IT University of Copenhagen

Variability for You

Proceedings of VARY International Workshop affiliated with ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems (MODELS'11)

Øystein Haugen
Krzysztof Czarnecki
Jean-Marc Jezequel
Birger-Møller Pedersen
Andrzej Wąsowski

IT University Technical Report Series **TR-2011-144**

ISSN 1600–6100 **October 2011**

Copyright © 2011, **Øystein Haugen**
Krzysztof Czarnecki
Jean-Marc Jezequel
Birger-Møller Pedersen
Andrzej Wąsowski

IT University of Copenhagen
All rights reserved.

Reproduction of all or part of this work is permitted for educational or research use on condition that this copyright notice is included in any copy.

ISSN 1600–6100

ISBN 978-87-7949-240-1

Copies may be obtained by contacting:

IT University of Copenhagen
Rued Langgaards Vej 7
DK-2300 Copenhagen S
Denmark

Telephone:	**+45 72 18 50 00**
Telefax:	**+45 72 18 50 01**
Web	**www.itu.dk**

Program Committee

Carmen Alonso	Tecnalia
Souvik Barat	Tata Research Development and Design Centre
Danilo Beuche	pure-systems GmbH
Krzysztof Czarnecki	University of Waterloo
Franck Fleurey	SINTEF
Sebastien Gerard	CEA, LIST
Øystein Haugen	SINTEF
Jean-Marc Jezequel	Irisa (INRIA and University of Rennes)
Andreas Korff	Atego Systems GmbH
Vinay Kulkarni	Tata Research Development and Design Centre
Jérôme Le Noir	Thales Research and Technology
Birger Møller-Pedersen	University of Oslo
Ran Rinat	IBM
Suman Roychoudhury	Tata Research Development and Design Centre
Patrick Tessier	CEA/LIST
Michael Wagner	Fraunhofer FOKUS
Andrzej Wąsowski	IT University of Copenhagen

Table of Contents

A Model-Driven Approach for Specifying and Configuring Variability in Business Applications **3**
Souvik Barat, Suman Roychoudhury and Vinay Kulkarni . . . .

Service Variability Meta-Modeling for Service-Oriented Architectures **13**
Mohammad Abu-Matar and Hassan Gomaa . . . .

A Metamodel-based Classification of Variability Modeling Approaches **23**
Paul Istoan, Jacques Klein, Gilles Perrouin and Jean-Marc Jezequel . . . .

Towards Evolution of Generic Variability Models **33**
Andreas Svendsen, Xiaorui Zhang, Øystein Haugen and Birger Møller-Pedersen . . . .

Towards a Family-based Analysis of Applicability Conditions in Architectural Delta Models **43**
Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe and Ina Schaefer . . . .

Complexity Metrics for Software Product Lines **53**
Xiaorui Zhang, Øystein Haugen and Birger Møller-Pedersen . . . .

A Model-Driven Approach for Specifying and Configuring Variability in Business Applications

Suman Roychoudhury, Souvik Barat and Vinay Kulkarni

Tata Research Development and Design Centre,
Tata Consultancy Services,
Pune - 411028, India
{suman.roychoudhury, souvik.barat, vinay.vkulkarni}@tcs.com

Abstract. In our experience, different business systems for the same intent show considerable commonality with well-defined differences. Thus precise adoption of variability modeling and resolution techniques of Software Product Line Engineering (SPLE) can be visualized as a possible solution for delivering such business systems. In our pursuit of adopting standard SPLE concepts within our delivery platform, we are investigating variability modeling techniques such that a purpose-specific business application can be derived by resolving variability in a product line in a consistent and comprehensive manner. Therefore, in this paper, we present a generic metamodel for specifying variability along with a model-2-model (M2M) transformation technique for deriving purpose-specific business solutions from a product line. The approach is furthermore elucidated with an illustrative example that validates the concepts that are described in the paper.

Keywords: Variability Metamodel, Business Application Configuration, Model Transformation.

1 Introduction

Software Product Line Engineering (SPLE) has been in practice for more than two decades and is adopted with varying degree of success by industry practitioners and research organizations in conceptualizing and developing various products and systems. However, adopting SPLE concepts in business application domains, such as those pertaining to banking and insurance domains, are typically limited to product conceptualization space (or problem space), i.e. product offerings are described using feature models [2, 6, and 8]. However, the development artifacts, i.e. artifacts that belong to solution space, are not organized as expected by the SPLE community. In practice, the variability modeling (VM) of solution space is mostly achieved by extending the base metamodel (or language), which is used for describing underlying base models. Therefore the use of variability modeling and approaches pertaining to variability resolutions are specific to a particular domain, e.g., embedded, legacy, mobile, business application. Since the basic principles of variability modeling is common across any domain-specific base model (or language), it is imperative to abstract out the key elements that identify a generic variability model and a uniform approach for resolution (or materialization). In congruence with Common Variability

Language (CVL) standardization initiative [12], we also envisage the need of coming up with such a generic approach for describing variability and resolve them appropriately to derive a purpose-specific solution to address business requirements. Therefore this paper tries to determine the key concepts of variability modeling and resolution mechanism and apply them to an industrial case study application for validation.

The rest of the paper is organized as follows. Section 2 of the paper identifies the key challenges with respect to variability modeling. Section 3 explains our approach and describes in detail the underlying metamodel. Finally, section 4 introduces our illustrative example (case study) and validates the concepts described in the previous sections.

2 Key Challenges

There are several challenges that should be addressed towards designing a variability modeling language and uniform resolution mechanism for business applications. The key concern of designing variability modeling language is to provide an ability to describe the variability of solution space and problem space independently, and establish relationship between these two kinds of models in an intuitive manner. Similarly the key concern of resolution mechanism is to resolve the variability of solution space model without any ambiguity by resolving the variability of problem space model. In congruence with CVL standardization effort, a joint response to CVL RfP [12], the challenges can be described as follows:

C1 – Designing a modeling language for describing the solution space variability, which we term as variability realization metamodel (VRM). The design of the VRM should be such that the domain-specific base metamodel (or language), henceforth base metamodel (BM), such as UML and BPMN should not be extended by any means e.g. additional stereotype or constructs to describe solution space variability. The current practice is to extend the base metamodel to capture variability, but this makes the variability model and thus approach specific to a base model.

C2 - There must be a way to specify the variability of problem space, which we term as variability specification. The use of feature model techniques defined in [2, 6, and 8] have became the de-facto standard for defining variability specification in the problem space. The key challenge in this space is to establish the relationship between variability specification and variability realization, and ensuring the conformance of the constraints defined in two different spaces.

C3 - Finally, a key challenge is to define the resolution semantics that is required to resolve a set of unresolved base model elements to a set of resolved target model elements. The resolution semantics help in the product configuration process and can be specified using model transformation or model composition techniques.

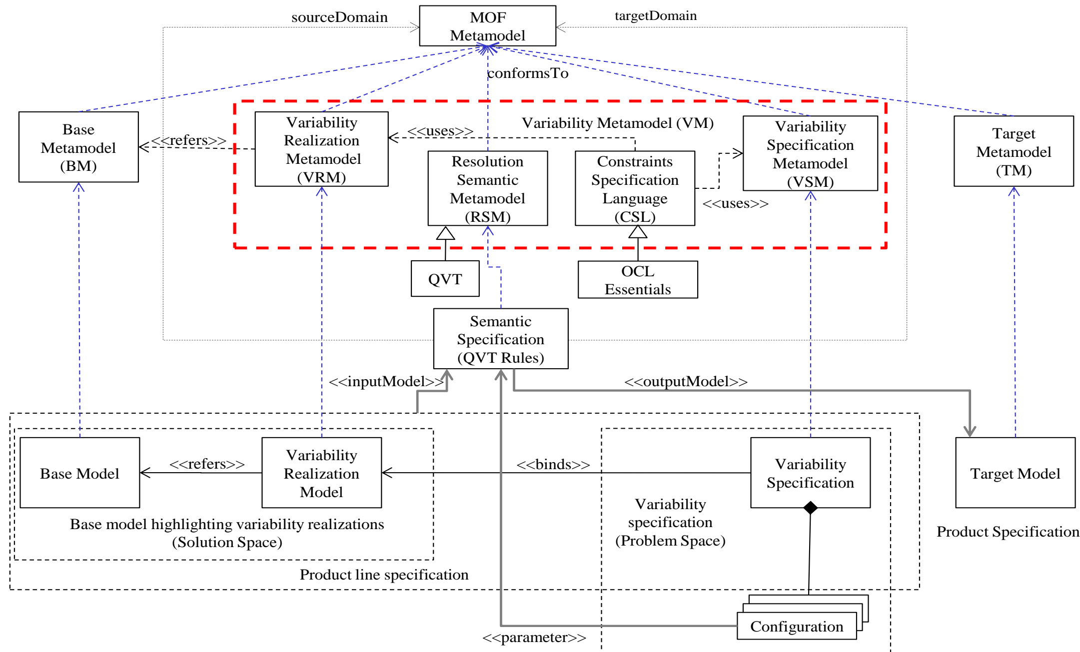

Figure 1: Overview of Variability Modeling and Configuration Approach

3 Approach

In view of the above challenges, Fig 1 presents our high-level approach towards variability modeling. The approach is based on three key concepts – the variability realization metamodel (VRM), the variability specification metamodel (VSM) and the resolution semantic metamodel (RSM), all of which conform to MOF metamodel [13]. Solution space variability can be described by two concepts - Variation Point (VP) and Variant (V). Essentially the VP describes the location where things can differ and variant describes the things that differs. VPs and Vs are captured in the VRM whereas abstract variability concepts like features and configurations are captured within the VSM. In our approach we provide a placeholder to describe the semantic interpretation of a variation point, i.e. how a variation point can be interpreted for a given variant. Essentially, this semantic behavior is captured by M2M transformations that act on the reference model elements of variation points and variants to produce resolved target model. Though the semantics of our variability modeling language is expressed in terms of operational Query View Transformations (QVT) [14], but any model-to-model transformation can be used with our approach. In addition, constraints on VRM and VSM are described by a declarative constraint specification language like OCL [15].

The left-hand-side (LHS) of Fig 1 shows the (unresolved) base metamodel that conforms to MOF. The base (meta-) model elements are referenced by elements in the VRM. Similarly, there is a binding from the VRM to VSM that maps solution space artifacts to abstract variability concepts. The steps are briefly described below:

A) The solution space specification begins with highlighting or annotating the base model. This results in instantiating the VRM with appropriate references (i.e., `<<refers>>`), to the base model. This separates out the base model and

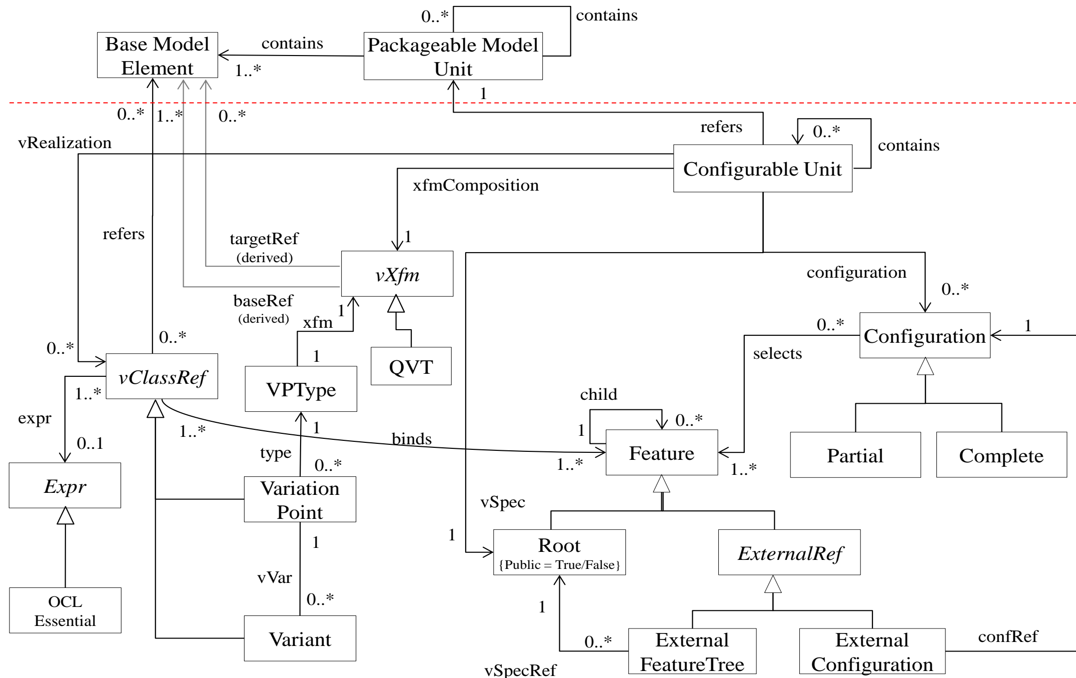

Figure 2: Variability Metamodel

variability realization model. An illustrative example is shown in Figures 3 and 4.

B) The problem space specification begins with the variability specification model (similar to feature tree specification). Steps A and B may be carried out in parallel, and once completed, appropriate bindings (i.e., `<<binds>>`) are provided from the solution space (realization) to the problem space (specification). In addition, a set of valid configurations can be specified on variability specification model. Please refer to Fig 5 for a specific example.

C) Once steps A and B are realized (along with bindings, configurations, and reference), the configuration or materialization process can begin. In this step, the semantic specification (QVT rules) is defined using the product line specification model[1] and a valid configuration (shown as parameter in Fig 1) as input. The semantic transformation rules (QVT rules) generate the target resolved model as output.

Figures 3-6 in the case study section portray the complete process steps as described above.

3.1 Variability Metamodel

The approach described in the previous sub-section is illustrated by the following metamodel as shown in Fig 2. Some of the key elements in the variability metamodel (VM) are described below:

[1] dotted line in Figure 1 showing VRM, BM, VSM along with corresponding bindings and references

Variation Point: A Variation Point (VP) is a placeholder in the VRM where variants can be plugged in. A VP is derived from the variability class reference (VClassRef), which is an instance of the MOF class. Also, VPs refers to base model elements via a reference handler. It is assumed that any base model element is an instance of the MOF class. A VP must have a variation point type (VPType) that captures the behavior of the variation point. In other words, VPType determines how the variation point will be handled by resolution semantics. The metamodel does not make explicit definition of VPType, instead the semantics is specified using QVT transformation rules. In accordance with OMG's ongoing CVL initiative [4], the variation point (type) in our metamodel is similar to opaque variation points.
Variants: Variants can be considered as individual parts that can be plugged into a variation point (with type safety). Variants are the second key component of VRM. Similar to VPs, variants are also derived from VClassRef and conform to MOF class. Constraint expressions on variation points and variants can be defined using OCL. Similar to VP, variants also refer to base model elements via a reference handler.
vXfm: Variability transformation or vXfm signifies transformation applied on a variability class reference (i.e., variations points and variants). They capture the resolution semantics of VM and are expressed in QVT. The QVT rules are used to resolve a target model from unresolved product line input specification.
Feature: A primary constituent of the VSM is a feature or vSpec tree. The top of the tree is denoted by a Root that facilitates in the composition of the tree. A feature tree can be composed of external references, i.e., external feature tree or external configurations (i.e., pre-configured). A feature is an abstract representation and is realized via bindings to concrete concepts like variation points and variants.
Configuration and Resolution: A variability configuration is a set of all valid resolutions from a variability specification tree (i.e., feature tree) whereas variability resolution is the process of resolving a single feature (VP) to a distinct choice (variant) from a set of possible choices (variants). A configuration can be either partial (unresolved resolutions) or complete when all resolutions are resolved.
Configurable Unit: A configurable unit is a reusable entity that can be composed of other configurable units. A CU can be either preconfigured when it contains valid configurations (i.e. a CU without any feature tree) or a CU can be partially configured/ unconfigured when it contains a set of valid configurations and a feature tree. A CU also guides in the composition of vXfms (resolution semantics). This is shown in Fig 2 by the xfmComposition association.

In the following section we validate the concepts described so far with an illustrative example as part of case study.

4 Illustrative Example (Case Study)

In this section, we will evaluate our approach by applying it to a set of banking applications that has both commonality and context-specific variability. The goal is to model the banking applications in the form of UML class diagrams and thereby consider all its variability requirements. Finally, we will configure the UML class diagrams (our base model) and derive a purpose-specific UML class model. Due to

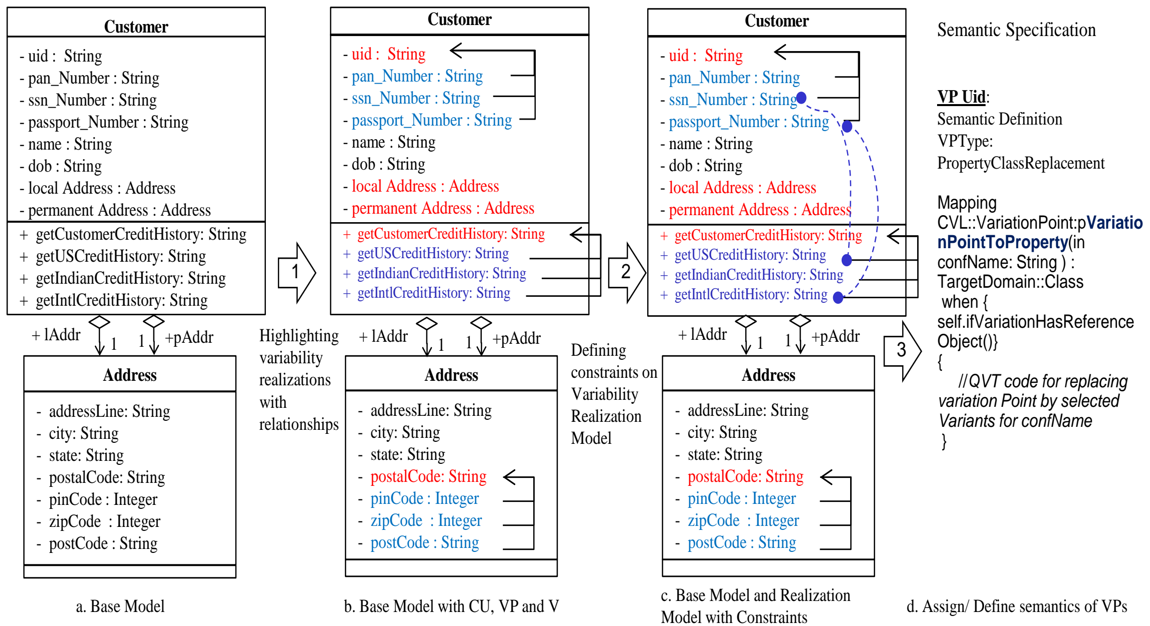

Figure 3: Defining Variability Realization Model

space constraint, we consider only two classes from the core banking application namely the Customer class and the Address class.

In view of the above scenario, let us consider that a bank has many customers and each customer has two addresses – a permanent address and a local address. In addition, a customer have other standard details like customer name, uid (universal identification no.), dob (date of birth) and an operation called getCustomerCreditHistory. Typically, the customer identification number and the address i.e. permanent and local address differ with operational context. For example, a customer located in US is identified by a 10 digit numeric SSN (social security number) and an address field described by a ZIP code, whereas a customer in India is identified by a PAN number (string) and an address represented by a PIN code. Similarly, a Non-Resident Indian (NRI, an Indian citizen living abroad) is identified by his/her passport number (String) and an address field expressed by a Postcode. Just as the properties of a customer model can vary, behavioral operations like getCustomerCreditHostory on a customer can also differ according to the following contexts - US based banks uses Credit Bureau Report to determine credit history, whereas Indian Banks uses CIBIL agency and other Credit Reference Agency Reports to determine credit history for their customers. Thus a product development organization needs to consider the following variability requirements for developing banking related products:

1. **For US Customer**: unique identification number is SSN based, local and permanent addresses is ZIP code based and getCustomerCreditHostory is based on Credit Bureau Report.
2. **For Indian Customer:** PAN based unique identification number, PIN code based local and permanent address and getCustomerCreditHostory is based on CIBIL.
3. **For NRI Customer:** Passport based unique identification number, PIN code based permanent addresses, Postcode based local address and getCustomerCreditHostory is based on Credit Reference Agency Report.

