A Dynamic Software Product Line Approach using Aspect Models at Runtime

Tom Dinkelaker[1] Ralf Mitschke[1] Karin Fetzer[2] Mira Mezini[1]

[1]Technische Universität Darmstadt, Germany
{dinkelaker,ralf.mitschke,mezini}@cs.tu-darmstadt.de

[2]SAP Research Dresden, Germany
karin.fetzer@sap.com

ABSTRACT

Dynamic software product lines (DSPLs) are software product lines, which support late variability that is built into the system to address requirements that change at runtime. But it is difficult to ensure at runtime that all possible adaptations lead to a correct configuration. In this paper, we propose a novel approach for DSPLs that uses a dynamic feature model to describe the variability in the DSPLs and that uses a domain-specific language for declaratively implementing variations and their constraints. The approach combines several trends in aspect-oriented programming for DSPLs, namely dynamic aspects, runtime models of aspects, as well as detection and resolution of aspect interactions. The advantage is, that reconfigurations must not be specified for every feature combination, but only for interacting features. We have validated the approach in an example dynamic software product line from industry and preliminarily evaluated the approach.

Keywords

Dynamic Software Product Line Engineering, Dynamic Feature Models, Domain-Specific Languages

1. INTRODUCTION

Large scale information technology infrastructures are the backbone of many enterprise processes. Yet these systems are driven to continuous adaptation, due to changing requirements [10]. However, the evolutionary transitions for crucial enterprise information systems must be smooth and not hamper current running business processes [18]. Hence, methods and mechanisms for dynamic adaptation of the software system are required.

The challenge of building dynamically adaptable software systems is how to define suitable methods and mechanisms for the dynamism. An ad-hoc approach is to use existing variability mechanisms (e.g., *if*-statements, method dispatch) directly in the architecture, and/or the underlying implementation. However, lacking appropriate methodologies for building such software systems, the rising complexity (i.e., number of configurations, complex re-configuration relationships) can limit the number of dynamic re-configuration points to a few well-defined ones (e.g., all points at which variability must be supported must be known at design time, and the corresponding if-statements and all possible variations must be provided).

Dynamic software product lines (DSPLs) [11, 13] are an emerging field that can systemize the configuration space in dynamically adaptable software system. Thus, DSPLs break down the complexity of managing dynamic re-configuration points by modeling them explicitly in a product line approach as *late variability* [22]. Central to software product lines (SPLs) are features, where a feature is a distinct property of the software product. The *late variability* can be represented through *dynamic features*, i.e., features that can be (de-)activated in a running software system.

A research challenge for DSPLs is to find suitable variability mechanisms to support *dynamic features* in the underlying architecture and implementation. The mechanism must not constrain the range of existing techniques used to build product lines, i.e., it should peacefully co-exist with model-driven and generative techniques. Further, the mechanism should be able to cope with dynamic features that affect several modules and require modification of several classes or components in the product line. It should also detect and resolve interactions between features, in particular feature interactions that are not statically detectable but arise for a set of dynamic contexts of the configuration.

An interesting research question for *aspect-oriented programming* (AOP) [16] is how far dynamic AOP [2, 20, 19, 5, 7] is capable as a variability mechanism for dynamic SPLs. For example, dynamic aspects [4] have been used to implement business rules. Although dynamic AOP solutions provide a flexible variability mechanism, they do not provide appropriate support for declaring, detecting, and resolving dynamic interactions. Most dynamic AO solutions only provide support for defining the precedence of aspects [2], i.e., defining the execution order of their advice. But these precedence relations are inappropriate for expressing exclusions, i.e., one cannot declare that one aspect does not allow another aspect to be present. Furthermore, dynamic dependency relations between the features implemented by aspects cannot declare that one aspect needs another aspect to work correctly. The works in [20, 5] support static declaration of exclusions and dependencies between aspects, but do not address dynamic interactions.

First Workshop on Composition and Variability'10 Rennes, France colocated with AOSD'2010

In DSPLs the dynamicity in the exclusion of features is of great interest. The problem with static exclusions and dependencies is that they must be declared permanently (e.g., aspect `A` always excludes aspect `B`) and are enforced independent of the fact that aspects `A` and `B` may not actually conflict because the aspects are never applied at the same points at runtime. This is too conservative and disallows meaningful compositions of dynamic features.

For example, a workflow requires an approval step, which can be automatic manual. Modularizing the variability in the approval step entails interaction between the automatic and the manual variants, because the goal of immediate automated approval is violated by waiting for a manual approval. The repetitive manual approval following an automated appoval would be a waste of human resources.

However, using more powerful declaration mechanism, we can declare that both dynamic features be selected at once and affect different parts of the DSPL, with the exception of cases in which conflict occurs. For example, when the automatic approval is only applied to orders that have a certain state (e.g., exceeds a certain amount), a conflict occurs only for those particular orders and must be resolved only when this runtime condition is satisfied. The other orders are only affected by the manual approval without any conflicts.

In this paper, we propose mechanisms to detect and resolve such context-dependent interactions between features in a DSPL. The contribution of this paper is twofold. On the one hand, we adopt DSPLs as a systematic framework in which complex dynamic software systems can be planned and managed by modeling late variability. On the other hand, we provide support for detecting and resolving context-dependent interactions by validating an aspect-oriented (AO) model at runtime.

