Mapping Features to Models: A Template Approach Based on Superimposed Variants

Krzysztof Czarnecki and Michał Antkiewicz

University of Waterloo, Canada

Abstract. Although a feature model can represent commonalities and variabilities in a very concise taxonomic form, features in a feature model are merely symbols. Mapping features to other models, such as behavioral or data specifications, gives them semantics. In this paper, we propose a general template-based approach for mapping feature models to concise representations of variability in different kinds of other models. We show how the approach can be applied to UML 2.0 activity and class models and describe a prototype implementation.

1 Introduction

Feature modeling is an important method and notation to elicit and represent common and variable features of the systems in a product line. It can be used at any level of abstraction, including requirements, architecture and design, components, and platforms; for any kind of artifacts, such as code, models, documentation; and in all stages of product-line engineering. At an early stage, feature modeling enables product-line scoping, i.e., deciding which features should be supported by a product line and which should not. In design, the points and ranges of variation captured in feature models need to be mapped to a common product-line architecture. Furthermore, feature models allow us to scope and derive domain-specific languages, which are used to specify product-line members in generative software development [1, 2, 3]. Finally, feature models are also useful in product development as a basis for estimating development cost and effort, and automated or manual product derivation.

Although a feature model can represent commonalities and variabilities in a very concise taxonomic form, features in a feature model are merely symbols. Mapping features to other models, such as behavioral or data specifications, gives them semantics. In this paper, we propose a general approach for mapping feature models to concise representations of variability in different kinds of other models. In contrast to variability approaches in which separate model fragments corresponding to different features are composed, our approach presents the modeler with a model representing a superimposition of all variants whose elements are related to the corresponding features through annotations. We argue that this approach is particularly desirable at the requirements level, as it directly shows the impact of selecting a given feature on the resulting model. The proposed approach is general; it works for any model whose metamodel is

R. Glück and M. Lowry (Eds.): GPCE 2005, LNCS 3676, pp. 422–437, 2005.

expressed in the Meta-Object Facility (MOF) [4] or a comparable modeling formalism, and it can be easily incorporated into an existing model editor. We give the details of our approach for mapping feature models to UML 2.0 activity diagrams and indicate how it can be applied to other kinds of models. We describe a prototype implementation of our approach. The sample models presented in this paper are taken from a large model of an e-commerce platform, which we used to test our approach.

The remainder of the paper is organized as follows. Section 2 reviews background concepts and related work on feature modeling. Next we describe the idea in sections 3 and 4. Section 5 presents the details of template instantiation algorithm. We describe the implementation of the prototype in section 6. We discuss related work in section 7 and conclude the paper in section 8.

2 Background: Feature Modeling

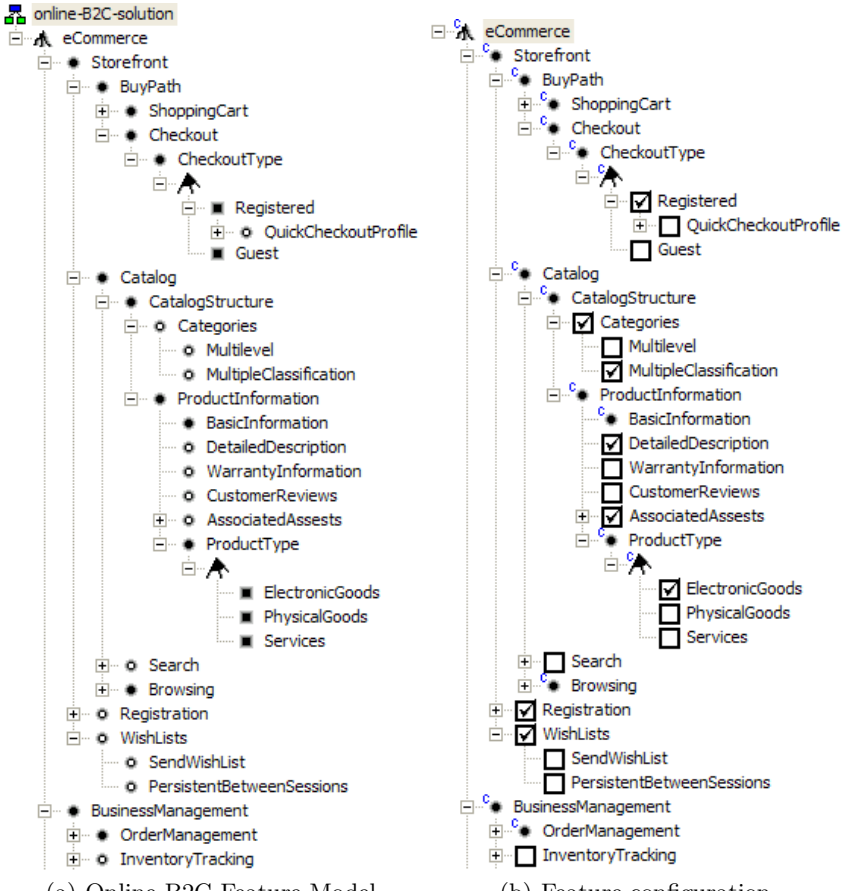
Feature modeling was originally proposed as part of the Feature-Oriented Domain Analysis (FODA) method [5], and since then, it has been applied in a range of business and technical domains (see [6] for list of applications with references). In this work, we use *cardinality-based feature modeling* [7], which extends the original feature modeling from FODA with feature and group cardinalities, feature attributes, feature diagram references, and user-defined annotations.