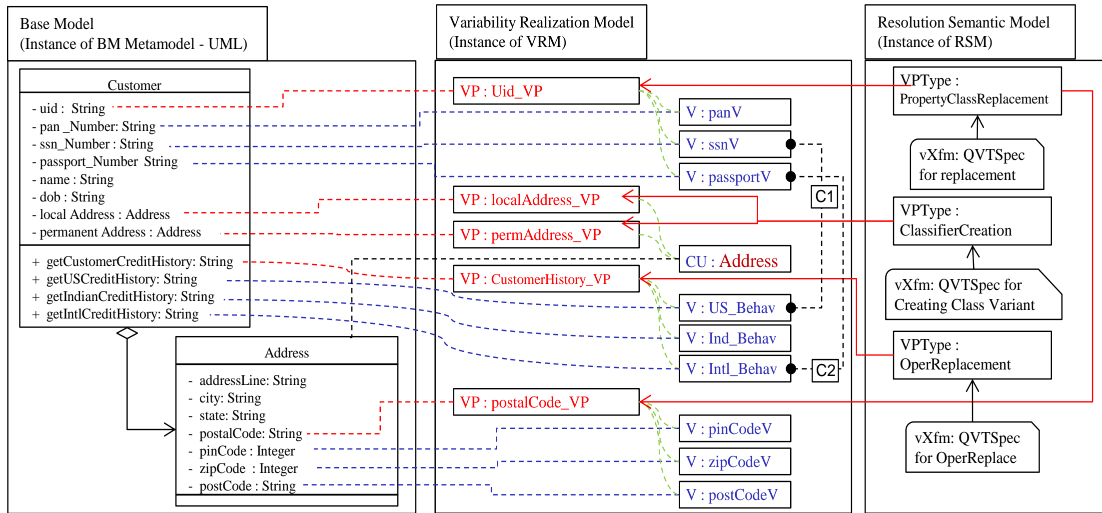

Figure 4: Underlying model of Variability Realization Model

In our experience, the standard practice of a typical product development organization is to create copies for each of these requirements. Such a brute force approach increases the complexity in versioning, change management and configuration management. Instead, a better approach is to explore the possibility of using variability modeling, as described in section 3, to model banking products.

As stated earlier, each of the Customer and Address class model from the banking product line is depicted in Fig 3.a. The class model, which we term as base model, captures the common requirements along with all variability requirements as instance of UML metamodel. For example, the common properties of a Customer class are name and dob, while the variable properties are uid (VP), pan number, ssn and passport number. Similarly, the only variable operation in the Customer class is getCustomerCreditHostory (VP) that maps to several variants like getUSCreditHistory, getIndianCreditHistory and getIntlCreditHistory. Moreover, two properties of the Customer class (permanent and local address) refers to the Address class, which has its own variability requirements as shown in Fig 3.a. The base model is a collection of all common and context-specific variable requirements without any distinction.

The process of realizing variability from a given base model (unresolved) is illustrated in Figures 3-6. The process steps follow the guideline as described in section 3.1. It begins with an unresolved base model (Fig 3.a). This is followed by highlighting or marking the variation points, variants and the relationship between variation points and variants. The above scenario is depicted in Fig 3.b. The figure shows the variation points of Customer class (i.e., uid, localAddress, permanentAddress and getCreditHistory) and Address class (i.e., postalCode). The relationship between variation points (highlighted with red color) and variants (highlighted with blue color) is shown by special multi-tail arrows where the head point to VP and the tail point to variants. In addition, one can define constraints between variation points and variations. For example, ssn_Number must be selected if getUSCreditHistory is selected, while passport_Number must be selected if getIntlCreditHistory is selected. This is depicted in Fig 3.c. Semantic interpretation of each variation points should be specified to complete the variability realization model.

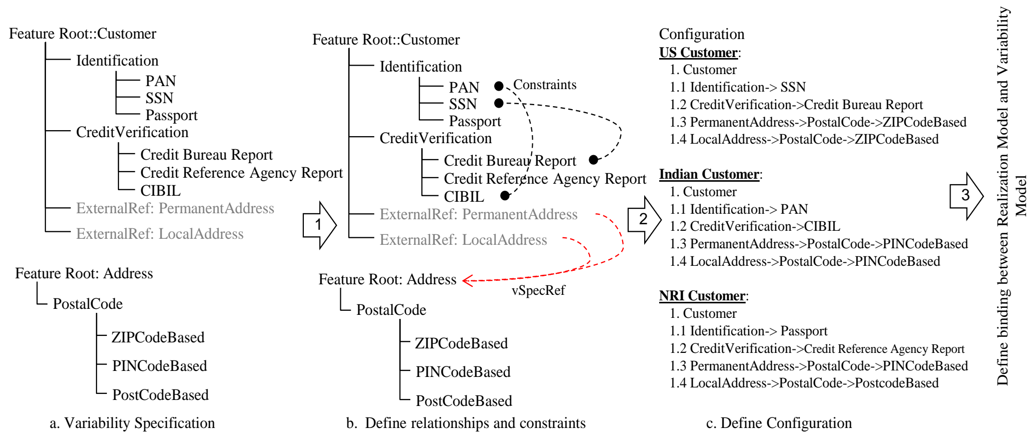

Figure 5 : Defining Variability Specification

As an example, Fig 3.d. shows a QVT specification that replaces a property of a class to another property of the same or different class for a given configuration.

Fig 4 shows the variability realization model along with its appropriate references to the base model. As stated earlier, VRM is independent of the base (meta-) model and refers to the base model elements via reference handlers (dotted red and blue lines). C1 and C2 are the two constraints defined in the realization model. In addition, Fig 4 also shows the semantics model that defines how the VPs would be handled by corresponding variation point types (VPTypes) and QVT rules. The model depicted in Fig 4 describes the model of solution space of the variability requirements for the banking product.

The process of defining variability specification, (problem space, see Fig 5) starts with identifying configurable units or CUs. In our example the two CUs are the Customer CU and the Address CU (Fig 5.a). Note that Customer CU contains Address CU via the external reference as shown in Fig 5.a. Fig 5.b describes the complete variability specification for the Customer feature along with various constraints. Configuration criteria for specifying a US Customer, an Indian Customer or a NRI Customer are depicted in Fig 5.c. Once the variability specification model is defined, bindings from the abstract specification model to concrete realization model must be accomplished. The binding process is illustrated in Fig 6 that shows how VPs and variants from the realization model are bound to various choices or features in the feature tree. Once all the above steps are completed, the configuration process can derive a purpose specific base model by applying appropriate M2M transformations

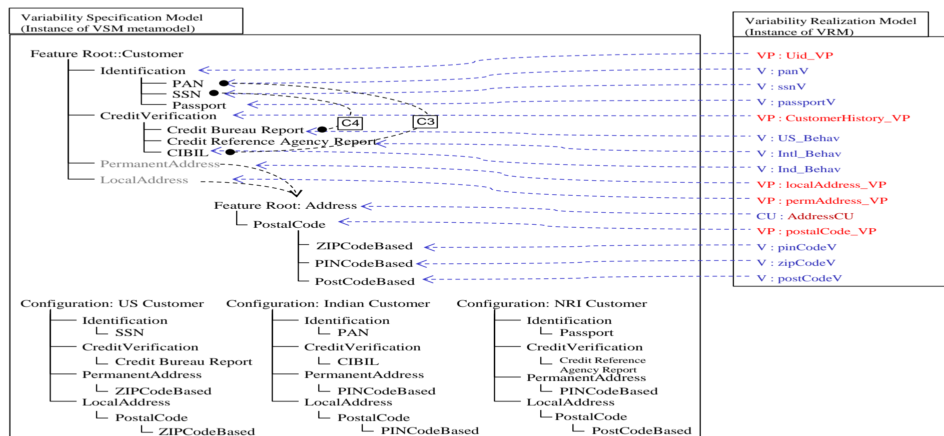

Figure 6: Underlying models of Variability Specification

on the input product line specification.

5 Related Work

There are several approaches that address different aspects of managing variability in software product lines. These aspects can be divided into three broad categories – a) approach for defining solution space variability, b) approach of defining problem space variability along with the mechanism to establish relationships with solution space models, and c) approach for defining the resolution semantics and resolutions.

In the first category (solution space), a proposal for modeling variability in software families with UML using the standardized extension-mechanisms of UML (using stereotype) is presented in [3]. A variation point model that allows user or application engineer to extend components at pre-specified variation points is proposed in [9]. A conceptual model for capturing variability in a software product line is presented in [1]. On the similar line, the extension of base metamodel using UML stereotype is presented in [10] to model variability. In the solution space, we have proposed a MOF compliant VRM that establishes an association with any MOF compliant base model (e.g., UML class model) instead of extending base metamodel. This provides a clear separation of concern and helps to define variability of any MOF compliant models.

In the second category (problem space), existing approaches for specifying variability are essentially based on one of the following approaches [2, 6, 8] or a combination of them. A concise representation of variability specification for different kinds of models is presented in [5]. In addition to variability specification, this paper tried to provide semantics of features by mapping them to base models using a template based approach. Our approach for defining variability specification is essentially based on approach presented in [6], however we unify the key concepts of variability specification metamodel with variability realization metamodel to establish bindings between them.

The third category (resolution semantics) uses model transformation techniques for configuring product line [7]. Essentially, there are two broad categories of resolution technique – model transformation based on pre-defined M2M transformation rules or model composition based on AOP-like technique. Composition approaches such as [11. 16 and 17] are AOP based. In our view, a limitation of AOP-like composition is that they are useful for handling crosscutting concerns, whereas other concerns may not be composed cleanly. Instead, our approach uses the concept of transformation based semantic composition. This enables customized semantics for each variation point to be composed by any M2M transformation language like QVT.

6 Conclusion

To specify and configure variability in business application product lines, we argued, the need for: i) a realization layer to specify concrete variability concepts, ii) a specification layer to indicate abstract variability concepts iii) appropriate bindings and reference from the realization layer to the base model and abstract concepts iv) a

mechanism to resolve variability using M2M transformation techniques (resolution semantics). We presented our solution that aims to address all the three challenges and shared early experience of using it in practice. Moreover, we have tried to align our approach with the ongoing OMG initiative in defining a common variability language [12] and plan to apply the standard throughout our delivery platform in future.

References

1. Bachmann, F., Goedicke, M., Leite, J., Nord, R., Pohl, K., Ramesh, B., and Vilbig. A.: A Meta-model for Representing Variability in Product Family Development. Software Product Family Engineering, volume 3014 of LNCS, pp. 66-80, Springer, 2004.
2. Batory, D.: Feature Models, Grammars, and Propositional Formulas. Software Product Lines, Volume 3714 of LNCS, pp. 7-20, Springer, 2005.
3. Clauß, M., Jena, I.: Modeling variability with UML. GCSE 2001Young Researchers Workshop, 2001
4. Common Variability Language Initiative: http://www.omgwiki.org/variability/doku.php
5. Czarnecki, K., and Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. Generative Programming and Component Engineering, Volume 3676 of LNCS, pp. 422–437. Springer, 2005.
6. Czarnecki, K., and Eisenecker. U.: Generative programming methods, tools and applications, Addison-Wesley, 2000.
7. Deelstra, S., Sinnema, M., Jilles V. G., Bosch, J.: Product derivation in software product families: a case study, Journal of Systems and Software, v.74 n.2, p.173-194, 15 January 2005
8. Kang, K., Kohen, S., Hess, J., Novak. W., and Peterson. A.: Feature-orientation domain analysis feasibility study, Technical Report, CMU/SEI-90TR-21, November 1990.
9. Gomaa, H., Webber, D. L.: Modeling Adaptive and Evolvable Software Product Lines Using the Variation Point Model. 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Volume 9.Page: 90268.3
10. Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Solberg, A.: An MDA®-based framework for model-driven product derivation. IASTED Conf. on Software Engineering and Applications 2004: 709-714
11. Mezini, M., Ostermann, K.: Variability Management with Feature-Oriented. Programming and Aspects. SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering, page 127-136. NY, USA, ACM, (2004).
12. OMG RFP – Common Variability Language (CVL) RFP: http://www.omg.org/techprocess/meetings/schedule/Common_Variability_Language_%28CVL%29_RFP.html
13. OMG Document (OMG document number formal/2006-01-01): Meta Object Facility (MOF) - Version 2.0. http://www.omg.org/spec/MOF/2.0/.
14. OMG Document (OMG document number - formal/2011-01-01): Meta Object Facility (MOF) 2.0 Query/View/Transformation, v1.1. http://www.omg.org/spec/QVT/1.1/
15. OMG Document (OMG document number - formal/2010-02-01): Object Constraint Language (OCL), Version 2.2. http://www.omg.org/spec/OCL/2.2/.
16. Sanchez, P., Fuentes, L., Loughran, N.: A Metamodel for Designing Software Architectures of Aspect-Oriented Software Product Lines. AMPLE Project (http://ample.holos.pt/) deliverable D2.2, September 2007.
17. Völter, M. and Groher, I.: Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In Proceedings of SPLC. 2007, 233-242.

Service Variability Meta-Modeling for Service-Oriented Architectures

Mohammad Abu-Matar, and Hassan Gomaa

Department of Computer Sceience
George Mason University, USA
{mabumata, hgomaa}@gmu.edu

Abstract. Service Oriented Architecture (SOA) has emerged as a paradigm for distributed computing that promotes flexible deployment and reuse. However, SOA systems currently lack a systematic approach for managing variability in service requirements. Our paper addresses this problem by applying software product line (SPL) concepts to model SOA systems as service families. We introduce an approach to model SOA variability with a multiple-view SOA variability model and a corresponding meta-model. The approach integrates SPL concepts of feature modeling and commonality/variability with different service views using UML and SoaML. This paper describes a multiple-view meta-model that maps features to variable service models as well as model consistency checking rules. We describe how to derive family member applications and also present a validation of the approach.

Keywords: Meta-Modeling, Software Product Lines, SOA, Feature Modeling

1 Introduction

Service Oriented Architecture (SOA) has emerged as an architectural style for distributed computing that promotes flexible deployment and reuse [1]. However, SOA systems currently lack a systematic approach for managing variability and are typically platform-dependent. Since services in SOA could be used by different clients with varying functionality, we believe that SOA variability modeling can benefit from software product lines (SPL) variability modeling techniques.

This paper describes a meta-modeling approach that integrates SPL concepts of feature modeling and commonality/variability to model SOA variability. The main goal of SPL is the reuse-driven development of SPL member applications by using reusable assets from all phases of the development life cycle. This goal is similar to the goal of SOA where flexible application development is a common theme.

Our approach integrates feature meta-modeling [2], [3] with service views using UML and SoaML the newly released SOA standardized modeling language. Such an approach facilitates variability modeling of service family architectures in a systematic and platform independent way.

At the heart of the approach is a meta-model that describes requirements and architectural views of service oriented systems. In addition, the meta-model describes variability in the service views and adds a feature view that addresses the variability in the SOA system. The meta-model also describes relationships among the services views and among the feature and services views. Our approach builds on previous research as follows: feature modeling of software product lines engineering [2], meta-

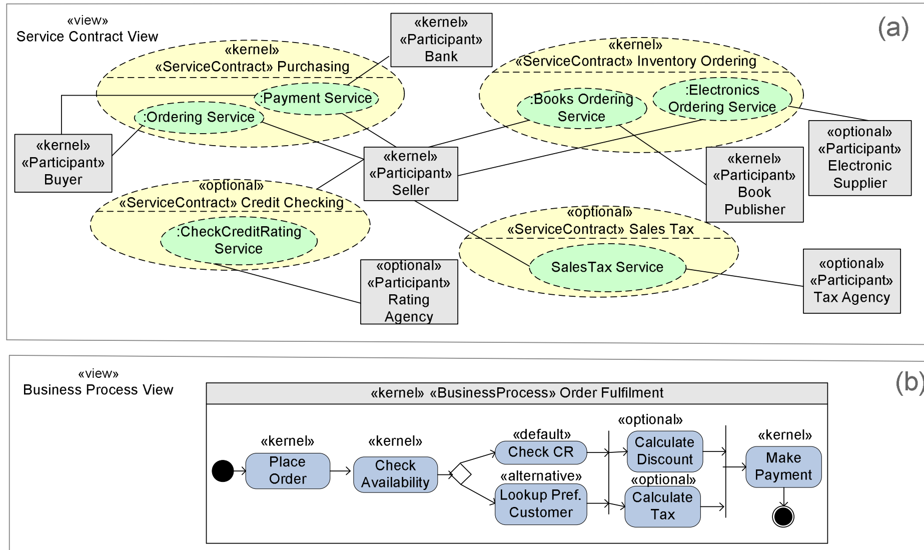

Fig. 1 E-Commerce Requirements Service Variability Views

modeling of SPL phases [4], software adaptation patterns for SOA systems [5], an early version of our meta-model [6], and SoaML.

The rest of the paper is structured as follows. Section 2 briefly present our multiple view variability model, section 3 details the description of our multiple view variability meta-model, section 4 describes feature to service relationships and constraints, section 5 presents validation of the approach, section 6 presents related work, and section 7 concludes the paper.

2 Multiple View Service Variability Model

Erl [1] describes service-oriented systems as having multiple perspectives where these perspectives depend on each other. In essence, each perspective describes a distinct view of the whole SOA system. In this paper, the different SOA perspectives are formalized into multiple Requirements and Architectural views. In this section, we briefly describe our multiple-view service model which is formalized by our meta-model. Each view of the multiple view model is depicted by a UML diagram that is extended by using stereotypes. In particular, each modeling meta-class is depicted using two stereotypes, one to represent an SOA concept and the other to represent a commonality/variability concept. A service modeling example is introduced in this section (Fig. 1, 2, and 3) and used throughout the paper to explain our approach.

The Service Contract Variability View is a Requirements view that describes service contracts, which are prescribed by collaborating organizations in order to govern and regulate their interactions. Service contracts (Fig. 1a) are modeled by SoaML's ServiceContract element. This view also contains SoaML's Participant elements that model providers or consumers of services. An example of the Service Contract View is given in Fig. 1a which models an E-Commerce SPL. We categorize Service Contracts and Participants as kernel, optional, or alternative. Kernel elements

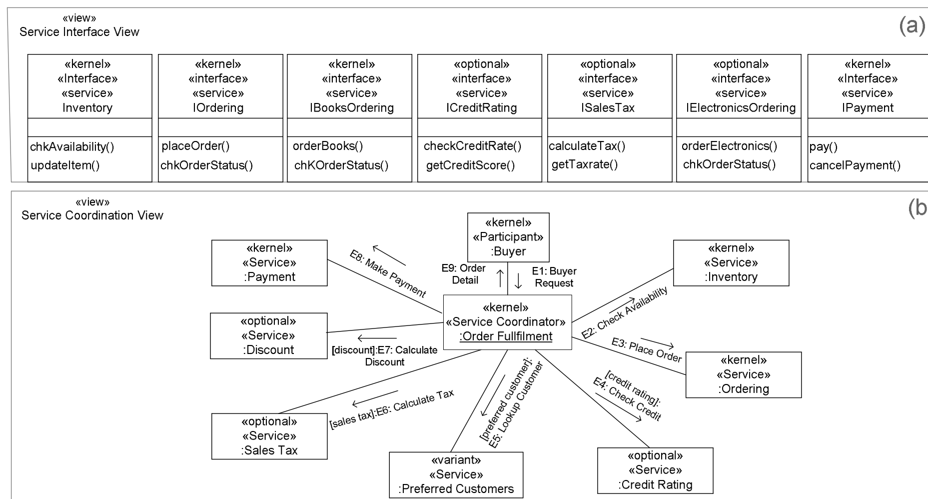

Fig. 2 E-Commerce Architectural Service Variability Views

are required by all members of an SPL, whereas optional elements are required by only some members. Alternative elements are required by different SPL members.