First, we extend existing DSPL approaches by a novel notation termed *dynamic feature model* that allows us to model *late variability*. Feature models are a widely used notation, that models the configuration space of software product lines, yet they lack explicit support to model *late variability*. Our notation extends feature models to capture *dynamic features*, i.e., features that may be (de-)activated at runtime, and to model their runtime constraints. Thus we provide an explicit representation of dynamic re-configuration points in an adaptable software system.

Second, we propose a novel approach for DSPLs on top of a dynamic AO runtime environment with a meta-aspect protocol [7] that is used as a *dynamic feature manager*. We map dynamic feature models to aspect-oriented models that are available as first-class entities in running products. DSPLs are delivered with the AO runtime environment, through which (de-)activation of dynamic features is enabled, which updates dynamic feature model representation, takes care of the composition of dynamic features, as well as it detects and resolves dynamic feature constraints.

We have validated the approach by evolving a static software product-line from industry into a DSPL in a case study. Furthermore, we have evaluated the performance overhead of dynamic aspects in the context of SPLs. The results indicate that the approach copes well with performance requirements, also in the presence of the special scalability requirements of SPLs.

The remainder of this paper is structured as follows. Sec. 2 illustrates the example SPL and motivates late variability. In Sec. 3, we present dynamic feature models for modeling late variability and give an overview of the methodology. In Sec. 4, we describe how dynamic aspects can be used to realize dynamic SPLs. Sec. 5 presents the case study. Sec. 6 evaluates the approach. Sec. 7 discusses related work. Sec. 8 concludes the paper and outlines future work.

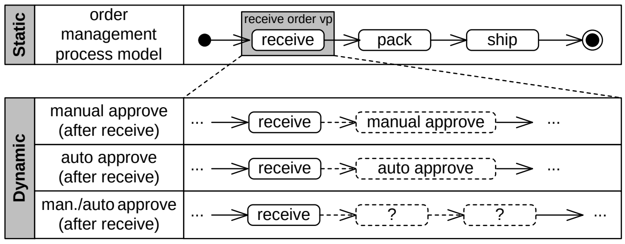

Figure 1: Late variation point for approval step in the order management business process

2. LATE VARIABILITY IN AN SPL

Our example SPL is the *Sales Scenario*, which is a software system for the management of business data, including central storage and user-centric information inquiry. The main focus of the *Sales Scenario* is on stock sales processes, where the core processes are customer order management, payment, account management, stock management and communication. The main goal of the *Sales Scenario* is to integrate all processes and corresponding data of an organization into one system. The system addresses sales processes for both mid-sized and large enterprises. The business processes themselves are customizable, often in multiple independent variations. For example the order management process can include a quotation management (i.e., placing strategic offers to customers) or sales order processing (i.e., tracking of individual orders).

The customer needs for these features vary depending on the business size of the customer, thus making it advantageous for the software provider to implement the system as a SPL. The products resulting from the SPL might be as small as a simple way to keep track of executed orders, or as large as a complete sales management system, in which everything from the first idea of a sales opportunity to the delivery and payment of the sold product can be managed.

The processes in the *Sales Scenario* are modeled using model-driven techniques. Thus, we can identify variation in the processes on the model level. For example, Fig. 1 depicts a short version of the order management process, which consists of at least three steps: "receive" (order is received from a customer), "pack" (order items are prepared for delivery), and "ship" (order is sent to the customer). The lower part of Fig. 1 models an example of late variability in the order management process. The modeling elements with solid lines describe the static part of the process. The receive step is subject to a dynamic variation that is described using modeling elements with dashed lines.

As an example for late variability, we model an additional approval step after an order is received. We have identified two subcategories of approval, namely a manual approval, which requires human interaction with the system, or an automated approval, e.g., a check if the order is issued by credit-worthy customers. To provide flexibility we allow customers of the product line to adapt their software system from one process model to another without the need to redeploy the whole system. Therefore, these two variants of approval processes are modeled as *dynamic features*.

A conceptual problem arises when multiple *dynamic features* are composed, e.g., when a customer activates both of the above features at once, as indicated in the last row of Fig. 1. When composing the features, it is necessary to take into account their semantics. One solution would be that the automated approval manages all orders and that we require human intervention only in cases where the automation does not approve of an order. However, we cannot describe such compositions with the current technology.

Existing approaches support a form of linear composition for features, by controlling their order through precedences [1, 19, 5]. But, declaring that the automated approval step precedes the manual approval step is not enough, since we want to declare that the automated approval must be executed and that the manual approval must be skipped. The above approaches also provide composition strategies to declare and enforce a static exclusion constraint between features. For example, the manual approval feature always excludes the automatic approval feature in all configurations.

The problem is that static precedences and exclusions are too conservative. If the constraint between features only occurs in a certain (runtime-) context, such strategies are inappropriate. A correct strategy must take into account the application context. For example, a manual approval is selected only for a certain group of orders, e.g., orders with a high quantity, while the rest is approved automatically. Without knowledge of the domain and application semantics, we cannot express such a composition scenario at the level of feature modeling.

