Three-level Customization of Software Products Using a Product Line Approach

Abstract

Many office and enterprise business applications are overloaded with features. As a result users struggle in finding the functionality needed to support their tasks. Customization support for existing applications is typically limited and often only accessible to technical experts. Software product line approaches provide support for customizing complex applications but typically focus on supporting software producers in deriving customized products from reusable components instead of supporting end-users. We present a decision-oriented software product line approach that supports customization at three levels: suppliers deriving products for customers, customers configuring products to the needs of specific user groups, and end-users customizing a system to their personal needs. We describe tool support and illustrate the approach with a case study from the domain of enterprise resource planning.

1. Introduction and Motivation

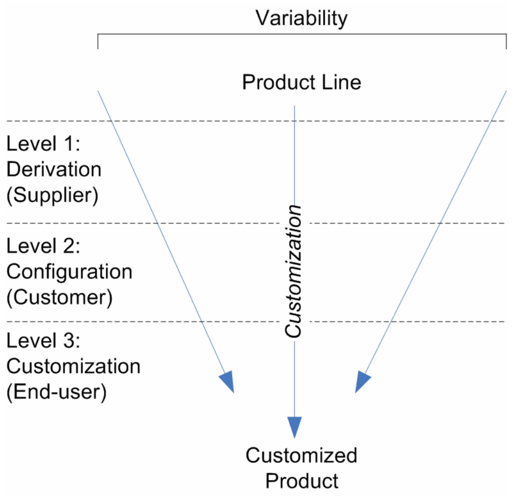
End-users of today's office and enterprise business applications often struggle to understand the offered functionality. Typically, they only need a small fraction of the features and are overwhelmed by the many unneeded capabilities. Unfortunately, applications provide only very limited capabilities for customization to specific user needs (e.g., via user preferences). The customization of applications remains a challenging task relevant at different levels and involving software producers, buyers, and end-users.

It has been demonstrated that software product lines provide a reasonable approach to customization of complex software systems [16, 25]. A software product line has been defined as "a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a pre-scribed way" [4]. Software product line engineering (SPLE) covers processes for building, managing, and using software product lines. It covers the areas of domain engineering and application engineering [16]: During domain engineering, the variability of the product line's solution components is explicitly captured in variability models. During application engineering, customized products are derived from the product line by using these models and deploying the needed solution components. The fundamental idea of SPLE is that the efforts of defining the product line in domain engineering should be outweighed by the benefits of being able to quickly derive customized products from the common core of shared, reusable assets during application engineering [7]. So far, product lines have mainly been used by software producers to derive and deploy customized products for different customers. In this paper, we demonstrate that product lines can be adopted to equally provide customization support for end-users. If software suppliers succeed in deriving customized products for their customers, end-users should similarly benefit from product line capabilities.

In our collaboration with several industrial and academic partners from the domains of industrial automation [20], enterprise resource planning [27], and service-oriented systems [5] end-users have requested easy-to-use capabilities allowing them to customize an application to their personal needs. We believe that software product line approaches can facilitate such end-user customization. In a product line approach variability models describe the possible choices for customization. Making these models accessible by end-users requires role- and task-specific views on the offered variability. Furthermore, from a technical perspective, run-time adaptation mechanisms are required as end-users should not have to worry about installation and system restarts after customization [27].

Figure 1 depicts the three levels of product customization we address with our approach together with the involved stakeholders. At each customization level, more variability is resolved as shown by the funnel:

Level 1 – Product derivation by suppliers. Suppliers resolve the variability captured in product line variability models to derive a product based on customers' requirements. We have described our support for this conventional product derivation in earlier work, e.g., [18, 27].



Variability
Product Line
Level 1: Derivation (Supplier)
Level 2: Configuration (Customer)
Level 3: Customization (End-user)
Customization
Customized Product

Figure 1. Levels of product customization: The three levels provide different views on the same product line variability model.

Level 2 – Product configuration by customers. Customers can further customize the product to organizational specifics, for example, to accommodate different tasks and roles in different departments. In many cases such as in enterprise resource planning (ERP) customers are not the end-users but rather accompany the introduction of the product for other customers.

Level 3 – Product customization by end-users. The end-users of the product further customize the product to their specific needs and wishes again using the variability models [12]. They personalize the application (ideally at run-time) to their tasks and responsibilities.

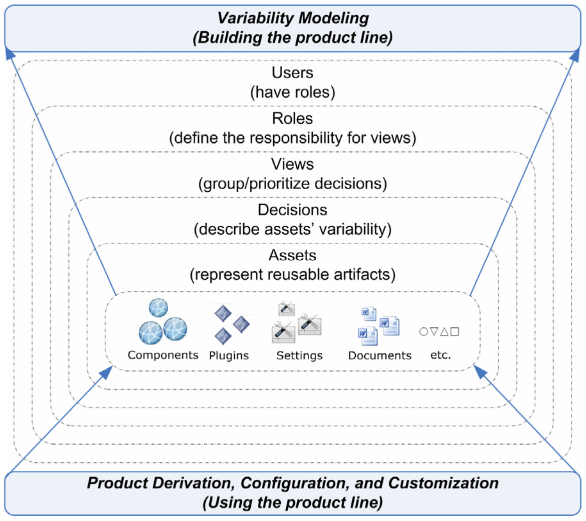
The remainder of this paper is structured as follows. Section 2 presents our approach and tool support for the mentioned levels of customization. In Section 3, we present a case study of applying our approach and tools in the enterprise resource planning domain for our industrial partner BMD. In Section 4 we discuss related work. Section 5 rounds out the paper with conclusions and an outlook on future work.

