VARY 2012 Workshop

09:00-10:30 Motivation (Invited speakers)
Krzysztof Czarnecki: Variability modeling in practice
Øystein Haugen, Andrzej Wasowski: CVL Revised Submission – a mini-tutorial
10:30-11:00 Morning Coffee
11:00-12:30 CVL and aspects
João Bosco Ferreira Filho, Olivier Barais, Jérôme Le Noir and Jean-Marc Jézéquel. Customizing the Common Variability Language Semantics for your Domain Models
Benoit Combemale, Olivier Barais, Omar Alam and Jörg Kienzle. Using CVL to Operationalize Product Line Development with Reusable Aspect Models
12:30-14:00 Lunch
14:00-15:30 Synthesizing Feature Models
Steven She, Krzysztof Czarnecki and Andrzej Wasowski. Usage Scenarios for Feature Model Synthesis
Hamzeh Eyal-Salman, Abdelhak-Djamal Seriai and Ra'Fat Al-Msie'Deen. Recovering Traceability links between Feature Models and Source Code of Product Variants
15:30-16:00 Afternoon Coffee and Poster Session
16:00-17:30 Applying CVL
Clara Ayora, Germán H. Alférez, Victoria Torres and Vicente Pelechano. Applying CVL to Business Process Variability Management
Shuai Wang, Arnaud Gotlieb, Marius Liaaen and Lionel Briand. Automatic selection of test execution plans from a Video Conferencing System Product Line modelling

VARY URLs

http://vary2012.irisa.fr/ is the workshop home page

http://variabilitymodeling.org/ is where you can find the CVL Revised Submission and other stuff related to the OMG standardization

http://www.varies.eu/ is the homepage of the sponsor project VARIES

http://www.inria.fr/en/teams/triskell is the homepage of the sponsor team at INRIA

Author Index

Program Committee

Souvik Barat	Tata Research Development and Design Centre
Danilo Beuche	pure-systems GmbH
Krzysztof Czarnecki	University of Waterloo
Sebastien Gerard	CEA, LIST
Hassan Gomaa	George Mason University
Oystein Haugen	SINTEF
Jean-Marc Jézéquel	Irisa (INRIA
University of Rennes)	
Andreas Korff	Atego Systems GmbH
Vinay Kulkarni	Tata Consultancy Services
Jérôme Le Noir	Thales Research and Technology
Jason Mansell	Tecnalia
Jabier Martinez	itemis France
Brice Morin	SINTEF ICT
Birger Møller-Pedersen	University of Oslo
Ran Rinat	IBM
Suman Roychoudhury	Tata Consultancy Services
Patrick Tessier	CEA/LIST
Michael Wagner	Fraunhofer FOKUS
Andrzej Wasowski	IT University of Copenhagen

Additional Reviewers

Sunkle, Sagar

Table of Contents

Customizing the Common Variability Language Semantics for your Domain Models

João Bosco Ferreira Filho
INRIA and IRISA
Université Rennes 1
Rennes, France
joao.ferreira_filho@inria.fr

Olivier Barais
INRIA and IRISA
Université Rennes 1
Rennes, France
barais@irisa.fr

Jérôme Le Noir
INRIA and IRISA
Thales Research & Technology
Palaiseau, France
jerome.lenoir@thalesgroup.com

Jean-Marc Jézéquel
Université Rennes 1
Rennes, France
jezequel@irisa.fr

ABSTRACT

The Common Variability Language (CVL) provides a well-structured mechanism to express variability and to relate this variability to any MOF-compliant model. This characteristic allows users to define the materialization of a given CVL resolution/configuration. Using variation points, it is possible to express and manipulate the links between the variability abstraction model and the base model. However, the meaning of a given variation point can vary according to the semantics of each domain. For example, a variation point that excludes an element in the base model can lead to further operations, like excluding other elements which were associated to the deleted element, or even to reassign references to another model element. Therefore, it is necessary to address this semantic variability in order to align the materialization semantics to the base model semantics. In this paper, we show how Kermeta can be used to easily implement and customize the semantics of the CVL's variation points, according to the semantics of the base model domain.

General Terms

Variability Modelling, CVL, Semantics, Extensibility

1. INTRODUCTION

Variation points are a common structure in software and systems families. These points can indicate choices among different algorithms or different data structures. In the context of Software Product Lines (SPL), the most common way to express these variation points is in a feature model. The first feature model was proposed by Kang and others [10], in 1990, as part of the method Feature-Oriented Domain Analysis (FODA). Since then, several other feature-oriented approaches for variability modeling were proposed based on FODA [4, 5, 6, 9, 13, 11, 12, 17, 19]. These variability modeling approaches are evolving towards supporting a great part of the product line lifecycle, and not only the domain modeling of a product family. As an effort to standardize and promote variability modeling, The Common Variability Language (CVL)[1] provides a well-structured mechanism to express variability and to relate this variability to any model that conforms to the Meta-Object Facility (MOF)[2]. The links to model-based assets can facilitate the derivation process, since that the choices in the variability level can be explicitly mapped to a realization layer and furthermore reflected in the assets level by means of executing a derivation algorithm. In this way, the CVL realization model works as an intermediate layer between the variability abstraction layer (can be seen as the features level) and the assets layer i.e., the set of model-based elements. Using variation points, this layer defines the set of modifications that must be executed over the set of base-model assets, according to selected features in the variability abstraction layer. These modifications have their defined semantics, for example, the *ObjectExistence* variation point defines the presence or absence of a given element in the materialized product model.

The *ObjectExistence* will exclude the binding model element if the related feature is not selected. However, this semantics can vary according to the semantics of the base-model domain. For example, excluding a class of a class diagram may not influence in the existence of the classes related to it, on the other hand, excluding an activity of an activity diagram may break an activity flow. This semantics variation can be identified even in the same base model, e.g., excluding a class attribute has different semantics then excluding an entire package, in the last situation it may lead to exclude all the classes contained in the package, whereas excluding an attribute may not lead to any further exclusions of model elements.

Given this variety of semantics that can be assigned to the same variation point, it is necessary to provide flexibility and adaptability for the materialization engines. Therefore, in this paper, we present how we can easily implement and customize the semantics of the variation points using Kermeta, a Model-Driven Engineering platform that leverages

[1] http://www.omgwiki.org/variability
[2] http://www.omg.org/mof/

the construction of aspects to handle static and dynamic semantics concerns around a metamodel.

The remainder of this paper is organized as follows. Section 2 presents a background about CVL and Kermeta. Section 3 shows motivating examples of how materialization semantics can vary for a given CVL variation point. Section 4 presents how to implement an operational semantics of a variation point and, after, three mechanisms to customize and to implement multiple operational semantics. Section 5 discusses related work and Section 6 presents the conclusions.

2. BACKGROUND

2.1 The Common Variability Language

CVL is a domain-independent language for specifying and resolving variability over any instance of any MOF-compliant metamodel. Also inspired by FODA, CVL has a Variability Abstraction Model (*VAM*), which is the part of CVL in charge of expressing the variability in terms of a tree-based structure. The core concepts of the VAM are the variability specifications (*VSpecs*). The *VSpecs* are nodes of the VAM and can be divided into three kinds: Choices, Variables and Classifiers. Each kind of *VSpec* has its own type of resolution (*VSpecResolutions*). The Choices are *VSpecs* that can be resolved to yes or no (through *ChoiceResolution*), *Variables* are *VSpecs* that require a value for being resolved (*VariableValue*) and *Classifiers* are *VSpecs* that imply the creation of instances and then providing per-instance resolutions (*VInstances*). In this paper, we mainly use the *Choices VSpecs*, which can be intuitively compared to features, in feature models, and can be decided positively or negatively during the product derivation.

Besides the VAM, CVL also contains a Variability Realization Model (VRM). This model makes possible to specify the changes in the base model implied by the *VSpec* resolutions. These changes are expressed as Variation Points in the VRM. Variation Points can mainly express three different types of semantics, which are following described.

- **Existence**. This type of variation point expresses whether an object (*ObjectExistence* variation point) or a link (*LinkExistence* variation point) exists or not in the materialized model.
- **Substitution**. This type of variation point expresses a substitution of a model object by another (*ObjectSubstitution* variation point) or of a fragment of the model by another (*FragmentSubstitution*)
- **Value Assignment**. This type of variation point expresses that a given value is assigned to a given slot in a base model element (*SlotAssignment* variation point) or a given link is assigned to an object (*LinkAssignment* variation point).

In this paper, we claim that it is necessary to customize or specialize the semantics of these variation points and we propose an approach that supports this customization by using different mechanisms. In Figure 1, we present a CVL overview. As illustrated, each variation point can be bound to a *VSpec*, which is referred by a *VSpecResolution*. Depending on the value of the resolution, the corresponding

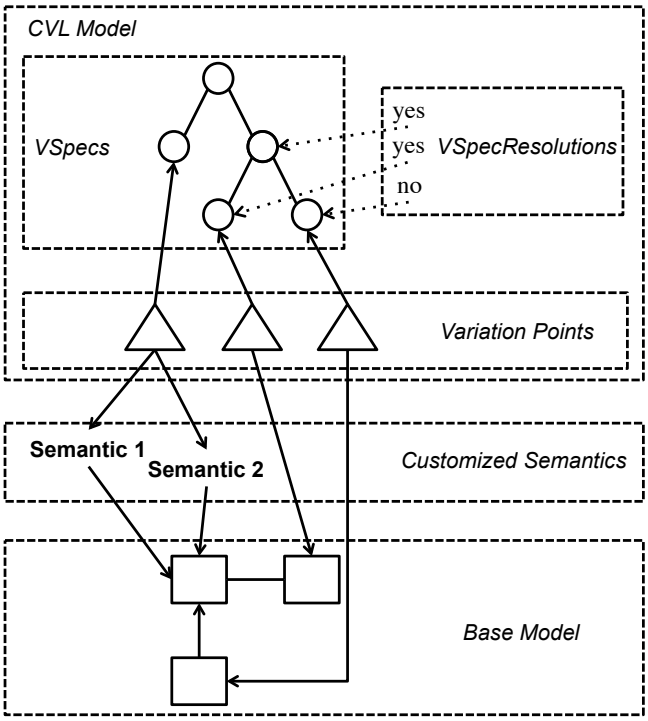


Figure 1: CVL overview with semantics customization.

variation point is executed during the materialization, modifying the corresponding model element. Thus, Figure 1 shows the point where this modification can be customized, making possible to have different semantics for a same variation point.

2.2 Kermeta

Our definition of the CVL semantics and its extension mechanisms are built on top of Kermeta [15]. Kermeta is a language workbench designed for specifying and designing domain-specific languages (DSL). For this, it involves different languages, depending on the concern: abstract syntax (we will also use the term "metamodel" to refer to it[3]), static semantics and behavioural/operational semantics. The workbench integrates the OMG *de facto* standards EMOF and OCL, respectively for specifying the abstract syntax and the static semantics. The workbench also provides the *Kermeta Language* to address the specification of the operational semantics and to integrate an action language in EMOF. The Kermeta Workbench also provides composition operators responsible for composing these different concerns into a standalone execution engine (interpreter or compiler) of the DSL.

The Kermeta language is imperative, statically typed, and includes classical control structures such as blocks, conditionals, loops and exceptions. The Kermeta language also implements traditional object-oriented mechanisms for handling multiple inheritance and generics. The Kermeta language provides an execution semantics to all EMOF constructs that must have a semantics at runtime, such as containment and associations. First, if a reference is part of a

[3]This is one definition in the community. For some researchers, "metamodel" is sometimes referred to abstract syntax plus static semantics.

bidirectional association, then the assignment operator semantics handles both ends of the association at the same time. Second, if a reference is part of a containment association, then the assignment operator semantics unbinds existing references, so that an object is part of another. Finally, for multiple inheritance, Kermeta borrows the semantics from the Eiffel programming language [14].

In Kermeta, all pieces of static and behavioral semantics are encapsulated in metamodel classes. An `aspect` keyword enables DSL engineers to relate the language concerns (abstract syntax, static semantics, behavioral semantics) together. It allows DSL engineers to reopen a previously created class to add some new pieces of information such as new methods, new properties or new constraints. It is inspired from open-classes [2]. The keyword `require` enables the composition itself. A DSL implementation *requires* an abstract syntax, a static semantics and a behavioral semantics. The require mechanism also provide some flexibility with respect to static and behavioral semantics. For example, several behavioral semantics could be defined in different modules (all on top on the same metamodel) and then chosen depending on particular needs (e.g., simulation, compilation). It is also convenient to support semantics variations of the same language. For instance, Kermeta proposes a built-in mechanism to specify several implementations of CVL semantics variation points as shown in Section 4.2.

3. SEMANTICS VARIATION SCENARIOS

In this section, we justify, motivate and illustrate the need for customizing the CVL materialization semantics. Our goal is to show scenarios in which the semantics of the variation point *ObjectExistence* can vary according to the base model semantics. The primary semantics of this variation point is to determine whether a model element exists or not in the materialized model. Considering a negative derivation, this is done by checking the *VSpecResolution* of the binding *VSpec*, if it is set to yes, nothing is done, if it is set to no, the referred model element is excluded. However, we have to consider that excluding a model element may lead to secondary operations to complement the primary semantics of the variation point. This semantics complements, or secondary operations, may vary according to different scenarios.

The first scenario to be considered is that this materialization semantics can vary within the same metamodel. In Figure 2 (a), we have a base model that conforms to the UML Class Diagram metamodel. This model has three classes: *Garage*, *Car* and *Sedan*. The class *Sedan* is a subclass of the class *Car*, which represents a car that can be parked in a garage. Therefore, the class *Garage* represents a place that can accommodate cars. To exemplify the semantics variation of excluding a model element, we remove each class of this model and observe the possible outcome for each class.

Although we exclude the same type of element (Class) in this example, we can see that the semantics of the exclusion operation leads to different possible secondary operations in the model. A possible result from excluding the class *Garage* is shown in Figure 2(b), which is to exclude the class *Garage* and all its relationships. This outcome is reasonable, considering that the other classes are not depending on the class *Garage* to exist. This same scenario can be observed in Figure 2(c). After removing the class *Sedan* , the inheritance

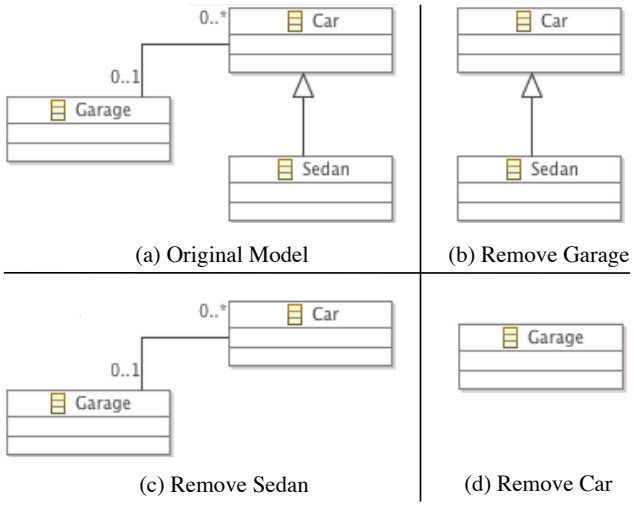


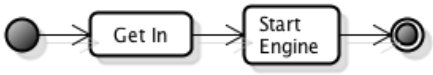
Figure 2: Different semantics for removing a class.

relationship is removed and the other classes remain in the model. On the other hand, as illustrated in Figure 2(d), removing the class *Car* leads to exclude not only itself and its relationships, but also removing its subclasses, in this case, the class *Sedan*.

The semantics of excluding an element can vary for the same type of model elements, but can also vary for different types. Excluding a package of a class diagram can imply on removing all its classes. However, excluding a class attribute may not lead to any further operations.

Another scenario to consider is related to the other kinds of base models, such as the activity diagram of the UML, in which we can observe that the materialization semantics can vary even more. In Figure 3, excluding the *Fasten Seat Belt* activity can imply on four operations: remove *Fasten Seat Belt* model element, remove the income link, remove the outcome link and create a new link from the *Get in* activity to the *Start Engine* activity. This is also discussed in [3].

(a) Original Model



(b) Remove Fasten Seatbelt

Figure 3: Removing an activity.

In fact, one can argue that this semantics variation can be expressed with the CVL standard semantics. However, we believe that such customization is important to leverage abstraction over the base model semantics, encapsulating secondary operations. Therefore, the engineer in charge of designing a CVL realization model can, for example, abstract over tedious operations, such as removing dangling references or excluding contained elements.

4. USING KERMETA TO CUSTOMIZE THE MATERIALIZATION

4.1 Weaving Semantics into CVL

Before showing how different semantics can be weaved into a variation point, we introduce standard pattern used in Kermeta to weave meta-model operational semantics. Using Kermeta, we are able to directly weave operational semantics into any EMF metamodel. Consequently, the CVL metamodel is required as an input and can be easily invoked in Kermeta using the `require` keyword, (*e.g.*, `require CVL-Metamodel.ecore`). It is possible to weave the operational semantics into any model element in the metamodel. The Listing 1 shows how we can simply attach operational semantics into the CVL abstract syntax (metamodel). The code that implements the operational semantics of a model element is placed in an `aspect class` block, named according to the model element name (line 1). In the case of the variation points, we define an abstract operation to evaluate the operational semantics of the variation point (line 2). This operation is abstract because the actual implementation is in the concrete class that inherits from the *Variation Point* class.

Listing 1: Eval method introduction in all the VPs

```
1 aspect class VariationPoint{
2  operation eval(ctx:CVLExecutionContext::
       CVLExecutionContext):Void is abstract
3 }
```

Each concrete variation point has an *eval* method, which overrides the abstract operation and must contain the operational semantics to be executed, following an implementation of the Interpreter Design pattern [8]. An execution context (*CVLExecutionContext*) is provided as a parameter of the *eval* method. This context stores relevant information to the execution of the materialization engine, such as the set of selected/unselected *VSpecs* and the set of model elements in the resolved model.

The Listing 2 presents the operational semantics of the *ObjectExistence* variation point. Firstly, in line 3, it is verified whether the binding *VSpec* of the current variation point is selected or not. If the *VSpec* is not selected for the current materialization, we need to remove the corresponding element in the base model. In line 4, we navigate to the binding object of the current variation point (`self`) to provide the optional element in the base model that is inside the collection *ctx.domainResource* to the method remove.

Listing 2: Excerpt of the ObjectExistence semantics

```
1 aspect class ObjectExistence {
2 method eval( ctx : CVLExecutionContext::
       CVLExecutionContext) : Void is do
3  if (not ctx.decision) then
4   ctx.toRemove.add(self.optionalObject.object)
5  end
6 end
7 }
```

4.2 Customizing the semantics for your domain metamodel

CVL proposes a set of VP with a well-defined semantics and keeps one as an extension point to implement its own semantics: *Opaque Variation Point* (OVP). The OVP is a black box that can define an arbitrary behaviour to execute during derivation. The use of OVPs can be seen as a mechanism to propose a particular semantic for the derivation engine. Besides the OVP, following we propose two other mechanisms to customize the semantics of CVL. The first one is the static introduction of a new semantic and the second one is using the strategy pattern. Finally, we explain how to introduce a home-made semantic using OVPs.

Static introduction of a new semantics

By using the built-in *require* composition mechanism of Kermeta, it is possible to statically customize the semantics of a CVL variation point. Indeed, *require* provides a linearisation mechanism to weave aspect in an existing metamodel. It allows a DSL engineer to reopen a previously created metaclass to add new pieces of information such as new methods, new properties or new constraints. It is inspired from open-classes [2]. It also allows engineers to easily replace the behaviour of an existing method. Indeed, the method (*eval*, see Listing 1), which is introduced in all the variation points (*ObjectSubstitution*, *ObjectExistence*, ...) can be changed by requiring a new Kermeta file. This modification is static, modifying the types and requiring to recompile all the CVL's Kermeta implementation.

This extension mechanism has two main drawbacks. First, Kermeta does not allow the new implementation to call the previous aspect implementation, contrarily as we can call the code of an operation contained in the super-class with the keyword *super*. Secondly, using this mechanism, the DSL engineer can change (and potentially break) completely the CVL implementation. Kermeta does not provide any checker to ensure that a new implementation is a *refinement* of the previous implementation. The main advantages is the fact that the extension is modular and can be statically plugged or unplugged. As an example, we present in Listing 3, an excerpt of a customization of the CVL Object Existence, which, besides removing the model element (line 4), also fixes dangling references (line 5).

Listing 3: Excerpt of the ObjectExistence semantics

```
1 aspect class ObjectExistence {
2 method eval( ctx : CVLExecutionContext::
       CVLExecutionContext) : Void is do
3  if (not ctx.decision) then
4   ctx.toRemove.add(self.optionalObject.object)
5   fixReferences(self.optionalObject.object, ctx
       )
6  end
7 end
8 }
```

Strategy pattern

The basic semantics used in the default CVL implementation is the following. Each variation point can modify the model to change relationships between model elements and can introduce new model elements. To remove a model element, each variation point acts on a context that contains a list *toRemove* of the model elements that must be removed. Removing elements of a base model is performed at the end of the derivation to avoid side-effects among variation points. With this behaviour, CVL combines positive variability and negative variability. The default semantics for this remove implementation is the following. Each element contained (containment relationship) by an element that must be removed is also removed. All the references of an element that must be removed are set to *null*. All the elements, that reference an object that must be removed through a

reference with a violation of the lower cardinality are also removed, *e.g.* if $a : A$ references $b : B$ and A is associated exactly with one B, if b is removed, a is also removed. This can already be seen as a semantic customization as it has an additional load of operations that must be performed for a given variation point. Consequently, we introduce in the default semantics a strategy pattern [8] to provide the ability of dynamically specializing the default semantics. The idea is that a domain expert can define a new CVL semantic extension and can register it. During the derivation, when a model element has to be removed, all the registered extensions are called to determine the list of model elements to be removed (as depicted in Figure 4). To implement a new metamodel extension, the DSL expert has to create an object that respects the following interface (see Listing 4).