The Business Process Variability View is a Requirements view that models the workflow of business processes. We use UML Activity diagrams to model this view with variability stereotypes (Fig. 1b).

Services expose their capabilities through interfaces only. The Service Interface Variability View is an architectural view that models service interfaces by using UML's Interface class in addition to applying a <<service>> stereotype (Fig. 2a). Service interfaces are categorized as kernel, optional, and variant.

The Service Coordination Variability View is an architectural view that models the sequencing of service invocations. Services should be self-contained and loosely coupled in order to have a high degree of reuse; dependencies between services should therefore be kept to a minimum [5]. Hence, coordinators are used in situations where access to multiple services needs to be coordinated and/or sequenced. The Service Coordination View consists of Coordinators which are modeled as classes with a <<Service Coordinator>> stereotype (Fig. 2b). Service Coordinators are categorized as kernel, optional, and variant.

With the above service modeling views, it is possible to define the variability in each view and how it relates to other views. However, it is difficult to get a complete picture of the variability in the service architecture because it is dispersed among the multiple views. The Feature View is a unifying view that focuses on service family variability and relates this to the other service views. Feature modeling is rooted in the seminal work of Kang et al. [3]. Feature models are used to manage similarities and differences among family members in a SPL. Features are analyzed and categorized as common, optional, or alternative. Related features can be grouped into feature groups, which constrain how features are used by a SPL member. Fig. 3 depicts the feature model for the E-Commerce product line.

3 Multiple View Service Variability Meta-Modeling

The multiple-view variability modeling approach is based on a meta-model that precisely describes all views and views relationships. Each view in the multiple-view model (Fig. 1, 2) is described by a corresponding meta-view in the meta-model (Fig. 4). There are two Requirements meta-views, Contract and Business Process, and two Architecture meta-views, Service Interface and Service Coordination. To get a full

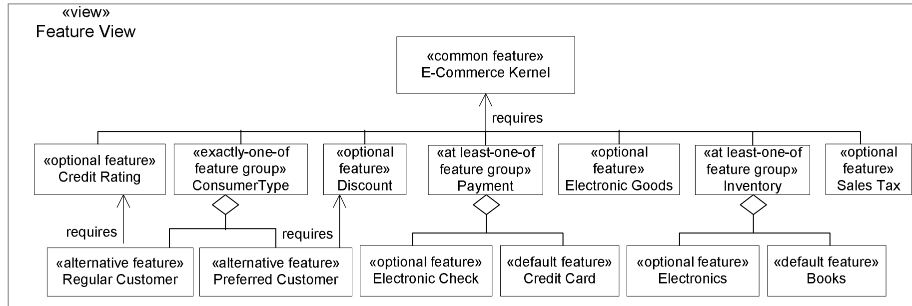

Fig. 3. E-Commerce Feature View

understanding of the variability in service architectures, it is necessary to have one view that focuses entirely on variability and defines dependencies in this variability, which is the purpose of the feature meta-modeling view described in Section 3.5. Our meta-modeling approach builds on previous work in SPL multiple-view modeling and meta-modeling [4].

3.1 Service Contract Meta-View

We use SoaML's ServiceContract meta-class to specify the agreement between providers and consumers, by using the <<ServiceContract>> stereotype. To model SOA variability, we categorize a ServiceContract as kernel, optional, or alternative.

Each service contract (Fig. 4) prescribes roles for the organizations participating in it. This view also models contract participants, which are entities that abide by service contracts. We use SoaML's Participant meta-class which specifies providers or consumers of services. This meta-class extends the UML Class meta-class by using the <<Participant>> stereotype.

3.2 Business Process Meta-View

Neither SoaML nor UML explicitly model business process workflow. Since a business process is composed of a sequence of activities, we use UML Activity meta-classes, as part of an activity diagram for each business process.

3.3 Service Interface Meta-View

We model service interfaces by UML's Interface meta-classes accompanied with a <<service>> stereotype to distinguish them from component interfaces. Interface meta-classes specify provided and required service interfaces. A service interface is categorized as kernel, optional, or variant.

It should be noted that SoaML has a ServiceInterface meta-class that describes service interfaces in addition to service interactions and protocols. However, the UML interface meta-class is suitable for our current research.

3.4 Service Coordination Meta-View

The service coordination view consists of coordinators which are modeled as classes with a <<Service Coordinator>> stereotype. Service coordinators, depicted on UML communication diagrams, interact with clients and services. The sequencing of

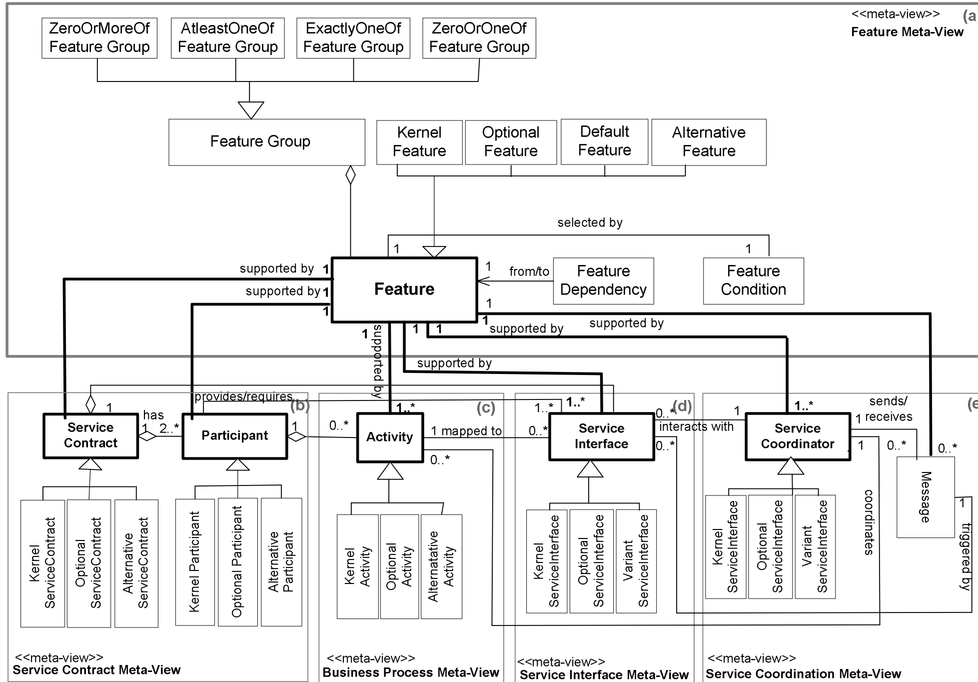

Fig. 4. Service Variability Meta-Model

service invocations is encapsulated within the Coordinator . Service Coordination is categorized by type of coordination (independent, distributed, or hierarchical) and degree of concurrency (sequential or concurrent) [5].

3.5 Feature Meta-View

Since UML has no native support for feature modeling, we use a UML based feature meta-model based on our previous work [2], [4]. Fig. 3 depicts a feature meta-model. Features are specialized into kernel, optional, alternative, and default depending on the characteristic of the requirements as described in section 2.

Kernel features are requirements common to all members of the SPL. Optional features are required by only some members of a SPL. An alternative feature is an alternative of a kernel or optional feature to meet a specific requirement of some members. A default feature is the default choice among the features in a feature group. Feature groups refer to constraints on the selection of a group of features (e.g., preventing selection of mutually exclusive features). Feature dependencies represent relationships between features.

4 Service Variability Meta-Model Relationships

In this section, we describe the relationships of the service variability meta-model (Fig. 4) that ties all the aforementioned views together. The meta-model consists of 5 meta-views (4+1 feature view) that correspond to each view in the multiple-view model (section 2). The Feature View (Fig. 3) unifies the service views as explained in

Section 2.5. The meta-model describes both intra-view and inter-view relationships, as follows:

The Intra-View Relationships describe associations and dependencies inside each view. A **ServiceContract** meta-class, in the Service Contract view, is associated with two or more **Participant** meta-classes (Fig. 1a), because a **ServiceContract** meta-class defines the rules for participating entities in the SOA system. The **ServiceCoordinator** meta-class in the Service Coordination view is associated with a **Message** meta-class as it sends/receives messages to/from services.

The Inter-View Relationships describe associations and dependencies between different service views. A **ServiceContract** meta-class is associated with one or more **ServiceInterface** meta-classes (Fig. 4b, d). **Participant** meta-classes provide or require service **Interface** meta-classes (Fig. 4b, d), because participating entities only interact through interfaces to minimize coupling among services. **Participant** meta-classes (Fig. 4b) may define their own internal business processes (Fig. 4c). **Activity** meta-classes (Fig. 4c) can be either *local* or *service* meta-classes. Local activities are executed within the **Participant** execution environment. Service activities require **ServiceInterfaces**. **ServiceCoordinator** meta-classes (Fig. 4e), in the Service Coordination View coordinate service invocations based on the workflow of **Activity** meta-classes in the Business Process view (Fig 4c). **Message** meta-classes in the Service Coordination View (Fig. 4e) trigger operation invocations on the service **Interfaces** in the Service Interface View (Fig. 4d).

Meta-classes in one view of the service model affect meta-classes in other views. For example, in Fig. 1b, when the Calculate Tax **Activity** is added to the Order Fulfillment Business Process View, a Sales Tax **ServiceContract** is introduced into the E-Commerce SPL in the Service Contract View (Fig. 1a). Consequently, a Tax Agency **Participant** is also added which provides a SalesTax service **Interface** in the Service Interface View (Fig. 2a).

Feature to Service Meta-Views Relationships describe relationships between the Feature View and Service views. In addition, we provide consistency checking rules, written in OCL, that add explicit constraints on relationships between the meta-classes of the multiple-view service variability meta-model (Fig. 4).

4.1 Feature to Service Contract Meta-View Relationship

A **Feature** (Fig. 4a) is associated with one or more ServiceContract meta-classes in the Service Contract View (Fig. 4b). The variability stereotype on a ServiceContract dictates the type of feature it may map to. For instance, an optional feature (e.g., Credit Rating) can only map to optional service contracts (e.g., Credit Checking service contract).

A Kernel ServiceContract can only support a kernel Feature

```
context Feature inv: reuseStereotype = 'kernel' implies
servicecontract->size() >= 1 and servicecontract.reuseStereotype =
'kernel'
```

A **Feature** is associated with one or more **Participants**. For example, if the *Electronic Goods* optional feature (Fig. 3) is selected, the *Seller* will sell electronic items in addition to books and the *ElectronicSupplier* **Participant** will participate in the *InventoryOrdering* **ServiceContract** (Fig. 1a). Consequently, the

ElectronicsOrdering **ServiceInterface** will be introduced into the *InventoryOrdering* **ServiceContract** (Fig. 1a). Hence, the selection of one feature meta-class in the feature meta-view is mapped to two service meta-classes (contract and interface) in the contract and interface meta-views.

4.2 Feature to Business Process Meta-View Relationship

A Feature is associated with one or more Activities in the Participant's business process (Fig. 4c). For example, when the Discount optional feature is selected (Fig. 3), which means that the system changes to provide the 'Discount' capability, the 'Calculate Discount' Activity is added to the Order Fulfillment business process (Fig. 1b). Thus, the Discount <<optional feature>> is mapped to <<optional>> 'Calculate Discount' Activity in the business process view.

An optional Activity can only support an optional Feature

```
context Feature inv: reuseStereoType = 'optional' implies
activity->size() >=1 and activity.reuseStereoType = 'optional'
```

4.3 Feature to Service Interface Meta-View Relationship

A Feature is associated with one or more service Interfaces. For example, if the Credit Rating optional feature is selected (Fig. 3), the Seller Participant has to provide a new service Interface that can interact with a credit rating agency. Thus, the Credit Rating <<optional feature>> is mapped to <<optional>> Credit Rating service Interface in the Service Interface View (Fig. 2a).

A variant ServiceInterface can only support an alternative Feature

```
context Feature inv: reuseStereoType = 'alternative' implies
serviceinterface->size() >= 1 and serviceinterface.reuseStereoType =
'variant'
```

4.4 Features to Service Coordination Meta-View Relationship

A **Feature** is associated with one or more ServiceCoordinator meta-classes in the Service Coordination View. For example, since the Order Fulfillment feature (Fig. 3) is supported by the Order Fulfillment Activities in the business process view (Fig. 1b), the same feature is supported by the Order Fulfillment ServiceCoordinator in the Service Coordination view (Fig. 2b). It should be noted that each business process is associated with a unique ServiceCoordinator.

A **Feature** is associated with one or more **Message** meta-classes. For example, the 'Preferred Customer' optional feature (Fig. 3) is supported in part, by the 'Lookup Customer' **Message** in Fig. 2b.

4.5 Service Variability Meta-Model Consistency Checking Rules

In this sub-section, we provide representative consistency checking rules to precisely describe the relationships among the variable service meta-model meta-classes in Fig. 4. We are inspired by our previous work [4] where we used OCL to describe consistency checking rules to describe the relationships among the various meta-modeling views of the SPL phases.

The following are typical meta-modeling consistency checking rules, which are expressed in both English and OCL.

1. *A kernel ServiceContract must have at least 2 kernel Participants*
```
context servicecontract inv: reuseStereotype ='kernel' implies
(select participant.reuseStereotype = 'kernel')->size() >= 2
```
2. *A kernel ServiceContract must be supported by at least one kernel ServiceInterface*
```
context servicecontract inv: reuseStereotype ='kernel' implies
serviceinterface->exists(si | si.reuseStereotype = 'kernel')
```
3. *A Participant must provide or require at least one ServiceInterface*
```
context participant inv: reuseStereotype ='kernel' implies
serviceinterface->exists(si | si.reuseStereotype = 'kernel')
```
4. *If kernel Activity is a Service Activity, it must call a kernel ServiceInterface.*
```
context activity inv: self.oclIsKindOf(Service) implies
activity.serviceinterface.reuseStereotype = 'kernel'
```

5 Validation of the Approach

To validate our approach, we created a proof-of-concept prototype for service oriented SPL. The prototype allows users to specify feature models, build service models, relate features to service views, and create SPL member applications. The purpose of the validation is to evaluate our approach with regard to:

1. The multiple views of the service oriented product line are consistent with each other.
2. The multiple-view service variability model is compliant with the underlying multiple-view service variability meta-model
3. Derived service oriented member applications are consistent with the service oriented SPL requirements and architectural models.

The prototype is based on the open-source Eclipse Modeling Framework (EMF). The prototype relies on Eclipse's plug-in mechanisms to provide integrated functionality for users. The prototype consists of the following components:

- EMF core modeling facilities.
- Apache ODE – ODE is an open source BPEL engine. The generated BPEL code is compiled and deployed to ODE. The BPEL code invokes services based on WSDL files.
- Apache CXF – CXF is an open-source web-services framework which supports standard APIs such as JAX-WS and JAX-RS as well as WS standards including SOAP, and WSDL.
- Eclipse Swordfish – Swordfish is an open-source extensible Enterprise Service Bus (ESB).

By building the E-Commerce SPL feature and multiple view service models correctly, i.e. without errors emitted from the underlying OCL rules, we validated that multiple views of the service oriented product line are consistent with each other. In addition, we validated that the multiple-view E-Commerce SPL model is compliant with the underlying multiple view variability meta-model, because EMF ensures the compliance of models by applying the underlying meta-model syntax rules.

We perform manual derivation of the E-Commerce SPL member applications, as described in [7] because automation of this capability is still in progress.

6 Related Work

There have been several approaches for modeling variability in SOA. This section discusses related work and examines them in light of our work.

Chang and Kim in [8] add variability analysis techniques to an existing service oriented analysis and design method (SOAD). Decision tables are used in [8] to record variability types in each phase of the SOAD process.

Topaloglu and Capilla [9] present architectural pattern approaches to model variation points in Web Services. Gomaa and Saleh [10] present an SPL engineering approach based on Web Services.

Capilla and Topaloglu [11] advocate an SPL engineering approach that has a specific phase for service composition in the SPL architecture. They introduce several variation points that can be used to customize the SPL during service selection. However, the authors do not tie service selection to the features required in the SPL.

In [12], the authors used the concept of features to solve variability problems for SOA. However, the authors' approach assumes the availability of service implementation code, which is not the norm in most SOA scenarios.

Park et al. [13] suggest a feature-based reusable domain service development approach to create reusable domain services. However, the approach in [13] above does not consider the relationships between features and services.

It should be noted that our research addresses design-time variability and not runtime SOA variability issues. Our previous work on dynamic adaptation has addressed some issues of runtime adaptation in SOA [5].

7 Conclusions

In this paper, we described a multiple-view meta-modeling approach that addresses service oriented variability concerns in a unified and platform independent manner. In particular, we described the integration of SPL concepts of feature meta-modeling and commonality/variability analysis with service views using UML and SoaML. We validated our approach by developing a proof-of-concept prototype, which we used to build a multiple view E-Commerce service oriented product line.

We believe that our approach has several benefits:

- Treatment of SOA variability concerns in a unified, systematic, multiple-view variability meta-model.
- A Multiple view meta-model for service oriented product lines.
- OCL Consistency checking rules that can be used with any UML/EMF environment.
- Facilitates variability modeling of service families in a platform independent way. For example, our approach does not restrict the representation of service interfaces to WSDL or restrict business workflows execution to BPEL.
- Applied feature modeling techniques to manage variability in SOA.
- Extended SoaML with variability modeling notation.
- Different service variants are explicitly modeled in the approach, thus maximizing reusability.
- A proof-of-concept prototype to validate our approach.

In our ongoing research, we are building on our existing research to introduce a service variability mediation layer to further decouple service providers and consumers. In addition, we intend to provide a feature-based discovery and

composition of service-oriented SPL. Finally, we are adding MDA concepts to our framework in order to automate the derivation of service member applications.

References

[1] T. Erl, *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall, 2005.

[2] H. Gomaa, *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison-Wesley Professional, 2004.

[3] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson, *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. 1990.

[4] H. Gomaa and M. E. Shin, "Multiple-view modelling and meta-modelling of software product lines," *IET Software*, vol. 2, no. 2, pp. 94-122, Apr. 2008.

[5] H. Gomaa, K. Hashimoto, M. Kim, S. Malek, and D. A. Menascé, "Software adaptation patterns for service-oriented architectures," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, New York, NY, USA, 2010, pp. 462–469.

[6] Abu-Matar, M., Gomaa, H., Kim, M., and Elkhodary, A.M., "Feature Modeling for Service Variability Management in Service-Oriented Architectures," in *SEKE(2010)*, 2010, pp. 468-473.

[7] M. Abu-Matar and H. Gomaa, "Feature Based Variability for Service Oriented Architectures," in *The 9th Working IEEE/IFIP Conference on Software Architecture*, Boulder, Colorado, USA, 2011.

