Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering

– Proceedings, Nashville, TN, October 23, 2008 –

Neil Loughran[1], Iris Groher[2], Roberto Lopez-Herrejon[3], Sven Apel[4], and Christa Schwanninger[5]

[1] Computing Department, Lancaster University, UK
loughran@comp.lancs.ac.uk

[2] Inst. for Systems Engineering and Automation, University of Linz, Austria
Iris.Groher@jku.at

[3] Computing Laboratory, University of Oxford, UK
Roberto.Lopez@comlab.ox.ac.uk

[4] Department of Informatics and Mathematics, University of Passau, Germany
apel@uni-passau.de

[5] Siemens Corporate Technology, Siemens AG, Germany
Christa.Schwanninger@siemens.com

Technical Report, Number MIP-0804
Department of Informatics and Mathematics
University of Passau, Germany
October 2008

Preface

Product Line Engineering (PLE) is an increasingly important paradigm in software development whereby commonalities and variations among similar systems are systematically identified and exploited. PLE covers a large spectrum of activities, from domain analysis to product validation and testing. Variability is manifested throughout this spectrum in artifacts such as requirements, models, code and documentation and it is often of crosscutting nature. These characteristics promote different kinds of modularization and composition techniques (e.g., objects, components, aspects, features, subjects, frames, etc.) as suitable candidates to manage variability. Prior work on Generative Programming (GP) and Component Engineering (CE) has shown their successful applicability to PLE and the potential benefits of modularization and composition techniques.

The Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering (McGPLE) aims at expanding and capitalizing on the increasing interest of researchers from these communities. It builds on the success of the Aspect-oriented Product Line (AOPLE) workshop which has run consecutively at GPCE for the past two years. AOPLE established an initial community and formulated a first joint research agenda. The main goal of the workshop is to broaden this agenda and strengthen the established collaborations, to share and discussed ideas, identify research opportunities and foster collaboration to tackle the challenges these opportunities may bring about.

Goals

Work on software modularization and composition concepts and techniques when applied to PLE has shown promising results. These results can be further strengthened when GGP and CE techniques are applied in concert. The main goal of the workshop is to foster and strengthen the collaboration between the different software composition and modularization techniques, PLE and generative research communities by identifying common interests and research venues. The new workshop builds on the success of the AOPLE workshops that established an initial community of researchers, but focuses on a broader range of issues, techniques and approaches.

Program Committee

- Jeff Gray, University of Alabama at Birmingham
- Vander Alves, Fraunhofer IESE
- Rob van Ommering, Philips Research
- Andreas Rummler, SAP
- Don Batory, University of Texas at Austin

October 2008, The McGPLE'08 organizers.

Content

Towards Reuse with "Feature-Oriented Event-B"

Michael Poppleton, Bernd Fischer, Chris Franklin, Ali Gondal, Colin Snook, Jennifer Sorge
Dependable Systems and Software Engineering
University of Southampton
Southampton, SO17 1BJ, UK
{mrp, b.fischer, aag07r, cf105, cfs, jhs06r}@ecs.soton.ac.uk

Abstract

Event-B [19] is a language for the formal specification and verification of reactive systems. The language and its RODIN toolkit represent a leading model-based technology for formal software construction. However, scalability is a major current concern, especially the dimension of reusability. We outline a proposed infrastructure for scalable development with reuse for Event-B. We focus specifically on our agenda for reuse in Software Product Lines, and explain how a form of feature modelling will be central to this programme.

1 Introduction

Event-B [19] is a formal modelling language that evolved naturally from the classical B language [1] of J.-R. Abrial. The recent project RODIN[1] saw the definition of Event-B and the creation of the rich Eclipse-based [14] RODIN toolkit [2] for formal modelling, animation, verification, and proof with Event-B. This includes a project repository, syntax- and type-checkers, proof obligation generator, animators, theorem provers, and various front-end plug-ins.

In software development with Event-B, *refinement* is the central method by which initially small, abstract models of requirements are elaborated through architectural and detailed design to code. Refinement[2] M1 of a model M0 will usually remove some nondeterminism (implementation-freedom) and add data and algorithmic structure. M1 is mathematically *proved* to be a "black-box" simulation of M0, i.e. to offer only behaviour specified by M0. Event-B allows us to formally state and prove both consistency properties for models and refinement properties between them; we call these properties *proof obligations* (POs) in Event-B. These capabilities are part of the extra "bang for the buck" that Formal Methods offer to critical systems developers.

While there is now growing evidence of successful industrial critical systems development using B technology, e.g. [13, 18], only limited (and commercially protected) tool support exists to scale up to large applications. Project DEPLOY[3] aims to address this by scaling methodology in requirements validation, requirements evolution, reuse, and resilience, and scaling tooling in simulation, analysis and verification of formal models. This paper adds "feature-oriented Event-B" to that agenda.

Modularization and structuring are key issues in scaling Event-B models: a number of model decomposition mechanisms [3, 16, 21] have been proposed, and tool support for them is under development. The *event fusion* of [21] is designed specifically for feature-oriented structuring with Event-B.

The authors are working towards defining a method for feature-based modelling with Event-B, specifically aimed at reuse in software product lines (SPLs). Feature modelling [12] is a well understood approach for variability modelling for SPLs. To date it has mostly been applied to code or detailed design documents; we apply it to an abstract, nondeterministic language with formal semantics and verification conditions (POs). This paper outlines definitions of features as generic Event-B model elements, and defines feature composition and specialization. Using a simple example we present a scheme for precise definition of product line instances as particular feature compositions. This suggests a graphical feature modelling notation in the usual style, but with rigorous semantic foundations.

We present an agenda for methodological and tool development to support feature-oriented software development with Event-B. While the detailed feature structuring ideas

[1] RODIN - Rigorous Open Development Environment for Open Systems: EU IST Project IST-511599

[2] The term *refinement* is overloaded, meaning (i) the process of transforming one model into another, and (ii) the concrete model which refines the abstract one.

[3] DEPLOY - Industrial deployment of system engineering methods providing high dependability and productivity (2008 - 2011): FP VII Project 214158 under Strategic Objective IST-2007.1.2. Further information and downloadable tools are available at http://www.deploy-project.eu/

of this paper are syntactic, we emphasise that this is scaffolding for the real, semantic value that we anticipate from this work: the scaling of verification through a product line. The starting point is designing an identified feature and its chain of refinements, and then proving each of these refinements consistent, and a correct refinement transformation of its predecessor. This is part of the legwork of feature construction for the application domain.

The theoretical job is then as far as possible to prove compositionality, i.e. that consistency and refinement are preserved when we compose features. For any consistent features f and g we must prove $f \oplus g$ consistent (for a defined composition operator "$\oplus$"). Given their refinements f_1 and g_1, we must further prove $f_1 \oplus g_1$ both consistent and a correct refinement of $f \oplus g$. While such compositionality has been proved for the operators of [3, 16, 21], much work remains for the intricate needs of feature composition.

2 The Event-B language

Event-B is designed for long-running *reactive* hardware/software systems that respond to stimuli from user and/or environment. The set-theoretic language in first-order logic (FOL) takes as its semantic model a transition system with guarded transitions between states. The correctness of a model is defined by an invariant property, i.e. a predicate, or constraint, which every reachable state in the system must satisfy. More practically, every event in the system must be shown to preserve this invariant; this verification requirement is expressed in a number of proof obligations (POs). In practice this verification is performed either by model checking or theorem proving (or both).

In Event-B the top level unit of modularization is the *model* consisting of a *machine* and zero or more *contexts*. The dynamic machine contains state variables, the state invariant, and the events that update the state. The static *context* contains sets, constants and their axioms.

The unit of behaviour is the *event*. An event E acting on (a list of) state variables v, subject to enabling condition, or *guard* predicate $G(v)$ and *action*, or assignment $R(v)$, has syntax

$$\mathrm{E} \mathrel{\widehat{=}} \text{ WHEN } G(v) \text{ THEN } R(v) \text{ END} \qquad (1)$$

That is, the action defined by $R(v)$ may occur only when the state enables the guard. An event E works in a model with constants c and sets s subject to *axioms* (properties) $P(s,c)$ and an *invariant* $I(s,c,v)$. Thus the event guard G and assignment with before-after predicate[4] $\hat{R}$ take s,c as parameters. Two of the consistency proof obligations (POs) for event E defined as (1) are FIS (feasibility) and INV (invariant preservation):

$$P(s,c) \wedge I(s,c,v) \wedge G(s,c,v) \Rightarrow \exists v' \bullet \hat{R}(s,c,v,v') \qquad (2)$$

$$P \wedge I \wedge G(s,c,v) \wedge \hat{R}(s,c,v,v') \Rightarrow I(s,c,v') \qquad (3)$$

Intuitively speaking, the static typing and axioms P and state consistency property I give the known properties of the system at any time. For event E (1), FIS states that when its guard G is true (enabled) at state v, then E - via its before-after predicate $\hat{R}$ - is able to make the state transition from v to v'. INV states that E will maintain the invariant, i.e. consistency: when G is enabled, any after-state v' reachable by E will satisfy invariant I.

In order to progress towards implementation, the process of *refinement* is used. A refinement is a (usually) more elaborate model than its predecessor, in an eventual *chain* of refinements to code.

The refinement of a context is simply its elaboration, by the addition of new sets, constants and axioms. When refining a machine, new variables may be added, and some or all abstract (refined) variables v may be replaced by new concrete (refining) ones w. New invariant clauses and events will usually be added, elaborating data and algorithmic structure. There are proof obligations for refinement, both for correctness of the simulation of an abstract model by its more concrete refinement, and for preservation of certain liveness properties. We do not discuss these further.

3 Feature-oriented Event-B ?

A small case study from project DEPLOY will be used to demonstrate the prototype scheme for product-line development with Event-B, based on formal feature modelling. The example consists of specifications and Event-B developments for two simple, related products: a switch and a pushbutton.[5] Switch and pushbutton each have a single two-valued output, off (false) or on (true). Each has one continuous input in the interval $[0,1]$. Rising and falling thresholds are used on the sampled input to determine switching conditions.

Neither specifications nor models have been developed through any product-line process: commonalities in the Event-B models were cut-and-pasted, and variabilities were modelled in situ for each model instance. Again, requirements features have simply been identified by intuition, rather than by any defined process.

A precise syntactic definition of an Event-B feature remains to be established after case study experience; for the present we regard the feature as a well-formed machine, context or model, and a subfeature as a well-formed element of such, e.g. a variable + typing invariant, a constant + typing axiom, an event. Expressiveness is required in the

[4] Here $R(v)$ is a syntax of actions, corresponding to a before-after predicate $\hat{R}(v,v')$.

[5] A 3-way and an n-way switch are also part of the product line, but have not been included for the sake of brevity.

Event-B feature definition to allow easy correspondence between requirements and model features.

The switch is specified as follows, paraphrasing [22], and adding named features. Note that all features here map to Event-B machine (i.e. behavioural) features, except for *bounce, threshr, threshf*, which are context features. The switch has four parameters (Debounce time BT, Rising threshold RTH, Falling threshold FTH, Cycle time CT). The input will be read cyclically with Cycle time CT.

1-1 Initially, the output is "off" (*initop*).

1-2i If the output is "off" and the switch on condition is true, the output is set to "on" (*switchopi*).

1-2ii If the output is "on" and the switch off condition is true, the output is set to "off" (*switchopii*).

1-3 A rising edge is detected if at time t the input is higher than RTH and at time t-CT it was lower than RTH (*edge*).

1-4 A falling edge is detected if at time t the input is lower than FTH and at time t-CT it was higher than FTH (*edge*).

1-5i The switch on condition is true if a rising edge was detected and the input exceeds RTH for BT after the rising edge (*swcond*).

1-5ii The switch off condition is true if a falling edge was detected and the input is lower than FTH for BT after the falling edge (*swcond*).

1-6 BT $>$ CT (*bounce*)

1-7i $0 <$ RTH $<= 1$ (*threshr*)

1-7ii $0 <=$ FTH < 1 (*threshf*)

The requirements are now expressed in terms of the composition of features, e.g. including two variants of *swcond*. Such variation is achieved by making features specializable by parameter. The pushbutton differs from the switch in that it uses a single switch condition based on a rising threshold.

The specification suggests a graphical notation for the Event-B feature model - shown in Fig. 1 - comprising a machine graph and a context graph. This notation will build on some version of standard feature modelling notation [12]. We add two kinds of edge: "c" for "consists of", as per standard notation, and "r" for "refines" (we make the distinction because of the verification POs denoted by a "refines" edge). As usual, black or white dots denote mandatory or optional features.

Thus a switch device comprises machine features *initop, switchopi, switchopii* and context features *bounce, threshr, threshf*, in an abstract (level 0) model. The symbol $\oplus$ denotes various feature compositions outlined below. *switchopi*, the switch-on feature, is enabled when the switch is off. It is a machine feature comprising variable *output*, its typing invariant, initialisation, and event

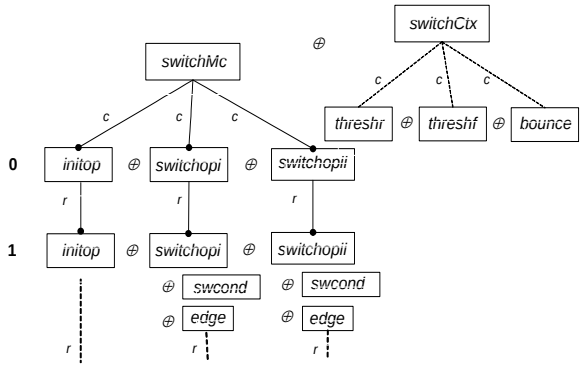


Figure 1. Switch feature diagram

out_F_T =
WHEN grd1: *output* = false
THEN act1: *output* := true
END

switchopii is the reverse, switch off feature. Since at level 0 the switch condition is abstracted away, the switch and pushbutton models are identical at level 0.

At refinement level 1 abstractions of the switch condition *swcond* and the rising/falling edge test *edge* are introduced. A counter variable c is introduced and initialised to zero. When a rising edge is detected on the input by *edge*, c is set to $n = BT/CT$. c is decremented (by *swcond*) in each cycle that the input remains high, and the switch condition is satisfied when n reaches 1.

For each product line instance, its composition from constituent features, and these features' specialization (parameterization) must be explicitly defined at each refinement level. We show refinement level 1 for the switch:

$$
\begin{aligned}
&Switch \mathrel{\widehat{=}} SwitchCtx \oplus_{scmc} SwitchMc\\
&SwitchCtx \mathrel{\widehat{=}} (threshr \oplus_{sccc} threshf) \oplus_{scac}\\
&\qquad\qquad Axiom(\text{“}fth{<}rth\text{”})\\
&SwitchMc \mathrel{\widehat{=}} initop \oplus_{smmc}\\
&\quad (switchopi \oplus_{smmc}\\
&\qquad swcond(lbl = \text{“}re\text{”}, grd = \text{“}output = \text{false”}) \oplus_{smmc}\\
&\qquad edge(lbl = \text{“}re\text{”}, grd = \text{“}output = \text{false”})\\
&\quad) \oplus_{smmc}\\
&\quad (switchopii \oplus_{smmc}\\
&\qquad swcond(lbl = \text{“}fe\text{”}, grd = \text{“}output = \text{true”}) \oplus_{smmc}\\
&\qquad edge(lbl = \text{“}fe\text{”}, grd = \text{“}output = \text{true”}))
\end{aligned}
$$

Next we elaborate the various composition operators $\oplus_{slrp}$ by describing these four modifiers:

- s: Strength of composition: s(default) denotes "strong", i.e. the two composing elements must be syntax- and type-consistent and must not require any user specialization. "w" denotes "weak", i.e. allowing

user specialization and resolution of inconsistencies at composition/instantiation time.

- *lr*: Syntactic kind of left *l* and right *r* elements being composed: these may be feature elements, i.e. *m*(default) for machine, *c* for context, and *b* for model, in any combination. Further, a feature may be composed with a subfeature of appropriate kind, i.e. machine *m* with variable(s) *v*, invariant *i*, event(s) *e*, or context *c* with constant(s) *o*, carrier set(s) *r*, axiom(s) *a*. A model *b* may compose with a consistent subfeature of any kind.
- *p*: Composition of predicates: whether to conjoin *c*(default) or disjoin *d* predicates, i.e. when combining invariant clauses, or adding guard clauses to an event, or fusing events.

For the switch instance *Switch*, context feature *SwitchCtx* is composed from context features for each of the rising (*threshr*) and falling (*threshf*) thresholds, as well as an extra axiom relating the two. All machine compositions are simply $\oplus_{smmc}$. For an example of specialization consider the single event in *edge* denoting threshold detection:

%lbl%_F_T =
WHEN %grd%:
grd1: $c = 0$
THEN act1: $c := n$
END

This skeleton feature requires a label and a guard to be completed. In the above definition of *Switch*, *edge* is instantiated twice, once for the rising edge (lbl="re", grd="*output* = false"), resulting in event re_F_T and once for the falling edge, resulting in event fe_F_T. Thus, a rising edge can only be detected when output is off, and conversely for the falling edge instantiation.

Considering the second instance in our little SPL, the pushbutton differs from the switch precisely in that it uses a single switch condition *swcond* based on a rising threshold. Thus refinement 1 for *Pushbutton* differs from *Switch* in that (i) in *swcond, edge* for *switchopii*, the label parameters become "re", (ii) the outer composition between the *switchopi* and *switchopii* is $\oplus_{smmd}$, and (iii) *PushbuttonCtx* is simply *threshr*. In this case the two instantiations of *edge* in the same instance produce two versions of the same event re_F_T. Hence the two versions of event re_F_T must be *fused* [21]. That is, duplicated guards and actions are ignored, and the *d*-modifier on $\oplus_{smmd}$ specifies that the extra guard clause be disjoined. This gives a guard of *output* = false $\vee$ *output* = true, which should be preprocessed to true during instantiation, giving an always-enabled switch condition.

4 Tooling for feature modelling

Our tool development takes place in support of some future feature modelling process for Event-B. A feature modelling phase, during domain analysis, will develop a feature model based on any existing feature database, at the same time developing new features. This will include feature consistency proof, refinement and verification, as far as possible: the question of exactly how much verification can be done on an unspecialized feature remains open. Athough an event like %lbl%_F_T can be interpreted as well-formed Event-B, and be consistency-verified, in general this will not be true. Further, [20] described how in general a feature, not containing all behaviour affecting its variables, will fail to verify liveness POs.

An instance modelling phase will follow where system instance specifications will be developed in the style of the *Switch*. Most probably there will be iterative feedback to the feature modelling phase. Finally, an instance production phase will follow.

The starting point in tooling was the construction of an Eclipse Modeling Framework [11] (EMF) editor for Event-B, based on a language metamodel produced in DEPLOY. EMF, based on metamodelling in the UML sense, enables quick construction of a simple editor with a tree-structured user interface reflecting the metamodel structure. A composition metamodel was developed by inheritance from the Event-B metamodel, to define a small number of prototype feature compositions. A prototype EMF feature composition editor (comp-editor) was then produced based on the composition metamodel. This enables recording of the composition and specialization parameters in a particular composition instance.

The comp-editor shown in Fig. 2 allows the user to specify all composition and specialization parameters interactively. Its interactive style will be useful during the early feature and instance modelling activities. In the figure, on

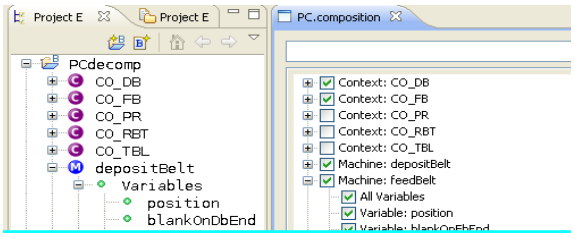


Figure 2. Composition tool

the left is a RODIN project explorer panel showing various project elements. When a project is clicked on, the comp-editor panel on the right is opened and blank. A drop-down menu allows selection of elements from the selected project, which produce corresponding tickboxes in the the comp-editor panel; in the figure we see identified context, machine and variable features. The user then instantiates the composition by ticking required features. On clicking "Compose", a third panel opens, allowing user specialization of selected

features and resolution of any conflicts. Dependency analysis is provided; e.g. given a variable the tool will identify all (sub-)features requiring that variable.

The next step for comp-editor is support for the automated composition variants "$\oplus_{slrp}$", which is a matter of suitably packaging existing functionality. Such automated compositions will be required for recording, managing, and generating predefined instance models. Next, an EMF feature metamodel must be constructed, against which the comp-editor should easily be adaptable for feature instance modelling. This will require full definition of the feature modelling language indicated by Fig. 1 and elaborated by the *Switch* definition. A further, more costly development, would be a graphical version of the EMF feature instance modeller.

Methodological work - beyond that in this paper - has started with an approach based on the "refinement by restriction" of [27]. A "maximal" Event-B model is constructed, containing all features. The feature model is annotated with mapping information to the Event-B model, so that instance modelling is done by slicing required features into the output instance model according to these mappings.

5 Related work

Recent proposals [8, 6] identifying generic algebraic models for feature-oriented software construction schemes are relevant to our work. These models can support instance construction; e.g. (i) associative composition operators give freedom in how they can be ordered, (ii) the occurrence of non-commuting compositions can indicate feature interactions. These ideas will inform the development of an algebra of Event-B features.

Turning to verification, another recent development [7] presents a product-line development where verification - theorem statements and their proofs - is modularized and assembled by features. For certain Event-B composition operators, certain properties (POs) are guaranteed by construction, as indicated in section 1. For most operators this will not be true, and patterns of construction will be sought that propagate POs, either partially or completely. [7] is encouraging, but we note that its case study exploits the fact that the feature increments are logical *conservative extensions*, i.e. each increment to the feature model does not interfere with prior features in the construction order. While Event-B *superposition* refinements, which simply add structure, should work similarly, in general refinements will not be modularizable in this way.

The notion of a feature as a reusable requirement [12] or an increment in functionality [10] emerged in the context of domain modeling and software product line engineering. However, features are often considered as concepts only, i.e., as names without any predefined semantics [12]. Feature diagrams can be given an (internal) semantics by translating them into propositional logic [10, 26], which can be used for checking the consistency of entire diagrams as well as individual configurations. Feature diagrams can also be "lifted" from a pure domain modeling method to a programming method by defining mappings into class diagrams [12], or by defining features as programming language constructs, e.g., in the language FeatureC++ [5]. Such feature-oriented programming languages [9] are usually implemented using generative techniques, e.g., mixins [23]. We anticipate that our approach will lift in the same way to UML-B [25], a grahical UML-like front-end for Event-B.

In formal methods, a variety of formally well-founded structuring methods have been developed, such as the ladder construction [24]. However, these typically focus on module composition and parameterization [15] and do not allow the combination of incomplete specification elements that could represent features.

6 Conclusion and future work

We have outlined a usable (if intricate) syntactic scheme and graphical notation for the automatable composition of each product line instance from a set of specializable features. The fact that this could be done based on a set of simple models with no prior generic structuring through some domain analysis process, gives us some confidence in this enterprise. It is of course a very modest start which must now be built on.

The future work required is extensive but clearly contributes to an existing agenda in both Generative Programming and Formal Methods communities, as identified at GPCE'06 [17]. This work consists of methodological (see section 4), theoretical, and tooling strands.

Theory:

1. From case study work, full definition of the feature composition operators outlined in section 4.
2. Establish the extent to which unspecialized features can be proved consistent, and can be proved refinements. Can this extend to the liveness POs ?
3. For all possible feature composition operators of section 4, proof of compositionality. For noncompositional operators, an investigation of what properties can be established.

Technology:

1. From case study work, definition of a feature modelling language for Event-B. To include graphical as well as composition/instantiation syntax as per section 3.
2. Implementation of a prototype working subset of such operators in RODIN, based on our current interactive composition editor prototype.

3. Development of a full feature and instance modelling toolset inspired by e.g. FeaturePlugin [4]. GUI design will be appropriate both to RODIN and to the visualisation demands of the user building or instantiating feature models.
4. Validation by case study application.
5. The RODIN provers build and manage proof trees for every proved PO, and take a reuse-oriented approach to the management of these trees, when models, then POs, then finally proof trees change. We need to investigate, for the many cases where full compositionality does not apply, whether unspecialized feature proof trees can be transformed for reuse in proving POs about their compositions. For example, if we have proof trees for features f, g and their refinements $\{f_i\}, \{g_j\}$, to what extent can we transform any of these proof trees to be be applicable for reuse in proof about $f \oplus_{slrp} g$ and its refinements ?

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] J. R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for Event-B. In Z. Liu and J. He, editors, *Proc. ICFEM 2006*, volume 4260 of *LNCS*, Macau, 2006.

[3] J.-R. Abrial and S. Hallerstede. Refinement, decomposition and instantiation of discrete models: Application to Event-B. *Fundamenta Informaticae*, 77(1-2):1–28, 2007.

[4] M. Antkiewicz and K. Czarnecki. FeaturePlugin: Feature modeling plug-in for Eclipse. In *Eclipse '04: Proceedings of the 2004 OOPSLA workshop on Eclipse technology eXchange*, pages 67–72, New York, NY, USA, 2004. ACM Press.

[5] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In R. Glück and M. R. Lowry, editors, *GPCE*, volume 3676 of *Lecture Notes in Computer Science*, pages 125–140. Springer, 2005.

[6] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebra for features and feature composition. In J. Meseguer and G. Rosu, editors, *AMAST*, volume 5140 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2008.