A *feature* is a system property that is relevant to some stakeholder and is used to capture commonalities or discriminate among systems in a family [1]. Features are organized in *feature diagrams*. A feature diagram is a tree with the root representing a concept (e.g., a software system) and its descendant nodes being features. A *feature model* consists of one or more feature diagrams plus *additional information* such as feature descriptions, global constraints, binding times, priorities, stakeholders, etc.

Figure 1(a) presents a small excerpt from a feature model describing a family of online business-to-consumer (B2C) solutions; the entire model has over 350 features. The model contains one feature diagram, with `eCommerce` as its *root feature*. The root feature has two solitary subfeatures: `Storefront` and `BusinessManagement`. The symbol ● indicates that `Storefront` has a feature cardinality of [1..1]. Feature cardinality is an interval denoting how often a feature with its subfeatures can be cloned as a child of its parent when specifying a concrete system. The cardinality of [1..1] indicates that a feature must exist at least and at most once. On the other hand, the symbol ○ indicates that `WishLists` is an optional feature with cardinality [0..1]. Available checkout types `Registered` and `Guest`, are members of a *feature group*. The group symbol ▲ indicates group cardinality $\langle 1\text{–}k \rangle$, where k is the group size. Thus available checkout types can be any non-empty subset of the two checkout types. *Grouped features* are indicated by the symbol ..■

One can specify additional constraints such as *requires* or *excludes*. For example, the feature `PersistentBetweenSessions` requires the system to implement `Registration` because a wish list is stored in a customer's account. Also, check-

out type `Registered` requires feature `Registration` to be selected. In general, additional constraints in cardinality-based feature models require tree-oriented navigation and query facilities, and may involve logic, arithmetic, string, and set operators on feature attributes and feature sets. Such constraints can be adequately expressed using XPath 2.0 [8].



(a) Online B2C Feature Model (b) Feature configuration

Fig. 1. Sample online B2C feature model and its feature configuration

Semantically, a feature model describes a set of all possible valid *configurations* [7]. Figure 1(b) presents a sample configuration of the online B2C feature model. A configuration specifies a concrete system. In this example, checkout for registered customers is the only available checkout type, the catalog is subdivided into categories, a product can be classified in multiple categories, the catalog contains only electronic goods, etc.

3 Basic Idea: Superimposed Variants

An overview of our approach is shown in Figure 2. A *model family* is represented by a *feature model* and a *model template*. The feature model defines a hierarchy of features together with the constraints on their possible configurations. The model template contains the union of the model elements in all valid *template instances*. The set of the valid template instances corresponds to the extent of the model family. The model template is itself a model expressed in the same *target notation* as the template instances. For example if we want to represent a family of UML activity models, both the model template and the template instances will be expressed using the UML activity modeling notation. The elements of a model template may be annotated using *presence conditions* (PCs) and *meta-expressions* (MEs). These annotations are defined in terms of features and feature attributes from the feature model, and can be evaluated with respect to a feature configuration. A PC attached to a model element indicates whether the element should be present in or removed from a template instance. MEs are used to compute attributes of model elements, such as the name of an element or the return type of an operation.

An instance of a model family can be specified by creating a feature configuration based on the feature model. Based on the feature configuration, the model template is instantiated automatically. The instantiation process is a *model-to-model transformation* with both the input and output expressed in the target notation. It involves evaluating the PCs and MEs with respect to the feature configuration, removing model elements whose PCs evaluate to false and, possibly, additional processing such as simplification (Section 5).

A particularly useful form of PCs are Boolean formulas over the set of variables, where each variable corresponds to a feature from the feature model. Given a feature configuration, the value of a given Boolean variable is true if and only if the corresponding feature is included in the feature configuration.

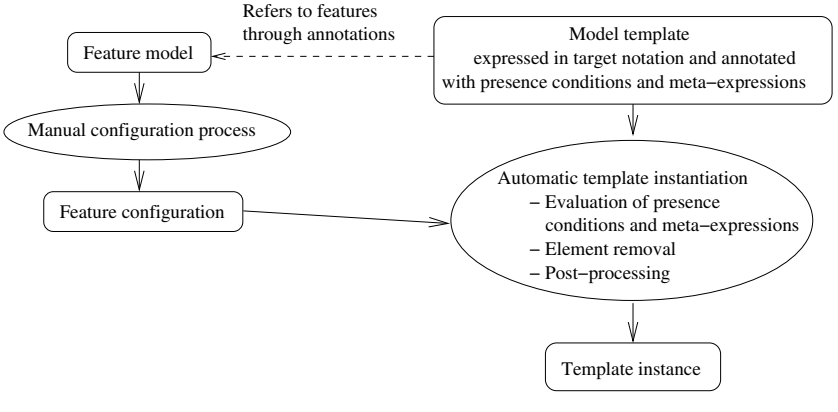
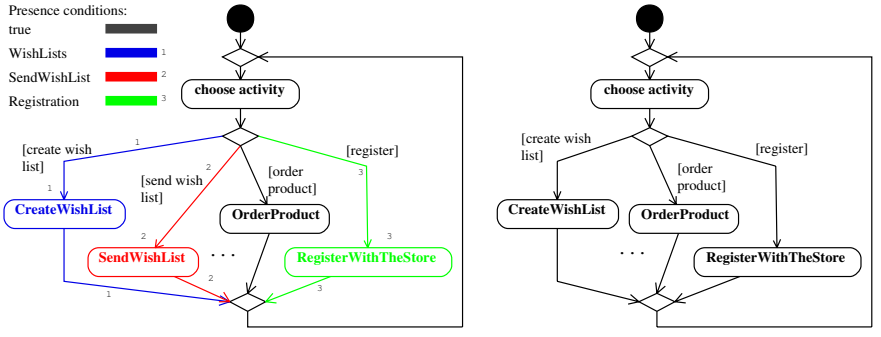