Listing 4: Excerpt of the ObjectExistence semantics

```
1 interface ToRemoveStrategy {
2 method remove ( objToRemove: Object, ctx:
      CVLExecutionContext) : Void is abstract
3 }
```

Listing 5: Excerpt of the ObjectExistence semantics

```
1  aspect class ObjectExistence {
2  method eval( ctx : CVLExecutionContext::
       CVLExecutionContext) : Void is do
3   if (not ctx.decision) then
4    ctx.remove(self.optionalObject.object)
5   end
6  end
7  class CVLExecutionContext {
8  method remove( obj:Object) : Void is do
9   self.toremove.add(obj)
10  //Call the strategies
11  self.toremoveStrategies.each{strat | strat.
       remove(obj, self)}
12 end
13 ...
14 }
```

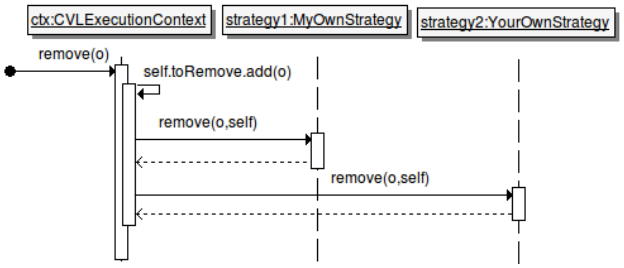


Figure 4: Strategies Sequence Diagram

This extension mechanism provides several benefits. Firstly, it ensures that the default semantics of the CVL variation point is respected. Indeed, domain engineers can only refine the semantics in removing elements, and not directly in the variation point. It can be compared to the idea of post directives in Kompose [7]. Secondly, new strategies can be registered or unregistered dynamically. Finally, each specialization can be modularized in a distinct building block.

Home-made semantics using Opaque Variation Point

The last way to customize the CVL derivation semantics is the use of *Opaque Variation Points* (OVP). An OVP is the Variation Point in which the behavior is defined by using an expression defined in an action language. We currently propose an implementation that supports OVP definition in Groovy [4], in Javascript or in Kermeta. With these action languages, designers can modify the base model directly. Each of this Variation point can access to a context that contains the list of Objects to remove (*toRemove*), the list of *objectHandles* associated to this Variation Point (*ctx*), the list of variable and their associated value defined in the the resolution model (*args*), and a map of key value that can be used to propagate value between the execution of variation points *map* (see Figure 5). An example of OVP is defined in Listing 6 as an expression attribute of the OVP, it adds all the UML properties that references a base model element that must be removed.

Listing 6: OVP example in Groovy

```
1  ctx.each {e  ->
2   org.eclipse.uml2.uml.PackageableElement elem =
        e;
3   org.eclipse.uml2.uml.Package p1 =elem.
        getPckage();
4   p1.getMembers().findAll{m ->
5    m instanceof org.eclipse.uml2.uml.Association
        }
6   .each{m->
7    m.getProperties().entrySet().findAll{ p2 ->
8     p2.getType().equals(e)}.each{ m2 ->
9      notSelected.add(m)
10    }
11  }
12 }
```

The main drawback of this extension mechanism is the own opacity of the OVP. No CVL checker can ensure the correctness of the variability model and it becomes complex to understand the expected behaviour of a variability realization model.

Synthesis

Table 1 shows a comparison of the three extension mechanisms provided in our CVL implementation to support the customization of the CVL semantics for a domain model. We can observe that the second mechanism is generally the best to specialize the CVL semantics for a specific metamodel. Indeed, it does not change the CVL semantics but it only refines the semantics of removing an element in the case of your domain model. Opaque Variation Point is often useful even if we loose the ability to understand the materialization in analizing the CVL model. Besides, it is currently missing in CVL the notion of Opaque Variation Type to ease the reuse of an existing OVP. It probably could also be done using CVL configurable units. The first mechanism is built-in within Kermeta but seems to be dangerous for the case of CVL because experts should be perfectly aware of the previous implementation to change it without introducing side-effects.

[4]http://groovy.codehaus.org/

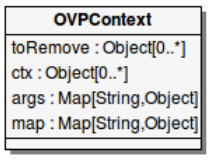


Figure 5: OVP context execution

	Benefits	Drawbacks
Static introduction of new semantics	Modular	No guarantee, highly invasive
New strategy definition	Modular, Dynamic	Can only change the negative derivation semantics
Opaque Variation Points	Flexible	Black Box, uncheckable, no reuse

Table 1: Synthesis of the three extension mechanisms

5. RELATED WORK

Cetina et al [1] presents three strategies for materializing variability. The authors use CVL for modeling runtime variability and they claim that the materialization must be adequated to some extra-functional properties, such as performance, history and persistence of base model changes. Therefore, the authors develop different strategies for materializing the CVL model. However, these strategies are not in the level of semantics of our work, in fact, these strategies share a single semantics and produce a same materialized base model, the only change is in the extra-functional properties.

Perrouin et al [16] proposes an approach for product derivation that provides both automation and flexibility. Instead of using CVL, the authors use the abstract syntax of the Free Feature Diagram (FFD) [19]. To link features to base model elements, the authors extended the FFD with a simple association between a Feature and a Model, limiting the expressiveness of the product derivation. The flexibility of the approach is provided by Kompose [7], which leverages a degree of customization similar to ours.

Schaefer et al [18] uses delta-oriented programming for implementing SPL. In their work, similarly to CVL, an intermediate layer between the features and the artifacts is created. This layer defines delta-modules, which specifies changes in the core model, like variation points in CVL. A key difference from our approach is the model integration provided by the Kermeta workbench, which is not completely addressed in Schaefer's work.

6. CONCLUSION AND FUTURE WORK

Variation points in CVL are elements that express a modification in the target base model. These elements are linked to a variability specification that triggers the modification, depending on their resolution value. The main goal of this paper is to motivate the fact that these modifications can have different semantics, depending on the base model semantics. Thus, we have shown how this semantic can be customized using Kermeta, comparing three different ways of implementing this customization. As future work, we are working on a full customization of CVL for UML and for the Thales [5] Domain Specific Language for Software and System design. We are also working on the use of high level DSLs to express new strategies and define properties that can be checked to ensure the correctness of the customization.

7. REFERENCES

[1] C. Cetina, O. Haugen, X. Zhang, F. Fleurey, and V. Pelechano. Strategies for variability transformation at run-time. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 61–70, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[2] C. Clifton and G. T. Leavens. Multijava: Modular open classes and symmetric multiple dispatch for java. In *OOPSLA*, pages 130–145, 2000.

[3] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In R. Gl§ck and M. Lowry, editors, *Generative Programming and Component Engineering*, volume 3676 of *Lecture Notes in Computer Science*, pages 422–437. Springer Berlin / Heidelberg, 2005. 10.1007/11561347:28.

[4] K. Czarnecki, T. Bednasch, P. Unger, and U. W. Eisenecker. Generative programming for embedded software: An industrial experience report. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, GPCE '02, pages 156–172, London, UK, 2002. Springer-Verlag.

[5] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged configuration using feature models. In R. L. Nord, editor, *SPLC*, volume 3154 of *Lecture Notes in Computer Science*, pages 266–283. Springer, 2004.

[6] D. Fey, R. Fajta, and A. Boros. Feature modeling: A meta-model to enhance usability and usefulness. In *Proceedings of the Second International Conference on Software Product Lines*, SPLC 2, pages 198–216, London, UK, UK, 2002. Springer-Verlag.

[7] F. Fleurey, B. Baudry, R. France, and S. Ghosh. A generic approach for automatic model composition. In H. Giese, editor, *Models in Software Engineering*, volume 5002 of *Lecture Notes in Computer Science*, pages 7–15. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-69073-3_2.

[8] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[9] M. L. Griss, J. Favaro, and M. d. Alessandro. Integrating feature modeling with the rseb. In *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98, pages 76–, Washington, DC, USA, 1998. IEEE Computer Society.

[10] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[11] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.*, 5:143–168, January 1998.

[12] K. C. Kang, J. Lee, and P. Donohoe. Feature-oriented product line engineering. *IEEE Software*, 19(4):58–65, 2002.

[13] C. Kästner, T. Thüm, G. Saake, J. Feigenspan, T. Leich, F. Wielgorz, and S. Apel. Featureide: A tool framework for feature-oriented software development. In *ICSE*, pages 611–614. IEEE, 2009.

[14] B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., 1992.

[15] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving Executability into Object-Oriented Meta-Languages. In *MoDELS*, volume 3713 of *LNCS*, pages 264–278. Springer, 2005.

[16] G. Perrouin, J. Klein, N. Guelfi, and J.-M. Jézéquel. Reconciling automation and flexibility in product derivation. In *SPLC*, pages 339–348. IEEE Computer Society, 2008.

[17] F. Roos-Frantz and S. Segura. Automated analysis of orthogonal variability models. a first step. In S. Thiel and K. Pohl, editors, *SPLC (2)*, pages 243–248. Lero Int. Science Centre, University of Limerick, Ireland, 2008.

[18] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In J. Bosch and J. Lee, editors, *Software Product Lines: Going Beyond*, volume 6287 of *Lecture Notes in Computer Science*, pages 77–91. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-15579-6.

[19] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Comput. Netw.*, 51:456–479, February 2007.

[5] `http://www.thalesgroup.com/`

Using CVL to Operationalize Product Line Development with Reusable Aspect Models

Benoit Combemale and Olivier Barais
University of Rennes 1, IRISA
Rennes, France
{benoit.combemale, barais}@irisa.fr

Omar Alam and Jörg Kienzle
McGill University
Montreal, QC, H3A 2A7, Canada
Omar.Alam@mail.mcgill.ca
Joerg.Kienzle@mcgill.ca

ABSTRACT

This paper proposes a software design modelling approach that uses the Common Variability Language (CVL) to specify and resolve the variability of a software design, and the aspect-oriented modelling technique Reusable Aspect Models (RAM) to specify and then compose the detailed structural and behavioural design models corresponding to the chosen variants. This makes it possible to 1) exploit the advanced modularization capabilities of RAM to specify a complex, detailed design concern and its variants by means of a set of interdependent aspect models; 2) use CVL to provide an easy-to-use product-line interface for the design concern; 3) automatically generate a detailed design model for a chosen variant using a custom generic CVL derivation operator and the RAM weaver.

1. INTRODUCTION

A well-established and convenient practice in variability management is to provide a specification of the variability in terms of features separately from the associated artifacts that provide an implementations of the actual reusable assets. While some de-facto standards such as feature diagrams [9] are widely used to represent commonality (i.e., common properties) and variability (i.e., differences) of a system in terms of features, they still have to rely on specific operators to modularly implement and then compose the reusable aspects.

In this paper, we propose an approach combining the *Common Variability Language* (CVL) [4] to specify and resolve the variability of a software design, and the aspect-oriented modelling technique *Reusable Aspect Models* (RAM) [10] to implement and compose reusable object-oriented software design aspects. We use RAM to describe and compose the assets, while the feature model and its resolution (which are currently not explicit in RAM) are made explicit using CVL.

The contribution of this paper is therefore twofold: On the one hand we show how CVL can be used to extend an existing approach for AOM with well-established practices coming from the variability management community. On the other hand we illustrate the use of CVL with specific modularity and composition operators tailored to work with an aspect-oriented modelling technique. The derivation operator of CVL is specialized to work with RAM, resulting in an implementation of a generic opaque variation point that can produce the composition directives allowing the RAM weaver to produce a woven model corresponding to the chosen configuration.

The remainder of the paper is structured as follows: section 2 presents an overview of the proposed approach; section 3 illustrates the details of the approach by means of a software design concern product line for workflow executions; section 4 presents related work and the last section draws some conclusions.

2. APPROACH OVERVIEW: COMBINING CVL AND RAM

The approach we propose is based on **RAM** and **CVL**.

Reusable Aspect Models (RAM) [10] is a modelling approach that allows a designer to specify models that define the structure and behaviour of recurring design solutions. RAM models are inherently reusable, which means that it is possible to customize the generic design solution models to application-specific needs when applying them in a software design of a specific application. Currently, the RAM tool comes with a growing library of reusable design models, including models for low-level utility concerns, design patterns, network communication, workflow definition and execution, and transactions.

In general, there is *no one single good way* to solve a specific design problem. This is why RAM supports the definition of families of interrelated design models – in RAM terminology called *concerns* – that describe different variations of how to address a design problem. Typically, at the core of such a design concern is at least one aspect model that encapsulates the structure and behaviour common to all variations. Additional structure and behavioural properties covering variations of the design are modelled within *extensions* to that core model.

The **Common Variability Language** (CVL)[1] [4] is a domain-independent language for specifying and resolving variability over any instance of any MOF-compliant metamodel. Inspired by feature model, CVL contains several layer. The Variability Abstraction Model (*VAM*) is in charge of expressing the variability in terms of a tree-based structure. The core concepts of the VAM are the variability

[1]CVL is currently a proposal submitted to OMG. Cf. `http://variabilitymodeling.org`.

specifications (*VSpecs*). The *VSpecs* are nodes of the VAM and can be divided into three kinds: Choices, Variables and Classifiers. The Choices are *VSpecs* that can be resolved to yes or no (through *ChoiceResolution*), *Variables* are *VSpecs* that requires a value for being resolved(VariableValue) and *Classifiers* are *VSpecs* that imply the creation of instances and then providing per-instance resolutions (*VInstances*). In this paper, we mainly use the *Choices VSpecs*, which can be intuitively compared to features, which can or cannot be selected during the product derivation (yes/no decision). Besides the VAM, CVL also contains a Variability Realization Model (VRM). This model provide a binding between the base model and the VAM. It makes possible to specify the changes in the base model implied by the *VSpec* resolutions. These changes are expressed as Variation Points in the VRM. The Variation Points capture the derivation semantics, i.e. the actions to perform during the Derivation. Finally, CVL model contains resolution models to fix the variability capture in the VAM.

In this paper, we propose an approach combining CVL to specify and resolve the variability, and RAM to implement and compose reusable object-oriented software aspects. We use RAM to describe and compose the assets while the feature model and its resolution (which are currently not explicit in RAM) are made explicit using CVL.

The global approach is a two-level process: first the reusable aspects are capitalized and their possible combinations are captured in a variability model. Second, the variability model is used to select an expected set of features (aka configuration) from which a woven model is produced by composition of the suitable reusable aspects.

In practice, as illustrated in Figure 1, the approach is divided into the following five steps:

① implementation of the reusable aspects using RAM;

② specification of the variability (called *variability abstract model*) using the choice diagram proposed by CVL;

③ resolution of the variability by selecting a set of features (called *resolution model*) using CVL;

④ derivation of the composition directives using the generic CVL derivation operator, with a dedicated opaque variation point that we propose (and include in the *variability realization model*);

⑤ composition of the corresponding reusable aspects as describe in the composition directives with the RAM weaver.

From a methodological perspective, we also distinguish two roles for users of our approach:

- **Design Concern Expert**. The design concern expert knows the domain captured in the reusable aspects and knows their possible combination. This person is thus in charge of leveraging this domain to model one or several reusable aspects with RAM (step ① in Figure 1), and the respective variability with CVL (step ② in Figure 1).

- **Application Engineer**. The application engineer create models in the application domain. These users, through their modelling activities, can select the expected features with CVL (step ③ in Figure 1), and then automatically derive the composition directives (step ④ in Figure 1) used by RAM to automatically generate the corresponding woven model (step ⑤ in Figure 1).

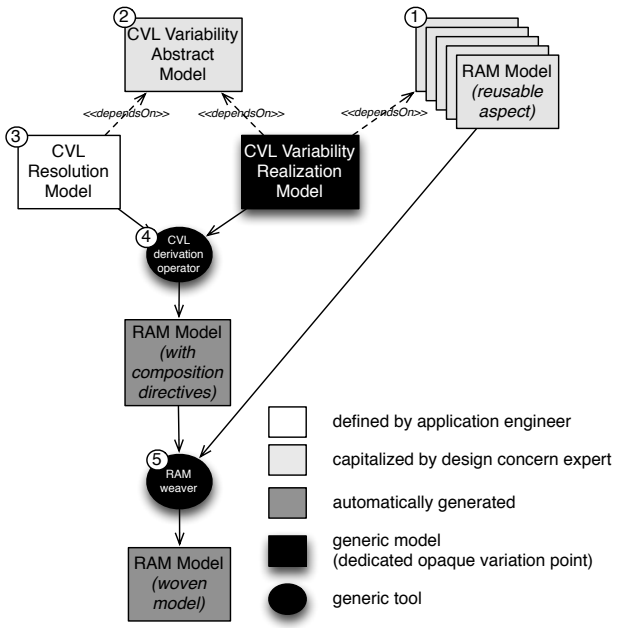


Figure 1: Approach Overview Combining CVL and RAM

3. APPROACH DETAILS: THE WORKFLOW CASE STUDY

A workflow is a set of operations that need to be completed in a certain order to fulfill a goal or task. For example, workflows have been used in software engineering to describe how a system under development is to interact with its environment.

At run-time, a system that is supposed to behave according to a workflow specification needs to incorporate a workflow execution engine in its design. To facilitate this, we have designed a reusable workflow design concern in RAM that provides such a functionality. Since workflows can be of varying sophistication, we designed a product line of workflow execution engines, which allows the designer to choose the most appropriate configuration for his specific application.

In this section of the paper we illustrate our process in detail by means of the workflow design concern case study. The following subsections correspond to the steps outlined in section 2.

3.1 Designing Reusable Assets using RAM

This subsection outlines the detailed design of the aspect models that are part of a RAM design concern. For space reasons, all presented models are simplified versions of the real RAM workflow design concern models: only the classes relevant to the definition of the different kind of nodes found in a workflow are shown. Structure related to the execution of the workflow, as well as all the sequence diagrams describing the behaviour of the design have been omitted. The in-

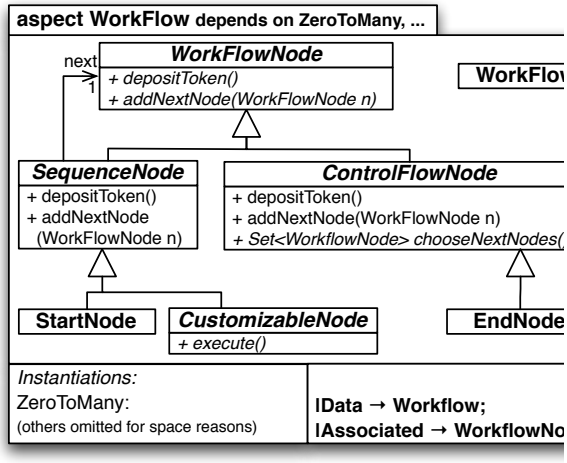


Figure 2: The Base Workflow Aspect Mode

terested reader can download the complete models fro website [2].

3.1.1 The Core Workflow Aspect Model

Figure 2 shows the *WorkFlow* aspect which define minimal model elements found in every workflow. It that a basic workflow is composed of nodes, which c sequence or control flow nodes. A special sequence n the `StartNode`, a special control flow node is the `End`. The `WorkFlowNode` has two abstract methods, `depositToken()` and `addNextNode(WorkFlowNode n)`, which are implemented differently by the two subclasses. This allows other parts of the system to treat workflow nodes in a uniform way. For example, the work flow execution engine (not shown for space reasons) can deposit a token into any kind of node in order to execute it, whether it is a `SequenceNode` or a `ControlFlowNode`. The design that encodes that a `Workflow` is composed of zero or more `WorkflowNode`s is implemented by another reusable aspect model, *ZeroToMany*. The instantiation directive on the bottom of Figure 2 instantiations that tell the RAM weaver how to compose *ZeroToMany* with *Workflow*.

With this base workflow aspect, a user can build very simple, sequential workflows. Application-specific actions are to be designed by extending `|CustomizableNode`, a class that executes the `execute` method when a token is deposited before scheduling the next node.

3.1.2 Workflow Extensions

The RAM workflow concern provides additional aspect models that define more elaborate control flows. For instance, there are control flow nodes that have multiple successor nodes, where each outgoing path is named using a string. Figure 3 shows an aspect called *OutPath* that extends the *WorkFlow* aspect and defines the structure needed for control flow nodes with more than one named outgoing path in the class `|CFNWithOutPath`. A new kind of named sequence node is introduced, `OutpathNode`, and a new kind of control flow node, `|CFNWithOutpath`. Internally, the *Map* aspect is reused to define a hash table that maps strings to `OutpathNode` as shown by the instantiation directives at the bottom of the figure.

[2] `http://www.irisa.fr/triskell/perso_pro/obarais/pmwiki.php?n=App.VARY2012`

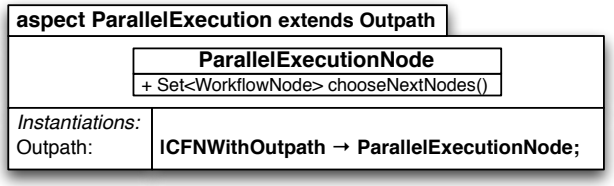


Figure 4: The ParallelExecution Aspect Model

The *ParallelExecution* aspect shown in Figure 4 is an example of an aspect that uses the *Outpath* aspect to define a control flow node that allows a workflow to continue execution of several following nodes in parallel. To reuse *Outpath*, `ParallelExecutionNode` is composed with `|CFNWithOutPath`.

