Structuring the modeling space and supporting evolution in software product line engineering

Deepak Dhungana, Paul Grünbacher, Rick Rabiser, Thomas Neumayer

Abstract

The scale and complexity of product lines means that it is practically infeasible to develop a single model of the entire system, regardless of the languages or notations used. The dynamic nature of real-world systems means that product line models need to evolve continuously to meet new customer requirements and to reflect changes of product line artifacts. To address these challenges, product line engineers need to apply different strategies for structuring the modeling space to ease the creation and maintenance of models. This paper presents an approach that aims at reducing the maintenance effort by organizing product lines as a set of interrelated model fragments defining the variability of particular parts of the system. We provide support to semi-automatically merge fragments into complete product line models. We also provide support to automatically detect inconsistencies between product line artifacts and the models representing these artifacts after changes. Furthermore, our approach supports the co-evolution of models and their respective meta-models. We discuss strategies for structuring the modeling space and show the usefulness of our approach using real-world examples from our ongoing industry collaboration.

1. Introduction and motivation

Many software product lines today are developed and maintained using model-based approaches, e.g., feature-oriented modeling (Kang et al., 1990; Czarnecki and Eisenecker, 1999; Asikainen et al., 2006), decision-based approaches (Dhungana et al., 2007a; Schmid and John, 2004), orthogonal approaches (Bachmann et al., 2003), architecture modeling languages (Matinlassi, 2004; Dashofy et al., 2001), or UML-based techniques (Atkinson et al., 2002; Gomaa, 2005). Tools have been developed to automate domain and application engineering based on models defining core assets and their commonalities and variability (Dhungana et al., 2007c). General purpose variability modeling approaches (Gomaa and Shin, 2002; Sinnema et al., 2004; Dhungana et al., 2007a) allow defining the variability of arbitrary domain-specific assets via meta-models that specify the possible elements of variability models (e.g., types of reusable assets, their attributes, types of dependencies).

No matter which modeling approach is followed, product line engineering (PLE) faces two challenges: (i) developing a single product line model is practically infeasible due to the scale and complexity of today's systems: the high number of features and components in real-world systems means that modelers need strategies and mechanisms to organize the modeling space. (ii) new customer requirements, technology changes, and internal enhancements lead to the continuous evolution of a product line's reusable assets: While product line engineers try to understand and capture the variability of a complex existing system, the reusable assets are frequently changed to meet evolving business needs. Evolution support becomes particularly important in a model-based approach to ensure consistency after changes to meta-models, models, and actual development artifacts. Product line approaches need to treat maintenance and evolution as critical due to the longevity of many systems. Many existing approaches are instead based on the assumption that a product line is fairly stable. However, such stability cannot be taken for granted. PLE should thus treat evolution as the normal case and not as the exception (Dhungana et al., 2008b).

An analysis of development practices of our industry partner Siemens VAI, the world's leading steel plant building company, revealed three important issues related to evolution in model-based PLE:

© 2010 Elsevier Inc. All rights reserved.


ARTICLE IN PRESS

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

0164-1212/$ - see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2010.02.018

Please cite this article in press as: Dhungana, D., et al. Structuring the modeling space and supporting evolution in software product line engineering. J. Syst. Software (2010), doi:10.1016/j.jss.2010.02.018
(i) **Modeling space structuring.** The size of many software systems is far beyond the ability of any individual or small group to understand them in detail. This prevents effective coordination because a single individual or small group cannot direct its work and keep all the implementation details in focus (Kraut and Streeter, 1995). To address this challenge the modeling space has to be structured, so that large product lines can be managed more easily. This challenge is related to Conway’s law (Conway, 1968; Herbsleb and Grinster, 1999) describing dependencies between the communication structure of a development team and the technical structure of a system.

(ii) **Model consistency.** Different parts of the system evolve at different speeds and have to be kept consistent with the models describing these parts. Product line assets evolve continuously to address changes such as new customer requirements, technology changes, or refactoring. For example, a large component may be divided into several parts, a component may be moved from one subsystem to another, or new relationships between components may be established. It is therefore essential to understand, model, and maintain the links between the product line’s variability models and its asset base. Engineers should be supported in detecting and keeping track of inconsistencies during modeling (Vier- hausser et al., 2010).

(iii) **Meta-model evolution.** As pointed out domain meta-models are also subject to evolution. In an effective model-driven development cycle modeling tools and techniques must be adaptable to changing requirements in the problem domain. For instance, the introduction of new asset types or the modification of existing asset types require updating existing models. The domain meta-models need to co-evolve with the variability models.

Based on our analysis of current practices and needs of our industry partner, we have developed a model-based approach for defining, managing, and utilizing product lines (Dhungana et al., 2007b; Rabiser et al., 2007). Supporting evolution has been a critical success factor during development. Our approach is based on a simple assumption: A small model is easier to maintain than a large one. Instead of creating a single large product line variability model we use model fragments to describe the variability of selected parts of the system. These model fragments also represent the units of evolution in our approach (Dhungana et al., 2008a). The approach meets the demands of real-world development processes as different teams can work on variability model fragments describing the parts of the system they know best.

We have presented our approach to deal with evolution in previous publications (Dhungana et al., 2008b,a; Grünbacher et al., 2009). Here, we further elaborate the underlying issues and present our experiences of applying the tool-supported approach for a real-world product line of Siemens VAI. The company is maintaining a software product line for the automation of continuous casting in steel plants.

In particular we claim three contributions: (i) An approach based on model fragments for the decentralized creation and maintenance of product line variability models. (ii) Tool support for the automated detection of changes to keep models and architecture consistent. (iii) Tools and techniques facilitating meta-model evolution for propagating changes in the domain to already existing variability models.

The paper is organized as follows: Section 2 elaborates on the needs for structuring the modeling space. In Section 3 we describe our model fragment-based approach. Section 4 presents tool support for creating and managing the model fragments, consistency checking, and meta-model evolution. In Section 5 we discuss our experiences of applying the approach at Siemens VAI and discuss strengths and weaknesses of our approach. Section 6 presents related work. Finally, we present conclusions and an outlook on future work in Section 7.