Fig. 2. Overview of the approach



(a) Storefront template (b) Storefront instance

Fig. 3. Sample template activity diagram and its instance

As an example, consider the UML activity diagram in Figure 3(a), which models the top-level activity of a storefront. This diagram is a template since elements relevant to features `WishLists`, `SendWishList` and `Registration` have been annotated with their PCs. The annotations are rendered using a coloring scheme in which each different PC is assigned a different color.[1] In this simple example, each PC consists of a single variable corresponding to a single feature. Figure 3(b) shows a template instance created based on the feature configuration from Figure 1(b). PCs corresponding to feature `SendWishList`, which is not included in the configuration, evaluated to false and the annotated elements were removed from the template.

It is important to note that PCs are interpreted locally with respect to containment hierarchies defined in the metamodel of the target notation. In other words, a PC on an element controls the presence of that element only with respect to its container; if the container is removed, all contained elements are also removed, regardless of their PCs. For that reason, we did not have to annotate the guards on flows in Figure 3(a) because they are contained in the flows according to the UML metamodel.

More complex PCs can be expressed using XPath [9]. Such conditions can access feature attributes, count the number of feature clones in a configuration, and use other XPath operations, as long as the XPath expression evaluates to a Boolean value. If necessary, XPath can be easily extended with user-defined functions.

MEs may be used to compute attributes of basic types, as well as references to model elements. In this paper, we only consider computing references to already existing elements. MEs can be expressed using XPath. As an exam-

[1] The colors are assigned per diagram, and the number of colors needed is limited since diagrams are usually split such that each diagram can fit on the computer screen. Note that colors in this paper are indexed in order for the annotations to be readable in black and white.

ple, consider the activity diagram fragment in Figure 4. The type of the input pin of action `DisplayProducts` is set to `b2cSoln::Category` or `b2cSoln::Catalog` depending on the presence of the `Categories` feature in the feature configuration.

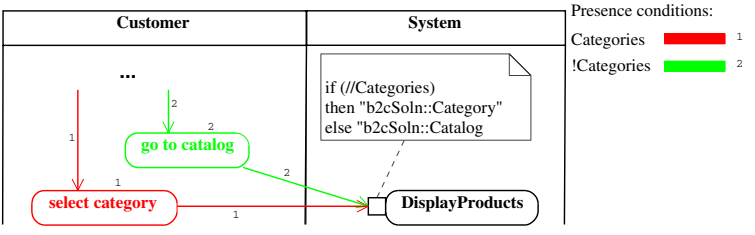


Fig. 4. Example of a type meta-expression

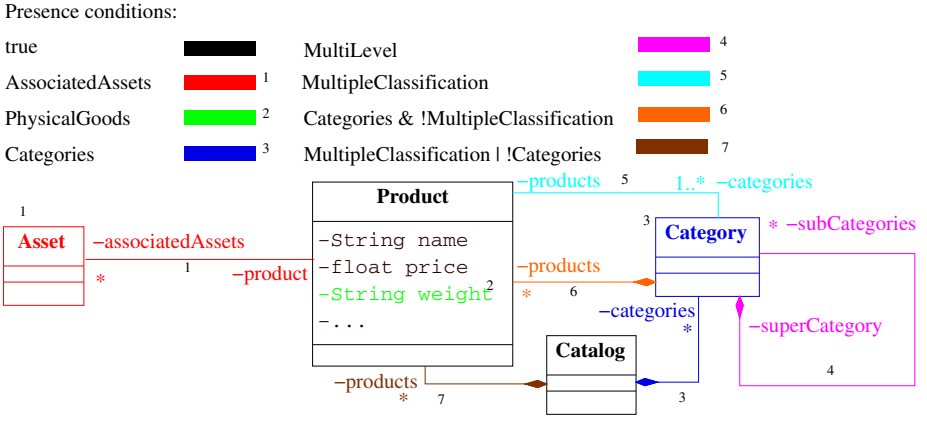


Fig. 5. Example of annotated class diagram

Figure 5 shows an annotated class diagram. Class `Category` is present in a template instance if the feature `Categories` is selected. Feature `MultiLevel` implies a containment hierarchy for `Category`. `MultipleClassification` implies that `Products` can be classified under multiple categories. `AssociatedAssets` implies the class `Asset`, which can be used for storing documents such as technical specifications and manuals and other media. Finally, `PhysicalGoods` implies the attribute `weight` in `Product`.