The RAM workflow design concern defines many other workflow extensions, which can unfortunately not be shown here for space reasons. They are:

- *ConditionalExecution*, which allows for selective execution of workflows;
- *Synchronization*, which allows concurrent workflows to wait for each other;
- *Conditional Synchronization*, which allows concurrent workflows to wait for each other conditionally;
- *Timed Synchronizaton* which allows concurrent workflows to wait for each other until a timer expires;
- *Input*, which allows workflows to wait for input messages coming from the network;
- *Output*, which allows workflows to send output messages to the network;
- *Nesting*, which makes hierarchies of workflows possible.

Each extension is designed in one aspect model that extends the base workflow model, and optionally depends on other models to provide lower-level functionality. The left hand side of Figure 5 shows an overview of all the aspect models of the workflow design concern and their dependencies (shown using black straight arrows). The aforementioned extensions are above the *Workflow* aspect, since they add structure and behaviour to the latter. At the bottom of the figure, below the *Workflow* aspect, are all the low-level design models implementing design patterns (e.g. *Singleton*), recurring data structures (e.g. *ZeroToMany*, *Stack* or *Map*), or utility aspects (e.g. *NetworkCommand*).

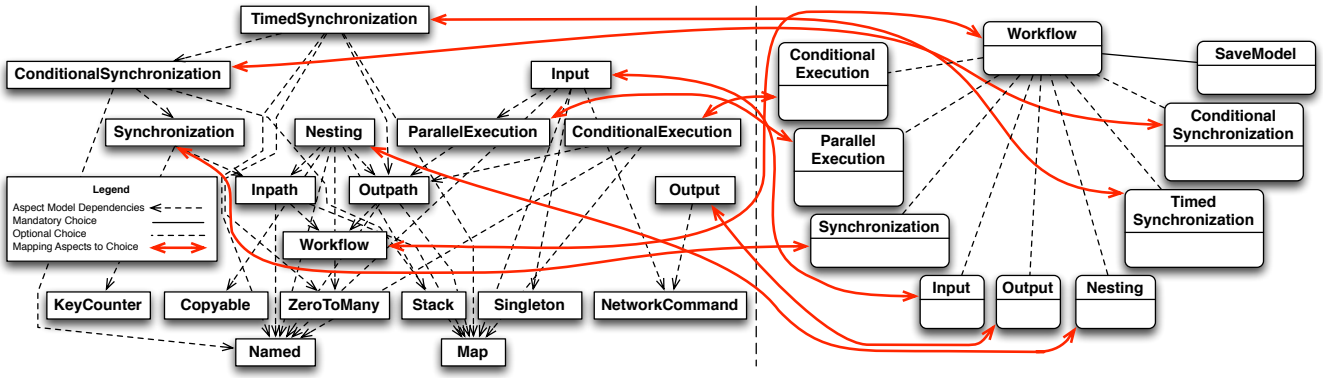


Figure 5: The Workflow Design Concern (RAM Models and Dependencies) and the CVL Variation Model

3.2 Specifying the Variability using CVL

To use a RAM design concern within an application model, the designer first needs to perform careful tradeoff analysis that takes into account the functional and non-functional requirements of the application under development to determine the desired concern variation. Then the designer needs to add instantiation directives into the application model that instantiate the RAM aspect models that correspond to the desired design concern variation. As a result, the RAM weaver can then compose the design concern models with the application model to yield the complete design model.

Unfortunately, this reuse process is quite cumbersome for the designer. In the current version of RAM, a design concern family does not have a well-defined user interface. The designer is confronted with a collection of interdependent aspect models as shown on the left hand side of Figure 5. Once the designer has determined which features of the design concern are relevant to her, she must determine which RAM models contain the design of these features, and then she must manually instantiate them.

To ease the task of the application developer, we propose to use the CVL Variation Model (VAM) to present a simple-to-use, feature-oriented view of a RAM design concern to the user. It encodes the set of choices and the constraints between choices. For the workflow example, we obtain the choice model as depicted on the right hand side of Figure 5. This choice model is quite simple, it contains a root choice $Workflow$ with eight optional sub-choices and one mandatory choice $saveModel$.

The mapping between the choice model and the RAM aspects, illustrated using red arrows in Figure 5, is designed in the CVL Variability Realization Model (VRM). The VRM contains ten *Opaque Variation Points* (OVP). An OVP is a black box variation point whose behaviour is defined with an action language expression specified in the CVL model. In our CVL implementation, we currently support OVPs defined in Groovy[3], in Javascript or in Kermeta [8]. With these action languages, the designer can modify the base model directly. Each variation point has access to a context that contains the list of objects to remove ($toRemove$), the list of *objectHandles* associated with this variation point (ctx), the list of variables and their associated value defined in the the resolution model ($args$), and a map of key/value pairs that variation points can use to pass data to subsequent variation points (map).

[3]http://groovy.codehaus.org/

Among the ten OVPs, there are only three different types. The first one is bound to the root choice and contains the action language expression to create a RAM aspect that defines the composition directives. The second one is bound to the $SaveModel$ choice and contains the action language expression to save the final model. The eight remaining OVPs are bound to RAM aspects and contains the code to create the composition directive in the root aspect previously created. The action language expressions for these three types of OVPs are shown in Listing 1.

Listing 1: Worflow OVPs in Groovy

```
1  //Expression for CreateAspectWorkflow OVP
2  Aspect aspectcreate=ca.mcgill.cs.sel.ram.
       RamFactory.eINSTANCE.createAspect();
3  aspectcreate.setName(args.get("name"));
4  maps.put("__NewAspect", aspectcreate);
5  ------------------------------------------
6  //Expression for the eight OVP associated to
       RAM apsects
7  Aspect newaspect = maps.get("__NewAspect");
8  Instantiation inst = ca.mcgill.cs.sel.ram.
       RamFactory.eINSTANCE.createInstantiation();
9  inst.setType(ca.mcgill.cs.sel.ram.
       InstantiationType.EXTENDS);
10 inst.setExternalAspect(ctx.get(0));
11 newaspect.getInstantiations().add(inst);
12 ------------------------------------------
13 //Expression for SaveModel OVP
14 Aspect aspecttoSave = maps.get("__NewAspect");
15 ResourceSet resourceSet = new ResourceSetImpl()
       ;
16 resourceSet.getResourceFactoryRegistry().
       getExtensionToFactoryMap().put(
17   "ram", new XMIResourceFactoryImpl());
18 URI fileURI = URI.createFileURI(new java.io.
       File(args.get("modelname").getAbsolutePath
       ());
19 Resource resource = resourceSet.createResource(
       fileURI);
20 resource.getContents().add(aspecttoSave);
21 resource.save(java.util.Collections.EMPTY_MAP);
```

3.3 Specifying the Resolution using CVL

With the CVL VAM model, the application engineer can do the selection according to the variability captured in the VAM model. For this point, we automatically generate an initial solution for the resolution model according to the choice model constraints (cardinalities, $isImpliedByParent$, $DefaultResolution$, ...). The application engineer can change this choice resolution decision using a graphical tool as shown in Figure 6. In this example, we take the decision to select

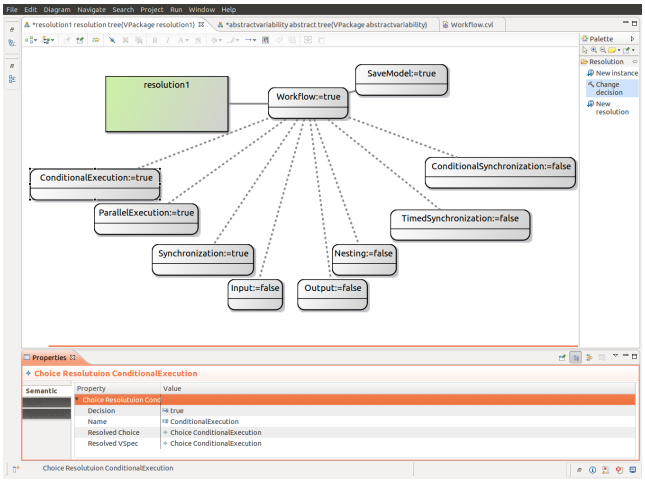
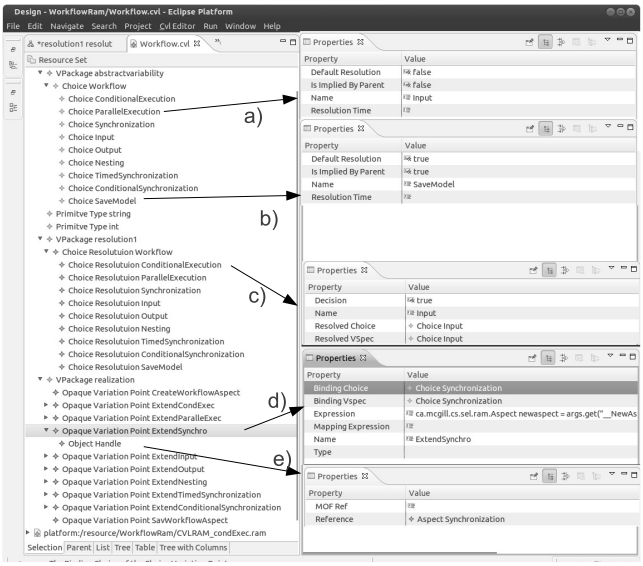


Figure 6: The Workflow Resolution model



the *ParallelExecution*, *ConditionalExecution* and *Synchronization* choices. The variable values are also set during resolution. In our case, the application engineer must defined the name of the composition directive aspect and the URI of its resource [4].

3.4 Derivation of a Software Design using CVL

The tool screenshot shown in Figure 7 illustrates the entire derivation process. Arrows *a*) and *b*) depict the VAM model. Arrow *c*) shows a choice resolution. Arrow *d*) shows an OVP which specifies the mapping to a specific RAM model (object handle shown in arrow *e*)). Based on this model, the derivation engine executes the code of the OVPs selected by the choice resolution. The expression for the OVP bound to the RAM aspect is generic and can be used as such for new aspects than can be composed. Only the object handle that maps to the concrete aspect must be respecified. The CVL derivation engine and the workflow models can be downloaded on our website [5].

3.5 Composing the Reusable Assets using RAM

Using the generated aspect model as input, the RAM weaver recursively composes all dependent aspect models to generate a complete design concern model that corresponds to the selected configuration. In our case we selected the *ParallelExecution*, *ConditionalExecution* and *Synchronization* variants. As a result, the RAM weaver will compose *ParallelExecution*, *ConditionalExecution*, *Synchronization*, as well as the dependent aspects *Outpath*, *Inpath*, *KeyCounter*, *Named*, *ZeroToMany*, and *Map*.

The resulting structural model is shown in Figure 8[6]. Since *Outpath* is instantiated both by *ParallelExecution* and *ConditionalExecution*, there are also two instances of *Map* that map `Strings` to `WorkflowNodes`. *Map* is used a third time in *Synchronization* to map `InpathNodes` to `Integers`.

[4]An EMF resources is a container of persisted model elements

[5]`http://www.irisa.fr/triskell/perso_pro/obarais/pmwiki.php?n=App.VARY2012`

[6]Again, only the classes directly related to the different kind of workflow nodes are shown. The structure pertaining to workflow execution (executors, execution contexts and parameters) have been omitted for space reasons.

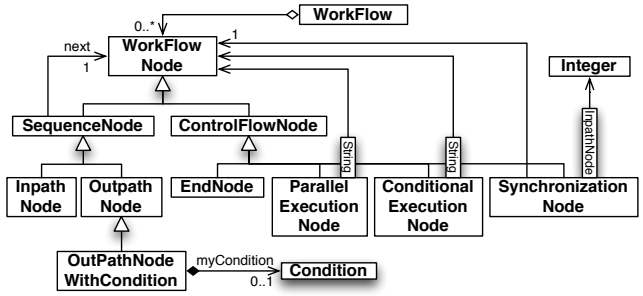


Figure 8: A Woven Workflow Design Concern Model

3.6 Discussion on the Use of OVPs

We could probably design the VRM to use other kinds of variation points such as *ObjectSubstitution* or *FragmentSubstitution*. Nevertheless, in combining AOM and CVL, the composition complexity is primarily in the aspect weaver. Consequently, the derivation engine remains simple. Consistency checking would, for instance, be part of the aspect weaver, undisturbed by the presence of the OVP. Additionally, we were able to build a generic OVP to create the aspect composition directives. Checking this OVP only needs to be done once. The CVL model could be further improved by using *ConfigurableUnit* and *ConfigurableUnitUsage* to avoid the eight repetitions of the same OVP. This will be investigated in future work.

4. RELATED WORK

This section discusses related work on aspect-oriented modelling and variability modelling approaches.

Because composing models by hand is a cumbersome task, tools and approaches have been proposed that automate significant parts of model composition [7]. The early aspect [15] and aspect-oriented modelling [1] workshop series introduced many aspect-oriented modelling (AOM) approaches that differ on (i) model composition activities, (ii) the types of inputs required other than the 2 models to be composed and (iii) the types of models that can be com-

posed. We chose RAM in this work because of its ability to compose structural and behavioural models and because the existence of large aspect-oriented models that capture variabilities. Indeed, RAM has been applied to model many software designs concerns. The biggest design concern is the AspectOptima case study, a transaction support middleware product line [10, 11]. AspectOptima offers support for multiple transaction models (flat, nested, multithreaded and open multithreaded transactions), different concurrency control strategies (pessimistic lock-based and optimistic time stamp-based), and different update strategies (inlace and deferred update). However, even if we used RAM in this paper, the proposed approach should also work for other AOM approaches.

Many formalisms were proposed in the past decade for variability modeling. For an exhaustive overview, we refer the readers to the literature reviews that gathered variability modeling approaches [16, 6, 2, 17, 3]. All formalisms for variability modeling could be used following the approach we introduce in this paper. In our case, we use the choice diagram proposed by CVL, very similar to an attributed feature diagram with cardinalities.

More recently, few works deal with the binding between the feature in the variability modeling and the actual assets. Let us cite for example *FeatureMapper* [5], which is a tool for combining SPL and MDE that makes it possible to bind a feature to a design model. Relying on CVL, we use in our approach the provided action language to describe in the realization model the binding between the features in the choice model and the actual RAM assets. A dedicated derivation operator is automatically obtained by implementing a dedicated opaque variation point in the CVL generic derivation operator.

Recently, several works have shown the benefits of coupling aspect-oriented modelling approaches and variability approaches. Voelter *et al.* [18] was the first to combine AOM and MDE techniques to achieve an explicit separation of concerns in software product lines. In the domain of software architecture, we cite Morin *et al.* [12] and Parra et al. [13] that combine architecture aspect models and feature models to ease the design of adaptive systems. In [14], Perrouin *et al.* proposes to specify variants by means of model fragments and the product derivation process consists in merging those fragments together.

5. CONCLUSION

In the *Reusable Aspect Models* approach, a design concern is a collection of interrelated aspect models describing a family of design solutions for a specific design problem. This paper showed how we used CVL to specify an easy-to-use product line interface for RAM design concerns. When faced with a specific design problem for which a RAM design concern exists, an application developer can consult the variation model provided by CVL to get an overview of all possible design choices. After making her choice, she passes the resulting resolution model to the CVL derivation engine, which knows about how to map features to RAM aspect models. Based on this knowledge, the derivation engine automatically creates an aspect that instantiates all the aspect models that are needed for the designer. Using this aspect, the RAM weaver generates a complete model of the chosen design concern configuration.

Acknowledgements

This work has been partially supported by VaryMDE, a collaboration between Inria and Thales Research & Technology.

6. REFERENCES

[1] Aspect-Oriented Modeling Workshop Series. http://dawis2.icb.uni-due.de/aom/workshop:start.

[2] D. Benavides, S. Segura, and A. Ruiz-cort. Automated Analysis of Feature Models 20 Years Later : A Literature Review 6. *Review Literature And Arts Of The Americas*, 2010.

[3] L. Chen and M. Ali Babar. A systematic review of evaluation of variability management approaches in software product lines. *Information and Software Technology*, 53(4):344–362, Apr. 2011.

[4] F. Fleurey, Ø. Haugen, B. Møller-Pedersen, A. Svendsen, and X. Zhang. Standardizing variability - challenges and solutions. In I. Ober and I. Ober, editors, *SDL Forum*, volume 7083 of *LNCS*, pages 233–246. Springer, 2011.

[5] F. Heidenreich, J. Kopcsek, and C. Wende. FeatureMapper: Mapping Features to Models. In *Companion Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 943–944. ACM, May 2008.

[6] A. Hubaux, A. Classen, M. Mendonça, and P. Heymans. A preliminary review on the application of feature diagrams in practice. In *VaMoS*, volume 37 of *ICB-Research Report*, pages 53–59. Universität Duisburg-Essen, 2010.

[7] C. Jeanneret. An analysis of model composition approaches. Master's thesis, Colorado State University and EPFL, 2008. http://goo.gl/iYcGy.

[8] J.-M. Jézéquel, O. Barais, and F. Fleurey. *Model Driven Language Engineering with Kermeta*. LNCS 6491, Springer, 2010.

[9] K. Kang, S. C. J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (foda). feasibility study. Technical Report CMU/SE-90-TR-21, Software Engineering Institute, Pittsburgh, PA 15213, Nov. 1990.

[10] J. Kienzle, W. Al Abed, and J. Klein. Aspect-Oriented Multi-View Modeling. In *AOSD 2009, March 1 - 6, 2009*, pages 87 – 98. ACM, March 2009.

[11] J. Kienzle, E. Duala-Ekoko, and S. Gélineau. AspectOPTIMA: A Case Study on Aspect Dependencies and Interactions. *Transactions on Aspect-Oriented Software Development*, 5:187 – 234, Mar. 2009.

[12] B. Morin, O. Barais, G. Nain, and J.-M. Jézéquel. Taming dynamically adaptive systems using models and aspects. In *ICSE*, pages 122–132. IEEE, 2009.

[13] C. A. Parra, X. Blanc, A. Cleve, and L. Duchien. Unifying design and runtime software adaptation using aspect models. *Sci. Comput. Program.*, 76(12):1247–1260, 2011.

[14] G. Perrouin, J. Klein, N. Guelfi, and J.-M. Jézéquel. Reconciling Automation and Flexibility in Product Derivation. In *12th International Software Product Line Conference (SPLC 2008)*, pages 339–348, Limerick, Ireland, Irlande, 2008. IEEE Computer Society.

[15] M. Pinto, R. Chitchyan, A. Rashid, A. Moreira, J. Araújo, P. C. Clements, E. L. A. Baniassad, and B. Tekinerdogan. Early aspects at icse 2008: workshop on aspect-oriented requirements engineering and architecture design. In *ICSE Companion*, pages 1053–1054. ACM, 2008.

[16] K. Pohl and A. Metzger. Variability management in software product line engineering. In *Proceedings of the 28th international conference on Software engineering (ICSE '06)*, pages 1049–1050. ACM, 2006.

[17] R. Rabiser, P. Grünbacher, and D. Dhungana. Requirements for product derivation support: Results from a systematic literature review and an expert survey. *Information and Software Technology*, 52(3):324–346, 2010.

[18] M. Voelter and I. Groher. Product line implementation using aspect-oriented and model-driven software development. In *11th International Software Product Line Conference (SPLC'07)*, pages 233–242. IEEE, 2007.

Usage Scenarios for Feature Model Synthesis

Steven She
Generative Software Development Lab
University of Waterloo
shshe@gsd.uwaterloo.ca

Krzysztof Czarnecki
Generative Software Development Lab
University of Waterloo
kczarnec@gsd.uwaterloo.ca

Andrzej Wąsowski
IT University of Copenhagen
wasowski@itu.dk

ABSTRACT

Feature models are menu-like hierarchies of features (i.e., configuration options) used in variability-rich software. Feature models have many applications such as domain analysis, describing design and implementation constraints in software, or for product configuration. The many applications of feature models have given rise to a wide range of scenarios involving feature model synthesis.

Feature model synthesis is the process of building a feature model for a given set of features and their allowed combinations, expressed as feature dependencies or feature configurations. We describe and classify software re-engineering scenarios involving feature model synthesis found in literature and industry. We analyze these scenarios to derive requirements for feature model synthesis techniques.

1. INTRODUCTION

Variability in software is the ability for the software to be adapted and customized for a particular context [26]. Unfortunately, variability is not restricted to just one part of a software system and is often scattered over multiple artifacts. Variability could be abstract and be represented in a single, variable artifact like a feature model. Alternatively, variability could be realized as a configurable platform—a variable artifact, or as a set of variants—where variability in each individual artifact is resolved. Software product lines (SPLs) are a form of variability-rich software that systematically handle variability and enable code reuse across a family of related products with common and variable product characteristics [8]. Other examples of variability-rich software are ecosystem platforms, such as the Linux kernel or the Eclipse IDE, which support extensions and products rather than a tightly-managed product portfolio.

Table 1 shows a classification of software artifacts with variability in feature-oriented, variability-rich software. Different software systems use different combinations of these artifacts. For example, feature-oriented SPLs have a feature model, variation points (VPs), a mapping from features to VPs, and a configurable platform describing the implementation. *Variation points (VPs)* are locations in solution space artifacts where variations occur. While VPs exist in the artifacts of a platform, they form an interface between the abstraction and realization of variability in the platform. SPLs are developed with variable artifacts and instances are derived via feature configurations based on a feature model.

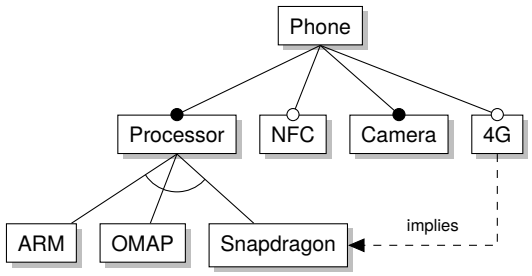


Figure 1: Feature model of a mobile phone product line

{ Phone, Processor, ARM, Camera },
{ Phone, Processor, ARM, Camera, NFC },
{ Phone, Processor, OMAP, Camera },
{ Phone, Processor, Snapdragon, Camera, 4G },
...