[8] S. H. Chang and S. D. Kim, "A Service-Oriented Analysis and Design Approach to Developing Adaptable Services," in *Services Computing, IEEE International Conference on*, Los Alamitos, CA, USA, 2007, vol. 0, pp. 204-211.

[9] N. Y. Topaloglu and R. Capilla, "Modeling the Variability of Web Services from a Pattern Point of View," in *Web Services*, vol. 3250, L.-J. (LJ) Zhang and M. Jeckle, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 128-138.

[10] H. Gomaa and M. Saleh, "Software product line engineering for Web services and UML," in *Proceedings of the ACS/IEEE 2005 International Conference on Computer Systems and Applications*, Washington, DC, USA, 2005, p. 110–vii.

[11] R. Capilla and N. Y. Topaloglu, "Product Lines for Supporting the Composition and Evolution of Service Oriented Applications," in *Principles of Software Evolution, International Workshop on*, Los Alamitos, CA, USA, 2005, vol. 0, pp. 53-56.

[12] S. Apel, C. Kaestner, and C. Lengauer, "Research challenges in the tension between features and services," in *Proceedings of the 2nd international workshop on Systems development in SOA environments*, New York, NY, USA, 2008, pp. 53–58.

[13] J. Park, "An approach to developing reusable domain services for service oriented applications," New York, NY, USA, 2010, pp. 2252–2256.

A Metamodel-based Classification of Variability Modeling Approaches *

Paul Istoan[1], Jacques Klein[2], Gilles Perouin[3], and Jean-Marc Jézéquel[4]

[1] CRP Gabriel Lippmann, Luxembourg - LASSY, University of Luxembourg, Luxembourg-Université de Rennes 1, France,
[2] SnT - University of Luxembourg, Luxembourg, Luxembourg
[3] PRECISE, University of Namur, Namur, Belgium
[4] IRISA, Université de Rennes 1, France
istoan@lippmann.lu, gilles.perrouin@fundp.ac.be,
jacques.klein@uni.lu, jezequel@irisa.fr

Abstract. Software Product Line Engineering (SPLE) is an emerging paradigm taking momentum that proposes to address flexibility and shorter time-to-market by maximizing software reuse. The key characteristic of SPLE is the effective modelling and management of variability, for which a number of Variability Modeling (VM) techniques have been developed during the last two decades. Therefore, understanding their commonalities and differences is important for selecting the most suitable technique. In this paper, we propose a metamodel-based classification of VM techniques gathered through a survey of relevant literature.

Keywords: Variability Modeling Approaches, Model-Driven Engineering, Survey

1 Introduction

Constant market evolution triggered an exponential growth in the complexity and variability of modern software solutions. It is frequently the case that software development is actually a redevelopment process, with many products being partially built before. *Software Product Lines* (SPL), or *software families*, are rapidly emerging as an important and viable software development paradigm designed to handle such issues [34]. Use of SPL approaches has allowed renowned companies like Hewlett-Packard, Nokia or Motorola to achieve considerable quantitative and qualitative gains in terms of productivity, time to market and customer satisfaction [1]. Their increasing success relies on the capacity to offer software suppliers/vendors *ways to exploit the existing commonalities in their software products*. SPL engineering focuses on capturing the *commonality* and *variability* between several software products [12]. This new concept started to draw the attention of the software community when software began to be massively integrated into hardware product families, with cellular phones [28] probably being the most well known example. More generally, automotive systems, aerospace or telecommunications are some of the areas targeted by SPL research.

* This work has been funded by the SPLIT project (FNR + CNRS, FNR/INTER/CNRS/08/02), the IAP Programme of the Belgian State, Belgian Science Policy (MoVES project) and the Walloon Region (NAPLES project) and the FNR CORE project MITER (C10/IS/783852)

Variability is seen as the key feature that distinguishes SPL engineering from other software development approaches [9]. In common language use, the term *variability* refers to "the ability or the tendency to change". It is a central concern in SPL development [19] and covers the entire development life cycle, from requirements elicitation to product testing. When talking about *SPL variability*, two concepts immediately stand out [23]: *commonalities* (assumptions true for each family member) and *variabilities* (assumptions about how individual family members differ). Variability management is thus growingly seen as being complex process that requires increased attention.

A traditional way used by scientists to master the increasing complexity and variability of real-world phenomena is to resort to *modelling*. In software engineering, *models* allow to express both problems and solutions at a higher abstraction level than code [24]. Model Driven Engineering (MDE) treats *models* as first-class elements for application development. The goal of MDE is to reduce design complexity and make software engineering more efficient by shifting the focus from implementation to modelling. *Models* are created based on concepts defined in a *meta-model*, which defines the concepts, relationships and (static) semantics of a domain. The relation between a model and its meta-model is defined as a conformity relation.

In recent years, several variability modelling techniques have been developed, aiming to explicitly and effectively represent SPL variability. The existing differences between them render each method unique, suitable for a particular domain and in a specific context. Hence the question of *which approach is the most suitable with respect to a particular context?* is of great interest to SPL engineers. There is a stringent need to extract, synthesize and analyse in a critical manner the research literature on SPL variability modelling. A review of all contributions related to this topic, outlining the individual characteristics of each method and possibilities of improvement, can facilitate and guide SPL engineers in the selection of a particular technique suitable for their specific development context. Furthermore, such a comparative analysis can provide practitioners with a qualified portfolio of available techniques and therefore play an important role in the transfer of knowledge from research to industry. In this context, this paper addresses the following research questions:

– RQ1. How can variability be modelled in SPLs?
– RQ2. How can existing techniques be classified?

In this paper, we argue that *VM techniques can be classified according to how variability is handled at the meta-model and model levels*. These two levels refer to both the product line artifacts and the product line variability. After having surveyed the relevant literature, we provide a classification framework that applies this two-level analysis to sort the VM techniques discussed, and highlight the fundamental differences between them in the way they capture variability. This classification provides a better understanding of these approaches and helps the engineers find the appropriate VM technique.

The remainder of this paper is structured as follows: Section 2 details how the survey of existing VM approaches was carried out. Section 3 presents our classification of VM approaches and briefly discusses them. Section 4 outlines some relevant related work while Section 5 concludes the paper.

2 Survey Protocol

With variability modelling being a major concern in SPL engineering, a plethora of methods have been developed by research and industry. So, in order to answer our first research question, a valid selection of relevant work on SPL variability modelling must first be performed. In this section we briefly explain the selection process followed to identify relevant contributions in the field.

The search process was performed in three steps. First, a thorough on-line research of relevant papers was performed using the Google search engine, using *search strings* based on the main concepts of the topic investigated. The search area was enlarged by using synonyms or other terms directly related to the topic of SPL variability as search strings. Similar searches were repeated on the main digital sources of research literature: ACM Digital Library, Lecture Notes in Computer Science, SpringerLink, SCOPUS (Elsevier), Web of Knowledge (ISI), IEEE Xplore, IEEE Computer Society Digital Library and ScienceDirect. In a second step, we performed a manual search in specific conference proceedings known to be classical venues of publication for SPL research: Software Product Line Conference (SPLC) and Product Family Engineering (PFE) conferences, Variability Modelling of Software-Intensive Systems (VaMoS) workshop. Finally, we also analysed other research projects addressing SPL engineering and variability to see which papers they considered relevant. The result of the search process produced a list of 236 papers.

Separately, we analysed the research literature for other surveys addressing the topic of SPL variability. Twelve papers were found: [10, 40, 32, 7, 16, 15, 20, 3, 26, 43, 33, 44]. For each of them, we extracted the list of referenced papers and regrouped them in a unique list, containing all papers cited in at least 2 surveys. Each paper on this list was assigned a value representing the number of surveys it appeared in. Based on this criteria, the list was ordered, resulting in a total of 55 papers. This selection criterion is relevant as it regroups the knowledge and expertise of other authors from SPLE.

The final list of papers to be analysed was obtained by comparing the previous two results. We identified 38 papers common to both lists. To obtain the final result, containing 20 papers, we also took into account the specific classification criteria we propose and discuss later on in this paper, and mapped them on the list of 38 papers.

3 Classification of Variability Modeling Methods

As variability is extensively used in SPL engineering, variability-related concepts can be gathered in a separate, dedicated language. In MDE, the structure of a domain is explicitly captured in a *meta-model*. Working at the level of *models* and *meta-models* makes it possible to analyse and classify SPL variability modelling methods at a high level of abstraction and objectiveness, and to extract general observations valid for an entire class of variability modelling approaches. We identify and analyse the central concepts used by a wide variety of VM techniques and show how they relate to each other. The analysis is performed at two levels: *meta-model* and *model*.

SPLs are usually characterized by two distinct concepts: a *set of core assets* or reusable components used for the development of new products (**assets model**); a *means*

to represent the commonality and variability between SPL members (**variability model**). Our classification is based on these two concepts. A thorough analysis of the research literature revealed two major directions in SPL variability modelling:

- Methods that use a **single (unique) model to represent the SPL assets and the SPL variability**:
  A. Annotate a base model by means of extensions: [11, 18, 35, 45]
  B. Combine a general, reusable variability meta-model with different domain meta-models: [31]
- Methods that **distinguish and keep separate the assets model from the variability model**:
  A. Connect Feature Diagrams to model fragments: [36, 13, 27, 2]
  B. Orthogonal Variability Modelling: [38, 30]
  C. ConIPF Variability Modeling Framework (COVAMOF): [42, 41]
  D. Decision model-based approaches: [14, 29, 17, 39, 4]
  E. Relate a common variability language with different base languages: [22]

In this classification, the terms *assets meta-model (AMM)* and *assets model (AM)* cover a broad spectrum, depending on the point of view of the different authors. They are further refined for each particular class of methods. Table 1 summarizes the proposed classification and the newly introduced concepts. It briefly describes what happens at meta-model and model level for the identified classes of variability modelling techniques. The papers cited here are analysed in more detail in the following.

3.1 Single model to describe the product line assets and the product line variability

This category contains techniques that extend a language or a general purpose meta-model with specific concepts that allow designers to describe variability. Their core characteristic is the mix of variability and PL assets concepts into a unique model. Concepts regarding variability and those describing the assets model are combined into a new language, that may either have a new, mixed syntax, or one based on that of the base model extended by the syntax of the variability language. This applies at both meta-model and model level. We further distinguish:

A. Annotate a base model by means of extensions [11, 45, 18, 35]: standard languages are not created to explicitly represent all types of variability. Therefore, SPL models are frequently expressed by extending or annotating such standard languages (models). The annotated models are unions of all specific models in a model family and contain all necessary variability concepts. Regarding our classification, we distinguish at meta-model level an assets meta-model enhanced with variability concepts (AMM+V).In this case, the term ”assets meta-model” (AMM) refers to a *base* or a *domain meta-model* (meta-model of standard language used, eg. UML). Then, at model level, product line models (PLM) can be derived. They conform to the AMM+V defined at meta-model level. Typical examples from this category are methods that extend UML with profiles and stereotypes: [11, 18, 35, 45].

B. Combine a general, reusable variability meta-model with different domain meta-models [31, 37]: this approach addresses in particular the meta-model level, where a

Technique Name	Meta-model level		Model level	
1. Unique model (combined) for product line assets and PL variability				
Annotating the base model by means of extensions	AMM+V		PLM (conform to AMM+V)	
Combine a general, reusable variability meta-model with base meta-models	AMM	VMM	PLM (confirm to AMM+V)	
	\ AMM+V	/		
2. Separate (distinct) assets model and variability model				
Connect Feature Diagrams to model fragments	AMM	VMM	AM	VM (FDM)
Orthogonal Variability Modelling (OVM)	AMM	VMM	AM	VM (OVM)
ConIPF Variability Modelling Framework (COVAMOF)	AMM	VMM (CVV)	AM	VM (CVV)
Decision model based approaches	AMM	VMM (DMM)	AM	VM(DM)
Combine a common variability language with different base modelling languages	AMM	VMM (CVL)	AM	VM (CVL)

Notation used:

AMM – assets meta-model
VMM – variability meta-model
AMM+V – assets meta model with variability
CVL – common variability language
DMM – decision meta-model
CVV – ConIPF variability view
AM – assets model
VM – variability model
PLM – product line model
FDM – feature diagram model
DM - decision model

Fig. 1. Classification of variability modelling techniques - meta-model and model level

two-step process is applied. Initially, two separate meta-models are created: an assets meta-model and a general, reusable variability meta-model. In a second step, they are combined, resulting in a unique assets meta-model extended with variability concepts. In this case, the term AMM denotes a domain meta-model (meta-model of domains specific language used for modelling). As for the previous category, at model level, PL models can be derived. A representative approach from this category comes from Morin et al. [31]. They propose a reusable variability meta-model describing variability concepts and their relations independently from any domain meta-model. Using Aspect-Oriented Modelling (AOM) techniques, variability can be woven into a given base meta-model, allowing its integration in a semi-automatic way into a wide range of meta-models.

3.2 Separate the assets model from the variability model

Techniques in this category have separate representations for the variability and the assets model. Elements from the variability model relate to assets model elements either by referencing or by other techniques. The key characteristic of such methods is the clear separation of concerns, which applies at both meta-model and model level. Some advantages of such approaches are: each asset model may have more than one variability model; designers can focus on modelling the SPL core assets and address the SPL variability separately; possibility for a standardized variability model. We further identify five sub-categories of methods pertaining to this category. The essential difference between all these sub-categories is the different type of variability model (meta-model) each one uses.

A. Connect Feature Diagrams to model fragments [36, 13, 27, 2]: Feature Diagrams (FD) [25] are the most popular VM technique in the SPL community. They organise features hierarchically in a tree-like structure where variability is defined via operators (or, xor, and) applied on child features. They also allow to model additional relations (mutual exclusion or dependence) via cross-tree constraints and have been subject to formalisation [5] and automated analyses [8]. Yet, how we associate model fragments to features is an emerging research direction. Different model fragment types can be associated to features. In this context, the feature diagram defines the PL variability, with each feature having an associated implementation. Concerning our classification, we notice a clear distinction between assets and variability related concepts at meta-model level. This situation extends to model level: separate assets and variability models exist. For this category, the assets model consists of a set of software artefact/asset fragments. The particular variability model used is a Feature Diagram.

B. Orthogonal Variability Modelling [38, 30]: as for all approaches in this category, the assets model and the variability model are distinct. The differentiating factor is the type of variability model used: an orthogonal variability model (OVM). There is also a difference regarding the assets model, which in this case is a compact software development artefact and no longer a set of model fragments. The variability model relates to different parts of the assets model using *artefact dependencies*. Pohl et al. [38] proposed the OVM concept, defined as: a model that defines the variability of a SPL separately and then relates it to other development artefacts like use case, component and test models. OVM provides a view on variability across all development artefacts. A slightly different OVM proposal comes from Metzger et al. [30].

C. ConIPF Variability Modeling Framework (COVAMOF) [42, 41]: this category contains the COVAMOF method proposed by Sinnema et al. Concerning our classification, we identify, at the meta-model level, separate variability and assets meta-models. This reflects also at model level, where a separate variability model, called COVAMOF Variability View (CVV), and an assets model can be distinguished. Sinnema et al. identify four requirements they considered essential for a variability modelling technique: uniform and first class representation of variation points at all abstraction levels; hierarchical organization of variability representation; first-class representation of dependencies; explicit modelling of interactions between dependencies. An analysis of existing variability approaches performed by Sinnema et al. revealed that none supported all four criteria. As a result they propose COVAMOF, an approach designed to uniformly model variability in all abstraction layers of a SPL.

D. Decision model based approaches: this class of approaches differs by using *decision models* as variability model. Decision-oriented approaches were designed to guide the product derivation process based on *decision models*. For Bayer et al. it is a model that "captures variability in a product line in terms of open decisions and possible resolutions" [6]. A decision model is basically a table where each row represents a decision and each column a property of a decision. The most well-known approach in this category is DOPLER [14]. It was designed to support the modelling of both problem space variability (stakeholder needs) using decision models, and solution space variability (architecture and components of technical solution) using asset models and also to assure traceability between them.

E. Relate a common variability language with different base languages [22]: methods belonging to this category propose a generic variability language which can relate to different base models, extending them with variability. Regarding our classification, at meta-model level there is a separate generic variability meta-model and an assets meta-model (AMM). The AMM is actually the meta-model of the base language on which the *common variability language* is applied. At model level, elements of the variability model relate to assets model elements by referencing and using substitutions. A representative approach in this category is the Common Variability Language (CVL) proposed by Haugen et al. [22].

4 Related Work

We identified several other surveys and studies that address to some extent the subject of product line variability modelling. In this section, the most relevant proposals are briefly analysed and compared to our work.

In [10] Chen et al. present the findings of their systematic literature review of papers on variability management in SPL engineering. The focus of the paper seems to be more to reveal the chronological background of various approaches and the history of variability management research rather than to classify the actual methods. Out paper differs significantly from the one of Chen et al. in this aspect, as our goal is not to detail the individual steps of a systematic review, but to focus on the actual classification of methods. In the conclusion of their paper, Chen et al. state that one of the aspects that needs immediate attention from SPL researchers and practitioners is to provide a classification of the different variability modelling approaches. This point summarizes precisely the contribution and focus of our work.

In [32] Mujtaba et al. use a systematic method to develop a SPL variability map and classify relevant literature accordingly. The main contributions of their work are: identification of emphasized and neglected SPL research areas, classification of contributions made by different approaches, providing an example of how to adapt systematic mapping studies to software engineering. They focus mostly on presenting the research methodology used. In contrast, our contribution is of a more practical nature: introduce general concepts regarding SPL variability and classify how exactly each of them captures variability.

In the technical report [44] Trigaux et al. present and compare different notations for modelling SPL variability: feature modelling, use cases, class diagrams. The criteria used for comparison are: representation of common and variable parts, distinction between types of variability, representation of dependencies between variable parts, support for model evolution, understandability and graphical representation. In our paper we cover a much broader spectrum of approaches and also classify them according to a model driven framework.

Another technical report that discusses SPL variability is [3]. Asikainen identifies the concepts suitable for modelling configurable SPLs, what is their semantics and what kind of language or modelling method can support these concepts. The core part of their discussion on previous existing literature consists of an analysis and comparison of methods for modelling variability. The evaluated methods fall in three categories:

feature-based, architecture-based and other methods. Compared to their work, we provide a clear classification of the methods studied from a model-driven perspective and point out the particular ways in which they express variability.

In [21] Haugen et al. introduce a reference model used for comparing system family modelling approaches. The proposed reference model is based on the distinction between the *generic sphere* (feature models, product line models) and the *specific sphere* (feature selection, product model). The authors identify three major approaches for modelling system families: using standard languages, annotating a general language, using dedicated domain-specific languages. Although some of the methods presented overlap in some way with methods we present in our paper, we use a different set of criteria for classifying variability modelling approaches.

In [43] Svahnberg et al. discuss the factors that need to be considered when selecting an appropriate technique for implementing variability. This paper focuses on how to implement variability in architecture and implementation artefacts, like the software architecture design and the components and classes of a software system. Their main contribution is to provide a taxonomy of techniques that can be used to implement variability. Svahnbeg et al. focus on discussing the actual implementation of variability, mostly at code level, while we discuss variability modelling at the higher level of abstraction of languages and models.

