Supporting Runtime System Adaptation through Product Line Engineering and Plug-in Techniques[1]

Reinhard Wolfinger Stephan Reiter Deepak Dhungana Paul Grünbacher Herbert Prähofer
Christian Doppler Laboratory for Automated Software Engineering
Johannes Kepler University
A-4040 Linz / Austria
{wolfinger, reiter, dhungana, gruenbacher, praehofer}@ase.jku.at

Abstract

Product-line engineering and plug-in techniques pursue different but complementary goals. Software product line engineering strives for modeling the variability of software systems on different levels of abstraction, whereas plug-in systems support software extensibility, customizability, and evolution. We present an approach demonstrating the benefits of integrating those two areas and discuss the integration of a plug-in platform for enterprise software with an existing product line engineering tool suite. The plug-in platform provides extensibility as well as runtime reconfiguration and adaptation mechanisms on the .NET platform. Automatic runtime adaptations are attained by using the knowledge documented in variability models. We discuss several usage scenarios developed in cooperation with our industry partner confirming the need of our approach in the enterprise software domain. Finally, the approach is illustrated and validated by an ERP system family of our industry partner.

1. Introduction

Runtime adaptation of software systems is an area of research that has received considerable attention in areas such as software architecture, product line engineering, or self-adaptive systems. The need for runtime adaptation of systems is obvious for new development paradigms such as mobile and pervasive computing or service-oriented systems that rely on such techniques to deal with context changes. However, even in more traditional environments practitioners are demanding capabilities to adapt a system to changing working conditions during system operation. For instance, ERP (Enterprise Resource Planning) software is inherently complex and feature-rich. Runtime adaptation makes it easier to provide just the right set of features required to support a particular business process.

In the area of software product line engineering numerous approaches exist to model the variability of software systems on different levels of abstraction. For instance, features models [11] and orthogonal variability modeling approaches [1] are well established approaches to deal with variability. More recently, researchers have also started to adopt variability models to support runtime adaptation of systems [10]. A particular challenge lies in utilizing variability models such that users can perform the adaptations of an application in an intuitive and controlled manner.

Another difficulty lies in actually performing the desired changes using runtime adaptation techniques. The plug-in approach [2] enables developers to build applications that are inherently extensible and customizable to the needs of individual users. A small core application is extended with features implemented as components that are plugged in and integrated seamlessly with the core application at runtime. Plug-in approaches became popular with Mozilla Firefox [18] or the Eclipse Platform [7] but are flexible enough to improve extensibility and customizability in other domains.

In this paper we demonstrate the integration of product line engineering and plug-in techniques. Together with our industrial partner we have identified several usage scenarios for runtime adaptation in the ERP domain confirming the need of such an approach. Our approach is based on extending an existing product line tool suite with a plug-in system that is based on the

[1] This work has been conducted in cooperation with BMD Systemhaus GmbH, Austria, and has been supported by the Christian Doppler Forschungsgesellschaft, Austria.

.NET platform. In our integrated approach the user adapts a system to his working situation guided by a product line variability model. The plug-in platform instantly re-configures the desired system at runtime.

The paper is organized as follows: Section 2 discusses the usage scenarios for runtime adaptation in the ERP domain. Section 3 presents our approach and key capabilities supporting the usage scenarios. Section 4 illustrates the tools we have been developing and their integration. Section 5 presents the case study. Section 6 compares our work to related research. The paper rounds out with a conclusion and outlook to further work.

2. Runtime adaptation scenarios

BMD Systemhaus GmbH (www.bmd.at) is a medium-sized company offering ERP software products mainly to SMEs. The company has a significant market share in Austria, Germany, and Hungary. In cooperation with BMD we have developed a set of usage scenarios demonstrating the need for an integrated approach that uses feature configuration, dynamic plug-in extensibility, and architecture reconfiguration mechanisms. The scenarios are motivated by the ERP domain and BMD's market environment and additional scenarios may emerge in other domains. We outline the scenarios and discuss the benefits of our envisaged runtime adaptation approach.

Scenario 1: On-the-fly product customization for sales process. Sales staff of our industrial partner offers products based on pre-defined feature sets to customers. Usually, this sales process leads to long lasting discussions with customers about the value and cost of features as customers cannot explore and experience the system before it is purchased and installed. Also, it is increasingly difficult for salespersons to understand the complex dependencies among features to offer technically feasible solutions.

The integrated product line engineering and plug-in approach supports a more rapid and interactive sales process: Salespersons explore valid feature combinations together with customers guided by the dependencies defined in the variability model. Rapid reconfiguration of the system allows a live preview of the system by the customer taking into account the IKIWISI ("I know it when I see it") phenomenon. The salesperson can instantly demonstrate the software in the desired configuration and explain the provided functions. If desired by the software vendor the customer is continuously informed about the price of the current configuration.

Scenario 2: Guided system upgrades. It is very common that customers upgrade their system with new features after initial delivery. This is also an important business case for our industrial partner as such upgrades represent a significant share of their revenues. Currently, a sales process is reinitiated with the customer. The salesperson needs to understand the current system configuration to suggest the most useful new feature upgrades.