2. Approach

Our three-level product customization approach is based on DOPLER [8], a decision-oriented approach to software product line engineering we have developed influenced by work of Schmid and John [21] and the results of the Synthesis project [24]. DOPLER provides integrated capabilities for domain engineering to define a product line and its variability in models [9] as well as application engineering to use the product line models for deriving and customizing concrete products [18].

2.1 Product line variability model

The key elements of our model are shown in Figure 2:



Variability Modeling
(Building the product line)
Users
(have roles)
Roles
(define the responsibility for views)
Views
(group/prioritize decisions)
Decisions
(describe assets' variability)
Assets
(represent reusable artifacts)
Components
Plugins
Settings
Documents
etc.
Product Derivation, Configuration, and Customization
(Using the product line)

Figure 2. A product line variability model defines the reusable assets and decisions specifying their variability. Users are provided with role-specific views on the decisions to derive and customize products.

Assets represent all reusable elements of a product line, for example, architectural elements, software components, documentation, test cases, requirements, etc. Assets can depend on each other, for example, a component can require another component to function properly. In our approach, we first define the assets to be modeled and their possible dependencies in a meta-model specific for a particular domain. Our approach allows defining arbitrary asset types, attributes, and dependencies to allow its use in different domains [9].

Decisions represent the variation points in a product line variability model. They allow documenting and planning variability in domain engineering, guiding users during derivation, and automating product configuration. Decisions have a name and are represented by questions for different users of the variability model. Questions can be asked to address variability in both problem space and solution space. For instance, a supplier might have to answer the questions “Shall the Archive plug-in be delivered?” and “Shall the Scanning plug-in also be included?” These questions are phrased in solution space language and require deep technical understanding. A single deci-

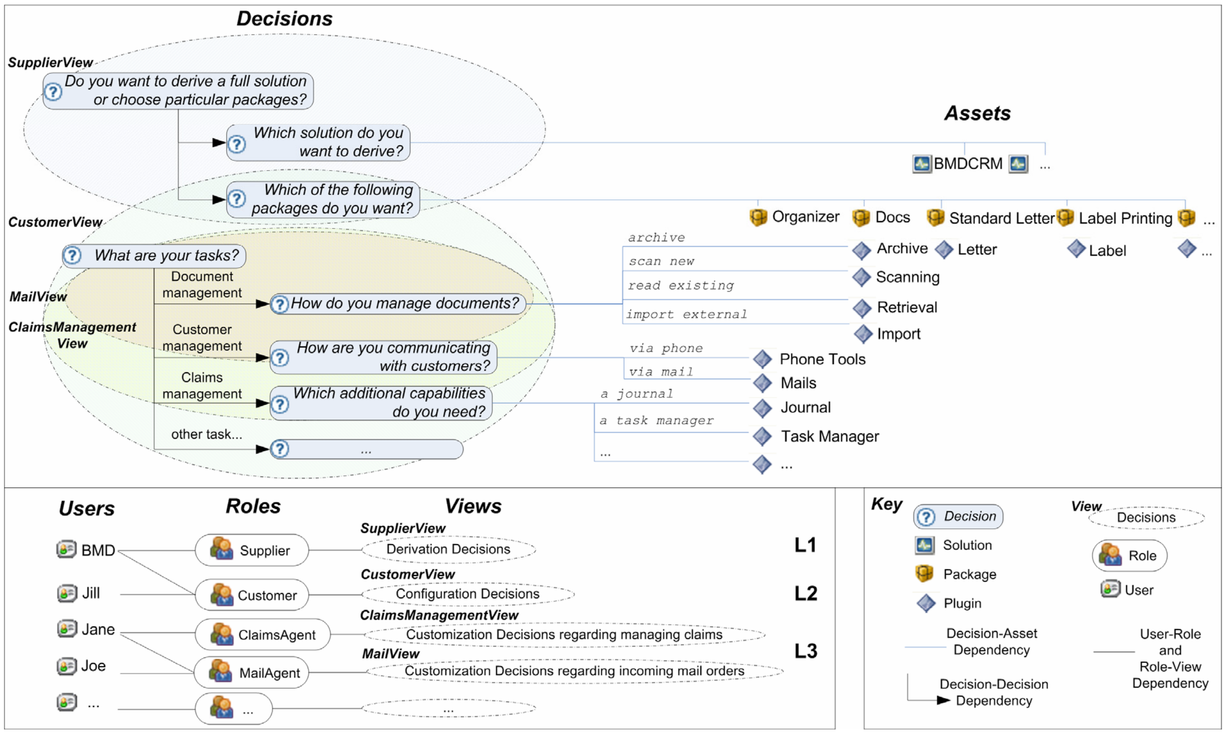


Figure 3. Partial BMD variability model (L1, L2, L3 denote the levels of customization).

sion documentManagement represented by the question "How do you manage documents?" with the possible answers "archive" and/or "scan new" can instead be provided in problem space language to make the question more easily understandable by end-users.