[7] D. Batory and E. Börger. Modularizing theorems for software product lines: The jbook case study. *JUCS*, to appear.

[8] D. Batory and D. Smith. Finite map spaces and quarks: Algebras of program structure. Technical Report TR-07-66, Department of Computer Sciences, University of Texas at Austin, December 2007.

[9] D. S. Batory. Feature-oriented programming and the ahead tool suite. In *ICSE*, pages 702–703. IEEE Computer Society, 2004.

[10] D. S. Batory. Feature models, grammars, and propositional formulas. In J. H. Obbink and K. Pohl, editors, *SPLC*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.

[11] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework*. Addison-Wesley, 2003.

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

[13] B. Dehbonei and F. Mejia. Formal development of safety-critical software systems in railway signalling. In M. Hinchey and J. Bowen, editors, *Applications of Formal Methods*, chapter 10, pages 227–252. Prentice-Hall, 1995.

[14] E. Gamma and K. Beck. *Contributing to Eclipse*. Addison-Wesley, 2003.

[15] J. A. Goguen. Parameterized programming. *IEEE Trans. Software Eng.*, 10(5):528–544, 1984.

[16] C. Jones. Intermediate report on methodology. Technical Report Deliverable 19, EU Project IST-511599 - RODIN, August 2006. http://rodin.cs.ncl.ac.uk.

[17] G. T. Leavens, J. R. Abrial, D. Batory, M. Butler, A. Coglio, K. Fisler, E. Hehner, C. B. Jones, D. Miller, S. Peyton-Jones, M. Sitaraman, D. R. Smith, and A. Stump. Roadmap for enhanced languages and methods to aid verification. In *Proc. 5th Int. Conf. Generative Programming and Component Engineering*, Portland, Oregon, 2006.

[18] T. Lecomte, T. Servat, and G. Pouzance. Formal methods in safety-critical railway systems. In *Proc. 10th Brasilian Symposium on Formal Methods*, 2007.

[19] C. Métayer, J.-R. Abrial, and L. Voisin. Event-B Language. Technical Report Deliverable 3.2, EU Project IST-511599 - RODIN, May 2005. http://rodin.cs.ncl.ac.uk.

[20] M. Poppleton. Towards feature-oriented specification and development with Event-B. In P. Sawyer, B. Paech, and P. Heymans, editors, *Proc. REFSQ 2007: Requirements Engineering: Foundation for Software Quality*, volume 4542 of *LNCS*, pages 367–381, Trondheim, Norway, June 2007. Springer.

[21] M. Poppleton. The composition of Event-B models. In E. Boerger, editor, *Proc. ABZ 2008*, volume 5238 of *LNCS*, page 209222, London, September 2008. Springer.

[22] Robert Bosch GMBH. Specification on/off switch. 2008.

[23] Y. Smaragdakis and D. S. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002.

[24] D. R. Smith. Toward a classification approach to design. In M. Wirsing and M. Nivat, editors, *AMAST*, volume 1101 of *Lecture Notes in Computer Science*, pages 62–84. Springer, 1996.

[25] C. Snook and M. Butler. U2B - a tool for translating UML-B models into B. In J. Mermet, editor, *UML-B Specification for Proven Embedded Systems Design*, chapter 5. Springer, 2004.

[26] J. Sun, H. Zhang, Y.-F. Li, and H. H. Wang. Formal semantics and verification for feature modeling. In *ICECCS*, pages 303–312. IEEE Computer Society, 2005.

[27] A. Wasowski. Automatic generation of program families by model restrictions. In *SPLC 2004*, volume 3154 of *LNCS*, pages 73–89. Springer, 2004.

Towards a Holistic Approach for Integrating Middleware with Software Product Lines Research

Aniruddha Gokhale
ISIS, Dept. of EECS
Vanderbilt University
Nashville, TN 37235, USA
gokhale@dre.vanderbilt.edu

Akshay Dabholkar
ISIS, Dept. of EECS
Vanderbilt University
Nashville, TN 37235, USA
aky@dre.vanderbilt.edu

Sumant Tambe
ISIS, Dept. of EECS
Vanderbilt University
Nashville, TN 37235, USA
sutambe@dre.vanderbilt.edu

ABSTRACT

Prior research on software product lines (SPLs) in different domains (e.g., avionics mission computing, automotive, cellular phones) has focused primarily on managing the commonalities and variabilities among product variants at the level of application functionality. Despite the fact that the application-level SPL requirements drive the specializations (i.e., customizations and optimizations) to the middleware that host the SPL variants, middleware specialization is seldom the focus of SPL research. This results in substantial and ad hoc engineering efforts to specialize middleware in accordance with the functional and quality of service (QoS) requirements (e.g., latency, reliability) of the product lines. To overcome these problems, this paper highlights the need to unify middleware specialization issues within SPL research, and argues for new research directions in modularization and generative programming techniques that can account for the deployment and runtime issues, such as QoS and resource management. Preliminary ideas demonstrating how feature-oriented programming and model-driven development tools together can address these challenges are presented.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Product Lines, Middleware Specializations, Modularizations

Keywords

Generative programming + Product lines, FOP/AOP + MDD

1. INTRODUCTION

Research on software product lines (SPLs) [7] has focused primarily on managing the commonalities and variabilities [8] in application-level functionality of product variants. Generative programming [9] and modularization techniques, such as feature-oriented programming (FOP) [20] and aspect-oriented programming (AOP) [12], play an important role in composing and synthesizing product variants from modularized units called features and aspects.

Middleware is an important asset for SPLs across many domains, such as avionics (*e.g.*, Boeing's Bold Stroke architecture [23]), telecommunications (*e.g.*, Ericsson's family of carrier class switches [1]) and even cell phones (*e.g.*, Nokia or Motorola's family of cell phones). Middleware manages the quality of service (QoS) (*e.g.*, latency, reliability, security), and resource management (*e.g.*, bandwidth, CPU, memory) issues in product variants of a SPL. SPL developers tend to rely on standardized, general-purpose middleware, such as but not limited to J2EE, .NET Web Services, and CORBA, since these middleware provide a reliable, robust, low cost and low maintenance solution with the added benefit of feature-richness, flexibility, and high degree of configurability.

Although existing research in SPLs has significantly improved the quality of product lines, and reduced their development and maintenance costs, these research efforts have seldom addressed the challenges in effectively using middleware for SPLs. Addressing middleware challenges as part of SPL research is necessary since the feature-richness and flexibility of general-purpose middleware often becomes a source of excessive resource consumption and a lost opportunity to optimize for significant performance gains and/or energy savings in SPLs. Moreover, it is infeasible for general-purpose middleware to provide solutions to all possible domain-specific requirements since they are developed with the aim of broader applicability. Developing proprietary middleware for SPLs, however, is not a viable solution due to the excessively high development and maintenance costs.

In the current state of the art these limitations are addressed through significant but often *ad hoc* engineering efforts at specializing (*i.e.*, customizing and optimizing) general-purpose middleware. To overcome these deficiencies, there is a compelling need for SPL research to consider middleware platforms as an integral part of the SPL engineering processes and methodologies. This in turn argues for new research directions in modularization and generative programming techniques that account for QoS and resource management challenges, which are inherently deployment- and run-time problems, while most generative/modularization techniques are limited to design-time.

This paper proposes an integrated SPL methodology that incorporates capabilities for *middleware specialization*. Specialization is a process that manipulates general-purpose middleware in accordance with the commonalities and variabilities of an SPL by (a) adding custom features supplied by the application, (b) pruning unwanted features, and (c) optimizing the resulting middleware to address QoS and resource requirements of SPLs. Our approach is based on exploiting a hitherto before untapped algebraic structure of middleware by synergistically integrating (a) Origami ma-

McGPLE GPCE '08 Nashville, TN, USA

trices [4], which provide a formal representation for feature composition, interaction and refactoring [14], (b) Aspects [12], which modularize software that exhibits crosscutting characteristics into reusable features, and (c) Generative programming [9], which promotes automation in middleware specialization.

The remainder of this paper is organized to portray our vision of middleware specialization shown in Figure 1. Section 2 determines the problem space for middleware specialization; Section 3 describes the details of our holistic approach to combining middleware specializations with SPL research; and Section 4 provides concluding remarks and discusses open research issues.

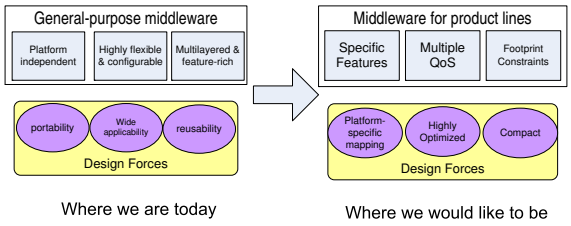


Figure 1: Middleware Specializations for SPLs

2. THE PROBLEM SPACE FOR MIDDLEWARE SPECIALIZATION

This section helps to define the problem space for middleware specialization in the context of SPLs.

2.1 Middleware System Model

The concept of middleware was born with the aim to shield applications from variabilities in lower-level artifacts, such as hardware, networks and compilers of programming languages. Years of middleware research resulted in a middleware model approximated by Figure 2. Middleware is made up of layers of software targeted to perform specific activities. At the bottom, the host infrastructure layer (*e.g.*, a Java virtual machine or the ACE [21] middleware) shields developers from the differences in operating systems and hardware. Next, the distribution layer (*e.g.*, CORBA or Java RMI) provides features for location transparency, request processing, and data marshaling, among others. The common services (*e.g.*, CORBA Naming or the UDDI discovery service) include features, such as naming, transaction, fault tolerance and real-timeliness, etc. At the top, the domain-specific middleware layer is tailored to a particular domain, such as avionics.

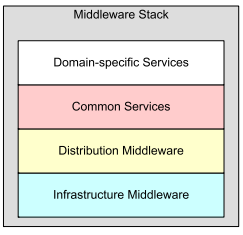


Figure 2: System Model for Middleware Specialization.

2.2 Survey of Related Research

Now we survey and organize related work along different dimensions that we observe to be prevalent in middleware specialization research.

• *Eliminating overhead of object-orientation:* Lohmann et. al. [15] argue that the development of fine-grained and resource-efficient system software product lines requires a means for separation of concerns [25] that does not lead to extra overhead in terms of memory and performance. The overhead of object-oriented programming (OOP), *e.g.*, due to dynamic binding and method dispatch, is not acceptable for some embedded systems. Aspect-oriented programming (AOP) [12] is shown to eliminate this overhead. Aspects are modularized pieces of code that traditionally are scattered across application code.

• *Aspects for footprint reduction:* AOP provides a novel mechanism to reduce footprint by enabling crosscutting concerns between software modules to be encapsulated into user selectable aspects. *FACET* [11] identifies the core functionality of a middleware framework and then codifies all additional functionality into separate aspects. To support functionality not found in the base code, FACET provides a set of features that can be enabled and combined subject to some dependency constraints. By using AOP techniques, the code for each of these features can be weaved at the appropriate place in the base code.

• *Combining modeling and aspects for refinement:* the *Modelware* [27] methodology adopts both the model-driven architecture (MDA) [17] and AOP. Borrowing terms from subject-oriented programming [10], the authors use the term *intrinsic* to characterize middleware architectural elements that are essential, invariant, and repeatedly used despite the variations in the application domains. They use the term *extrinsic* to denote elements that are vulnerable to refinements or can become optional when the application domains change. Modelware advocates the use of models and views to separate intrinsic functionalities of middleware from extrinsic ones. Modelware considerably reduces coding efforts in supporting the functional evolution of middleware along different application domains.

• *Combining computational reflection and aspects:* computational reflection is an efficient and simple way of inserting new functionality into reflective middleware, such as *LOpenOrb* [5]. It uses a meta-object protocol to abstract away the implementation details so that it is necessary only to know the components and interfaces. To conserve resources and provide dynamic adaptation, AOP can be used to specialize the reflective middleware. Aspects that are not in the application code can be dynamically inserted using a meta-object protocol.

• *Layer collapsing and bypassing:* In a typical middleware platform every request passes through each layer, whether or not the services provided by that layer are needed for that specific request. This rigid layered processing can lower overall system throughput, and reduce availability and/or increase vulnerability to security attacks [19]. For use cases where the response is a simple function of the request input parameters, bypassing middleware layers may be permissible and highly advantageous. Devanbu et. al [26, 19] have shown how AOP can be used to bypass middleware layers.

• *Importance of lifecycle stages:* Traditionally, performance problems in middleware layers have been addressed by optimizing the source code and data structures. Edicts [6] is an approach that shows how optimizations are also feasible at other application lifecycle stages, such as deployment- and run-time. Just-in-time middleware customization [28] shows how middleware can be customized after application characteristics are known. These efforts discover the configuration of the target environment and compose only the necessary modules that are best suited among alternatives and configure them in the most optimal way.

3. INTEGRATING MIDDLEWARE SPECIALIZATIONS WITH SPL METHODOLOGIES

We now describe our proposed approach to integrate middleware specialization with SPL methodologies.

3.1 A Middleware Case Study

To make the description of our proposed approach concrete we use a middleware case study. Figure 3 illustrates the CORBA middleware architecture, which is compliant with our layered middleware system model. Also shown in the figure are CORBA services, real-time CORBA (RTCORBA) [18] enhancements, and component-based abstractions. CORBA is used here only for illustration purpose, however, our approach is general.

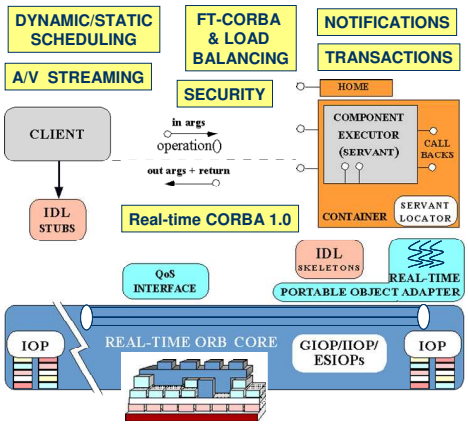


Figure 3: CORBA Architecture

The different RTCORBA features are shown in Figure 4. RTCORBA defines standard interfaces and QoS policies that allow applications to configure and control (1) *processor resources* via thread pools, priority mechanisms, intra-process mutexes, and a global scheduling service, (2) *communication resources* via protocol properties and explicit bindings, and (3) *memory resources* via buffering requests in queues and bounding the size of thread pools. Applications typically specify these real-time QoS policies along with other policies when they call standard CORBA operations, such as `create_POA` or `validate_connection`. For example, the priority at which requests must be handled can be propagated from the client to the server (the `CLIENT_PROPAGATED` model) or declared by the server (the `SERVER_DECLARED` model).

3.2 Uncovering the Algebraic Structure of Middleware

Despite a rich repertoire of features, specializations including feature additions, pruning or customizations to general-purpose middleware is a hard problem due to the following challenges posed by their design and implementation:

a. fundamental restrictions and limited flexibility of programming languages such as C++ or Java do not allow interception of the control flow at arbitrary points in the control flow graph to inject required application-specific functionality or remove certain unnecessary functionality. This is currently feasible only at limited points in the code known as *interception points*, which is often not sufficient.

b. although object-oriented designs help develop modular middleware code, this modularity incurs a performance penalty.

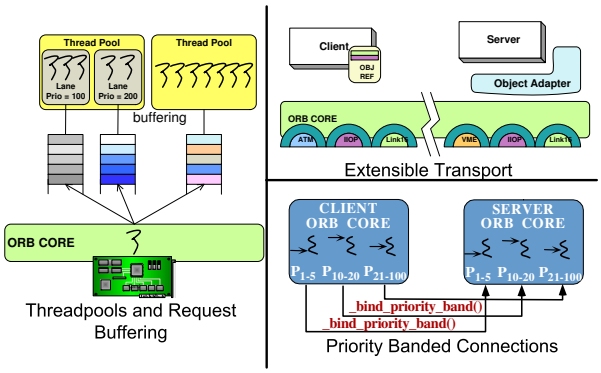


Figure 4: RTCORBA Features

Maintaining the modular design, which promotes longevity of the software, is desirable yet obtaining optimal performance is also required. There is a need to decouple specializations from the modular design, which is a hard problem.

c. the combinatorial complexity of the feature compositions makes it hard to find valid configurations manually because of the large number of middleware configuration options and complex semantic relationships between them.

d. deployment- and run-time specializations are even harder because feature removal and additions need to be considered simultaneously, systematically and in a semantically consistent and coordinated manner such that domain-specified requirements on performance and footprint are satisfied.

In Section 2.2 we discussed how Aspect-oriented programming (AOP) [12] is extensively used for middleware specialization (*e.g.*, [11, 27, 19]). AOP, however, does not support any architectural model to define transformations to the structure of programs, particularly the ability to encapsulate new classes, which limits its suitability for middleware specialization. Feature-oriented programming (FOP) [20] on the other hand can represent single aspects or collections of aspects, and also can complement model-based development since both paradigms stress the importance of transformations in automated program development [3]. Moreover, FOP has better support to provide bounded (*i.e.*, selective) quantification for feature manipulation in contrast to AOP techniques which often result in unbounded quantification.

FOP is thus a candidate approach for middleware specializations since it involves manipulation of middleware features. FOP is best suited when the underlying construct on which it operates displays a well-defined algebraic structure. FOP for middleware specializations is not straightforward, however, due to a lack of an explicit algebraic structure in the middleware design as explained above. We therefore ask ourselves whether it is possible to impose an algebraic structure on the middleware. A closer scrutiny of the middleware design reveals that if we raise the level of abstraction [24] to the level of features the middleware offers instead of focusing on source code-level details, then a strong algebraic structure unfolds wherein features can be manipulated using the FOP paradigm subject to some constraints.

We have therefore chosen the principles of AHEAD (Algebraic Hierarchical Equations for Application Design) [4], which is an implementation of FOP that uses stepwise refinement to synthesize application product lines, as the basis of the proposed approach. The notion of a feature in AHEAD is tied to basic object-oriented programming concepts, such as classes and methods. Al-

though middleware also often uses object-oriented design principles, our notion of features is at a higher level of abstraction involving patterns and frameworks that provide properties, such as real-timeliness and fault tolerance.

AHEAD starts with a small set of base capabilities and refines them by incrementally adding features. In contrast our middleware specializations start with a much larger software base pruning unwanted features and customizing the needed ones with domain-specific properties. Our goal is to enhance AHEAD and similar research to support design, deployment and run-time feature manipulation.

3.3 Exploiting the Algebraic Structure of Middleware

We now lay down our initial ideas on our proposed approach to middleware specializations based on recursive algebraic approaches such as AHEAD, however, by operating at a level of abstraction for features that is closer to patterns and frameworks, and across all stages of the application lifecycle.

Table 1 depicts our attempt to capture the algebraic structure of RTCORBA capabilities as features within an Origami matrix as proposed by the AHEAD approach [4] with the difference that our level of abstraction for features is different and we consider all stages of application lifecycle. Origami is a generalization of binary decision matrices, where matrix axes define different sets of features, and matrix entries define feature interactions. Origami matrices possess a special property in that they allow folding along the rows or columns or both. We discuss how this property will be used.

Base \ RT	BasicRT	Priority	Conc	Synch
ORB	RTORB	PriMapper	TPReactor	
POA	RTPOA			
Xport	ExtXport	BandConn		
ReqHndl		CLI_PROP	TPLane	MUTEX

Table 1: Origami Matrix for RTCORBA

We use rows to denote the basic CORBA features, such as the object request broker (ORB) that mediates requests and manages resources; the portable object adapter (POA) that manages object lifecycle; the Transport (shown as Xport) which handles communication; and ReqHandling which provides the data marshaling and handling of requests. The columns denote the real-time features that refine the basic features of CORBA with real-time capabilities. For example, BasicRT indicates the base capabilities that introduce real-time properties; Priority indicates the priority handling mechanisms; Concurrency and Synchronization are classical distributed computing properties and describe the RTCORBA mechanisms that support these.

The individual cells illustrate the feature interaction across the row and column. For example, the CLI_PROP cell indicates the priority model to be used in request handling. We assume that the RTORB shown in the top-left cell is the constant required by AHEAD. In reality, however, a single cell such as RTORB can itself be formed by its own nested Origami matrix where different features are composed to realize the notion of an RTORB. An empty cell indicates a composition *identity*, which does not change anything to the feature on which it is composed.

Now imagine a stepwise folding of columns onto each other, which in turn folds individual cells onto each other for all the rows. This cell-wise folding results in the composition of features of the folded cells. Table 2 depicts the folding of the third and fourth column in the original matrix. Continuing this folding along all columns and then rows (order does not matter) gives rise to a composition of features that constitutes the overall RTCORBA middleware and can be represented by Equation 1. Features are composed with each other using the composition operator •.

Base \ RT	BasicRT	Priority • Conc	Synch
ORB	RTORB	PriMapper • TPReactor	
POA	RTPOA		
Xport	ExtXport	BandConn	
ReqHndl		CLI_PROP • TPLane	MUTEX

Table 2: Folded Matrix for RTCORBA

$$RTCORBA = MUTEX \bullet TPLane \bullet CLI\_PROP \bullet BandConn \bullet ExtXport \bullet TPReactor \bullet PriMapper \bullet RTPOA \bullet RTORB \quad (1)$$

Now let us explore how such equations will help us. Our previous work [13] on handcrafted middleware specialization has showed how the RTCORBA middleware stack characterized by Equation 1, forces the software components of our avionics mission computing scenario to use all the features, many of which are sources of excess generality. We claim that an approach to prune unwanted features can follow a similar folding operations of the Origami matrix that produces an equation of features to be pruned (*e.g.*, bypassing the request demultiplexing logic) and customized (*e.g.*, caching requests). This can be attempted by the application developer or middleware developers who are given the requirements by domain experts. The algebraic difference between the RTCORBA equation and the equation describing the excess generality provides a formal approach to specializing middleware.

Notice how this proposed approach is no longer *ad hoc* unlike handcrafted specializations. This desired property stems from the significant benefit of an Origami matrix in that it can realize only valid compositions of features. Notice that erroneous compositions (*e.g.* folding along the diagonal) or differences are impossible due to the constraints imposed by the folding capability of the Origami matrix. A model-based tool can provide an approach to collect all the domain requirements, which then can be used to drive the Origami folding and synthesis of the different equations.

3.4 Feature Manipulations across Application Lifecycle Stages

The composition operator • is part of a well-defined algebra [2], which to our knowledge works only for design-time feature composition. AHEAD (and hence Origami) does not support deployment- and run-time feature manipulation. We argue for new research in enhancing existing SPL research, such as AHEAD, to include deployment- and run-time phases of application lifecycle. Applying AHEAD principles to cover all the stages of application lifecycle is hard however because the level of abstraction it operates at (*e.g.*, code level) is not suitable for feature manipulations in the deployment- and run-time stages, and it is conceivable that the existing feature algebra will be incompatible at these stages.

We make an initial attempt to enhance this theory. Imagine a third dimension added to Table 1, which defines the deployment dimension. We can visualize this scenario as comprising multiple planes each having its own Table 1, where each table corresponds to the middleware specialization for the hosted component of the product variant. Suppose that the deployment of the product variant must ensure that the middleware is specialized for the CLIENT_PROPAGATED priority model. Now suppose that one

such table uses a SERVER_DECLARED priority model for request handling instead. We need an approach by which the folding operation along the third dimension should throw an exception due to a misconfiguration in one of the matrices. Run-time issues such as adding features for, say, a coordination layer for fault management can be handled by extending the Origami matrix in the fourth dimension to cover these run-time issues.

3.5 Feature Interactions across Application Lifecycle Stages

Our discussions so far have assumed that features are independent of each other and that they can be seamlessly added or pruned. However, we cannot make such simplified assumptions in all cases. For example, Narasimhan et. al. [16] have illustrated how real-time and fault-tolerance properties of applications conflict with each other thereby requiring tradeoffs.

Figures 5 and 6 illustrates how features can interact [14] with each other at the framework level (which is our level of abstraction for features). We show two design possibilities for an RTORB that supports thread pools with lanes. The thread pool serves as an additive refinement to the RTORB (*i.e.*, an Introduction). However, as shown in Figures 5 and 6, the request handling strategy interacts with the RTORB in different ways each with its benefits and consequences on performance.

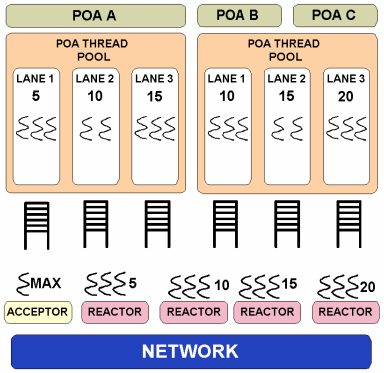


Figure 5: Queue-per-lane Design for Threadpool-with-Lane

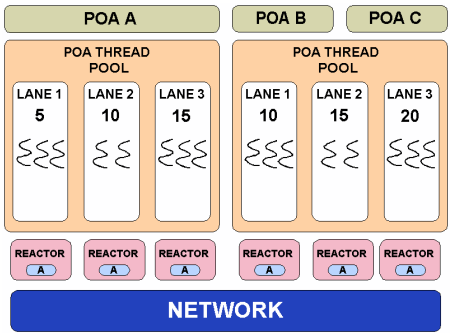


Figure 6: Reactor-per-lane Design for Threadpool-with-Lane

In the `queue_per_lane` strategy, a separate thread listens for requests over the network and hands over the request to a worker thread, which is the Half-Sync/Half-Async architectural pattern [22]. This model simplifies the design but incurs message queuing and thread synchronization overhead. In the `reactor_per_lane` approach, the thread that receives the request also handles the request, which is the Leader-Follower architectural pattern [22]. This model is difficult to implement and debug for race conditions.