2. **Structuring the modeling space**

Software evolution is challenged by the fact that development teams require a mix of skills. In many software development organizations development teams are quite fragmented. Single stakeholders can only maintain a small part of a large system. As a result product line engineers need to modularize and organize the modeling space regardless of the concrete modeling approach used. There are several strategies for structuring and organizing the modeling space:

2.1. **Mirroring the solution space structure**

Whenever product lines are modeled for existing software systems, the structure of already available reusable assets can provide a starting point for organizing the modeling space. Models can be created that reflect the structure of the technical solution, e.g., separate variability models for different subsystems of a product line. Similarly, the package structure of a software system or an existing architecture description can serve as an initial structure. The number of different models should be kept small to avoid negative effects on maintainability and consistency. This strategy can be suitable for instance if the responsibilities of developers and architects for certain subsystems are clearly established.

2.2. **Decomposing into multiple product lines**

On a larger scale complex products are often organized using a multi product line structure (Reiser and Weber, 2006). For example, there may be separate product lines for different target customers, e.g., mobile phone product lines for senior citizens, teenagers, and business people (Jaaksi, 2002). Other examples are complex software-intensive systems such as cars or industrial plants with system of systems architectures, which may contain several smaller product lines as part of a larger system. Models have to be defined for each of these product lines and must be kept consistent during domain and application engineering. This strategy often means that different teams create and maintain variability models for the product line they are responsible for.

2.3. **Structuring by asset type**

Another way of dealing with the scale of product line models is to structure the modeling space based on the asset types in the domain. Separate models can then be created for different types of product line assets. Examples are requirements variability models based on use cases (Halms and Pohl, 2004), architecture variability models (Dashofy et al., 2001), or models for technical and user documentation (John, 2002). Structuring by asset type allows managing variability in a coherent manner. It is however important to manage the dependencies between the different types of artifacts which can easily cause additional complexity. This strategy works well with orthogonal approaches (Pohl et al., 2005) that suggest using few variability models that are related with possibly many asset models.

2.4. **Following the organizational structure**

This strategy suggests following the structure of the organization when creating product line models. Different stakeholders
are interested in different concerns of a product line (Dolan et al., 1998). In many organizations, architectural knowledge is distributed across different stakeholders independent of their roles and responsibilities in the development process. Conway's Law (Conway, 1968) states that “...organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations”. In a multi-team environment, individual teams collaborate closely on certain aspects of a product line. It can thus be a good strategy to structure the product line modeling space based on the team structure to reflect the modeling concerns of the involved stakeholder groups. However, creating product line models driven by stakeholders can easily increase the redundancy in models.

2.5. Considering cross-cutting concerns

Using concepts from aspect-oriented development to structure product line models is helpful when many cross-cutting features need to be described. For instance, aspect-oriented product line modeling can be used to model both problem and solution space variability. Voelter and Groher (2007) describe an approach that involves creating a model of the core features all products have in common and defining aspect variability models for product-specific features shared by only some products. Complex aspect dependencies can however lead to difficulties with regard to managing their interaction.

2.6. Focusing on market needs

Structuring the modeling space can also be driven by business and management considerations, e.g., from product management (Helferich et al., 2006) or marketing (Kang et al., 2002) perspective. Focusing variability modeling on business considerations eases the communication with customers. If following this strategy in pure form, models might however become unrelated with the technical solution thereby making it hard to understand the actual implementation and to automate the derivation of products.

3. Model fragments for structuring the modeling space

We discussed a number of strategies for structuring the modeling space. Implementing these strategies requires support for modeling in the large and modeling in the small. The high number of components in many product lines suggests a coarser granularity of modeling to reduce complexity. The approach we present in this paper is based on model fragments. A model fragment is a partial model with defined dependencies to other model fragments. For instance, model fragments can be defined for different subsystems of a product line. Model fragments have to be merged to create a complete model of the product line. Our model merging approach allows detecting syntactical and structural merge conflicts (Mens, 2002). The approach works semi-automatically: some merge conflicts can be resolved automatically without user intervention, some require user intervention. Our tool-supported approach provides capabilities for creating model fragments, defining inter-fragment relationships, and merging the model fragments. Model fragments can be used to structure the modeling space and provide coarse-grained support for evolution.

From a high-level point of view, there are two possible strategies for specifying model fragments and their dependencies:

Explicit dependencies. Model fragment owners explicitly refer to elements in other model fragments when specifying dependencies. This requires knowledge about the model elements in other fragments at modeling time and can be compared to explicit import statements in programming languages. Explicit dependencies simplify the integration of model fragments but result in a lack of flexibility during modeling. Model explicit dependencies mean that the different teams need to consult each other and agree on the “model interfaces”. In a multi-team development environment this can be challenging and even counterproductive. Engineers of our industry partner have thus demanded more leeway for creating and evolving product line models.

Lazy dependencies. Instead of explicit links to elements in other fragments modelers define placeholder elements at modeling time that are mapped to real elements before the models are used (i.e., in product derivation). This approach allows temporary inconsistency during modeling (Balzer, 1991) which is for instance relevant when modeling new features or when making local changes to a subsystem. This approach allows users to create and evolve model fragments without explicit coordination and increases the flexibility for modelers in multi-team environments. It however requires support for identifying and resolving temporary inconsistencies.

Whenever multiple fragments are created, they need to be integrated before being used, i.e., in product derivation. In case of explicit dependencies the integration process is easier as ambiguities have already been avoided by modelers when creating the model fragments. In case of lazy dependencies (i.e., placeholder elements) modelers have to resolve dependencies between fragments manually or with the help of a tool. Our approach uses lazy dependencies to increase the freedom for modelers. We aim to compensate the more difficult integration process with proper tool support.

3.1. Model fragments

We use a decision-oriented variability modeling approach (Dhungana et al., 2007a,b) that supports variability modeling of the problem space (stakeholder needs or desired features), the solution space (the architecture and the components of the technical solution), and traceability between these spaces. Problem space models are created using decisions while solution space models are built using assets. The generic meta-model (cf. Fig. 1) is adaptable.
to domain-specific concepts and allows defining concrete asset types, asset type attributes, and relationships between assets. A formal semantics of the modeling approach can be found in Dhungana et al., 2010.