Backed with a product line variability model and a configuration tool the salesperson is aware of the already installed features and explores useful and valid extensions to negotiate the upgrade. Furthermore, customers can purchase new features themselves guided by a feature exploration tool without the explicit interaction of sales personnel.

Scenario 3: Renting features. It is an interesting business case for customers to rent and use particular product features for a limited period instead of purchasing them permanently. The product line and plug-in approach allows customers browsing the available rentable features, immediately installing features from a remote site, trying out features during an evaluation period, and using the features for a defined period. Customers can continuously keep track of the accumulated rental fees.

Scenario 4: Instant help desk support. BMD is currently handling up to 3000 help desk calls per day. Each customer has a very specific and unique software configuration the help desk staff needs to understand before answering users' questions. Traditional screen sharing applications are of little use in this case as help desk staff avoid working on real customer databases.

Our envisaged help desk support allows users to automatically transmit their current system configuration to the help desk staff. The system on the help desk is reconfigured to exactly match the configuration of the customer. It is important that this reconfiguration happens without delays: Restarting the system has to be avoided as rapid response is crucial for achieving high customer satisfaction.

Scenario 5: Role-specific views. Enterprise software is inherently complex and feature-rich as modern enterprises need to support a high number of success-critical business processes. Individual users often participate only in a few of these processes. Hence, the user interface of an enterprise application is often cramped with features not needed for a particular task.

The plug-in approach enables customization of systems and their user interfaces to individual tasks and responsibilities on the fly. This helps improving focus and reducing clutter. Users are involved in diverse business processes and tasks during their working day.

Dynamic reconfiguration relies on feature configurations for the different roles and dynamic switching of roles.

Scenario 6: Optimizing training. A major problem in training is that new users are often overwhelmed by the high number of features of the software application. A trainer explaining a basic feature has to guide the trainees through numerous menus and user dialogs to activate a function needed for the next training unit.

Obviously, it is more promising to offer an individually configured system to trainees in accordance with the training schedule. This allows starting with a small configuration showing only some basic functions and adding new features for each new training unit. Training can thus be organized in small steps adding complexity incrementally and in coordination with the training program.

3. Approach

Providing support for these scenarios requires an approach for modeling and managing the variability of the adaptable system together with capabilities performing the actual adaptation of the system at runtime.

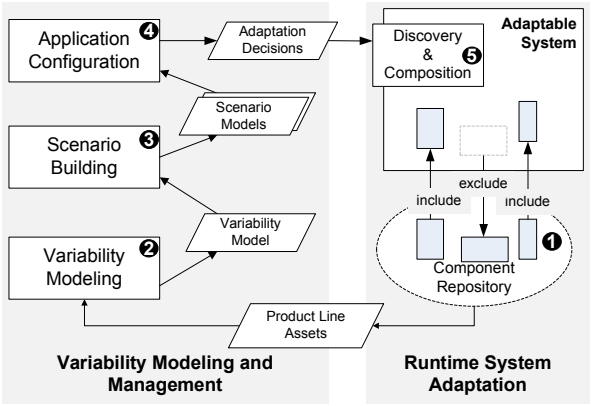

Fig. 1: Integrating variability management, configuration, and adaptation techniques.

Fig. 1 gives an overview about our approach. It relies on a software system that is organized as a set of reusable components stored in a *component repository* (#1) providing all features available. A system can be adapted by simply including and excluding components. The composition of components is done automatically.

Variability modeling (#2) allows to precisely define all possible ways in which the system can be adapted. The basis of variability modeling are the product line assets representing all available components. The variability model captures all meaningful adaptation decisions together with their technical implications, i.e., the list of components that need to be included/excluded based on user decisions. Adaptation decisions are expressed at a high-level taking a user-centered perspective. However, variability models also capture the technical dependencies of the product line assets.

The *scenario design* (#3) allows building subsets of the variability model to support different adaptation scenarios as described above. For instance, one sub-model may exist for training scenarios while another may define tasks, roles, and responsibilities. This customization addresses coarse grain adaptations that constrain the decision space for later end-user driven adaptations.

Application configuration (#4) is based on the scenario-specific variability models. It provides an easy-to-use interface for end users to take the desired decisions and thereby to initiate the necessary adaptation. It presents decisions to users in a structured manner taking into account the importance of decisions. For instance, an end user might decide to switch to a new working task thereby activating new features. The user configuration offers the set of sub-features which are meaningful in the new context.

Finally, our approach relies on a *discovery and composition* mechanism for components (#5) that allows building the system automatically based on users' adaptation requests. The variability model thereby works as the knowledge source to determine the effects of the adaptation, i.e., the necessary technical updates in form of components to include or exclude.

A typical scenario based on these capabilities looks as follows:

1. The software architect of the company offering a COTS software product reengineers the system into a repository of reusable, dynamically composable components.
2. The product line engineer defines the dependencies of the components and the technically allowed runtime variability of the system in a variability model.
3. The project manager analyses the required adaptation scenarios and defines useful customer- and scenario-specific variability models.
4. The end user uses a configuration tool to adapt the system to a new application context. The configuration tool uses a customer- and scenario-specific variability model to present the possible adaptation decisions to the user.
5. The discovery and composition tool determines the technical impact of the desired change. The runtime system performs the necessary change by including and excluding the required components.