The realization of our approach for a given target notation involves the following steps:

1. decide on the form of PCs and MEs, for example Boolean formulas and/or XPath expressions;
2. decide on *implicit PCs*. Model elements that are not explicitly annotated by the user will have implicit PCs; implicit PCs will be explained shortly;
3. decide on the annotation mechanism and rendering options for the annotations, e.g., if the target notation is UML, the annotations can be realized as stereotypes; rendering options include labels, icons, and/or coloring;
4. decide on additional processing.

Steps 2–4 depend on the target notation. In the following sections, we will demonstrate details for UML activity diagrams as a target notation. [2]

4 Implicit Presence Conditions

When an element has not been explicitly assigned a PC by the user, an implicit PC (IPC) is assumed. In general, assuming a PC of `true` is a simple choice which is mostly adequate in practice; however, sometimes a more useful IPC for an element of a given type can be provided based on the presence conditions of other elements and the syntax and semantics of the target notation. For example, according to UML syntax, a binary association requires a classifier at each of its ends. Thus, a reasonable choice of IPC for a binary association would be the conjunction of the PCs of both classifiers. This way, removing any of the classifiers will also lead to the removal of the association. IPCs reduce the necessary annotation effort of the user. For example, given the IPC for associations as described, the association between `Product` and `Asset` in Figure 5 does not need to be annotated explicitly.

Table 1 shows our choice of IPCs for UML class and activity model elements. An IPC for a given element is assumed based on its type. In order to determine the IPC for a given model element, we look up the closest matching supertype in Table 1 and take the corresponding IPC. For example, the IPC for instances of `Class` and `Action` is `true` because their closest matching type in Table 1 is `Element`. Since `ActivityFinalNode` and `FlowFinalNode` are subclasses of `FinalNode` according to the UML metamodel, the IPC for `ActivityFinalNode` and `FlowFinalNode` is the same as for `FinalNode` in Table 1.

The choice of IPCs in Table 1 reflects the cardinality and other integrity constraints specified in the UML metamodel. For class models, the IPC for all elements except relationships is `true`. In the case of generalization, which is a binary relationship, the IPC reflects the fact that such a relationship can exist in a template instance only if the classifiers at both ends of the relationship are also present in the template instance. The IPCs for dependencies and associations need to handle the more general case of n-ary relationships. For activity models, the IPC for all elements except control nodes, central buffer node, and call actions is `true`. Control nodes have to have at least one incoming flow and/or at least

[2] For simplicity, we limit ourselves to the intermediate level of activity diagrams as defined in the UML Superstructure document [10].

Table 1. Implicit presence conditions for UML class and activity model elements

Model kind	Element type	Implicit presence condition[a]
General	`Element`	`true`
Class	`Generalization`	Conjunction of the PCs of the `general` and `specific` classifiers
	`Dependency`	`true` iff at least one `client` element's PC evaluates to `true` and at least one `supplier` element's PC evaluates to `true`
	`Association`	`true` iff two or more `memberEnd` properties such that each has a classifier with PCs evaluating to `true` as its type
Activity	`InitialNode`	Disjunction of the PCs of all `outgoing` flows
	`FinalNode`	Disjunction of the PCs of all incoming flows
	`DecisionNode` and `ForkNode`	`true` iff exactly one `incoming` flow's PC evaluates to `true` and one or more `outgoing` flows' PCs evaluate to `true`
	`MergeNode` and `JoinNode`	`true` iff exactly one `outgoing` flow's PC evaluates to `true` and one or more `incoming` flows' PCs evaluate to `true`
	`CentralBufferNode`	Disjunction of the PCs of all `incoming` and `outgoing` flows
	`CallOperationAction`	true iff accumulated PC of the called `operation` evaluates to true
	`CallBehaviorAction`	true iff accumulated PC of the called `behavior` evaluates to true

[a] PC stands for presence condition (both explicit or implicit). Names in typewriter font (except `true`) refer to properties of the corresponding element.

one outgoing flow in an instance as specified in the metamodel. Central buffer has to have at least one incoming flow or outgoing flow. Finally, the target of call actions has to be present.

Control nodes are not intended to be annotated with PCs explicitly since their IPCs will always be adequate. This is not true for relationships in class models because we might want to remove a relationship in a template instance even if the elements the relationship connects are not removed.

IPCs for call actions relfect the fact that removal of the target should also force removal of all actions calling it. Accumulated PC of an element is true iff PCs of all parents of that element evaluate to true.

5 Template Instantiation

A simple and general template instantiation process involves computing MEs and removing elements whose PCs are false; however, the general process can be specialized for a given notation with some additional processing steps, which allow expressing templates in that notation more compactly. We have identified two categories of such additional steps: *patch application* and *simplification*. A patch is a transformation that automatically fixes a problem which may result from removing elements. It is defined for situations in which there exists a unique and intuitive solution to a problem created by element removal.