3. MODELING DYNAMIC SPLS

Dynamic feature models are an extension of existing 'static' feature model notations, such as [6], and provide a specialized notation to specify the product lines dynamism and runtime constraints. The dynamic constraints allow expressing i) that the activation of one dynamic feature requires that another dynamic feature be active as well. This constraint is termed *implies* in dynamic feature models and allows SPL designers to model requirements on the reconfiguration logic of the DSPL. Furthermore, we model ii) that two features must not be simultaneously active, by constraining them with *excludes*. This constraint allows restricting the running system from activating both features, if an SPL designer deems their combination harmful due to possible interactions. The iii) *precedes* constraint declares that an interaction of features is allowed and states a resolution strategy, which grants one feature precedence over another. The precedence is not exclusive, thus all features are active but at interacting points, the precedence defines an order in which the features are taken into account.

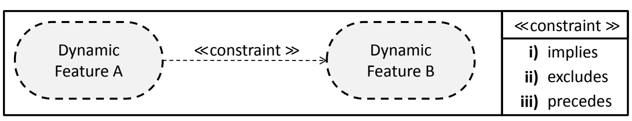

Figure 2: Notation of dynamic features

Fig. 2 depicts the visual representation of dynamic feature models and the possible constraints. Dynamic features are depicted with dashed border lines. Likewise dynamic constraints are modeled using dashed arrows.

We impose some limitations in the usage of our constraints. First of all, the conjunction of *implies* and *excludes* is forbidden, as this combination is not satisfiable by the DSPL. For multiple features with multiple *precedes* constraints we disallow cycles, thereby all involved precedence constraints must form a chain. Furthermore, the combination of *precedes* and *excludes* ii) + iii) is allowed and states that the feature with the highest precedence is the only one taken into account at points where these features interact.

Static constraints may also be formulated between dynamic and static features. For example, a dynmic feature requires the presence of a static feature. The static constraints must be enforced with the same semantics as having static constraints between static features.

4. DSPLS USING DYNAMIC AOP

Our approach realizes a DSPL by mapping dynamic features and their interactions to an *aspect-oriented model* at runtime. The AO model consists of first-class entities, such as *dynamic aspects* and their *pointcuts-and-advice*, *primitive pointcut designators* that match *join points*, and *rule base* with declared constraints on aspect interactions. Dynamic features are mapped to dynamic aspects, which adapt *late variation points* in the DSPL. To enforce the modeled constraints, feature interactions are mapped to constraints on dynamic aspects. The aspect model is used as a first-class runtime representation of the dynamic feature model. The AO model is validated to ensure consistency when features are dynamically activated and deactivated at runtime.

To express dynamic features in terms of dynamic aspects, SPL developers define a DSL. This language, also denoted as *dynamic feature language* (DFL), incorporates the necessary abstractions for the specification of dynamic features in terms of domain concepts. In addition, the DFL provides the required aspect-oriented machinery to declare constraints on dynamic features, as well as semantics for the composition of dynamic features. The DFL enables a safe feature composition, because dynamic feature interactions are automatically detected.

As the underlying language technology for implementing the DFL, we use Popart [7], which is a dynamic aspect-oriented language that provides generic AO mechanisms and that is extensible for new domain-specific syntax and semantics. Domain extensions are integrated into Popart to create a complete SPL-specific aspect language.

The process of defining a DFL consists of the steps: 1) late variation points are identified and modeled as a *domain-specific join point model* (DS-JPM) [8], 2) late VPs are made available for the DFL using a *domain-specific pointcut language* DS-PCL [8] that quantifies over the DS-JPM, 3) the results are integrated with a generic declarative aspect-oriented language, that provides commonly used AO concepts (e.g., before/after/around advice). The resulting *dynamic feature language* is thus summarized as: DFL = DS-JPM + DS-PCL + Generic AO Concepts.

4.1 Modeling Late VPs (DS-JPM)

The first step of the SPL developer is to analyze the design of the SPL for possible *late variation points* and model them in a dynamic DS-JPM. For each late variation point, the developer defines the *context* for this VP, i.e., a properties map that defines identifiers referring to relevant values, e.g., the identifier "customerOrder" refers to the business object in the dynamic context of the running application.

Late VPs can be defined at various levels of abstraction in the SPL and thus identify different artifacts, e.g., model elements or source code points. At the model level, late VPs are modeled as annotations on modeling elements. For example, in a UML activity diagram, one activity is annotated to be dynamically variable. These annotated modeling elements are treated in a special way by code generators. Late VPs require a facility for dynamic de-/activation, thus, the respective source code elements for the model elements are generated, e.g., classes or methods, and the de-/activation is provided by generating domain-specific aspects for these source code elements. At the code level, the late VPs refer to code elements, such as a class, an attribute, a method, or an expression in the body of a method, e.g., when an activity is implemented in a method, the late VP casts on the call to this method.

An excerpt of late VPs that we have identified in the *Sales Scenario* inside the order management workflow is presented in the following:

1) Payment type selection: The customer chooses a specific payment type, e.g., credit card or cash-on-delivery. This step presents various payment types to the customer. Variation at this point can restructure the choice of payment types, e.g., filtering to a more specific list. The context made available at this late variation point are the choice of payment types presented to the customer as well as the customer's concrete choice.