Decisions can depend on each other hierarchically (e.g., a decision needs to be taken before another one) and/or logically (e.g., taking a decision changes the value of another one). This way, by modeling dependencies, processing sequences for taking the decisions can be defined. Decisions are related with assets by explicitly modeling inclusion conditions. Such conditions define for each asset when it will be part of a product. The inclusion conditions thus also establish trace links between user decisions and the core assets. In our example (cf. Figure 3), the inclusion condition of the Archive plug-in is contains(documentManagement, {"archive"}) while for the Scanning plug-in it is contains (documentManagement, {"scan new"}). Answers can be selected in arbitrary combinations, however, if the user selects "scan new" (because Scanning requires Archive) the answer "archive" is also selected and thereby also the Archive plug-in.

Views help dealing with the complexity of product line variability models that can easily contain thousands of assets and hundreds of decisions with thousands of often non-trivial dependencies among them. Our approach allows creating views on such large decision spaces. Decisions can be grouped in views which allow treating several decisions as one entity. Grouped decisions can also be prioritized within views. For product derivation, such views can be seen as a structure of the provided variability for those who have to take the decisions. In our example, the view ClaimsManagementView groups decisions relevant for claims agents while the view MailView groups decisions relevant for mail agents. Decisions may be part of more than one view, i.e., ClaimsManagementView also groups decisions that are relevant for the MailView.

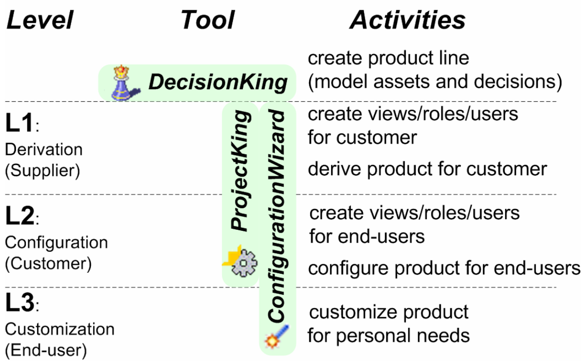
Roles allow defining responsibilities for views. For instance, the role MailAgent is allowed to see decisions in the MailView but not those in the ClaimsManagementView which are only available for the ClaimsAgent role.

Users. Different people are responsible for taking decisions. In our approach users can be assigned one or more roles. They have a default role and can switch roles. The roles define which views are visible to the user which in turn defines the decisions a user is allowed to see and take. In our example (cf. Figure 3), user Jane can take on the roles of a MailAgent or of a ClaimsAgent which makes the ClaimsManagementView and the MailView visible for her. User Joe only has the role MailAgent and can only work with decisions visible in the MailView.

All presented elements of our approach are useful for supporting the three levels of customization (cf. Figure 1). The supplier creates variability models with assets and decisions. Based on these models, the supplier derives a product for a particular customer. The supplier further defines views, roles, and users for its customers. Based on these views and roles, customers can further configure their product and define views, roles, and users for the end-users. The end-user eventually only has quite a small view on the provided variability but this small view allows him to customize the product to his particular needs determined by his role.

2.2 Tool support

We have developed three tools to support our decision-oriented product line approach. Figure 4 depicts how these tools support our three-level customization approach:



Level
Tool
Activities
DecisionKing
create product line (model assets and decisions)
L1: Derivation (Supplier)
ProjectKing
ConfigurationWizard
create views/roles/users for customer
derive product for customer
L2: Configuration (Customer)
create views/roles/users for end-users
configure product for end-users
L3: Customization (End-user)
customize product for personal needs

Figure 4. Tool support for the three levels of product customization.

DecisionKing [10] supports variability modeling and management. The tool allows to model assets and decisions with their attributes and dependencies. It can be parameterized by creating a meta-model defining the assets to be modeled and their possible dependencies. This allows creating variability models for arbitrary domains. Users of DecisionKing use a self-developed rule language (based on JBOSS Rules[1]) to define dependencies between decisions and assets. DecisionKing is based on a flexible plug-in architecture that allows extending it with arbitrary company-specific tools.

ProjectKing [18] supports preparing and guiding product derivation and customization. Based on the assets and decisions created with DecisionKing, the user of ProjectKing can define views, roles, and users. The tool also allows defining default answers for decisions. Furthermore, meta-information and recommendations on decisions can be modeled by using multimedia objects (e.g., audio or video files) that provide further details and rationale for taking decisions.