Simplification involves removing elements that have become redundant after removing other elements. In the case of activity models, we found it useful to provide *automatic flow closure* as a patch and *removal of redundant control nodes* for simplification, which will be explained later.

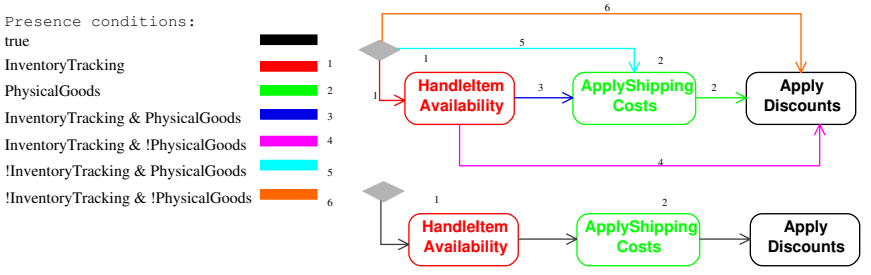


Fig. 6. Model templates with two optional actions without and with automatic flow closure

The motivating example for automatic flow closure is presented in Figure 6. The two actions *HandleItemAvailability* and *ApplyShippingCosts* are optional and implement features `InventoryTracking` and `PhysicalGoods`, respectively. The top part of the figure presents how the two optional actions would have to be modeled without automatic flow closure. The bottom part contains the same fragment expressed in a natural way thanks to the automatic flow closure. The latter ensures that after removing an optional action the still remaining incoming flow and outgoing flow will be closed. If desired, the closure can be prevented by the user by annotating the flows such that they are removed together with the action. It is easy to see that, without flow closure, the number of flows needed in a chain of optional actions grows exponentially with the number of the actions.[3]

The complete template instantiation algorithm can be summarized as follows:

1. *Evaluation of MEs and explicit PCs.* The evaluation is done while traversing the element containment hierarchy in the template in depth-first order. Children of elements whose PCs evaluate to false are not visited because they will be removed.
2. *Removal analysis.* Removal analysis involves computing IPCs and information required for patch application, if any. The IPCs in Table 1 can be computed in a single additional pass after computing the explicit PCs; however, a different choice of IPCs could require multiple iterations. Furthermore, given the IPCs in Table 1, the necessary analysis for automatic flow closure can be performed separately after the IPCs are computed. Again, depending on the choice of IPCs and patches, such separation may not be possible.
3. *Element removal and patch application.* In this step, elements whose PCs are false are removed and patches, if any, are applied. Application of a patch depends on its type and can be performed before or after removal.
4. *Simplification.* Simplification is performed at last.

[3] It is interesting to note that removing an optional action from a sequence of actions in an activity model corresponds to removing an optional statement from a statement list in a textual language, e.g., when the C preprocessor removes a statement within `#ifdef` and `#endif` in a C program. In the latter case, however, flow closure happens naturally without the need for any additional processing.

5.1 Template Instantiation for Activity Diagrams

The removal analysis for activity diagrams identifies situations where flows interrupted by removed elements can be closed. The identification is performed during removal analysis after all IPCs have been computed and proceeds as follows. Let F be the set of elements contained in an activity whose PCs (both explicit and implicit) evaluated to false. We partition F into a set of regions R such that elements in each region are connected, but no two elements from two different regions are connected. Furthermore, let A_r be a set of flows *adjacent* to the region r.

A region r is said to be *closable* iff

1. there is exactly one incoming and exactly one outgoing adjacent flow,[4] i.e., $A_r = \{i, o\} \wedge target(i) \in r \wedge source(o) \in r$
2. there is a flow path connecting $target(i)$ and $source(o)$
3. types of i and o are consistent i.e., both are control or object flows

All closeable regions are closed before elements with PCs being false are removed. Closing a region r with A_r means that o is removed and the target of i is set to the target of o. If a region is not closeable and A_r is not empty, there is an annotation error because flows from A_r would become dangling after the removal of region r. An annotation error can also occur if a flow is not within the region itself, but its ends are.

Simplification for activity nodes involves removing redundant control nodes such as (1) a `DecisionNode` or a `ForkNode` having one outgoing flow, and (2) a `MergeNode` or a `JoinNode` having one incoming flow. More sophisticated control flow simplification could also be applied at this point, such as merging parallel flows without actions between a decision and merge nodes.

As an example of template instantiation for activity models with automatic flow closure, consider the checkout items template in Figure 7 and its instance in Figure 8(b). The instance implements the features selected in the feature configuration from Figure 1(b), where the only specified checkout method is for registered customers (feature `Registered`). Features `Guest`, `QuickCheckoutProfile`, `InventoryTracking` and `PhysicalGoods` are not selected. Note that all control nodes in the template are gray, indicating that they are not annotated and therefore IPCs are assumed.

The result of removal analysis and patch application is shown in Figure 8(a). Based on our configuration, the instantiation algorithm removes four regions: blue (3) for `QuickCheckoutProfile`, green (2) for `Guest` checkout, magenta (4) for `InventoryTracking`, and pink (5) for `PhysicalGoods`. Note that they are all closeable and will be closed before removal. In the case of the blue (3) region, the adjacent flows are those in red (1). Also note that the decision node `type?` has been included in the blue region because its implicit PC evaluated to false.