4. Tool support

We have been realizing the approach outlined in Section 3 by integrating our product line tool suite [6] and plug-in platform [20], [21]. The DOPLER tool suite offers capabilities for variability modeling and high-level decision making. Our .NET-based plug-in platform offers the required runtime system adaptation capabilities by dynamic loading/unloading and composition of components. Fig. 2 shows the integrated approach as an instantiation of the approach outlined in Section 3 (cf. Fig. 1). The product line assets are provided as a repository of available plug-in components (#1). The tool DecisionKing (#2) is employed to set up the variability model. The tool ProjectKing (#3) allows defining scenario-specific configurations of the product line variability model. A ConfigurationWizard (#4) tool processes the scenario-specific variability model and presents decisions to the user in a wizard-like interface. Finally, a special discovery mechanism in the plug-in platform (#5) supports loading/unloading the requested plug-in components based on users' decisions.

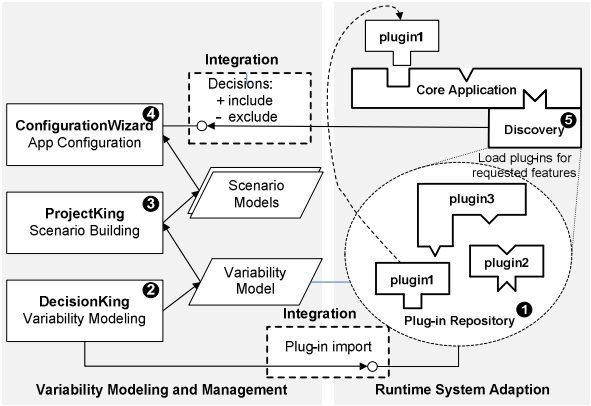

Fig. 2: Integration of PLE tools and plug-in techniques to support runtime adaptation.

In the following, we present the product line tools in Section 4.1 and the plug-in platform in Section 4.2. In Section 4.3 we discuss the integration of the product line tools and the plug-in platform.

4.1. DOPLER product line tools

Our DOPLER tool suite [6] comprises a set of highly integrated product line engineering tools that cover different capabilities needed in our approach:

DecisionKing (#2) [5] is a tool for variability modeling and management that has been developed on top of the Eclipse platform. The tool can be customized through a domain meta-model defining concrete asset types, their attributes, and relationships. For instance, we have defined a meta-model containing the asset type *plug-in* to support modeling the available plug-ins from the plug-in repository (#1) as assets of the product line variability model. The existing technical architecture of the system is also captured in the variability model which defines the dependencies between the core assets. The model also defines decisions, i.e., variation points that allow users to customize the system and allow selecting useful and desirable combinations of assets. Decisions can exist on different levels of granularity, i.e., as high-level decisions related to groups of features or low-level decisions on whether particular assets should be included. Decisions and assets are linked using inclusion conditions. This means that for each answer of the user the model specifies which assets will be included.

The variability model thus describes all assets and their dependencies and all possible decisions to derive a system from the product line. Even for a moderately-sized system such a model would be unsuitable for end users due to its size and complexity. We have thus devised the tool ProjectKing (#3) [15] which takes DecisionKing's variability models as input. The tool allows the definition of partly pre-configured variability models to constrain the decision space. Those are sub-models of the whole decision model built to support particular application contexts still leaving some space for special adaptations. In this way, variability models can be adapted and pruned for various usage scenarios. A sub-model is defined by selecting relevant parts from the decision model, by pre-answering decisions and locking them, and by defining roles and permissions for answering the remaining questions. For instance, a company might use the tool to define users and their permissions to adapt the system to their needs.

ConfigurationWizard (#4) [14] is an end user tool that uses the scenario-specific variability model created by ProjectKing and presents the possible adaptations to end users in form of questions. The answers to those questions then will result in inclusion/exclusion decisions for assets as modeled in the variability model.

4.2. .NET plug-in platform

The fundamental idea of a plug-in system is to provide users with a small core application they can easily extend with plug-ins to meet their requirements in a specific working situation. In our .NET-based plug-in platform [20],[21], a plug-in is a deployable .NET assembly with explicit specifications of its slots and extensions that supports customization of applications through addition, removal, and

replacement of components at runtime. A *slot specification* represents the contract for extending a plug-in (called *slot host*). Contributing plug-ins will provide respective *extensions* which fill the slot (cf. Fig. 3). In essence, slots declare the types of information a host plug-in expects and the extensions fill this information slots accordingly. In its simplest form a slot specification is a structured list of name/value-pairs where the slot specifies the required names and value ranges and the extension specification defines appropriate values for the extension at hand. The slot host will rely on the information provided to do the integration of the extension.

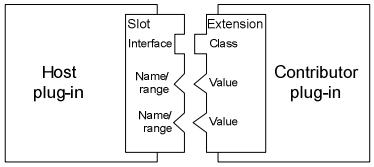

Fig. 3: Slot and extension in host and contributor plug-in.