ConfigurationWizard [17] supports taking decisions in product derivation and customization. It displays decisions' questions for users based on their assigned roles and views. Currently required assets (calculated based on the taken decisions) can also be displayed. By default, ConfigurationWizard displays decisions as a flat list (cf. Figure 5). However, decisions can also be visualized in graphs and trees (based on decisions' dependencies) to ease navigation in complex variability models [19]. ConfigurationWizard is based on a flexible plug-in architecture that allows the integration of domain-specific configuration generators. In our case study (cf. Section 3), such a generator adapts a plug-in-based system at run-time based on the list of required assets (which changes by taking decisions) [27].

3. Case Study

We have conducted a case study in collaboration with our industrial partner BMD Systemhaus GmbH[2], a medium-sized company offering enterprise software products to 18.400 customers and 45.000 active users mainly in Austria, Germany, and Hungary. BMD Software is a comprehensive suite of enterprise applications for customer relationship management (CRM), accounting, cost accounting, payroll, enterprise resource planning, as well as production planning and control. BMD's target market is fairly diversified, ranging from small tax counselors to medium-sized auditing firms or large corporations. Customized products are an essential part of BMD's marketing strategy to address the needs of those markets.

3.1 Problem: Customizing legacy software

BMD's software has evolved over time to its current state but has originally not been created as a product line. Still it supports product customization on different levels through configuration mechanisms.

At the *supplier level* BMD offers its software as seven solutions that can be individually licensed and composed into five main products covering major markets, for example BMD-Consult for chartered accountants or BMD-Commerce for corporations. However, the deployed binaries are the same for each product, regardless of the actual product features. An

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

[2] http://www.bmd.com

individual license key shipped with the binaries determines which features are licensed. Unlicensed features can be either configured as visible but disabled or completely hidden from the user interface. While this approach worked well for a long time its downside is the monolithic application architecture resulting in huge binaries.

At the *customer level* configuration is accomplished in a similar fashion through permissions. A customer can build individual feature subsets for different departments by revoking permissions for unneeded features. Features for which a user lacks privileges can thus be hidden.

At the *end-user level,* the permission mechanism can also be applied to individual user accounts. A user account can be granted permissions to individual feature sets. However, since in practice this is typically done also by system administrators, end-users have only limited ways to customize.

BMD's customization approach of deploying the full feature set and simply hiding unused features resulted in three problems: (i) The application executable is about 90 megabytes in size, regardless of how many features are used. This leads to a high network load and constitutes problems for customers without a broadband connection when deploying patches over the internet. (ii) Another problem with patching is when multiple users use the same deployed application and one user urges a patch, while another user is worried to apply a patch now, because it might break other things. (iii) Furthermore, end-user customization by disabling features in the user account is cumbersome when end-users want to perform adaptations frequently, e.g., to specific working situations, several times a day.