[4] We only allow one incoming and one outgoing flow. The reason is that in the case of more than one incoming and/or outgoing flow, there is more than one way to close the flows.

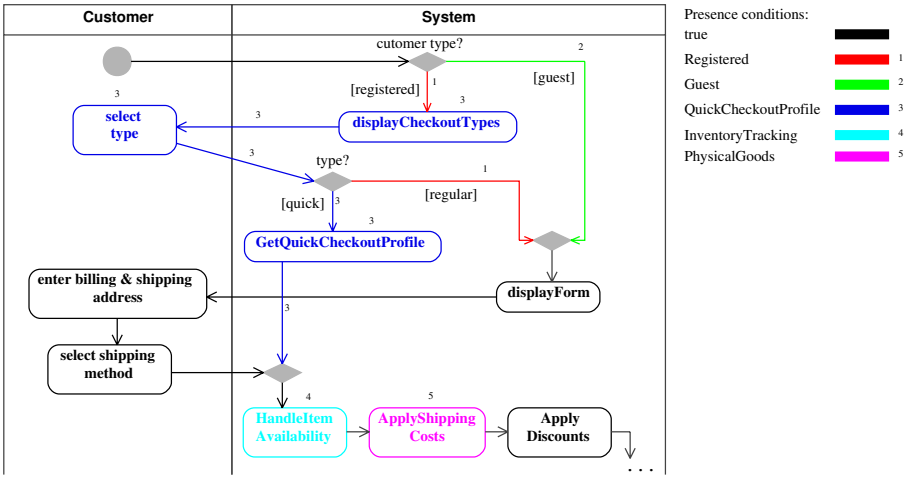
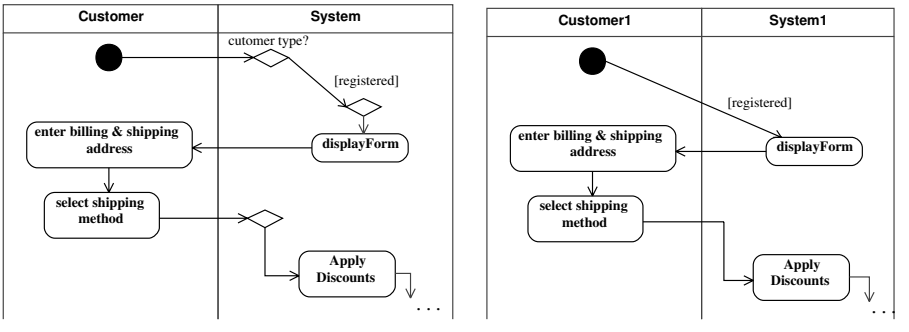


Fig. 7. Checkout Items diagram template



(a) Checkout instance after region removal

(b) Checkout instance after simplification

Fig. 8. Checkout template instance

The final result after simplification, which removed one decision node `customer type?` and two merge nodes, is shown in Figure 8(b).

Examples of useful patches for class models include *generalization chain closure* and *containment chain closure*. They are the counterpart of automatic flow closure for generalization and containment relationships: given the classifiers `A`, `B`, and `C`, if `A` is connected to `B` and `B` is connected to `C`, removing `B` while the PCs of the incoming relationship and the outgoing relationship are true will connect `A` to `C`.

6 Prototype Implementation

We have built a prototype to illustrate how our approach works in practice. The prototype, *fmp2rsm*, is an Eclipse plug-in which integrates our Feature Modeling Plug-In (fmp) [8] with Rational Software Modeler (RSM), a UML modeling tool from IBM.[5] The plug-in implements the template instantiation algorithm from Section 5, and it also performs the automatic coloring of templates based on the PC annotations. The implementation handles all of UML; however, at this point, the convenience of additional processing is available only for activity models as described in Section 5.1.

The plug-in works with four artifacts: (1) UML model template created using RSM, (2) feature model created using fmp (Figure 1 contains screen shots of fmp), and two *variability profiles*, (3) *PC profile* for PC annotations, and (4) *ME profile* for ME annotations.

The PC profile offers two forms of PC annotations: Boolean formulas in Disjunctive Normal Form (DNF) and the more general XPath expressions. Each disjunct (i.e., a conjunction of literals) of a PC in DNF is represented as a stereotype, e.g., `<<f1∧!f2∧f3>>` for the Boolean formula $f1\overline{f2}f3$, and can be created on a selection of multiple features automatically through a menu operation in fmp.[6] Once created, the stereotype becomes available in RSM for annotating template elements. Application of multiple such stereotypes is interpreted as a disjunction. The more general annotations using XPath are created by applying the stereotype `<<PC>>`, which allows the user to enter the desired XPath expression as the value of its `expression:String` property.

The ME profile contains several stereotypes structured similarly to `<<PC>>`, but each applicable to elements of a specific type. For example, `<<NameME>>` can be applied to any element of type `NamedElement` in order to compute its name. Similarly, `<<TypeME>>` can be used to set the `type` property of any `TypedElement`. For `<<TypeME>>`, an expression has to return fully qualified name of either a primitive type (e.g., `UML2::String`), a class (e.g., `b2cSoln::Category` as in Figure 4), an interface, or an enumeration. The ME profile could be automatically generated based on the metamodel of a given notation.