Variability model fragments contain Assets and Decisions as the two key modeling elements (Dhungana et al., 2007b,a). A decision is defined whenever for a given goal (e.g., configuring a component) there exist two or more ways of achieving it. Decisions represent the variation points in a product line model. Taking a decision involves judging the merits of multiple options and selecting one of them for action (e.g., when considering customer requirements). Taking a decision defines a value for that decision. Possible values depend on the type of the decision (e.g., Boolean or Number). Assets describe the product line’s reusable artifacts and their dependencies that are available in a certain domain. We use the term asset as a generic term to represent all kinds of artifacts whose variability needs to be modeled. This is required, as different mechanisms are typically used to achieve variability at different artifact levels such as requirements, architecture, or implementation. Decisions and Assets are linked using inclusion conditions. For instance, components refer to the possible values of the decisions to specify the conditions under which the components are part of a derived product. In this way, assets are “aware” of the conditions under which they are required for the final product whereas decisions are “unaware” of the assets. It is noteworthy, that such one-way dependencies from assets to decisions allow having different decision models to configure the same set of assets. Such flexibility is important to enable changes in the problem space (e.g., reflecting changing marketing plans), without having to adapt the reusable assets.

Assets and Decisions in a model fragment are either model elements or placeholder elements (cf. lower half of Fig. 1). We use concepts from object-oriented programming languages to define the visibility of model elements in model fragments. Similar to private and public elements in classes, modelers can specify public elements of a fragment to make them visible outside the model (cf. Fig. 2). Model elements are defined as private elements if they are not relevant in other parts of the system and must not be known outside, i.e., they are internal to a subsystem with no direct relationships to elements in other models.

Placeholder elements are comparable to “required interfaces” in architecture description languages. Placeholders have a data type defined in the meta-model and can be seen as tagged and typed variables which will be replaced with model elements defined in other model fragments during merging. Placeholder elements are introduced in a model fragment whenever relationships to elements from other model fragments need to be defined. This is for instance necessary when specifying product composition rules between elements. The explicit location or the exact names of the referenced elements are not needed during modeling (Dhungana et al., 2008a).

For example, the model fragment PIM in Fig. 2 contains a placeholder for the decision archive that is used to define the dependency between dbSupport and archive. The decision dbRequired in model fragment Database is a placeholder for a decision defined in another model fragment. The modeler creating fragment Database does not need to know the real name of the element she is referring to (dbSupport). During merging dbRequired is replaced with dbSupport from model fragment PIM.

All references are refactored accordingly.

3.2. Product line structuring and evolution with model fragments

Fig. 3 depicts an overview of our approach to product line evolution. Based on the chosen strategy for structuring the modeling space (see Section 2), several model fragments are created and managed independently. Whenever required, product line engineers create a single model (e.g., at time t1) by merging the fragments available at this time. The original model fragments are not changed during merging, rather a new model is created for derivation based on the fragments. The process of merging the models and the user actions for resolving conflicts are recorded for future use. After a set of changes to various fragments another merge process is initiated at time t2. The “re-merge process” benefits from the conflict resolutions recorded in the merge at t1. The merged model can then be used for another product derivation.

Product line evolution occurs in two dimensions (cf. Fig. 3) as both the meta-model and the variability models can evolve independently: (i) Models are subject to change whenever the product line changes; e.g., as a result of improving or extending functionality, changing technology, or reorganization of existing

![Fig. 2. Two variability model fragments with public and private model elements. Public elements are visible outside a model fragment. Other fragments can define lazy dependencies using placeholder elements. Private elements are only visible inside a specific fragment.](image)

![Fig. 3. Product line evolution based on model fragments.](image)
assets. (ii) Meta-models evolve due to changes in the scope of the product line; e.g., new asset types are introduced or the product line itself is extended to support new business units.

More specifically, the key elements of our approach are (cf. Fig. 3):

A model fragment describes the reusable assets and their variability for an arbitrary part of the product line (e.g., a set of features, a subsystem, or cross-cutting functionality). Model fragments serve as the basic unit of evolution and are created and maintained by individual teams only loosely coupled with each other. Model fragments are never directly utilized in product derivation. They may evolve independently and at different speeds.

A variability model is merged from a set of model fragments at certain points in the product line life-cycle (e.g., before starting product derivation). Unlike model fragments, a variability model can be used in product derivation (Rabiser et al., 2007) assuming that inconsistencies between the constituent fragments have been resolved during merging. The resulting model must not be changed. It can only be updated by modifying and re-merging the model fragments from which it has been initially created.

The merge history establishes trace links between model fragments and a merged model. Model fragment owners use the merge history to revise their individual fragments based on the applied conflict resolution actions to expedite future merge processes.

The domain-specific meta-model is used to take into account the specifics of an organization or domain. It defines the types of assets to be reused in the product line (e.g., components, services, documents, parameters, etc.) together with their attributes and dependencies (Dhungana et al., 2007a). Variability models and model fragments are both based on a particular domain-specific meta-model.

### 3.3 Fragment merging

Model fragments are incomplete as they represent only a partial view of the system. Lazy dependencies to other parts of the system – modeled using placeholders – need to be resolved before a single model can be generated for product derivation. During merging the elements of the constituent model fragments are collected in a new model. The placeholders are replaced with corresponding model elements from other model fragments. The variability model resulting from a merge process is based on the same domain-specific meta-model as the source model fragments. These remain unchanged during merging. Table 1 lists four types of merge conflicts that can occur together with a resolution strategy.

#### 3.3.1 Multiple occurrence of identifiers

It is essential that all the elements in a model have a unique identifier. However, as modelers are often not aware of other model fragments during modeling, elements in different model fragments can have the same name which leads to a conflict during merging. At least one of the conflicting elements has to be renamed by the user.

#### 3.3.2 Name mismatches

The name of a placeholder might not match the name of the intended element. For example, a model fragment might contain a model element named dbSupport and some other fragments may refer to the same element using the name dbRequired (cf. Fig. 2). Such cases are difficult to resolve automatically and we rely on human experts during merging to confirm the semantic equality of the used element names. Domain-specific glossaries defining synonyms of the used names ease merging in such cases.