3.2 Converting the legacy software to a product line

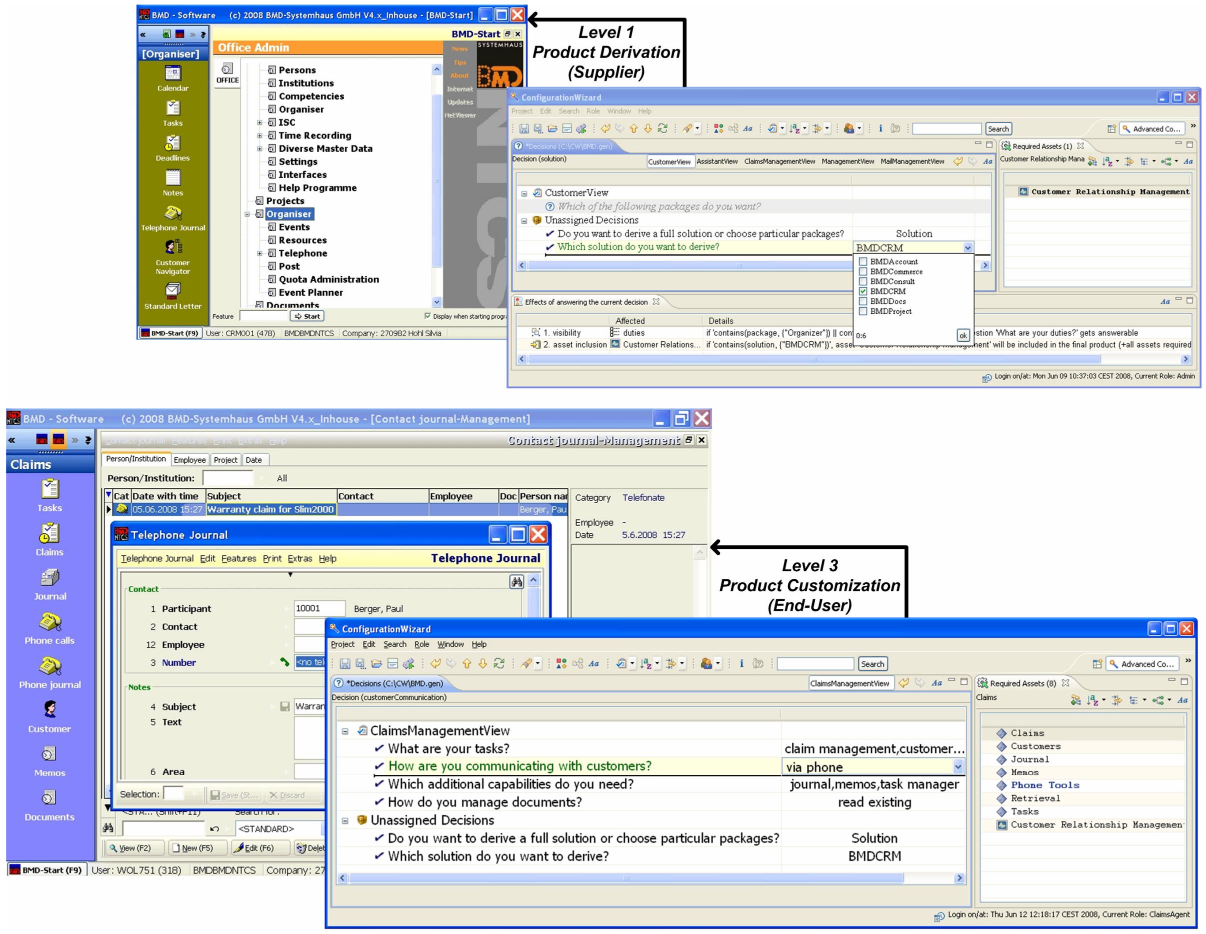
The main goal of our case study was to validate the multi-level customization approach using a real-world system. BMD's software has a total size of 4 MLOC. To test our approach, we decided to first improve the customizability of one significant subsystem of BMD's software. We chose BMDCRM which has a total size of about 890 KLOC (420 KLOC are specific to BMDCRM, 470 KLOC are framework code used by all of BMD's solutions). To be able to improve the customizability of BMDCRM at the supplier, customer and end-user level, we first had to decompose the monolithic legacy software into a small core system and a set of pluggable extensions [27]. Each extension should contain a single user-visible feature which can be integrated with the core system using plug-in techniques. To understand and document the possible combinations of these plug-ins and to support customization, we applied our product line approach and modeled the variability of the plug-in-based BMDCRM system:

We identified the features available in BMDCRM and organized them hierarchically on three levels of granularity: Fine-grained *plug-ins* contain single user-visible features that integrate individually into the application's user interface; *packages* combine tightly related features that are commonly used together into groups; coarse-grained *solutions* combine packages into a solution, in our case study only one solution (BMDCRM) was modeled. We defined plug-ins, packages, and solutions as product line asset types (cf. Figure 2). Solutions can contain packages and packages can again contain plug-ins. Plug-ins can require each other functionally. For example, the plug-in Scanning providing document scanning functionality requires the plug-in Archive providing document archiving functionality. Archive can be used without Scanning but Scanning requires Archive. We identified artifacts such as source code and resources related to a feature to decompose features to the granularity of plug-ins. We then reengineered the individual components such that they can be used with a plug-in platform as well as the core application such that plug-ins can be integrated. For the resulting plug-in solution (comprising 20 specific plug-ins and 28 components for the core system), we created a variability model by defining plug-ins, packages and solutions as assets and relating them with each other. We modeled possible adaptations as decisions for the three levels of customization. Decisions on higher levels abstract decisions on lower levels. The relations between decisions and assets are described using inclusion conditions (see Section 2). Because of these conditions, taking decisions allows determining the set of required solutions, packages, and plug-ins. Please note that there is no 1:1 mapping between decisions and plug-in assets. Taking a decision can include/exclude several plug-ins at once. Overall, for BMDCRM, the variability model contains 7 decisions, 20 plug-ins, 4 packages, and 1 solution. Figure 3 partly depicts this variability model. Some dependencies (especially those between assets) have been omitted in this figure for the sake of simplicity.

3.3 Applying the three-level customization approach

The following scenario illustrates how our tool-supported approach supports customizing BMDCRM according to the three levels of customization:

Level 1 – Product derivation by suppliers. Based on the product line variability model created with



CustomerView
Which of the following packages do you want?
Unassigned Decisions
Do you want to derive a full solution or choose particular packages? Solution
Which solution do you want to derive? BMDCRM

Affected	Details
1. visibility duties	if 'contains(package, {"Organizer"}) || con
2. asset inclusion Customer Relations...	if 'contains(solution, {"BMDCRM"})', asse

ClaimsManagementView
What are your tasks? claim management,customer...
How are you communicating with customers? via phone
Which additional capabilities do you need? journal,memos,task manager
How do you manage documents? read existing
Unassigned Decisions
Do you want to derive a full solution or choose particular packages? Solution
Which solution do you want to derive? BMDCRM

Figure 5. Product derivation and (run-time) product customization of the BMDCRM system with the ConfigurationWizard based on the product line variability model.