2) Price calculation: The price of an order or a quotation is calculated as the sum of the prices of the contained order items. This is a late variation point, that allows to introduce new pricing strategies and override existing strategies, e.g., to define a discount and allowances on the price. The context is the order for which the price is calculated. From the domain model this implies the exposure of the individual items in the order, since they are accessible via the order.

3) Receive order: The first step in the order management is the reception of new orders. In the *Sales Scenario* orders enter the workflow with all information on the customers and the payment modalities. This step is a late variation point such that we can insert an approval step before packing and shipping the order. Such an additional step then approves whether the customer is trustable before an unpaid cash-on-delivery order is shipped. The available context is the customer, the order, and the selected payment method.

From the identified late variation points the SPL developer builds the DFL for a particular dynamic software product line. To allow the dynamic aspects to intercept the late variation points of the DSPL, the SPL developer declares an instrumentation [1] of the SPL implementation, that reifies SPL concepts. Using this instrumentation we define a *domain-specific join point model*. In this DS-JPM, each late variation point is represented through a SPL-specific *join point type* as a sub-class of `JoinPoint` and its context is a properties map. For the above late variation points, the SPL developer defines join point types: 1) `PaymentTypeSelectionJP`, 2) `PriceCalculationJP`, and 3) `ReceiveOrderJP`.

[1] In the case-study AspectJ aspects are used to reify runtime information.

```
class SalesScenarioDFL extends PointcutDSL {
  ...
  Pointcut receive_order (long quantity) {
    return new ReceiveOrderPredicate(quantity);
} }

class ReceiveOrderPredicate extends PrimitivePCD {
  ReceiveOrderPredicate(long quantity) {...}
  ...
  boolean match(ReceiveOrderJP jp) {
    long orderOuantity = computeQuantity(jp.context.get("order"));
    if ( orderOuantity < quantity ) return true;
    return false;
} }
```

Figure 3: Excerpt of the *Sales Scenario* DFL implementation for selecting an example late VP

The AO instrumentation of the SPL binds context values at late variation points to instances of these join point types. The full technical details of the definition of a DS-JPM are elaborated in [8].

4.2 Quantification over Late Variation Points (DS-PCL)

To allow the dynamic features to quantify over late variation points, the SPL developer declares predicates on the late variations points. In principle, a dynamic feature can select every one of the defined late variation points. All late variation points are made available by a set of predicates, which can be combined using logical expressions (and, or, not). For each late variation point, a predicate is defined that selects this point. For the above late variation points, the following predicates are defined: 1) `payment_type`, 2) `price_calculation`, and 3) `receive_order`.

Further, the predicates can be parameterized, e.g., to constrain late variation points depending on runtime values of the application context. For example, `receive_order(quantity)` defines a parameterized predicate that filters late variation points, where the quantity of the order is less than a certain threshold, e.g., `receive_order(1000)` selects the late variation points of all orders with a quantity of less than 1000 units. Fig. 3 depicts the implementation of the `receive_order(quantity)` predicate, which constitutes a domain-specific keyword in the *Sales Scenario* DFL. Using the POPART framework, the `receive_order(long)` method becomes a keyword in the aspect runtime that can be used to declare where a dynamic feature is active. The concrete matching happens in the `ReceiveOrderPredicate`, where the `match` method of the framework is adopted to match at join points (`ReceiveOrderJP`) that have an order in their runtime context, that contains a quantity lower than the threshold. For every predicate, the *domain-specific pointcut language* (DS-PCL) defines a domain-specific keyword that selects the join points of the corresponding late variation point. Each DS-PCL keyword creates a *primitive pointcut designator* as a sub-class of `Pointcut` in the AO model used to filter `JoinPoint` objects. More details about implementing a DS-PCL are elaborated in [8].

4.3 Generic AO Concepts

To define dynamic features as aspects the following general-purpose AO concepts are provided by POPART and reused in the DFL. The `aspect` keyword defines a new dynamic aspect module, parameters define its name and initial activation status (`deployed`), i.e., active (`true`) or inactive (`false`). POPART allows to define where to insert actions at a late variation point using `before`/`after` advice, which execute before or after reaching the late variation point. In addition, `around` advice replaces the actions of a late variation point and `proceed` invokes the replaced actions in an around advice. To define constraints from the feature model the following mechanics are used: i) `assert` validates a boolean expression when loading the aspect and is used to model dependencies to static features. ii) `declare_dependency`, `declare_exclusion`, and `declare_precedence` are used to declare aspect interaction constraints that are detected and resolved at runtime by POPART.

4.4 DSPL specific AO language (DFL)

To instantiate the DFL, the SPL developer mixes the existing generic AO language with the specific parts for the SPL. POPART take care that all common and specific parts of the AO syntax are integrated together into a SPL-specific aspect language

Using the DFL, each dynamic feature is mapped to a dynamic aspect. The `aspect` is declared with a unique name, that maps to the corresponding feature and the aspect is either active or not depending on the (default) choice of the user. For each specific variation at a late variation point, the aspect defines a pointcut-and-advice. Its pointcut uses the DS-PCL to quantify over join points (i.e., it intercepts the execution of a late variation point) and its advice defines how to adapt selected variation points by inserting or replacing certain actions at the join point (i.e., it adapts a late variation point). We will discuss concrete examples of dynamic aspects in the next section.