#### 3.3.3 Multiple definitions

Different model fragments may define the variability of a common part of a system. This can for example happen when shared components are used by more than one subsystem, and several subsystem owners decide to model the shared components’ variability as a part of their subsystem. Our merging algorithm includes only one instance in the merged model (based on naming conventions, types of elements, and relationships among elements). For example, whenever a component with the same name is contained in more than one model fragment, the user either decides to rename one of the components before merging or to include the component only once in the merged model.

#### 3.3.4 No matches for placeholders

It is also possible that several model fragments define placeholders for which no “real” model element exists. Again user intervention is required to resolve the problem. The user selects a binding element from a list of suggested candidate elements. If

### Table 1

<table>
<thead>
<tr>
<th>Merge conflict</th>
<th>Resolution strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple occurrence of identifiers</td>
<td>Rename involved elements</td>
</tr>
<tr>
<td>Name mismatches</td>
<td>Synonym check with glossary</td>
</tr>
<tr>
<td>Multiple definitions</td>
<td>User-confirms semantic equality</td>
</tr>
<tr>
<td>No matches for placeholders</td>
<td>Automatically suggest a candidate resolution</td>
</tr>
</tbody>
</table>

Please cite this article in press as: Dhungana, D., et al. Structuring the modeling space and supporting evolution in software product line engineering. J. Syst. Software (2010), doi:10.1016/j.jss.2010.02.018
no binding element is available, the resulting variability model will still contain unresolved placeholders.

Algorithm 2: FindDefinition.

Input: PlaceHolderElement placeholderElem, ModelFragment Fragment
Output: ModelElement

init ModelElement resolvedElement = null
Boolean found = false // look for placeholderElem in Fragment

foreach ModelElement elem in Fragment do
    if (type of elem matches type of placeholderElem)
        if (name of elem matches name of placeholderElem)
            resolvedElement = elem
            found = true
        end
    if (synonym of elem matches synonym of placeholderElem)
        Rename the model element to match the placeholder element
        Rename variables in attributes and dependency conditions
        resolvedElement = elem
        found = true
    end
end

if (NOT found) then
    foreach ModelElement elem in Fragment do
        // check semantic equality
        boolean confirm = User-Confirms-Equality(elem, placeholderElem)
        if (confirm) then
            resolvedElement = elem
        end
    end
end

return resolvedElement

Fig. 4 depicts the result of merging the two model fragments from Fig. 2. Whenever model elements are renamed (in case of name conflicts), deleted, or dropped (in case of multiple occurrences), their attributes, dependencies, and conditions also need to be updated accordingly. For example, the condition \( \text{visible if dbRequired}==\text{true} \) in model fragment 2 of Fig. 2, was automatically changed to \( \text{visible if dbSupport}==\text{true} \) during merging (cf. Fig. 4). This is because the placeholder element \( \text{dbRequired} \) was mapped to \( \text{dbSupport} \). Algorithm 1 shows a high-level view of our merging algorithm which merges \( \text{Fragment1} \) and \( \text{Fragment2} \) into \( \text{ResultFragment} \). Algorithm 2 shows how matching elements for a particular placeholder are found in other fragments.

We record the applied changes and bindings in a merge history (cf. Fig. 3) which enables three important features:

3.3.5. Forward and backward traceability
The merge history links the model fragments and the variability model to support forward traceability (“How are the elements of the model fragment used in the merged variability model?”) and backward traceability (“What is the originating model fragment for a certain element in the merged variability model?”).

3.3.6. Feedback to model fragment owners
The original model fragments remain unchanged after merging. However, model fragment owners are informed about conflict resolution changes such as deleting or renaming elements, references, and relationships applied during merging to avoid that teams slowly disconnect from each other. Teams get information regarding the change actions that were necessary during merging (e.g., which elements had to be renamed). They may decide to revise their model fragments based on this feedback (cf. Table 2). This helps teams to converge and agree on definitions in the model fragments.

3.3.7. Repeatability of merging
In case of frequent changes to the fragments and a high number of fragments, repeating the merging process each time from scratch would be tedious. Whenever the merge process has to be repeated the merge history is thus used to replay the previously taken change actions with minimal user intervention (quick re-merge). Besides user choices made during merging, the merge history contains all change actions automatically performed during merging (e.g., renames, reference mappings, or changes in attributes or relationship links).

3.4. Consistency of model fragments and the system
Models have to be kept consistent with the system they represent. Engineers frequently change components and variability model fragments (e.g., when introducing new variants). Inconsistencies resulting from such changes need to be detected and fixed to support the co-evolution of the system with the respective model fragments and vice versa. Changes to software components of the product line architecture have to be propagated to the models. Similarly, models should not simply be changed without changing the corresponding architectural elements. We found that in many cases it is possible to keep track of the changes in the underlying core assets and to synchronize the models automatically because the elements in the variability models directly map to the existing

Table 2
Different types of merging strategies and possible feedback.

<table>
<thead>
<tr>
<th>Resolution strategy</th>
<th>Feedback to fragment owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element renamed</td>
<td>Rename element in the fragment to ensure positive match</td>
</tr>
<tr>
<td>Element deleted</td>
<td>Drop element from fragment</td>
</tr>
<tr>
<td>Missing resolutions of</td>
<td>Either remove the placeholder or change to real placeholder model element</td>
</tr>
</tbody>
</table>
assets in our approach (Vierhauser et al., 2010). There are two main types of inconsistencies:

3.4.1. Orphaned model elements
A model may contain elements that have been deleted or re-named in the asset base. In order to resolve this inconsistency, either the obsolete model element is deleted or the asset base is changed to match the model. The model may also contain dependencies between elements that are no longer correct or available.

3.4.2. Missing model elements
It is also possible that not all parts of the system are already covered in the model. Such cases happen, e.g., if components are added to the asset base or modified to meet the needs of daily business. Our approach automatically detects such inconsistencies and informs the users about them.

3.5. Consistency of model fragments and meta-model
Our experience of modeling and maintaining a product line with our industry partner has revealed that it is necessary to continuously modify and fine-tune the meta-model, especially in early phases of modeling. The usual strategy is to start with a smaller meta-model consisting of few asset types and relationships only and to extend this meta-model over time to incorporate additional asset types and relationships. Adequate support for meta-model evolution is thus essential as otherwise variability models and model fragments based on changing meta-models would become invalid. Table 3 depicts some of the changes that can occur in the meta-model and the strategy to propagate the changes to existing variability models.