Figure 2: Legal configurations of the mobile phone FM

Feature models (FMs), first introduced by Kang et al. [17], describes *features*—the common or variable characteristics of the products in a SPL—as a visual hierarchy with additional constraints between features. Since FMs were first introduced, they have been used in a wide variety of tasks such as domain analysis [17], model management [1], describing design or implementation constraints in variability-rich software [9], and product configuration.

Figure 1 shows a FM of a mobile phone product line. Features are represented as rectangles and may be *optional*—

Table 1: Breakdown of artifacts in feature-oriented software

	Variability Abstraction	Abstraction-Realization Interface	Variability Realization
Variable Artifacts	Feature model	VPs and feature-to-VP mapping	Configurable platform *requirements, models, code, etc.*
Instances	Feature configs.	VP configs.	Variants *requirements, models, code, etc.*

```
options  SCHED_ULE  # ULE scheduler
options  PREEMPTION # Enable kernel thread preemption
options  INET       # InterNETworking
options  INET6      # IPv6 communications protocols
...
```

Figure 3: Snippet of a FreeBSD feature configuration

denoted by an empty circle at the top—or *mandatory*—denoted by a filled circle. An edge between features denotes a dependency, where a solid line is used for the feature tree and a dashed line is a *cross-tree implies edge*. *Cross-tree excludes edges* can also exist in the diagram, but are not shown. Features may also be part of a *feature group*. An XOR-group, specified with a clear arc, denotes that exactly one member of the group must be selected if its parent is selected. OR-groups, where one or more members must be selected, and MUTEX-groups, where zero or one members must be selected, can also exist but are not shown in the figure. The feature tree, feature groups, and cross-tree edges form the *feature diagram*. A FM consists of the diagram and a *cross-tree formula* to describe additional constraints. We use the term *cross-tree constraints (CTCs)* to refer to cross-tree implies and excludes edges, and the cross-tree formula.

The configuration semantics of a FM is a set of *legal configurations* defined by the satisfying assignments to its propositional formula [5]. Figure 2 shows some of the legal configurations of the FM in Figure 1.

Some variability-rich software systems, such as FreeBSD—an operating system kernel, do not have a FM and contain a mixture of variable artifacts and instances instead. In FreeBSD, feature configurations are used for different hardware architectures and devices. For example, Figure 3 shows a snippet from the generic i386 feature configuration contained in the FreeBSD source code. Each line specifies a feature to be included in the kernel.

FreeBSD is implemented as a configurable platform (Table 1) in C using #ifdef statements for VPs. A feature-to-VP mapping is realized as a build system. Configuration is driven by creating feature configurations based on templates and documentation. FreeBSD can benefit from having an explicit FM describing variability among features. Synthesizing a FM for FreeBSD is just an example of applying FM synthesis. FMs can also be synthesized from requirements for domain analysis, from code variants to drive configuration of existing and future products, or from a set of other FMs as part of a model management operation such as a model merge.

Feature model synthesis is the process of building a FM given the following *abstract input*: (1) a set of features, (2) allowed combinations represented as a set of feature dependencies or configurations, and optionally (3) supplemental information to help with tree and group recovery (e.g. tree hierarchy). Figure 4 shows the workflow for feature model synthesis.

Recovering the abstract input is done in a prior *analysis stage*. Extracted VPs in solution space artifacts can be interpreted as fine grained features that are close to implementation. In addition to features, analysis of the input artifact extracts allowed combinations as either feature dependencies or configurations. Synthesis can also be supplemented with information extracted in the analysis stage, such as a tree hierarchy, or feature descriptions.

Different synthesis scenarios assume different input artifacts (Table 1), each with different properties. In this paper, we characterize several re-engineering scenarios that involve

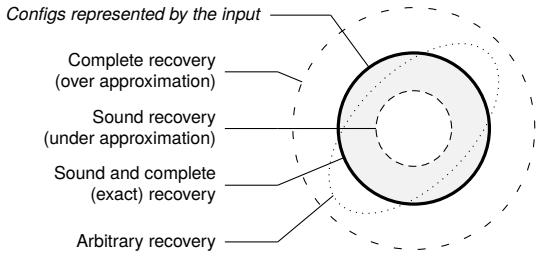


Figure 5: Property of a transformation step

feature model synthesis found in literature and industry. We compare the expected input artifacts, the analysis needed to obtaining the input for synthesis, and the expected synthesis result for each scenario. We derive requirements for feature model synthesis techniques from these scenarios and discuss existing synthesis techniques.

2. SCENARIOS

2.1 Feature Model Synthesis Workflow

The first stage in a scenario involving FM synthesis is *analysis*, where the abstract input is recovered from the input artifacts (Figure 4). Whether dependencies or configurations are recovered depends on the type of input artifacts (Table 1). Dependencies can be recovered from a variable artifact. Some projects provide feature configurations directly (e.g. FreeBSD), while other scenarios provide variants that would require a separate step for recovering a feature configuration describing each variant. For example, a requirements document or codebase can realize a single variant. The corresponding feature configuration is implicit in this variant and would require analysis to recover.

Feature model synthesis builds a FM using the abstract input. There are two components in this stage: *tree recovery* where the feature tree is recovered, and *group and cross-tree constraint (CTC) recovery* where feature groups and CTCs are identified once a tree is determined. We use this re-engineering workflow to describe the processes of the scenarios in Section 2.3.

2.2 Scenario Criteria

In this section, we introduce the criteria used to classify the re-engineering scenarios:

Input Artifacts. Table 1 lists the possible input artifacts to feature model synthesis. The input could be variable artifacts (i.e. artifacts with variability), such as a FM, or a requirements document or code with VPs. Other input type are feature configurations (i.e. sets of selected features) or variants (i.e. artifacts with resolved variability), such a requirements document or code for a particular product.

Precision of Configuration Analysis. The analysis step is responsible for recovering features and feature combinations needed for synthesis. We assume that the full set of features is recovered prior to FM synthesis. This criterion classifies the precision of recovering feature configurations (i.e., feature combinations) from the input artifacts.

Figure 5 shows the classification of both the analysis and synthesis stages with respect to its input feature configurations. A *sound and complete*, or exact recovery is one that is able to recover the exact set of configurations present in the input artifacts. A *complete* recovery is one that does not lose

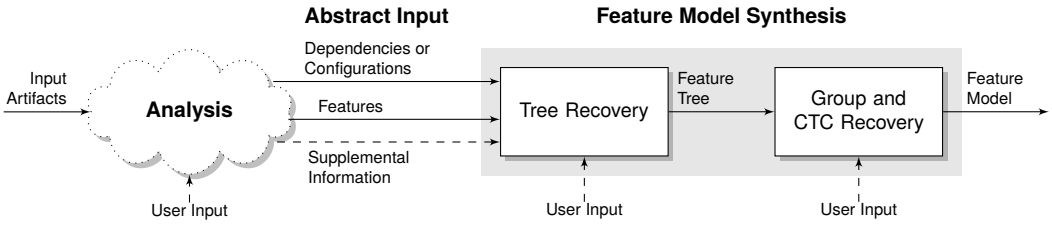


Figure 4: Feature model synthesis workflow

any configurations present in the input artifacts. Conversely, a *sound* recovery is one that does not add new configurations compared to what is the input.

Required Synthesis Precision. FM synthesis takes the abstract input recovered in the analysis stage and synthesizes a FM. The required synthesis precision is based on the precision of the abstract input and uses the same classification in Figure 5. In this case, an exact recovery is one that synthesizes a FM that describes exactly the configurations in the abstract input. Note the definition of soundness and completeness in this paper are based on the set of configurations. There is a duality between configurations and dependencies such that a *sound* technique in terms of configurations is *complete* in terms of dependencies and vice versa.

Size. We classify the size of each scenario by estimating the number of features it requires for FM synthesis. A *small* scenario has several hundred features. A *medium* size scenario has roughly a thousand features, and a *large* scenario has several thousand features. We base this categorization on existing models found in literature and in FM repositories. The 232 models in the SPLOT model repository have a median of 20 features and at most 290 features[1]. BigLever and pure-systems have reported that models in industry are typically in the range of hundreds of features. In addition to these models, we collected a set of 13 FMs from the systems domain with a median of 1600 features [6]. The smallest FM in this dataset was ToyBox with 71 features and the largest was the Linux x86 kernel model[2] with 6320 features. Finally, FreeBSD is a variability-rich system with 1203 features[2] that could benefit from having a managed FM [25]. Researchers have generated models with 5,000–10,000 features for testing tools on large-scale FMs [18]. As more FMs become available in the future, these numbers may change.

2.3 Scenario Descriptions

We have identified several scenarios from industry and literature based on the artifacts types in Table 1. Figure 6 shows the workflow for each scenario and Table 2 classifies each scenario according to the criteria previously described.

Scn. 1. Synthesizing from a Configurable Platform.

A configurable platform consists of variability-rich assets with VPs. The platform could contain different artifact types, such as requirements, models, or code. We separate the cases below based on the input artifacts:

Scenario 1a. The input to this scenario is a configurable platform of code with VPs. For example, the FreeBSD kernel with 1203 features, is such a platform where the implementation is given in C with VPs defined using `#ifdef` preprocessor statements. The first stage in this scenario is to identify VPs and dependencies in the code using static analysis (Figure 6). These VPs are fine-grained and are closely related to the solution space and need to undergo a further feature abstraction step. While static analysis is typically automatic, it is not guaranteed to find all dependencies between VPs. As a result, the recovery of valid feature combinations is complete but unsound (i.e., an over-approximation of the configurations allowed by the platform). A sound synthesis is needed to compensate for the over-approximated abstract input. In other words, additional dependencies may need to be introduced during the synthesis. This scenario motivates our previous work on reverse engineering FMs [25].

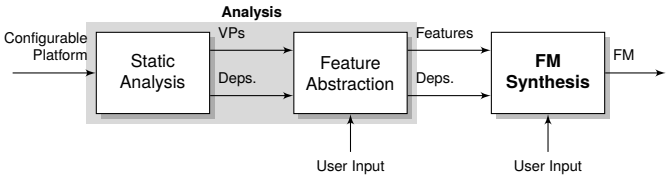
Scenario 1b. The input in this case is a platform consisting of a requirements document with variability (i.e. optional requirements) describing a product line. Niu et al. [19] and Weston et al. [27] both describe this scenario to motivate their synthesis techniques. In Figure 6, requirements are first identified using a combination of natural language analysis and the domain knowledge of an expert. These techniques treat a feature as a group of requirements and use clustering to build the tree hierarchy. For each cluster, an abstract feature is introduced with the clustered requirements as sub-features. This approach leads to the analysis and FM synthesis stages being intertwined. User input is needed in this stage to name abstract features or adjust the clustering algorithm. This scenario also matches the experience described to us by an industry partner in the automotive sector, but instead, the requirements clustering and FM building were performed manually. Niu extracted 22 features from two sample applications [19]. Weston used requirements documents describing a Smart Home with 87 requirements in total [27].

Scn. 2. Synthesizing from Variants.

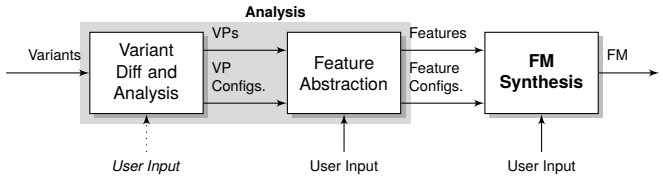
In this scenario, the input is a set of variants where variability in each artifact is resolved. Variants come in different artifact and we separate this scenario into the following cases:

Scenario 2a. The variants are a set of related models in this case. In Figure 6, the first step in this scenario is to compare the variants and extract a set of VPs and VP configs. Rubin and Chechik describe an approach for identifying similarities between model instances by comparing and matching model elements [20, 21]. Ryssel et al. also described an algorithm that uses model matching and difference to identify VPs and abstract the model variants into a set of VP configurations [23]. Since the input artifacts are in the solution space, VPs undergo a feature abstraction step. In Ryssel's scenario, the resulting FM must describe the exact configurations provided as input [22], thus requiring an exact synthesis. Ryssel's performed two case studies for their synthesis technique [23]. The first case study involved a set of related models that had 14 features over 17 variants [23].

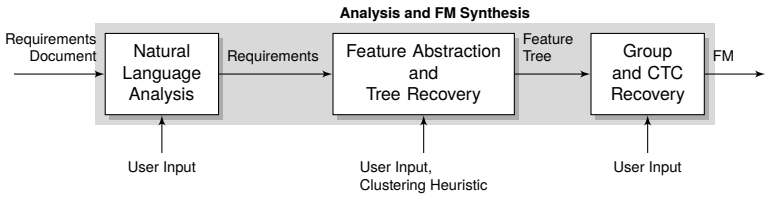
[1] `http://www.splot-research.org` as of August 5, 2012
[2] as of Linux v2.6.32 and FreeBSD v8.0.0



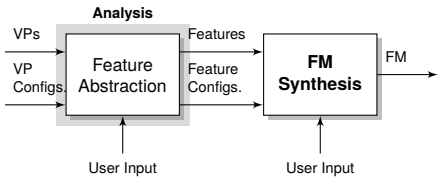
(Scn. 1a) Configurable platform (code [25])



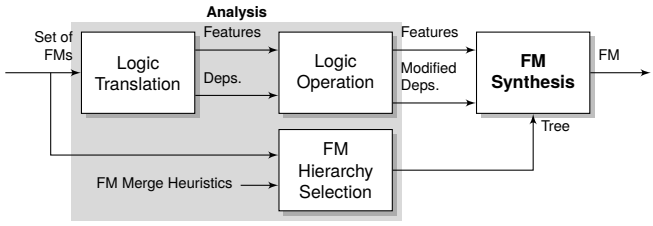
(Scn. 2a, 2b, 2c) Variants (requirements, models [22], and code [16])



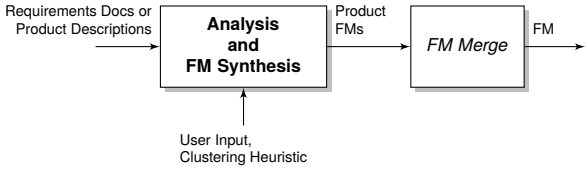
(Scn. 1b) Configurable platform (requirements) [27][19]



(Scn. 2d) Variants (VP configs)



(Scn. 3) Feature model operations [1]



(Scn. 4) Variants or configurable platform (product descriptions [2] and requirements [3, 7])

Figure 6: Scenario workflows

The second case study had 415 features across 49 variants.

Scenario 2b. This case was described by an industry partner and uses a set of requirements, each describing a specific variant. The workflow is identical to Scenario 2a. The requirements are compared and VPs are identified between each document. Each requirements document is converted to a VP configuration manually. The goal of the analysis step was to describe the precise set of configurations (i.e., products) making this scenario an exact analysis. However, the resulting FM is intended for domain analysis in this scenario, thus requiring a complete synthesis that allows more configurations than provided as input.

Scenario 2c. In this scenario at Danfoss drives described by Jepsen et al. [16], code variants developed using a clone-and-own approach are used as input. The developers first compared and merged the code variants into a single codebase by placing conditional compiler flags on code fragments based on the products that contain the fragment. Each conditional fragment is a VP. A VP configuration consists of all included code fragments for a product. These VPs were then abstracted to features and a FM is synthesized from the resulting set of features and configurations. Similar to Scenario 2b, the synthesis is intended to allow more configurations making it complete.

Scenario 2d. This case uses a set of VPs and VP configurations directly as input where each configuration represents a product in product line. This scenario was described to us by an industry partner in the automotive sector. The company wanted to build a FM that described their existing product line and supported instantiation of future products. The input configurations are exact, however, complete synthesis is required to support additional configurations.

Scn. 3. Feature Model Operations.

Acher describes model management operations on FMs that include merge, difference, or projection (slice) [1]. Similarly, Fahrenberg et al. discuss use cases for a semantic difference between models using FMs as an example [13].

The workflow in Figure 6 depicts the scenario realized by Acher [1]. This scenario begins with a set of FMs as input. The models are translated to propositional logic through their configuration semantics [5]. The FM operation is performed on the formula and used as input to an existing synthesis technique [11]. This modified formula describes the set of configurations from the input models, modulo the operations semantics, making the analysis step sound. Unlike the previous scenarios where user input is required for the synthesis stage, Acher describes heuristics for automatically determining the resulting FM hierarchy based on the input FMs [1]. Thus, the FM operations are automated. FM operations can be applied to FMs of any size. Acher applied their implementation on the models in SPLOT where the largest had 290 features, and also on generated models with up to 2000 features [1].

Scn. 4. FM Merge Workflows.

Our last set of scenarios synthesize an individual FM for each product, then apply a FM merge to create the final FM describing all products.

Scenario 4a. In this scenario described by Acher et al. [2], a configuration-like input in the form of a product descriptions is used. However, product descriptions can contain variability where a product could support one or more storage methods,

Table 2: Scenario classification

Scn.	Input Artifacts	Form*	Feature Combinations Analysis Precision	Required Synthesis Precision	Size
1a.	Platform (code)	D	Complete	Sound	Medium–Large
1b.	Platform (requirements)	D	Exact†	Arbitrary	Small
2a.	Variants (models)	C	Exact	Exact	Small–Medium
2b.	Variants (requirements)	C	Exact†	Complete	Small
2c.	Variants (code)	C	Exact	Complete	Small–Medium
2d.	Variants (VP configs)	C	Exact	Complete	Small
3.	Feature models	D	Sound	Exact	Medium–Large
4a.	Platform / Variants (descriptions)	C/D	Exact	Exact	Small
4b.	Platform / Variants (requirements)	C/D	Exact†	Arbitrary	Small

*Abstract Input Form. C is *configurations*, D is *dependencies*.
†Analysis performed manually.

or exactly one operating system, for example. Each product description is transformed into a product FM—a FM describing the product and its variability. The product FMs are then merged to create a FM describing variability in all products (e.g. Scenario 3). Acher used product descriptions ranging from 9 to 190 features as input [2].

Scenario 4b. In this case, a set of requirements documents, each describing a single product, is used as input. Alves et al. [3] applied clustering to build a FM describing each document, then merged these individual FMs to create a FM describing all documents. Chen et al. also applied clustering to synthesize an individual FM for each requirements document where each document describes a single variant [7]. The individual FMs are then merged in a subsequent step. Alves used two requirements documents with 59 and 23 requirements respectively [3]. Chen's scenario included a sample application with 23 requirements [7].

3. DISCUSSION

In this section, we derive requirements from the scenarios and discuss existing synthesis techniques.

Input Form. The input artifacts distinguish one scenario from another. In Scenarios 1 and 2, these artifacts abstract to either dependencies or a set of configurations. For dependencies, our previous work synthesized a model from dependencies by using binary decision diagrams (BDDs) [11] and SAT solvers [4]. Configuration-based approaches include the FCA-based approach by Ryssel et al. [22], and the DNF input for our approach [4]. Techniques for requirements operate on natural language text and combine both the analysis and synthesis stages by using clustering [3, 19, 27].

Scenario 3 is a FM operation where a set of FMs are first translated to their propositional formulas and an operation is applied. The resulting formula is a set of dependencies and can use any technique that supports dependencies, e.g. Acher uses our BDD-based approach [11] for FM synthesis [1].

In Scenario 4, each artifact describes a product with added variability. These abstract to individual product FMs and are combined with a FM merge technique [1, 2, 3, 7] to form the final FM describing all products.

Hierarchy. The first stage of synthesis is recovering the feature hierarchy. However, given a set of valid configurations, there is more than one possible hierarchy. Figure 7 shows two FMs that describe the same configurations but with different hierarchies. As a result, some synthesis techniques recover

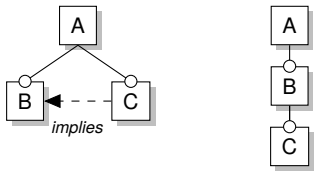


Figure 7: Two feature models, same configurations

a directed acyclic graph (DAG) that represents all possible feature hierarchies given the input [4, 10, 11, 12, 22].

Techniques have also been proposed to select a distinct tree from the set of possible hierarchies. Our previous work proposed a semi-automated technique using feature similarity to select a tree [25]. Janota et al. developed an interactive FM building tool that used dependencies and a recovered DAG as input [15]. His approach guaranteed that the final tree was complete with respect to the input configurations. Hierarchy selection can be made automatic by using heuristics to select the hierarchy. For FM operations (Scenario 3), Acher determines a tree automatically by using heuristics based on the input FMs [1]. Acher et al. combines both heuristics and a manual specification for deriving the hierarchy from product descriptions (Scenario 4) [2]. Clustering techniques build hierarchy by grouping related requirements under abstract features [3, 7, 19, 27].

Synthesis Precision. The precision of a synthesis technique is dependent on the precision of the analysis step and the expected use of the FM. For example, in Scenario 1a, the abstracted feature combinations is complete, or an over approximation. As a result, a sound synthesis technique is needed to add additional constraints to remove unwanted configurations. However, the abstracted feature combinations in Scenario 2b are exact, but require a complete synthesis to remove unwanted constraints and support additional configurations. Our previous work [4, 11] and Ryssel's approach [22] derives an exact FM describing the input. We later extended our approach with a feature similarity heuristic to deal with complete, but unsound input [25]. The interactive model building tool by Janota et al. support building FMs that are complete with respect to the input. Clustering techniques are not focused on maintaining a set of input feature combinations, but are geared towards exploratory activities such as domain analysis. As a result, we classified the required synthesis precision as arbitrary, since constraints can be added or removed during FM synthesis.