Each constraint on dynamic features is implemented as an aspect interaction in one of the aspects. A *dynamic constraint* is defined using one of the `declare`-keywords. For *implies*, an aspect declares `declare_dependency` between the two aspects with the corresponding feature names. For *excludes*, the aspect uses `declare_exclusion`; and for *precedence*, the aspect uses `declare_precedence` instead. Note that for symmetric constraints such as *excludes* it does not matter which aspect actually declares the interaction. Recall that dynamic feature models are an extension to 'static' feature models. A *static constraint* on features can also be defined using the `assert` keyword.

Using our AO model at runtime for the composition of features and the detection and resolution of constraints has several advantages. First, there is no need to consider all combinations of feature selections. When dynamic features are selected or deselected at runtime, the DSPL is automatically adapted by POPART as aspects are composed at join points in the runtime model of the application. Second, the dynamic AO mechanisms allow the declaration of runtime context-dependent feature interactions in conjunction with a continuous enforcement of these constraints by validating possible aspect interaction as specified in the rule base of the AO model. Thus the DFL allows the safe specification of features that interact with each other, because advice are ordered, conflicting advice are never executed at the same time, and dependencies are enforced. We will see example resolutions in the next section.

5. CASE STUDY

To validate our concept, we have implemented the *Sales Scenario* as an example dynamic SPL, parts of which were introduced in Sec. 2. While static variability is modeled and implemented using existing technologies, the late variability is modeled and implemented using the technology that is presented in this paper and that helps to manage late variability. The late variability technology seamlessly integrates with the above technologies in the Eclipse-based workbench. In the remainder of this section, we first summarize the static part of the *Sales Scenario*, then we elaborate how the dynamic features are implemented using the feature DSL.

For the implementation of the static part of the SPLs, we used the Eclipse based facilities for developing SPLs, provided by the feasiPLe research project [9]. The static variability is modeled and implemented using extension of existing methodology, adapted by feasiPLe to better support model-driven and aspect-oriented software development of SPLs.

To give a short overview of the methodology: 1) we designed *domain-specific languages* (DSLs) for the different application domains (e.g., process, business objects, graphical view, etc.) as Ecore[2] metamodels, and instantiated them into *variant independent models* (VIMs). 2) The model elements in these models were then mapped to features using the *FeatureMapper* [15], and 3) using this mapping the VIMs were transformed into *variant specific models* (VSMs) using *pure::variants*[3] For each DSL, we also implemented 4) one code generator in Xpand[4] , and generated Java and AspectJ [1] code based on these VSMs using the code generators. In summary, the *Sales Scenario* has 27 static features and six dynamic features. The implementation consists of 4,000 Java hand-written lines of code (LOC), 17.000 LOC generated Java, 10.000 LOC related to oAW artifacts, 6,000 LOC AspectJ, and 130 LOC Groovy/POPART.

In Fig. 4 the dynamic feature model of the *Sales Scenario* is presented, of which we will discuss first the dynamic approval feature, and then the dynamic pricing strategy feature. The purpose of the dynamic approval feature is to validate customer creditability to reduce risk for large quantity orders. An implementation of the dynamic approval feature from the *Sales Scenario* is shown in Fig. 5. In lines 1–8, the class `OrderManager` (realizing the `Customer Order Mgmnt` feature) is shown that is part of the static part of the SPL. It implements one method for each step in the order management use case, i.e. `receive`, `pack`, and `ship`. The execution of the method `receive` (lines 3–5) constitutes a late VP as modeled in the previous section. In Fig. 3, the class `ReceiveOrderJP` represents executions of the `receive` method and is used to declare the `receive_order` predicate. This predicate is used in lines 10 and 15 (parameter `quantity` is optional), to specify where the different approval steps are inserted.

For the dynamic features, the three aspects in the example are deployed to the running system. The `ManualApproval` aspect defines a pointcut that selects the late VP of the `receive` step by using the corresponding predicate `receive_order` defined in Sec. 4. The advice extends the SPL at the selected variation point by opening a dialog (line 11)