### 4. Tool support

We have developed support for our approach as part of our DOPLER tool suite (Dhungana et al., 2007c, 2008a).

#### 4.1. Defining and merging model fragments

Our tools allow defining separate model fragments for arbitrary parts of the product line to support different strategies for structuring the modeling space. Fig. 5 shows the model merger tool integrated in our variability modeling tool Decision – x’ing (Dhungana et al., 2007a). Suggestions for conflict resolution are presented to the user (cf. Fig. 6). The most likely choice can be selected automatically by activating the auto-resolution feature before model merging. In this case the tool selects the most likely resolution without user interaction.

#### 4.2. Consistency checking

Our variability modeling tool Decision – x’ing provides an error viewer, which displays inconsistencies between models and the system they represent. The tool uses existing models and artifacts to find inconsistencies: Whenever models are changed, existing architectural elements are used as a reference for comparison. Whenever architectural elements are changed, the existing models serve as a lookup table. For example, if a new variant is introduced by changing the variability model, the tool ensures that there exists an artifact with the same name and structure (together with dependencies to other artifacts). Similarly, when a new component is added to the architecture, the tool automatically looks for the existence of its representation in the available variability models.

Decision – x’ing allows integrating plug-ins to enable domain-specific model checking. For example, we extended the default model builder in Eclipse to deal with inconsistencies in Spring configuration files used by Siemens VAI. There are four types of error messages detected by the Siemens VAI Builder, as depicted in Fig. 7:

- **Missing relationship:** Dependencies among assets are detected in the file system but are not reflected in the model.
- **Missing asset:** Assets are detected in the file system but are not modeled.
- **Orphaned relationship:** Dependencies are modeled in the variability model, but not present in the file system.
- **Orphaned asset:** Assets are modeled in the variability model but are not found in the file system.

More recently we have been developing an incremental consistency checker (Vierhauser et al., 2010) to improve scalability and to provide immediate feedback to engineers who need to detect and keep track of inconsistencies during modeling.

#### 4.3. Supporting Meta-model evolution

Our tool also provides a meta-model change propagator which allows updating existing variability models after changes to the meta-model. Different versions of domain-specific meta-models are compared and the differences are shown in different colors. The tool presents suggestions for actions, which can be carried out to update the outdated models. Adding new elements (asset types, attributes, or relationships) to an existing meta-model is straightforward as they can just be added to the new model without affecting already existing model elements. Decision – x’ing’s meta-model change propagator also supports the deletion of meta-model elements. However, user confirmation is required in such cases as the update will result in the deletion of model elements, their attributes, and relationships. The variability models which are based on the meta-models can be updated by carrying out the suggested actions.

### 5. Industrial experiences

We report on experiences of using the approach for Siemens VAI, the world’s leading engineering and plant building company for the iron, steel, and aluminum industries. In our collaboration we have been developing the decision-oriented product line engineering approach DOPLER (Dhungana et al., 2007a,c). The domain of interest is our industry partner’s process automation software for continuous casting in steel plants (CC-L2; Java-based, about 1.5 MLOC). In the development process different teams are in charge for various subsystems of the product line (e.g., cooling.

---

Table 3

<table>
<thead>
<tr>
<th>Meta-model Change</th>
<th>Model update strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition of asset type</td>
<td>Introduce a new type in the model</td>
</tr>
<tr>
<td>Deletion of asset type</td>
<td>Delete all assets of the corresponding asset type</td>
</tr>
<tr>
<td>Addition of asset attribute to an existing asset type</td>
<td>Add the new attribute to all existing assets of the corresponding type and initialize with default value</td>
</tr>
<tr>
<td>Deletion of asset attribute from an existing asset type</td>
<td>Delete the corresponding attribute from all existing assets of the corresponding type</td>
</tr>
<tr>
<td>Addition of a new asset relationship link</td>
<td>Update the models to allow the user to create links between the newly changed types</td>
</tr>
<tr>
<td>Deletion of an existing asset relationship link</td>
<td>Delete all links to related assets that were linked with the deleted link</td>
</tr>
</tbody>
</table>

---

Please cite this article in press as: Dhungana, D., et al. Structuring the modeling space and supporting evolution in software product line engineering. J. Syst. Software (2010), doi:10.1016/j.jss.2010.02.018
The Spring Framework is used to configure software components and their dependencies. Product derivation is currently supported via changes in Spring files to specify the desired variants of components and via specification of thousands of configuration parameters stored in property files to control the initialization and behavior of these components. While these mechanisms work considerably well, the complexity of product derivation led Siemens VAI to complement their product line with a more formal approach to variability modeling and tool support for the semi-automatic creation of valid system configurations (Schwanninger et al., 2009).

Our first brute-force approach was to put all model elements into one single model. This did not scale due to the size and complexity of the product line (over thousand components and about 100 decisions with several hundred non-trivial dependencies). It also became apparent early on in the project that working with a single variability model is inadequate to support evolution in the multi-team development environment at Siemens VAI. This led

---

Fig. 5. Model merger tool. Markers (small boxes with an x top right of elements’ icons) are used to indicate conflicts. Users can choose from automatically generated suggestions to resolve merging conflicts.

Fig. 6. Merger suggestions for resolving conflicts.
us to develop the presented model fragment approach. We then experimented with the different strategies for structuring the modeling space described before. We did not explicitly choose a single strategy but rather started combining them on demand.

5.1. Initial modeling space structure: solution space and business concerns

The initial strategy was to use the existing technical solution structure as the starting point for structuring our models. People in charge of a particular subsystem have intimate knowledge about this part of the product line, however, no single developer has the knowledge required to model the entire system.

The variability of the system was therefore elicited in two ways (see Fig. 8): (i) We conducted moderated workshops with engineers and project managers responsible for various subsystems to identify major differences of products delivered to customers. We described these differences as decisions in separate models for each subsystem (Rabiser et al., 2008). Business decisions (BD) represent external variability (relevant for and visible to the customers). Technical decisions (TD) define internal variability relevant for the engineers and not visible to customers. (ii) We used automated tools to understand the technical variability at the level of components by parsing existing configuration files. The tools identified variabilities at the technical level and made suggestions which were either accepted as decisions or rejected as irrelevant manually by developers.