Scalability. The input form affects the scalability of a synthesis technique. Techniques that use dependencies as input are more scalable than techniques that use a set of configurations since dependencies represent a set of configurations symbolically. Other factors that affect a synthesis technique's scalability is the required amount of user interaction. Techniques requiring significant user interaction become intractable as the size of the models grow. However, user interaction can be replaced by using heuristics to automate the process and is largely dependent on the supplemental information extracted from the input artifacts (e.g. hierarchy data). Finally, the reasoning technique can affect a synthesis technique's scalability. We found that our SAT-based implementation significantly improved upon our BDD-based approach [4]. Ryssel et al. use a FCA-based approach [22]. An evaluation comparing the different reasoning techniques is planned for future work.

Probabilistic Feature Models. A special case involves synthesizing a probabilistic FM from configurations. A probabilistic model provides soft constraints that are valid in most configurations, but not necessarily all [10]. Our previous work recovered a probabilistic FM describing framework usage in a set of applications [24]. Dumitru et al. recover a probabilistic FM for a recommender system that suggests features for domain analysis [12]. Fukuda et al. build a probabilistic model to identify trends in a product transaction database [14]. We proposed a technique for building a probabilistic FM by using association rule mining on a dataset of configurations [10].

4. CONCLUSIONS

As FM usage has grown in practice, FM synthesis has become involved in an increasing number of scenarios. In this paper, we have described the workflows and classified several software re-engineering scenarios involving FM synthesis. We have derived requirements from these scenarios for synthesis techniques and briefly contrasted them with the existing techniques. We hope the scenarios and requirements presented in this paper encourage research on new FM synthesis techniques and improvements to existing techniques.

5. REFERENCES

[1] M. Acher. *Managing Multiple Feature Models: Foundations, Language, and Applications*. PhD thesis, University of Nice Sophia Antipolis, France, 2011.

[2] M. Acher, A. Cleve, G. Perrouin, P. Heymans, P. Collet, and C. V. Lahire, Philippe. On extracting feature models from product descriptions. In *VaMoS*, 2012.

[3] V. Alves, C. Schwanninger, L. Barbosa, A. Rashid, P. Sawyer, P. Rayson, C. Pohl, and A. Rummler. An exploratory study of information retrieval techniques in domain analysis. In *SPLC*, 2008.

[4] N. Andersen, K. Czarnecki, S. She, and A. Wasowski. Efficient synthesis of feature models. In *SPLC*, 2012.

[5] D. Batory. Feature models, grammars, and propositional formulas. Technical Report TR-05-14, University of Texas at Austin, Texas, 2005.

[6] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. Variability modeling in the systems software domain. Technical Report GSDLAB-TR 2012-07-06, GSD Lab, University of Waterloo, 2012.

[7] K. Chen, W. Zhang, H. Zhao, and H. Mei. An approach to constructing feature models based on requirements clustering. In *ICSE*, 2005.

[8] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[9] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, MA, 2000.

[10] K. Czarnecki, S. She, and A. Wąsowski. Sample spaces and feature models: There and back again. In *SPLC*, 2008.

[11] K. Czarnecki and A. Wąsowski. Feature models and logics: There and back again. In *SPLC*, 2007.

[12] H. Dumitru, M. Gibiec, N. Hariri, J. Cleland-Huang, B. Mobasher, C. Castro-Herrera, and M. Mirakhorli. On-demand feature recommendations derived from mining public product descriptions. In *ICSE*, 2011.

[13] U. Fahrenberg, A. Legay, and A. Wasowski. Vision paper: Make a difference! (semantically). In *MoDELS*, 2011.

[14] T. Fukuda, Y. Atarashi, and K. Yoshimura. An approach to evaluate time-dependent changes in feature constraints. In *SPLC*, 2011.

[15] M. Janota and G. Botterweck. Formal approach to integrating feature and architecture models. In *FASE*, 2008.

[16] H. P. Jepsen, J. G. Dall, and D. Beuche. Minimally invasive migration to software product lines. In *SPLC*, 2007.

[17] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, CMU, 1990.

[18] M. Mendonca, A. Wąsowski, and K. Czarnecki. SAT-based analysis of feature models is easy. In *SPLC '09*. IEEE, 2009.

[19] N. Niu and S. M. Easterbrook. On-demand cluster analysis for product line functional requirements. In *SPLC*, 2008.

[20] J. Rubin and M. Chechik. From products to product lines using model matching and refactoring. In *SPLC Workshops*, 2010.

[21] J. Rubin and M. Chechik. Combining related products into product lines. In *FASE*, 2012.

[22] U. Ryssel, J. Ploennigs, and K. Kabitzsch. Extraction of feature models from formal contexts. In *FOSD*, 2011.

[23] U. Ryssel, J. Ploennigs, and K. Kabitzsch. Automatic library migration for the generation of hardware-in-the-loop models. *Science of Computer Programming*, 77(2):83 – 95, 2012.

[24] S. She. Feature model mining. Master's thesis, University of Waterloo, Waterloo, ON, Canada, 2008.

[25] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. Reverse engineering feature models. In *ICSE*, 2011.

[26] J. van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *WICSA*, 2001.

[27] N. Weston, R. Chitchyan, and A. Rashid. A framework for constructing semantically composable feature models from natural language requirements. In *SPLC*, 2009.

Recovering Traceability Links between Feature Models and Source Code of Product Variants

Hamzeh Eyal-Salman
LIRMM, University of Montpellier 2
161, rue Ada
34095 Montpellier cedex 5, France
33(0)4 67 41 85 71
Hamzeh.Eyalsalman@lirmm.fr

Abdelhak-Djamel Seriai,
Christophe Dony
LIRMM, University of Montpellier 2
161, rue Ada
34095 Montpellier cedex 5, France
33(0)4 67 41 85 71
{Seriai, Dony}@lirmm.fr

Ra'fat Al-msie'deen
LIRMM, University of Montpellier 2
161, rue Ada
34095 Montpellier cedex 5, France
33(0)4 67 41 85 71
Rafat.Al-msiedeen@lirmm.fr

ABSTRACT
Usually software product variants, developed by clone-and-own approach, form often a starting point for building Software Product Line (SPL). To migrate software products that deemed similar into a product line, it is essential to trace variability among software artifacts because the distinguishing factor between traditional software engineering and software product line engineering is the variability. Variability tracing is used to support conversion from traditional software development into software product line development and automate products derivation process such that core assets can be automatically configured for a product according to the features selection from the feature model. Tracing and maintaining interrelationships between artifacts within a software system also are needed to facilitate program comprehension, make the process of maintaining the system less dependent on individual experts. This paper presents a method based on information retrieval approach namely, latent semantic indexing, to establish traceability links between object-oriented source code of product variants and intended feature model as representative of variability model.

Categories and Subject Descriptors
D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement–*Restructuring, reverse engineering, and reengineering*; H.3.3 **[Information Systems]:** Information Search and Retrieval- *Clustering , Information filtering.*

General Terms
Theory, Design, Documentation.

Keywords
Traceability links, feature models, source code, variability, software product line, latent semantic indexing.

1. INTRODUCTION
Product variants often evolve over the time from an initial product to a family of similar product variants that meet the need of a large group of consumers. The successful development of the initial product attracts new customers. For example, Wingsoft Financial Management System (WFMS) was developed for Fudan University and then evolved many times so that all evolved WFMS systems have been now used in over 100 universities in China [18].

Usually, developers use copy-paste-modify technique to build a new product variant from existing ones. When number of features and product variants grows, ad hoc reuse technique causes critical problems such as: we must maintain each product variant separately from others and it also becomes difficult to find and trace features for reuse in new products.

As these problems accumulate, it became necessary to re-engineering product variants into a SPL for systematic reuse. SPL is a set of software-intensive systems sharing a common, managed set of features that satisfies the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [5]. SPL aims to decreasing development cost and time by developing a family of systems rather than one system at a time [17]. In SPL, there are two models: feature model (FM) as representative of variability model and core asset model. FM has a pivot role because it represents a set of configurations where each valid configuration represents a specific product and it also is extensively used to automate the product derivation process [5]. Figure1 shows an example of a simplified FM inspired from mobile media feature model [3].

There are three issues that must be considered to reengineering product variants into SPL: extraction of a FM for product variants, building SPL artifacts (core assets model) and mapping between FM and SPL artifacts [13].

FM of product variants can be provided by system's developers and domain experts who accompanied and contributed product variants evolution. FM may also be reverse engineered from the documentations of products variants [1].

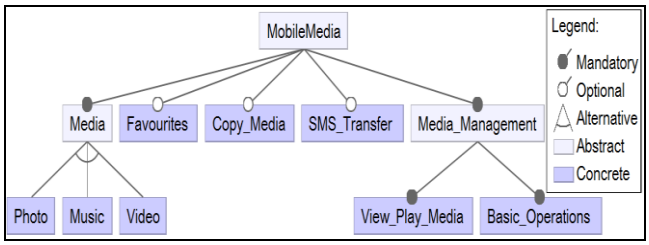


Figure 1. A sample feature model [6].

Regarding building SPL artifacts, the development team can utilize and benefit from the available reusable elements such as: source code, design documents, test cases, etc. to build the required SPL core assets.

Conference'10, Month 1–2, 2010, City, State, Country.

These parts (FM and core asset model) must be connected to exploit them during SPL life cycle [13]. The traceability links between source code of product variants and their FM are be used to automate products derivation process in order to automatically configures all the assets for a product according to the features selection from the FM, ensure consistency between extracted FM and source code, facilitate program comprehension process, make the process of maintaining the system less dependent on individual experts and recovery of various architectural elements.

A lot of work [2,12,19] has addressed recovering traceability links between software artifacts and sometimes between FM and source code but that work does not use information related directly to FM such as common and variable features, and other available information [7].

Our paper proposes a method based on information retrieval (IR) methods namely, latent semantic indexing (LSI), considering information provided by FM to establish and maintain traceability links between source code of product variants and textual description of features as representative of FM. Features description can be extracted from documentation and code comments.

IR has proven useful in many disciplines such as image extraction, speech recognition, horizontal search engines like Google and software maintenance and evolution. Furthermore feature location is one of the most common applications of IR in software engineering [4]. We believe that IR techniques can provide a way to establish the traceability links between source code of product variants and their FM.

The remainder of this paper is organized as follows. Section 2 discusses background and related work. Section 3 shows the traceability links recovery process. Section 4 details latent semantic indexing. Section 5 shows the experimental results. Finally, Section 6 presents conclusions and feature work.

2. BACKGROUND & RELATED WORK

Software traceability is the ability to describe and follow the life of an artifact (requirements, code, tests, models, reports, plans, etc.) developed during the software lifecycle in both forward and backward directions (e.g., from requirements to the software architecture and the code components implementing them and vice-versa) [8].

Traceability relations can refer to overlap, satisfiability, dependency, evolution, generalization/refinement, conflict or rationalization associations between various software artifacts [14]. In general, traceability relations can be classified as horizontal traceability or vertical traceability relations. The former type refers to relationships between different levels of abstraction (e.g. from requirements to design to implementation) and the latter type refers to relationships between artefacts at the same level of abstraction (e.g. between related requirements) [11, 16].

FM is a variability modeling technique widely used in SPLE to cover the variability in all SPL life cycle from requirements to test cases [3]. Variability defines what the allowed combinations of features (also called configurations) are. FM consists of feature diagram and cross tree constraint likes require and exclude constraints. Feature diagram is a tree like representation, the root of the tree refers to the complete system, tree nodes are features and tree edges represent dependency rules [15]. In the literature, there are many definitions of feature; in this paper we will consider the following definition [9]:

> "A distinctively identifiable functional abstraction that must be implemented, tested, delivered, and maintained."

Many researchers attempted to establish traceability link via information retrieval (IR) approach. IR-based approaches assume that all software artifacts are in textual format. Then, they compute textual similarity between two software artifacts using cosine similarity, e.g., a class and a requirement. The three IR methods which commonly used in traceability generation are probabilistic method (PM), vector space method (VSM) and LSI [2].

Antoniol et al. [2] used PM and VSM to establish traceability links between C++ source code to manual pages and Java code to functional requirements. In each method, one type of particular artifacts represents a query and another type of artifacts represents a document. For example, source code represents a query against functional requirements as a document. Traceability links are retrieved by applying cosine similarity between two types of artifacts.

Andrian et al. [12] used LSI to establish traceability links between source code files and manual sections. A set of experiments was conducted and experimental results proved that LSI performs at least as well as Antoniol's methods using PM and VSM with low processing of source code and documentation.

Ghanam et al. [7] assumed that there are traceability links between FM and source code for a given system and as systems evolve, the traceability links become broken or outdated so this work presented an approach to keep the already existing traceability links up to date by using executable acceptance tests (EAT). EAT refers to English-like specifications (such as: scenario tests and story tests) that represents the specifications of a given feature and can be executed against the system to test the correctness of its behaviour.

Ziadi et al. [19] proposed an approach to automate feature identification from the source code of similar products variants. This approach assumes that product variants use the same vocabulary to name packages, classes, attributes and methods; it treats the source code as a set of construction primitives and then applies an algorithm to identify features. However this approach cannot identify the separated mandatory features because it gathers all common construction primitives for all products as a single mandatory feature.

3. OUR APPROACH TO RECOVER TRACEABILITY LINKS

This section describe our proposed approach to recover traceability links between source code of product variants and their FM. Figure 2 gives an overview about the recovering traceability links process. The inputs of this process are product variants FM, features description and object-oriented source code of product variants. The figure also highlights two main phases:

1. Variability point extraction: In this phase, variability points are reversed engineered from the source code where it can reflect four types of variations:
    - Package variation: means a set of packages that make the differences among product variants in term of the provided functionality.

- Class variation: means a set of classes that make the differences among product variants in term of the provided functionality, considering that all product variants have the same packages.
- Method variation: means a set of methods that make the differences among product variants in term of the provided functionality, considering that all product variants have the same classes and packages.
- Attribute variation: means a set of attributes that make the differences among product variants in term of the provided functionality, considering that all product variants have the same methods, classes and packages.

This paper will consider just class variation as a variability point in the source code.

2. Applying a traceability method: In this phase, we will use LSI to recover traceability links between source code and FM according to our mapping model as shown in the figure 3. In this model, variability (variable features) can be implemented by four types of variations: package variation, class variation, method variation and attribute variation.

   In the following sections, we will explain in more details about LSI method.

Figure 2.Traceability recovering process overview.

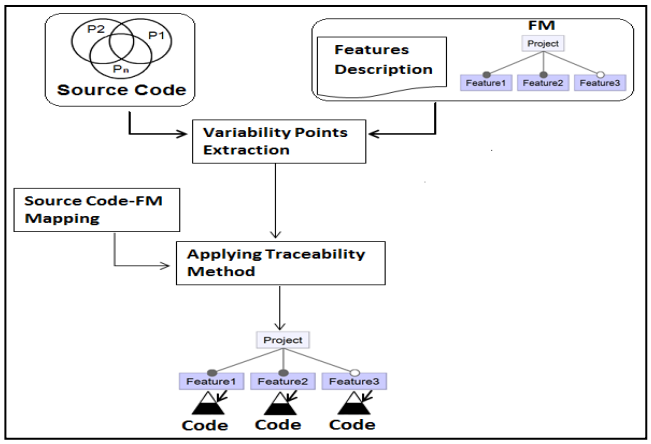
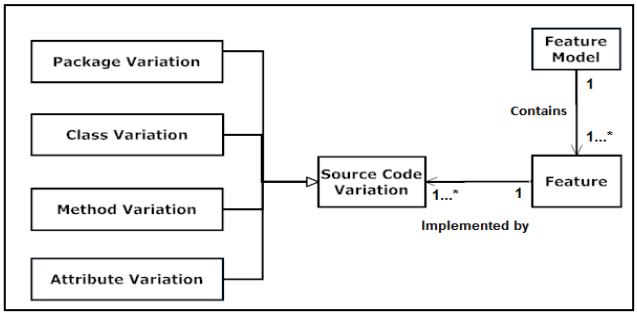


Figure 3: Feature to source code mapping model.



4. LSI AS A RECOVERING TRACEABILITY LINKS METHOD

LSI is a technique that projects queries and documents into a space with latent semantic dimensions. The basic assumption of LSI is that there exists some implicit relationships among the words of documents, that is to say, there are some latent semantic structures in free text (e.g. a set of words that always appears togther has an imlicit relationship). In the LSI, a query and a document can have high cosine similarity even if they do not share any terms - as long as their terms are semantically similar [12].

LSI tries to overcome two classic problems arising in natural languages: synonymy and polysemy, by replacing the original term–document matrix with an approximation. This is done using singular value decomposition (SVD), a technique originally used in signal processing to mitigate noise while preserving the original signal. Assuming that the original term–document matrix is noisy (the aforementioned synonymy and polysemy), the approximation is interpreted as a noise reduced by reducing the dimensionality without great loss of descriptiveness [10].

LSI will use features description as query to retrieve the classes related to the feature. In most object oriented languages classes' names are composed of concatenated terms like (EmailAddressFormatChecker) so that each term reflects partially the class functionality. We assume that programmers use meaningful names (i.e. names derived from the domain) to name classes.

Features descriptions and classes' names must be manipulated and normalized to become suitable as input of LSI. This preprocessing step include: all capital letters must be transformed into lower case letters, removing stop-words (such as articles, punctuation marks, numbers, etc.), all classes' names must be split into terms and performing word stemming.

LSI as a recovering traceability links method is comprised of the following steps [12]:

1. Constructing a term-document matrix whose [i, j]th element refers to the association between the ith term and jth document. In our work, terms represent all extracted words from classes' names and features description while documents represent classes' names. We will measure the weight of each term using TF-IDF (term frequency–inverse document frequency).
2. Dividing term-document matrix into LSI subspace by applying SVD. SVD is performed on the matrix to determine patterns in the relationships among the terms. The optimal number of dimensions is typically around between 200 and 300 and may vary from corpus to corpus, domain to domain.
3. Computing the cosine similarity in LSI subspace by equation1. Where Q refers to a query, i ranges over the entire documents set, W_j refers to elements of query and document vectors.

$$Similarity(Q, D_i) = \frac{\sum W_{Q,j}\, W_{i,j}}{\sqrt{\sum_j W^2_{Q,j}}\ \sqrt{\sum_i W^2_{Q,j}}} \qquad \text{(Eq 1)}$$

4. Filtering results according to a predetermined threshold, and then the traceability links between FM and source code are established. Classes that will be retrieved have a similarity with a feature description greater than or equal the threshold value. In our work, threshold value will be determined in a heuristic way.

The effectiveness of IR methods is measured using IR metrics: recall and precision. For a given query, recall is the percentage of correctly retrieved links over the total number of relevant links while precision is the percentage of correctly retrieved links to the total number of retrieved links (see equation 2 and 3) where (i) represents query set [2].

$$Precision = \frac{\sum_i \#(Relevant_i \wedge Retrieved_i)}{\sum_i \#Retrieved_i} \% \quad (Eq2)$$

$$Recall = \frac{\sum_i \#(Relevant_i \wedge Retrieved_i)}{\sum_i \#Relevant_i} \% \quad (Eq3)$$

Where i ranges over the entire features set. Both measures have values in [0, 1]. If recall equals to 1, it means that all the relevant links are recovered, however there could be recovered links that are not relevant. If the precision equals to 1, it means that all the recovered links are relevant, however there could be relevant links that were not recovered. Choosing a higher threshold for the link recovery will result in higher precision, while lowering the threshold will increase the recall.

It is important to mention her that LSI, in our work, will be applied two times. First, to recover traceability links between common feature and common classes while the second time to recover traceability links between variable features and variable classes. This task aims to improve precision and recall values by reducing search space. We can extract common classes by conducting a lexical matching among product variants' classes while other classes form variable classes.

5. EXPERIMENTAL RESULTS

In order to validate our approach as traceability recovering method between source code and FM, we will consider simple mobile media software to test this method. Figure 2, in the introduction section, represent a simplified FM for mobile media software. **Favourites**, **Copy_Media** and **SMS_Transfer** are optional features while **View_Play_media** and **Basic_Operations** are mandatory features. **Media** is alternative feature. Three configurations were chosen to realize three products including all mobile media features. In order to process a source code, a java parser like abstract syntax tree (AST) is needed to extract all source code elements.

We assumed that each feature is described with certain words as shown in the table 1 below. For example, **SMS_Transfer** feature in the row 6 is described by **send photo and receive photo**.

Also, we assumed that each feature is implemented with certain classes as shown in the table 2. For example, **Favourites** feature in the row 4 is implemented by **SetFavourites** and **ViewFavourites classes**.

Table1. Features description.

#	Feature name	Description
1	photo	capture photo, compression photo, scrambling photo, count photo
2	Music	Play, generate music, organize music
3	Video	capture video, compression video, scrambling video
4	Favourites	set favourites, view favourites, Save favourites
5	Copy_Media	copy media, store media
6	SMS_Transfer	send photo, receive photo
7	Basic_Operations	create media, delete media, edit media, label media, sort media, move media, search media,save media
8	View_Play_Media	view media, play media

Table 3 summarizes the results we obtained on recovering the traceability links between source code and FM using LSI. The first column represents the threshold value of cosine similarity. Traceability links with similarity value greater than or equal to the threshold value just will be considered.