DecisionKing BMD can use ConfigurationWizard to derive a product for a customer. The partial example shown in Figure 5 shows two decisions: “Do you want to derive a full solution or choose particular packages?” and “Which solution do you want to derive?” A user might choose to derive the BMDCRM solution which configures the default BMDCRM product (top left in Figure 5) based on the underlying variability model. All subordinate lower level decisions are determined automatically and a list of required plug-ins is compiled. These can be deployed together with the plug-in run-time environment and constitute a product that can be shipped to customers.

The user at BMD then uses ProjectKing to create views on the variability, to define roles responsible for these views, and to specify users which can be assigned the roles. This way, BMD can define which open decisions later can be taken by the customer. The customer is not allowed to change the chosen solution but can only select additionally available packages and particular plug-ins not included in the solution by default.

Level 2 – Product configuration by customers. The binaries of the BMDCRM solution are deployed to the customer together with the BMD variability model. Decisions taken at the supplier level constrain the decision space for customers who use the variability model to configure products for internal use. The customer also gets the ProjectKing and ConfigurationWizard tools. Using ConfigurationWizard the customer can further configure the BMDCRM solution based on the variability model. In the example shown in Figure 5, this step is omitted for the sake of simplicity. Customers can choose additional packages and configure chosen packages by taking the decision "What are your tasks?" For example, the sales department typically answers “Customer Management” or “Claims Management” which includes packages from the BMDCRM solution. In the same way as at supplier level, based on the taken decisions, ConfigurationWizard computes a list of required plug-ins for each internal product. These plug-ins constitute the department's customized product and are deployed to individual department locations.

The customer uses ProjectKing to define the views, roles, and users for end-users. Thereby the customer specifies which end-users can take which decisions to further customize the solution to their personal needs.

Level 3 – Product customization by end-users. End-users are provided with an already pre-configured application. However, the application still allows further customization by end-users using the ConfigurationWizard. The tool utilizes the variability model and the views, roles, and users defined by the customer. The end-user level comprises two aspects of customization:

Customizing application environments for individual users. Different users have different tasks. For example, a user Joe in the sales department might be responsible for incoming mail orders and take the decision "What are your tasks?" by answering “Document Management”. For the subsequent decision "How do you manage documents?" he selects scanning and archiving incoming letters to customize the document management features.

Dynamically adapting the system to a working situation at hand. A user Jane might decide her responsibilities on certain occasions. Most of the time she is responsible for incoming mails just like Joe. But on some days she needs to step in for a colleague who is a claims agent. Since most of the time she does not deal with claims she does not need these features constantly present. In the example shown in Figure 5, Jane takes decisions to adapt the BMDCRM solution to her needs regarding claims management. Taking decisions automatically leads to run-time adaptation of the BMDCRM system [27]. For example, Jane answers the question “How are you communicating with customers?” with “via phone” which includes phone tools in BMDCRM. She answers the decision question “Which additional capabilities do you need?” with “journal”, “memos”, and “task manager” to denote that she needs to write a journal as well as memos and that she needs to manage tasks for claims management. Jane answers the question “How do you manage documents?” with “read existing” as she does not need to archive, scan, and/or import documents as a claims agent. Based on Jane’s answers, the required plug-ins are automatically selected and BMDCRM adapts at run-time. Whenever Jane changes her working situation from mails to claims she now just has to switch her role. Already taken decisions are stored but can be changed later.

3.4 Benefits

Even though in the case study we only focused on one of BMD’s solutions, we were able to test the feasibility of our tool-supported approach. Our three-level customization approach improves customizability on all three levels from product derivation down to end-user customization. On the supplier level BMD can use the new technology to generate the same solution as before, but additionally diverse combinations of plug-ins can now be derived from the product line by taking decisions. Decomposing the monolithic application results in smaller binaries to deploy; patches are much smaller and can be selectively deployed for affected components. When comparing that to the existing way of customizing we achieved a substantial improvement: Where in the prior approach the deployed binary used to be about 90 megabytes in size, the binaries now are just 25 megabytes, i.e., for the BMDCRM solution. Each additional solution adds another 5-15 megabytes. Patches also got a lot lighter since individual plug-ins now range from less than 100 kilobytes to a maximum of 2 megabytes for large framework components. At the customer level the configuration process is easier because of the tool support. Different combinations of packages and plug-ins can easily be selected by answering the questions presented with ConfigurationWizard. At the end-user level, where end-users before had to use a preference dialog in a user manager to manually activate or deactivate single features, they now can take high-level decisions with ConfigurationWizard to adapt the application for the working situation at hand. And where the earlier solution required the application to be restarted to conduct the required adaptation, the application now instantaneously adapts on-the-fly at run-time [27].

4. Related Work

We focus our discussion of related work on product derivation, configuration and personalization for end-users, as well as run-time adaptation in SPLE.