Using this strategy, we created variability model fragments for 11 subsystems of CC-L2. The developed variability model fragments vary in size and complexity due to the different scope of the subsystems. The average model fragment contains 50 assets (Spring components and configuration parameters), 12 decisions, and 23 placeholders to express inter-fragment dependencies (Dhungana et al., 2008a). The model merger was frequently used to create an integrated variability model from the fragments. Most of the merging process was done automatically using our tool’s conflict resolution features. In initial stages of the merge process intervention was required in 28 cases to resolve ambiguities. The total number of mappings that occurred was 120; meaning user intervention is required in about 23 percent of mapping. This number however changes depending on the models in use. The current version of the merged model contains 324 components, 160 configuration parameters, and 78 decisions.

Our experiences confirm several advantages of our approach: It supports the clear separation of concerns through variability model fragments for different subsystems. This eases model development and evolution in multi-team modeling environments but also increases model complexity. Different stakeholder perspectives are considered by modeling both business and technical variability of the system. However, we also noticed two weaknesses which made it necessary to further refine the approach and to revisit the strategy for structuring the modeling space: Mixing technical and business decisions in one model fragment can cause problems as different people are responsible for maintaining these decisions. Mixing assets and decisions in one model fragment has negative effects on model maintenance. While assets change often (e.g., due to frequently required refactoring of the technical solution), decisions turned out to be more stable. We thus decided to refine our initial strategy.
5.2. Refined model space structure: layered modeling of problem and solution space

Based on the lessons we learned from applying the initial strategy we refined our modeling space structure and created separate model fragments for assets and decisions. Decision model fragments were further divided into business and technical decision model fragments. This allowed us to more clearly separate external variability (captured as business decisions) from internal variability (captured as technical decisions). In the initial strategy we considered all subsystems at the same level. However, a refactoring of the CC-L2 architecture led to a clearer separation of architectural layers. We thus followed this new architectural style and created asset and decision model fragments in a layered fashion (see Fig. 9).

Following this new structure, we have been refactoring the existing variability models for Siemens VAI’s CC-L2 system that resulted from the initial strategy. We created initial variability model fragments for two architectural layers, one layer representing the platform for CC-L2 (comprising common functionality that could also be used by other systems than CC-L2) and one representing CC-L2-specific functionality. We also defined additional business decisions based on existing documents (i.e., contracts and specifications), a source of information we had neglected in our initial strategy.

With our second strategy, we created 6 variability model fragments: two business decision model fragments based on existing documents, two technical decision model fragments for the CC-L2 platform layer and the CC-L2-specific layer, and two asset model fragments (one for each layer). The developed fragments vary in size and complexity. The two business decision model fragments in total contain 68 decisions (one 53 and the other 15) and 14 placeholders (one 10 and the other 4). The two technical decision model fragments are quite similar in size: in total, they contain 27 decisions (one 14 and the other 13) and 17 placeholders (one 9 and the other 8). The platform layer asset fragment contains 99 components and 57 configuration parameters. The CC-L2-specific layer asset fragment contains 363 components and 64 configuration parameters. Again, we used the model merger tool frequently to create an integrated variability model from the fragments. Again, most of the merging process was done automatically using our tool’s conflict resolution features. The current version of the merged model created from the diverse fragments resulting from following our second strategy contains 462 components, 121 configuration parameters, and 95 decisions. Compared to the first strategy, the number of components in the merged model increased and the number of parameters decreased. This is mainly due to the refactoring of the technical solution Siemens VAI developers conducted. However, the higher number of decisions is not only a result of this refactoring but also from exploring additional sources of variability information, i.e., we extracted business decisions from existing documents like contracts and specifications.

Fig. 9. The refined layered strategy. Asset model fragments and decision model fragments are defined for different architectural layers. This structure allows to handle business decisions (BD) and technical decisions (TD) separately.
Compared to the initial approach the new strategy allows to separately manage decisions and assets at different layers of the architecture. Changes of the underlying components can be semi-automatically propagated to the asset model fragments which helps to reduce the maintenance effort. The new approach also is more explicit regarding the definition of stakeholder roles. Model fragments can be assigned to designated individuals or groups to ensure their maintenance. We used existing documents such as specifications and contracts to identify business decisions. This helped to better understand the external variability of the system and to narrow the gap from technical- to business-oriented stakeholders. We achieved a clear separation of concerns based on external and internal variability.

5.3. Evolution

We have designed and developed the presented product line evolution capabilities iteratively and in close collaboration with engineers of our industry partner Siemens VAI to benefit from continuous feedback. We have been experiencing the continuous evolution of the product line which allowed us to validate the approach by iteratively developing and testing our evolution capabilities using real-world models. Our tools have been used by engineers at Siemens VAI to create and evolve variability models.

Engineers changed the underlying product line assets frequently during modeling which again confirmed the need for evolution support. For instance, they updated Spring XML configuration files and introduced new relationships between components frequently when refactoring components. The inconsistencies resulting from these changes were detected automatically by our consistency checking tool and the engineers were able to fix the variability models accordingly. We are currently working on automated support for fixing inconsistencies after changes have been made to the model or the architecture.

Meta-model evolution capabilities were particularly important to support product line adoption at our industry partner. The meta-model evolution capability was developed early on as a static meta-model did not provide the necessary flexibility. We started with a relatively simple meta-model which was extended and modified to model further aspects of the product line as the project progressed. The initial simple meta-model reflected the basic asset types Component (representing Spring XML components), Property (representing individual configuration parameters), and Resource (representing additional files such as third party licenses or hardware specs). Based on this simple meta-model we introduced new and changed existing asset types, attributes, and dependencies in the project over time as required. For instance, we added a new asset type Document to the meta-model for modeling arbitrary pieces of documentation (technical specifications, detailed designs, user manuals, etc.). This iterative approach ensured continuous validation of the evolving meta-model and helped avoiding the introduction of unneeded concepts. Our meta-model evolution features were essential to automatically propagate the changes to existing model fragments.