Table 2. Real implementation of features

#	Feature	Classe Name	Class ID
1	Photo	CapturePhoto	1
		CompressionPhoto	2
		ScramblingPhoto	3
		CountPhoto	4
2	Music	GenerateMusic	5
		OrganizeMusic	6
3	Video	CaptureVideo	7
		CompressionVideo	8
		ScramblingVideo	9
4	Favourites	SetFavourites	10
		ViewFavourites	11
		SaveFavourites	12
5	Copy_Media	CopyMedia	13
		StoreMedia	14
6	SMS_Transfer	SendPhoto	15
		recievePhoto	16
7	Basic_Operation	CreateMedia	17
		DeleteMedia	18
		EditMedia	19
		LabelMedia	20
		SortingMedia	21
		MoveMedia	22
		SearchMedia	23
		SaveMedia	24
8	View_Play_Media	ViewMedia	25
		PlayMedia	26

Column 2 represents the number of correct links retrieved, column 3 represents the number of incorrect links retrieved, column 4 represents the number of correct links that were not retrieved, column 5 represents the total number of retrieved links (correct + incorrect), and the last two columns are the precision and recall values.

Table 3. LSI results.

Threshold	Correct links retrieved	Incorrect links retrieved	Missed links	Total links retrieved	Precision	Recall
0.50	21	5	5	26	0.80	0.80
0.55	20	3	6	23	0.87	0.77
0.6	20	2	6	22	0.91	0.77

Table 4 shows an example about the traceability links between source code of product variants and their FM. Based on LSI method, **Copy_Media** feature is linked to **StoreMedia**, **PlayMedia** and **CopyMedia** classes with cosine similarity greater than or equal to 0.60. When comparing the expected implementation of Copy_Media feature with its real implementation, we found that number of correct links equals to **2**, number of incorrect links equals to **1** while number of missed link equals **0**.

Table 4. An example for traceability links.

Feature Name	Class Name	Similarity
Copy_Media	StoreMedia	0.9517
	PlayMedia	0.7844
	CopyMedia	0.6797

Based on table 3, we can note that recall and precision values are the same when threshold equals 0.50 because number of retrieved links equals number of relevant links. Also, we can note that when the threshold value increases, precision improves while recall deteriorates. Based on our example, LSI gives high precision and

recall values when it is used to establish the traceability link as show in the table 3.

6. CONCLUSIONS AND FUTURE WORK

This paper presents a method based on information retrieval namely, LSI, to establish traceability links between object-oriented source code of product variants and their FM. This method establishes the traceability links based on cosine similarity between features description and classes names.

The results obtained in the simple reported case study proved that, in general, LSI can be used to recover traceability links between FM and object oriented source code of product variants with high recall and precision to achieve the following main targets:

- To support conversion from traditional software development into software product line development and that leads to automate products derivation process.
- To ensure consistency between the FM and the code artifact by tracing commonality and variability in the source code artefact.
- To facilitate software evolution and maintenance.

As future work we will consider other types of source code variations (package variation, method variation and attribute variation) and use all other information provided by the FM (such as cross tree constraints, alternative features and ect.) to recover more reliable traceability links. Also we can combine LSI with formal concept analysis (FCA) to find dependency between product variants features.

7. REFERENCES

[1]. Acher, M., Cleve, A., Perrouin, G., Heymans, P., Vanbeneden, C., Collet, P. and Lahire, P. 2012. On extracting feature models from product descriptions. In Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS '12). ACM, New York, NY, USA, 45-54.

[2]. Antoniol, G., Canfora, G., Casazza, G., Lucia, A. and Merlo, E. 2002. Recovering Traceability Links between Code and Documentation. IEEE Trans. Softw. Eng. 28, 10 (October 2002), 970-983.

[3]. Benavides, D., Segura, S. and Ruiz-Cort, A. 2010. Automated analysis of feature models 20 years later: A literature review. Inf. Syst. 35, 6 (September 2010), 615-636.

[4]. Binkley, D. and Lawrie, D. 2010. Information Retrieval Applications in Software Maintenance and Evolution. In Encyclopedia of Software Engineering (P. Laplante, ed.), Taylor & Francis LLC. 454-463.

[5]. Clements, P. and Northrop, L. 2001. Software product lines: practices patterns. Addison-Wesley Longman Publishing Co., Boston, MA, USA,

[6]. Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Filho, F. and Dantas, F. 2008. Evolving software product lines with aspects: an empirical study on design stability. In Proceedings of the 30th international conference on Software engineering (ICSE '08). ACM, New York, NY, USA, 261-270.

[7]. Ghanam, Y. and Maurer, F. 2010. Linking feature models to code artifacts using executable acceptance tests. In Proceedings of the 14th international conference on Software product lines: going beyond (SPLC'10), Jan Bosch and Jaejoon Lee (Eds.). Springer-Verlag, Berlin, Heidelberg, 211-225.

[8]. Gotel, O. and Finkelstein, C.1994. An analysis of the requirements traceability problem. In Proceedings of 1st International Conference on Requirements Engineering (Colorado Springs, CO). IEEE Computer Society Press, Los Alamitos, CA, 94–101.

[9]. Kang, K., Kim, S., Lee, J., Kim, K., Shin, E., and Huh, M. (1998). FORM: a feature oriented reuse method with domain-specific reference architectures. Annals of Software Engineering, 5(1):143–168.

[10]. Kuhn,A., Ducasse, S. and Girba,T. 2007. Semantic clustering: Identifying topics in source code. Inf. Softw. Technol. 49, 3 (March 2007), 230-243.

[11]. Lindvall, M. and Sandahl, K. 1996. Practical implications of traceability. Softw. Pract. Exper. 26, 10 (October 1996), 1161-1180.

[12]. Marcus, A. and Maletic, J. 2003. Recovering documentation-to-source-code traceability links using latent semantic indexing. In Proceedings of the 25th International Conference on Software Engineering (ICSE '03). IEEE Computer Society, Washington, DC, USA, 125-135.

[13]. Pohl,K., Böckle, G. and Linden, F. 2005. Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[14]. Ramesh, B. and Jarke, M. 2001. Toward Reference Models for Requirements Traceability. IEEE Trans. Softw. Eng. 27, 1 (January 2001), 58-93.

[15]. Schobbens, P.Y, Heymans, P. and Trigaux, J.C. 2006. Feature Diagrams: A Survey and a Formal Semantics. In Proceedings of the 14th IEEE International Requirements Engineering Conference (RE '06). IEEE Computer Society, Washington, DC, USA, 136-145.

[16]. Spanoudakis, G. and Zisman, A., Software Traceability: A Roadmap, in Handbook of Software Engineering and Knowledge Engineering, Chang, S. K., Ed. World Scientific Publishing Co, 2005, pp. 395-428.

[17]. Weiss, D. and Lai, C. 1999. Software Product-Line Engineering: A Family-Based Software Development Process. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[18]. Ye, P., Peng, X., Xue, Y. and Jarzabek, S.2009: A Case Study of Variation Mechanism in an Industrial Product Line. ICSR. 126-136.

[19]. Ziadi, T., Frias, L., Silva, M. and Ziane, M. 2012. Feature Identification from the Source Code of Product Variants. 16th European Conference on Software Maintenance and Reengineering. 417-422.

Applying CVL to Business Process Variability Management

Clara Ayora, Victoria Torres, Vicente Pelechano
PROS, Universitat Politècnica de València
Camino de Vera s/n, 46022
Valencia, Spain
{cayora,vtorres,pele}@pros.upv.es

Germán H. Alférez
Universidad de Montemorelos
Apartado 16-5
Montemorelos N.L., Mexico
harveyalferez@um.edu.mx

ABSTRACT

Business Processes (BP) exist in many variants depending on the application context. The use of variability mechanisms in BPs becomes essential for organizations to reduce development and maintenance efforts. However, such mechanisms entail big challenges. At design time, poor model specifications can turn process variants into difficult-to-maintain and understand artifacts. At runtime, process variants require continuous adaptations to achieve business goals in highly dynamic execution contexts. In order to address these challenges, in this paper we present a proposal to manage process variants at design time and runtime. At design time, process variants are modeled by combining a BP modeling language with the *Common Variability Language*. Then, at runtime, based on the created models and on the current context, process variants are adapted by means of MoRE-BP, a reconfiguration engine that is capable of performing dynamic adaptations automatically. An online-shop scenario illustrates our proposal and a proof-of-concept prototype validates its feasibility.

Keywords

Business Process Variability, CVL, Self-Adaptive Business Processes, Models at Runtime

1. INTRODUCTION

A Business Process (BP) consists of a set of related activities whose execution reaches a specific goal [29]. The increasing adoption of BPs in recent years has resulted in large repositories with numerous collections of BP models [24, 11]. Since these models often vary depending on the application *context* (i.e., the execution environment) [13, 23], the existing repositories usually comprise large collections of related *process variants*. Examples of such collections can be found in almost every domain. For example, [14] describes a repository for vehicle repair and maintenance comprising more than 900 process variants that depend on country, garage, and vehicle differences. Another research shows more than 90 process variants for handling medical examinations [20]. These process variants pursue the same or similar business objective (maintenance of vehicles in a garage or the treatment of a patient). However, they differ in their logic due to varying application context either at design time (e.g. existing regulations in different countries and regions) or runtime (e.g. type of car being repaired) [11, 26]. In such situations, managing properly process variants constitutes a fundamental challenge to reduce development and maintenance efforts [21]. However, managing process variants is not a trivial task. Although different proposals have been developed to deal with variability in BPs at design time [22, 25, 14], their current support to represent such variability is limited. For example, not all the perspectives contained in a BP model (i.e., functional, behavioral, organizational, informational, operational, and temporal) are addressed. Therefore, process variants become error-prone and complex to build, manage, and understand [13]. In addition, to deal with highly dynamic execution contexts, adaptation techniques are necessary to properly change running process variants so that the real business world is accurately reflected [28]. Therefore, *adaptability* of process variants at runtime emerges as a necessary underlying capability for BPs that are executed in changing contexts [16].

Our contribution is a proposal to support process variants at design time and runtime. On one hand, at design time we define the process variants in such a way that variability is considered as a first-class concern during the modeling process. For this purpose, we choose the *Common Variability Language* (CVL) [15]. CVL provides the mechanisms to represent variations in any *Domain Specific Language* (DSL). A big advantage of using CVL is that DSLs do not have to be extended or overloaded with variability information [12]. Specifically, we combine CVL with the *Business Process Modeling Notation* (BPMN), the standard DSL to represent BPs. CVL plays a key role when combining it with BPMN because neither the current specification of BPMN supports process variability modeling nor current BPMN execution engines support variability. On the other hand, at runtime we rely on *models at runtime* [7] to perform dynamic adaptations of process variants according to the current context. Instead of programming cumbersome and error-prone code to carry out such adaptations, models at runtime allow to reuse the knowledge created with CVL at design time to guide the adaptations during execution over the underlying technologies [8, 1]. Adaptations are automatically achieved by our *Model-based Reconfiguration*

Engine for Business Processes (MoRE-BP). MoRE-BP automatically transforms the adapted process into executable code (i.e., BPEL code), which in turn is hot-deployed in an execution engine. MoRE-BP is an extension of MoRE-WS, which has been successfully used in the autonomic adaptation of Web service compositions using models at runtime [1].

The reminder of this paper is structured as follows: Section 2 provides the foundations of our proposal. Section 3 presents the case study that is used to illustrate the proposal. Section 4 describes how we make use of CVL to deal with BP variability. Section 5 describes our proof-of-concept prototype. Section 6 presents related work. Finally, Section 7 concludes the paper and outlines the future work.

2. FOUNDATIONS

This section provides the foundations for the three major topics our proposal relies on, which are BP modeling, models at runtime, and autonomic computing.

2.1 Business Process Modeling

BP models define how business goals can be achieved. Thus, they should accurately represent the organizational reality relevant for reaching a specific goal [29]. This is done by the following set of concepts: *Activities, Connectors, Edges, Resources, Events, Operations,* and *Data objects*. These concepts define the activities that should be done (*what*), their coordination and implementation (*how*), the events that trigger the activities (*when*), and the resources (*who*) and *data* that are involved. In combination, such concepts cover the different perspectives that can be found in a BP model (i.e., functional, behavioral, organizational, informational, temporal, and operational) [10]. In our previous work, we have focused on supporting variability in the organizational, informational, and temporal perspectives [2]. In this paper, we extend our previous research to support variability in the functional and behavioral perspectives (i.e., activities and their coordination).

2.2 Models at Runtime

Models are typically used to describe systems using concepts that abstract the system knowledge over the underlying computing technologies. The purpose of *models at runtime* is to extend the use of models from design time to execution time [7]. In this way, the modeling effort made at design time is not only useful for producing the system, but also provides a richer semantic base for reasoning, monitoring, or adapting the system during execution [8, 1]. Thus, by using models at runtime, the knowledge gathered in BP models can be used during execution to drive BP adaptations. In order to use BP models at runtime, they have to be linked in such a way that they constantly mirror the system and its current state and behavior. It means that if the system changes, the models change accordingly, and vice versa.

2.3 Autonomic Computing

Inspired by biology, *autonomic computing* [17] has evolved as a discipline to create software that self-manage in a bid to overcome the complexities to maintain systems effectively. Autonomic computing covers the broad spectrum of computing in domains as diverse as mobile devices [30] and home-automation [8], thereby demonstrating its feasibility and value by automating tasks such as installation, healing, and updating. Since doing manual adaptations is a difficult task, our proposal is based on IBM's reference model for autonomic control loops [18] (which is sometimes called the MAPE-K loop) to automatically adapt process variants.

3. CASE STUDY

In order to illustrate our proposal, we use a typical process for product shopping in an online shop such as *Amazon* (cf. Fig. 1). The process starts when a customer looks for a product on the shop's website. If the product is not found, the customer can refine the search. Otherwise, the product information and a list of related products are presented to the customer. Afterwards, the customer selects the product in which he or she is interested in. Once this selection has been performed, the customer is identified as an authorized user. If the identification is successful, the product has to be paid. The payment is carried out depending on the customer preferences: *Visa* credit card, *PayPal*, *mobile phone* payment, or using the *shop's card* (a special credit card for VIP customers). When using the *shop's card*, the system asks for a specific PIN that only the VIP members know. Then, the product can be delivered to the customer by a shipping company or the customer may want to pick it up in a collection point. Usually, delivery is carried out by *UPS*. However, if *UPS* does not operate in a specific country, delivery is carried out by *DHL* or *FedEX*. The process finishes by sending a confirmation e-mail to the customer.

Despite it is a simple process, there are 16 process variants depending on either the type of payment or the shipping company. According to the process specification, all possible process variants from the online product shopping process (cf. Fig. 1) can be defined at design time since the different alternatives (e.g. *UPS* and *DHL*) are known in advance. However, depending on runtime decisions (e.g. customer preferences), some adaptations of process variants may be required. In addition, since an online shopping process is expected to be available 24/7, these adaptations need to be carried out without stopping the system.

4. HANDLING VARIABILITY IN BUSINESS PROCESSES

This section describes our proposal to manage variability in BPs which covers two phases: 1) at design time when the process variants are modeled, and 2) at runtime when the process variants are adapted in response to context changes.

4.1 Process Variant Modeling

At design time, the aim is to properly model the existing process variants. Current proposals model process variants by extending the original DSLs (e.g. BPMN, EPC, or UML Activity Diagrams) [22, 25, 14]. However, most of these works result in complex models that are overloaded with variability specifications [13]. On the contrary, CVL provides the mechanisms to represent variations in a separate

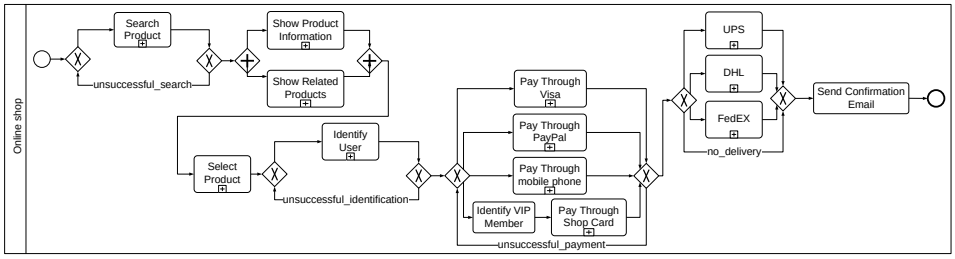


Figure 1: BPMN model that represents the online product shopping process

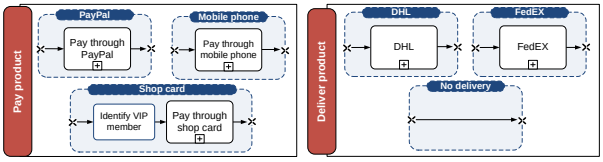


Figure 3: *Variation model* for the online product shopping process

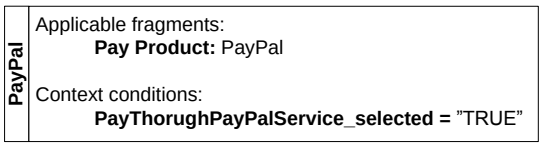


Figure 4: Example of a *resolution model* for the *Pay product placement*

way. Therefore, CVL alleviates the impact that variability issues have on the BP model, resulting in better legibility, understandability, and scalability [15]. CVL is based on the Base-Variation-Resolution (BVR) approach that is supported by three models: the *base model*, the *variation model*, and the *resolution model* [6]. We model the process variants according to this set of models.

First, the *base model* is used to specify commonalities between the process variants (i.e., process fragments that are shared by all process variants). In addition, this model specifies the *placement fragments* (*placements* for short) of the model that may vary (i.e., variation points). Fig. 2 depicts the *base model* of the online product shopping process. Only two variation points are defined in this case (i.e., *Pay product* and *Deliver product*). These variation points already include process fragments for their execution (i.e., *Pay through visa* and *UPS* respectively).

Then, the *variation model* is used to specify the *replacement fragments* (*replacement* for short) that alternatively exist for the *placements* defined in the *base model*. Fig. 3 shows the *variation model* for the online product shopping process. Specifically, it depicts the possible *replacements* for the *Pay product* and *Deliver product placements*.

The *resolution model* is used to specify the set of *context conditions* that determine the conditions under which the *replacements* can be instantiated. Fig. 4 illustrates an example of a *resolution model* for the online product shopping process. This model is structured in two blocks: 1) the *applicable fragments* block specifies the selected *replacements* for the *placements* of the *base model* (i.e., the *Pay product placement* will be instantiated with the *PayPal replacement*); 2) the *context conditions* block specifies the *context conditions* that determine the instantiation of the *replacements*. For example, in order to instantiate the *PayPal replacement*, the *PayThroughPayPalService_selected* context variable has to be valued to TRUE. The value of this variable is stated at runtime when the customer selects the type of payment.

Finally, we propose a *context model* to support the formal reasoning of the context information. Specifically, we propose and ontology-based model since it provides a strong semantic vocabulary for the representation of the context knowledge and for describing specific situations in the context [1]. The main benefit of the context model is that it enables the analysis of the domain knowledge using first-order logic.

4.2 Variability Management at Runtime

Since the BP current execution context can be highly dynamic, process variants need to be adapted at runtime to better achieve business goals. Therefore, this section describes our proposal to perform the dynamic adaptation of process variants. Specifically, the *base model* is dynamically adapted according to the *variation model*. Adaptations are supported by a computing infrastructure based on the components of the MAPE-K loop, i.e., *Monitor, Analyze, Plan, Execute* and *Knowledge* (cf. Fig. 5) [18]. In our case, *sensors* collect information about the BP's context and *actuators* carry out changes in the BP at runtime. Variability management at runtime is described as follows in the context of MAPE-K phases.

Monitor. In order to support the dynamic adaptation of the *base model*, it is necessary to collect information about the context. This task is in charge of the *Context Monitor*. The collected information is used to update a stream database that deals with continuous online data streams.

Analyze. The stream database that is updated with the measures taken from the context needs to be queried to determine if any adaptation has to be carried out. This task is in charge of MoRE-BP, which periodically queries the stream database to find new context information. When a new context event is found, MoRE-BP inserts it into the *context model*. Then, MoRE-BP evaluates the values of this model to find out if a context condition has been accomplished. For

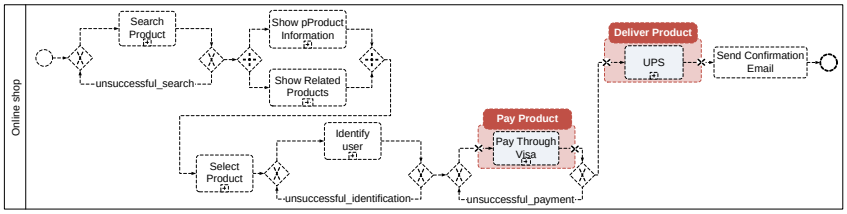


Figure 2: *Base model* for the online product shopping process

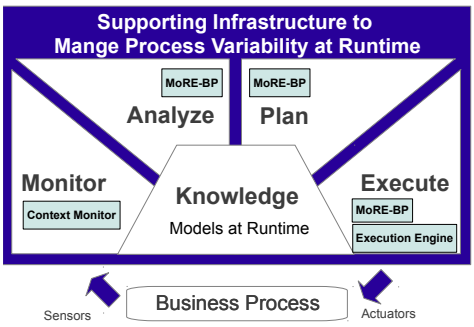


Figure 5: Variability management at runtime

instance, the *PayThroughPayPalService_selected*=TRUE context event (i.e., the user prefers to pay through *PayPal*) accomplishes the *PayThroughPayPalService_selected* context condition that triggers an adaptation in the *Pay Product placement*.