We showed design-time feature interactions above, which is a hard problem since our features represent patterns and frameworks. Feature interactions at other lifecycle stages are even harder to address. Simple foldings of Origami columns and rows may not suffice since the foldings have no capabilities to tradeoff one feature over the other as in the case of fault tolerance and real-time. Traditionally the tradeoff problems have been mapped to combinatorial optimization problems where heuristics are developed to find near-optimal solutions.

4. CONCLUDING REMARKS AND OPEN ISSUES

Middleware is an important asset of SPLs that operate in a distributed computing environment. In this paper we argued for extending SPL research to incorporate middleware specializations. We showed how an algebraic structure can be imposed on the middleware which in turn makes it suitable for feature manipulation. We then explored the use of generative programming and modularization techniques based on AHEAD for middleware specialization outlining how they can be extended to address deployment- and run-time issues in middleware.

A number of open issues remain unresolved as explained below.

- *Mapping higher-level feature abstractions to code:* Since the algebraic structure we consider is at a higher level of abstraction, we require a mappping from the high level artifacts to low level details such as code. Naturally, such a mapping cannot break existing code. Hence we will need out-of-band mechanisms such as source code annotations including those we developed in our preliminary work [13] or aspect definitions to refactor existing middleware into the algebraic form we require.
- *Semantics of the composition and difference operator for deployment- and run-time phases of the application lifecycle:* Is a single equation feasible that can capture the specializations to middleware by accounting all three phases of application lifecycle. An important open issue points to the algebra of these operators across the lifecycle. A number of questions must be answered: What is the associativity and precedence relationship of the operators along the lifecycle stages? Do the semantics of Origami folding change in different lifecycle stages? How are features represented at the other lifecycle stages? How can Origami folding handle distributed coordination at run-time? Can Origami capture system schedulability and performance optimizations?
- *Runtime Tradeoffs via Origami foldings:* Adaptive systems must make runtime tradeoffs among inherently conflicting system properties such as real-timeliness and fault-tolerance. Many questions must be answered if Origami abstractions are used to solve these challenges: Do individual cells encode constraints? Do foldings give rise to cost functions? How do constraints get refined during folding? What does the final equation represent? Does the composition operator encode a heuristic to solve the optimization problem? How can feature manipulations be considered simultaneously, systematically and in a semantically consistent and coordinated manner such that domain-specified requirements on QoS and footprint are satisfied?

5. REFERENCES

[1] G. Ahlforn and E. Örnulf. Ericsson's Family of Carrier-class Technologies. *Ericsson Review*, 4:190–195, Apr. 2001.

[2] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An Algebra for Features and Feature Composition. In *Proceedings of the International Conference on Algebraic Methodology and Software Technology (AMAST)*, pages 36–50. LNCS vol. 5140, Springer-Verlag, 2008.

[3] D. Batory. Using Modern Mathematics as an FOSD Modeling Language. In *To Appear in the Proceedings of the Generative Programming and Component Engineering (GPCE 08)*, New York, NY, USA, 2008. ACM.

[4] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[5] G. S. Blair, G. Coulson, P. Robin, and M. Papathomas. An Architecture for Next Generation Middleware. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 191–206, London, 1998. Springer-Verlag.

[6] V. Chakravarthy, J. Regehr, and E. Eide. Edicts: Implementing Features with Flexible Binding Times. In *AOSD '08: Proceedings of the 7th International Conference on Aspect-oriented Software Development*, pages 108–119, New York, NY, USA, 2008. ACM.

[7] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, 2002.

[8] J. Coplien, D. Hoffman, and D. Weiss. Commonality and Variability in Software Engineering. *IEEE Software*, 15(6), November/December 1998.

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

[10] W. Harrison and H. Ossher. Subject-oriented Programming: A Critique of Pure Objects. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 411–428, New York, NY, USA, 1993. ACM.

[11] F. Hunleth and R. K. Cytron. Footprint and Feature Management Using Aspect-oriented Programming Techniques. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems (LCTES 02)*, pages 38–45. ACM Press, 2002.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242, June 1997.

[13] A. Krishna, A. Gokhale, D. C. Schmidt, J. Hatcliff, and V. Ranganath. Context-Specific Middleware Specialization Techniques for Optimizing Software Product-line Architectures. In *Proceedings of EuroSys 2006*, pages 205–218, Leuven, Belgium, Apr. 2006.

[14] J. Liu, D. Batory, and C. Lengauer. Feature Oriented Refactoring of Legacy Applications. In *Proceedings of the International Conference on Software Engineering*, pages 112–121. ACM Press New York, NY, USA, 2006.

[15] D. Lohmann, O. Spinczyk, and W. Schröder-Preikschat. Lean and Efficient System Software Product Lines: Where Aspects Beat Objects. *Transactions on AOSD II*, 4242:227–255, 2006.

[16] P. Narasimhan. Trade-Offs Between Real-Time and Fault Tolerance for Middleware Applications. Workshop on Foundations of Middleware Technologies, Nov. 2002.

[17] Object Management Group. *Model Driven Architecture (MDA)*, OMG Document ormsc/2001-07-01 edition, July 2001.

[18] Object Management Group. *Real-time CORBA Specification*, 1.2 edition, Jan. 2005.

[19] Ömer Erdem Demir, P. Dévanbu, E. Wohlstadter, and S. Tai. An Aspect-oriented Approach to Bypassing Middleware Layers. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 25–35, New York, NY, USA, 2007. ACM Press.

[20] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In M. Aksit and S. Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241, pages 419–443, Jyväskylä, Finland, 9–13 1997. Springer.

[21] D. C. Schmidt. The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software. In *Proceedings of the* 12^{th} *Annual Sun Users Group Conference*, pages 214–225, San Jose, CA, Dec. 1993. SUG.

[22] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. Wiley & Sons, New York, 2000.

[23] D. C. Sharp. Reducing Avionics Software Cost Through Component Based Product Line Development. In *Software Product Lines: Experience and Research Directions*, volume 576, pages 353–370, Aug 2000.

[24] J. A. Stankovic, P. Nagaraddi, Z. Yu, Z. He, and B. Ellis. Exploiting Prescriptive Aspects: A Design time Capability. In *EMSOFT '04: Proceedings of the 4th ACM International Conference on Embedded Software*, pages 165–174, New York, NY, USA, 2004. ACM Press.

[25] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE '99: Proceedings of the International Conference on Software Engineering*, pages 107–119, May 1999.

[26] E. Wohlstadler, S. Jackson, and P. Devanbu. DADO: Enhancing Middleware to Support Crosscutting Features in Distributed, Heterogeneous Systems . In *Proceedings of the International Conference on Software Engineering*, Portland, OR, May 2003.

[27] C. Zhang, D. Gao, and H.-A. Jacobsen. Generic Middleware Substrate Through Modelware. In *Proceedings of the 6th International ACM/IFIP/USENIX Middleware Conference*, pages 314–333, Grenoble, France, 2005.

[28] C. Zhang, D. Gao, and H.-A. Jacobsen. Towards Just-in-time Middleware Architectures. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 63–74, New York, NY, USA, 2005. ACM Press.

Modeling Dependent Software Product Lines

Marko Rosenmüller, Norbert Siegmund, Christian Kästner, Syed Saif ur Rahman

School of Computer Science,
University of Magdeburg, Germany
{rosenmue, nsiegmun, kaestner, srahman}@ovgu.de

Abstract

Techniques to model software product lines (SPLs), using feature models, usually focus on a single SPL. Larger SPLs can also be built from smaller SPLs which results in a dependency between the involved SPLs, i.e., one SPL uses functionality provided by another SPL. Currently, this can be described using constraints between the involved feature models. However, if multiple differently configured instances are used in a composition of SPLs, dependencies between the concrete instances have to be considered. In this paper, we present an extension to current SPL modeling based on class diagrams that allows us to describe SPL instances and dependencies among them. We use SPL specialization to provide reuse of SPL configurations between different SPL compositions.

1. Introduction

Reuse in *software product lines (SPLs)* is achieved by combining assets, e.g., components, to produce a number of similar programs [4]. The resulting concrete products of an SPL (*SPL instances*) are variants tailored to a specific use-case or environment. Large SPLs can be built by reusing functionality provided by smaller SPLs and sometimes functionality of multiple SPLs is integrated into one SPL [18]. This results in a composition of SPLs where compatibility between interacting SPLs has to be ensured. As an example, consider a mail application developed as an SPL (MailClient in Figure 1). The client uses mail communication functionality provided by a MailFramework SPL (e.g., different mail protocols) and two differently configured instances of list SPLs (SortedList and SynchronizedList). To ensure correct composition the MailFramework has to be configured according to the requirements of the MailClient. For example, using the IMAP mail protocol in the MailClient requires the MailFramework to provide this protocol. This is getting more complex if multiple product lines are involved, e.g., the mail client in Figure 1 uses two additional instances of a product line of list data structures that also have to be configured appropriately. Such systems can be seen as large SPLs composed from smaller SPLs, i.e., *product lines of product lines* or *nested product lines* [13]. Proper configuration of such *dependent SPLs* not only ensures compatibility but also reduces consumed resources by removing unneeded functionality, avoids unneeded dependencies to other programs, and can reduce the user interface.

A user who configures an SPL that depends on other SPLs is usually only interested in configuration decisions of her problem domain and not in the configuration of underlying SPLs. For example, configuring the MailClient should not involve configuration of the underlying MailFramework SPL. Hence, SPLs used within other SPLs should be automatically configured to match the requirements of the enclosing SPL and only functionality a user is interested in has to be configured manually. This is possible by defining constraints between dependent SPLs that enforce only valid combinations and can be automatically resolved at configuration

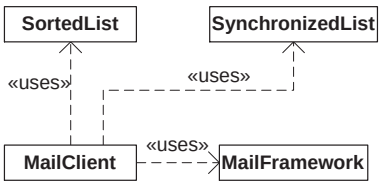


Figure 1. A MailClient SPL using a MailFramework SPL and different instances of a List SPL.

time. Such constraints (e.g., *requires* constraints between MailClient SPL and MailFramework) can be described as constraints between the feature models of these SPLs [4]. However, if multiple similar variants of one SPL are used, constraints between concrete SPL instances (*instance constraints*) are needed. For example, the MailClient uses two different instances of a list SPL (cf. Fig. 1). These instances have to be be configured differently, i.e., one as a sorted list and one as a synchronized list. A domain level constraint between mail client SPL and list SPL as used in current domain modeling cannot describe this dependency.

In this paper, we extend existing product line modeling with an approach that aims at modeling compositions of dependent SPLs. Our goal is to connect domain modeling and domain implementation: while feature models describe the features of an SPL we use SPL instance models to describe the composition of SPLs. Furthermore, we want to separate dependencies needed for SPL configuration, i.e., the uses-relationship between SPL instances, from concrete SPL implementation. Furthermore, we integrate domain modeling and SPL instances by mapping a feature of an SPL to instances of SPLs that are referenced by this feature. This is in line with *feature-oriented software development* where all software artifacts are decomposed with respect to the features of a domain [2]. By including *SPL specialization* [5] we are able to reuse SPL configurations in different SPL compositions. A combination of domain modeling and the presented instance modeling can be used to derive configuration generators that create instances of all dependent SPLs of a composition and thus provide the basis for an automated configuration process.

2. Software Product Line Engineering

In the following, we shortly present foundations of *software product line engineering (SPLE)* and the current state of techniques used to model and implement SPLs.

Domain Modeling. An SPL is used to create similar programs that share some common *features*. The features of an SPL are distinguishable characteristics of software that are of interest to some stakeholder [4]. As part of *feature-oriented domain analysis*

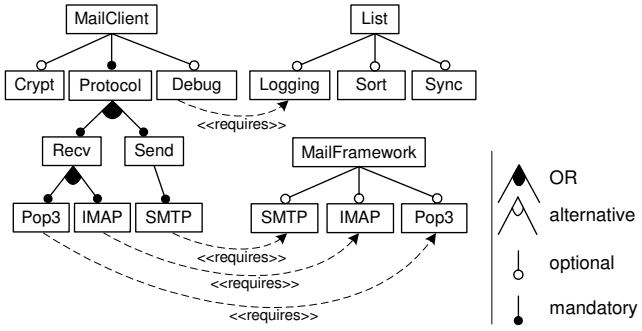


Figure 2. Feature diagram of a mail client SPL that uses a mail framework SPL with requires-constraints between SPLs (shown as dashed arrows).

(FODA), SPLs can be described using *feature models* [10, 4]. These are usually visualized using *feature diagrams* [10] as shown for a MailClient product line in Figure 2. The MailClient SPL uses other SPLs: a MailFramework SPL that provides different mail protocols and another small SPL of list data structures. The root of a feature diagram (e.g., node MailClient) represents the SPL itself and remaining nodes represent features of that SPL (e.g., feature IMAP represents the IMAP mail protocol). Features can be optional (depicted with an empty dot) or mandatory (depicted with a filled dot). Variability introduced by features provides means to create tailor-made applications. For example, mail clients using different protocols are created by including the according features IMAP, POP3, and SMTP.

Domain Constraints. Feature models often contain *domain constraints* that ensure only valid feature combinations on the domain level. For example, *requires* (shown as dashed arrows in Fig. 2) and *mutual-exclusion* relations are used to describe dependencies between features [4]. Domain constraints can also be used to describe dependencies between different product lines [6, 16]. For example, if feature IMAP is used in the MailClient, also feature IMAP of the MailFramework SPL is required (cf. Fig. 2). A user of the MailClient SPL usually only wants to configure the MailClient itself and not all accompanied SPLs which she might not have any domain knowledge of. This can be achieved by automatically resolving constraints between SPLs, e.g., between MailClient and MailFramework.

Product Line Implementation. SPLs are implemented using a variety of technologies. Examples are components that are combined to build large systems [3] or C/C++ preprocessor definitions used to build SPLs in the embedded domain. New paradigms like *aspect-oriented programming (AOP)* [11] and *feature-oriented programming (FOP)* [14, 2] can also be used to implement SPLs. The approach that we present in this paper is independent of the used implementation technique.

Based on the SPL implementation a user derives a concrete product by selecting the needed features from an SPL. The resulting *SPL configuration* (i.e., feature selection) is used to compose the corresponding software assets that implement an SPL resulting in a tailored *SPL instance*. The created SPL instance might be a library, a component, a program, or a collection of programs. The concrete composition mechanism depends on the implementation technique.

3. Dependent Software Product Lines

By using domain constraints, dependencies within an SPL and between different SPLs can be modeled. In the following, we show

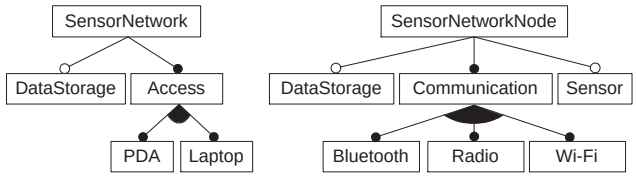


Figure 3. Feature diagrams of an SPL for a sensor network (left part) and an SPL for software used on sensor network nodes (right part).

that existing models have to be extended to completely describe arbitrary compositions of product lines and present requirements needed for an extension of current product line modeling.

Large Scale Product Lines. Complex and distributed systems, e.g., sensor networks, can be developed as product lines built from a number of heterogeneous SPL instances. For example, a SensorNetwork SPL as shown in Figure 3 may consist of different sensor nodes, data storage nodes, and access nodes each of them being an instance of a SensorNetworkNode SPL. Additionally, a client application accessing the sensor network might be developed as an SPL to support different client hardware (e.g., Laptops and PDAs) to interface with sensor network nodes. Dependencies between network nodes and client applications may exist to ensure a valid sensor network as a whole. Communication between sensor nodes, for instance, requires the same communication protocol and the access node of a sensor network might additionally require Bluetooth to communicate with clients (Laptop or PDA).

In contrast to the MailClient SPL, the SensorNetwork SPL is not an SPL from which a program is created but a number of interacting programs (i.e., the software running on nodes of the network and the client software to access the network). Hence, there might not be any source code needed for the SensorNetwork and only the smaller SPLs contain program code. This also affects the instantiation process: there is no particular composition process needed (e.g., using code transformation and compilation of code) but only instantiation of used product lines.

The SensorNetworkNode SPL again might use other SPLs that provide lower level functionality, e.g., an SPL for database management systems (DBMS) to store data. Hence, there can be chains of SPLs using instances of smaller SPLs. This composition might lead to large systems and also systems of systems. Each SPL in such a chain of SPLs requires an own model to describe dependencies to lower-level SPLs that it uses. By providing a separate composition model for each of these SPLs we can reuse these models in other product lines.

Compositions of Product Line Instances. Compositions of multiple SPLs imply that we have to handle these SPLs and constraints between them on the model level. Domain constraints can describe dependencies between different SPLs but do not take concrete instances into account. These instances, however, have to be considered if one SPL uses multiple differently configured instances of another SPL or if different instances of the same SPL depend on each other.

As an example consider our MailClient that uses multiple differently configured list data structures as shown in Figure 1. One instance of the List SPL is a synchronized List, i.e., using feature SYNC, and one is a sorted List, i.e., using feature SORT (cf. Fig. 2). In such a composition, we describe the requires relationship between feature DEBUG of the MailClient and feature LOGGING of the List using a domain constraint. This is not possible for features SORT and SYNC because the MailClient requires two different instances, one using feature SORT and one using feature SYNC. That

OO-concept	SPL representation
class	SPL
object	SPL instance
class specialization	staged configuration
aggregation	uses-relationship of SPLs
type of member variable	type of SPL instance
name of member variable	name of SPL instance

Table 1. OO-concepts and the corresponding representation of concepts in product lines.

is, we cannot describe constraints that affect only a concrete instance of an SPL.

We can find another example in the sensor network scenario. In this case, differently configured instances of nodes (e.g., data storage nodes and sensor nodes) are communicating with each other and one instance (e.g., a sensor node) depends on the functionality of another instance (e.g., a data storage node). Again, we cannot describe the dependencies in the feature model, which is the same for all nodes, because we would refer to the same feature model and not a concrete instance of it. To solve this problem we propose to extend feature modeling with explicit modeling of SPL instances.

Instance Identification. Using multiple instances of one SPL requires assigning a unique name to each instance to identify the differently configured instances and define constraints between them. For example, we have to create a name for the synchronized and sorted list that are used by the MailClient. Furthermore, we can use these names on the implementation level of an SPL in order to create class instances (e.g., list nodes) that are part of the different instances of an SPL. For example, name spaces or packages can be used to identify the SPL instances on the source code level. The concrete technique used to identify instances (e.g., Java packages) depends on the SPL implementation and is outside of the scope of this paper.

4. An Extension of Product Line Modeling

We have seen that constraints between SPL instances are needed to ensure correct configuration for a number of dependent SPLs. To avoid manual implementation of these constraints at the source code level of an SPL we present an extension to current product line modeling that allows a domain engineer to describe SPLs and SPL instances and specify constraints between them.

Modeling SPL Instances. The term *instantiation* is used in product line engineering as well as in OOP. In product line engineering, creating an SPL instance means to derive a concrete product from an SPL. In OOP, classes are instantiated resulting in concrete objects. Czarnecki et al. compared SPLs to classes of OOP and SPL configurations to class instances [5]. We adopt this correspondence and model SPLs and SPL instances using classes and objects of OOP. This also means that class instantiation corresponds to SPL instantiation. Using the concept of aggregation furthermore allows us to have members within classes where the type of a member corresponds to an SPL and the object assigned to such a member corresponds to an SPL instance. *Staged configuration* of SPLs, i.e., specialization of the feature model [5], can be represented by specialization as known from OOP. This means, a specialized class C_B of class C_A corresponds to a specialized SPL S_B of SPL S_A. Also subtyping of SPLs and polymorphism can be applied: S_B is a subtype of S_A and variables of type S_A can refer to instances of S_A or S_B. We summarized all corresponding constructs in Table 1.

Based on the correspondence of OOP classes and SPLs we can use class diagrams to model SPL compositions. By using class

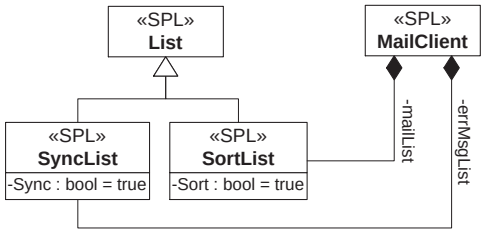


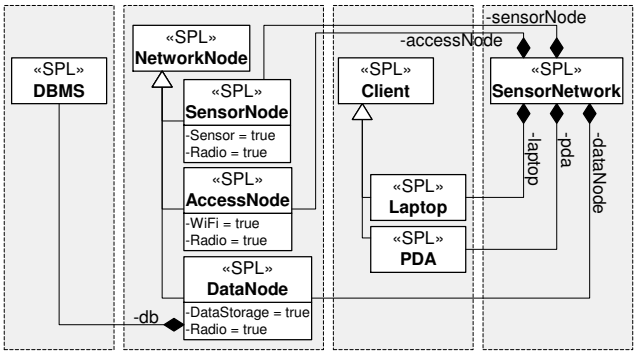
Figure 4. A MailClient SPL that uses different specializations of an SPL of list data structures (represented by aggregation). SPL specialization is represented by inheritance.

diagrams also complex compositions of SPLs can be created using existing tools and a familiar concept. Furthermore, existing support for generation of object-oriented code from class diagrams can be used to derive configuration generators from SPL composition models.

Using a class diagram, the MailClient example that uses a List SPL (cf. Fig 2) can now be modeled as shown in Figure 4. SPLs are represented by classes `MailClient` and `List`. Classes `SortList` and `SyncList` represent specialized variants of the `List` SPL that provide sorting and synchronization. The specialization, i.e., a pre-configured feature model, can be represented using special attributes of the classes (e.g., attribute `Sync` in class `SyncList`). Instances of SPLs used by other SPLs are described using aggregation, e.g., members `mailList` and `errMsgList` of class `MailClient` represent instances of different specialized List SPLs. By using specialized variants we can avoid constraints between `MailClient` and `List` that would be needed to define the different variants `SortList` and `SyncList`. Thus, we only have to refer to the specialized variants and can reuse the configuration of the specialized SPLs in other SPL compositions. Names of class members (e.g., `mailList` and `errMsgList`) are used to identify instances of an SPL.

Domain constraints are defined in the domain model and are still used to define constraints that apply for all instances of an SPL. For example, constraint `MailClient.Debug => List.Logging` (cf. Fig. 2) means that feature DEBUG of the MailClient SPL requires feature LOGGING of the List SPL. We can now provide additional constraints for specialized SPLs. For example, we can use `MailClient.Debug => SyncList.Logging` to enable feature LOGGING only in instances of synchronized lists because SyncList is a specialized variant of the List SPL. These constraints are part of the MailClient SPL and are separated from reusable specialized variants defined in the List SPL.

Instance Constraints. We use *instance constraints* to describe dependencies between SPLs and concrete instances. As an example, consider the model for a sensor network in Figure 5. The `SensorNetwork` SPL uses specialized instances of SPLs `Client` and `NetworkNode`. The specialized variant `DataNode` again uses an instance of SPL `DBMS` to store data. In the lower part, we depict constraints of the model. Domain constraint (1) is part of the domain model and shown for completeness. Additionally, we specify constraints between SPLs and specialized variants (2): feature PDA implies feature BLUETOOTH only in specialized variant `AccessNode`. We also used an instance constraint (3): if feature DATASTORAGE is used, we enable feature QUERIES in instance `pda` of the `SensorNetwork` SPL. Thus, only a concrete instance is affected and not the whole SPL.




```
domain constraints:
(1) SensorNetwork.Logging => Client.Logging
(2) SensorNetwork.PDA => AccessNode.Bluetooth

instance constraints:
(3) SensorNetwork.DataStorage => SensorNetwork.pda.Queries

conditional dependencies:
(4) SensorNetwork.Laptop => SensorNetwork.laptop
(5) SensorNetwork.PDA => SensorNetwork.pda
(6) SensorNetwork.DataStorage => SensorNetwork.dataNode
```

Figure 5. A sensor network SPL using network nodes and client applications. The specialized DataNode SPL itself uses a DBMS SPL.

In contrast to specialization, it might be simpler to define instance constraints; however, specialization provides better reuse because the specialized variants can also be used in other SPLs. The constraints presented here, are only *requires* or *implies* constraints but arbitrary propositional formulas might be supported [1]. Object Constraint Language (OCL) might be a good candidate for specifying the constraints [4].

Conditional Dependencies. A description of a composition of multiple SPLs using the model presented above includes all SPLs and SPL instances that might be needed in arbitrary configurations. Some of these SPLs and SPL instances, however, are only needed if features that use those SPLs are actually present in a concrete configuration. For example, if feature DATASTORAGE of the `SensorNetwork` SPL is not used, we also do not need special network nodes that store data. This means that such SPLs do not have to be considered at configuration time if we know that they are not used. We achieve this by defining constraints for attributes of SPLs. In the example shown in Figure 5, we used constraints 4–6 to define that attributes `laptop`, `pda`, and `dataNode` are only needed if a particular feature is selected. Hence, if feature DATASTORAGE is not selected, there will be no instance of SPL `DataNode`. Other SPLs that `DataNode` depends on are also not added to the configuration. In our example, this applies to SPL `DBMS` which, as a result, does not have to be configured. This is similar for SPLs that have specialized variants. These are only added to a configuration if there is an instance of the SPL itself or one of its specializations needed. For example, if there is neither an instance of `Laptop` nor of `PDA`, the whole `Client` SPL is not needed. Conditional dependencies are mappings of features to elements of the instance model. To implement such mappings we propose to use existing tools, e.g., FeatureMapper [9]. This allows us to visualize such dependencies and use views on a composition model to show only elements that belong to a particular feature or a number of features.