5.4. Discussion

Having applied our approach and tools at Siemens VAI we have learned that it is important to find a balance between the separation of concerns and the complexity and effort of merging required when separating too much. Too many fragments and infrequent merging makes the approach infeasible. However, too few models or too frequent merging make the approach impractical too. One has to find a balance between the practicability of distributed modeling and the number of model fragments. The pay-off point strongly depends on the concrete domain or context. In our case, we have learned that it makes sense to reflect the architectural structure of the software to be modeled. It is also helpful to take the structure of the team into account (subsystem owners are model owners). Our experience shows that it is advisable to frequently merge the fragments to resolve conflicts early and often.

In a paper on the state-of-the-art of software merging Mens (2002) discusses six design criteria and their influence on the design of a merge tool. We use them to structure the discussion of the strengths and weaknesses of our tool-supported approach:

5.4.1. Degree of formality

Our merging approach is a lightweight formal approach. Our tool support detects syntactical and structural differences between two model fragments to be merged based on the formalism given by the syntax and structure of our model fragments. The support for re-merging contributes to the practical use of the tool.

5.4.2. Accuracy

Our approach is as accurate as the user wants it to be. When the fragments are merged, the tool only makes suggestions as how the conflicts can be resolved. The user is responsible for the accuracy of her decisions.

5.4.3. Domain independence and customizability

Our tool allows merging variability model fragments created with our decision-based approach. However, the principles of our approach (creating model fragments independently, defining placeholder elements, semi-automatic merging) are generally applicable also to models (or model fragments) created with other approaches. Our approach is customizable as it is based on a meta-modeling approach that allows to define arbitrary domain-specific decision and asset types, dependencies, and attributes.

5.4.4. Granularity

Our merge tool does not consider the granularity of the elements to be merged. For example, it does not matter whether assets to be merged are characters, lines of code, parameters, variables, methods, classes, packages, subsystems, or other artifacts. The granularity is fixed as it always compares elements, their attributes, and dependencies to detect merge conflicts.

5.4.5. Scalability and efficiency

Our experiences with Siemens VAI models suggest that our approach scales for real-world, large scale systems. With a size of about 1,5 MLoC distributed among about 1000 assets in diverse subsystems, Siemens VAI’s system is quite big. When creating product line models it typically does not make sense to model every technically possible variability. One should follow a value-based approach to variability modeling. Nevertheless, we modeled over 100 decisions in our diverse model fragments. Hundreds of non-trivial dependencies among decisions, among assets, and among assets and decisions further demonstrate the size and complexity of the product line we modeled. The efficiency of our merging approach strongly depends on the frequency of merging, i.e., scalability might suffer if many models are developed separately over a long period of time without merging. On the other hand, if merging frequently, the required user intervention typically remains small.

5.4.6. Degree of automation

Our approach is semi-automatic as not all merge conflicts can be resolved automatically and some user intervention is required. The strengths of our approach are the high degree of leeway given to modelers through lazy (placeholder) references; tool support for semi-automatic merging and re-merging to reduce user intervention; as well as support for propagating meta-model
changes to existing models. Our approach does not detect semantic merge conflicts, it only supports syntactical and structural merging. While the idea of our approach is generally applicable to graph-based models of arbitrary kinds, our tool support has been developed to support merging of model fragments created with our decision-based variability modeling tool.

Jeanneret et al., 2008 propose a process framework for model composition that can be used to compare different composition approaches. One of the key insights of the paper is that model composition is not simply an operator allowing complete automation. Our experience has confirmed this. Here we present a summary of how our approach fits into the comparison criteria presented in Jeanneret et al., 2008:

**Ordering of composition rules:** The users choose the best ordering for resolving conflicts based on their knowledge and experience.

**Stopping criteria:** No more matching elements for placeholders are found or all placeholder elements have a mapping.

**Contribution of input elements:** The contributing elements come from the input model fragments.

**Location of inserted elements:** The location of the elements in the input models is reflected in the merged model.

**Combination of fragments:** More than one model contributes to the composition process and the contributions are combined into a single a set of model elements.

**Translation of meta-models:** There is no such translation in our merging approach and the input models must conform to the same meta-model.

**Fastening of inserted elements:** This is supported as once a fragment is inserted into its location, it is adjusted to its new context. This action makes sure that the fragment is connected to the rest of the composed model.

### 6. Related work

Our discussion of related work covers the areas of model evolution, product line evolution, multi-team modeling, organizing modeling spaces, and model merging.

#### 6.1. Model evolution

Our work provides a model-driven perspective on product line engineering and evolution. There are several approaches to deal with the differing and merging of models. Deng et al. (2008) describe a model-driven product line approach that explicitly focuses on the problem of domain evolution with regard to product line architectures. They discuss several challenges for the evolution of model-driven software product line architectures and present their solution, i.e., supporting evolution with automated domain model transformations. Xing and Stroulia (2005) present an algorithm for automatically detecting structural changes between the designs of subsequent versions of object-oriented software. Sprinkle (2003) discusses the model migration problem for evolving meta-models, i.e., that models defined based on an earlier version of a meta-model become invalid when the meta-model evolves. Their solution is to automatically migrate existing models such that they conform to the new meta-model, while preserving the information the models contain as much as possible. We provide a similar solution to meta-model evolution in our approach. Another related area is consistency between models and their implementation: Murta et al. (2006) address the consistency of architecture models to implementation focusing on evolution. Their approach is based on recording every change. While we record changes in the merging process to ease subsequent merging, Murta et al. (2006) record changes in the configuration management system to support arbitrary evolution policies.