Product derivation. Compared to the vast amount of research results on domain engineering and the definition of software product lines, comparably few approaches and tools are available for product derivation. Deelstra *et al.* [7, 22, 23] present a product derivation framework supporting configuration in industrial product lines and report on problems and issues based on their industrial experiences. Chastek *et al.* [3] present a study on how different product line organizations create products. Czarnecki *et al.* [6] report on staged configuration with feature models, an approach to resolve variability step-by-step in product derivation. Halmans and Pohl [14] present work on how to communicate product line variability to customers. Ziadi *et al.* [28] describe product derivation by using extended UML notations for representing product line variability. Bayer *et al.* [2] describe PuLSE-I, a process for product derivation as part of the Product Line Software Engineering methodology developed at Fraunhofer IESE (Institute for Experimental Software Engineering).

Configuration and personalization for end-users. While product line approaches do typically not emphasize end-user customization this idea itself is not novel. Some software producers already go beyond providing basic installation wizards and provide more sophisticated support for end-users. For example, the enterprise business application SAP/R3[3] can be delivered with product configurators that allow users adapting the application for their specific needs. Another example is the SuSE Studio configurator[4] allowing end-users to construct a customized Linux distribution themselves using a web-based front-end. Such *configurators* are however not based on variability models that can be used at multiple levels during the customization process. *Personalization* aims at providing users with applications customized to their very specific needs and adapting "on-the-fly" if their needs change. For instance, e-commerce applications can adapt themselves automatically based on acquired user information to provide personalized services [1]. In contrast to such knowledge-based approaches we do not acquire the required information automatically but present customization choices to end-users based on product line variability models.

Run-time adaptation in SPLE. Diverse researchers in different areas have developed approaches and tools contributing to run-time adaptation of systems. However, only few approaches combine software product lines and run-time adaptation: Lee and Kang [15] propose a feature-oriented approach for dealing with run-time adaptation. In their approach, reconfiguration is based on identifying binding units in feature models. The authors do however only describe conceptual support for a reconfiguration tool with no actual implementation. Wang *et al.* [26] describe an approach based on patterns and rules to privacy that can be used to support feature adaptation of web applications at run-time. They also describe a prototypic implementation within the ArchStudio product line architecture tool. In [13] Hallsteinsen *et al.* present the MADAM approach which uses variability models to describe the choices for run-time adaptation of component-based architectures. The goal of MADAM is to support adaptation of mobile devices to changing environmental conditions such as available bandwidth or network connectivity. Their variability models therefore define choices based on sensed context information. Decisions are local to particular components and not stored in a complete decision model as in our case. While MADAM is context-centered our approach leaves the decisions to users at different levels.

5. Conclusions and Future Work

We presented a decision-oriented product line approach providing support for product customization on three levels. At the first level the supplier uses variability models to derive products from a product line customized to the requirements of a particular customer. At the second level the customer can refine the configuration of the product to address organization-specific aspects constrained by the variability model used by the supplier before. End-users can customize the product to their personal needs, again using the same variability model. Going from level to level means to resolve more variability. This is achieved in our approach by defining decisions, views on decisions, roles that are responsible for these views and users that can be assigned roles. We have presented three tools supporting our approach. DecisionKing [10] supports creating variability models, ProjectKing [18] allows suppliers and customers to define views, roles, and users, and Configuration-Wizard [17] is an easy-to-use end-user tool for taking decisions and thereby resolving variability.

An important conclusion is that *variability needs to be described in the language of the problem space*. To facilitate end-user product customization the variability offered by a complex software system needs to be described in the language of the end-user, i.e., in problem space language. Our initial decision models represented the technical structure of the system but did not provide the abstraction needed by end-users. For instance, it is better to ask a claims agent in what forms he wants to communicate with customers instead of asking whether particular mail and/or phone tool plug-ins should be included. Another example is asking the end-user how he wants to manage documents instead of asking about specific plug-ins for retrieving, importing, archiving and/or scanning.

Another lesson we learned is that t*he amount of variability resolved per level of customization is variable*. We have experienced that the amount of variability resolved by different users at different customization levels can differ from domain to domain. BMD does not resolve a lot of variability beforehand but delivers a highly customizable product to its customers who then configure the product. In another project where the system of interest was a software product line for automation of continuous casting in steel plants [11], we have seen that most of

[3] http://www.sap.com
[4] http://studio.suse.com/

the variability is resolved by the supplier and only minor configuration is done by customers.

End-user customization results in additional challenges for support which we will investigate in future work. In case of problems end-users seek for support, e.g., by contacting a help desk. Help desk staff needs capabilities to reproduce the exact configuration of the end-user. As every customization decision is stored in the underlying variability model in our approach we can utilize the model for exactly that purpose. We have been developing an initial prototype integrated with ConfigurationWizard that allows end-users to send requests to help desk staff over network. The help desk staff can then reproduce the user's configuration to handle the request.

The paper presented a case study of applying our tool-supported approach in the ERP domain together with our industrial partner BMD. In this case study, we have also developed run-time adaptation capabilities that allow customizing applications at run-time by taking decisions [27]. We will also apply the three-level customization approach in additional case studies to validate its usefulness in other domains. Also, we will convert additional BMD solutions to a product line to further test our approach and tools.

