We use the JBOSS rule engine to evaluate the dependencies among decisions and the inclusion conditions between assets and decisions (cf. Section 2). We have not yet implemented a specific visualization for service-oriented variability models. The model can however be visualized using the default model viewer. We will develop a connector to tools for the *i** modeling approach, e.g., the REDEPEND tool [9] that is capable of storing *i** models in XML.

Experiences. The use of DecisionKing in the project confirmed the need for a general-purpose model API that allows arbitrary external tools to update and query the variability model. This capability will allow us to use DecisionKing as one component in our framework for service monitoring and adaptation. We are planning to utilize variability models to support the controlled runtime adaptation of service-

oriented systems, e.g., by replacing a malfunctioning service with a similar service specified in the variability model.

6 Related Work

We focus the discussion of related work on variability modeling approaches and tools, meta-tools, and plug-in frameworks.

Variability modeling approaches and tools. Many variability modeling approaches have been proposed. Our work was strongly influenced by the work of John and Schmid [19] who presented an approach for orthogonal variability modeling and management across different stages of the software development life-cycle. Similar to their approach we also use decision models for describing the variation of products in a product line. Bachmann *et al.* [1] have described an approach for representing variability in a uniform way separated from the representation of concrete assets. Their view on variability is similar to our approach. Berg *et al.* [2] emphasize on the importance of mapping variability between the problem and solution space, an aspect we also address with our approach. Numerous commercial and research tools for variability modeling and management have been developed, for example: *Pure::variants* [18] by pure-systems GmbH is a variant and variability management tool for managing software product lines based on feature models and family models. Feature models describe the variability whereas asset modeling is supported by family models describing the software in terms of architectural elements. The family model is extensible; however no specialization hierarchy for the model elements is supported. No explicit support is provided to model domain-specific asset types such as hardware resources, data models, development process guidance, libraries, etc. *Gears* [13] by Big Lever Software Inc. is a development environment for maintaining product family artifacts and variability models. Variability is handled at the level of files and captured in terms of features, product family artifacts, and defined products that can be derived from the variability model. The tool supports the identification of common and variable source code files. Our approach differs form this because we treat all assets as model elements and don't deal with them at file level.

Meta-Tools. Meta-tools can be seen as generators for domain-specific tools. Examples for Meta-tools are MetaEdit+ [21] and Pounamu [11, 26]. MetaEdit+ [21] is a tool for designing a modeling language, its concepts, rules, notations, and generators. The language definition is stored as a meta-model in the MetaEdit+ repository. MetaEdit+ follows the given modeling language definition and automatically provides full modeling tool functionality like diagramming editors, browsers, generators, or multi-user support. Pounamu [26] is a meta-tool for the specification and generation of multiple-view visual tools. The tool permits rapid specification of visual notational elements, the tool information model, visual editors, the relationships between notational and model elements, and behavior. Tools are generated on the fly and can be used for modeling immediately. Changes to the meta-tool specification are immediately reflected in tool instances. Typically meta-tools provide support for their target domain environments but are restricted in

their flexibility and integration capabilities with other tools [23]. They do not treat variability as a prime modeling concept, which hampers their use for product line modeling.

Plug-in frameworks. Plug-in concepts are widely used in modern development platforms. DecisionKing is an Eclipse[6] Rich Client Application based on the Eclipse plug-in platform [24]. It uses the platform's plug-in mechanisms to define extension points allowing the integration of different domain-specific plug-ins.

7 Conclusions and Further Work

In this paper we described the DOPLER approach which adopts method engineering concepts supporting the creation of domain-specific variability modeling tools. We presented DecisionKing, a meta-tool that can easily be tailored to a particular organization's needs by refining its core meta-model and exploiting its plug-in architecture. DOPLER provides tools for the creation and management of the models. The approach does not assume any particular approach to software product line engineering beyond the basic tenets implied by the definition of a software product line. We showed the adaptability of the approach using two case studies in different domains. It is noteworthy mentioning that an automated approach is only as good as the model underlying the approach. Meta-model evolution capabilities allow us to start with a small language first that can be extended in the project after the team has gained some experience and confidence.

We are currently working on the following issues and will report about them in the future:

Use of variability models to support runtime adaptation of systems. We are currently adapting DecisionKing to the domain of ERP systems. We are developing plug-ins allowing to adapt an ERP system at runtime based on variability models.

Validation of the model evolution capability. Our model evolution framework is a great help in coping with changing architectures and implementations of a product line under development. We are currently refining and evolving the variability models for Siemens VAI to further validate our capabilities for model evolution and meta-model evolution.

Improvement of generic visualization support. We intend to make the current model visualization more generic. The graphical representation of a model has to be changed for different domains because of domain-specific symbols and layouts. This enables the use of symbols and layouts which stakeholders of the domain already know and understand. In particular, we are interested in using graphical ways to specify variability to overcome shortcomings of a purely text-based approach.

[6] http://eclipse.org

Acknowledgements

This work has been conducted in cooperation with Siemens VAI and has been supported by the Christian Doppler Forschungsgesellschaft, Austria. We would like to express our sincere gratitude to Klaus Lehner, Christian Federspiel, and Wolfgang Oberaigner from Siemens VAI for their support and the valuable insights.