The specification of slots and extensions is based on .NET custom attributes [13], i.e., meta-information that can be attached to language constructs such as classes, interfaces, methods, or fields in the source code of an application. Slots are not limited to behavior specification in the form of required and provided method interfaces, but allow any type of information. For example, slots will specify the set of interfaces to be implemented by extensions but also information like user interface properties, or properties configuring the execution environment for plug-ins. For instance, our runtime environment features security properties to restrict a plug-in's code access security privileges.

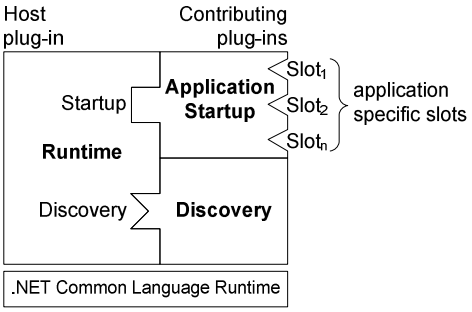

Fig. 4: Plug-in runtime environment and contributing plug-ins.

The runtime environment provides a very small fixed core which defines just two slots, i.e., a slot for replaceable discovery and a slot for startup extensions (Fig. 4). A plug-in based application is composed by initially activating a discovery plug-in, then discovery looks for startup plug-ins which, when activated, typically open further application specific slots where plug-ins with further functionality can be attached.

In the remainder of this section we focus on three features of our plug-in platform that are essential for the integration of runtime adaptation and product-line engineering: customizable discovery, dynamic loading and activation, and dynamic unloading and deactivation of plug-ins.

Customizable discovery

In a plug-in based system a *discovery mechanism* is needed to find out which plug-ins are available and should be loaded into the currently running application. In our platform the discovery mechanism is also realized as a plug-in which is plugged into the discovery slot of the core runtime. This approach improves both customizability and adaptability: (i) Developers can either use a default discovery implementation or they can implement individual discovery schemes satisfying their special needs. (ii) Applications can adapt to changing environments more easily as the discovery plug-in can be replaced at runtime and multiple simultaneous discovery plug-ins are supported. For example, there could be one discovery mechanism for the enterprise network and another for mobile working environments. The first could be used while being in office and substituted by the second as soon as leaving house.

To bootstrap the process, the runtime environment loads an initial discovery plug-in from a predefined location. We currently use a discovery plug-in which treats a specific folder in the file system as its plug-in repository. At startup time this discovery plug-in browses the folder to find available plug-ins to load. During runtime it continuously monitors the repository folder for modifications. Any time a plug-in assembly file is added to, removed from, or exchanged in the repository folder, it sends a respective notification to the runtime system to load and activate, unload and deactivate, or update the respective plug-in.

Plug-in discovery is now used to accomplish runtime adaptation of the discovery mechanism itself. As soon as a new discovery plug-in that implements an extension for the discovery slot is discovered it is loaded and plugged into the slot. There it either substitutes the current one or, depending on the strategy, is added as an additional discovery mechanism.

Dynamic loading and activation

After a plug-in has been found by the discovery mechanism the runtime takes actions to load and activate it. Loading is based on .NET assembly loading

provided by the .NET Common Language Runtime (CLR) but augmented with additional functions as follows:

(1) *Lazy loading and static integration*: By utilizing .NET capabilities to read meta-information of assemblies without loading the code, plug-in integration starts by exploring the plug-in assembly for extension specifications and their properties. Newly discovered plug-ins initially are only attached to hosts, i.e., a notification is sent to all plug-ins with matching slots, thereby giving hosts a chance to perform static integration based on declarative information. For instance, a host could create user interface widgets such as menu or toolbar items solely by exploiting meta-information and defer the loading of the implementation to the time when the user actually clicks the widget.

(2) *Dynamic activation*: Dynamic activation means creating instances of plug-in objects, to wire up host and contributing plug-in, and to create a communication path so that a host and an extension can call each other.

(3) *Plug-in isolation*: The dynamic loading mechanism automates loading of plug-ins into different application domains or operating system processes and set up respective isolation boundaries. Based on isolation settings provided by the host, a plug-in will be loaded into different application domains or processes and appropriate communication paths will be established. As outlined in [21] such mechanisms are essential to allow unloading of plug-ins at runtime (see also below), or to support fault tolerance in application systems.

Dynamic unloading and deactivation

In order to keep a system slim and perfectly conforming to the desired feature set in a current working situation, not only the capability to add features at runtime is required, but also to remove them when they are no longer needed. Our runtime environment supports deactivation of features with or without unloading plug-in assemblies from main memory. The reason for this distinction lies in the CLR, which cannot unload individual assemblies [16]. To allow unloading of plug-in assemblies, our runtime environment adds a declarative mechanism to load and isolate plug-in assembly groups into separate .NET application domains. All the assemblies in one application domain can then be unloaded together by the CLR. Unless separate application domains are used, deactivation of plug-ins is limited to feature deactivation with the assembly staying in memory.

Usually unloading takes place when the discovery mechanism determines that the plug-in should be detached. Thereupon, the discovery plug-in sends a *detachment notification* to the runtime which will remove corresponding extensions from slots and release their object instances.