[2] http://www.eclipse.org/modeling/emf/

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

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

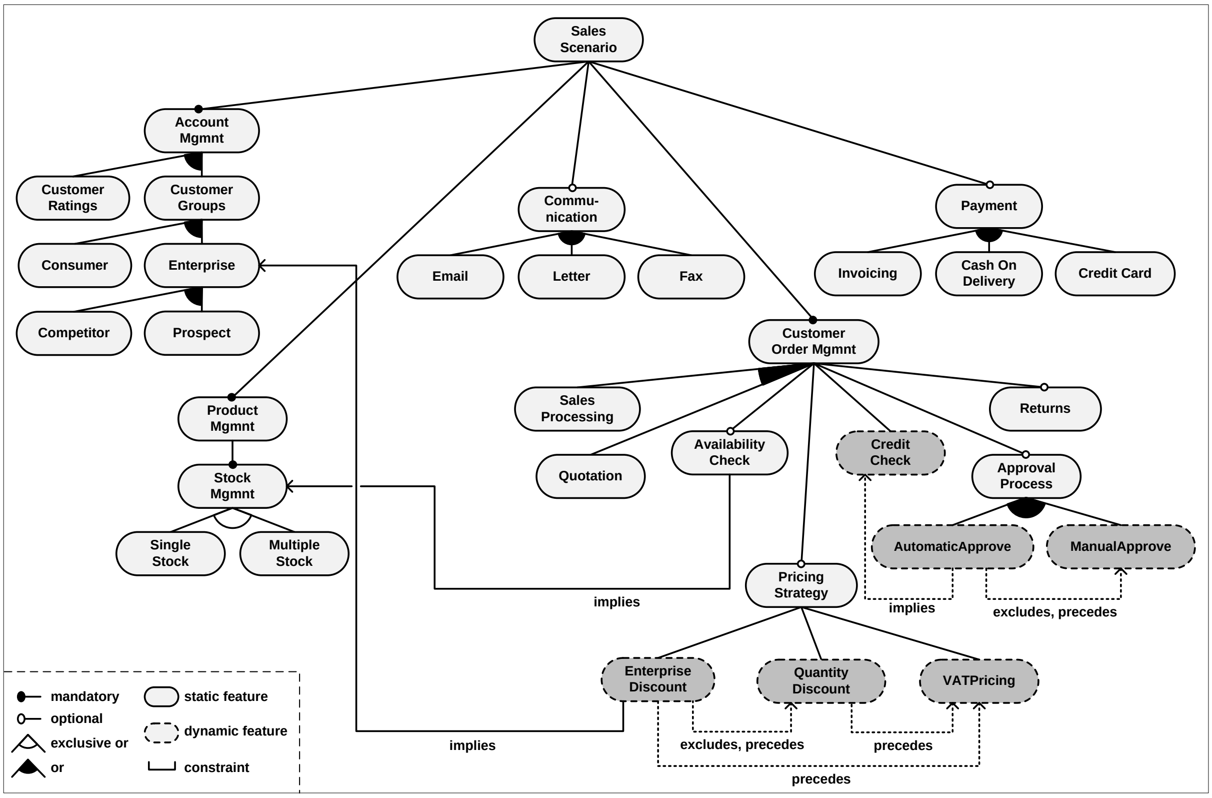

Figure 4: Dynamic feature model of the *Sales Scenario*; extending notation from [6] with dynamic features

```
1  class OrderManager {
2    ...
3    void receive (Order order) {
4      //receive the order from customer (is a dynamic VP)
5    }
6    void pack(Order order) {...}
7    void send(Order order) {...}
8  }
9  aspect(name:"ManualApproval", deployed:true) {
10   after( receive_order () ) {
11     boolean trustable = UI.openApprovalDlg(order).isCustomerTrustable();
12     if (! trustable ) // clarify customers creditibility
13 } }
14 aspect(name:"AutomaticApproval", deployed: false ) {
15   after( receive_order (100) ) { ... } }
16 aspect(name:"CreditCheck") {...}
17 aspect(name:"ApprovalInteraction ", deployed: true) {
18   declare_exclusion "ManualApproval", "AutomaticApproval";
19   declare_precedence "AutomaticApproval", "ManualApproval";
20   declare_dependency "AutomaticApproval", "CreditCheck";
21 }
```

Figure 5: Dynamic feature: order approval

for the manual approval in a user interface. If the user is not trustable, a sub-workflow is invoked (line 12) to clarify the status of the customer, e.g., the customer presents further credentials, pays the order before shipment, or the process is aborted. The aspect `AutomaticApproval` (line 15) is implemented similarly to the manual approval feature, but only executes at late VPs where the quantity of the received order is smaller than 100. As an automation, customer creditability is checked via the corresponding credit card. To illustrate the SPL dynamicity, the `AutomaticApproval` aspect is not deployed (i.e. inactive) during startup. At a later point, i.e. when the company exceeds a certain amount of placed orders, the automated approval is deployed.

The feature interaction between the two approval features is mapped to an aspect constraint, which is delcared in a separate aspect (`ApprovalInteraction`). Line 18 declares the aspects `ManualApproval` and `AutomaticApproval` to be mutually exclusive, i.e., they may not affect variation points at the same time. The following line declares the precedence constraint between the two features. We choose to declare the interaction in a separate aspect, because this has the advantage that the implementation of the two aspects is independent from each other. When the `AutomaticApproval` aspect is deployed, the interaction at the late VP is detected and resolved according to the constraints. Because of the dynamic exclusion constraint in line 18, a conflict is detected that is resolved by taking into account the dynamic precedence constraint in line 19. As the `AutomaticApproval` has a higher precedence, its advice will be executed and the advice of `ManualApproval` are skipped. Because of the dynamic dependency constraint in line 20, the advice of `AutomaticApproval` requires the `CreditCheck` feature to be deployed.

For the *Sales Scenario*, we have implemented a flexible pricing strategy feature using our late variability support, to introduce new pricing strategies as dynamic features into a running product line. The static part of our SPL comes with a simple pricing strategy that calculates the price of an order by calculating the sum of the price of its order items. However, in the context of discounts, allowances and taxes, the actual price of an order depends on various requirements

```
1  aspect(name:"VATPricing") {
2    final float VAT_FACTOR = 1.19; //Currently the German VAT is 19%
3    around ( pricing ()) {
4      return proceed() * VAT_FACTOR;
5  } }
6  aspect(name:"EnterpriseDiscount") {
7    assert Class.forName("EnterpriseCustomer") != null;
8    ...
9  }
10 aspect(name:"QuantityDiscount") {...}
11 aspect(name:" PricingInteraction ") {
12   declare_precedence "QuantityDiscount", "VATPricing";
13   declare_precedence "EnterpriseDiscount", "VATPricing";
14   declare_exclusion "EnterpriseDiscount", "QuantityDiscount";
15   declare_precedence "EnterpriseDiscount", "QuantityDiscount";
16 }
```

Figure 6: Dynamic feature: pricing strategy

from the business domain, e.g., there are business rules that add the value added tax (VAT) to the order price, depending on the customers country or rules that give various discounts to certain customers. In the context of discounts, the interaction of features is again of high interest. Depending on the actual context, two discounts are applicable to one order at the same time, or only one discount is allowed to be applied.