Acknowledgements

[redacted]

References

[1] G. Adomavicius and A. Tuzhilin, "Personalization Technologies: A Process-Oriented Perspective," *Communications of the ACM*, vol. 48(10), pp. 83-90, 2005.

[2] J. Bayer, C. Gacek, D. Muthig, and T. Widen, "PuLSE-I: Deriving Instances from a Product Line Infrastructure, "Proc. of the *7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, Edinburgh, Scotland, UK, IEEE Computer Society, 2000, pp. 237-245.

[3] G. Chastek, P. Donohoe, and J. D. McGregor, "A Study of Product Production in Software Product Lines," CMU/SEI-2004-TN-012 2004.

[4] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*: SEI Series in Software Engineering, Addison-Wesley, 2001.

[5] [redacted] *2nd International Workshop on Variability Modelling of Software-intensive Systems (VAMOS 2008)* [redacted]

[6] K. Czarnecki, S. Helson, and U. W. Eisenecker, "Staged configuration using feature models, "Proc. of the *3rd International Software Product Line Conference (SPLC 2004)*, Boston, MA, USA, Springer Berlin Heidelberg, 2004, pp. 266-283.

[7] S. Deelstra, M. Sinnema, and J. Bosch, "Product derivation in software product families: a case study," *Journal of Systems and Software*, vol. 74(2), pp. 173-194, 2005.

[8] [redacted] *Sixth Working IEEE/IFIP Conference on Software Architecture*, Mumbai, India, IEEE Computer Society, 2007 [redacted]

[9] [redacted] *IFIP WG 8.1 Working Conference on Situational Method Engineering: Fundamentals and Experiences*, Geneva, Switzerland, International Federation for Information Processing, Springer Series in Computer Science, 2007 [redacted]

[10] [redacted] *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 [redacted]

[11] [redacted] *32nd EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, Cavtat/Dubrovnik, Croatia, IEEE Computer Society, 2006 [redacted]

[12] [redacted] *23rd IEEE/ACM International Conference on Automated Software Engineering*, L'Aquila, Italy, IEEE/ACM, 2008 (to appear).

[13] S. Hallsteinsen, E. Stav, A. Solberg, and J. Floch, "Using Product Line Techniques to Build Adaptive Systems, "Proc. of the *10th international on Software Product Line Conference*, Baltimore, Maryland, USA, IEEE CS, 2006, pp. 141-150.

[14] G. Halmans and K. Pohl, "Communicating the Variability of a Software-Product Family to Customers," *Informatik - Forschung und Entwicklung*, vol. 18(3-4), pp. 113-131, 2004.

[15] J. Lee and K. C. Kang, "A Feature-Oriented Approach to Developing Dynamically Reconfigurable Products in Product Line Engineering, "Proc. of the *10th International Software Product Line Conference (SPLC 2006)*, Baltimore, MD, USA, IEEE CS, 2006, pp. 131-140.

[16] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles, and Techniques*: Springer, 2005.

[17] [redacted]

[redacted] *33rd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA'07)*, Lübeck, Germany, IEEE Computer Society, 2007[redacted]

[18] [redacted] *11th International Software Product Line Conference (SPLC 2007)*, Kyoto, Japan, IEEE Computer Society, 2007[redacted]

[19] [redacted] *11th International Software Product Line Conference (SPLC 2007), 1st International Workshop on Visualisation in Software Product Line Engineering (ViSPLE 2007)*, Kyoto, Japan, Kindai Kagaku Sha Co. Ltd., 2007[redacted]

[20] [redacted] *IEEE Intelligent Systems*, vol. 22(1)[redacted] 2007.

[21] 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(3), pp. 259-284, 2004.

[22] M. Sinnema and S. Deelstra, "Industrial Validation of COVAMOF," *Journal of Systems and Software*, vol. 81(4), pp. 584-600, 2008.

[23] M. Sinnema, S. Deelstra, and P. Hoekstra, "The COVAMOF Derivation Process, "Proc. of the *9th International Conference on Software Reuse (ICSR 2006)*, Turin, Italy, Springer Berlin Heidelberg, 2006, pp. 101-114.

[24] Software Productivity Consortium, "Synthesis Guidebook," SPC-91122-MC. Herndon, Virginia: Software Productivity Consortium 1991.

[25] F. van der Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action - The Best Industrial Practice in Product Line Engineering*: Springer Berlin Heidelberg, 2007.

[26] Y. Wang, A. Kobsa, and A. van der Hoek, "PLA-based Runtime Dynamism in Support of Privacy-Enhanced Web Personalization, "Proc. of the *10th International Software Product Line Conference*, Baltimore, Maryland, USA, IEEE CS, 2006, pp. 151-162.

[27] [redacted] *7th IEEE International Conference on Composition-Based Software Systems (ICCBSS)*, Madrid, Spain, IEEE Computer Society, 2008[redacted]

[28] T. Ziadi, J. M. Jezequel, and F. Fondement, "Product Line Derivation with UML, "Proc. of the *Software Variability Management Workshop*, Groningen, The Netherlands, 2003, pp. 94-102.