4.3. Integration

Integration of DOPLER PLE engineering tools and the .NET plug-in platform is accomplished by two mechanisms as shown in Fig. 5: (a) asset import and (b) decision-based plug-in discovery and activation.

Asset import: The DecisionKing tool uses a programming interface of the plug-in runtime to import available plug-ins as assets from a repository. A discovery plug-in which browses a repository folder is used to get all the available plug-ins which are then imported to a variability model in DecisionKing as product line assets.

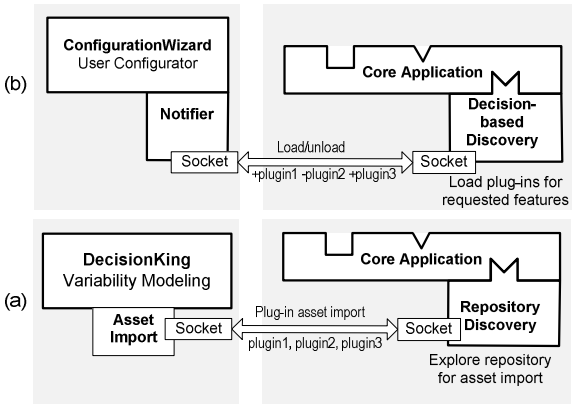

Fig. 5. Integration: (a) asset import, (b) decision-based discovery and activation.

Decision-based plug-in discovery and activation: Our integration approach connects the ConfigurationWizard front-end, where the end user initiates adaptations, with the discovery and composition mechanism in the plug-in runtime environment. Possible adaptations are presented to the end user by the ConfigurationWizard tool in form of questions. The necessary architectural adaptations are determined based on the variability model and sent to the plug-in runtime which initiates loading and unloading of plug-ins.

Fig. 5 (b) illustrates the technical realization of the integration. On the side of the PLE tool suite a *Notifier* component reacts to user decisions by determining changes in the required assets. Those changes are signaled as notification events. Each notification event includes data that reflects the changes in the assets required. On the side of the plug-in platform a special discovery plug-in listens to and evaluates notification

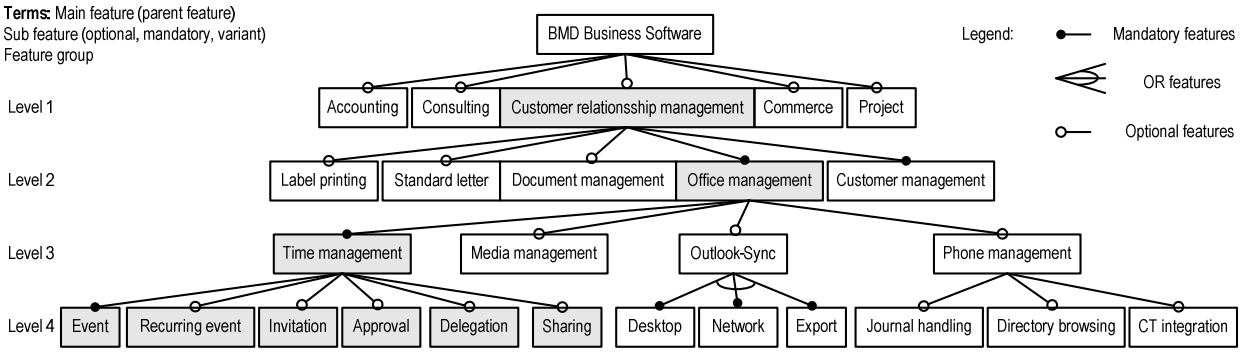

Fig. 6: Partial BMDCRM feature model.

events. It determines the plug-ins it has to load or unload and initiates the loading and unloading processes as described in Section 4.2. *Berkeley sockets* are used to bridge the Java-based DecisionKing and .NET-based plug-in technology.

5. Case study

To validate our approach we have been conducting a case study in collaboration with our industrial partner BMD. The object of our case study is the customer relationship management product BMDCRM, which is part of a larger suite of enterprise application systems. BMDCRM is a monolithic Delphi 7 application meaning that all customers get the same application binary and individual license codes determine whether particular features are active or not. Runtime system adaptation on an architectural level presumes fine-grained modularization where each feature is contained in an individual component. In a first step the original Delphi 7 application (unmanaged Win32) has been ported to managed Delphi.NET and has been decomposed into distinct components guided by BMDCRM's feature model (see Fig. 6).

BMDCRM is a large application with hundreds of features and 1.2 million lines of source code. Therefore the reengineering effort has been organized in two phases: In phase 1, which has already been completed, the system has been decomposed into components implementing the features up to level 3 in Fig. 6. Phase 2 has started recently and will evolve the decomposition to the finer grained level 4.