5 Conclusion

Initiated more than two decades ago and developed by an active research community, variability modelling became the key concern in SPL engineering and important research topic in software engineering in general. Therefore, a lot of efforts of the SPL community were in this direction. As a result, the number of variability modelling approaches proposed by research or industry quickly increased. Such techniques are needed in ever growing number of applications, from complex manufacturing activities to online configurators needed for e-commerce websites. Thus, it is of the utmost importance to review VM techniques and to understand their fundamental characteristics in order to choose the most appropriate one for a particular application context. The classification provided in this paper is a first step in this direction, outlining major trends in variability modelling and declining them at the metamodel and model levels. Future work includes the evaluation of the surveyed approaches against a set of criteria enabling a fine-grained comparison and giving practical insights to engineers who need to ground their decisions. We also plan to apply the surveyed approaches on different examples, which would a allow for a more pertinent comparison and also point out the relative advantages and disadvantages of each individual approach.

References

1. Software product line conference - hall of fame. `http://splc.net/fame.html`
2. Apel, S., Janda, F., Trujillo, S., Kästner, C.: Model superimposition in software product lines. In: ICMT '09: Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations. pp. 4–19. Springer-Verlag, Berlin, Heidelberg (2009)

3. Asikainen, T., Soininen, T.: Modelling methods for managing variability of configurable software product families (2004)
4. Atkinson, C., Bayer, J., Muthig, D.: Component-based product line development: the kobra approach. In: Proceedings of the first conference on Software product lines : experience and research directions: experience and research directions. pp. 289–309. Kluwer Academic Publishers, Norwell, MA, USA (2000)
5. Batory, D.S.: Feature models, grammars, and propositional formulas. In: SPLC. pp. 7–20 (2005)
6. Bayer, J., Flege, O., Gacek, C.: Creating product line architectures. In: IW-SAPF. pp. 210–216 (2000)
7. Bayer, J., Gerard, S., Haugen, Ø., Mansell, J.X., Møller-Pedersen, B., Oldevik, J., Tessier, P., Thibault, J.P., Widen, T.: Consolidated product line variability modeling. In: Software Product Lines, pp. 195–241 (2006)
8. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: A literature review. Information Systems 35(6), 615 – 636 (2010), http://www.sciencedirect.com/science/article/pii/S0306437910000025
9. Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, J.H., Pohl, K.: Variability issues in software product lines. In: PFE '01: Revised Papers from the 4th International Workshop on Software Product-Family Engineering. pp. 13–21. Springer-Verlag, London, UK (2002)
10. Chen, L., Ali Babar, M., Ali, N.: Variability management in software product lines: a systematic review. In: Software Product Line Conference. pp. 81–90. Carnegie Mellon University, Pittsburgh, PA, USA (2009)
11. Clauss, M.: Generic modeling using uml extensions for variability. In: OOPSLA (2001)
12. Coplien, J., Hoffman, D., Weiss, D.: Commonality and variability in software engineering. IEEE Software 15(6), 37–45 (1998)
13. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: GPCE. pp. 422–437 (2005)
14. Dhungana, D., Grünbacher, P., Rabiser, R.: The dopler meta-tool for decision-oriented variability modeling: a multiple case study. Automated Software Engineering 18(1), 77–114 (2010)
15. Djebbi, O., Salinesi, C.: Criteria for comparing requirements variability modeling notations for product lines. In: Proceedings of the Fourth Internationa Workshop on Comparative Evaluation in Requirements Engineering. pp. 20–35. IEEE Computer Society, Washington, DC, USA (2006)
16. Elena Alana, A.R.: Domain engineering methodologies survey. Tech. rep., CORDET (2007)
17. European Software Engineering Institute Spain, IKV++ Technologies AG Germany: Master: Model-driven architecture instrumentation , enhancement and refinement. Tech. Rep. IST-2001-34600, IST (2002)
18. Gomaa, H., Shin, M.E.: Multiple-view modelling and meta-modelling of software product lines. IET Software 2(2), 94–122 (2008)
19. Halmans, G., Pohl, K.: Communicating the variability of a software-product family to customers. Inform., Forsch. Entwickl. 18, 113–131 (2004)
20. Harsu, M.: A survey on domain engineering. Tech. rep., Institute of Software Systems, Tampere University of Technology (2002)
21. Haugen, .o., Pedersen, B.M., Oldevik, J.: Comparison of System Family Modeling Approaches. In: Software Product Lines, 9th International Conference. Lecture Notes in Computer Science, vol. 3714, pp. 102–112. Springer (2005)
22. Haugen, O., Moller-Pedersen, B., Oldevik, J., Olsen, G.K., Svendsen, A.: Adding standardized variability to domain specific languages. Software Product Line Conference, International 0, 139–148 (2008)

23. Jean-Christophe TRIGAUX, P.H.: Modelling variability requirements in software product lines: a comparative survey. Tech. rep., FUNDP Namur (2003)
24. Jézéquel, J.M.: Model driven design and aspect weaving. Software and System Modeling 7(2), 209–218 (2008)
25. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (foda) feasibility study. Tech. rep., Carnegie-Mellon University Software Engineering Institute (November 1990)
26. Khurum, M., Gorschek, T.: A systematic review of domain analysis solutions for product lines. J. Syst. Softw. 82 (December 2009)
27. Laguna, M.A., González-Baixauli, B.: Product line requirements: Multi-paradigm variability models. In: 11th Workshop on Requirements Engineering WER (2008)
28. Maccari, A., Heie, A.: Managing infinite variability in mobile terminal software: Research articles. Softw. Pract. Exper. 35(6), 513–537 (2005)
29. Mansell, J.X., Sellier, D.: Decision model and flexible component definition based on xml technology. In: PFE. pp. 466–472 (2003)
30. Metzger, A., Pohl, K., Heymans, P., Schobbens, P.Y., Saval, G.: Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. Requirements Engineering, IEEE International Conference on 0, 243–253 (2007)
31. Morin, B., Perrouin, G., Lahire, P., Barais, O., Vanwormhoudt, G., Jézéquel, J.M.: Weaving variability into domain metamodels. In: MoDELS. pp. 690–705 (2009)
32. Mujtaba, S., Petersen, K., Feldt, R., Mattsson, M.: Software product line variability: A systematic mapping study (2008)
33. Myllymäki, T.: Variability management in software product lines. Tech. rep., Tampere University of Technology Software Systems Laboratory ARCHIMEDES (2001)
34. Northrop, L.: A framework for software product line practice. In: Proceedings of the Workshop on Object-Oriented Technology. pp. 365–376. Springer-Verlag London, UK (1999)
35. de Oliveira Junior, E.A., de Souza Gimenes, I.M., Huzita, E.H.M., Maldonado, J.C.: A variability management process for software product lines. In: CASCON. pp. 225–241 (2005)
36. Perrouin, G., Klein, J., Guelfi, N., Jézéquel, J.M.: Reconciling automation and flexibility in product derivation. In: SPLC '08: Proceedings of the 2008 12th International Software Product Line Conference. pp. 339–348. IEEE Computer Society, Washington, DC, USA (2008)
37. Perrouin, G., Vanwormhoudt, G., Morin, B., Lahire, P., Barais, O., Jézéquel, J.M.: Weaving variability into domain metamodels. Software and Systems Modeling pp. 1–23 (2010)
38. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2005)
39. Schmid, K., John, I.: A customizable approach to full lifecycle variability management. Sci. Comput. Program. 53, 259–284 (December 2004)
40. Sinnema, M., Deelstra, S.: Classifying variability modeling techniques. Inf. Softw. Technol. 49 (July 2007)
41. Sinnema, M., Deelstra, S., Hoekstra, P.: The covamof derivation process. In: ICSR. pp. 101–114 (2006)
42. Sinnema, M., Deelstra, S., Nijhuis, J., Bosch, J.: Covamof: A framework for modeling variability in software product families. In: SPLC. pp. 197–213 (2004)
43. Svahnberg, M., Gurp, J.V., Bosch, J.: A taxonomy of variability realization techniques. Software Practice and Experience 35, 705–754 (2005)
44. Trigaux, J.C., Heymans, P.: Modelling variability requirements in software product lines: a comparative survey. Tech. rep., University of Namur, Computer Science Institute (2003)
45. Ziadi, T., Jézéquel, J.M.: Software Product Lines, chap. Product Line Engineering with the UML: Deriving Products, pp. 557–586. Springer Verlag (2006)

Towards Evolution of Generic Variability Models

Andreas Svendsen[1,2], Xiaorui Zhang[1,2], Øystein Haugen[1], and Birger Møller-Pedersen[2]

[1]SINTEF, Pb. 124 Blindern, 0314 Oslo, Norway
[2]Department of Informatics, University of Oslo, Pb. 1080 Blindern, 0316 Oslo, Norway
andreas.svendsen@sintef.no, xiaorui.zhang@sintef.no, oystein.haugen@sintef.no, birger@ifi.uio.no

Abstract. We present an approach for evolving separate variability models when the associated base model is altered. The Common Variability Language (CVL) is a generic language for modeling variability in base models. The base models are oblivious to the associated variability models, causing additional challenges to this association when the base models are maintained. Assuming that a base model has been changed, we suggest using CVL to record this change. Further analysis of this CVL model reveal the impact of the change, which if possible, can result in automatic evolution of the variability model corresponding to the changed base model. We illustrate and discuss the approach using an example from the train domain.

Keywords: Variability modeling, variability model evolution, coupled evolution, Common Variability Language.

1 Introduction

Model-Driven Development (MDD) has in the recent years increased in popularity, since it allows the developer to solve problems at a higher level of abstraction. Techniques, such as software product line modeling, are increasingly adopted by the industry to produce software more efficiently. Thus, the development of methods for creating software product line models is important. One such method is to use separate variability models to describe how a base model, representing a software system, can be changed to form other (product) models, representing variations of the original software system.

The Common Variability Language (CVL) is a generic language for modeling variability in base models [5, 6, 8]. CVL consists of a variability model, specifying the possible variations on the base model, and a resolution model, resolving the variability in the variability model to form new product models. Thus, CVL models the variants of a base model without adding annotations or variability concepts to the base model (and base language).

The importance of a standardized and generic variability language has been recognized, and a standardization process has been initiated to create such a language [6]. However, keeping the variability concepts separate from the base model adds certain challenges which should be addressed. One of these challenges, which will be

the focus of this paper, is the maintenance of a variability model when the associated base model is altered. Assume that a product line consists of a base model and several variability models associated to this base model. If the base model is maintained, how can we ensure that the variability models are still valid? Updating the variability models according to the changed base model can be a manual and tedious task.

In this paper we suggest an approach for automating the maintenance of variability models when their associated base model is changed. We apply CVL to record the changes in the base model, and perform analysis of these CVL models to reveal the impact of the change on the variability models. Based on the results of the analysis, we give feedback on all changes that invalidate the variability model, and if possible, we update the variability model to correspond correctly to the changed base model.

More specifically, the contribution of this paper is an approach for evolving separate variability models when an associated base model is changed. The approach originates and uses concepts from CVL, and can be a useful contribution to the CVL standardization process. We perform preliminary evaluation and exemplify the approach using a prototype implementation based on CVL in Eclipse and an example from the train domain.

The outline for the rest of the paper is as follows: In Section 2 we give some background information about CVL and the example domain used throughout the paper. Section 3 further elaborates and exemplifies the challenge raised when the base model is changed. Section 4 explains the approach of using CVL to evolve CVL models and illustrates the approach using an example from the train domain. In Section 5 we discuss the prototype implementation, and the advantages and challenges with the approach. Section 6 gives some related work, before Section 7 gives some concluding remarks and future work.

2 Background

2.1 Common Variability Language

CVL is a generic language for modeling variability in any models in any MOF-based[1] modeling language. In other words, CVL can be applied to both models in Domain-Specific Languages (DSL) and models in more general languages like UML. One key feature of CVL is that it is separate from the base model and applies one-way associations to the base model. Since CVL is separate, no annotations or variability concepts is added into the base model or base language.

The core part of CVL consists of substitutions, which replace model elements and model element attributes to produce new variants of the base model. In addition to the substitutions, CVL also includes concepts for abstractions, such as using features as part of the concrete syntax, as known from feature models. A prototype implementation of CVL, as an Eclipse plug-in, has been developed and a case study has been conducted for evaluation (see [14]).

[1] http://www.omg.org/mof/

In this paper we focus on the most significant substitution in CVL, namely the *fragment substitution*. A fragment substitution replaces a *placement fragment* in the base model, which is a set of model elements, with a *replacement fragment*, which is another set of model elements. Since the model elements in the replacement fragment are copied, the only change performed in the base model is to the placement fragment. This substitution is illustrated in Fig. 1. Both the placement fragment and replacement fragment are represented by *boundary elements*, recording all references to and from the model elements inside the fragments. A fragment substitution binds these boundary elements (*ToBinding* and *FromBinding*) such that executing the substitution will replace the references according to the binding (i.e. the reference from *A* to *B* will be redirected to *F*). Note that these references must follow the type rules from the metamodel, so that the substitution is type safe.

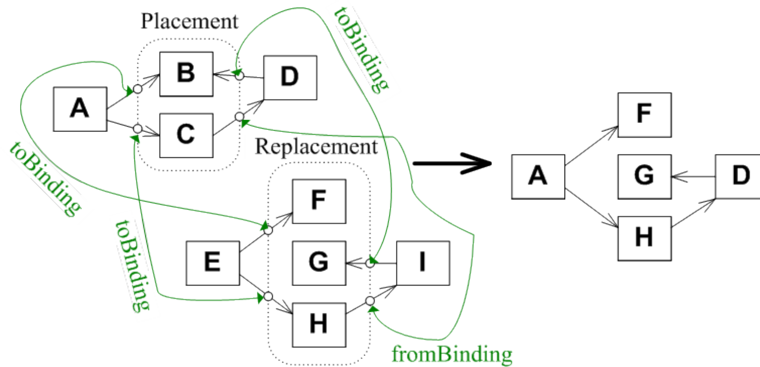

Fig. 1. Fragment substitution replaces a placement fragment with a replacement fragment

2.2 Train Control Language

The Train Control Language (TCL) is a DSL for modeling signaling systems on train stations [3, 13]. The intention of TCL is to automate the development of interlocking source code which ensures safe train movement on a train station. TCL has been developed in cooperation with ABB, Norway[2].

TCL is defined by a metamodel and has been developed as an Eclipse plug-in with an editor, model analyzer and code generator. The concrete syntax of TCL is illustrated in Fig. 2, with the most significant concepts annotated: *TrainRoute*, *TrackCircuit*, *LineSegment*, *Switch*, *Endpoint* and *Signal*. A TrainRoute is a path between two signals that must be allocated before a train can move into or out of the station. A TrainRoute is divided into TrackCircuits, which are segments where a train can be located. A TrackCircuit is further divided into LineSegments and Switches, which are connected by Endpoints.

We will use a TCL model as a base model to illustrate how we can evolve the CVL model when the base model changes.

[2] http://www.abb.no

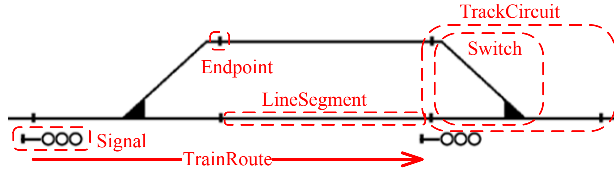

Fig. 2. TCL concrete syntax (with annotations)

3 Problem Description

Since CVL models are kept separate from the base model, changes can be conducted to the base model that may invalidate the CVL models associated with this base model. This is extra challenging since CVL models, to be truly separate, only contains one-way references to the base model. The CVL model can be invalidated either syntactically, e.g. null-pointer references, or semantically, resulting in meaningless product models. In this paper we focus on the syntactic changes in the base model and the evolution of the CVL model according to these kinds of changes.

The most significant substitution of CVL is the fragment substitution, allowing a set of model elements to be replaced by another set of model elements. In other words, the fragment substitution is flexible, and can express any kind of structural changes, where base model elements are added, deleted or modified. If the placement or replacement fragments refers base model elements that are changed in this way, these fragments are invalidated. We limit the analysis presented in this paper to placement fragments, since replacement fragments easily can be repositioned into library models, which are kept separate from the base model (see [14]).

Fig. 3 illustrates the challenge of evolving CVL models and shows an overview of the approach (see Section 4). Step 1 involves executing a CVL model to transform a base model to a product model, i.e. inserting a *side track* into a two-track station model. However, assume that the base model is modified, ending up with an evolved base model, i.e. a three-track station model (top right). The original CVL model does not apply to the evolved base model. Our approach is concerned with evolving the original CVL model according to the evolved base model (step 3). Step 4 involves executing the evolved CVL model to obtain an evolved product model. We explain step 2 and give further details about step 3 in Section 4.

4 Using CVL to Evolve CVL Models

4.1 The Approach

We suggest using CVL and fragment substitution to record the evolution of the base model (see Fig. 3, step 2). This CVL model, the *evolution CVL model*, can then be

compared to the original CVL model, and the comparison can be analyzed to obtain inconsistencies. We let the user decide whether to obtain this CVL model manually or automatically by comparing the base model and the evolved base model [15].

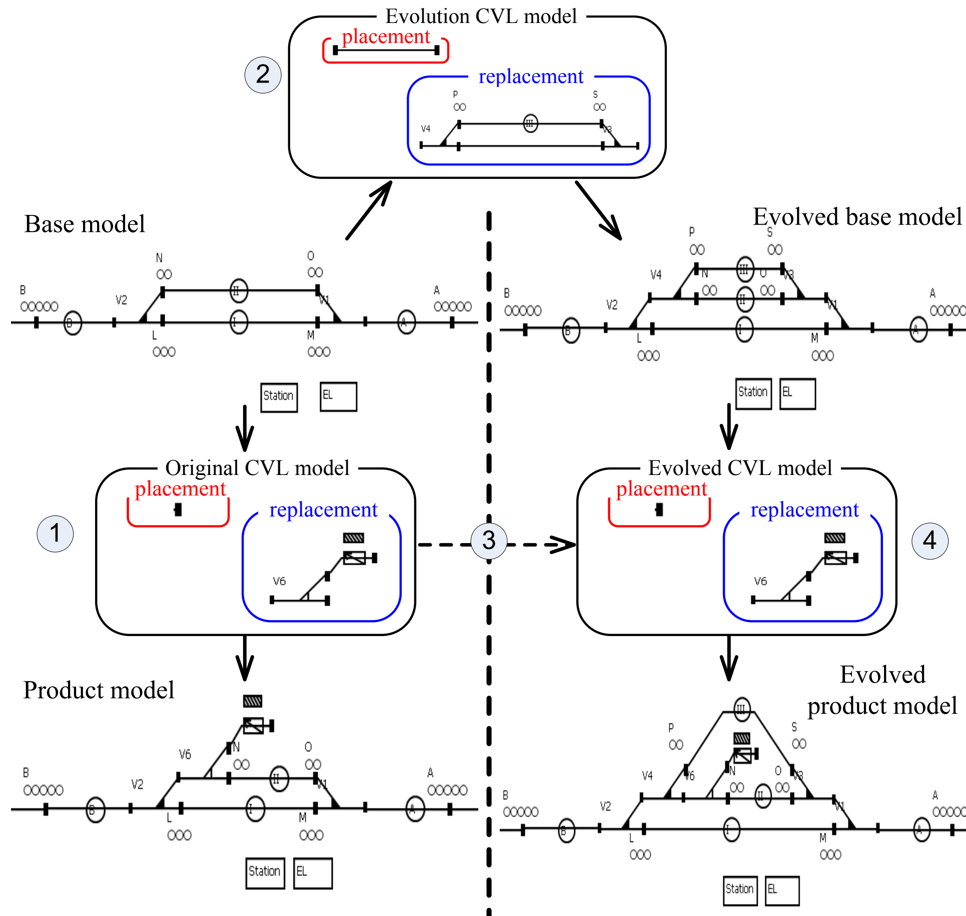

Fig. 3. Overview of the approach

Recall that a fragment in CVL is defined by boundary elements, which record the references to and from the base model elements in the fragment. Furthermore, note that the base model elements in a placement fragment are replaced by the model elements in a replacement fragment. Thus, two placement fragments cannot overlap, meaning that two changes cannot be performed to the same base model element.