Plan. In this step, MoRE-BP generates an *Adatation Plan*, which contains a list of actions to adapt the *base model*. These actions are stated as CVL actions called *fragment substitutions*. Concretely, one *fragment substitution* replaces the process fragment included in any *placement* of the *base model* with any *replacement* of the *variation model*. For such substitutions, CVL uses the concept of *boundary points*. These points record the inbound and outbound references to and from the *placement* where the substitution takes place, the *replacement* to be replaced, and the new *replacement* (i.e., fragment connections). Therefore, a *fragment substitution* (FS) can be defined as follows:

FS = ((*PlacementInboundReference*, *FragmentToBeReplaced*, *PlacementOutboundReference*), (*ReplacementInboundReference*, *Replacement*, *ReplacementOutboundReference*))

For example, given the *PayThroughPayPalService_selected*= TRUE context condition, the resulting *Adaptation Plan* is the following:

FS = ((*Pay ProductsInputSequenceEdge*, *Pay through Visa*, *Pay ProductsOutputSequenceEdge*), (*PayThroughPayPalInputSequenceEdge*, *Pay through PayPal*, *PayThrough PayPalOutputSequenceEdge*))

Execute. Once the adapted *base model* is obtained, it is necessary to hot deploy[1] it in the *Execution Engine*. To this end, MoRE-BP creates a *deployment directory* for all the relevant deployment artifacts (i.e., the deployment descriptor and the process schema (i.e., the BPEL file). This directory is put into the Web application directory of the *Execution Engine*. We have implemented a versioning strategy for the *deployment directory* to prevent the *Execution Engine* from deleting all the running process instances when a new process schema is deployed. To this end, a new *deployment directory* with an increasing version number is deployed with every adaptation. This approach is in line with the dynamic adaptation of service compositions at the process schema level [27]. In this way, adaptations that are applied to the process schema are propagated to all process instances that run on this schema. Then, the adapted *base model* is transformed into the executable code (i.e., BPEL), which is in turn hot deployed in the execution engine.

[1]Hot deployment is the process of adding new components to a running server without having to restart it

5. PROTOTYPE IMPLEMENTATION

A prototype system validates the feasibility of our proposal. In order to create the set of models described in Sec. 4, we use the following tools. First, the *base model* and the *variation model* are defined with the *Eclipse BPMN Modeler*[2]. Then, we make use of the *CVL Editor*[3] to create a *CVL Model*. This model supports CVL variability elements (i.e., *placements* and *replacements*) independently from the original DSL (e.g. BPMN). Simple references are used to relate these elements to the fragments of the *base* and *variation models* that conform to the *placements* and the *replacements*, respectively (cf. Fig. 6). A set of APIs have been implemented for integrating the CVL Editor with the Eclipse BPMN Modeler in order to define these relations. The resulting CVL-enabled BPMN Editor highlights with colors the referenced fragments. This helps business analysts to define the *CVL Model*. The CVL Editor also allows to define the *resolution model* by adding to the *CVL Model* textual specifications of the *context conditions* that determine when adaptations should be carried out.

Astro [4] was chosen as the *Context Monitor* because it provides an infrastructure that monitors the BP logic. Concretely, ASTRO reports whether a user changes the preferred payment method or introduces a new shipping address. MoRE-BP makes use of the SPARQL Protocol and RDF Query Language (SPARQL)[4] in order to analyze the collected contextual information. In order to obtain the executable BPEL code from the adapted *base model*, MoRE-

[2]http://www.eclipse.org/projects/project.php?id=soa.bpmnmodeler
[3]http://www.omgwiki.org/variability/doku.php
[4]http://www.w3.org/TR/rdf-sparql-query

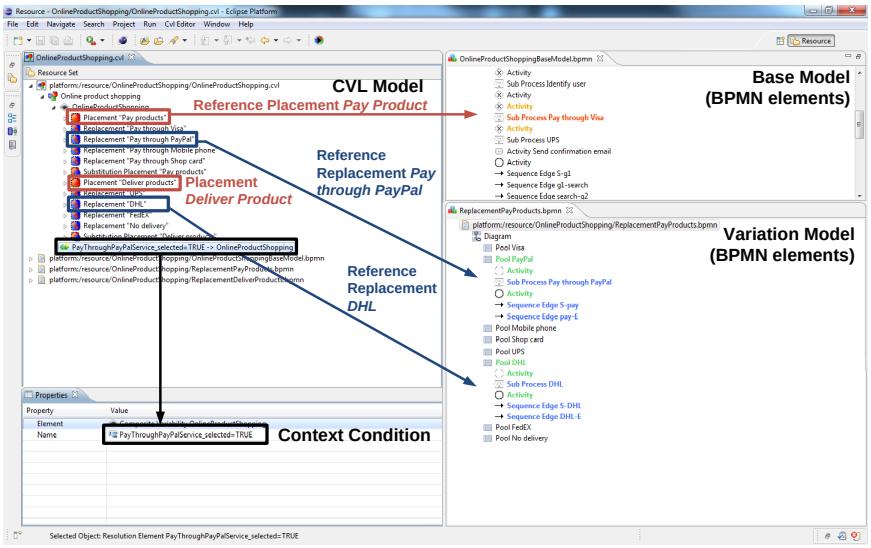


Figure 6: Running prototype

BP makes use of the *Babel* tool[5]. Additional details about the BPMN (i.e., *base model*) to BPEL (i.e., executable version) transformation can be found on our website[6]. The adapted BPEL code is hot-deployed in the Apache Orchestration Director Engine (Apache ODE)[7]. Apache ODE was chosen as the industrial BP engine because it is compliant with the widely used BPEL standard and it offers mature hot-deployment support. A demonstration of our prototype is available[8].

6. RELATED WORK

In this section, we describe research works related to BP variability management. Since our work spans through design time and runtime, the related work is divided into these two phases.

6.1 Process Variant Management at Design time

This section describes relevant research works that deal with BP variability at design time. These works were developed due to the limitation of DSLs to properly model variability in BPs. First, PESOA [22] abstracts the BP in an unique model that includes a set of annotations that identify variable behavior. In [25], the authors define C-EPC, a language extension to configure reference BP models that formalize recommended practices for specific domains. A single BP model contains configurable elements, alternatives that depend on the context of use, and context conditions. In [14], the authors propose an operational approach, named Provop, that allows to configure process variants at design time by adjusting a *base model* to a given context through a set of high-level change operations (INSERT, DELETE, MOVE and MODIFY). The operations that constitute a process variant are selected after evaluating a given context through a set of *context variables*. In order to carry out dynamic adaptations, variants are split via conditional branches, where the split condition corresponds to the context variable of the option.

Unlike our proposal, PESOA and C-EPC integrate all possible process variants in a single model resulting in large and difficult-to-understand models. In addition, they define process variants statically, which makes it impossible to adapt them at runtime after deployment. Thus, if any change is produced, redeployment of the whole set of process variants is needed. Although process variants are defined through a set of change operations in Provop, it forces the deployment of all process variants using conditional branching. This is not real adaptability since all alternatives are transformed into executable versions [3].

6.2 Dynamic Adaptation of Business Processes

Several research works have implemented dynamic adaptations for BPs at the language level. SCENE [9] extends BPEL with Event Condition Action (ECA) rules that define consequences for conditions to guide the execution of binding and rebinding self-reconfiguration operations. VxBPEL [19] is an adaptation of BPEL that allows adapting a BP in a service-centric system. In [5], monitoring directives are expressed in Web Service Constraint Language (WSCoL), and recovery strategies, which follow the ECA paradigm, are stated in the Web Service Recovery Language (WSReL). CEVICHE [16] is a framework that makes use of Complex Event Processing to implement context-adaptive BPs. By extending BPEL, CEVICHE allows the user to directly include into the code the adaptation points and conditions that are required to create dynamic adaptable BPs.

We argue that the dynamic adaptation of BPs at the language level is complex and time consuming, especially in large systems. On the contrary to these revised works, we provide a model-based solution that describes these adaptations at a more abstract level (i.e., within the *resolution*

[5]http://www.bpm.scitech.qut.edu.au/research/projects/oldprojects/babel/tools/
[6]http://www.pros.upv.es/labs/index.php?option=com_content&task=view&id=14&Itemid=35
[7]http://ode.apache.org
[8]http://www.pros.upv.es/images/stories/videos/demoClaraAyora20120919.avi

model), which is more intuitive for non-technical stakeholders.

7. CONCLUSIONS

In this paper, we have described a proposal to manage BP variability based on the *Common Variability Language*. First, we have used CVL to model the possible process variants. Since CVL is an independent language, no annotations or variability concepts need to be added to the original DSL (i.e., *Business Process Modeling Notation*). Therefore, CVL improves the quality of the model in terms of legibility, understandability, and scalability. At runtime, MoRE-BP uses the CVL specifications to perform dynamic adaptations of the process variants. Also, we have presented a proof-of-concept prototype to illustrate our proposal. We learned that CVL facilitates process variant modeling and guides self adaptations of BPs in an abstract manner. As future work, we will provide constraint mechanisms to ensure consistent resolutions leading to well-formed process variant models. In addition, we will investigate the adaptation of process variants in response to unexpected context changes. These changes may need the definition of new alternatives that were not considered at design time.

Acknowledgements

This work has been developed with the support of MICINN under the project EVERYWARE TIN2010-18011.

8. REFERENCES

[1] Alferez, G.H., Pelechano, V.: Context-Aware Autonomous Web Services in Software Product Lines. In Proc. SPLC'11, 100–109 (2011).

[2] Ayora, C., Torres, V., Pelechano, V.: Feature Modeling to deal with Variability in Business Process Perspectives. In Proc. JCIS'12, (to appear).

[3] Ayora, C., Torres, V., Reichert, M, Weber, B, Pelechano, V.: Towards Run-time flexibility for Process Families: Open Issues and Research Challenges. In BPM Workshops 2012, (to appear).

[4] Barbon, F., Traverso, P., Pistore, M., Trainotti, M.: Run-Time Monitoring of Instances and Classes of Web Service Compositions. In Proc. ICWS, 63–71 (2006).

[5] Baresi, L., Guinea, S.: Self-Supervising BPEL Processes. IEEE Trans. Soft. Eng. 37(2), 247–263 (2011).

[6] Bayer, J., Gerard, S., Haugen, Ø., Mansell, J., Møller-Pedersen, B., Oldevik, J., Tessier, P., Thibault, J.P, Widen, T.: Consolidated Product Line Variability Modeling. In Proc. SPLC'06, 195–241 (2006).

[7] Blair, G., Bencomo, N., France, R.B.: Models@run.time. Computing 42, 22–27 (2009).

[8] Cetina, C., Giner, P., Fons, J, Pelechano, V: Autonomic computing through reuse of variability models at runtime: the case of smart homes. Computing 42(10), 37–43 (2009).

[9] Colombo, M., Di Nitto, E., Mauri, M.: SCENE: A Service Composition Execution Environment Supporting Dynamic Changes Disciplined Through Rules. In Proc. ICSOC'06, 191–202 (2006).

[10] Curtis, B., Kellner, M., Over, J.: Process modeling. Communication of the ACM 35(9), 75–90 (1992).

[11] Dijkman, R., Dumas, M., van Dongen, B., Käärik, R., Mendling, J.: Similarity of business process models: metrics and evaluation. Information Systems 36(2), 498–516 (2011).

[12] Fleurey, F., Haugen, Ø., Møller-Pedersen, B., Svendsen, A., and Zhang, X.: Standardizing Variability - Challenges and Solutions. In Proc. SDL'11, 233–246 (2011).

[13] Hallerbach, A., Bauer, T., Reichert, M.: Context-based configuration of process variants. In Proc. TCoB'08, 31–40 (2008).

[14] Hallerbach, A., Bauer, T., Reichert, M.: Capturing variability in business process models: the Provop approach. Soft. Proc.: Impro. and Prac. 22(6–7), 519–546 (2010).

[15] Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G.K., Svendsen, A.: Adding Standardized Variability to Domain Specific Languages. In Proc. SPLC'08, 139–148 (2008).

[16] Hermosillo, G., Seinturier, L., Duchien, L.: Creating Context-Adaptive Business Process. In Proc. ICSOC'08, 228–242 (2008).

[17] Horn, P.: autonomic computing: IBM's perspective on the state of information technology (2001).

[18] Technical Report: An architectural blueprint for autonomic computing. IBM (2003).

[19] Koning, M., ai Sun, C., Sinnema, M., Avgeriou, P.: VxBPEL: Supporting variability for web services in BPEL. Inf. and Soft. Tech. 51(2), 258–269 (2009).

[20] Li, C, Reichert, M., Wombacher, A.: Mining business process variants: challenges, scenarios, algorithms. Data & Knowledge Engineering 70 (5), 409–434 (2011).

[21] Müller, D., Herbst, J., Hammori, M., Reichert, M.: IT support for release management processes in the automotive industry. In Proc. BPM'06, 368–377 (2006).

[22] Puhlmann, F., Schnieders, A., Weiland, J., Weske, M.: Variability mechanisms for process models. Technical report, BMBF-Project (2006).

[23] Reinhartz-Berger, I., Soffer, P., Sturm, A.: Organizational reference models: supporting an adequate design of local business processes. IBPIM 4(2), 134–149 (2009).

[24] Rosemann, M. Potential pitfalls of process modeling: Part A. Business Process Management Journal 12(2), 249–254 (2006).

[25] Rosemann, M., van der Aalst, W.M.P.: A configurable reference modeling language. Inf. Syst. 32(1), 1–23 (2007).

[26] Vervuurt, M.: Modeling business process variability: a search for innovative solutions to business process variability modeling problems. Student Theses of University of Twente. October 2007.

[27] Weber, B., Reichert, M., Rinderle-Ma, S.: Change patterns and change support features - Enhancing flexibility in process-aware information systems. Data & Knoweldge Engineering 66(3),438–466 (2008).

[28] Weber, B. Sadiq, S. Reichert, M. Beyond rigidity - dynamic process lifecycle support. Comp. Science 23, 47–65 (2009).

[29] Weske, M.: Business process management: concepts, languages, architectures. Springer-Verlag Berlin Heidelberg Publisher (2007).

[30] White, J., Schmidt, D.C., Wuchner, E., Nechypurenko, A.: Automating product-line variant selection for mobile devices. In Proc. SPLC'07, 129–140 (2007).

Automatic selection of test execution plans from a Video Conferencing System Product Line

Shuai Wang
Certus Software V&V Center,
Simula Research Laboratory
Oslo, Norway
shuai@simula.no

Arnaud Gotlieb
Certus Software V&V Center,
Simula Research Laboratory
Oslo, Norway
arnaud@simula.no

Marius Liaaen
CISCO Systems
Oslo, Norway
marius.Liaaen@cisco.com

Lionel C. Briand
Certus Software V&V Center,
Simula Research Laboratory
SnT Center, University of
Luxembourg
lionel.briand@uni.lu

ABSTRACT

The Cisco Video Conferencing Systems (VCS) Product Line is composed of many distinct products that can be configured in many different ways. The validation of this product line is currently performed manually during test plan design and test executions' scheduling. For example, the testing of a specific VCS product leads to the manual selection of a set of test cases to be executed and scheduled, depending on the functionalities that are available on the product. In this paper, we develop an alternative approach where the variability of the VCS Product Line is captured by a feature model, while the variability within the set of test cases is captured by a component family model. Using the well-known pure::variants tool approach that establishes links between those two models through restrictions, we can obtain relevant test cases automatically for the testing of a new VCS product. The novelty in this paper lies in the design of a large component family model that organizes a complex test cases structure. We envision a large gain in terms of manpower when a new product is issued and needs to be tested before being marketed.

1. INTRODUCTION

Video Conferencing Systems (VCS) play a key role in the field of communication by offering means to organize high-quality face-to-face meetings without the need of gathering physically the participants. Cisco is the worldwide leader of VCS and sells many distinct products which are used by different clients [1]. VCS can be designed, developed and marketed very quickly and efficiently by means of Product Line Engineering (PLE), which is a very efficient method to create a large number of similar products based on a standardized process [2] [3]. Meanwhile, VCS are usually verified regarding to properties such as robustness and quality-of-service. Unlike free VCS that are available online, commercial VCS offer high-quality services for call establishment, communication control, connexion facilities, hardware systems compatibility and so on. To offer these, Cisco VCS products are thoroughly tested through manual test cases selection and execution procedures. However, facing the increasing complexity of functionalities and diversity of products, managing the selection and execution of test cases by hand becomes increasingly time-consuming and error-prone. Designing test plans and selecting test cases for testing new functionalities or new products requires increasing efforts [4] [5]. Meanwhile, inaccuracies and misunderstandings in the selection of relevant test cases are frequent because thorough domain knowledge and expertise from test engineers is continuously required.

To address the above-mentioned challenge, we propose to capture the commonalities and variabilities of both the VCS PL and the set of test cases that need to be executed through PLE techniques [6]. Like the variabilities of the VCS PL, there are also variabilities of the test cases applied to the same set of VCS products resulting from the PL. Different sets of test cases can be selected to execute different VCS products and a common structure is used to organize all test cases in Cisco. Modelling the structure of test cases in parallel with a clear view of the PL can alleviate the effort needed to select test cases for a given product [7]. On the one hand, using Feature Model (FM), which is a well-known representation of variability [8] [9], a set of distinct VCS products can be modelled by paying attention to commonalities and variabilities within the PL. On the other hand, using Component Family Model (CFM) [10] which is usually called Family Model, the overall structure of test cases that are attached to certain functionalities of the products can be captured. In this paper, we describe ongoing work that aims at modelling the Cisco VCS PL with these two models (FM and CFM) and linking these two models in order to support the automatic selection of test cases plans. This approach allows us to extract automatically the test cases that are associated with a given VCS product, to select the test cases for testing a given functionality and more generally to manage the overall testing strategy of the VCS PL. The most notable difficulty lies in the design of links between these models through the usage of restrictions, and to associate VCS products with test cases [11]. We implement these models in pure::variants, which is an excellent tool to capture variability within a PL [10]. The tool comes with a method to help designers to automatically derive

configurations of products through features selection. Using this approach, we associate automatically a set of test cases with a new VCS product represented by a combination of features, which makes test cases selection cheaper and less error-prone. Once a product variant is selected, no more manual effort is required to identify test cases related to test a given functionality. We envision a large gain in terms of man-power savings when a new product is issued and needs to be tested before marketed since selections of relevant test cases take too much time and a large number of man efforts in the Cisco VCS PL.

The remainder of the paper is organized as follows: Sec. 2 reviews the concept of FM and describes our modelling of the Cisco VCS PL through FM. Sec. 3 gives a thorough presentation of CFM. Sec. 4 explains how to make the connection between FM and CFM by using restrictions. Sec. 4 also explains how to select test cases automatically from a product variant selection. Finally, Sec. 5 concludes the paper and describes further work.

2. VIDEO CONFERENCING SYSTEMS PRODUCT LINE MODELLING

This section presents some background on Feature Models. It also introduces the FM we have built to capture and manage the variabilities of the Cisco VCS PL.

2.1 VCS Product Line

Product Line Engineering is a convenient way to design and develop Video Conferencing Systems, as these systems are highly parametric and can be configured differently for distinct clients [12]. To mention an example, when designed for the international market, a VCS product can be configured either in English or in other languages. Considering that VCS products are sold in more than fifty distinct countries, this parameter introduces a high degree of variability. Hopefully, configuration engineers do not need to configure VCS products for each new requirement and many common features can be captured efficiently through experience and documentation [13]. In addition to these configurability issues, there are also intrinsic common and variable features. For example, considering two distinct VCS products, namely the "C60" and "C90" in the PL, some common features exist such as supporting the same multisite dual stream functionality. However, there are also variable features such as supporting their own max resolution. The goal of PLE in the case of the VCS PL is to present explicitly what is common and what differs among the VCS PL. In our work and what follows, we propose to capture these common and variable features through the usage of a well-recognized model which is a Feature Model (FM) [8].

2.2 Feature Model (FM)

Feature modelling is a hierarchical modelling approach for capturing commonalities and variabilities in PL [14]. It can be used to document similarities and differences among a large number of products. Depending on interests of users or clients, a feature can be a requirement, a technical function, a configuration option and so on. Thanks to its hierarchical structure, complete configurations from a PL can be selected incrementally from a FM through feature selection [15]. Quoting White et al. [16], "*FM are arranged in a tree-like structure when each successively deeper level in the tree corresponds to a more fine-grained configuration option for the PL variant. Then parent-child and cross-tree relationships capture the constraints that must be adhered to when selecting a group of features for a variant*".

FM can be represented as $FM = \{features, relations, constraints\}$. It contains four different types of relations among features, namely *mandatory, alternative, optional* and *or* [17]. A *mandatory* relation between a father feature and a child feature specifies that, if the father feature is included in the current selection, then the child feature must also be included. An *alternative* relation among a father feature and a set of children features is used when the selection of only one of the children is required, not less, not more. An *optional* relation is used when the selection is optional. An *or* relation is used when any number of children features can be selected, but at least one. In addition, FMs can contain *cross-tree constraints* which are supplementary relations among unrelated features. There are two kinds of such constraints, namely the *require* and the *mutually exclusive* constraints. A *require* relation among two features (a source and a target) means that if the sink feature is included into the current selection, then the targeted feature must also be included. A *mutually exclusive* relation among two features has the opposite meaning, saying that if one is included then the other one cannot be included as well.

FMs are used in a large number of activities, including domain engineering, application engineering, or variability management [18]. They implicitly contain a great deal of useful information such as the set of valid products of a PL, the number of valid products, the number of variation points within a PL, and so on. In our work, we designed the Cisco VCS PL using a Feature Model.

2.3 A Feature Model for representing the VCS Product Line

The Cisco VCS PL is described by documents including synthetic excel sheets, where each line represents a valid product and each column shows a feature. Fig. 1 shows an excerpt of such an excel sheet. In the document, we can see that each product supports similar and different functionalities. For example, the VCS product "Asterix" and "C20" both share video and audio calls, but just "Asterix" features the multisite call functionality, while "C20" does not.