In a second step the components are reengineered such that they can serve as plug-ins in our .NET plug-in platform by adding slot and extension specifications. The plug-in based application mirrors the feature model as shown in Fig. 6 (see Fig. 7). A thin core application comprises some core libraries and the plug-in runtime. A main application window is plugged into the *Startup* slot and the discovery plug-in into the *Discovery* slot. The main window plug-in is a multiple document container and is used to accept the plug-ins realizing the main features. For example, the *Office Management* feature is implemented by the *Organizer* plug-in, where in turn additional plug-ins like *Media Manager*, *Phone Tools*, *Outlook-Sync,* or *Calendar* can get attached. *Calendar* provides basic time management functionality and in turn will allow further plug-ins for sub-features, such as *Recurring events, Event approval,* or *Event delegation* to be added (realization of those features in distinct plug-ins is subject of phase 2).

The plug-ins have been imported in DecisionKing as assets and possible dependencies between assets have been defined. This asset base represented the basis for building a high-level decision model which is intended to model all possible adaptation decisions. At the higher levels, decisions take the form of user-centered questions like "Do you want to manage time?" with "yes" and "no" as possible answers. When a user takes such a decision the ConfigurationWizard tool determines the assets to be included/excluded based on the variability model. For example, when the user answers "yes" to the above question, the corresponding *Calendar* plug-in together with all the plug-ins *Calendar* depends on, such as *Organizer* and *Main Window*, will be included and immediately loaded into the running application.

Beyond reengineering the monolithic application into a set of plug-in components, we are aiming to demonstrate new usage scenarios benefiting from runtime system adaptation as outlined in Section 2. For illustrative purpose we have built a decision model for the role-specific views scenario. We have taken fictitious user roles *engineer*, *manager* and *assistant* from an engineering company. We have assigned each role a different feature set according to their job descriptions (see Fig. 8). The engineer only uses *Time management* with simple *Event* and *Recurring event*

sub-features. A manager has to have the possibility to approve and delegate events (sub-features *Delegation* and *Approval*) and additionally synchronizes his calendar with Microsoft Outlook (*Outlook-Sync*). The assistant uses *Invitation* and *Sharing* sub-features and additionally needs *Phone* and *Media management* to book meeting rooms or media equipment.

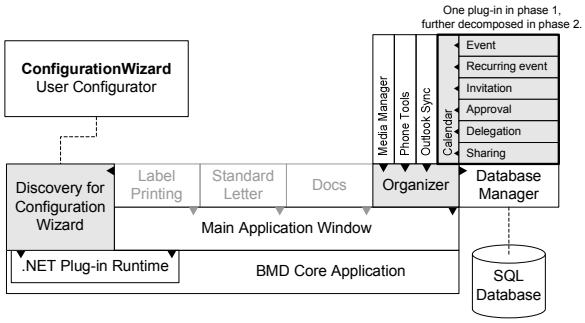

Fig. 7: Plug-ins in the BDMCRM software system.

Role switching is modeled as one high-level decision in DecisionKing, i.e., as a user question "What is your role?" with possible answers “Engineer”, “Assistant”, or “Manager” (multiple choices are possible). When this higher-level role decision is taken, several subordinate decisions are determined, e.g., when the answer is “Assistant”, decisions "Do you manage time?" and "Do you manage media?" are set to "yes" and the corresponding assets get included.

	Engineer	Assistant	Manager
Time management	●	●	●
- Event	●	●	●
- Recurring event	●	●	●
- Invitation		●	
- Approval			●
- Delegation			●
- Sharing		●	
Media management		●	
Outlook-Sync			●
Phone management		●	

Fig. 8: Feature sets of different roles.

Change of roles can be done instantaneously. For example, a manager might need to change to the assistant role, as his assistant is off work. To change to the assistant's role, the manager invokes the ConfigurationWizard tool where he chooses “Assistant” from the list of available roles. The resulting change in assets is forwarded to the discovery plug-in. Dispensed plug-ins are unloaded, added plug-ins are loaded, the application adapts its feature set on-the-fly, and the application system becomes customized to the particular role as shown in Fig. 9. Because such adaptations are conducted instantaneously at runtime, a user can change roles back and forth without restarting the application.

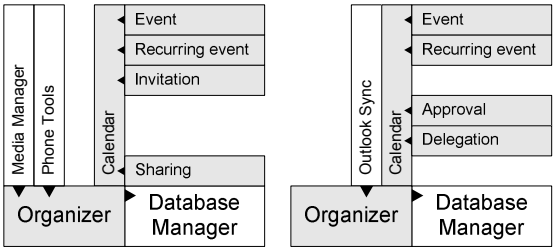

Fig. 9: Loaded plug-ins for the Assistant and Manager roles.

6. Related work

Numerous researchers from different areas have developed approaches and tools contributing to runtime adaptation of systems (see [20] for a comparison of our plug-in platform to other approaches). However, only a few approaches exist which combine product line engineering and runtime adaptation of systems. For instance, [12] have proposed a feature-oriented approach for dealing with runtime adaptation. Their approach is based on identifying binding units in feature models that serve as the basis for later reconfiguration. The authors do however only provide conceptual support for a reconfiguration tool with no actual implementation. The work of [19] shows how product line architectures can be used to support feature adaptation in the area of Web system personalization. Their approach is based on patterns and rules to privacy. A prototype implementation is discussed based on the ArchStudio PLA tool.