Fig. 4 gives an overview of possible inconsistencies between the two CVL models. A *border inconsistency* indicates that two model elements that are replaced in two different substitutions are directly connected. Since the substitutions are independent, the association between them cannot be set in either of the substitutions. An *element inconsistency* indicates that a base model element is being replaced by two different substitutions. The base model element cannot be replaced twice. Note that the purpose of the figure is for illustrating the kinds of inconsistencies and not all possible

situations where inconsistencies can occur. For instance, there will still be inconsistencies if the associations are inverted.

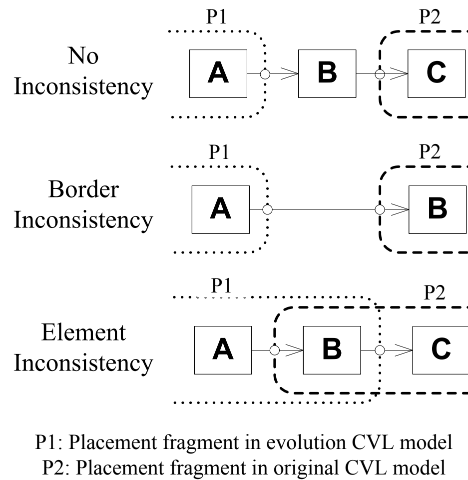

Fig. 4. Types of inconsistencies between two CVL models

We have developed an algorithm to deal with the inconsistencies between CVL models. Intuitively, when an inconsistency is found, the algorithm makes an attempt to solve the inconsistency by using the model elements in the replacement fragment. For instance, for the border inconsistency in Fig. 4 the base model element *A* is replaced by the evolution CVL model while base model element *B* is replaced by the original CVL model. In this case, the algorithm transforms the original CVL model such that it refers the replacement of *A* (from the evolution CVL model) instead of *A* as the context of the fragment *P2*. For the element inconsistency in Fig. 4 the base model element *B* is being replaced by both CVL models. Thus, the algorithm transforms the original CVL model such that the replacement of base model element *A* (from the evolution CVL model) is referred instead of *A* as the context of fragment *P2*. In addition, the replacement of *B* (from the evolution CVL model) is recorded as a contained element instead of *B*. Note that in some cases, with too little context information, the algorithm may not be able to find a unique base model element from the replacement fragment (evolution CVL model). The user is then prompted to make a decision for which one to use.

As a summary, our approach involves creating a CVL evolution model to record the evolution of the base model (Fig. 3, step 2). By comparing and analyzing the differences between this CVL model and the original CVL model, we reveal and solve inconsistencies between them, and transform the original CVL model to an evolved CVL model (Fig. 3, step 3), which applies to the evolved base model.

4.2 Evolving CVL Models

To illustrate the approach, we briefly walk through an example where we apply CVL to a TCL model, evolve this TCL model and finally evolve the original CVL model

according to the evolved TCL model. The example is illustrated in Fig. 3, where the base model is a two-track station, which is evolved to a three-track station, and the original CVL model adds a side track to the base model.

We first develop (either manually or automatically) the evolution CVL model, which applies to the base model. This CVL model is then compared to the original CVL model to reveal any inconsistencies between them. The algorithm discovers an element inconsistency, since both CVL models contain a placement fragment that spans over a common TCL endpoint. The inconsistency is illustrated in Fig. 5 (middle), where the placement fragment *P1* (original CVL model) replaces the endpoint with a side track, and the placement fragment *P2* (evolution CVL model) replaces the endpoint, together with a line segment and another endpoint, with a double-track. The replacement fragments are illustrated on the left and right side of the figure, where *R1* is bound to *P1* and *R2* is bound to *P2*.

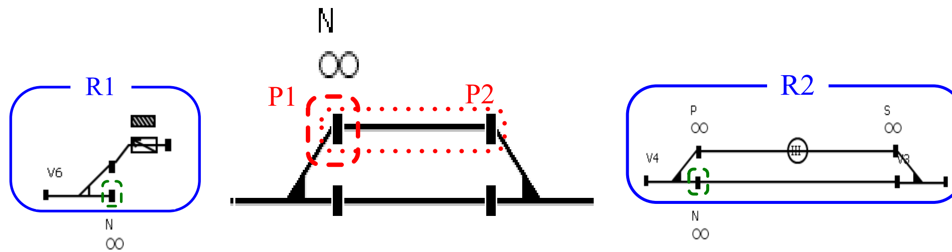

Fig. 5. Element inconsistency between two CVL placement fragments on a TCL model

To solve the inconsistency, the algorithm fetches the model element from the replacement fragment (in the evolution CVL model) that is bound to the common endpoint in Fig. 5. Note that in this replacement fragment there are several TCL endpoints that can potentially match the common endpoint. E.g. the side track can be placed on top of the third track. However, the context of the placements include among others a reference to signal *N*, which is unique and located at the second track in the evolved base model. This is illustrated in Fig. 5 with circles around the endpoints (in *R1* and *R2*) which have references (context) to signal *N*. Thus, the matching is unique and the inconsistency can be solved automatically by the algorithm. The solution to the inconsistency is stored in a mapping table for use when transforming the original CVL model. For this example, a mapping is created between the endpoint in *P1* and the endpoint in the circle in *R2*.

When the strategy for how to solve the inconsistency is known, the algorithm transforms the original CVL model to the evolved CVL model (Fig. 3, step 3). This is a one-to-one mapping where the references to the two-track station model are replaced with references to the three-track station model. For any inconsistency, the mapping table is used to obtain how to associate the evolved CVL model to the evolved base model. For this example, the placement in the evolved CVL model contains the model element in the circle in *R2*, with the appropriate context, instead of the endpoint in *P1*.

When the evolved CVL model is created, it can be executed to obtain the evolved product model, which yields a three track station with a side track on the second track

(see Fig. 3, step 4). Note that the procedure of evolving the CVL model and executing it is automatic, and do not require any user interaction, unless the inconsistencies cannot be solved automatically.

5 Discussion

To evaluate the feasibility of the approach, we have extended the CVL editor with functionality to perform the algorithm described in this paper. The user can choose two CVL models as input, one evolution CVL model and one original CVL model. The algorithm is then executed to find inconsistencies and to transform the original CVL model to obtain the evolved CVL model. Our preliminary evaluation shows that the approach is feasible and works well for the example described in this paper.

Even though our approach is specific for CVL and fragment substitution, CVL and fragment substitution are generic and can describe variability in any model in any DSL. The approach takes advantage of the nature of CVL, which specifies specifically where and how the variability is applied to the base model, to perform the analysis. Thus, the approach fits well with the intentions of the upcoming CVL standard.

Since the approach is performed automatically, it has its strength when more than one substitution and/or more than one original CVL model is associated with the base model. Then manual work of evolving the CVL models or the product models without tool support can be huge. Furthermore, since base models most often is updated based on bug-fixes or other small changes, the amount of inconsistencies, and their impact, remains small. Thus, our approach can in particular be useful in these situations.

Even though a stronger association between the CVL model and the base model can avoid some of the issues discussed in this paper, e.g. by using two-way references, having a clear separation has its advantages. For instance, having several variability models associated with a single base model is possible, for describing different kinds of product lines. Furthermore, CVL can model variability in a base model without the need to change the base language to add variability concepts or associations. This results in the possibility of applying CVL and creating product lines more rapidly.

Only using simpler kinds of substitutions, limiting the type of replacement to attributes or single base mode elements, would simplify the possible inconsistencies when performing the evolution of the CVL model. However, we believe that the fragment substitution plays an important role in making CVL flexible and generic for expressing all kinds of variability. On the other hand, note that this approach can easily be modified to support these kinds of substitutions instead or in addition to the fragment substitution.

6 Related Work

Much research effort has been put forward in the area of model coupled-evolution in the recent years. Existing work mainly fall into two categories: (1) when the

metamodel evolves, how to update the existing instance models in order to conform to the evolved metamodel [7, 9, 10]; (2) when a model changes, how to update its existing related models in order to eliminate all the possible inconsistencies caused by the model changes. The latter is similar to the coupled-evolution we deal with in this paper.

Approaches for bidirectional model transformation have been proposed to keep two models consistent by updating one model in accordance with the other [12]. Chivers and Paige [1] propose a reversible template language that supports round-trip transformations between UML models and predicate logic, such that new information encoded in logic can be seamlessly integrated with information encoded in the model. Mu et al. [11] present an algebraic approach to bidirectional updating, where a formal model of the bidirectional transformations is proposed. The developer writes the transformations as a functional program, such that the synchronization behavior is automatically derived by algebraic reasoning. The approach is able to deal with duplication and structural changes.

Finkelstein et al. [4] propose an approach for inconsistency handling in multi-perspective specifications by combining their *ViewPoints* framework for perspective development with a logic-based approach for inconsistency handling.

Deng et al. [2] present techniques for addressing domain evolution challenges in software product lines. They show how to minimize the inconsistencies caused by the evolution of MDD-based product line architectures for large-scale distributed real-time and embedded systems by adopting a layered architecture and model-to-model transformation tool support.

7 Conclusion and Future Work

This paper has presented an approach for evolving a CVL model when the associated base model is changed. We applied CVL to record the change (evolution step) in the base model and presented an algorithm for transforming the original CVL model accordingly. We presented the kinds of inconsistencies that can occur in this process, and gave suggestions for how to solve them. The approach was illustrated on a concrete example using a CVL model applied on a two-track station model from the Train Control Language. Furthermore, we indicated how the approach has been implemented and discussed advantages and challenges with the approach.

We see further evaluation of the approach using additional examples and other domains as important future work. Furthermore, the current approach only considers the syntax of the base models when performing the evolution step. In other words, the evolved product models are syntactically correct, but can be semantically invalid according to the base language semantics. Hence, extensions of the approach to take the semantics of the base language into account will be investigated. Another extension to the approach to also consider language evolution is significant and should be investigated. We can then be able to handle not only changes to base models, but also changes to the metamodels.

Acknowledgements. The work presented here has been developed within the MoSiS project ITEA 2 – ip06035 part of the Eureka framework and the CESAR project funded by ARTEMIS Joint Undertaking grant agreement No 100016.

References

1. Chivers, H. and Paige, R.: Xround: Bidirectional Transformations and Unifications Via a Reversible Template Language. In: Model Driven Architecture – Foundations and Applications, Lecture Notes in Computer Science, vol. 3748, pp. 205-219. Springer (2005)
2. Deng, G., Lenz, G., and Schmidt, D., "Addressing Domain Evolution Challenges in Software Product Lines", Satellite Events at the MoDELS 2005 Conference, Lecture Notes in Computer Science, (2006)
3. Endresen, J., Carlson, E., Moen, T., Alme, K.-J., Haugen, Ø., Olsen, G.K., and Svendsen, A., "Train Control Language - Teaching Computers Interlocking", Computers in Railways XI (COMPRAIL 2008), Toledo, Spain, (2008)
4. Finkelstein, A.C.W., Gabbay, D., Hunter, A., Kramer, J., and Nuseibeh, B.: Inconsistency Handling in Multiperspective Specifications. IEEE Trans. Softw. Eng. 20, 569-578 (1994)
5. Fleurey, F., Haugen, Ø., Møller-Pedersen, B., Olsen, G.K., Svendsen, A., and Zhang, X., "A Generic Language and Tool for Variability Modeling," SINTEF, Oslo (2009),
6. Fleurey, F., Haugen, Ø., Møller-Pedersen, B., Svendsen, A., and Zhang, X., "Standardizing Variability - Challenges and Solutions", in 15th International Conference on System Design Languages (SDL 2011). Toulouse, France, (2011)
7. Gruschko, B., "Towards Synchronizing Models with Evolving Metamodels", Int. Workshop on Model-Driven Software Evolution held with the ECSMR, (2007)
8. Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G., and Svendsen, A., "Adding Standardized Variability to Domain Specific Languages", in Proceedings of the 2008 12th International Software Product Line Conference: IEEE Computer Society, (2008)
9. Herrmannsdoerfer, M., "Cope: A Workbench for the Coupled Evolution of Metamodels and Models", in Proceedings of the Third international conference on Software language engineering. Eindhoven, The Netherlands: Springer-Verlag, (2011), pp. 286-295
10. Herrmannsdoerfer, M., Benz, S., and Juergens, E., "Automatability of Coupled Evolution of Metamodels and Models in Practice", in Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems. Toulouse, France: Springer-Verlag, (2008), pp. 645-659
11. Mu, S., Hu, Z., and Takeichi, M., "An Algebraic Approach to Bi-Directional Updating", ASIAN Symposium on Programming Languages and Systems, (2004)
12. Stevens, P.: A Landscape of Bidirectional Model Transformations. In, L. Ralf, mmel, V. Joost, Jo, and S. o, (eds.) Generative and Transformational Techniques in Software Engineering Ii, pp. 408-424. Springer-Verlag (2008)
13. Svendsen, A., Olsen, G.K., Endresen, J., Moen, T., Carlson, E., Alme, K.-J., and Haugen, O., "The Future of Train Signaling", Model Driven Engineering Languages and Systems (MoDELS 2008), Tolouse, France, (2008)
14. Svendsen, A., Zhang, X., Lind-Tviberg, R., Fleurey, F., Øystein, H., Møller-Pedersen, B., and Olsen, G.K., "Developing a Software Product Line for Train Control: A Case Study of Cvl", in Proceedings of the 14th international conference on Software product lines: going beyond. Jeju Island, South Korea: Springer-Verlag, (2010)
15. Zhang, X., Haugen, Ø., and Møller-Pedersen, B., "Model Comparison to Synthesize a Model-Driven Software Product Line", in 15th International Software Product Line Conference. Munich, Germany, (2011)

Towards a Family-based Analysis of Applicability Conditions in Architectural Delta Models

Arne Haber[1], Thomas Kutz[1], Holger Rendel[1],
Bernhard Rumpe[1], and Ina Schaefer[2]

[1] Software Engineering, RWTH Aachen University, Germany
[2] Institute for Software Systems Engineering, TU Braunschweig, Germany

Abstract. Modeling variability in software architectures is a fundamental part of software product line development. Δ-MontiArc allows describing architectural variability in a modular way by a designated core architecture and a set of architectural delta models modifying the core architecture to realize other architecture variants. Delta models have to satisfy a set of applicability conditions for the definedness of the architectural variants. The applicability conditions can in principle be checked by generating all possible architecture variants, which requires considering the same intermediate architectures repeatedly. In order to reuse previously computed architecture variants, we propose a family-based analysis of the applicability conditions using the concept of inverse deltas.

Keywords: Software Architectures; Delta-oriented Architectural Variability Modeling; Family-based Product Line Analysis

1 Introduction

Modeling variability of the software architecture is an integral part in software product line development. Δ-MontiArc [11] is a modular, transformational variability modeling approach for software architectures. In Δ-MontiArc, a family of software architectures is described by a designated core architecture model and a set of delta models containing modifications to the core architecture. A delta model can add and remove components, ports and connections and modify the internal structure of components. By applying the modifications contained in a delta model, an existing architecture model is transformed into another architectural variant. A particular variant in the architecture family is specified by a product configuration comprising the deltas that have to be applied to the core architecture. In order to resolve conflicts between delta models modifying the same architectural elements, an application order constraint can be attached to each delta model determining which other delta models have to be or should not be applied before this delta model.

Application order constraints are also used to ensure that each delta model is applicable to the core or intermediate architecture during product generation. Applicability means that all elements removed or modified by the delta exist

and that all elements added by the delta do not yet exist. If these applicability conditions hold, the architecture resulting from delta application is defined, otherwise the result is undefined, following [17]. In order to check that the application order constraints guarantee the applicability of the delta models during product generation, a naive *product-based approach* is to generate and check the architectures for all possible product configurations and all possible intermediate architectures. This naive approach is very inefficient because for examining all possible product architectures, the same intermediate products might have to be re-generated several times.

In this paper, we propose *inverse deltas* in order to enable an efficient family-based analysis of the applicability conditions in architectural delta models. A *family-based analysis* checks all products that can be derived by traversing the whole artifacts base of the product line only once, without generating all possible products explicitly. An inverse delta reverts the operations carried out by the original delta such that applying the delta and its inverse to an architecture retrieves the original architecture. The family-based analysis constructs the *family application order tree (FAOT)* which contains all possible delta application orders that comply to the application order constraints attached to the delta models. Using inverse deltas, it is possible to traverse the *FAOT* without generating the same intermediate architectures several times. Instead, the tree is only traversed once in a depth-first manner. In this traversal, already computed intermediate architectures are reused by reconstructing them with the application of inverse deltas. If the analysis of the *FAOT* passes the applicability conditions checks, it is guaranteed that for all possible product configurations, which are subsets of the set of deltas models, satisfy the applicability conditions and lead to a defined resulting architecture.

This paper is structured as follows: Section 2 briefly introduces Δ-MontiArc. Product Generation is described in Section 3. The family-based analysis using inverse deltas is proposed in Section 4 and discussed in 5. Section 6 describes related approaches. Section 7 concludes with an outlook to future work.

2 Δ-MontiArc

Δ-MontiArc [11] is a modular and transformational approach for describing architectural variability and is based on the textual architecture description language (*ADL*) MontiArc [10]. MontiArc focuses on the domain of distributed information-flow architectures. An example for a MontiArc architecture is given in Listing 1.1 which represents an Anti Lock Braking System (ABS). It contains inputs for four wheelsensors which measure the current speed of the four wheels of a car, a signal for the braking command, and four outputs to control the brake actuators. The component `abs` calculates the individual braking pressures for all wheels. If a wheel is close to a blocking state indicated by the corresponding wheel sensor, it reduces the braking pressure for this wheel to maintain the stability of the vehicle.

```
component BrakingSystem {
  autoconnect port;
  port
    in WheelSensor wheelspeed1,
    in WheelSensor wheelspeed2,
    in WheelSensor wheelspeed3,
    in WheelSensor wheelspeed4,
    in BrakeCommand brake,
    out BrakePressure wheelpressure1,
    out BrakePressure wheelpressure2,
    out BrakePressure wheelpressure3,
    out BrakePressure wheelpressure4;
  component ABS abs;
}
```

Listing 1.1. MontiArc Model for an Anti-Lock Bracking System.

```
delta ElectronicStabilityControl after TractionControl {
  modify component BrakingSystem {
    add port in AccelerationSensor lateralaccel;
    remove component tc;
    add component ESC esc;
    connect lateralaccel -> esc.accel;
  }
}
```

Listing 1.2. Delta for Electronic Stability Control.

In Δ-MontiArc, MontiArc is extended with the concept of delta modeling [4, 16, 15] to represent architectural variability. Based on a core architecture specified in MontiArc, architectural deltas are specified that add, remove or modify architecture elements using the operations `add`, `remove` and `modify` for ports, components and corresponding parameters. For connectors, the operations `connect` and `disconnect` are available. Further possible operations are listed in [9], but not required for the comprehension of this paper.

An example for a delta model specified with Δ-MontiArc is given in Listing 1.2. The depicted `ElectronicStabilityControl` delta can only be applied if the `TractionControl` delta, which adds an input for accelleration pedal position, is executed before. This information is provided by an application order constraint in an `after` clause which specifies deltas which must or must not be executed before the current delta (l. 1). The `BrakingSystem` is modified (l. 2) by adding a new input for the lateral acceleration (l. 3) and replacing the traction control subcomponent `tc` with subcomponent `esc` (l. 4). Finally, the new input is connected to the new component (l. 5).