The Configuration Process. The configuration of a composition of SPLs is based on an instance model and the domain models of all involved SPLs. The configuration process should be supported by a configuration tool and the underlying models need not to be shown to the user. When creating a configuration, the user starts with an empty feature selection of the top-level SPL (e.g., SPL `SensorNetwork` in Fig. 5) and selects functional features. This selection is based on the domain model of an SPL but is additionally checked against constraints in the instance model. Dependent SPLs are only added when they are needed. This avoids conflicts of SPLs that are not used and simplifies the verification of a feature selection. After finishing the configuration of one SPL the configuration of dependent SPLs follows. Ideally, there is no further selection of functionality of dependent SPLs needed because it is implicitly provided via constraints; however, manual configuration of underlying SPLs might be needed or wanted.

In the following, we describe how a configuration for the SensorNetwork SPL can be derived. We start with an empty feature selection which includes instances of SPLs `SensorNode` `AccessNode` that are always needed in a network. The user selects feature DATASTORAGE which adds an instance of SPL `DataNode` (conditional dependency 6) and SPL `DBMS`. Selection of feature PDA adds an instance of SPL `PDA`. The configuration of functional features is finished with this selection and constraints have to be checked. Constraint (1) is ignored because feature LOGGING is not selected. Constraint (2) enables feature BLUETOOTH in all instances of the `DataNode` SPL and constraint (3) enables feature QUERIES in instance `pda` of SPL PDA. After that, all dependent SPLs have to be configured. This is done per SPL instance, e.g., if we would have selected clients `pda` and `laptop`, both have to be configured separately. In our case, there are no other features to be configured within dependent SPLs and the configuration is already finished. This should be the usual scenario, but there might also be a detailed configuration process of lower-level SPLs like the `DBMS` SPL.

5. Discussion and Further Work

The presented model can be used to describe SPLs that are composed from other SPLs. However, there are some issues that have to be further analyzed and can be subject to extensions of the presented model. In the following, we discuss some of these issues.

Integration with Domain Models. To provide an integrated software product line engineering process SPL instance modeling has to be integrated with domain modeling. Based on a domain model that describes constraints between features of one SPL and also constraints to other SPLs we can ease the transition to SPL instance modeling using a model transformation. The result is a basic instance model that consists of classes representing SPLs and constraints between these SPLs which represent domain constraints. Staged configuration of SPLs can also be automatically transformed into an inheritance hierarchy of classes that represent different stages of configuration. There are only a few steps to support such a transformation:

- create a class for each feature model of a composition of SPLs and
- create a hierarchy of subclasses to represent staged configuration.

Resulting classes that represent SPLs or specialized versions of an SPL can be linked to their feature model to support visualization, e.g., using a feature diagram. Since there are a number of special extensions to feature models, e.g., cardinality based feature models, a mapping of these models has to be further analyzed. In this paper, we have not clearly defined which types of constraints should be supported. There are a number of approaches to define constraints in domain models (e.g., [4, 7, 15, 1]) which should also be applicable to the presented instance model. Which approach is used, however, is not important for the presented solution in general.

Tool Support. Mapping domain models and instance models is the basis for tools that support development of such models and automates the configuration process. Further visualization support is possible by mapping features to elements in the instance model (conditional dependencies) using tools like FeatureMapper [9]. In further work, we aim at developing an integration of existing tools and an automated configuration process as part of FeatureIDE. [1] FeatureIDE is a plug-in for the Eclipse IDE, used to support the complete SPL development process. It is based on feature-oriented programming and supports domain models in the *guidsl* format [1].

Configuration Generators. As an extension to this basic tool support we want to use the presented model to derive configuration generators. These generators can be created by generating OO code from the instance model as supported by current UML tools. The model can be extended using an object-oriented language (e.g., Java) to include user-defined code. This code can include code specific to a composition technique and also code to interact with a user in the configuration process. By using an OO language for configuration we can directly access SPLs that are represented by classes and make use of polymorphism and method overriding to simplify SPL configuration. Execution of the resulting configuration generator results in an interactive configuration process for the composition of dependent SPLs.

Adaptation to the Environment. The presented approach can be applied to systems developed as SPLs where the developer has access to all subsystems (i.e., used SPLs) to configure them according to the needs of the top-level SPL. However, an SPL also interacts with its environment, i.e., the operating system, hardware, other software, etc., which usually cannot be changed. An SPL also has to be configured with respect to this external variability. Using the presented model we can also represent *external SPLs* (e.g., an operating system SPL [16]) and create constraints between the SPL of the problem domain and SPLs of the environment. These constraints have to ensure that the domain SPL configuration changes according to the environment. Providing a configuration for external SPLs as they appear in a concrete scenario (e.g., describing the actually used hardware) results in an SPL configuration that automatically adapts to this environment.

6. Related Work

There is a large amount of work addressing domain modeling and dependencies between multiple SPLs. Cardinality-based feature models with constraints were proposed by Czarnecki et al. [6, 12]. They allow a domain engineer to specify specializations and constraints in feature models where multiple selections of one feature are possible. The used *feature model references* [6] and *feature cloning* might be applicable for modeling product line instances; however, it mixes (1) domain modeling with domain implementation of a product line (handling instances of other product lines, etc.) and (2) does not provide means to create named instances of used product lines which is needed for implementation. Application product lines consuming different service-oriented product lines in a SOA environment where described by Trujillo et al. [17]. Their focus was on modeling the interfacing between SPLs in a service-oriented environment. This includes service registration and service consumption. Hence, their work is complementary to the presented approach and both might be combined in service-oriented environments. An approach that integrates feature models of different product lines was presented by Streitferdt et al. [16]. Their goal is to derive the configuration of a hardware product line based on the requirements of an SPL for embedded systems. The presented integration of multiple SPLs does not consider SPL instances or instance constraints which were not needed in their context.

In contrast to these modeling approaches, we found that feature models and dependencies between them are not sufficient to describe compositions of dependent SPLs where multiple instances of the same SPL are used. As a solution, we propose a model that describes SPL instantiation and dependencies between SPL instances. We see our approach as an extension of other product line modeling techniques and we think that their combination is needed to completely describe complex product lines that are composed with other product lines.

Product populations built from Koala components were described by van Ommering [18]. Koala components can be recursively built from smaller components leading to a set of complex products which is similar to dependent product lines described here. The focus of van Ommerings work was on interactions between components via interfaces using different connectors to support flexible component composition. Interfaces between components and their description, e.g., as defined in Koala, are also needed for safe composition when using our approach. Hence, in this respect the Koala approach is complementary to our work. Furthermore, the goal of our work is to describe compositions of SPLs independent of the implementation technique by focusing on features and dependencies between SPLs. Koala components are defined by composing smaller components at configuration time which is in contrast to our work. We aim at defining compositions of whole SPLs and not concrete components. That is, the composition of a concrete product (e.g., a component), built from other products, automatically changes depending on a feature selection, which is a modification of the composed architecture. This is different from manual composition of components to derive a larger component or a concrete product.

Fries et al. presented an approach to model SPL compositions for embedded systems [8]. They use *feature configurations* which are a selection of configured features to describe a group of instances that share this feature selection. Hence, feature configurations are similar to specialized SPLs in staged configuration; however, they do not allow a user to describe multiple configuration steps or sub-typing between specialized variants. A composition model described by Fries et al. is defined for a complete composition of product line instances. Our approach uses an instance model that is part of a product line and defines a composition of related SPLs. Each referenced SPL itself has its own instance model defining other SPLs it is composed from. Hence, we define the composition for each SPL separately which eases reuse of instance models. Furthermore, we map features to referenced SPLs and SPL instances and combine instance and feature models of multiple SPLs only when this is needed, i.e., when a feature that references another SPL is selected. This avoids any evaluation of composition rules of product lines that are not used.

Tools like *pure::variants*[2] and *Gears*[3] allow a domain engineer to build feature models and also to describe dependencies among them. Both tools support modeling of dependencies between product lines and Gears explicitly supports nested product lines that can be reused between different product lines. *guidsl* is a tool to specify composition constraints for feature models using a grammar [1]. It provides means to check models and interactively derive a configuration for a feature model.

Batory et al. have shown that SPL development using layered designs scales to *product lines of program families*. The focus of their work was on generating families of programs from a single code base and reasoning about program families. The work does

[1] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/

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

[3] http://www.biglever.com

not address relations between different product lines developed independently or between instances of such product lines.

7. Conclusion

Compositions of SPLs are used to structure and decompose large SPLs and also to reuse SPLs within other SPLs. Current feature models can be used to describe such compositions only if an SPL uses one instance of other SPLs. This is not sufficient if multiple instances of the same SPL are used in a larger SPL.

We presented an approach based on class diagrams and OOP that extends domain modeling. We provide means to model SPLs, SPL instances, their relationships, and constraints between them. In our model, *nested* or *hierarchical* SPLs, where only one instance of each involved SPL is used, are included as a special case. The presented model describes the high-level architecture of compositions of SPLs and their dependencies. We propose to use it to complement domain modeling and integrate it into the SPL development process if multiple SPLs are involved. We showed how conditional dependencies can be handled by using constraints that map features of an SPL to referenced instances of other SPLs. This serves a better understanding of compositions of dependent SPLs (e.g., supported by advanced visualization techniques) and can be used to automate the configuration process of a whole SPL composition scenario.

Acknowledgments

Marko Rosenmüller and Norbert Siegmund are funded by German Research Foundation (DFG), project number SA 465/32-1 and German Ministry of Education and Research (BMBF), project number 01IM08003C.

References

[1] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Software Product Line Conference (SPLC)*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer Verlag, 2005.

[2] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, 2004.

[3] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

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

[5] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration Through Specialization and Multi-level Configuration of Feature Models. In *Software Process Improvement and Practice 10*, pages 143–169, 2005.

[6] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged Configuration Using Feature Models. In *Proceedings of the International Software Product Line Conference (SPLC)*, volume 3154 of *Lecture Notes in Computer Science*, pages 266–283. Springer Verlag, 2004.

[7] A. Deursen and P. Klint. Domain-specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.

[8] W. Friess, J. Sincero, and W. Schroeder-Preikschat. Modelling Compositions of Modular Embedded Software Product Lines. In *Proceedings of the 25th Conference on IASTED International Multi-Conference*, pages 224–228. ACTA Press, 2007.

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

[10] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Verlag, 1997.

[12] C. H. P. Kim and K. Czarnecki. Synchronizing Cardinality-Based Feature Models and Their Specializations. In *European Conference on Model Driven Architecture Foundations and Applications (ECMDA)*, pages 331–348, 2005.

[13] C. W. Krueger. New Methods in Software Product Line Development. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 95–102. IEEE Computer Society Press, 2006.

[14] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer Verlag, 1997.

[15] D. Streitferdt, M. Riebisch, and I. Philippow. Details of Formalized Relations in Feature Models Using OCL. pages 297–304. IEEE Computer Society Press, 2003.

[16] D. Streitferdt, P. Sochos, C. Heller, and I. Philippow. Configuring Embedded System Families Using Feature Models. In *Proceedings of Net.ObjectDays*, pages 339–350. Gesellschaft für Informatik, 2005.

[17] S. Trujillo, C. Kästner, and S. Apel. Product Lines that Supply Other Product Lines: A Service-Oriented Approach. In *SPLC Workshop: Service-Oriented Architectures and Product Lines - What is the Connection?*, 2007.

[18] R. van Ommering. Building Product Populations with Software Components. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 255–265. ACM Press, 2002.

Modelling Variability in Self-Adaptive Systems: Towards a Research Agenda

A. Classen[⋆∗] A. Hubaux[⋆] F. Sanen[†] E. Truyen[†] J. Vallejos[‡]
P. Costanza[‡] W. De Meuter[‡] P. Heymans[⋆] W. Joosen[†]

[⋆]PReCISE Research Centre, Faculty of Computer Science, University of Namur
{acs,ahu,phe}@info.fundp.ac.be

[†]Distrinet Research Group, Department of Computer Science, K. U. Leuven
{frans.sanen,eddy.truyen,wouter.joosen}@cs.kuleuven.be

[‡]Programming Technology Lab, Vrije Universiteit Brussel
{jvallejo,pascal.costanza,wdmeuter}@vub.ac.be

Abstract

The combination of generative programming and component engineering applied to software product line engineering (SPLE) has focused thus far mostly on static systems (as previous editions of AOPLE indicate), with variability that is bound once. Meanwhile, an emergent paradigm in software engineering deals with self-adaptive and dynamic systems. While there is a well-known and agreed SPLE process for static systems, there has been less focus on dynamically adaptive systems. As such it appears imperative to include it in an extended research agenda.

In the present paper we observe limitations related to domain engineering in SPLE and identify what fundamental concepts, such as context *and* binding time*, must be retought in order to achieve SPLE for dynamically adaptive systems. The main contribution of this paper is a set of research questions, aimed at defining a common research agenda for addressing these limitations.*

1 Introduction

As previous editions of the AOPLE workshop indicate, the combination of generative programming and component engineering applied to software product line engineering (SPLE) has focused thus far mostly on systems with static variability binding. Meanwhile, an emergent paradigm in software engineering deals with self-adaptive systems (viz. SEAMS workshop at ICSE, or DSPL at SPLC). Self-adaptive systems are systems that are able to autonomously adapt to changing circumstances without human intervention, or, in other words, systems that are able to cope with dynamic variability. To support dynamic variability, a self-adaptive system typically implements a control loop that consists of monitoring the application and its context, analyzing the situation, planning and executing any required adaptations (as depicted in Figure 1).

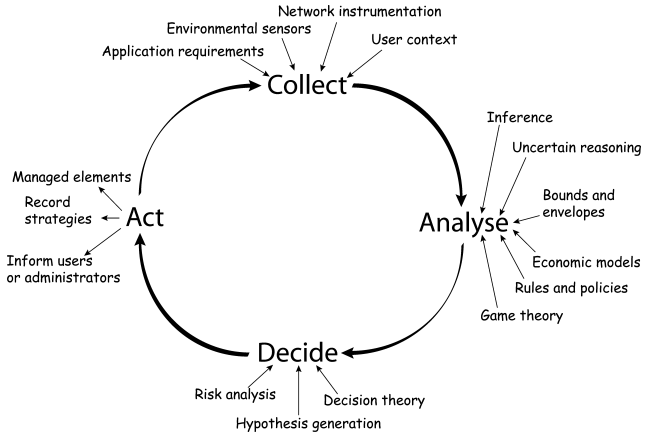


Figure 1. Typical control loop (from [6])

System-specific adaptation knowledge is used to specify when, where and how to adapt the system. This adaptation knowledge is typically specified by developers as adaptation rules following the well-known Event-Condition-Action format. For example, the Rainbow framework from Garlan et al. [8] allows to express rules like "when the response time observed by a client exceeds a well-defined threshold, and the server is overloaded, then migrate the client to a less loaded server". The control loop of a self-adaptive system is thus basically programmed as a set of such adaptation rules. Depending on the application requirements, the realization of these dynamic adaptations may be complex, and the number of adaptations may become unwieldy and therefore difficult to manage. On the

[∗]FNRS Research Fellow.

other hand, reuse of this system-specific adaptation knowledge across a family of related self-adaptive systems is possible and desirable [8].

While there is a well-known SPLE approach for static systems [19], there has been less focus on self-adaptive systems. Yet, it would be desirable to at least apply a domain engineering approach that focuses at the analysis and design of reusable adaptions. However, the engineering process to come to generic adaptations that are reusable across multiple applications is far from trivial. We believe the main reason is that there has been not much support for explicitly capturing context-dependent dynamism in variability models such as Feature Diagrams (FDs) [16]. Albeit more recent work on variability models acknowledges the concept of dynamic variability in SPLE, e.g. [18], key concepts in SPLE, such as *context* and *binding time*, are not well understood in the light of self-adaptive systems. This limits the applicability of current SPLE approaches in real adaptive scenarios.

For the purpose of this paper, we stick to the definition of *context* as *any piece of information which is computationally accessible* [11], and *binding time* as *the time at which the system is bound to a particular variant* [24]. With *static binding* we refer to the binding to a variant prior to the usage of the system, while *dynamic binding* is considered to be the binding that occurs during runtime [23].

The goal of the present paper is to take the first steps towards an extended research agenda including modelling dynamic variability as part of the domain engineering phase of SPLE: what are the most challenging problems concerning modelling dynamic variability and the fundamental concepts of context and binding time, and how these issues affect the research community. In this regard, we list six common research questions aimed at defining a common research agenda for tackling these problems with domain engineering (issues with application engineering are out of the scope of this paper). This set of questions originated from three distinct research cases that motivate the different issues related to modelling dynamic variability in self-adaptive SPLs. Most of this research is actually carried out in the context of the Belgian MoVES project,[1] fostering the collaboration among different Belgian universities.

The paper is structured as follows. Section 2 presents three different cases that each raise several issues w.r.t. modelling dynamic variability. From these separate accounts, we formulate common research questions in Section 3, which are used to bootstrap a first high-level research agenda in Section 4. Section 5 introduces related work and confronts it to our research questions. We conclude the paper in Section 6.

[1] More information at `http://prog.vub.ac.be/moves`

2 Motivating scenarios

We now elaborate on three different motivating scenarios that prove the lack of support for modelling dynamic variability in adaptive SPLs. In each motivating scenario, we discuss the issues that were experienced when performing a concrete case study as part of our research: context-aware cellphone systems, a web-based e-government application and runtime interactions in domotics systems respectively.

2.1 Context-aware cellphone systems

2.1.1 Overview

Currently, there is little explicit support for context awareness in traditional software engineering techniques and tools, which makes the development of these applications even more complex. We introduce an intelligent cell phone as an illustration of a context-aware system. Whereas traditionally a cell phone's action (e.g. to receive or to make a call) typically corresponds to a single behaviour, context-aware cell phones aim to have multiple behavioural variations associated to a single action. The choice of the appropriate variation is determined by the use context of the system. For instance, context-dependent variations can look as follows: if the battery level is low, ignore all phone calls except for contacts classified as VIP; if the user is in a meeting, redirect all calls and messages to the secretary; if there is a WiFi connection available, try to make phone calls or send messages via VoIP since this is cheaper for the user; if there is a GPRS connection available, try to send messages using TCP/IP also since this is cheaper.

2.1.2 Current achievements

In [5], we proposed the Context-Oriented Domain Analysis (CODA) model as a specialised approach for analysing, structuring, and formalising context-aware software requirements. In this work we identify a number of relationships that may exist among context-dependent adaptations. A context-dependent adaptation can include another adaptation which means that the applicability of the second adaptation is verified only if the first one is activated. Next, a context-dependent adaptation can conditionally depend on another adaptation. In this case, the applicability of the second adaptation depends on the result of the first adaptation, yielding a sequential execution. We finally introduce the notion of a choice point which is a variation point or context-dependent adaptation which has multiple adaptations associated to it. Optionally, one can associate a resolution strategy to deal with semantically interacting adaptations.

The CODA model proposes a solution that considerably differs from the existing design and programming tech-

niques to model runtime variations such as if-statements, polymorphism or design patterns. Using such techniques for context-awareness would lead to cluttered implementations (due to scattered context-dependent if-statements), combinatorial explosion of class definitions (when using polymorphism to model all the possible adaptations to the context), or to considerable infrastructural overhead (when using design patterns). By separating the system's default behaviour from the behavioural variations, the CODA model enables to make a clear distinction between the tasks of context reasoning, and dynamic binding of the variations.

2.1.3 Open research issues

Although the CODA model can help in tackling some of the challenges for modelling context-aware software, a number of challenging issues needs to be further explored.

In our approach, we assume that context-dependent variations primarily occur while the program is running. This approach gives the highest dynamicity but rises the issue on how requests that are currently being processed should be affected by the dynamic software update. A more exhaustive analysis should be done to determine what exactly should occur at runtime (context acquisition, context reasoning, variation binding, etc.) and what can be derived to earlier stages of the software development. The decision as to when the behaviour should vary has an impact on the design of the system (use of dedicated design patterns to achieve variability) and the choice of technology (implementation platform, programming language paradigm, execution environment).

Experience has pointed out that the evolution of relationships among context-dependent adaptations in the CODA model is error-prone since contradictions and cases of under-specification might seep into the specifications. This is mainly caused by the fact that a new adaptation can possibly interact with all existing adaptations in the system. Nevertheless, the solution for this issue is not to add extra information to the CODA diagram as this can dramatically diminish its understandability.

2.2 Web based eGovernment application

2.2.1 Overview

PloneGov is an open source project fostering the development of web based *eGovernment* applications and gathering around 55 international public organizations into 19 products [1]. All these products promote the cooperative development of applications and web sites targeted to public organizations and their citizens. The worldwide scope of PloneGov yields specific legal, social, political or linguistic aspects to deal with. All constrain the features required from a given product, hence the need for flexibility regarding product derivation. For now, we focus on one of PloneGov's products, namely PloneMeeting, a Belgian government-initiated project offering advanced meeting management functionalities to national authorities.

Central to PloneMeeting is the concept of *meeting* itself. The allowed states of a meeting are defined according to a workflow, which can be changed in the configuration. There is, however, no restriction as of when such changes may be made, e.g. *installation* time or *runtime*. Yet, changing the workflow at runtime might result in an inconsistent system since the states of already existing meetings might not be compatible with those of the newly selected workflow.

The Plone internationalisation initiative intends to provide a flexible mechanism to manage language selection and display. The so-called PlacelessTranslationService (PTS) is Plone's built-in translation management service. The PTS uses the language of the web browser to automatically determine the display language of the pages.

2.2.2 Current achievements

In previous work [4], we introduced the idea of using SPL principles to engineer the PloneGov project. Our conclusion showed a number of organisational and technical problems that had to be tackled. such as handling the distributed developers and managing the already existing variability.

In [14], we focused on the identification and modelling of the variability in PloneMeeting. Since no variability model formerly existed, the variation points had to be reverse engineered from stakeholders, developers and existing artefacts to enable the re-engineering of configurable artefacts. We therefore defined a reverse engineering process taking these various information sources as input and producing separate FDs for the different concerns we identified.

The most significant results we obtained so far are four modelling challenges identified during the variability reverse engineering of PloneMeeting [13]. The first one refers to the implicit modelling viewpoint underlying the variability modelling. The second one discusses the modelling of contextual elements whose availability is unpredictable. The third one focuses on the consistency between the FD and its constraints as they both evolve over time. The fourth one addresses the representation of large sets of features in a FD. The workarounds we proposed to tackle these issues still have to be systematically applied to concrete cases and properly assessed.

2.2.3 Open research issues

Apart from the workflow selection and browser-dependent translation aspect of Plone, the dynamic side of PloneMeeting has been disregarded is recent work. Although being

a rather static application, PloneMeeting still exhibits some dynamic, typically runtime, configuration alternatives.

As the language selection scenario shows, the changing environment requires some extra flexibility from the system to keep it displaying pages flawlessly. Since each web browser encodes the language differently, elicitating, scoping and modelling this contextual information is our first issue. Secondly, a recurrent issue was the classification of binding times and the identification of the proper time granularity. Although extensively covered in mainstream literature [12], solutions to these issues are still fragmentary. Thirdly, a more practical research issue was the selection of the most suitable means to identify and mark dynamically configurable variation points. Finally, we struggled to express runtime constraints conditioning the evolution of product configurations. Further investigations will tell whether existing solutions are comprehensive enough.

2.3 Runtime interactions in domotics systems

2.3.1 Overview

Domotics systems typically combine a wide range of features in the area of home control, home security, communications, personal information, health, etc. We will motivate the relevance of runtime context information when modelling interactions by exploring two scenario's for protecting the housing environment. The first scenario concerns a *fire control* feature that turns on some sprinklers during a fire and a *flood control* feature which shuts off the water main to the home during a flood. Turning the sprinklers on during a fire and flooding the basement before the fire is under control results in the house further burning down. The second scenario involves a *presence simulation* feature that turns lights on and off to simulate the presence of the house occupants and a *doorkeeper* feature which controls the access to the house and allows occupants to talk to the visitor. Obviously, we would like the doorkeeper not to give away the fact that nobody is home if there is an unidentified person in front of the door in order to prevent the house owners from a burglary. However, if the person can be identified and trusted, there aren't any problems. Both the basement being flooded and the fact if a person can be identified or not are conditions that are only available at runtime.

2.3.2 Current achievements

Domotics systems already have been introduced in a product line context elsewhere by Kang et al. [17]. What is missing in their FDs is that the dependencies and other relationships between features cannot be expressed in terms of runtime context information. As a result, runtime behavioural feature interactions (FIs) caused by runtime variability cannot be modelled. A *behavioural FI* is a situation where a feature that works correctly in isolation does not work correctly anymore when it is combined with other features. The fact that an interaction only might or might not occur depending on the runtime context at hand makes it a *runtime behavioural FI*.