A late VP has been inserted into the price calculation of orders that exposes the necessary context, such as the `order`, its `items`, and the `customer`. We use 3 aspects shown in Fig. 6 for realizing the dynamic pricing features: 1) `VATPricing`: calculating the VAT, 2) `EnterpriseDiscount`: giving a discount to enterprise customers[5], and 3) `QuantityDiscount`: a special discount is applied when the order contains a large quantity of items. Note that all aspects advise the same late variation point through the `pricing` predicate.

In this scenario the dynamic feature interactions between the pricing strategies must be handled by appropriate aspect constraints (Fig. 6), as declared in the `PricingInteraction` aspect. The tax calculation is performed after all other calculations have been applied, consequently the `VATPrice` aspect is declared to have the lowest precedence, using the aspect precedences in lines 12 and 13. The `EnterpriseDiscount` feature requires that the static feature *Enterprise* has been selected for the product. To check the presence, an assertion is used to check whether the class `EnterpriseCustomer`, corresponding to the *Enterprise* feature, is available in the product. This static assertion is checked during startup of the application.

The `EnterpriseDiscount` and the `QuantityDiscount` exclude each other (line 14), since for these particular discounts in the product line we choose to disallow double discounts. Such a situation arises only if an order contains a significant quantity of items and is ordered by an `EnterpriseCustomer`. Because of the exclusion constraint in line 14, the interaction of `QuantityDiscount` and `EnterpriseDiscount` is detected as a conflict. Because in line 15 the `EnterpriseDiscount` is declared to have a higher precedence than `QuantityDiscount`, POPART can resolve this conflict by not excluding the effects of the dynamic feature with a lower precedence, i.e. `QuantityDiscount`, in the composition.

[5]The order business object (BO) refers to the corresponding customer BO, which is typed as a representative of a company or a private customer. We use the type to decide if the discount should be applied.

6. EVALUATION

There are certain limitations in the current implementation that prevent our technology to be used in production.

1) In a real-world business scenario, new business rules would need to be defined and loaded to the *Sales Scenario* during its lifetime. In the case study, we provide support only for de-/activation of features via a management console. For convenient runtime evolution a special management console is required, through which new dynamic feature can be uploaded into a running system. POPART comes with the necessary support, since it allows to deploy aspect definitions provided as a String, due to its roots in Groovy.

2) Our approach does detect feature interactions if the interacting aspects affect the same join point, but omits certain indirect interactions, e.g., two aspect accessing shared state. Such interactions are also possible using our approach, i.e. through the contexts available to aspects. Using these interactions in a structured way can prove advantageous. For example in the *Sales Scenario* this allows us to define a manual approval, that checks in the context of the join point, that the order was not automatically approved and only in this case asks the user with a feedback. However, such interactions are currently not detected by POPART and are not modeled at the level of the feature model. How to capture such feature interactions in a structured way at the modeling level is an interesting research question.

3) When deploying new dynamic features at runtime, the integrity of internal state of adapted use cases is not ensured by POPART. Adapting a running use case that has an internal state is difficult, as for example previously started stack-frames may be omitted from the aspects execution leading to erroneous internal state. In our case study, we did not experience such problems because deploying aspects was only allowed after all running instances of a business process, e.g. the order management use case, were completed.

To evaluate our approach w.r.t. performance scalability, we have determined the relative instrumentation overhead incurred by the AO runtime. We executed our *Sales Scenario* case study with and without our AO runtime, i.e., in POPART and in Java. To measure the bare instrumentation overhead, we do not apply any dynamic aspects at the declared late VPs. Thus the basic AO instrumentation is in place and delegates to our matching algorithm, which does almost nothing, i.e. iterating over an empty list of potential dynamic aspects. We measured the relative overhead incurred by the instrumentation for repeating execution of the variation point and found the approach to scale well with the number of late VPs. When the VP is executed only once there is a large overhead of 97%, but when the late VPs is visited more often, the overhead is reduced to a value as low as 0.8% (1000 executions). This is due to a *dynamic adaptive optimization* applied by the Java virtual machine, which identifies frequently called methods at runtime and performs more advanced optimizations such as inlining.

7. RELATED WORK

Most AOP tools only support aspect precedences similar to AspectJ [1]. Several AOP tools allow expressing aspect dependencies (such as [19, 7]) but there is little work on context-dependent interactions [14, 17, 7].

Context-oriented programming [5] supports modularizing features into layers of functionality that can be activated and deactivated at runtime. This work supports only static dependencies between interacting features.