The MADAM approach presented in [10] is based on variability modeling and component-based architectures and shares some similarities with our work. MADAM is also based on a platform for runtime adaptation and extensions. The component and instance management platform allows discovering components at runtime to support adaptability and extensibility. Moreover, as in our approach a runtime representation of the architecture variability model is used to guide system adaptation and reconfiguration. However, our approach also differs from MADAM in several respects, which we see grounded mainly in the different goals pursued. The goal of MADAM is to support system adaptation of mobile devices to changing environmental conditions such as available bandwidth or network connectivity. Their variability model defines architecture decisions based on sensed context information. The decisions are local to particular components and a heuristic search is applied for finding an optimum system configuration in a set of

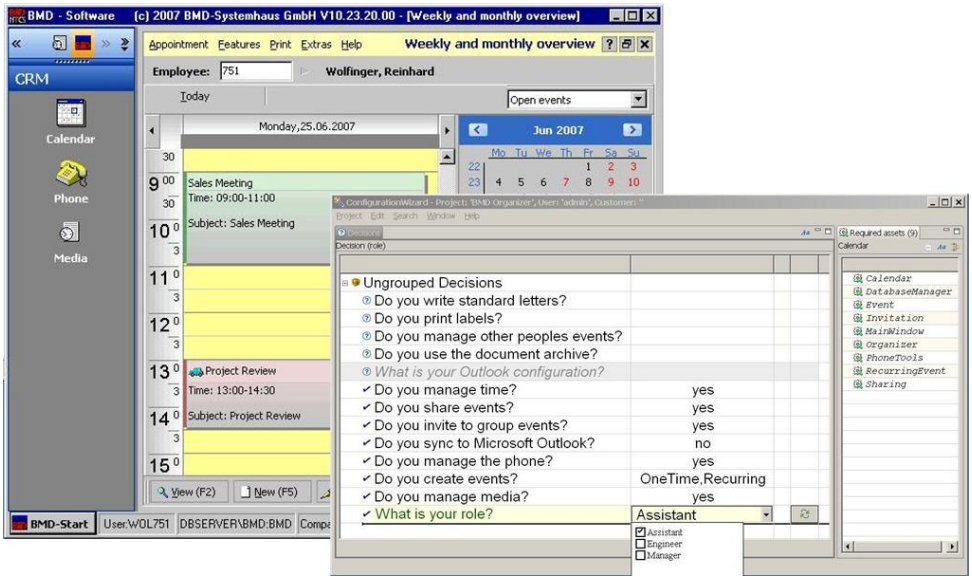

Fig. 10: Screenshot showing ConfigurationWizard dialog (front window) and adapted CRM system (back window). The ConfigurationWizard presents possible adaptations to end users in form of questions. Taking a decision immediately reconfigures the CRM system. The right pane shows a list of currently loaded plug-ins which is displayed for illustration purposes only.

local decisions for a current context. The authors argue that a complete decision model which represents all the possible decisions is very complex and difficult to evolve. Although this is certainly true in their domain, we, however, strive for such a complete decision model in our approach. Actually, in our approach we try to aggregate a set of small, local decisions to several high-level decisions which are meaningful for the user. So, while the decisions of MADAM are context-centered, our variability decision model is user-centered. Moreover, the approaches differ in their architecture style pursued. In MADAM, a classical object-oriented approach is pursued where component variability is realized by polymorphic components. Our approach, however, adopts a plug-in approach which provides a higher degree of flexibility as shown in Section 4.2.

In the area of requirements engineering researchers have explored runtime deviations of systems from original requirements. In [8] an approach based on goal models specified in the formal language KAOS is presented. The approach adopts a set of agents to monitor runtime behavior of systems and to suggest either automated or runtime adaptation of the systems. Variability is expressed via alternative refinements in goal models. The approach has been illustrated in several domains [9].

In [3] different levels of requirements engineering for dynamic adaptive systems have been explored. There aim is to provide a general framework bridging human-centered requirements and machine-centered adaptation mechanisms. A similar framework has been proposed by [4] in the area of multi-stakeholder distributed systems, illustrated with examples from the area of service-oriented systems. The authors emphasize the need for integrating different modeling techniques (negotiation models, goal-models, variability models) needed to inform dynamic adaptation of systems. In [22] an approach based on Petri nets to formally specify the behavioral changes of adaptive programs has been suggested.

7. Conclusion and further work

In this paper we have presented the integration of a product line tool suite and a plug-in platform for supporting runtime adaptation of systems. The plug-in platform facilitates runtime adaptation and composition of systems by loading and unloading of plug-ins. The product line tools are used for modeling high-level adaptation decisions together with their technical implications and to present them to users in easy-to-use wizard-like dialogs. Together with our industrial partner we have developed advanced usage scenarios and have shown the feasibility and usefulness of the approach in a case study. Our integrated approach allows precisely modeling high-level adaptation decisions together with their technical implications. Furthermore we can create multiple decision models

for distinct adaptation scenarios and support the user with different, scenario-dependent adaptation dialogs which are automatically generated from the decision models. According to our industrial partner, the combination of allowing highly customized working environments and supporting different adaptation scenarios has great potential to achieve both, higher user satisfaction and improved in-house productivity, in particular in sales processes, help desk support, and training.