7 Related Work

Variability mechanisms most commonly used in models are those already available in the target notation, such as using a decision node in an activity model to decide between alternative flows or representing class variants as subclasses of an

[5] The fmp2rsm plug-in can be downloaded at `http://gp.uwaterloo.ca/fmp2rsm`.

[6] Each stereotype extends `Element` and has read-only properties encoding the actual Boolean formula. The encoding uses fully qualified feature names which are available as literals of an enumeration type that is automatically generated by fmp2rsm given a feature model. In other words, the stereotype's name is just for documentation and may use abbreviated feature names.

abstract class. Inheritance—a classical variability mechanism in class diagrams—has also been adapted for activities [11] and statecharts [12, 13]. Limitations of these approaches include the lack of static configuration, as in the case of dynamic choice such as a decision node, the potential of combinatorial explosion for static inheritance hierarchies, and complexity increase and limited traceability in the case of design patterns.

Another class of approaches is based on annotations expressing variability. In the case of UML, such annotations are usually provided as a profile with stereotypes, such as `<<optional>>` and `<<variant>>`, e.g., [14]. Although our approach is also annotation-based, we provide a separate representation of variability in the form of a feature model. Without the latter, there is no clear notion of features and the user has to find and select variable elements in the model directly. Furthermore, patching and simplification, as proposed in our approach, results in simpler templates. Finally, we provide full template support by means of MEs.

Wasowski describes automatic generation of variants of behavioral models (in particular statecharts) by restrictions [15]. As in our approach, the modeler creates a single model containing all variants. A variant is automatically created using a form of partial evaluation and slicing based on specified restrictions. The restriction approach differs from our template approach in several ways. First, it involves more sophisticated analysis in which restrictions on inputs and outputs are propagated throughout the model automatically. While this may result in a significantly reduced annotation effort, the effect of automatic partial evaluation and slicing may be hard to predict for the user. In a template approach, the user has full control through explicit annotations. Another difference is that the model to be restricted has to be semantically correct in the sense that it is ready to be executed without any processing, whereas templates only need to be syntactically well-formed. The restriction approach is adequate, particularly if there is a variant that contains all of the initial model without any restrictions; however, if this is not the case, it is likely that a template will be simpler than an unrestricted model. For example, alternative flows can simply be attached to an activity node, while the model restriction approach would require using a decision node. Also, additional processing, such as automatic flow closure, further reduces the complexity of a template, whereas in the model restriction approach, each optional action would need an extra decision node. Finally, the template can be easily adapted to any notation. This is different with the restriction approach, which is semantics-based and needs to be individually developed for each notation.

Another group of work are concern separation approaches, such as AHEAD [16], and the hyperspace approach [17] and its UML realization HyperUML [18]. These approaches allow the composition of crosscutting model fragments. In particular, HyperUML uses feature models to represent the composition space of UML model fragments. Mixin-based composition of statecharts [19, 20] also falls into this category, but with the particular focus on ensuring provably correct composition. Concern separation approaches focus on separation. Templates, on the other hand, work best if the user wants to see the model fragment correspond-

ing to a feature embedded in the context of the entire model. For example, separating the blue (3) region corresponding to the feature `QuickCheckoutProfile` in Figure 8 as a component does not seem to be interesting. This is because the fragment is not reusable and can be best understood in the context where it is applied. Separation approaches are of particular interest when features should be realized as components that can be composed in many ways, which is the case in mature and highly flexible architectures. They are also preferred for representing crosscutting concerns that can be meaningfully stated in separation, such as logging or security. For example, the fact that the checkout activity requires authorization can be expressed as a security annotation (a kind of join point), leaving the insertion of call actions to the approprite authorization and authentication activities to an aspect weaver.

A few modeling tools on the market support model templates in some form, but usually in an ad hoc manner. For example, templates in Rational XDE are modeled as parameterized collaborations, even though the template can contain meta-code creating arbitrary models, i.e., the template instance is not a collaboration. Obviously, variability can also be realized by direct model manipulation, such as using composition directives [21] or XMI manipulation [22].

Finally, feature models have been previously used together with textual templates as the structure definition of template input, e.g., [23]. However, the application to model templates is, to our knowledge, new.

8 Concluding Remarks

The purpose of this work can be seen from different perspectives: (1) giving semantics to features in feature models by mapping them to other models and (2) using feature models to provide a concise representation of variability contained in other models. Expressing PCs and MEs in terms of features provides traceability between features and their realization in models.

Although we think our approach is particularly useful at the requirements level, it can be applied for models at any level, e.g., architecture and implementation models.

From the usability perspective, the approach is intuitive. Model templates are in the target notation, so there is no need to learn new specialized languages (except for simple feature models) and existing tools can also be reused. Implicit conditions, patching, and simplification minimize annotation effort and decrease visual complexity, which makes model templates more concise. Coloring makes it easy to see what will be contributed by selecting a given feature. Model templates can be created incrementally and simultaneously with the feature model.