References

1. F. Bachmann, M. Goedicke, J. Leite, R. Nord, K. Pohl, B. Ramesh, and A. Vilbig, "A Meta-model for Representing Variability in Product Family Development," in *Lecture Notes in Computer Science: Software Product-Family Engineering*. Siena, Italy: Springer Berlin / Heidelberg, 2003, pp. 66-80.
2. K. Berg, J. Bishop, and D. Muthig, "Tracing Software Product Line Variability – From Problem to Solution Space," presented at 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries, White River, South Africa, 2005.
3. G. Böckle, P. Clements, J. D. McGregor, D. Muthig, and K. Schmid, "Calculating ROI for Software Product Lines," *IEEE Software*, vol. 21, pp. 23-31, 2004.
4. P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*: SEI Series in Software Engineering, Addison-Wesley, 2001.
5. R. Clotet, F. Xavier, P. Grünbacher, L. López, J. Marco, M. Quintus, and N. Seyff, "Requirements Modelling for Multi-Stakeholder Distributed Systems: Challenges and Techniques. ," presented at RCIS'07: 1st IEEE Int. Conf. on Research Challenges in Information Science, Quarzazate, 2007.
6. E. M. Dashofy and A. van der Hoek, "Representing Product Family Architectures in an Extensible Architecture Description Language," presented at 4th International Workshop on Software Product-Family Engineering, Bilbao, Spain, 2001.
7. D. Dhungana, P. Gruenbacher, and R. Rabiser, "DecisionKing: A Flexible and Extensible Tool for Integrated Variability Modeling," in *First International Workshop on Variability Modelling of Software-intensive Systems - Proceedings*, K. Pohl, P. Heymans, K.-C. Kang, and A. Metzger, Eds. Limerick, Ireland: Lero - Technical Report 2007-01, 2007, pp. 119-128.
8. D. Dhungana, R. Rabiser, and P. Grünbacher, "Decision-Oriented Modeling of Product Line Architectures," presented at Sixth Working IEEE/IFIP Conference on Software Architecture, Mumbai, India, 2007.
9. G. Grau, X. Franch, N. A. M. Maiden, and " REDEPEND-REACT: an architecture analysis tool," presented at 13th IEEE International Conference on Requirements Engineering, 2005. Proceedings.
10. P. Grünbacher, D. Dhungana, N. Seyff, M. Quintus, R. Clotet, F. Xavier, L. López, and J. Marco, "Goal and Variability Modeling for Service-oriented System: Integrating i* with Decision Models," presented at Software and Services Variability Management Workshop: Concepts, Models, and Tools, Helsinki, 2007.
11. J. Grundy, J. Hosking, N. Zhu, and N. Liu, "Generating Domain-Specific Visual Language Editors from High-level Tool Specifications " presented at 21st IEEE International Conference on Automated Software Engineering (ASE'06), Tokyo, Japan, 2006.

12. R. J. Hall, "Open modeling in multi-stakeholder distributed systems: requirements engineering for the 21st Century," presented at First Workshop on the State of the Art in Automated Software Engineering, Irvine, California, 2002.
13. C. W. Krueger, "Software Mass Customization," BigLever Software, Inc 2005.
14. C. W. Krueger, "New Methods in Software Product Line Development," presented at 10th International Software Product Line Conference, Baltimore, USA, 2006.
15. K. Kumar and R. J. Welke, "Method Engineering: a proposal for situation-specific methodology construction " in *Systems Analysis and Design : A Research Agenda*: John Wiley & Sons, Inc., 1992 pp. pp257-268.
16. D. Muthig, I. John, M. Anastasopoulos, T. Forster, J. Dörr, and K. Schmid, "GoPhone - A Software Product Line in the Mobile Phone Domain," *IESE-Report No. 025.04/E*, 2004.
17. L. Northrop, "SEI's Software Product Line Tenets," *IEEE Software*, vol. 19, pp. 32-40, 2002.
18. pure-systemsGmbH, "Technical White Paper, Variant Management with pure::variants,," 2004.
19. K. Schmid and I. John, "A Customizable Approach to Full-Life Cycle Variability Management," *Journal of the Science of Computer Programming, Special Issue on Variability Management*, vol. 53, pp. 259-284, 2004.
20. D. C. Schmidt, A. Nechypurenko, and E. Wuchner, "MDD for Software Product-lines: Fact or Fiction?," presented at 8th international Conference on Model driven Engineering Languages and Systems (MODELS '05), Jamaica, 2005.
21. J.-P. Tolvanen and M. Rossi, "MetaEdit+: defining and using domain-specific modeling languages and code generators," presented at Conference on Object Oriented Programming Systems Languages and Applications, Anaheim, CA, USA, 2003.
22. F. van der Linden, "Software Product Families in Europe: The Esaps & Cafe Projects," *IEEE Software*, vol. 19, pp. 41-49, 2002.
23. A. I. Wasserman, "Tool integration in software engineering environments," presented at Proceedings of the international workshop on environments on Software engineering environments Chinon, France, 1990
24. R. Wolfinger, D. Dhungana, H. Prähofer, and H. Mössenböck, " A Component Plug-in Architecture for the .NET Platform," presented at Proceedings of 7th Joint Modular Languages Conference, (JMLC'06), Oxford, UK, 2006.
25. E. S.-K. Yu., "Modeling Strategic Relationships for Process Reengineering," vol. PhD Thesis. Toronto: University of Toronto 1996.
26. N. Zhu, J. Grundy, and J. Hosking, " Pounamu: A Meta-Tool for Multi-View Visual Language Environment Construction," presented at 2004 IEEE Symposium on Visual Languages and Human Centric Computing, 2004.