A product configuration for a concrete product is a set of deltas which must be applied to the core model. Listing 1.3 gives an example of a product configuration for a motorbike (l. 2) which is equipped with traction control (TC, l. 3) and an

```
deltaconfig StreetMotorbike {
  TwoWheel,
  TractionControl,
  TwoWheelTC,
  ElectronicStabilityControl,
  TwoWheelESC
}
```

Listing 1.3. Product configuration of a Street Bike with TC and ESC.

electronic stability control (ESC, l. 5). To adapt TC and ESC to a motorbike, additional deltas (ll. 4, 6) are needed.

3 Product Generation

Product generation in Δ-MontiArc is the process of generating a concrete product architecture by applying selected deltas to a given core architecture. The product generator of Δ-MontiArc processes three different kinds of input models. As shown in [9], at first a product configuration is needed that determines a selection of deltas to be applied for a concrete product architecture. Second, a MontiArc architecture model for the core architecture is required and, third, Δ-MontiArc delta models determine variants of the core architecture.

Product Generation Process. Product generation is performed in four steps. At first, MontiArc models for the core architecture are loaded, and their corresponding abstract syntax tree is stored. In the second step, a product configuration is parsed, and the delta models contained in the configuration are loaded. The *generation order* of the selected delta models is computed based on the given application order constraints. When a linear generation order is determined, delta models are applied to the core architecture in the third step of product generation. All modification operations of the deltas are applied stepwise to the core architecture. To assure the definedness of the generated product architecture, the applicability of the modification operations needs to be ensured. The following *applicability conditions* [9] are necessary for the delta operations add, remove, or modify:

- A component c can only be modified, if c exists.
- An architectural element ae must not be added to component c, if c already contains ae.
- An architectural element ae must not be removed from component c, if c does not contain ae.
- A port p must not be removed from component c, if c contains a connector with p as its source or target.
- A subcomponent sc must not be removed from component c, if c contains a connector that has a port of sc as its source or target.

The application order constraints capture dependencies between deltas to ensure the validity of the applicability conditions. If the current delta modification operation satisfies the given applicability conditions, it is applied to the core model. After all delta modifications are applied, MontiArc context conditions are checked for the generated architecture that ensure its internal consistency (see [10] for a complete list of MontiArc context conditions). In contrast, the intermediate architectures are not required to be valid MontiArc architectures.

Checking of Applicability Conditions. The checking of the applicability conditions is closely connected to the product generation process as the applicability of one delta operation depends on the intermediate product architecture resulting from the application of all former delta operations. When generating a concrete product architecture, the respective product configuration defines which deltas are applied, and a possible generation order can be derived and checked. However, when it should be established that all possible product configurations satisfy the applicability conditions, all possible (intermediate) architectures have to be considered. In a naive product-based analysis, all product architectures are generated and analyzed separately. Thus, the same intermediate product architectures which occur in several products during product generation are repeatedly regenerated.

4 Family-based Analysis of Applicability Conditions

In a family-based analysis, the core architecture model and the delta models of a product line are analyzed only once, without generating all possible product architectures by applying the respective delta models to the core architecture explicitly. Instead of repeatedly generating intermediate products, intermediate products are reused which is more efficient than a naive product-based analysis.

Family Application Order Tree. In order to check the applicability constraints by a family-based analysis, a *family application order tree (FAOT)* is created. The $FAOT$ for the example introduced in Section 2 is shown in Figure 1. In a $FAOT$, the nodes represent the deltas of the product line. Each path in the $FAOT$ starting from the root is a generation order that is valid according to the application order constraints attached to the deltas. The root node corresponds to the core architecture indicating that no delta has yet been applied and combines the forest of possible generation orders into a tree. Leaves of the $FAOT$ correspond to maximal possible generation orders, where the addition of another delta will violate the application order constraints of the deltas on the path to the leaf. To each node in the $FAOT$, an architecture is associated that is generated by applying the deltas leading to this node including the node itself to the core architecture. This architecture is either a product architecture that is valid according to the MontiArc context conditions or an intermediate architecture.

The applicability conditions of the deltas can be checked by traversing all paths in the $FAOT$ and establishing the applicability conditions for each modification operation encountered. In this way, all possible product architectures are

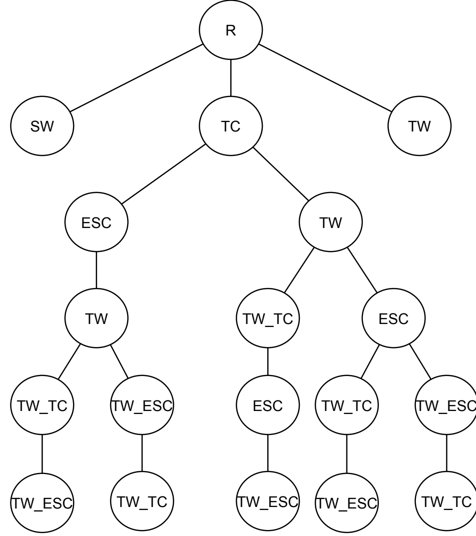

Fig. 1. *FAOT* for the example product line

analyzed that can be generated by a (sub-)path in the *FAOT*. Due to a large number of deltas and a sparse set of application order constraints, the *FAOT* can be fairly complex. Therefore, the efficient computation of the intermediate products that are necessary to traverse the *FAOT* is essential. Two approaches can be distinguished:

1. The intermediate architectures for certain tree nodes are stored such that they can be reused for calculating further intermediate products which, however, requires a hugh amount of memory for large architecture models.
2. Inverse deltas can be applied to a generated architecture to undo the application of a delta in order to backtrack in the *FAOT* without storing intermediate architectures.

Inverse Deltas. For each delta model D consisting of a set of delta operations, there is an inverse delta model D^{-1} such that for any product architecture P, it holds that $apply(apply(P, D), D^{-1}) = P$ where application of the modification operations in a delta is defined by $apply : Arch \times Delta \rightarrow Arch$ for $Arch$ the set of MontiArc architectures and D the set of delta models in Δ-MontiArc. An inverse delta D^{-1} is derived from a delta D by inverting each delta modification operation in D and also inverting the ordering of the modification operations. For each `add` statement, the inverse operation is a `remove` statement and, vice versa. The operations `connect` and `disconnect` statements are inverses for each other. The enclosing component modification operations remain unchanged. An example for an inverse delta is shown in Listing 1.4.

FAOT Analysis Using Inverse Deltas. Using inverse deltas, the *FAOT* can be traversed in a depth-first manner without storing any intermediate architectures

```
delta A {                        |  delta A_Inverse {
  modify component Base {        |    modify component Base {
    add port Integer p;          |      disconnect p -> sub.input;
    connect p -> sub.input;      |      remove port p;
  }                              |    }
}                                |  }
```

Listing 1.4. Inverting the delta on the left side results in the delta on the right side.

at the *FAOT* nodes. For the *FAOT* depicted in Figure 1, the following depth-first traversal is computed: $SW \rightarrow SW^{-1} \rightarrow TC \rightarrow ESC \rightarrow TW \rightarrow TW_TC \rightarrow TW_ESC \rightarrow TW_ESC^{-1} \rightarrow TW_TC^{-1} \rightarrow TW_ESC$...

The applicability conditions are checked by processing the deltas during the traversal one by one. After processing one delta, its inverse is computed, if necessary, and stored for later application. Since an inverse delta depends on the intermediate architecture to which the original delta is applied, it is not possible to compute all required inverse deltas up front.

5 Discussion

Comparing the family-based analysis using inverse deltas to an analysis in which all intermediates architectures at decision-nodes in the *FAOT* are stored (intermediate approach), the inverse delta approach requires less memory. In large product lines where the *FAOTs* contain many decision nodes, memory space might become a severe problem, as every intermediate architecture that has to be stored comprises the ASTs of the modified core architecture. Compared to the naive product-based approach, the inverse delta approach uses the same amount of memory, as in both approaches no intermediate products are stored.

Regarding runtime complexity, the worst case is if there are n deltas without application order constraints. Then, every possible permutation of deltas is contained in the *FAOT*. The amount of edges in a *FAOT* is $AE(n) = \sum_{i=0}^{n-1} \frac{n!}{i!} = n! * \sum_{i=0}^{n-1} \frac{1}{i!}$. The intermediate approach computes every delta once, such that $AE(n)$ steps are needed to check every possible product. The inverse delta approach visits every edge twice, once applying a delta, and once applying its inverse. The most right path of the *FAOT* is visited only once. Thus, in the inverse delta approach $2 * AE(n) - n = n! * (2 * \sum_{i=0}^{n-1} \frac{1}{i!} - \frac{1}{(n-1)!})$ steps are necessary where $\sum_{i=0}^{\infty} \frac{1}{i!} = e$ is a constant factor, and for $n \rightarrow \infty$, the term $\frac{1}{(n-1)!}$ converges to zero. Hence, both factors, as well as the constant factor of 2, may be neglected for an estimation of complexity such that the inverse delta approach as well as the intermediate approach belong to complexity class $O(n!)$ The naive product-based analysis approach generates $n!$ products by applying n deltas for each product. In total, $n * n!$ delta applications are performed leading to a complexity of $O(n*n!)$. This yields that the family-based analyses are about n times faster in the worst-case than the product-based analysis. Nevertheless,

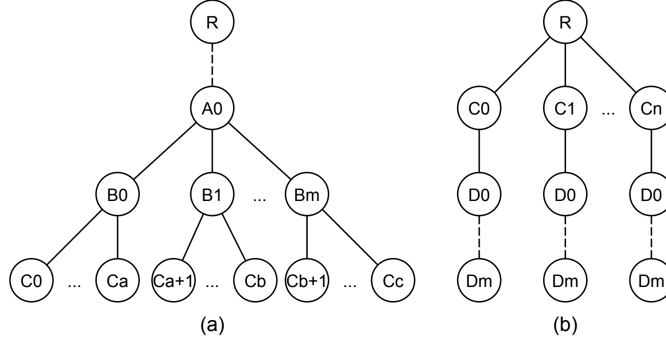

Fig. 2. *FAOTs* with late and early decision-nodes

a complexity of $O(n!)$ is still very high, but this is accounted to the inherent complexity of family-based analyses.

The shape of the *FAOT* influences the number of inverse deltas that are required to get from one leaf to the next. Figure 2 shows two examples. Tree `(a)` contains many decision-nodes close to the leafs. So deriving $C1$ based on $C0$ is done in two steps by applying the inverse delta $C0^{-1}$ and afterwards delta $C1$. In contrast, tree `(b)` contains only one decision-node (the root). To get from the product on the very left to the next product whose path starts with $C1$, m inverse deltas have to be applied. As the root node is the only decision-node, no intermediate architectures have to be stored such that the inverse delta approach is about $2 * m$ steps slower without saving any memory. Accordingly, we suggest a hybrid approach that considers the shape of the *FAOT* and stores intermediate architectures at selected decision-nodes. This way, some backtracking steps with inverse deltas can be omitted such that a balance between memory consumption and runtime effort can be achieved. For example, consider the worst-case *FAOT* and assume that we store intermediate architectures at the last decision-nodes before the leaves. On level $n-2$ of the *FAOT*, each node has two children which each has one child that are leaves, since there are only 2 more deltas left which have to be applied. Storing these intermediate architectures saves 4 inverse delta applications for each of the nodes on level $n-2$, except for the most right node where only 2 steps will be saved. On level $n-2$, we have $\frac{n!}{2}$ nodes such that a reduction of $4 * \frac{n!}{2} - 2 = 2n! - 2$ inverse delta applications can be achieved with only minor increase in memory consumption.

6 Related Approaches

Architectural variability modeling approaches can be classified into annotative, compositional and transformational modeling approaches. Annotative approaches, e.g., [6], consider one model representing all products and define which parts of the model are removed to derive a product model. Compositional approaches, e.g., [1], associate model fragments with product features that are

composed for a particular feature configuration. Transformational approaches, such as CVL [12], represent variability of a base model by rules describing how a base model is transformed in order to obtain a particular product model. Δ-MontiArc can be classified as a transformational approach.

Product line analyses can be classified in three main categories [18]: first, product-based analyses consider each product variant separately. Second, feature-based analyses consider the building blocks of the different product variants in isolation to derive results about all variants, but in general rely on heavy restrictions on the admissible product line variability. Third, family-based analyses check the complete code base of the product line in a single analysis to obtain a result about all possible variants.

Family-based product line analyses are currently used for type checking [2, 7] and model checking [5, 8, 14] of product lines. The approach presented in [2] also constructs all possible application orders of feature modules (which are comparable to delta models in our approach) and checks that in any possible combinations of feature modules all required references are provided. The type checking approach proposed in [7] uses a constraint-based type system where a large formula is constructed from the product line's feature model and the feature module constraints that is true if all product variants are type safe. In [17], the type safety of all product variants is checked based on the analysis of a product abstraction that is generated from constraints derived for delta modules. Thus, it can be classified as an mixture between product- and feature-based analyses.

Storing only the differences between products, as we do with inverse deltas, is also applied in versioning systems. For instance, the Revision Control System (RCS) [19] only keeps the most recent version and a sequence of inverse modifications in order to retrieve prior versions which is more efficient than working with complete version snapshots. The formalization of DARCS patch theory [13] has a concept of inverses although on a fairly abstract level. In recent work, Batory et al. [3] apply the idea of differencing for updating a program obtained by feature-oriented composition. However, in that approach it is unclear whether the differences can be expressed by means of feature modules, while in the approach presented in this paper, inverse deltas can be expressed by the same linguistic means as ordinary deltas.

7 Conclusion

The family-based analysis to validate Δ-MontiArc product lines is an extension of our previous work [11, 9]. In this paper, we have introduced the concept of inverse deltas that allows traversing the FAOT without storing intermediate architectures. For future work, we are planning to evaluate the proposed approach at large case examples. Furthermore, we will extend the inverse delta approach to deal with the convenience operations presented [9], such as the replacement of components, where there is no obvious inverse.

References

1. S. Apel, F. Janda, S. Trujillo, and C. Kästner. Model Superimposition in Software Product Lines. In *International Conference on Model Transformation (ICMT)*, 2009.
2. S. Apel, C. Kästner, A. Grösslinger, and C. Lengauer. Type safety for feature-oriented product lines. *Automated Software Engineering*, 17(3):251–300, 2010.
3. D. Batory, P. Höfer, and J. Kim. Feature interactions, products, and composition. In *Proc. of GPCE'11*, 2011. (to appear).
4. D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract Delta Modeling. In *GPCE*. Springer, 2010.
5. A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines). In *ICSE 2010*, 2010.
6. K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *GPCE*, 2005.
7. B. Delaware, W. Cook, and D. Batory. A Machine-Checked Model of Safe Composition. In *FOAL*, pages 31–35. ACM, 2009.
8. A. Fantechi and S. Gnesi. Formal Modeling for Product Families Engineering. In *Software Product Line Conference (SPLC)*, 2008.
9. A. Haber, T. Kutz, H. Rendel, B. Rumpe, and I. Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *1st International Workshop on Software Architecture Variability SAVA 2011*, 2011.
10. A. Haber, T. Kutz, J. O. Ringert, and B. Rumpe. MontiArc - Architectural Modeling Of Interactive Distributed Systems. Technical report, RWTH Aachen University, 2011. (to appear).
11. A. Haber, H. Rendel, B. Rumpe, and I. Schaefer. Delta Modeling for Software Architectures. In *MBEES*, 2011.
12. Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. Olsen, and A. Svendsen. Adding Standardized Variability to Domain Specific Languages. In *SPLC*, 2008.
13. J. Jacobson. A formalization of Darcs patch theory using inverse semigroups. Technical Report CAM report 09-83, UCLA, 2009.
14. K. Lauenroth, K. Pohl, and S. Toehning. Model checking of domain artifacts in product line engineering. In *ASE*, pages 269–280, 2009.
15. I. Schaefer. Variability Modelling for Model-Driven Development of Software Product Lines. In *VaMoS*, 2010.
16. I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented Programming of Software Product Lines. In *SPLC*. Springer, 2010.
17. I. Schaefer, L. Bettini, and F. Damiani. Compositional type-checking for delta-oriented programming. In *Intl. Conference on Aspect-oriented Software Development (AOSD'11)*. ACM Press, 2011.
18. T. Thüm, I. Schaefer, M. Kuhlemann, and S. Apel. Proof Composition for Deductive Verification of Software Product Lines. In *Workshop on Variability-intensive Systems Testing, Validation and Verification (VAST 2011)*, 2011.
19. W. F. Tichy. Design, implementation, and evaluation of a revision control system. In *Proceedings of the 6th international conference on Software engineering*, ICSE '82, pages 58–67, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

Complexity Metrics for Software Product Lines

Xiaorui Zhang[1,2], Øystein Haugen[1], and Birger Møller-Pedersen[2]

[1]SINTEF, Pb. 124 Blindern, 0314 Oslo, Norway
[2]Department of Informatics, University of Oslo, Pb. 1080 Blindern, 0316 Oslo, Norway
Xiaorui.Zhang@sintef.no, Oystein.Haugen@sintef.no, birger@ifi.uio.no

Abstract. This paper proposes a metric suite for assessing the complexity of software product lines. The Common Variability Language (CVL), a generic variability modeling language, has been proposed as an approach for software product line development. A CVL model specifies both the variability of the product line and its variability implementation details for product realization. Our metric suite evaluates the complexity of CVL-based product lines in two dimensions: (1) the complexity of variability specifications, based on how many products can be derived from it. We believe that a CVL model with more products has more variability accounted for, therefore in need of greater effort to develop and maintain; (2) the complexity of variability implementations, based on how much effort is required to develop them. The application of the metric suite is illustrated with product line case studies.

Keywords: Common Variability Language, software metrics, Software Product Line

1 Introduction

Software metrics has been widely used by developers and managers to assess the quality of software products and processes. Software Product Line (SPL) has been increasingly adopted in industry to produce a set of software-intensive systems sharing a common, managed set of features [2]. The increased adoption of SPL in practice has also increased the demand for metrics measuring SPL artifacts and processes.

Several SPL metrics have been proposed mainly on: (1) evaluating the underlying architecture of the SPL in terms of tailorability, architectural requirement conformance and etc [6]; (2) assessing the complexity of the variability specification of the product line, such as counting variation points, calculating the cyclomatic complexity of variation points [5, 7]; (3) evaluating the complexity of a SPL based on the costs, schedule, asset development, quality, productivity and etc [13]. Despite of the existing work, we see the lack of studies on metrics assessing the complexity of SPLs based on both variability specification and implementation.

In our earlier work [11], we proposed the Common Variability Language (CVL) [3, 4] being standardized at Object Management Group (OMG), as an approach for software product line development. CVL is a generic language for modeling variability that can be applied to any model which is defined in any Meta Object

Facility [9] (MOF)-based language. With the CVL approach, the SPL developer first chooses a base model in the domain. The developer then specifies in a CVL model, not only the variability of the product line relative to the base model, but also the corresponding variability implementations in terms of executable CVL operations in order to derive final product models.

We see that such CVL characteristics on dealing with both variability specification and implementation can be exploited to develop an improved complexity metric for SPL. In this paper we propose a metric suite that valuates the complexity of CVL-based product lines in two dimensions: (1) the complexity of variability specifications, based on how many products can be derived form it, which depends on how variability is specified. We believe that this is an indicator for the complexity, since there are more variants to develop and manage; (2) the complexity of variability implementations, in terms of how much effort are required to develop and maintain them.