In previous work [20], we proposed a conceptual model to enable the management of interactions so that knowledge about interactions can be shared and used in the course of system evolution. An important part of the conceptual model consisted of a concern hierarchy that resembles the feature hierarchy in FDs. In this model, we already introduced the notion of a *condition* to take into account certain runtime circumstances. A shortcoming of current FD approaches is the lack of support for modelling exactly this runtime context information. To the best of our knowledge, no appropriate formalism or methodology exists to reify information about runtime behavioural FIs, reason about them and enforce resolutions. A number of extensions to FODA have been proposed to express more complex feature relations, e.g. using propositional logic. However, we are not convinced that propositional logic can distinguish between all possible interpretations of runtime behavioural FIs. In [21], we argument for instance that traditional logic is not suited to represent the fire and flood control FI.

It is also important to realise that traditionally used mechanisms, such as e.g. prioritisation, are not always feasible for representing runtime behavioural FIs. Next to the fact that an interaction between two features with the same priority cannot be resolved, the relative priority of two features to one another can be different in varying circumstances. The latter is illustrated by the second domotics scenario from above where everything depends on the result of identifying the visitor.

2.3.3 Open research issues

Based on this need for modelling runtime context information relevant for FIs, we can identify the following open research issues. First of all, although it is easy to come up with the relevant context information for particular FIs, answering the question what context to model is a non-trivial problem. Secondly, we need to decide on how to model runtime context information. Coming up with a widely applicable methodology for modelling runtime context information poses an interesting challenge. One of the usual suspects here are propositional logic, but are these sufficient to express the possibly complex relationships including runtime context information? Moreover, it is not clear how we can express more complex interactions involving more than two features. Finally, we need to ask ourselves if this runtime context information should be part of the FD itself or should be specified in a separate, dedicated modelling lan-

guage, complementary to the FD. In the latter case, an obvious need arises for traceability links between the different models. For now, we want to leave this open for discussion (cfr. RQ2). Either way, we are convinced that this kind of knowledge is an important form of information that can be (re)used to manage runtime variability and therefore should be modelled.

3 Common research questions

After having presented three distinct cases, each of which identifies different modelling problems, we will derive from them a number of crosscutting research questions. The goal of the present section is thus to formulate a common research focus for the coauthors of this paper.

RQ1: How to determine what context drives dynamic change? To the best of our knowledge, there is currently no methodological support to determine which are the context variables that will have an influence on dynamic change and to determine the course of action to deal with a change of these context variables. For instance, in case of the mobile phone example (Section 2.1), the decision of whether the low battery level is a trigger for forwarding phone calls could reasonably be made by a project manager or developer. The decision that VIP phone calls should be able to circumvent this, however, needs to be taken on a higher level (it requires an infrastructure that lets users decide who VIPs are). Furthermore, the need for additional context information might emerge from the combination of several features, as shown by the presence simulation and doorkeeper features in the domotics example (Section 2.3).

Such a methodology could be inspired, for instance, by Kang et al. FODA's *context analysis* [16]. It could also be based on the Problem Frames approach [15], which aims at identifying physical context given a requirement, or the KAOS method [25] whose goal is to elicit requirements based on high-level goals. The output of this methodology would be (i) a set of relevant context info specifying what elements in the environment of the system are relevant, (ii) constraints specifying when adaptations must be performed (due to changes in the context), (iii) concrete actions specifying what adaptations must be taken (to deal with the contextual changes).

RQ2: How to explicitly model context-dependent adaptations and how to compose it with the FD? Part of this question is whether or not the context information and adaptation knowledge uncovered in the methodology needs to be incorporated into the FD. In the case of CODA, for instance, it appears that including all context information in the FD leads to a highly complex and unwieldy diagram. A more scalable approach might be to model this kind of information in separate diagrams and to trace them to the features that are concerned. At the same time, it might appear natural to consider environment variables such as *battery level low* in the FD.

RQ3: How do non-functional concerns constrain the execution of context-dependent adaptations? Performing dynamic adaptations should preserve the non-functional properties of the system. For instance, in the event of a dynamic change, the structural integrity and global state-consistency of the system have to be ensured. Other non-functional properties of interest are reliability, correctness or efficiency. In the scenario of PloneMeeting, for instance, an issue is how to *reliably* change a running workflow. In this case, we need to specify when it is safe to change a workflow so that it remains compatible with the workflows of already existing meetings. In the scenario of the context-aware cellphone, several variations may apply for the context conditions in which the phone calls are received or made, and therefore a *correct* integration between the variations and the cellphone's base behaviour should be ensured. Other type of constraints involve taking into account the dependencies and conflicts between different context-dependent adaptations

RQ4: How to specify constraints in order to avoid under-specification? Given constraints (e.g. binding time and runtime behavioural interaction constraints) have to be expressed and formalised in some way. Take, for instance, the interaction between the fire control and flood control features in the domotics system. With current constraint languages for FDs, we found it hard to express this kind of constraint in order to capture it and/or process it later. It is difficult to capture all possible context combinations in a generic way (instead of enumerating all context combinations) or referring to context information at all.

RQ5: How to map domain models to implementation-specific elements? Given a suitable notation for expressing constraints (see RQ2 & RQ3), these constraints are ideally expressed at a level that makes abstraction of specific context info related to a particular implementation technology. For example, in the case of PloneGov (Section 2.2), there are many different ways in which a browser communicates the user language to the web server. At the level of domain analysis and domain design, however, one would like to identify and reason about the desired user language as a context element that is independent from the particular implementation technology. Yet during domain implementation, a specification is needed that maps this abstract context element to the appropriate browser-specific information.

RQ6: How to classify context-dependent adaptations according to their context and binding-time? As our three cases from above already indicate, there is seemingly no consensus as to what different types of context-dependent adaptations can be supported by a self-adaptive SPL. One of the key issues here is that the relation between

the three concepts, that are at the heart of what we call self-adaptive SPLs, namely "dynamism", context and binding time is not clear. And so, it seems imperative at some point to classify the different types of adaptation, primarily, by their suitable binding times and contexts.

4 Towards a research agenda

Having stated the research questions, let us examine how they affect the classical SPLE process by Pohl et al. [19]. This leads us to a more concrete research agenda. The well-accepted SPLE process, consists of a domain engineering and an application engineering phase. In the domain engineering phase, the scope of the product line is decided, and a set of customisable components developed. The application engineering phase exists for each product that is to be delivered. Following a requirement analysis, it starts by configuring the product, i.e. deciding what goes into the product, and ends with integrating and testing it.

By mapping the research questions to the SPLE process, we are able to identify a set of concrete objectives that must be achieved in order to realise a suitable SPLE approach for self-adaptive systems. Before we proceed, note that an SPLE approach for self-adaptive systems is defined as an extension and not as a replacement of a classical SPLE approach. For example the design of reusable components that make up the technical infrastructure of self-adaptive system (sensors, effectors, monitors, planners,..) remains largely the same. The extension that a self-adaptive SPLE approach brings focuses mostly on the domain engineering of the control loop in a family of self-adaptive systems:

RQ1 To implement RQ1, one would need to extend the domain engineering phase by (i) a *context scoping activity*, that decides what part of the context must be monitored; and by (ii) a *context modelling activity* that explicitly captures the essence of the context-dependent adaptations in a model, so that it can be referred to.

RQ2 Complementary to RQ1, addressing RQ2 would lead to a *scalable structure* for relating the context-dependent adaptation knowledge to standard feature models.

RQ3 Addressing RQ3 would involve a *quality attribute analysis* [2] to determine the important non-functional requirements (performance, reliability, ...) and to identify *reconfiguration tactics* that aim at preserving these quality attributes in the presence of dynamic adaptations.

RQ4 We expect that the outcome of RQ4 will lead to the *selection or the improvement of existing modelling and constraint languages* supporting constraint specification at the different stages of the context modelling and integration process.

RQ5 In order to address RQ5, a *translation infrastructure* is necessary that bridges the gap between the concepts of the context models elicited during analysis, and the concrete artefacts of the implementation. Generally speaking, this translation infrastructure involves the mapping from the context models to system-specific sensors and actuators, but also involves connecting context elements to specific state properties of software components.

RQ6 This RQ is rather of conceptual nature, we hope that it will lead to a better understanding of key concepts and to a clearer terminology. It thus affects the whole SPLE process, albeit indirectly.

The application engineering process also needs to be extended with activities involving the analysis of required context-dependent adaptations and the configuration, integration and testing of these adaptations into a fully operational control loop. But as stated in the introduction, this paper has focused mostly on domain engineering, and leaves the study of issues with application engineering for future work.

5 Related work

Cheng *et al.* [3] propose a research roadmap focusing on the requirements, modelling, engineering and assurance activities of self-adaptive systems. For each of them, the inadequacy of existing techniques to support the development of reliable self-adaptive systems and the challenges ahead are systematically formulated. Out of their study, they notably conclude that the design time and runtime processes can no longer be dealt with separately and advocate SPLE as a possible opportunity to drive the development of self-adaptive systems. All the research questions we set forth go along the same line of research by further precising the issues self-adaptivity raises in SPLE.

Fernandes *et al.* [7] present UbiFEX, a modelling notation extending existing feature diagram languages with contextual information. The general feature model generated with UbiFEX is composed of a feature model, a context feature model with the associated activation expressions, and context rules binding the context and feature models together. UbiFEX also comes with a simulation tool checking the consistency of the produced models.

Desmet *et al.* [5], propose the Context-Oriented Domain Analysis (CODA) which is heavily inspired by the original Feature-Oriented Domain Analysis (FODA) [16] used in product-line development. It enforces software engineers to think of context-aware systems as pieces of basic context-unaware behaviour which can be further refined by means of context-dependent adaptations at certain variation points. A context-dependent adaptation is a unit of behaviour that

adapts a subpart of a software system only if a certain context condition is satisfied. Both this work and the one from Fernandes *et al.* should provide valuable solution elements to RQ2 and RQ3.

Hartmann *et al.* [10] introduce context variability models (CVM) to represent high-level context information of software supply chains. A CVM is a general classifier of the context of products that is expressed in a FODA like notation. The combination of the CVM and the SPL feature model results in a Multiple Product Line Feature Model where the dependencies between both models are expressed with `requires` and `excludes` links. They do not explicitly present their work as suited to self-adaptive or dynamic systems. Once adopted, the context configuration choices are immutable and do not lead to self-adaptive behaviours. Conversely, Desmet *et al.* and Hartmann *et al.* [7, 10] consider the dynamic evolution of the context and its impact on the model. The relevance of this work to our current research will therefore require further evaluations.

Lee *et al.* [18] propose an approach grouping features in binding units which are assigned binding times and binding states used to constrain the SPL configuration in a dynamic context. Their framework also provides a product reconfiguration process encompassing a context analysis, a reconfiguration strategy selection and a reconfiguration implementation phase. Their solution might notably offer means to clarify the issues outlined in RQ6.

Gomaa *et al.* [9] present a solution based on reconfiguration patterns for dynamic reconfiguration of software product families. Their reconfiguration patterns are based on UML collaboration and state diagrams. They focus on components and do not model contextual information nor provide explicit links with variability models. Nevertheless, their approach might be part of a solution to RQ5.

In contrast, Schmid *et al.* [22] propose a taxonomy of issues that can arise when migrating a system from development time (static) to runtime (dynamic) variability. They analyse the impact of dynamicity on the input of the processes, the processes themselves and the output of the processes, and delineate the required capabilities of the code base for each of them. Such a taxonomy might help us investigating solutions to RQ5.

Although promising, these solutions still call for systematic assessments and need to be augmented with thorough guidelines covering the steps going from the context scoping down to the implementation of self-adaptive SPLs. We hope the outcome of this analysis will generate meaningful results that will help us answering RQ1 and RQ4.

Finally, different research projects indicate the relevance of defining a roadmap when it comes to investigating dynamic variability. DiVA[2] will combine aspect-oriented and model-driven techniques in an innovative way to provide a new tool-supported methodology with an integrated framework for managing dynamic variability in adaptive systems. Their basic idea is to use models at both the design time and runtime level to manage dynamic variability in combination with aspect-oriented modeling techniques in order to tackle the issue of the combinatorial explosion of variants. Model-driven techniques are then used to automate and improve the creation of (re)configuration logic. The MUSIC project[3] is a European project intending to offer an open platform for the development of self-adaptive mobile applications. Among the expected results of this project are a methodology, tools and a middleware suited for software developers. MUSIC builds further on the results of the MADAM project[4] in which adaptation requirements of mobile applications were studied and a theory of adaptation was developed. A set of reusable adaptation strategies and adaptation mechanisms, based on dynamically reconfigurable component architecture was one of their main results. Compared to DiVA, the main variability mechanism in these two projects consists in loading different implementations for each component type of the architecture. The idea of the AMPLE project[5] is to holistically treat variability at each lifecycle stage by combining aspect-orientation and model-driven techniques (similar to DiVA). Obviously, different implementation possibilities exist for binding variation points in various development stages, e.g. at design, development, deployment or even at runtime. Implementation artefacts will not only include traditional program code but also runtime configuration and domain specific languages. Therefore, one of the most promising results will be AMPLE's variability framework with integrated tool support for each lifecycle stage. The careful study of the ongoing research in these projects appears to be imperative to evaluate the sustainability of the coming results of our research.

6 Conclusion

SPLE process support for dynamically adaptive systems is fragmented, although a well-known SPLE process for static systems already exists. Therefore, the main contribution of this paper is our list of six common research questions indicating limitations we observed related to modelling variability in self-adaptive SPLs and identifying the need for clarification of fundamental concepts such as context and binding time. Starting from this set of research questions, we defined a high-level research agenda in which we discuss the needed enhancements to the traditional SPLE process in order to achieve SPLE for dynamically adaptive systems.

[2] `www.ict-diva.eu`

[3] `www.ist-music.eu/MUSIC/about-music`

[4] `www.ist-music.eu/MUSIC/madam-project`

[5] `ample.holos.pt`

Acknowledgements

This work was partially funded by the Interuniversity Attraction Poles Programme, Belgian State, Belgian Science Policy, the FNRS, and the VariBru project of the ICT Impulse Programme of the Institute for the encouragement of Scientific Research and Innovation of Brussels (ISRIB).

References

[1] PloneGov. http://www.plonegov.org/.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley.

[3] B. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors. *Software Engineering for Self-Adaptive Systems, 13.1. - 18.1.2008*, volume 08031 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2008.

[4] G. Delannay, K. Mens, P. Heymans, P.-Y. Schobbens, and J.-M. Zeippen. PloneGov as an Open Source Product Line. In *Workshop on Open Source Software and Product Lines (OSSPL07), collocated with SPLC*, 2007.

[5] B. Desmet, J. Vallejos, P. Costanza, W. De Meuter, and T. D'Hondt. Context-Oriented Domain Analysis. In *6th International and Interdisciplinary Conference on Modeling and Using Context (CONTEXT 2007)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, August 2007.

[6] S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli. A survey of autonomic communications. *ACM Trans. Auton. Adapt. Syst.*, 1(2):223–259, 2006.

[7] P. Fernandes and C. Werner. Ubifex: Modeling context-aware software product lines. In *2nd International Workshop on Dynamic Software Product Line Conference*, Limerick, Ireland, 2008.

[8] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10):46–54, 2004.

[9] H. Gomaa and M. Hussein. Dynamic software reconfiguration in software product families. In *Software Product-Family Engineering*, Lecture Notes in Computer Science, 2004.

[10] H. Hartmann and T. Trew. Using feature diagrams with context variability to model multiple product lines for software supply chains. In *12th International Software Product Line Conference*. IEEE Computer Society, 2008.

[11] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-Oriented Programming. *Journal of Object Technology.* `http://www.jot.fm`, 7(3), March-April 2008.

[12] A. Hubaux and A. Classen. Taming time in software product lines. Technical Report Draft Version, University of Namur, June 2008.

[13] A. Hubaux, P. Heymans, and D. Benavides. Variability modelling challenges from the trenches of an open source product line re-engineering project. In *Software Product Line Conference (SPLC'08)*, 2008. To appear.

[14] A. Hubaux, P. Heymans, and H. Unphon. Separating Variability Concerns in a Product Line Re-Engineering Project. In *International workshop on Early Aspects at AOSD*, 2008.

[15] M. A. Jackson. *Problem frames: analyzing and structuring software development problems*. Addison-Wesley, Boston, MA, USA, 2001.

[16] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, CMU, November 1990.

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

[18] J. Lee and K. C. Kang. A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In *10th International Software Product Line Conference*, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[19] K. Pohl, G. Bockle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, July 2005.

[20] F. Sanen, E. Truyen, and W. Joosen. Managing concern interactions in middleware. In J. Indulska and K. Raymond, editors, *7th International Conference on Distributed Applications and Interoperable Systems*, volume 4531 of *Lecture Notes in Computer Science*, pages 267–283. Springer, 2007.

[21] F. Sanen, E. Truyen, and W. Joosen. Modeling context-dependent aspect interference using default logics. In *5th Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAMSE)*, 2008.

[22] K. Schmid and H. Eichelberger. From static to dynamic software product lines. In *2nd International Workshop on Dynamic Software Product Line Conference*, Limerick, Ireland, 2008.

[23] K. Schmid and H. Eichelberger. Model-Based Implementation of Meta-Variability Constructs: A Case Study using Aspects. In *Proceedings of VAMOS 2008*, pages 63–71, Essen, January 2008.

[24] M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques. *Software – Practice and Experience*, 35(8):705–754, 2005.

[25] A. van Lamsweerde. Goal-oriented requirements enginering: A roundtrip from research to practice. In *RE '04: Proceedings of the Requirements Engineering Conference, 12th IEEE International*, pages 4–7, Washington, DC, USA, 2004. IEEE Computer Society.

Features as First-class Entities – Toward a Better Representation of Features

Sagar Sunkle, Marko Rosenmüller,
Norbert Siegmund,
Syed Saif ur Rahman, Gunter Saake
School of Computer Science
University of Magdeburg, Germany
{sagar.sunkle,rosenmue,norbert.siegmund,
srahman,saake}@iti.cs.uni-
magdeburg.de

Sven Apel
Dept. of Informatics and Mathematics
University of Passau, Germany
apel@uni-passau.de

ABSTRACT

Features are distinguishable characteristics of a system relevant to some stakeholder. A product line is a set of products that differ in terms of features. Features do not have first-class status in contemporary programming languages (PLs). We argue that various problems related to features are a result of this abstraction and representation mismatch and that features should be elevated to a first-class status. We propose an extension to Java that implements features as first-class entities. We give examples of the syntax and semantics of this extension and explain how the new representation can handle features better.

General Terms

Languages, Design

Keywords

Feature-Oriented Programming, JastAdd, Separation of Concerns

1. INTRODUCTION

Separation of concerns is one of the most important principles in software engineering [16]. Abstractions like features and classes are viewed as dimensions in concern space [36]. Separation of concerns means decomposing software into manageable pieces along a dimension in concern space. It consists of *identification*, *encapsulation*, and *integration*. *Identification* means a software is decomposed into entities that represent the abstraction, e.g., classes and features, *encapsulation* means some mechanism is provided so that these entities can be manipulated as a first-class entities [32], *and integration* means that some composition mechanism is provided that integrates concerns represented as first-class entities[36]. The first-class status of an entity in a programming language (PL) indicates the degree to which one can address or manipulate concepts in a given domain arranged along the dimensions of a concern space and the ease with which this is made possible in a given PL [32].

A feature is defined as an end-user-visible characteristic of a system, or a distinguishable characteristic of a concept (system, component, and so on) that is relevant to some stakeholder [14]. A product line contains different products that vary in features. Consequently, features are used to understand the commonalities (shared features) and variabilities (optional or unshared features) between the products of a product line.

Many technologies have been used to implement features [3, 4, 9, 28, 31, 38]. The main kind of concern supported by them is one of functions, classes, aspects, hyperslices, mixins, and frames, etc. Features, which are themselves a kind of concern, are essentially implemented in terms of entities that basically represent some other kind of concern. Instead of thinking only about features, the developer has to organize them in terms of the modular structure of the approach he is using and see to it that the intent of features is precisely represented by the entities in this approach. We take the position that this abstraction and representation mismatch causes problems [26, 29] such as, e.g., hierarchical misalignments, limitations in feature composition and order, and inexpressive program deltas, etc. Our claim is that such problems can be addressed and various possibilities for features can be achieved more easily if features were represented not in terms of other entities, but as first-class entities themselves.

In this position paper, we propose to represent features as first-class entities. We discuss what it means when certain programmatic entities have first-class status in a given PL. We review various feature implementation approaches and enumerate related problems. We argue that these problems arise due to an inadequate representation of features. We put forward an agenda that establishes features as first-class entities. Finally,

we propose an implementation of the extension and discuss how features implemented as first-class entities can be used to address the problems.

2. BACKGROUND

2.1 First-class Entities in PLs

There is no specific definition for first-class status of entities in a given PL. Certain properties have been observed that indicate a first-class status of a given programmatic entity [13, 35]. We deem the following five properties as the defining properties that must be exhibited by entities in a given PL to be called first-class entities.

1. First-class entities can be instantiated at compile-time or run-time and possibly other stages of program execution.
2. First-class entities can be stored in variables and data structures.
3. First-class entities can be statically or dynamically typed, thus allowing compile-time or run-time structural manipulation.
4. First-class entities can be passed as parameters to other program elements such as methods and returned from methods.
5. First-class entities can be part of various expressions and statements in this PL, giving a program developer ample options to represent his intent in representing the problem domain.

Various PLs that claim first-class status for a kind of concern, support different subsets of these properties differing in their semantic treatment. The degree of manipulation of first-class entities may depend on the kind of typing and the kind of composition supported by given PLs. Runtime manipulation of such entities creates new possibilities. In this case, such entities can have identity and be aware of other entities of the same kind. This makes it possible to represent and manipulate interactions among these entities more naturally. Also, such entities can store context and be aware of the state of a program, thus making possible changes at wider range of stages in the program. These two properties are indicative of reflective and meta programming support for the first-class entities. They depend on the reflection and meta programming support of a given PL and may increase the degree of manipulation substantially for the first-class entities.

2.2 Features

In feature-oriented domain analysis (FODA) [22], features are organized in feature diagrams. A feature diagram is a tree with the root representing a concept and its descendant nodes being features. These features can be mandatory, optional, or alternative. Feature-oriented decomposition is a feature modeling activity used to capture commonalities and variabilities in terms of features, of systems in a domain [14]. It is used to model a domain in terms of features from the ground up. Feature-oriented refactoring is the process of decomposing an already existing system to a system exposing features [25].

2.3 Feature Implementations

Features as a programming model was first conceived by Prehofer [33], citing the rationale behind using features to be the flexible composition of objects possible from a set of features. The implementation technique for FODA is broadly referred to as feature-oriented programming [9, 10], but as asserted earlier, there are many ways in which features can be implemented. Kästner et al. [24] distinguish between compositional and annotative approaches. The same distinction can also be applied to various feature implementation approaches. Compositional approaches for implementing features represent features as distinct modules, which are composed at compile time or deployment time or similar. Examples of compositional approaches are mixin layers [5], HyperJ hyperslices [32], and Scala traits [31]. The *ifdef* statements in C, frames in XVCL [38] and color annotations in CIDE [38] are, on the other hand, examples of annotative approaches. Annotative approaches implement features by identifying code belonging to a feature in the source and annotating it, so that variants may be created by including or removing annotated code from the source [24].

The compositional approaches generally allow coarse-grained refinements to programs due to the fact that naming schemes of container entities such as classes and methods are required to be kept invariant as they are used in identifying parts of the program to which refinement must be applied. They are not suitable for fine-grained refinements in which order of statements or expressions added by features needs to be controlled [24]. Fine-grained refinements are possible with the annotative approaches. Annotative approaches allow refinements of arbitrary granularity since the important concerns of compositional approaches like naming schemes and order of composed code do not matter as all code fragments belonging to features are at their final position and only need to be annotated [24].

We propose that, by using a combination of compositional and annotative approaches, we can create a better representation of features. In the following section, we address the problems faced by feature-oriented approaches in general and then state the proposed solution which uses elements from compositional and annotative approaches to tackle these problems.

3. THE PROBLEM

Mezini and Ostermann [29] identified weaknesses of various current feature-oriented approaches in managing the variability in product lines. Similarly, Lopez-Herrejon et al. [26] evaluated support for features in advanced modularization technologies and concluded that despite the crucial importance of features, features are rarely completely modularized. The shortcomings described below are not necessarily present in all the approaches considered, but none of the approaches provides a uniform treatment of the various shortcomings either. The weaknesses of various current feature-oriented approaches identified in [5, 26, 29] follow:

- ***Hierarchical refinements*** – Features are implemented as refinements to base classes. Mezini and Ostermann [29] claim that this is a shortcoming, because the hierarchical modularity of the refinements to the base classes imposes a structure on features which are not in hierarchical relationship to each other. The problem with this is that for a given feature, there may not be a class in one-to-one relation to which this feature may be mapped. For further details refer to [3], [5], and [29]. Similarly, because features are refinements, a feature that is in fact reusable, would need to be encoded separately as a refinement to each class that needs it. This makes reuse of common features hard [29].
- ***Feature composition and feature order*** – Feature modules should be composable in different orders and should follow the commutativity or pseudo-commutativity of features [1]. Feature composition should be closed under composition, which means that features may be grouped to larger features and such a composite feature is valid wherever the constituents features are used [26] as this increases the reuse of features. Different approaches support either or both of these properties. Even in those approaches that support both closed composition and feature order, actually implementing it can be a nontrivial task [26].
- ***Program deltas*** – Various program refinements are deltas with respect to the base program [26]. New classes, interfaces, fields, method statements, and method arguments, etc., are examples of program deltas. Considering features as semantic blocks of code, preferably any statement or expression, or group of statements and expressions in a given programming language can be part of the refinements a feature makes. The order of blocks of code to be inserted into a method for example, cannot be controlled in simple method refinement approaches.
- ***Type support for features*** – Feature modules and composite modules should be well defined via type support. Types for features can be extremely beneficial not only in safe composition of features, but also in controlling interactions among features and also between features and the regular types in a given programming language [23, 37]. But types for features have been treated in isolation, e.g., it is not known how features represented as types will fare in dynamic composition. Other treatments of types for features consider only an extension to a subset of Java [2]. Type checking or similar concepts are difficult to apply to features because it requires some way of identifying and localizing feature code and representing features as types that interact with programming language types. By expressing a type checking mechanism for features as a calculus language that interfaces with a feature description language, type checking may be more clearly applied to an entire programming language [2, 23].
- ***Dynamic composition and separate compilation*** – It should be possible to alter the configuration of features which have already been instantiated [29]. Similarly, it should be possible to bind features dynamically based on conditions related to specific expressions [21] and this must happen considering the performance of application that uses such dynamic reconfiguration of features. Some approaches have been suggested for dynamic composition of features [29, 34], but dynamic composition remains a largely unexplored issue in other feature-oriented approaches. Separate compilation of features is also desirable for better debugging of feature implementation and distribution of byte code [26].