Endpoint Product line							
Product	Software features						
	Calls (video+audio)	Multisite Call option	Presenter option	High definition option (720p30)	Premium resolution option (720p60/1080p30 and above)	Dual Display option	1080p60 p2p
Asterix	1+1	3+1 (3:720p30)	y	x	y	y	n
C20	1+1	n	y	y	y	y	n
MX200 Intrepid	1+1	n	x	x	y	n	n
EX 60 Pluto	1+1	n	x	x	y	n	n
EX90 Falcon	1	3+1 (3:720p30)	x	x	y	n	n
C40	1	3+1 (3:1024x576)	y	x	y	x	n
C60	1	3+1 (3:1280x720)	y	x	y	x	n
C60-2	1	3+1 (3:720p30)	y	x	y	x	x
C90	1	3+1 (3:1080p30/720p60)	y	x	y	x	n
C90-2	1	3+1 (3:1080p30/720p60)	y	x	y	x	x
SX20	1	3+1 (3:1080p15/720p60)	y	x	y	x	x
	n=not avail						
	x=default						
	y=optional						

Figure 1: VCS PL description

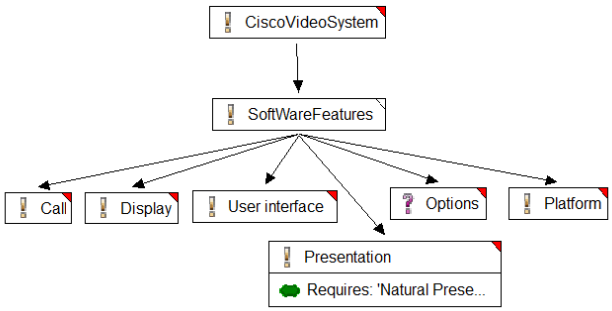


Figure 2: High level graphical structure of VCS PL

It is now well-accepted that excel sheets are not the best manner to capture variabilities and commonalities within a PL, since there is no hierarchical structure among features within an excel sheet and the matrix representation of the PL is clearly insufficient when users want to reason about the overall PL and add new products/features [8].

In our approach, based on the excel document and the commercial descriptions of products(which can be downloaded from: www.cisco.com), we propose to represent the Cisco VCS PL with a Feature Model. In ongoing work, we started with the modelling of eleven distinct VCS products (i.e., Asterix, C20, MX200 Intrepid, Ex60 Pluto, Ex90 Falcon, C40, C60, C60-2, C90, C90-2, SX20). Our VCS FM has a hierarchical tree structure in which features are represented and organized as nodes of the tree and relations among features can be classified as mandatory, or, optional and alternative. The variants of the VCS PL can be described by performing different selections on the features of the FM. Fig. 2 shows the highest level of the FM where six essential VCS features are represented, namely "Call", "Display", "User Interface", "Presentation", "Options" and "Platform", while Fig. 3 shows the part of the FM corresponding to the "Presentation" feature(Exclamation marks represent "mandatory" features, question marks show "optional" features). For instance, there is a mandatory relation between the "Call" feature and the "SoftWareFeatures" feature because any VCS product must contain this functionality. On the contrary, there is an optional relation between the "Options" feature and the "SoftWareFeatures" feature because a VCS product may choose to contain some options according to different individual preferences of requirements. In general, our case has 69 features in all, which can represent distinct products by different combinations of features.

Based on the textual description of the PL, we also add cross-tree relations into the VCS FM. For example, as shown in Fig. 3, "require" relations among the "WXGAp30" feature under "DualStreamP2P", the "C20" and "C40" under "Platform" are added since only the platforms for "C20" and "C40" support resolution up to WXGAp30. These cross-tree relations are not easy to determine as they require the expertise of engineers in charge of the design of these products, but they can be useful to represent VCS products more accurately in a practical manner. The modelling part of the PL is thus currently undertaken with the help of Cisco PL experts. Note also that cross-tree relations are a convenient means to limit and control the number of potential products represented by the FM. Indeed, the more cross-tree constraints, the smaller the set of valid products that is represented by the FM. At this stage, we have built a FM to capture the variabilities of the Cisco VCS PL. Our goal is now to relate this FM to the large number of existing test cases that have been designed in the past to test VCS products.

3. A FAMILY MODEL TO CAPTURE TEST EXECUTION PLANS

In this section, we briefly introduce Family Models (CFM) and present our CFM that models the large structured set of test cases for VCS products.

3.1 VCS Product Line Testing

VCS Testing is currently performed through manual test cases selection and planning. If a product needs to be tested, engineers have to select relevant test cases based on the product for getting a test execution plan. However, as stated in the introduction, manually extracting a large number of test cases is a tedious and error-prone process [19]. As a result, some test cases can be erroneously selected leading to conflicts during test cases execution. These conflicts have then to be solved manually, entailing a significant waste of efforts [20]. In Cisco's structure of test cases, there are several test suites. A test suite is a collection of test tasks that belong logically together such as "Audio", "Video" and "Web" for a better overview. A test task is a collection of test jobs that has a common test resource requirement such as "Image Layouts" and "Video controller functionality" under "Video". Each test job is a test case (a parameterized test script) which can be run on applicable platforms. It

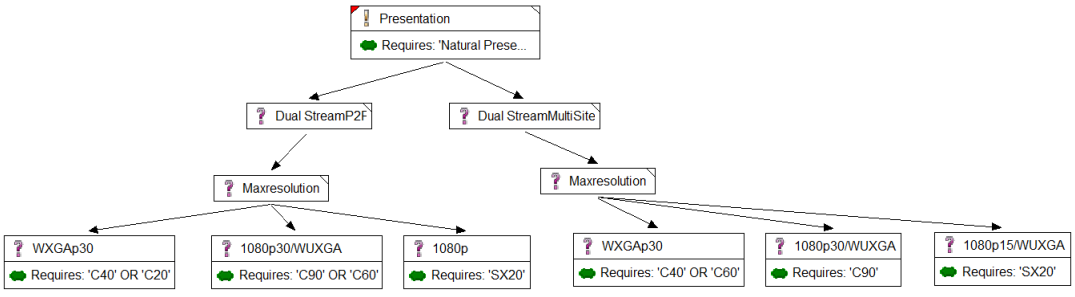


Figure 3: Graphical structure of the feature "Presentation"

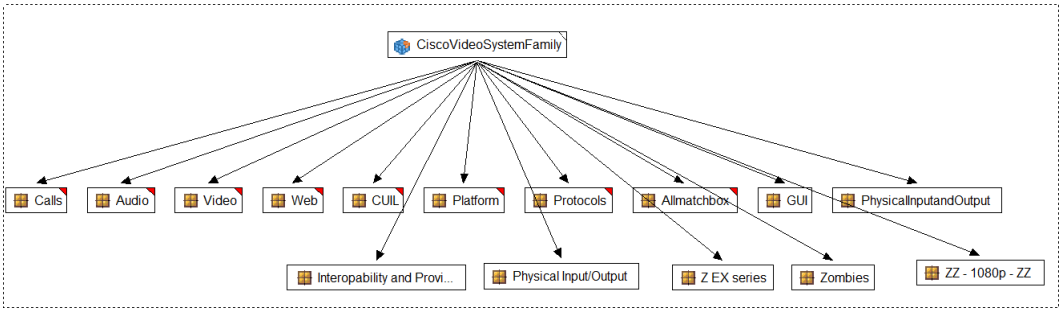


Figure 4: High level graphical structure of CFM

is worth noticing that test plans are usually composed of many test cases and test engineers spend significant time organizing test cases within these plans. In order to model the variabilities across numerous test cases and to automatically obtain test execution plans, we propose to capture the overall structure of test cases with a Family Model.

3.2 Component Family Model (CFM)

CFM can be used to represent how products are assembled and generated in a PL by modelling relations among software architectural elements [21]. It has a hierarchical structure including items such as components and parts. For the purpose of automatic product generation from a valid selected feature model in PL, these items can be organized and used with relevant information about the concrete architecture.

CFM can be represented as $CFM = \{components,\ parts,\ source\ elements,\ restrictions\}$. *Components* are named entities and organized into a tree-like structure that can be of any depth. Each component represents one or more functional elements of the products in PL such as functions of software or documentation. *Parts* are named and typed entities. Each part belongs to a component and contains one or more source elements. A part can be associated with given programming language features, classes or objects, but it can also be associated with other key elements. A *source element* is an unnamed but typed entity. Source elements are usually used to determine how the source code for the specified element is generated [21]. *Restrictions* play a key role for linking FM and CFM. A *restriction* constrains the relationships between an element in CFM and features in FM. They are added into CFM in order to decide whether an element can be part of a product in PL. An element in CFM can not be associated with a product unless restrictions defined on the element evaluate to true.

3.3 A CFM to capture the structure of test cases

In our modelling, CFMs are not used to represent software architecture, but rather to model the hierarchical structure of test cases. In our CFM, a component represents a test suite or a test task which can be regarded as a named set of test cases and it is hierarchically decomposed into further components or into part elements. A part represents a test case which belongs to a test task. Fig. 4 represents the highest level of the CFM where each component is represented by its name, while Fig. 5 represents the subtree of the

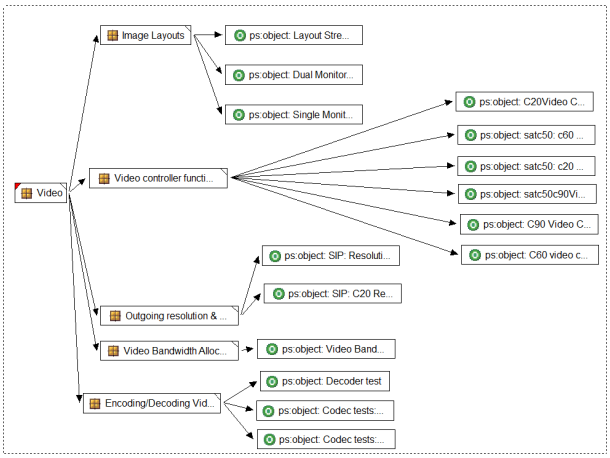


Figure 5: Graphical structure of "Video"

"Video" component. These names have usually been selected according to the functionality of the VCS to be tested.

Physical elements such as test scripts that contain several test cases within test execution plans could also be modelled through source elements in CFM. However, in our ongoing work, we focus on the association of the VCS PL and the test cases. Hence, our CFM contains only components and parts which can be linked with FM, but no source elements, meaning that the automatic selection of test cases is currently possible, but not automatic execution of obtained test plans. Another important characteristic of CFM is the notion of restrictions. A typical restriction over a CFM is *hasfeature()* stating that a component or a part is part of the resulting selection iff the relevant feature is included in the set of features selection. Besides the *hasfeature()* restriction, some more complex restrictions can also be defined using the Prolog programming language for the purpose of linking FM and CFM more accurately in a practical manner. Thus more complex links between these two models can be made to perform the selection process of test cases in a more systematic way. In our approach, we use the *hasfeature()* restriction to associate features with test cases, meaning that each test case is linked with the corresponding features of the FM of the VCS PL. Using restrictions allows us to establish links between the test cases structure and the VCS PL. Now we detail the implementation of our overall approach.

4. IMPLEMENTATION

In this section, we suppose that both a FM for the Cisco VCS PL and a CFM for representing the test cases structure have been built. We propose to establish links between these two models and present an example of automatic selection of test cases.

Our approach is implemented within pure::variants, a software modelling tool that allows users to design FM, CFM and restrictions [21]. In pure::variants, an important advantage that makes CFM powerful is its support of flexible rules for the inclusion of components as well as parts, meaning we can use various rules to determine whether a component or a part can be included into a product. This is achieved by placing various restrictions on these elements. Each element can have any number of restrictions. A component or part is contained in the resulting configuration iff its parent is included and its restrictions evaluate to true or there are no restrictions on it [21]. Therefore, based on these rules, we can link our FM of the VCS PL and CFM of the structuring test cases.

In our approach, *hasfeature()* is used to link the VCS FM and the CFM of test cases. For example, we assign the restriction *hasfeature ('Dual StreamMultiSite')* to the part "MBT multisite - h323" under the component "Multisite presentation" because the test case "MBT multisite - h323" is related with test functionality "Dual StreamMultiSite". It means that during the automatic selection of test cases, the test case "MBT multisite - h323" can be included into the test cases plan iff the feature "Dual StreamMultiSite" is in the selection set of features. According to thorough discussions with Cisco test engineers, we added many relevant restrictions into our family model. For example, we added the restriction *hasfeature('Dual StreamMultiSite')* to each part under the "Multisite presentation" component.

Based on our FM and CFM models built in pure::variants, relevant test cases can be selected automatically when a new product needs to be tested. The configuration process is shown in Fig. 6. Test engineers just need to choose the set of features they want to test on the FM, then, after the selection of relevant features, test cases can also be obtained automatically. The selection description of features and corresponding test cases can be exported in XML and HTML file formats. These description documents can then be modified or improved in order to be used in the process of test execution. However, in our current work, we do not yet exploit these exported files to configure the test execution process.

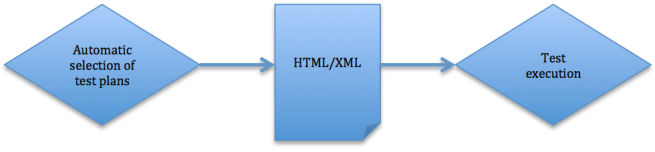
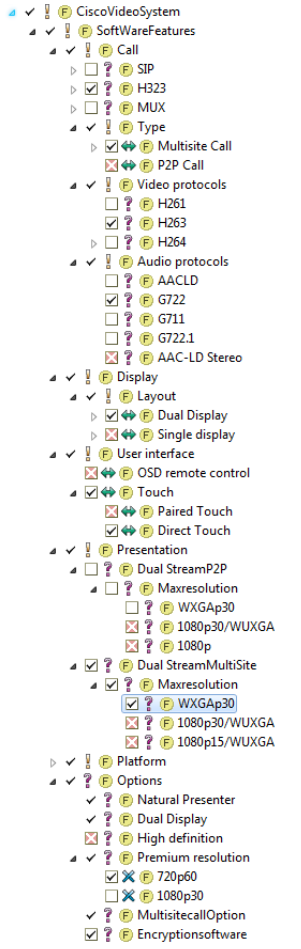


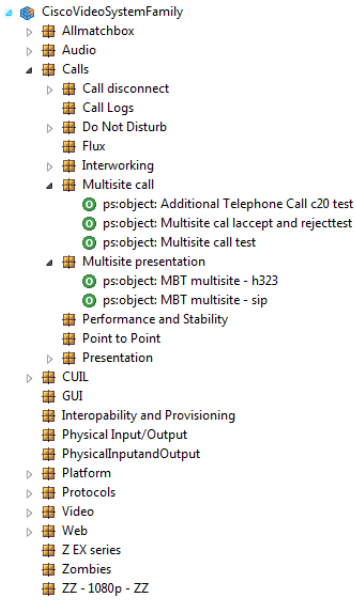
Figure 6: Process of automatic selection and test execution

4.1 A configuration scenario

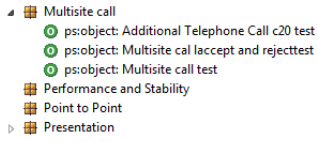
We now describe a scenario where a new VCS product comes into play, and needs to be tested. Firstly, test engineers need to select the set of features corresponding to the new product in the FM, as shown in Fig. 7 (a). Using pure::variants is advantageous in this process as selection conflicts, such as those due to mutual and require cross-tree constraints, are automatically detected and resolved. For example, as shown in Fig. 7 (a), if the feature "Multisite Call" under the feature "Type" of "Call" is selected, then the features "P2P Call" is automatically tagged as impossible to select. Similarly, the selection of "Multisite Call" leads to the automatic selection of the feature "MultisitecallOption" under the feature "Options" because there is a "require" relation between them.



(a) Features selection of a VCS product



(b) Automatic test cases selection



(c) Test cases related with multisite presentation are removed

Figure 7: A configuration scenario

Automatic feature selection leads to the automatic selection of test cases, as shown in Fig. 7 (b). Because the feature "Dual StreamMultisite" is included in the selection, test cases related to multisite presentation are also included in the test case selection. However, if the feature "Dual StreamMultisite" is removed from the selected set of features, test

cases relevant to multisite presentation are then removed, as shown in Fig. 7 (c). Therefore, automatic selection of test cases can be achieved using the FM and CFM we designed for the Cisco VCS PL. When a VCS product needs to be tested, corresponding test cases can be automatically selected by selecting relevant sets of features on the FM.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we propose an alternative approach to complete the process of automatic selection of test cases to test the products of a product line by using Feature Models (FM) and Family Models (CFM). FM captures the variabilities of the PL and CFM manages the variability within the test cases structure. Using restrictions, a simple feature selection over the FM leads to the automatic selection of test cases for a given product. We are applying this approach to the modelling and testing of a complete Video Conferencing System Product Line. The paper presents the first steps of this ongoing work. Using this approach, we envision a large gain in terms of effort savings, as automatic selection and configuration of test cases is possible and allows us to avoid this costly and error-prone process.

However, some open research questions still need to be addressed in order to show that the approach is fully viable:

1. How to evaluate the benefits of this automatic test cases selection process? We believe that the approach presented in this paper is particularly useful to obtain test cases when new products are issued as it should save significant efforts by avoiding the costly process of manual test case selection. To evaluate precisely these benefits, a thorough experimental study is required;

2. How to capture the test cases structure through an improved CFM? In our modelling, we use only the *has-feature()* restriction, that might not be sufficient for automatic selection of test cases plans. Other restrictions could be advantageously exploited and logical combined in order to link FM and CFM more accurately and reasonably. We should also mention that using attributes of FM and CFM could be interesting to parameterize the testing process, by considering explicit links with test scripts and parameters because a test script with different parameters can represent different test cases in VCS product line;

3. How to utilize a configuration tool within a well software validation process? Utilizing a standard tool like pure::variants within a software validation process is still questionable as the methodology for using our approach within a complete process is still an ongoing work. To start, we envision to exploit the results of our approach within a test configuration tool that is used at Cisco to drive the test execution process. A related question concerns the integration of this improved configuration tool into the process that is used for the testing of the VCS PL.

6. REFERENCES

[1] Cisco Systems. Cisco telepresence codec c90. data sheet. available from http://www.cisco.com. 2010.

[2] David Benavides Cuevas. *On the Automated Analysis of Software Product Lines Using Feature Models*. PhD thesis, Universidad de Sevilla, 2007.

[3] S. Oster; F. Markert; P. Ritter. Automated incremental pairwise testing of software product lines. In *Software Product Line Conference(SPLC'10),2010*.

[4] D.R. Kuhn; D.R. Wallace; A.M. Gallo. Software fault interactions and implications for software testing. *IEEE Trans. Softw. Eng.*, 30(6):418–421, 2004.

[5] A. Tevanlinna; J. Taina; R. Kauppinen. Product family testing: a survey. *SIGSOFT Softw. Eng. Notes*, 29(2):12, 2004.

[6] D.M. Cohen; S.R. Dalal; M.L. Fredman; G.C. Patton. The aetg system: an approach to testing based on combinatorial design. *Software Engineering, IEEE Transactions on*, 23:437–444, July 1997.

[7] E. Uzuncaova; D. Garcia; S. Khurshid; D.batory. Testing software product lines using incremental test generations. *ISSRE. IEEE Computer Society*, pages 249–258, 2008.

[8] D. Benavides; S. Segura; A. R. Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, (35):615–636, 2010.

[9] A. Metzger; K. Pohl; P. Heymans; P.Y. Schobbens; G. Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. *In Requirements Engineering Conference*, pages 243–253, 2007.

[10] Pure systems GmbH. Variant management with pure::variants. technical white paper. available from http://web.pure-systems.com. 2006.

[11] K. Czarnecki; M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. *Generative Programming and Component Engineering (GPCE)*, 3676:422–437, September 2005.

[12] R. Rabiser; P. Grunbacher; D. Dhungana. Supporting product derivation by adapting and augmenting variability models. *In Software Product Line Conference*, (141-150), September 2007.

[13] J. White; B. Doughtery; D. Schmidt; D. Benavides. Automated reasoning for multi-step software product-line configuration problems. *In Software Product Line Conference*, pages 11–20, 2009.

[14] K. Czarnecki; S. Helsen; U.W. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process : Improvement and Practice*, 10(2):143–169, June 2005.

[15] D. Benavides; A. Ruiz-Cortés; P. Trinidad; S. Segura. A survey on the automated analyses of feature models. *In Jornadas de Ingenieréa del Software y Bases de Datos (JISBD)*, pages 367–376, 2006.

[16] J. White; D. Benavides; D. C. Schmidt; P. Trinidad; B. Dougherty; A. Ruiz-Cortés. Automated diagnosis of feature model configurations. *Joural of Systems and Software*, 83(7):1094–1107, 2010.

[17] K. Czarnecki; C. Kim; K. Kalleberg. Feature models are views on ontologies. *In proceedings of the 10th International Software Product Line Conference (SPLC)*, pages 41–51, August 2006.

[18] D. Beuche; H. Papajewski; W. Schröder-Preikschat. Variability management with feature models. *Science of Computer Programming*, 53(3):333–352, 2004.

[19] M.B. Cohen; M.B. Dwyer; J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engineering*, 34(5):633–650, July 2008.

[20] G. Perrouin; S. Sen; J. Klein; B. Baudry; Y. le Traon. Automated and scalable t-wise test case generation strategies for software product lines. *Software Testing, Verification and Validation (ICST)*, pages 459–468, 2010.

[21] Pure systems GmbH. pure::variant user's guide. available from http://web.pure-systems.com. 2011.