We believe that our metric suite which covers both aspects will provide an improved evaluation of the SPL complexity. Such metrics can be of great help for SPL developers and managers when they make design or planning decisions. It is also, to the best of our knowledge, one of the very few metrics that are dedicated for model-driven software product line development which apply to models instead of source code. In addition, the metric suite bases itself on the CVL approach, which can be regarded as a contribution to the CVL standardization effort as well. We perform preliminary evaluation of the metric suit by applying it to two product line cases that are taken from industry.

The remainder of the paper is organized as follows: Section 2 gives an introduction of adopted technologies. Section 3 gives a detailed description of the metric suite. Section 4 gives an application of the metrics to product lines. Section 5 discusses some of the issues and challenges regarding the metrics. Related work is summarized in Section 6. Section 7 concludes the paper and proposes the future work.

2 Background

2.1 Common Variability Language

CVL is a generic language to specify variability in any model that is defined in any Meta Object Facility (MOF)-based language. A CVL model comprises three types of model [4]: (1) the base model, created in the base language; (2) the variability model, which specifies the variability relative to the base model; (3) the resolution model, which resolves the variability in the variability model. The final product models are generated by executing the full CVL description.

To develop an SPL using CVL, the SPL developer starts with choosing a model defined in the base language as the base model for the product line. Then the developer creates a CVL model with the variability of the product line relative to the base model defined. More specifically the variability can be specified in two layers [3]:

Feature specification layer specifies the high level variability of the product line relative to the base model, which is analogous to feature modeling. The CVL construct *CompositeVariability* is used to model features. The CVL construct *Iterator* alone is sufficient to express *mandatory/optional*, *OR/XOR* and *multiplicity* among features.

Product realization layer defines the implementations of the variability in the feature specification layer in terms of low-level CVL operations, which will be executed to derive product models from the base model: (1) *Value Substitution*, to change the value of an attribute of a model element; (2) *Reference Substitution*, to redirect a reference from one model element to another; (3) *Fragment substitution*, to substitute a given set of model elements *(placement fragment*) with another arbitrary set of model elements *(replacement fragment*) defined within the same base language. Any arbitrary model fragment can be defined using the CVL concept *boundary element*. The *boundary elements* record all references to and from the model fragment. As illustrated in Fig. 1 [12], *ToP, FrP1 and FrP2* define the *placement fragment*, whereas *ToR, FrR1* and *FrR2* define the *replacement fragment.* During the *fragment substitution,* the *boundary elements* representing the *replacement fragment* are bound to the *boundary elements* representing the *placement fragment* accordingly. As shown in Fig. 1, *ToR* is bound to *ToP, FrR1* is bound to *FrP1* and *FrR2* is bound to *FrP2.*

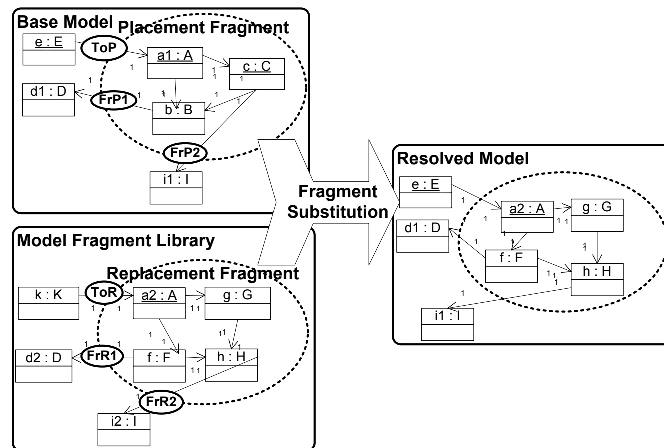

Fig. 1. Fragment substitution in CVL

2.2 Train Control Language

Train Control Language (TCL) [10] is a Domain Specific Language (DSL) developed by SINTEF in cooperation with ABB, Norway[1]. With the TCL language and tools, the train control experts can specify railway station models according to the structural drawings sent by the railway authorities. Interlocking source code can be generated

[1] http://www.abb.no

from TCL models. Such code is deployed Programmable Logic Circuits (PLC) to control station-related machinery. Fig. 2 illustrates the concrete syntax of TCL with annotations. A *train route* is a route between two *main signals* in the same direction. A *track circuit* is the shortest segment where the presence of a train can be detected. A *track circuit* consists of *line segments* and *switches* connected by *endpoints*.

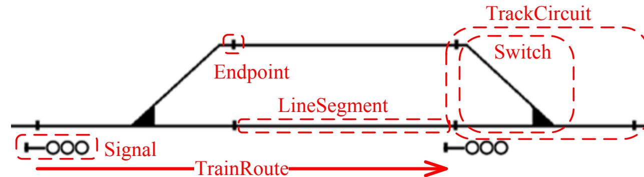

Fig. 2. Annotated basic TCL concrete syntax

3 CVL Complexity Metrics

The metric suite aims at assessing the complexity of a CVL-based product line in two dimensions: the variability specification complexity and the variability implementation complexity. The CVL model is the prime subject of the metrics, so that all the measurements are taken from properties of the CVL model.

3.1 Variability Specification Complexity

Metric 1 **VSC (Variability Specification Complexity)**

Assumption **The number of all possible products is an indicator for the complexity of the variability specification.**

The high level variability (features) of the product line is defined in the feature specification layer of a CVL model. Products are configured by choosing different legal combinations of the features. Therefore the number of possible products depends on how much variability is specified in this layer. For example, it probably indicates: (1) more features are defined, which adds to the complexity of the variability specification; (2) more choices over features are allowed, by having more iterators representing choices and imposing more complex hierarchies to the features.

Definition Consider a CVL model with variability defined. Let *NOP* be the actual number of all possible products allowed by the model. Then:

$$VSC = NOP$$

The range of this indicator is $[0, \infty]$. CVL model with *VSC* equals 0 does not have a valid variability specification so that no product can be derived from the model. Higher *VSC* value indicates more possible products, thus reflects a more complex variability specification, which in turn is need of more effort in development and maintenance.

3.2 Variability Implementation Complexity

Metric 2 WVS (Weighted Value Substitutions)

Assumption **The overall complexity of developing the value substitutions contributes to the complexity of the variability implementations.**

The value substitution is to change the value of an attribute of a model element. It is a fairly straightforward operation. The developer just needs to point to the target attribute of the target model element in order to indicate the *placement value,* and then provide a *replacement value* while configuring a product afterwards. Hence, when the internal complexity of one single value substitution can be regarded minimal, the number of them plays more significant role contributing to the complexity of the entire variability implementations.

However, it is possible that in certain domains, the development effort for each value substitution can vary because of domain specific reasons. For example, the developer may put more effort to decide on a particular *Placement Value* because of the complexity of the domain and the base model. Therefore, we believe that not only the number of the value substitutions alone, but the overall complexity of developing the value substitutions contributes to the complexity of the variability implementations.

Definition Consider a CVL model with value substitutions $vs_1,..., vs_n$ defined in the product realization (variability implementation) layer. Let $cv_1,..., cv_n$ be the complexity of developing them. Then:

$$WVS = \sum_{i=1}^{n} cv_i$$

If all the value substitution complexities are considered to be unity, then $WVS = n$, the number of the value substitutions.

Metric 3 WRS (Weighted Reference Substitutions)

Assumption **The overall complexity of developing the reference substitutions contributes to the complexity of the variability implementations.**

The reference substitution is to redirect a reference from one model element to another one. The developer needs to point to the reference for change and the target object to which it is referred afterwards. Similar to the case of value substitutions, the number of reference substitutions contributes to the complexity of the entire variability implementations. In addition, the complexity of developing each one may differ in certain cases, which is also taken into account in our metric design.

Definition Consider a CVL model with reference substitutions $rs_1,..., rs_n$ defined. Let $cr_1,..., cr_n$ be the complexity of developing them. Then:

$$WRS = \sum_{i=1}^{n} cr_i$$

If all the reference substitution complexities are considered to be unity, then $WRS = n$, the number of the reference substitutions.

Metric 4 WFS (Weighted Fragment Substitutions)

Assumption 1 **The overall complexity of developing the fragment substitutions contributes to the complexity of the variability implementations.**

Fragment substitution is regarded as the most essential and sophisticated CVL operation. The overall complexity of all fragment substitutions in a CVL model definitely contributes to the complexity of its variability implementations.

Assumption 2 **The number of the bindings is an indicator for the complexity of developing a fragment substitution.**

As illustrated in Fig. 1, fragment substitution is to replace an arbitrary model fragment (*placement fragment*) with another arbitrary model fragment (*replacement fragment*) that is defined in the same base language. The boundary elements *to/from placement* representing the placement fragment need to be correctly bound to those *to/from replacement* representing the replacement fragment. The CVL tooling provides certain intelligence to reduce the manual effort needed to deal with those bindings. It suggests default binding choices based on the base language definition and the type of the element to which the boundary element is pointed. Nevertheless, normally most of the development effort for fragment substitution is put on fixing the bindings, such as inspecting default bindings and performing changes if necessary. Hence we believe that in general the more bindings are involved in a fragment substitution, the greater development effort is needed.

Definition Consider a CVL model with fragment substitutions $fs_1,..., fs_n$ defined. Let $nob_1,..., nob_n$ be the number of bindings involved in each substitution. Then:

$$WFS = \sum_{i=1}^{n} nob_i$$

4 Assessing the Complexity of Station Product Lines

To evaluate the feasibility of our work, we have extended the CVL tool with the functionality for calculating the metrics. The metric suite was applied to a regional station product case study of our earlier work [12]. The regional station product line is developed using the CVL approach and the base mode is defined by the TCL language. The base model of the product line is a basic two track station as shown in the top left pane of Fig. 4 and the CVL model is shown in Fig. 3.

As shown in Fig. 3 [12], a regional station can be designed for either urban or rural areas. A rural station is allowed to have one additional track and/or one parking track compared to the base model station. An urban station can only have two tracks as in the base model but can also choose to have one parking track if needed.

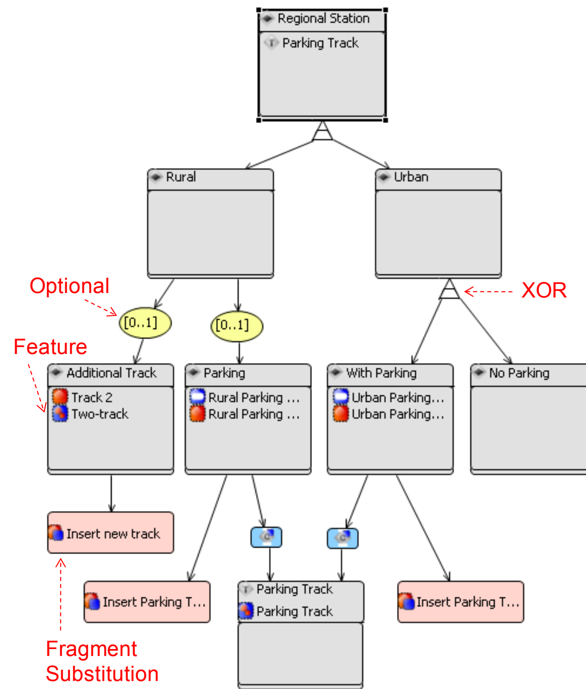

Fig. 3. The CVL model of the regional station product line in the CVL graphical editor with annotations

Our tool takes the CVL model as input and returns with the following calculation of the metric suite:

– *Metric 1*: Number of all the products (NOP) = 4, Variability Specification Complexity (VSC) = 4.
– *Metric 2*: Number of value substitutions n = 0, Weighted Value Substitutions (WVS) = 0.
– *Metric 3*: Number of reference substitutions n = 0, Weighted Reference Substitutions (WRS) = 0.
– *Metric 4*: Number of fragment substitutions n = 3, Weighted Fragment Substitutions (WFS) = Number of bindings for *Insert new track* + Number of bindings for *Insert parking track* (1) + Number of bindings for *Insert parking track* (2) = 27 + 12 + 12 = 51.

Take the fragment substitution *Insert new track* for example: as shown in Fig. 4, the placement fragment is annotated by the dashed rectangle in the top left pane while the replacement fragment is annotated in the bottom left pane. The right pane shows the bindings in the fragment binding editor, which has been counted to 27 in the calculation.

In addition, we apply the metric suite to another station product line [11] also developed with TCL and CVL. All its product models are real stations in Norway. Both product lines were developed by us with ABB, Norway.

The development of the latter product line was more complicated and time-consuming even though we gained experience from developing the previous one – the

regional station product line. This is probably because this product line has more variability defined and leads to more products. This product line is regarded as a more complex one by developers and domain experts.

The result of metrics are: VSC = 52; WVS = 23 (all the value substitution complexities are considered to be unity); WRS = 0; WFS= 69, suggesting a higher complexity in every dimension than the regional station product line, which confirms to the perception of the developers.

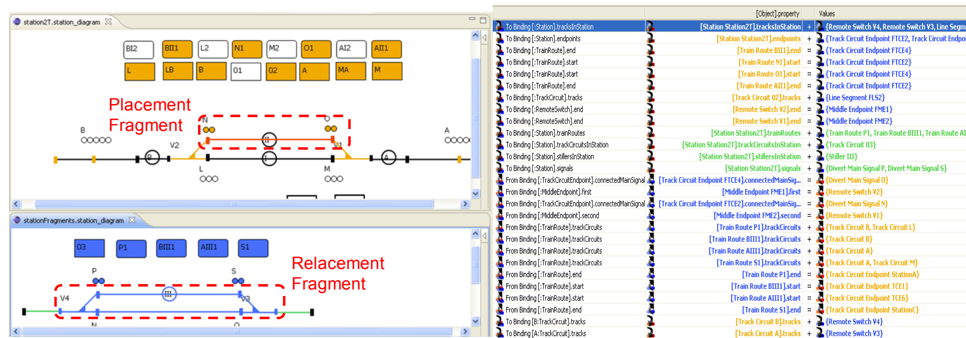

Fig. 4. For the fragment substitution *Insert new track*: placement/replacement fragment annotated in the base and library model, bindings in the CVL fragment binding editor

5 Discussion

Our metric suite is based on the CVL approach. Unlike other metrics which have dependencies on different variability implementation techniques, our metrics benefit from generic CVL substitutions for handling variability implementations with different base languages.

Our metrics evaluates the complexity of product lines based on both the variability specifications and the variability implementations. This provides the SPL developers or managers a more complete overview of the product line complexity to assist their decision making.

Some open issues are identified for the metric development [7], such as exploring the range for the complexity values, which can be further used to categorize product lines into different complexity levels. It is also important to study any possible limitations of our metrics due to its additive nature. Extensive case studies and empirical data collection need to be done to address these issues.

6 Related Work

Several SPL metrics have been proposed, which are partly summarized as follows:

Van der Hoek et al. [6] present a set of metrics to evaluate the SPL architecture. The metrics are based on the concept of service utilization and consider the context in

which individual architectural elements are placed. This work has different focus on the subject of the metrics from ours. Our metrics focus on the variability specification and implementation while their prime subject is the structure of the SPL architecture.

Her et al. [5] propose a framework for evaluating reusability of core asset in product line engineering. Metrics are proposed to evaluate the functional/non-functional commonality, variability richness, applicability, tailorability and other properties of the core asset in product lines, which again have different prime subjects from our metrics.

Our *Metric 1 Variability Specification Complexity (VSC)* was first inspired by the work of Lopez-Herrejon et al. [7]. They adapt McCabe's metric [8] which is originally used for calculating cyclomatic complexity and apply it to assess the cyclomatic complexity of variation points in the product lines. In a CVL model, the complexity of variability specifications does not solely depend on its structural complexity, but also on the CVL constraints across the features. Therefore we considered it not sufficient to apply the cyclomatic complexity metric to CVL models. We further got inspired by the metric suite for object oriented design from Chidamber et al. [1], which is to quality the complexity of an object oriented design based on much of counting class members and weighted aggregation. We adapted part of its basic ideas which are applicable in our case during the development of our metric suite.

To the best of our knowledge, we are not aware of any existing metrics providing evaluation of the SPL complexity based on both variability specification and implementation, as well as being generic benefiting from the CVL approach.

7 Conclusion and Future Work

In this paper, we proposed a metric suite for evaluating the complexity of product lines. Our metrics apply to the CVL-based product line development. It benefits from the generic nature of the CVL approach thus can work with any product line with a MOF-based base language. Our metric suite comprises four metrics, assessing the complexity of an SPL based on the complexities of variability specification and implementations. A CVL tool extension has been made to calculate the metrics and applied to two product line case studies.

Ideas for future work include: (1) deciding the complexity value range and complexity categorization based on empirical data collection; (2) exploring the possible limitations of the metrics due to its additive nature from extensive real case studies.

Acknowledgements. The work presented here has been developed within the MoSiS project ITEA 2 – ip06035 part of the Eureka framework and the CESAR project funded by ARTEMIS Joint Undertaking grant agreement No 100016.

References

1. Chidamber, S.R. and Kemerer, C.F.: A Metrics Suite for Object Oriented Design. IEEE Trans. Softw. Eng. 20, 476-493 (1994)
2. Clements, P. and Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley Longman Publishing Co., Inc., (2001)
3. Fleurey, F., Haugen, Ø., Møller-Pedersen, B., Olsen, G.K., Svendsen, A., and Zhang, X., "A Generic Language and Tool for Variability Modeling," SINTEF, Oslo (2009),
4. Haugen, O., Møller-Pedersen, B., Oldevik, J., Olsen, G.K., and Svendsen, A., "Adding Standardized Variability to Domain Specific Languages," the 13th International Software Product Line Conference, Limerick, Ireland, (2008)
5. Her, J.S., Kim, J.H., Oh, S.H., Rhew, S.Y., and Kim, S.D.: A Framework for Evaluating Reusability of Core Asset in Product Line Engineering. Inf. Softw. Technol. 49, 740-760 (2007)
6. Hoek, A.v.d., Dincel, E., and Medvidovic, N., "Using Service Utilization Metrics to Assess the Structure of Product Line Architectures," in Proceedings of the 9th International Symposium on Software Metrics: IEEE Computer Society, (2003), pp. 298
7. Lopez-Herrejon, R.E. and Trujillo, S., "How Complex Is My Product Line? The Case for Variation Point Metrics," VaMoS workshop, (2008)
8. McCabe, T.J., "A Complexity Measure," in Proceedings of the 2nd international conference on Software engineering. San Francisco, California, United States: IEEE Computer Society Press, (1976), pp. 407
9. MOF, "The Metaobject Facility Specification." http://www.omg.org/mof/
10. Svendsen, A., Olsen, G.K., Endresen, J., Moen, T., Carlson, E., Alme, K.-J., and Haugen, O., "The Future of Train Signaling," Model Driven Engineering Languages and Systems (MoDELS 2008), Tolouse, France, (2008)
11. Svendsen, A., Zhang, X., Lind-Tviberg, R., Fleurey, F., Øystein, H., Møller-Pedersen, B., and Olsen, G.K., "Developing a Software Product Line for Train Control: A Case Study of Cvl," in Proceedings of the 14th international conference on Software product lines: going beyond. Jeju Island, South Korea: Springer-Verlag, (2010)
12. Zhang, X., Haugen, Ø., and Møller-pedersen, B., "Model Comparison to Synthesize a Model-Driven Software Product Line," in the 15th International Software Product Line Conference. Munich, Germany, (2011)
13. Zubrow, D. and Chastek, G., "Measures for Software Product Lines," (2003),