Feature implementations also lack a common ground with feature modeling concepts. In order to use features for creating program variants, some sort of structure has to be imposed on them. Such structure indicates the relationship between features, their grouping into different collections and imposes certain constraints about which choices of features are valid. Though, in current feature-oriented approaches, no programmatic or language level mechanisms are provided for it. Below we summarize the problem and then describe the proposed solution in the next section.

Problem Summary

Feature implementations either weave refinement code based on a naming scheme or employ some sort of redirection or delegation mechanism for executing feature related code. Features cannot be stored in variables or data structures, neither can they be used in pure Java code. Features are not aware of their or other features'

contents via some sort of interface, consequently their interactions cannot be easily modeled. The general shortcomings of various feature-oriented approaches indicate in a way also the desirable properties of a feature implementation which should be considered in concert instead of providing support for only some of them. We propose to rectify this situation by providing a better representation of features as well as combining the feature modeling concepts of product lines, for a complete feature based software solution.

4. SOLUTION PROPOSED

We propose a feature extension to a programming language under consideration. This extension will have two parts, one as a feature and feature models description language and the other as the feature development and refactoring language used to manipulate code and program fragments. In case of Java, the first part can be expressed as an embedded domain specific language [18]. The second part, i.e, the feature development and refactoring language can be implemented as an extension of the Java syntax and semantics in accordance with the first part. In the following, we establish an agenda for features as first-class entities.

1. The kind of features (such as mandatory and optional), parent-child relations (such as AND,OR, and alternative) about features and constraints between features should be expressible in the extension.

2. Features should be represented as types and interaction between features and regular types should be controlled. The mechanisms of feature normalization and conversion to disjunctive normal form for finding valid feature instances [15] could be coupled with the meta information about features in the programs to compose safe variants. We propose to implement the composition core based on feature algebra [6].

3. It should be possible for features to contain classes and various class members. It should also be possible for classes to contain feature annotations. Such a representation would gain from both compositional and annotative syntax. Coarse-grained program deltas can be represented in a compositional manner while fine-grained deltas can be represented by annotative syntax.

4. Feature models should be expressed adequately in the extension. A feature model should be modifiable at runtime, reflecting in a changed program variant. Reification, i.e., storing information about a feature such as the container entity of a code fragment, can be used to create altered feature variants at runtime. Changing a feature model may entail removing a child feature from a parent feature, relocating it elsewhere or remove it entirely. All such changes need to be supported with the above mechanism.

5. A program delta that is refined by some feature may be required by other features. This information should be expressible at language level so that the choice of creating a variant with altered code or creating variants in which one variant contains the original code and the other variant contains altered code remains with the user.

The above indicates that a mechanism for encapsulating various code fragments that constitute a feature in a programmatic entity should be available. If operations were available on such an entity for code composition as well as product line customization, then a direct correspondence can be established from features at the modeling level and implemented features and both could be manipulated with precise control.

5. IMPLEMENTATION DETAILS

In the following we show how a feature extension can be implemented in Java.

5.1 JastAdd

We propose to use JastAdd[1] which is a Java based compiler construction system [19]. We choose JastAdd because it implements Java 1.4 and 1.5 in terms of modular compiler analyses [17]. JastAdd considers an object-oriented abstract syntax tree (AST) as the basis for language design. It uses AspectJ introductions to add behavior to various classes representing language constructs [17, 19]. Behavior can be added to AST nodes both in a declarative and imperative manner using the extended versions of synthesized and inherited attributes. The declarative specification ensures internally that attributes and analyses need not be ordered by the programmer. Different transformations can be applied to an AST in terms of attributes and an AST can be prepared as required [30].

5.2 Syntax and Semantics of the Proposed Extension

In the following, we give some examples of syntax and semantics of the proposed extension. Consider the feature diagram for a Graph Product Line (GPL)[27], shown in Figure 1.

A feature model representing a product line is declared using the keyword productLine (Figure 2). Figure 2 shows the feature description for the feature model shown in Figure 1. The one, more and all operators in Figure 2 indicate the alternative, the OR and the

[1]http://jastadd.org/the-jastadd-extensible-java-compiler

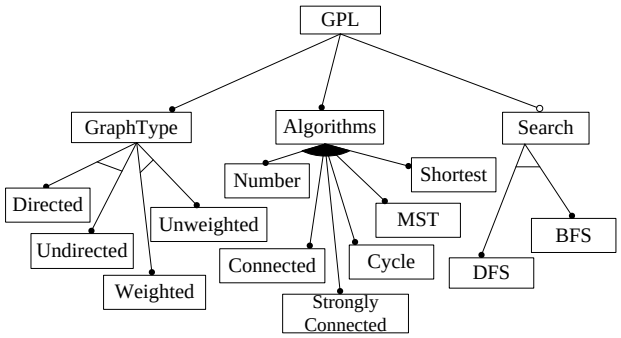


Figure 1: Graph Product Line

```
1 productLine GPL {
2  GraphType : all(one(Directed, Undirected),
3                  one(Weighted, Unweighted))
4  Search     : one(DFS, BFS)
5  Algorithms: more(Number, Connected,
6                   StronglyConnected, Cycle, MST,
7                   Shortest)
8 }
9 all(GraphType, Algorithms,Search?)
```

Figure 2: Feature Description of Graph Product Line

AND features respectively. Optionality is denoted by **?**. This feature description is sufficient to create feature types, so that it is semantically expressible that, e.g., the feature Number is of feature type Algorithms and feature Weighted is of the type GraphType. This feature description provides the language level specification of what to do with features in the software system, i.e., how to group them, how to compose them with respect to any constraints if present. The type representation for features consists of representation for feature specific properties such as whether they are mandatory or optional. Various advanced feature modeling concepts such as feature attributes, groups, and cardinalities can be implemented in the type representation for features in the extension compiler. Once the features are defined in the program, different feature models may be associated with these features. This allows creating not only different products per product line but also different product lines per set of features.

A specific variant graphProduct of the product line

```
1 variant GPL.graphProduct {
2   GraphType = Directed and Unweighted ,
3   Algorithms= StronglyConnected,
4   Search    = DFS
5 }
```

Figure 3: Creating a program variant for GPL

```
1 public variant alterGraphProduct(
2  variant graphProduct) {
3   removeFeature graphProduct DFS,
4   addFeature graphProduct BFS,
5   modifyVariant graphProduct
6   Algorithms=MST;
7   return graphProduct ;
8 }
```

Figure 4: Typed modification, addition and removal of features

GPL is created using the keyword variant (Figure 3). A program variant can be modified by altering the choice of features that constitute it. In Figure 4, feature modification, addition and removal are shown. Specific type for a feature need not be given as in Algorithms=MST, the type of feature is inferred from the feature description. The keywords addFeature, removeFeature and modifyVariant are used in the Java method alterGraphProduct() as operators, to alter the configuration of previously instantiated variant graphProduct.

The feature modeling constraints can be expressed explicitly in the embedded DSL. A feature model can be converted to a constraint satisfaction problem and various Java CSP solvers can be used to obtain valid configurations of features [8, 11, 15]. This can be implemented as a part of the type checking the AST nodes representing productline and variant types. Both mutual inclusion and mutual exclusion constraints between features can be represented as attributes of AST nodes representing the feature type. The implicit implementation constraints between the features are similarly taken care of in type checking the features, e.g., calls between methods of two features making these features dependent on each other. This type of constraints can be handled as a specialized checking of relations between related program elements [23], e.g., a feature that adds a call to a method must ensure that the method itself already exists.

Features can contain not only the class definitions and the class bodies, but also be part of classes and various statements and expressions. In Figure 5, feature Weighted contains definitions of feature specific introductions to separate classes in one place. JastAdd operates with the AST as the only repository of program information. The AST in JastAdd can be copied, extended, and rewritten based on conditions as well as compiled to byte code [17, 19, 30]. This provides a unique opportunity to modify the AST noninvasively both at compile time and run time. Therefore, we can implement features in such a way that the feature definitions need not be complete and additional code fragments can be added to features at runtime as well. A program element like a method can be part of many fea-

```
feature Weighted {
 class Graph {
  public void addEdge(Vertex begin,Vertex
  end, int weight) {
   addEdge(new Edge(begin,end, weight));
  }
 }

 class Edge {
  public int weight;
  public Edge(int the_weight){
   weight = the_weight;
  }
 //constructor with three arguments.
 ...
 }
}
```

Figure 5: feature containing various classes

tures, thus restricting duplication of code. Assume that addEdge() is part of features Weighted and Shortest. This can be achieved as shown in Figure 6. Currently, we intend to provide support for modularizing classes on the basis of features, but in future, we can include aspects in our extension. This is possible in JastAdd because aspects related extensions to their base Java compiler have already been added [7]. In the following,

```
public class Graph {
  feature Weighted,Shortest {
   public void addEdge(Vertex begin,
   Vertex end,int weight) {
    addEdge(new Edge(begin,end, weight));
   }
  }
...
  feature Directed {
   public static final boolean
   isDirected = true;
  ...
  }
...
}
public class Edge {
  feature Weighted {
   public int weight;
   public Edge(int the_weight) {
    weight = the_weight;
   }
 //constructor with three arguments.
 ...
  }
}
```

Figure 6: Classes containing features

we briefly explain how we propose tackle the problems of features mentioned before.

5.3 Solving Problems Related to Features

The combination of feature descriptions and first-class status for features in the extension compiler provides a clearcut way to approach feature-based software development.

- ***Hierarchical refinements*** – Features are no longer related to the class hierarchy as seen in Figures 5 and 6. The feature definitions, whether occurring inside classes/methods or themselves containing definitions of specific elements, are reconciled in one coherent collection when instantiating a product. Once features are reified internally, different transformations can be applied easily to the AST so that version of classes without feature annotations or feature definitions can be generated.
- ***Program deltas*** – Features in this extension use both compositional and annotative syntax as shown in Figures 5 and 6. Not only classes, methods and fields, but method parameters, various statements and expressions in Java can be assigned to features. Because parsing and semantic specifications in JastAdd are modular, our feature extension to Java can be modified easily to support deltas of only the required granularity.
- ***Feature composition and feature order*** – Features can be composed based on feature types. For example, Weighted and Directed features from Figure 1 may be composed to obtain a feature WeightDirected, based on the fact that both of them are of the type GraphType. Order may be specified between features and feature groups whenever required.
- ***Type support for features*** – We represent features as a reference type in the compiler. Various consistency checks for safe compositions can be straightforwardly implemented as lookups and Java typechecks which are implemented as inherited and synthesized attributes respectively in JastAdd.
- ***Dynamic composition and separate compilation*** – For implementing dynamic composition, we intend to use the capability of obtaining transformed copies of the AST as well as the possibility to reify feature code to byte code which can be used via variety of byte code manipulation packages. We intend to explore the use of contextual information for separate compilation of individual features.

6. RELATED WORK

Deursen and Klint [15] propose a language for describing feature models, but they implement features

using UML and Java code generation. In Caesar [29], classes can act as crosscutting layer modules containing many classes or types contributing to features. But it does not provide any interface for feature descriptions, or programmatic means of changing feature configurations. In Object Teams [20], a team is a container for classes and also at the same language level as class. Although it can be used to implement features, it is non-trivial to do so, as teams have a complex inheritance model in which features must be accommodated. Classbox/J [12] provides support for localized refinements, such that original and refined classes co-exist and can be referred to separately. But classboxes have the same problems as other compositional approaches that use redirection mechanisms in implementing features [12]. Like these approaches, we propose to use a more flexible containment for features with respect to classes. At the same time, we combine both feature descriptions and feature-oriented programming concepts together in features represented as first-class entities. Unlike the above mentioned approaches, a developer need not concern himself of how to represent features in terms of underlying technologies, e.g., how to represent features in terms of layers and bidirectional interfaces [29], teams with bindings [20], or classboxes [12]. Features have a structure set by a feature model expressed as feature descriptions and no extra representation is required to relate different code fragments to specific features.

7. CONCLUSION

We have proposed to raise the implementation level of features to first-class status by representing them as types with crosscutting containment in the extension. We have identified various properties that such an implementation should have in order to tackle various problems related to features. In future, we intend to work on extending the Java implementation of the JastAdd extensible compiler framework to include features.

ACKNOWLEDGEMENT

We thank Christian Kästner and Mario Pukall for comments on an earlier draft of this paper.

8. REFERENCES

[1] S. Apel, C. Kästner, and D. Batory. Program refactoring using functional aspects. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*. ACM Press, Oct. 2008.

[2] S. Apel, C. Kästner, and C. Lengauer. Feature Featherweight Java: A calculus for feature-oriented programming and stepwise refinement. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*. ACM Press, Oct. 2008.

[3] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 125–140. Springer, 2005.

[4] S. Apel, T. Leich, and G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In *Proceedings of the International Conference on Software Engineering*, pages 122–131. ACM Press, 2006.

[5] S. Apel, T. Leich, and G. Saake. Aspectual feature modules. *IEEE Transactions on Software Engineering*, 34(2):162–180, 2008.

[6] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebra for features and feature composition. In *Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology*, volume 5140 of *Lecture Notes in Computer Science*, pages 36–50. Springer-Verlag, 2008.

[7] P. Avgustinov, T. Ekman, and J. Tibble. Modularity first: a case for mixing aop and attribute grammars. In *Proceedings of the 7th international conference on Aspect-oriented software development*, pages 25–35. ACM, 2008.

[8] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Software Product Line Conference*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.

[9] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for Implementing Domain-Specific Languages. In *Proceedings of the International Conference on Software Reuse*, pages 143–153. IEEE Computer Society, 1998.

[10] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[11] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortes. Using Java CSP Solvers in the Automated Analyses of Feature Models. *LECTURE NOTES IN COMPUTER SCIENCE*, 4143:399, 2006.

[12] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: controlling the scope of change in Java. *ACM SIGPLAN Notices*, 40(10):177–189, 2005.

[13] R. Burstall. Christopher Strachey - Understanding Programming Languages. *Higher-Order and Symbolic Computation*, 13(1):51–55, 2000.

[14] K. Czarnecki and U. Eisenecker. *Generative*

Programming: Methods, Tools, and Applications. Addison-Wesley, 2000.

[15] A. v. Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.

[16] E. W. Dijkstra. On the Role of Scientific Thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer, 1982.

[17] T. Ekman and G. Hedin. The JastAdd system - modular extensible compiler construction. *Science of Computer Programming*, 69(1-3):14–26, 2007.

[18] S. Freeman and N. Pryce. Evolving an embedded domain-specific language in Java. *Conference on Object Oriented Programming Systems Languages and Applications*, pages 855–865, 2006.

[19] G. Hedin and E. Magnusson. Jastadd: an aspect-oriented compiler construction system. *Sci. Comput. Program.*, 47(1):37–58, 2003.

[20] S. Herrmann. Object Confinement in Object TeamsReconciling Encapsulation and Flexible Integration. *Aspect-Oriented Software Development*, 2003.

[21] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented Programming. *Journal of Object Technology*, 2007.

[22] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[23] C. Kästner and S. Apel. Type-checking software product lines - a formal approach. In *Proceedings of the International Conference on Automated Software Engineering*. IEEE Computer Society, 2008.

[24] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *In Proceedings of the International Conference on Software Engineering*, May 2008.

[25] J. Liu, D. Batory, and C. Lengauer. Feature-Oriented Refactoring of Legacy Applications. In *Proceedings of the International Conference on Software Engineering*, pages 112–121. ACM Press, 2006.

[26] R. Lopez-Herrejon, D. Batory, and W. R. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 169–194. Springer, 2005.

[27] R. E. Lopez-Herrejon and D. S. Batory. A standard problem for evaluating product-line methodologies. In *Proceedings of the Conference on Generative and Component-Based Software Engineering*, pages 10–24, 2001.

[28] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New-Age Components for Old-Fashioned Java. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–222. ACM Press, 2001.

[29] M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 127–136. ACM Press, 2004.

[30] A. Nilsson, A. Ive, T. Ekman, and G. Hedin. Implementing java compilers using rerags. *Nordic Journal of Computing*, 11(3):213–234, 2004.

[31] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. *LAMP-EPFL*, 2004.

[32] H. Ossher and P. Tarr. Multi-Dimensional Separation of Concerns and the Hyperspace Approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art InSoftware Development*. Kluwer, 2000.

[33] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer, 1997.

[34] M. Rosenmüller, N. Siegmund, G. Saake, and S. Apel. Code Generation to Support Static and Dynamic Composition of Software Product Lines. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*. ACM Press, Oct. 2008. to appear.

[35] C. Strachey. Fundamental Concepts in Programming Languages. *Higher-Order and Symbolic Computation*, 13(1):11–49, 2000.

[36] P. Tarr, H. Ossher, W. Harrison, and J. S. M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the International Conference on Software Engineering*, pages 107–119. IEEE Computer Society, 1999.

[37] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *Proceedings of the 6th international conference on Generative programming and component engineering*, pages 95–104. ACM, 2007.

[38] H. Zhang and S. Jarzabek. XVCL: a mechanism for handling variants in software product lines. *Science of Computer Programming*, 53(3):381–407, 2004.

Integrating Compositional and Annotative Approaches for Product Line Engineering

Christian Kästner
School of Computer Science
University of Magdeburg, Germany
kaestner@iti.cs.uni-magdeburg.de

Sven Apel
Department of Informatics and Mathematics
University of Passau, Germany
apel@uni-passau.de

ABSTRACT

Software product lines can be implemented with many different approaches. However, there are common underlying mechanisms which allow a classification into compositional and annotative approaches. While research focuses mainly on composition approaches like aspect- or feature-oriented programming because those support feature traceability and modularity, in practice annotative approaches like preprocessors are common as they are easier to adopt. In this paper, we compare both groups of approaches and find complementary strengths. We propose an integration of compositional and annotative approaches to combine advantages, increase flexibility for the developer, and ease adoption.

1. INTRODUCTION

In recent years, *software product lines (SPLs)* have gained momentum [9, 37]. Instead of implementing each program from scratch, SPLs enable systematic reuse in a domain by generating a family of related programs – so called *variants* – from a common code base. In this context, features are domain abstractions to distinguish different variants. Typically, features implement increments in functionality. Developers who want to adopt SPL technologies for their product can choose from a wide range of different mechanisms to implement SPLs: from version control systems [42], over simple *#ifdef* statements [35], over frameworks and components [9], to various specialized languages or tools [19, 39, 17, 10]. Each of these approaches has different advantages and disadvantages and there is plenty discussion about which approach is suited best, e.g., [33, 34, 3, 24, 35].

In earlier work, we used various approaches to implement SPLs. We started with AHEAD [10] and AspectJ [28] and their integration Aspectual Feature Modules [7]. These languages can be used to compose variants from reusable code units (compositional approaches). In recent work, however, we found several limitations [23, 24], especially when adopting SPL technology for legacy applications. Therefore, we looked at more traditional approaches like *#ifdef* preprocessors and improvements thereof (annotative approaches). Without ever making it explicit in our research agenda, we pursued both paths – compositional and annotative approaches – in parallel.

While we addressed the specific problems like granularity, language independence, expressiveness, or type-safety of either group of approaches in earlier work [24, 8, 25, 7, 23, 22, 5], we noticed that both groups complement each other. There are several problems for which compositional approaches required severe overhead, but an annotative approach can solve straightforwardly, or the other way around. In this paper, we give an overview of both groups of approaches, discuss differences and synergies, and show how to integrate both.

We focus especially on SPL adoption. Adopting SPL technologies for a project is difficult, especially if the target application is not developed from scratch but derived from a legacy application. There is some discussion whether lightweight implementation approaches can lower the adoption barrier or whether more sophisticated approaches are needed for maintainability and long-term project success [13]. In this paper, we show how an integration of both groups of implementation approaches can ease SPL adoption, but still support long-term design qualities.

Specifically, we make the following contributions: (a) We put annotative and compositional approaches (two groups of common approaches for SPL implementation) in contrast and analyze advantages and disadvantages of each. (b) We discuss an integration of both. (c) We outline how the integration can lower the adoption barrier but still support long-term qualities.

2. SPL IMPLEMENTATION APPROACHES

There are many approaches to SPL implementation. Most of them can be grouped either as compositional or as annotative [24]. In this section, we briefly introduce both groups, before we compare them to discuss advantages and disadvantages in Section 3.

Compositional Approaches. Compositional approaches implement features as distinct (physically separated) code units. To generate a product line member for a feature selection, the corresponding code units are determined and *composed*, usually at compile-time or deploy-time. There is a large body of work on feature composition, usually employing component technologies [44], frameworks [21], feature-oriented programming with some form of feature modules [39, 41, 10, 7], subjects [19], multi-dimensional separation of concerns [45], and aspects [28]. Depending on the concrete approach or language, the composition mechanism varies from assembling plug-ins to complex code transformations, but the general idea of composition as illustrated in Figure 1 is the same.

In this paper, we use AHEAD [10] – respectively the compatible, newer, and language-independent FSTComposer [8] that uses the same mechanisms – as representative for compositional approaches. In AHEAD and FSTComposer, features are implemented in separate modules that modify the base code. In Figure 2, we show a

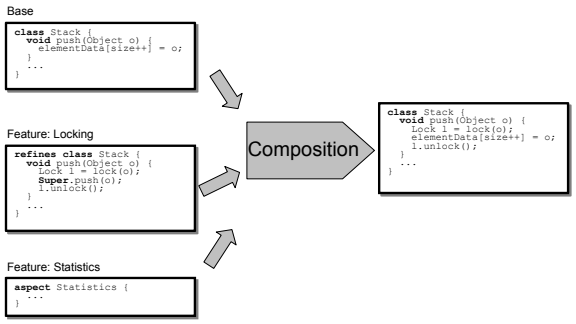


Figure 1: Composing code units.

```
1  class Stack {
2    void push(Object o) { ... }
3    Object pop() { ... }
4  }
```

```
5  refines class Stack {
6    void backup() { ... }
7    void restore() { ... }
8    void push(Object o) {
9      backup();
10     Super.push(o);
11   }
12 }
```

Figure 2: AHEAD example (compositional approach).

simple code example. The *base code* implements a stack, while in a separate module a feature *Undo* is implemented. The refines keyword indicates that the feature module extends an existing class. It introduces the new methods (*backup* and *restore*) and extends the existing method *push* similar to overriding using the keyword *Super*. The base code and different feature modules are composed with the AHEAD tool suite. Depending on which features are included in the composition process, different variants are generated.

Annotative Approaches. In contrast, annotative approaches implement features with some form of explicit or implicit annotations in the source code. The prototypical example, which is commonly used in industrial SPLs, are *#ifdef* and *#endif* statements of the C preprocessor to surround feature code. Such techniques are also common in commercial SPL tools as *pure::variants* [11] or *Gears* [31]. Other examples of annotative approaches are *Frames/XVCL* [20], *explict programming* [12], *Spoon* [36], *software plans* [14], *metaprogramming with traits* [47], and *annotation-based aspects* [29].

In this paper, we use our own tool CIDE as representative for annotative approaches. It is similar to *#ifdef* preprocessors in that code fragments are annotated and can be removed before compilation depending on the feature selection. However, CIDE improves over traditional preprocessors in several ways: (1) annotations are represented by background colors and do not obfuscate the source code with additional boilerplate code [24]; (2) annotations in CIDE are based on the underlying structure of the artifact and, thus, *disciplined* (e.g., it is not possible to annotate only an opening bracket but not the closing one) and ease the generation process [25, 24]; finally, (3) all annotations are managed by the tool infrastructure which allows virtual views on the source code (e.g., show all code that is annotated with feature *Backup*) and navigation support [27, 24].

In Figure 3, we show the previous example as implemented with CIDE. All feature code is located in the same code base. In the

```
1  class Stack {
2    void push(Object o) {
3      backup();
4      ...
5    }
6    Object pop() { ... }
7    void backup() { ... }
8    void restore() { ... }
9  }
```

Figure 3: Conditional compilation example (annotative approach).

printed version of this paper, all code annotated with the *Backup* feature is underlined. In order to create a variant without this feature, all annotated code is removed before compilation. Nevertheless, this paper is not specifically on CIDE, but other annotative approaches could be used or visualized in the same way.