During our case study we have observed that the majority of PCs are single features. However the ability to write more complex PCs allows us to avoid polluting the feature model with features related to the implementation details of the template. For example, a “glue” element usually requires a PC being a conjunction of features. If a PC could only be simple features, an additional feature corresponding to the “glue” element would need to be introduced.

A possible concern is that annotation is not always simple and may require few iterations; however, further tool support can be offered, e.g., for filtering model template parts relevant to certain features and subset of systems, and automatic verification guaranteeing the well-formedness of all possible template instances. Those additional capabilities, as well as support for element cloning in model templates, will be covered in future work.

References

1. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Boston, MA (2000)
2. Czarnecki, K.: Overview of Generative Software Development. In: Proceedings of the European Commission and US National Science Foundation Strategic Research Workshop on Unconventional Programming Paradigms, September, 15–17, 2004, Mont Saint-Michel, France. (2004) http://www.swen.uwaterloo.ca/kczarnec/gsdoverview.pdf.
3. Batory, D.: Feature Models, Grammars, and Propositional Formulas. Technical Report TR-05-14, University of Texas at Austin, Texas (2005)
4. Object Management Group: Meta-Object Facility. (2002) http://www.omg.org/technology/documents/formal/mof.htm.
5. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1990)
6. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration through specialization and multi-level configuration of feature models. Software Process Improvement and Practice **10** (2005) 143–169 http://swen.uwaterloo.ca/~kczarnec/spip05b.pdf.
7. Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization. Software Process Improvement and Practice **10** (2005) 7–29
8. Antkiewicz, M., Czarnecki, K.: FeaturePlugin: Feature modeling plug-in for Eclipse. In: OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop. (2004) Paper available from http://www.swen.uwaterloo.ca/~kczarnec/etx04.pdf. Software available from gp.uwaterloo.ca/fmp.
9. World Wide Web Consortium: XML Path Language (XPath) 2.0. (2005) http://www.w3.org/TR/xpath20/.
10. Object Management Group: Unified Modeling Language 2.0. (2004) http://www.omg.org/cgi-bin/apps/doc?ptc/04-10-02.zip.
11. Schnieders, A., Puhlmann, F.: Activity diagram inheritance. In Abramowicz, W., ed.: BIS 2005 - Business Information Systems. 8th International Conference, Poznan, Poland, 2005, Proceedings. (2005)
12. Lee, J., Xue, N.L., Kuei, T.L.: A note on state modeling through inheritance. SIGSOFT Softw. Eng. Notes **23** (1998) 104–110
13. A J H Simons, M P Stannett, K.E.B., Holcombe, W.M.L.: Plug and play safely: Rules for behavioural compatibility. In: Proc. 6th IASTED Int. Conf. Software Engineering and Applications. (2002) 263–268
14. Ziadi, T., Hélouët, L., Jézéquel, J.M.: Towards a uml profile for software product lines. In: PFE. (2003) 129–139

15. Wasowski, A.: Automatic generation of program families by model restrictions. In Nord, R.L., ed.: Software Product Lines: Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September 2, 2004. Proceedings. Volume 3154 of Lecture Notes in Computer Science., Heidelberg, Germany, Springer-Verlag (2004) 73–89
16. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. In: Proceedings of the 25th International Conference on Software Engineering (ICSE), Portland, Oregon, Los Alamitos, CA, IEEE Computer Society (2003) 187–197
17. Tarr, P., Ossher, H., Harrison, W., Stanley M. Sutton, J.: N degrees of separation: multi-dimensional separation of concerns. In: ICSE '99: Proceedings of the 21st international conference on Software engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1999) 107–119
18. Philippow, I., Riebisch, M., Boellert, K.: The hyper/UML approach for feature based software design. In Akkawi, F., Aldawud, O., Booch, G., Clarke, S., Gray, J., Harrison, B., Kandé, M., Stein, D., Tarr, P., Zakaria, A., eds.: The 4th AOSD Modeling With UML Workshop. (2003)
19. McNeile, A.T., Simons, N.: State machines as mixins. Journal of Object Technology **2** (2003) 85–101
20. Prehofer, C.: Plug-and-play composition of features and feature interactions with statechart diagrams. In: FIW. (2003) 43–58
21. Straw, G., Georg, G., Song, E., Ghosh, S., France, R., Bieman, J.M.: Model composition directives. In Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J., eds.: UML 2004 - The Unified Modeling Language. Model Languages and Applications. 7th International Conference, Lisbon, Portugal, October 11-15, 2004, Proceedings. Volume 3273 of LNCS., Springer (2004) 84–97
22. Jarzabek, S., Zhang, H.: Xml-based method and tool for handling variant requirements in domain models. In: RE. (2001) 166–173
23. Czarnecki, K., Bednasch, T., Unger, P., Eisenecker, U.W.: Generative programming for embedded software: An industrial experience report. In Batory, D., Consel, C., Taha, W., eds.: Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02), Pittsburgh, October 6-8, 2002. Volume 2487 of Lecture Notes in Computer Science., Heidelberg, Germany, Springer-Verlag (2002) 156–172