#### 6.2. Product line evolution

Many PLE approaches assume that activities in domain and application engineering can take a fairly stable product line for granted. However, PLE should treat evolution as the normal case not as the exception. Despite its importance surprisingly few papers are available on product line evolution (e.g., Bosch, 2000; McGregor, 2003; Svahnberg and Bosch, 1999). Evolution support becomes success-critical especially in a model-based evolution approach to ensure consistency after changes to meta-models, models, and actual development artifacts. Several authors (Johnson and Bosch, 2000; Siy and Perry, 1998) stress the importance of approaches for product line evolution to avoid the erosion of a product line, i.e., the deviation from product line models up to the point where key properties no longer hold. Mende et al. (2008) describe tool-support for the evolution of software product lines based on the “grow-and-prune” model, i.e., they support identifying and refactoring code that has been created by copy & paste and which might be moved from product to product line level. Svahnberg and Bosch (1999) report on experiences regarding the evolution of products, software components, and software architecture. Mattsson and Bosch (1998) discuss the evolution of frameworks, and the types of changes affecting frameworks: internal reorganization, change in functionality, extension of functionality, and reduction of functionality. These types of changes were also highly relevant in our industrial research collaboration. Finding the right granularity for evolution is an art but essential to make evolution manageable. A popular approach is to support product line evolution on the level of architectural elements with components as the units of evolution (Magee et al., 1995; van Ommering et al., 2000): components are treated as black boxes and their internal structure is thus not a concern for evolution. Our industry partner is using such a component-based approach supported by the Spring framework.

#### 6.3. Multi-team modeling

The problem of communication in multi-team development environments has been addressed by several authors. For example, the dependencies between the communication structure of a development team and the technical structure of a system are stated in Conway’s law (Conway, 1968). Parnas describes modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time (Parnas, 1972). Nuseibeh et al. (1993) report on multi-perspective software development and discuss the inevitability of multiple inconsistent views. Thurimella et al. (2008) also recognize the need for a better collaboration between distributed stakeholders in variability management and product line evolution and propose an approach to capture the rationale of important issues. We experienced similar needs for separating different concerns, decentralized specification, and support for integrating different views through relationships and composition when creating variability models for the product line of Siemens VAI.

#### 6.4. Organizing modeling spaces

The aspect of organizing modeling spaces in product line engineering has already been addressed by some researchers: For instance, Hunt (2006) discusses the challenge of organizing an asset base for product derivation. Cho et al. (2008) propose aspect-oriented implementation patterns to support the management of...
model dependencies, i.e., dependencies between product line assets and variable features. Jorgenson (2006) presents an approach to model a product line at different levels of abstraction. While these approaches give some initial answers, structuring the modeling space remains challenging. In practical product line settings it does not make sense to use the described strategies in their pure form. Instead, the different approaches have to be combined: For example, a system of systems can be modeled for different customer types while at the same time also structuring the resulting models by asset types. The importance of such hybrid approaches is also reflected in recent research. For instance, several papers appeared about linking problem space and solution space models (Heidenreich and Wende, 2007; Metzger et al., 2007; Cho et al., 2008).

6.5. Software merging

Mens (2002) presents an extensive overview and an analysis of existing merge approaches. He presents different alternatives for categorizing merge techniques, i.e., two-way vs. three-way merging: textual, syntactical, semantic, or structural conflicts; state-based, change-based, or operation-based merging; reuse vs. evolution. A detailed discussion of merge conflicts (detection, resolution, reduction), delta algorithms, and design criteria for merge tools is also provided. The survey is focused on merging software and not on merging models. However, many of the discussed principles can also be applied for model merging. Our approach supports merging syntactical and structural conflicts between our graph-based models.

6.6. Model merging

In Reddy and France, 2005, the authors present a model composition technique that relies syntactic pattern matching. A model is said to match with another if their signatures match. A signature consists of some or all properties of an element as defined in the UML meta-model. The technique proposed in this paper can be used to detect conflicts that arise during composition. Morin et al., 2008b presents an approach describing how to ensure that an aspect model with variability can be safely integrated into an existing model. A similar approach for aspect-oriented systems is presented in Morin et al., 2008a, which introduces GeKo, a generic weaver for supporting product lines. GeKo relies on the definition of mappings between the different views of an aspect, based on the concrete (graphical) syntax.

There are other papers related to model merging. However, the goal of these approaches is slightly different to ours; these papers are not motivated by scalability of modeling or structuring of the modeling space. For example, Saval et al. (2009) present initial ideas and a preliminary approach to merging class diagrams based on feature models. Sabetzadeh et al. (2007) use model merging to detect structural inconsistencies by transforming a static model into a graph and then into a relational model. These approaches have not been developed with the primary intention of achieving better scalability of modeling or structuring of the modeling space.

7. Conclusions and future work

Support for structuring the modeling space and support for evolution are key requirements for model-based product line engineering which are not adequately addressed by current approaches. We presented a tool-supported approach that aims to treat evolution as the normal case and not as the exception. The approach has been validated using real-world models developed and evolved in an ongoing industry collaboration with Siemens VAI. While our tool support has mainly been developed for Siemens VAI, our approach (i.e., model fragments) is generally applicable to graph-like structures and models. Approach and tools are customizable and extensible and not constrained to a specific industrial environment.

Our approach works well with component-based development and provides model merging capabilities to support evolution in multi-team environments. We use model fragments as the units of evolution. This allows working with smaller models to reduce complexity during creation, maintenance, and evolution of variability models. As model fragments are of manageable size it is easier to keep them consistent with the architecture, in particular using tool features for detecting changes automatically. While component-based approaches rely on matching provided and required interfaces, our model merging approach relaxes this purely technical view and aims at giving teams more leeway through a lazy consistency approach (indirect references through placeholder elements). It also supports the evolution of different subsystems at different speeds.

We believe that model fragments have two benefits from the perspective of product line variability modeling: (a) They allow to define variability locally and in a bottom-up manner (e.g., by starting at the level of teams or individual subsystems). This was the case in strategy 1 described in our experience section. (b) They allow to define variability at different levels of abstraction and to separate those levels more clearly. This was relevant in strategy 2.

Our industrial experiences have revealed that such an approach must be supported by stable and compatible tools. It should also not introduce more complexity than it removes, as the users are usually reluctant to try new tools and technologies. Our approach can be used to resolve scalability issues of model-driven development, at different levels (teams, subsystems, software features, etc.). Apart from that, it also provides support in reducing the complexity of model-based applications by automating the enforcement of consistency, wherever required.

In future work we plan to develop guidelines helping product line engineers to decide in which model fragment a subsystem’s variability should be modeled. We will work on methods aiding modelers in finding the right granularity for customization decisions in our variability models. We will also explore additional visualization support for different aspects of our approach such as the visualization of interdependencies among model fragments or the visualization of model fragment evolution over time. Finally, we plan to further validate the approach by applying in other domains and development environments.

References