3. COMPARISON

After briefly introducing the two groups of approaches, we compare them based on several characteristics. We selected characteristics that emphasize the differences between these approaches or which arose during our prior research or case studies. The results are summarized in Table 1 with approximated grades. This evaluation is based on prior work in SPL research and our own experience.

Feature Traceability. Feature traceability is the ability to directly trace a feature from the feature model (domain space) to the implementation (solution space) [15, 2]. For example, feature traceability is important when developers want to debug an error that occurs in a specific feature and want to find all code related to this feature.

Compositional approaches directly support feature traceability as the code that implements a feature can be traced to a single code unit (component, plug-in, aspect, feature module, etc). For example, in Figure 2, all code of the backup feature can be found in the second module (Lines 5–12). In contrast, in annotative approaches feature traceability is poorly supported as feature annotations can be scattered over the entire code base (cf. Fig. 3). However, with special tools like CIDE it is still possible to provide feature traceability in an annotative approach at a tool level. As explained above, virtual views and navigation support can be used to explore all code that belongs to a feature (see [27] for details). Nevertheless, feature traceability in annotative approaches is a matter of tool support.

Modularity. Modularity as needed for modular reasoning or even separate compilation is possible in some compositional approaches. For example, when using components [44], plug-ins [21], subjects [19] or hypermodules [45] this is well supported. However, many more advanced compositional approaches like many aspect languages or the analyzed approaches AHEAD or FSTComposer are based on source code transformations and provide only limited modularity, e.g., separate compilation is not supported. There are no interfaces for feature modules in these approaches, thus to understand a feature, it is often necessary to look also at the base code or even other features[1].

In annotative approaches a modularization is not intended. Modular reasoning can be simulated with tool support (views and navigation support as in CIDE) to some degree, but separate compilation is not possible.

[1]In the context of aspect-oriented programming, modularity has been addressed intensively, e.g., [43, 18], and can potentially be adapted for other compositional approaches.

```
1  class Stack {
2    void push(Object o) {
3      if (o==null) return;
4      hook();
5      ...
6    }
7    Object pop() { ... }
8    void hook() {}
9  }
```

```
10 refines class Stack {
11   void backup() { ... }
12   void restore() { ... }
13   void hook() {
14     backup();
15   }
16 }
```

Figure 4: Fine-grained extension with AHEAD.

Granularity. The granularity of implementation mechanisms provided by an approach is closely related to its expressiveness. Very coarse-grained approaches only assemble files in a directory, while fine-grained approaches allow modifications on the level of methods, statements, parameters or even expressions [24].

Annotative approaches support even fine-grained extensions well. As many are line-based or character-based, they scale from annotating entire files to even small code fragments of statements (cf. Fig. 3). Even when the underlying structure is used as in CIDE, annotations on the level of AST nodes allow even fine-grained annotations on statements, parameters, or expressions [24].

In contrast, compositional approaches only provide a coarse granularity composing usually only components or – in some approaches like aspects, AHEAD, or FSTComposer – down to introducing or extending methods in existing classes (cf. Fig. 2). However, manipulation of statements inside the middle of a method, of parameters or of expressions is not possible in any compositional approach due to conceptual limitations [24].[2] Instead, workarounds like hook methods are needed. For example, a slight modification in the original example makes an implementation with AHEAD difficult. Imagine that the *backup* call in Figure 2 is not the first statement executed in the *push* method, but located after some sanity checks. In this case workarounds as the hook method in Figure 4 are required, because – in contrast to annotative approaches – introducing a statement in the middle of a method is not supported.

Safety. For both compositional and annotative approaches recent research has provided solutions that ensure that all (potentially millions) variants of the SPL are syntactically correct and well-typed. While compositional approaches ensure syntactical correctness with their composition mechanism, many annotative approaches are line-based or character-based and can easily generate syntactically incorrect variants. However, CIDE – our annotative approach which enforces disciplined annotations – provides such safety using the underlying structure [25] and can thus achieve the same level of safety.

Although they use different implementations, there are several approaches to type-check entire SPLs for both compositional [46, 5] and annotative approaches [16, 22, 30]. Beyond type-safety, we know of no approach to verify behavior of all SPL variants that would scale to mid-sized SPLs. There are several approaches on SPL testing [38]; however, they work on the level of generated variants and are thus independent of the implementation mechanism.

Language independence. Language independence is another characteristic where both approaches perform similarly well. While, many annotative approaches are line-based or character-based and thus completely language-independent, even the more disciplined CIDE which uses the underlying structure can be extended for arbitrary languages (generated from the language's grammar) [25]. In contrast, compositional approaches are usually depending on a particular host language, where AHEAD and FSTComposer provide notable exceptions. Especially in FSTComposer, there is a general composition mechanism which can be easily extended for a new language with only little manual effort [8].

SPL Adoption. Industry is very careful on adopting compositional approaches because it influences their existing code base and development process too much. At most, after careful planning, frameworks or components are used [9]. In contrast, annotative approaches can be adopted much quicker, because they introduce only lightweight tools which do not change the code or development process too much at first [13]. Annotative approaches, thus, make adoption of SPL technologies easier and more likely in the initial steps of evaluation and early development. In this context, CIDE's concept of storing annotations separate from the source code is worth mentioning, because it allows annotating a legacy application without changing its source code representation. This makes CIDE well-suited for evaluating SPL technologies.

In an earlier case study, we experienced ourselves that refactoring a legacy application (in this case Oracle's Berkeley DB) into separate code units that can later be composed is by far more difficult and tedious than just annotating code [24].

4. COMPOSITIONAL AND ANNOTATIVE APPROACHES IN CONCERT

The comparison in the previous section showed that compositional and annotative approaches are quite different and have different – often complementary – strengths and weaknesses, which are also differently important in various phases of SPL adoption. Research on SPL implementation focuses almost exclusively on compositional approaches, ignoring the advantages of annotative approaches. Instead, the demand from industry has been answered mostly by commercial vendors like *BigLever* and *Pure Systems* which (among others) provide tools to annotate code. In this paper, we show that integrating compositional and annotative approaches is beneficial. In the following, we propose a simple integration and discuss its impact on the characteristics above.

4.1 Integration

Conceptually, an integration is straightforward. Using a compositional approach, features can be physically separated into code units (e.g., components, aspects, feature modules). Inside these physically separated code units, an annotative approach can be used to additionally annotate code fragments.

In Figure 5, we show a possible different implementation of the extended example of Figure 4. The method declarations for *backup* and *restore* are implemented in a physically separated AHEAD feature module, while some code (that would be hard to extract) is left as an annotation (underlined) in the base code. To generate a variant, the code units are composed and the annotations are evaluated to remove unneeded code fragments.

Also technically, such integration is straightforward. The annota-

[2]Some aspect languages like AspectJ can intercept join points in the body of methods, which can be used for extending statements to some degree (*'call && withincode'*). However, there are several limitations as discussed elsewhere [23, 24].

	AHEAD/FSTComposer (compositional)	CIDE (annotative)	Integrated Approach	Importance in the life cycle
Traceability	++	+	+	all phases
Modularity/Separate comp.	+	−−	+/−	maintenance
Granularity	−	++	++	early adoption, implementation
Safety	+	+	+	all phases
Language independence	+	+	+	all phases
Adoption	−	++	++	project start

++ very good support, + good support, +/− medium support, − poor support, −− no support

Table 1: Comparison

```
1  class Stack {
2    void push(Object o) {
3      if (o==null) return;
4      backup();
5      ...
6    }
7    Object pop() { ... }
8  }
```

```
9  refines class Stack {
10   void backup() { ... }
11   void restore() { ... }
12 }
```

Figure 5: Fine-grained extension with AHEAD.

tive approach – in our case CIDE – must only be extended to support the additional language constructs of the compositional approach (e.g., *refines* keyword), and the generation process must be adapted to handle both composition and evaluating annotations. If annotations are evaluated before the actual composition, the composition process itself does not even need to be adapted. On the tool level, an integration is more difficult as both tools must be integrated. Views and navigation support for the annotative approach must be extended for physically separated feature implementations, and existing support for the compositional language as in FeatureIDE [32] (e.g., syntax highlighting, code completion, outline view) must be extended to understand the annotations. However, integrating tools is not a conceptual challenge, but merely an engineering task.

Interestingly, *automated refactorings* that transform annotated code into physically separated AHEAD feature modules [26] and automated refactorings that transform AHEAD feature modules into annotated CIDE code [30] have been developed. While the result of these refactorings might be difficult to read (e.g., the transformation from CIDE to AHEAD heavily requires hook methods as in Figure 4, which makes generated code hard to read in the presence of fine-grained extensions) or unambiguous (transforming AHEAD to CIDE, there are many different possible annotated programs that express the same behavior), these automated refactorings still help developers to deal with the different possible representations of the source code. Note, refactorings can also be used to convert only parts of the SPL, e.g., convert individual features or only code from certain classes or methods.

4.2 Comparison

An integration of compositional and annotative approaches does not automatically dissolve all disadvantages of either approach. For example, when using annotations in physically separated feature modules, modularity is lost as if only annotations were used in the first place. However, the main advantage is that developers can always decide when to use which approach and when to use a combination of both, e.g., to achieve fine granularity or ease adoption. In the following, we discuss the criteria listed in Section 3 for the integrated approach. Approximated grades are shown in Table 1.

Feature Traceability. First, feature traceability is weaker than in pure compositional approaches. A feature must not be physically separated, but can instead or *additionally* be implemented by some scattered (annotated) code fragments, as shown in the example in Figure 5. However, still the same views and navigation support from CIDE can be used, so feature traceability is not worse than in the CIDE solution. Moreover, if a full (or partial) physical separation is desired it can be achieved and even improve feature traceability. In that case, all (or most) relevant feature code can be found in one corresponding code unit.

Modularity. When integrating compositional and annotative approaches modularity depends on how the developers implement a feature. They can choose between a modular implementation (using classes, modules, aspects, feature modules) or a non-modular implementation with (at least some) scattered code fragments as in Figure 5. In general, modularity is weakened and separate compilation is no longer possible. However, using gradual refactorings it is possible to achieve modularity in the long run.

Granularity. The integrated approach benefits from the fine granularity of the annotative approach. Features can still be implemented with physically separated code units as far as reasonable or possible with the low available granularity. However, additionally fine-grained extensions can still be added to the base code or other features and marked with annotations. For example, instead of using workarounds as the hook method in Figure 4, such fine-grained extensions can be implemented with an annotation as shown in Figure 5. Again, the integrated approach allows a quick solution using annotations at first, while further refactorings are possible to change the implementation to avoid fine-grained extensions and use only the compositional approach in the long run.

Safety and language independence. As discussed above, there are striking commonalities in the solutions for type-checking and language-independence developed in recent research for both, compositional and annotative approaches. As the mechanisms are already related, and integration is mostly an engineering task. Thus, the same level of safety and language independence can be achieved as in the original isolated approaches.

Adoption.

The most interesting results from an integration affect the process of adopting SPL technologies for a project. While industry is very careful about adopting compositional approaches, they are usually seen as superior in academia because of modularity, separations of

concerns and thus promised improvements during maintenance and evolution in the later life cycle phases. Nevertheless, companies often use lightweight annotative approaches for faster results and lower initial risk [13].

In this scenario, the integration of compositional and annotative approaches pays off. In early evaluation and adoption stages, developers can simply annotate legacy code. They can use the lightweight capabilities of annotative approaches without having to change their code base. As annotations are stored separately in CIDE, these annotations do not even affect the code base at all.

In later stages, when the idea of developing SPLs is established, and annotations already provide variability they can gradually change from the annotated code base to separated feature modules by automated or manual refactoring. Still, it is not necessary to refactor all annotated code fragments at once, but developers can start with the obvious coarse-grained ones (separate entire classes or method introductions) and gradually prepare the code (e.g., by introducing explicit extension points) to avoid even the annotated fine-grained extensions. This way it is possible to adopt a compositional approach gradually and eventually achieve long-term goals of modularity and maintainability.

4.3 Discussion

Instead of being forced to choose between an annotative or compositional approach, an integration allows to start with one and gradually refactor to the other. For each problem, developers can choose the mechanism that suits best at first and only later refactor to a different version if reasonable. Though this allows developers to break modularity, it also gives them expressive power to express fine-grained extensions. Goals that have been achieved in either approach like traceability, safety, or language independence can be adapted also for the integrated approach.

Instead of a one-step effort with uncertain costs and risks, the integration allows a lightweight adoption with annotative approaches and gradual refactoring to compositional approaches. This eases the initial adoption barrier significantly, while long-term goals are explicitly supported.

5. RELATED WORK

Several related publications compare different approaches to SPL implementation. First, Lopez-Herrejon et al. [33] and Mezini and Ostermann [34] compare several compositional approaches. Both focus in detail on modularization support and do not cover annotative approaches. Next, Anastasopoulos and Gacek [1] briefly compare 11 concrete implementation approaches, Muthig and Patzke [35] also compare 6 approaches. Both comparisons include conditional compilation and frames as annotative approaches. However, in both works analysis is focused on expressiveness and several details of analyzed languages and does not consider an integration of approaches. We provide a broader comparison on a higher level of abstraction, where we each subsume all compositional approaches and annotative approaches. For our analysis the subtle distinctions between different compositional approaches is not relevant, but the importance lies in the bigger picture achieved with an integration.

The idea of adopting SPL technology slowly and stepwise emerged from Spinczyk in a discussion at the Dagstuhl seminar 'Software Engineering for Tailor-made Data Management' [4]. The participants proposed a migration path from 'thinking in product lines', over using advanced preprocessors, over object-oriented decomposition, toward more advanced compositional approaches of aspect-oriented programming and feature-oriented programming, and eventually toward model-driven development or a decoupling using service-orientation. We follow this migration path on the lower levels and actually combine annotative and compositional approaches to be able to evolve gradually from one approach to the other.

FeatureC++ [6], a compositional approach for C++ based on feature-oriented and aspect-oriented programming mechanisms, still allows to use the C preprocessor (*#ifdef*). FeatureC++ can therefore be considered as an existing implementation that already integrates compositional and annotative approaches to some degree. Nevertheless, this integration was not planned and never made explicit. The used annotative approach is line-based, does not use the underlying structure and there are neither views nor navigation support. In prior work using FeatureC++, developers explicitly avoided to use the preprocessor to achieve variability [40].

6. CONCLUSION

There are many different approaches to implement SPLs. Most can be classified as either compositional or annotative. While annotative approaches like simple *#ifdef* directives are common in industry, research focuses mostly on compositional approaches like aspect-oriented or feature-oriented programming.

In this paper, we compared both groups of approaches and found that there are many differences in strengths and weaknesses, but that it is also possible to base both on similar foundations based on the artifact's structure. To combine the advantages of each and address the shortcomings we propose an integration. With this integration a developer can separately choose which implementation mechanism to use for each problem. This eases SPL adoption. When adopting SPL technologies, it is possible to first use the lightweight annotative approaches and then gradually refactor toward compositional approaches as far as possible and reasonable.

In future work, we want to implement the integration and experimentally merge our tools CIDE and FSTComposer, and evaluate our approach empirically in an industrial case study or experiment.

7. REFERENCES

[1] M. Anastasopoules and C. Gacek. Implementing product line variabilities. *SIGSOFT Softw. Eng. Notes*, 26(3), 2001.

[2] G. Antoniol, E. Merlo, Y.-G. Guéhéneuc, and H. Sahraoui. On Feature Traceability in Object Oriented Programs. In *Proc. ASE Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)*. 2005.

[3] S. Apel and D. Batory. When to Use Features and Aspects? A Case Study. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, 2006.

[4] S. Apel, D. Batory, G. Graefe, G. Saake, and O. Spinczyk, editors. *Software Engineering for Tailor-made Data Management*. Dagstuhl Seminar Proceedings. 2008. to appear.

[5] S. Apel, C. Kästner, and C. Lengauer. Feature Featherweight Java: A Calculus for Feature-Oriented Programming and Stepwise Refinement. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. 2008.

[6] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. 2005.

[7] S. Apel, T. Leich, and G. Saake. Aspectual Feature Modules. *IEEE Trans. Softw. Eng.*, 34(2), 2008.

[8] S. Apel and C. Lengauer. Superimposition: A Language-Independent Approach to Software Composition. In *Proc. ETAPS Int'l Symposium on Software Composition*, 2008.

[9] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[10] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Softw. Eng.*, 30(6), 2004.

[11] D. Beuche, H. Papajewski, and W. Schröder-Preikschat. Variability Management with Feature Models. *Sci. Comput. Program.*, 53(3), 2004.

[12] A. Bryant et al. Explicit programming. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*. 2002.

[13] P. Clements and C. Krueger. Point/Counterpoint: Being Proactive Pays Off/Eliminating the Adoption Barrier. *IEEE Software*, 19(4), 2002.

[14] D. Coppit, R. Painter, and M. Revelle. Spotlight: A Prototype Tool for Software Plans. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. 2007.

[15] K. Czarnecki and U. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press, 2000.

[16] K. Czarnecki and K. Pietroszek. Verifying Feature-based Model Templates against well-formedness OCL Constraints. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. 2006.

[17] M. Griss. Implementing Product-Line Features by Composing Aspects. In *Proc. Int'l Software Product Line Conference (SPLC)*. 2000.

[18] W. Griswold et al. Modular Software Design with Crosscutting Interfaces. *IEEE Software*, 2006.

[19] W. Harrison and H. Ossher. Subject-oriented programming: a critique of pure objects. *SIGPLAN Not.*, 28(10), 1993.

[20] S. Jarzabek et al. XVCL: XML-based Variant Configuration Language. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. 2003.

[21] R. E. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2), 1988.

[22] C. Kästner and S. Apel. Type-checking Software Product Lines - A Formal Approach. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. 2008.

[23] C. Kästner, S. Apel, and D. Batory. A Case Study Implementing Features Using AspectJ. In *Proc. Int'l Software Product Line Conference (SPLC)*. 2007.

[24] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. 2008.

[25] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Language-Independent Safe Decomposition of Legacy Applications into Features. Technical Report 2, School of Computer Science, University of Magdeburg, Germany, 2008.

[26] C. Kästner, M. Kuhlemann, and D. Batory. Automating Feature-Oriented Refactoring of Legacy Applications. In *Poster presented at Europ. Conf. Object-Oriented Programming*, 2007.

[27] C. Kästner, S. Trujillo, and S. Apel. Visualizing Software Product Line Variabilities in Source Code. In *Proc. SPLC Workshop on Visualization in Software Product Line Engineering (ViSPLE)*, 2008.

[28] G. Kiczales et al. An Overview of AspectJ. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*. 2001.

[29] G. Kiczales and M. Mezini. Separation of Concerns with Procedures, Annotations, Advice and Pointcuts. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*. 2005.

[30] C. H. P. Kim, C. Kästner, and D. Batory. On the Modularity of Feature Interactions. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. 2008.

[31] C. Krueger. Easing the Transition to Software Mass Customization. In *Proc. Int'l Workshop on Software Product-Family Eng.* 2002.

[32] T. Leich, S. Apel, and L. Marnitz. Tool Support for Feature-Oriented Software Development: featureIDE: an Eclipse-based Approach. In *OOPSLA workshop on eclipse technology eXchange*. 2005.

[33] R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*. 2005.

[34] M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. *SIGSOFT Softw. Eng. Notes*, 29(6), 2004.

[35] D. Muthig and T. Patzke. Generic Implementation of Product Line Components. In *Proc. Net.ObjectDays*. 2003.

[36] R. Pawlak. Spoon: Compile-time Annotation Processing for Middleware. *IEEE Distrib. Sys. Onl.*, 7(11), 2006.

[37] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[38] K. Pohl and A. Metzger. Software product line testing. *Commun. ACM*, 49(12), 2006.

[39] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*. 1997.

[40] M. Rosenmüller et al. FAME-DBMS: Talor-made Data Management Solutions for Embedded Systems. In *Proc. EDBT Workshop on Software Engineering for Tailor-made Data Management*, 2008.

[41] Y. Smaragdakis and D. Batory. Mixin Layers: an Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2), 2002.

[42] M. Staples and D. Hill. Experiences Adopting Software Product Line Development without a Product Line Architecture. In *Proc. Asia-Pacific Software Engineering Conf.* 2004.

[43] K. Sullivan et al. Information Hiding Interfaces for Aspect-Oriented Design. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering*. 2005.

[44] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

[45] P. Tarr et al. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. 1999.

[46] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. 2007.

[47] A. Turon and J. Reppy. Metaprogramming with Traits. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*. 2007.

Refactoring in Feature-Oriented Programming: Open Issues

Ilie Şavga Florian Heidenreich

Institut für Software- und Multimediatechologie, Technische Universität Dresden, Germany
{ilie.savga, florian.heidenreich}@tu-dresden.de

Abstract

Similar to refactoring, feature-oriented programming can be seen as a metaprogramming paradigm, in which programs are values and composition operators transform programs to programs. In this position paper we discuss open issues of applying refactoring in the context of feature-oriented programming. First, we elaborate on the role of refactoring in maintaining features and their implementations as well as the impact of refactoring on the relation between the *problem* and *solution spaces*. Second, we discuss issues of relating well-known refactoring formalisms to existing formal approaches used in feature-oriented programming. Third, we suggest to use refactoring semantics to upgrade and test final products of a product line.

1. Introduction

Research in the area of Software Product Lines (SPL) focuses on the design and automatic synthesis of product families (11). An important concept in this area is that of a *feature*–a refinement in the product functionality (33). Each product within a product family can be identified by a unique combination of features, from which it is created. By modeling the *problem space* of a domain, a *feature model* defines all legal feature configurations. A particular configuration chosen by the user is used to generate a final product out of feature modules that comprise the *solution space* (12).

Feature-oriented programming (FOP) is concerned with designing and implementing features and can be seen as a metaprogramming paradigm (6): feature composition modifies (by addition and extension) base programs using features. Another well-known metaprogramming paradigm is *refactoring*—program-to-program transformations that "does not alter the external behavior of the code yet improves its internal structure." (18, p.9) Several recent publications (e.g., (6), (9), (28), (40)) point out various contexts where refactorings and FOP overlap. Inspired by their observations, in this position paper we elaborate on open issues relating FOP and refactorings. In particular, we overview related work and discuss:

- Refactoring software artifacts in FOP (Section 2). In FOP refactoring means restructuring software artifacts of the solution space, the problem space, or both. When refactoring the solution space, how to keep multiple representations of a feature consistent? When refactoring the problem space, what is a meaningful library of refactoring operators to restructure a feature model? Moreover, in some cases refactoring one space requires refactoring the another space. How could one synchronize both spaces to preserve their consistency?
- Refactoring definition in a formal FOP calculus (Section 3). Existing work on refactoring formalization varies in the way refactorings are defined. Which formalism is more appropriate to be integrated into existing FOP formal definitions? And which FOP formalism, if any, would be appropriate for such integration?
- Refactoring-based product maintenance (Section 4). As refactorings are formally defined, the refactoring history of a component can be treated as a formal specification of its change representing a kind of *maintenance delta* (10) to be used for automatic software construction and maintenance. In the context of FOP, how to use formal specification to alleviate maintenance tasks, such as upgrade and testing, of final feature-based software products?

Our main goal is to foster discussions on the role of refactoring in developing and maintaining software artifacts in the context of FOP.

2. Refactoring Software Artifacts in FOP

Extensively used software artifacts have to evolve considerably, because, according to the Lehman's first law, "a large program that is used undergoes continuing change or becomes progressively less useful." (30, p. 250) The second

McGPLE'08 October 23, 2008, Nashville, Tennessee, USA.
ACM [to be supplied].

Lehman's law says, that "as a large program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it." (30, p. 253). This work is performed by program restructuring, called refactoring in object-oriented systems (32).

Since both the problem and the solution spaces are represented by software artifacts, evolving the latter will inevitably imply their refactoring.[1]

2.1 Refactoring Solution Space

An example of refactoring the solution space is refactoring the source code of a feature module to address changed security requirements. Developers may need to move some functionality previously accessible in more general features to more specialized features restricting the visibility of functionality to more specific features. This operation will involve moving members among feature modules and is an example of a reactive refactoring. An example of a proactive refactoring is renaming certain members of feature modules to follow established naming conventions improving thus future maintainability of modules.

In general, besides source modules, features can be implemented by modules containing, for instance, binary code, grammars and makefiles (8), XML documents (2) or UML diagrams (13). Therefore, in addition to code refactoring (e.g., (15; 18; 32; 34)), in the context of FOP one should also consider existing work on refactoring of other software components, such as sequence and protocol state machines (36) or UML class diagrams (37). A practical problem is how to correlate refactorings in multiple representations of the same feature.

A possible solution is to use functions that map refactorings of one representation to refactorings of other representations (similar to *mapdeltas* as proposed by Batory (7)). It is, however, not clear whether it is possible to find a refactoring counterpart in any possible representation, that is, to map refactorings in all representations used in the solution space. This problem stems from the lack of rigor in formal specifications of refactorings for certain representations (e.g., what is a grammar refactoring?). Moreover, as Batory mentions (7), creating and maintaining functions that relate transformations may burden programmers considerably.

2.2 Refactoring Problem Space

Facing new and changing requirements, besides evolving the solution space, the variability of a product line has to evolve, too. If not refactored, features of a feature model will not reflect changing domain requirements. Moreover, the number of features may explode making the model unmanageable.