Research on dynamic product-lines [12] [13] is particularly relevant. In [11], DSPLs are specified as a set of components that can be exchanged at runtime. The components follows the design of *software reconfiguration patterns* and have a set of state charts that define all valid reconfiguration cases. In contrast to our approach, components have to implement patterns and interfaces, features with a crosscutting character are not modularized, and runtime evolution is disallowed because all possible reconfigurations must be known and enumerated into the state chart models at design time.

Cetina et al. [3] discuss possible architectures of dynamic software product-lines and distinguish connected and disconnected architectures for DSPLs, depending on whether the DSPL or the product is responsible for reconfiguration. They propose to follow a hybrid approach that combines the best of both, our approach can be used to implement such a hybrid approach, because every product is delivered with a runtime model of the DSPL. Our approach complements their discussion by prosing a concrete realization.

Trinidad et al. [21] propose the realization of DSPLs through a mapping of features onto components in a component model. Their component architecture introduces the specialized concepts of feature component that can be de-/activated and feature relationship that can be un-/linked. However, the approach does not consider crosscutting features, it enforces only static constraints on features, and it does not allow to consider runtime context.

8. CONCLUSION

We have proposed a novel approach for dynamic software product-lines that uses a dynamic feature model to describe the variability in the DSPLs. The approach combines several trends in aspect-oriented programming for DSPLs, namely dynamic aspects, runtime models of aspects, as well as detection and resolution of aspect interactions. We have implemented and validated the approach and preliminarily evaluation results show its scalability.

Although, current support for managing aspect interactions is weak in existing dynamic AOP tools, we strongly believe that dynamic AOP solutions in general can be used for dynamic product-lines. The biggest challenges for dynamic AOP for DSPLs are a) addressing the limitations found when building (static) SPLs that are also present in dynamic AOP, b) improving the support to handle aspect interactions in particular context-dependent interactions, and c) scalability requirements, such as performance in case of large DSPLs.

Future work will address the current limitations. In particular, we would like to provide better means to scope feature constraints and wildcards in constraint expressions, e.g., to specify that a constraint must be enforced a global application scope, for all sub-features of a certain feature, and for all features that names starts with a certain prefix.

9. REFERENCES

[1] AspectJ Home Page. `http://www.eclipse.org/aspectj/`.

[2] CaesarJ Homepage. `http://caesarj.org/`.

[3] C. Cetina, V. Pelechano, P. Trinidad, and A. Cortes. An Architectural Discussion on DSPL. In *Software Product Line Conference*, pages 59–68, 2008.

[4] A. Charfi and M. Mezini. Hybrid Web Service Composition: Business Processes meet Business Rules. In *International Conference on Service Oriented Computing*, pages 30–38, 2004.

[5] P. Costanza and T. D'Hondt. Feature Descriptions for Context-oriented Programming. In *Workshop on Dynamic Software Product Lines [12]*, 2008.

[6] K. Czarnecki and A. Wasowski. Feature diagrams and logics: There and back again. In *SPLC '07: Proceedings of the 11th International Software Product Line Conference (SPLC 2007)*, pages 23–34, Washington, DC, USA, 2007. IEEE Computer Society.

[7] T. Dinkelaker, M. Mezini, and C. Bockisch. The Art of the Meta-Aspect Protocol. In *AOSD*, 2009.

[8] T. Dinkelaker, M. Monperrus, and M. Mezini. Untangling crosscutting concerns in domain-specific languages with domain-specific join points. In *Workshop on Domain-specific Aspect languages (at AOSD)*, 2009.

[9] The feasiPLe Homepage. `http://feasiple.de/`.

[10] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers. Examining the challenges of scientific workflows. *Computer*, pages 24–32, 2007.

[11] H. Gomaa and M. Hussein. Dynamic software reconfiguration in software product families. *LNCS*, pages 435–444, 2004.

[12] S. Hallsteinsen and et al. International Workshop on Dynamic Software Product Lines (DSPL). In *International Software Product Line Conference*, 2007-2009.

[13] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic software product lines. *Computer*, 41(4):93–95, 2008.

[14] J. Hannemann, R. Chitchyan, and A. Rashid. Analysis of aspect-oriented software. *LNCS*, 2004.

[15] F. Heidenreich, J. Kopcsek, and C. Wende. Featuremapper: Mapping features to models. In *ICSE*, 2008.

[16] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP*, 1997.

[17] G. Kniesel. Detection and Resolution of Weaving Interactions. *Transactions on Aspect-Oriented Software Development V*, page 186, 2009.

[18] M. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B. Krämer. Service-oriented computing: A research roadmap. *International Journal of Cooperative Information Systems*, 17(2):223–255, 2008.

[19] É. Tanter. Aspects of composition in the Reflex AOP kernel. *LNCS*, 4089:98–113, 2006.

[20] E. Tanter and J. Noye. A Versatile Kernel for Multi-language AOP. In *GPCE*, 2005.

[21] P. Trinidad, A. Ruiz-Cortés, J. Pena, and D. Benavides. Mapping Feature Models onto Component Models to Build Dynamic Software Product Lines. In *DSPL [12]*, 2007.

[22] J. van Gurp. *Variability in software systems: the key to software reuse.* PhD thesis, Dept. of Software Engineering & Computer Science, Blekinge Institute of Technology, 2000.