In future work we will develop variability models for further scenarios. We will also conduct studies together with our industry partner to validate the effectiveness and efficiency of our approach in different scenarios. We also aim to improve the integration of the ConfigurationWizard and the end user application.

8. References

[1] Bachmann, F., M. Goedicke, J. Leite, R. Nord, K. Pohl, B. Ramesh, and A. Vilbig, "A Meta-model for Representing Variability in Product Family Development Software Product-Family Engineering", *5th International Workshop PFE 2003*, Siena, Italy, Nov. 4-6, 2003.

[2] Beck, K., and E. Gamma, *Contributing to Eclipse*, Addison-Wesley, 2003.

[3] Berry, D., B. Cheng, and J. Zhang, "The Four Levels of Requirements Engineering for and in Dynamic Adaptive Systems", *11th International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ'05)*, Porto, Portugal, June 2005.

[4] Clotet, R., X. Franch, P. Grünbacher, L. López, J. Marco, M. Quintus, and N. Seyff, "Requirements Modeling for Multi-Stakeholder Distributed Systems: Challenges and Techniques", *1st IEEE Int. Conf. on Research Challenges in Information Science*, Quarzazate, Apr. 23-26, 2007.

[5] Dhungana, D., P. Grünbacher, and R. Rabiser, "Domain-specific Adaptations of Product Line Variability Modeling", *IFIP WG 8.1 Working Conference on Situational Method Engineering: Fundamentals and Experiences*, Geneva, Switzerland, Sep. 12-14, 2007.

[6] Dhungana, D., R. Rabiser, P. Grünbacher, K. Lehner, and C. Federspiel, "DOPLER: An Adaptable Tool Suite for Product Line Engineering", *11th International Software Product Line Conference (SPLC 2007)*, Kyoto, Japan, Sep. 10-14, 2007.

[7] *Eclipse Platform Technical Overview*, Object Technology International, Inc., www.eclipse.org, 2003.

[8] Feather, M. S., S. Fickas, A. Van Lamsweerde, and C. Ponsard, "Reconciling System Requirements and Runtime Behavior", *Proceedings of the 9th international Workshop on Software Specification and Design*, Washington, DC, April 1998.

[9] Fickas, S., M.S. Feather, "Requirements monitoring in dynamic environments", *Second IEEE International Symposium on Requirements Eng.*, 1995, p. 140.

[10] Hallsteinsen, S., E. Stav, A. Solberg, and J. Floch, "Using Product Line Techniques to Build Adaptive Systems", *Proceedings of the 10th international on Software Product Line Conference*, Washington, DC, Aug. 21-24, 2006, pp. 141-150.

[11] Kang, K., S. Cohen, J. Hess, W. Novak, and S. Peteson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study". Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[12] Lee, J., and K.C. Kang, "A Feature-Oriented Approach to Developing Dynamically Reconfigurable Products in Product Line Engineering", *Proceedings of the 10th international on Software Product Line Conference, Washington*, DC, Aug. 21 - 24, 2006, pp. 131-140.

[13] Microsoft, *Microsoft C# Language Specifications*, Microsoft Press, Redmond, 2001.

[14] Rabiser, R., D. Dhungana, P. Grünbacher, K. Lehner, and C. Federspiel, "Involving Non-Technicians in Product Derivation and Requirements Engineering: A Tool Suite for Product Line Engineering", *15th IEEE International Requirements Engineering Conference (RE'07)*, New Delhi, India, Oct. 15-19, 2007.

[15] Rabiser, R., P. Grünbacher, and D. Dhungana, "Supporting Product Derivation by Adapting and Augmenting Variability Models", *11th International Software Product Line Conference (SPLC 2007)*, Kyoto, Japan, Sep. 10-14, 2007.

[16] Richter, J., *Applied Microsoft .NET Framework Programming*, Microsoft Press, Redmond, 2002.

[17] Schmidt, H. W. et al., "Predictable Component Architectures Using Dependent Finite State Machines", *9th International Workshop RISSF 2002*, Springer-Verlag, 2004.

[18] Shaver, M., and M. Ang, "Inside the Lizard: A Look at the Mozilla Technology and Architecture", www.mozilla.org, 2000.

[19] Wang, Y., A. Kobsa, A. van der Hoek, and J. White, "PLA-based Runtime Dynamism in Support of Privacy-Enhanced Web Personalization", *Proceedings of the 10th international on Software Product Line Conference*, Washington, DC, Aug. 21-24, 2006, pp. 151-162.

[20] Wolfinger, R., D. Dhungana, H. Prähofer, and H. Mössenböck, "A Component Plug-in Architecture for the .NET Platform", *Proceedings of 7th Joint Modular Languages Conference*, JMLC 2006, Oxford, UK, September 13-15, 2006.

[21] Wolfinger, R., and H. Prähofer, "Integration Models in a .NET Plug-in Framework", *SE 2007 Conference on Software Engineering*, Hamburg, Germany, March, 2007.

[22] Zhang, J., and B.H. Cheng, "Model-based development of dynamically adaptive software", *28th International Conference on Software Engineering*. Shanghai, China, May 20-28, 2006, pp. 371-380.