In a large industrial case study, Loesch and Ploedereder (31) show how formal concept analysis (19) of existing products can be used to find obsolete (unused) features as well as to derive missing feature constraints. Their restructuring proposals, such as merge/remove variable features and mark mutually exclusive features as alternatives, can be considered feature model refactorings.

Alves et al. (1) define a feature model refactoring as a transformation that improves the model's configurability (i.e., the number of valid variants defined by the model). Based on their definition, the authors suggest a number of refactorings that either increase configurability of the model (while addressing new model requirements or merging feature models) or does not affect the number of model's variants (while maintaining the model).

As opposed to enlarging the number of model's variants, Kim and Czarnecki (29) discuss the impact changes made to a feature model may have on model's specializations (i.e., models derived successively by specializing the initial model). For several changes applied to the initial model (e.g., cardinality change or feature addition), the authors define synchronizing changes in specializations, including the final specialization (i.e., configuration). Changes mentioned by Kim and Czarnecki (29) can also be seen as refactorings operating on feature models.

A future research direction is reusing the work of (1; 31; 29) to define a set of feature model refactorings, such as `MakeMandatory`, `MakeAlternative`, `DeleteFeature`, `CopyFeature` or `ReduceGroupCardinality`. For that, an important requirement is to define precisely preconditions and, perhaps, synchronization actions of such refactorings for relating changes of the problem space to the solution space (as discussed in the next section).

2.3 Synchronizing Refactored Spaces

The key issue in refactoring of the two spaces is that their refactoring should not be considered in isolation; otherwise seemingly safe changes may lead to wrongly composed final products. For example, consider refactoring the solution space that makes one feature module dependent on another module.[2] If applied only to the solution space, this valid (with regard to the module implementation) refactoring is not reflected in the feature model as an additional constraint between the features involved. As a consequence, the model will permit a configuration that includes the depending feature without the feature it depends on. As another example, if a feature in the feature model is made optional, whereas other features depend on its functionality in their implementation, the feature may be missing in a configuration leading to invalid final product implementation. As an extreme case, for a wrongly defined feature model the set of its configurations may be empty.

In general, changes made to one space should propagate to another space. More precisely, in case refactorings change

[1] Terminology note: in this section by refactoring of problem and solution spaces we mean refactoring of feature models and their corresponding feature modules as opposed to *feature-based refactoring* of legacy software into a set of feature modules (e.g., (3; 27; 28; 40)).

[2] Our examples are inspired by Czarnecki and Pietroszek (13) and Thaker et al. (38)

constraints on the valid combination of features or feature modules, constraints of another space must be updated correspondingly. It is important that the overall "strictness' of the implementation constraints of the solution space and the domain constraints of the problem space are not equivalent. The implementation constraints are defined by the language (metamodel), in which feature modules are defined, and by the modules' implementation itself. Domain constraints are defined by the feature model. In general, as argued by Czarnecki and Pietroszek (13) and Thaker et al. (38), the domain constraints must imply the implementation constraints. Representing constraints in propositional formulas and using SAT solvers, it is possible to automatically detect when such implication does not hold, and then manually solve inconsistencies between the two spaces (13; 38).

The elegant solution of using SAT solvers is, however, not perfect in the context of agile refactoring of problem and/or solution spaces. Drawing analogies with conventional code refactoring, it would mean manually changing the program, recompiling it to detect possible problems and then solving the latter manually. Instead, when applying small changes to features or their implementation, it would be preferable to immediately know whether changes are safe and do not lead to space inconsistencies. Moreover, similar to a refactoring engine updating calls to a renamed method or prompting for a default value of a new parameter, some space inconsistencies could be interactively fixed, at least, semi-automatically.

Generalizing this discussion, an important issue with regard to refactoring in FOP is how to synchronize the spaces being refactored to ensure no invalid product may be generated afterwards. With this regard, two important issues to be considered are:

1. How the relation between the two spaces is defined. Space relation may be defined as feature template annotations (13), code annotations relating features to AST nodes (28)[3], a separate metamodel-based specification (20), or a systematically organized directory and file structure (8).
2. Which safeness constraints (in other words, invariants that these constraints preserve) a refactoring must respect. Safeness constraints may be defined by separate specifications (e.g., OCL expressions (13), propositional formulas (38) or type rules (28)) or may be embedded into the language type system (4; 27).

The space relation complemented with safeness constraints can be treated as a model to reason about and detect space inconsistencies. Janota and Botterweck (26) explicitly define such a model, which they call feature-component model, by formally specifying the feature model, component model and constraints on their relations. They derive the feature model induced by a feature-component model, compare it with the provided feature model and detect possible weaknesses of the latter.

Motivated by the aforementioned work, an important question is how to define and realize an interactive refactoring environment permitting for safe refactoring of the problem and solution spaces and semi-automatic space synchronization.

3. Refactoring Definition in a Formal FOP Calculus

Recent work on formalization of feature-based software development (5; 9) aim for an algebra to represent and reason about features and their composition.[4] The key idea is of both approaches is the same: find an atomic unit of feature representation, use it to uniformly define feature structure and then define precisely how those structures are composed. Moreover, feature composition is always feature addition and/or feature modification. Feature modification is performed by modifiers—`(selector,rewrite)` pairs applied to atomic units, where `selector` finds program units (using pattern matching) and then `rewrite` applies to the units found. However, the two aforementioned approaches differ in the way they model features and divide composition power between addition and modification.

Batory and Smith (9) use as the atomic unit of feature representation a (primitive) term, that is, a key-value pair. A features is either a vector of terms or a delta vector. The latter is a unary function that transforms vectors to vectors by addition and modification. Feature addition uses set union (using term names) of vector terms and raises an error when conflicting term names occur. Feature modification is term selection and term rewriting.

In the feature algebra of Apel et al. (5), the atomic representation unit is a tree node (a so-called *atomic introduction*). As a consequence, a feature is modeled as a tree, called a *feature structure tree* (FST) of various abstraction levels. Feature addition is tree superimposition (i.e., node conflicts are resolved by language-specific overriding of leaf nodes).[5] Feature modification is tree traversal and tree rewrite. The key difference to the work of Batory and Smith (9) is the shift of composition power from modification to addition: due to overriding, such concepts as mixins can be unified with introduction (performed by union set) and need not be modeled by modifiers, as in (9).

The vision of Apel et al. (5) and Batory and Smith (9) is that by implementing an uniform calculus one will develop

[3] On the contrary to feature composition, in CIDE (28) annotations are used for feature *decomposition* to support feature-oriented refactoring of existing programs into features.

[4] The algebra of Höfner et al. (23) focuses on the analysis phase of feature-oriented development and considers neither the structure of features nor their implementation. Since the latter two are our main concerns regarding refactoring, we do not discuss the aforementioned work in this paper.

[5] More exactly, FST can be seen as a set of superimposed (added) atomic introductions and superimposition is modeled by the operator *introduction sum* composing introductions, hence FSTs.

an object-oriented framework for feature composition that is independent of a concrete implementation language. In such framework, all kinds of manipulated programs are represented uniformly under a common superclass. This superclass provides standard operations (implemented specifically for each supported kind of programs) to query and transform programs. For us it is interesting to investigate how existing work on program refactoring can be transferred to such uniform frameworks of program transformations. With this regard, the dual research questions is 1) which existing refactoring formalism could be adopted easier into a FOP calculus, and 2) which FOP calculus, if any, is appropriate for such adoption.

While classical refactoring definitions (32; 34) use predicate logic to define preconditions, in the context of tree manipulation more recent work using graphs to define preconditions, either by testing of presence conditions (42) or by defining a graph pattern to be matched (22; 41), may be more appropriate for defining pattern matching used by modifiers. Moreover, while the actual transformation of a refactoring is usually described informally (18; 32; 34; 39; 42), for uniform transformations of program graphs one should also consider formal definition of refactoring transformations using graph rewriting (22; 41) to define `rewrite` of modifiers.

Batory and Smith define two types of modifiers (9). While a *universal modifier* finds all program terms that have the name or value `selector` (and rewrite these terms), an *existential modifier* attempts to find `selector` by name. If the term is undefined, it assigns the term an existence error; otherwise, the modifier does nothing (9). Whereas probably all refactorings require universal modifiers (for example, to rewrite multiple method calls or push down methods to several subclasses at once), the question is whether some refactorings may also require existential modifiers. For instance, it may make sense to test for the method usage before deleting it and signal an error in case it is in use. In a sense, it would be similar to the code analysis implemented in the conventional refactoring engines, but at the general and uniform level of the finite map space.

Finally, and most important, the key difference of a refactoring from a feature is that refactoring may also require term (node) deletion. For example, moving a method can be seen as deleting it from one class and adding to another class. Future work mentioned by Batory and Smith (9) is defining delta vectors that support vector subtraction. However, allowing such modification to parts of features may lead to a situation that the target of a subsequent feature composition is eliminated by a previously executed refactoring. An appealing research direction is to investigate and define a proper interaction (composition) of features and refactorings.

4. Refactoring-based Product Maintenance

Since refactorings are program transformations with formally defined semantics (32; 34), a history of refactorings applied to a component can be treated as a formal specification of the component's structural change (16). Such specification could be used to alleviate tasks of maintaining a product line, for example, upgrading and testing.

4.1 Module Upgrade

After a feature module is refactored, one may want to also update existing products to propagate improvements of the new module version. In some cases, simple recompilation of all modules would take too much time and their complete redeployment. Instead, it may be preferred to update only the refactored modules.

Several software engineering approaches (17; 21; 35) use refactoring history to automatically upgrade a software library (or a framework). They base on the fact that more than 80% of library changes that break library-dependent applications are API refactorings (16). Using refactoring information, it is possible to adapt existing applications to the new library version (21), the new library to existing applications (17) or create adapters that translate between the library and its applications (35).

In the line of these approaches, refactoring semantics can be used to upgrade existing products of a product line. For example, similarly to the approach of Henkel and Diwan (21) refactorings could be effectively re-executed on the product implementation, synchronizing it with the refactored implementation. As another example, using refactoring history it would be possible to partially decompose an existing product extracting obsolete feature implementation and then compose back the final product using refactored feature implementation.

4.2 Product Line Testing

To ensure that generated feature-based programs are correct, Batory (7) suggests to use specification-based product line testing using Alloy (25). An Alloy specification describes properties of the program to be verified. Out of this specification, a set of input tests represented by a propositional formula is generated, solved by a SAT solver and converted into a test (7).

Although we do not have any practical results, we envisage using refactoring semantics to automatically derive such specifications (and, hence, product line tests). Several formalisms of refactoring definition use the notion of postconditions for refactoring definitions, either as logic predicates (34) or as modified graphs (41; 42). Although actual definitions differ, the intuition is the same: a postcondition reflects the semantics of the refactoring transformation and describes important structural particularities of the refactored program. An approach would be to convert the postconditions into a propositional formula and generate tests in the

similar manner as Batory suggests (7). This would detect program errors introduced, for example, by bugs in refactoring engines (14). Furthermore, depending on how the invariants are reflected by postconditions, it could also detect "bad smells" specific for FOP, like inadvertently overriding a method in a base class by a renamed method in a feature refining that base class. Moreover, because refactorings can be composed (34), given a refactoring history as a sequence of refactorings its composed postcondition can be derived. In such cases, there is no need to create tests for each single refactoring—a set of tests can be created at once for the whole refactoring history.

To guarantee type-checking software produce lines for the price of reduced language power, Kästner and Apel (27) adopt the Featherweight Java (FJ)–a formally specified minimal functional subset of Java (24)–as a implementation language of product lines. They propose Color Featherweight Java (CFJ) as a FJ-based calculus to describe the entire (valid) software product line in combination with annotations and prove that, if the product line is well-typed (with regard to the CFJ language grammar), then all generated FJ variants will be well-typed (i.e., the generation will preserve typing). When using such a language subset as FJ with proved type-soundness, a question is which transformations considered refactorings for its original superset (i.e., Java) can be considered as such for FJ, that is, do not lead to typing errors (and invalid program variants) according to the grammar of FJ.

5. Summary

In our position paper we discuss open issues of applying refactoring in the context of FOP. We believe that by addressing the research and practical questions formulated in the paper, it is possible to integrate existing work on program refactoring into the context of FOP building a framework for uniform program transformation and tools that combine feature-oriented programming and refactoring. With regard to the inherent complexity of developing software product lines, it is important that these tools will foster agile development and maintenance of feature-related software artifacts and will give a uniform view on a feature-oriented development environment.

Acknowledgements

This research has been co-funded by the German Ministry of Education and Research (BMBF) within the project feasiPLe (cf. `http://www.feasiple.de`).

References

[1] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Refactoring product lines. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 201–210, New York, NY, USA, 2006. ACM.

[2] F. I. Anfurrutia, O. Díaz, and S. Trujillo. On the Refinement of XML. In *ICWE'07: Proceedings of the Seventh International Conference on Web Engineering*, volume 4607 of *Lecture Notes in Computer Science*, pages 473–478. Springer, 2007.

[3] S. Apel, C. Kästner, and D. Batory. Program refactoring using functional aspects. In *GPCE'08: Proceedings of the Seventh International Conference on Generative Programming and Component Engineering*, Nashville, Tennessee, USA, October 2008. ACM.

[4] S. Apel, C. Kästner, and C. Lengauer. Feature Featherweight Java: a calculus for feature-oriented programming and stepwise refinement:. In *GPCE'08: Proceedings of the Seventh International Conference on Generative Programming and Component Engineering*, Nashville, Tennessee, USA, October 2008. To appear.

[5] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebra for features and feature composition. In J. Meseguer and G. Rosu, editors, *AMAST'08: 12th International Conference on Algebraic Methodology and Software Technology*, Lecture Notes in Computer Science, pages 36–50. Springer, 2008.

[6] D. Batory. Program refactoring, program synthesis, and model-driven development. In *CC'07: Proceedings of the 16th International Conference on Compiler Construction*, volume 4420 of *Lecture Notes in Computer Science*, pages 156–171. Springer, 2007.

[7] D. Batory. From practice towards theory: Using elementary mathematics as a modeling language. In *GPCE'08: Proceedings of the Seventh International Conference on Generative Programming and Component Engineering*, Nashville, Tennessee, USA, October 2008. To appear.

[8] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling stepwise refinement. *IEEE Transactions in Software Engineering*, 30(6):355–370, 2004.

[9] D. Batory and D. Smith. Finite map spaces and quarks: Algebras of program structure. Technical Report TR-07-66, University of Texas at Austin, Dept. of Computer Sciences, 2007.

[10] I. D. Baxter. Design maintenance systems. *Commununication of the ACM*, 35(4):73–89, 1992.

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

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

[13] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness ocl constraints. In *GPCE'06: Proceedings of the Fifth International Conference on Generative Programming and Component Engineering*, pages 211–220, New York, NY, USA, 2006. ACM.

[14] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *ESEC-FSE'07: Proceedings of the the Sixth Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 185–194, New York, NY, USA, 2007. ACM.

[15] D. Dig. *Safe Component Upgrade*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, USA, October 2007.

[16] D. Dig and R. Johnson. The role of refactorings in API evolution. In *ICSM'05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society.

[17] D. Dig, S. Negara, V. Mohindra, and R. Johnson. ReBA: Refactoring-aware binary adaptation of evolving libraries. In *ICSE'08: Proceedings of the 30th International Conference on Software Engineering*, pages 441–450, New York, NY, USA, 2008. ACM.

[18] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[19] B. Ganter and R. Wille. *Formal Concept Analysis - Mathematical Foundations*. Springer, 1996.

[20] 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, New York, NY, USA, May 2008. ACM.

[21] J. Henkel and A. Diwan. Catchup!: Capturing and replaying refactorings to support API evolution. In *ICSE '05: Proceedings of the 27th International Conference on Software engineering*, pages 274–283, New York, NY, USA, 2005. ACM Press.

[22] B. Hoffmann, D. Janssens, and N. van Eetvelde. Cloning and expanding graph transformation rules for refactoring. In *Electronic Notes in Theoretical Computer Science*, volume 152, pages 53–67, Mar. 2006.

[23] P. Höfner, R. Khedri, and B. Möller. Feature algebra. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *International Symposium on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 300–315. Springer, 2006.

[24] A. Igarashi, B. Pierce, and P. Wadler. Featherweight java: A minimal core calculus for java and gj. In *ACM Transactions on Programming Languages and Systems*, pages 132–146, 1999.

[25] D. Jackson. Alloy: A leightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[26] M. Janota and G. Botterweck. Formal approach to integrating feature and architecture models. In *FASE'08: Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering*, volume 4961 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2008.

[27] C. Kästner and S. Apel. Type-checking software product lines - a formal approach. In *ASE'08: the 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, September 2008.

[28] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Language-independent safe decomposition of legacy applications into features. Technical Report 02/2008, School of Computer Sience, University of Magdeburg, 2008.

[29] C. H. P. Kim and K. Czarnecki. Synchronizing cardinality-based feature models and their specializations. In *ECMDA-FA'05: Proceedings of the First European Conference on Model Driven Architecture - Foundations and Applications*, volume 3748 of *Lecture Notes in Computer Science*. Springer, 2005.

[30] M. M. Lehman and L. A. Belady. *Program evolution: processes of software change*. Academic Press Professional, San Diego, CA, USA, 1985.

[31] F. Loesch and E. Ploedereder. Restructuring variability in software product lines using concept analysis of product configurations. In *CSMR'07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 159–170, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[32] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, USA, 1992.

[33] C. Prehofer. Feature-oriented programming: A fresh look at objects. In M. Aksit and S. Matsuoka, editors, *ECOOP'97: Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*. Springer, 1997.

[34] D. B. Roberts. *Practical analysis for refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, USA, 1999.

[35] I. Şavga, M. Rudolf, S. Götz, and U. Aßmann. Practical refactoring-based framework upgrade. In *GPCE'08: Proceedings of the Seventh International Conference on Generative Programming and Component Engineering*, Nashville, Tennessee, USA, October 2008. ACM.

[36] R. V. D. Staeten, V. Jonckers, and T. Mens. A formal approach to model refactoring and model refinement. *Software and Systems Modeling (SoSyM)*, 6(2):139–162, June 2007.

[37] G. Sunyé, D. Pollet, Y. L. Traon, and J.-M. Jézéquel. Refactoring UML models. In *UML'01: Proceedings of the Fourth International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 134–148. Springer, 2001.

[38] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In C. Consel and J. L. Lawall, editors, *GPCE'07: Proceedings of the Sixth International Conference on Generative Programming and Component Engineering*, pages 95–104, New York, NY, USA, 2007. ACM.

[39] S. Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Bern, December 2001.

[40] S. Trujillo, D. Batory, and O. Díaz. Feature refactoring a multi-representation program into a product line. In S. Jarzabek, D. C. Schmidt, and T. L. Veldhuizen, editors, *GPCE'06: Proceedings of the Fifth International Conference on Generative Programming and Component Engineering*, pages 191–200, New York, NY, USA, 2006. ACM.

[41] N. van Eetvelde. *A graph transformation approach to refactoring*. PhD thesis, Antwerp, April 2007.

[42] M. Werner. *Facilitating Schema Evolution With Automatic Program Transformation*. PhD thesis, Northeastern University, July 1999.

The Applicability of Common Generative Techniques for Textual Non-Code Artifact Generation

Johannes Müller Ulrich W. Eisenecker

University of Leipzig
Information Systems Institute
Marschnerstraße 31, 04109 Leipzig, Germany
http://www.iwi.uni-leipzig.de
{eisenecker, jmueller}@wifa.uni-leipzig.de

Abstract

Configuration or generation of software artifacts is a widely adopted approach to implement software system families e.g. by applying the generative software development paradigm. The generation of artifacts not directly related to software but rather related to a delivered software product is not widely examined. This paper discusses the applicability of three well-known software artifact generation techniques to natural language textual non-code artifacts. Therefore an overview of these techniques and adequate tools is given. The frame technology with the adaption and the abstraction concept and and the template approach of the model driven software development are examined. The tools *XFramer*, *XVCL* and *openArchitectureWare* are used to evaluate these techniques by implementing an exemplary toy use case. The experience gained by implementing the use case is presented. The three selected tools are compared with respect to the task to generate natural language texts as non-code artifacts.

Categories and Subject Descriptors D [*2*]: m

General Terms GSE, Non-code artifact

Keywords frame, MDSD, XVCL, XFramer, oAW

1. Introduction

Generative software development (GSD) [6] is one widely accepted approach to develop software intensive systems within a software system family amongst others such as model driven software development (MDSD) and aspect oriented software development (AOSD). There are mainly two development processes for software system families, namely *domain engineering* (development for reuse), and *application engineering* (development with reuse).

A software intensive system comprises more artifacts than only source files or executable program files, e.g. user-manual or man-pages but also graphics, sounds, animations, test data and so on. Such artifacts are subsumed under the term *non-code artifacts*. A special kind of such non-code artifacts are natural language textual non-code artifacts. These kinds of non-code artifacts will be subject of this paper.

When the generative paradigm is applied to develop a system family, the elementary reusable components of the solution space should be maximally reusable and minimally redundant, as the authors point out in [6]. The features of a system family member can be described by a *domain specific language* (DSL) which belongs to the problem space. The *configuration knowledge* is required for mapping the specification of a system family member to the solution space by means of a (configuration) generator. These components form the generative domain model (GDM).

The term *generator* covers not only facilities to generate code artifacts, but also to configure and parameterize available components which have well defined, asserted and possibly tested qualities. An exemplary realization of such a configuration generator in C++ could apply template meta-programming. Obviously this technique is not really suitable for generating non-code artifacts. There are other techniques which are more appropriate to generate non-code artifacts. One approach which explicitly aims to synthesize, beside code-artifacts, non-code-artifacts is the AHEAD tool suite [4]. The ancestor of it is the GenVoca approach [5], which is a methodology for creating system families.

Other approaches, which also use GenVoca as architectural style, are described as *technology projections* of the GSD. A *technology projection* is a mapping of the GDM to a specific technique, platform or programming language (see [6] for details). Some of them are also able to generate non-code artifacts, namely the projection to the adaption and the abstraction concept of the frame technology as described in [10] and the projection to the generator-framework openArchitectureWare [14]. The benefit of using one of these approaches to generate non-code artifacts is, that, if the stated approaches are used to realize a system family, one can use the existing environment to also generate non-code artifacts: No further tools are required. The paper will examine the ability of these three projections to generate non-code artifacts by using the theoretical background given by the specification of the projections.

A member of a system family may require a considerable number of related documents, e.g. program-documentation, instructions for manual system tests, customized licensing agreement or a contract between software supplier and customer to account a created software product. The listed documents raise distinct demands for a text generation system. Documentation, for example, could be created by simply assembling the specific documentation of the miscellaneous components, the system is built from. A contract on the other hand requires the creation of parts of a sentence, which are differently composed depending on the distribution model for the software system and the assembled components, to get a well-

GPCE '08 October 19-23, Nashville, Tennessee.

defined content. An example of a contract generation system can be found on [19].

The preceding example, the contract generation, could be achieved by *natural language generation* (NLG), a field of research in the area of artificial intelligence and computer linguistics [20, p. 1]. Its aim is to create natural language texts out of non-linguistical representations—e.g. out of a database. Another approach is to create texts by assembling text components. In [20, p. 4] the thesis is stated that every functionality which is realized by a NLG-system can also be realized by assembling text components. For this reason this paper focuses exclusively on the text-assembly approach, while NLG-techniques will be not pursued.

A human readable document must be well presented to make the contained information easily accessible to human readers. To format the output of a generation run, amongst others, there are the following techniques available:

- Tag the document by LaTeX commands. A TeX processor translates the tagged document into a human readable form in natural language.
- Tag the document by XHTML[1] commands and format it with CSS[2].
- Create a XML[3]-document and format it with XSL-FO[4].

If the approach with the LaTeX tags is used, the document creation process is a multi level generation process. At a first stage, the document with LaTeX tags is created. At a second stage, a TeX processor transforms the representation created before into a document ready for press. Because of that, LaTeX sometimes is referred to as a DSL for the domain of document creation [16].

2. Techniques for Text Generation

Not all technology projections to the generative software development paradigm which are useful to generate software artifacts are useful for generating non-code artifacts. Applying template meta programming in C++ for example is an effective way to create configuration generators to configure software components in C++. In fact this technique uses the features of the C++ language and requires adequate components for assembly. Therefore it can not be reasonably used to generate natural language texts. Furthermore a technique for non-code artifact generation must provide a possibility to modularize text blocks to handle potentially complex non-code artifacts within a system family. These prerequisites are provided by the adaption and the abstraction concept of the *frame* technology [2] and the template approach of *model driven software development* used to transform a model to code [8, 14].

2.1 Frames

The idea of *frames* was developed in the research area of artificial intelligence. In the seminal work "*A Framework for Knowledge Representation*" [13] Marvin Minsky explains a system to describe the mental processing of concepts appearing in the real world.

A frame defines constant values for a concept. These constant values are part of all instances of a frame. Beside this a frame contains also variable parts which are organized in so called *slots*. A slot of a frame can be either an instance of an other frame or finally a terminal value. Because of this relationship between frame instances a complex frame hierarchy can be composed which is useful to represent or analyse concepts of the real world [9, p. 120].

In 1987 Bassett describes in his seminal work [2] at the first time the usage of frames to foster reuse of software components. Independently of this work the company *Delta Software Technology* developed a technology to generate software components based on the idea of Minsky.

In [11, p. 55] the approach of Bassett is called the *adaption concept* and the solution of *Delta Software Technology* is called *abstraction concept*.

2.1.1 Adaption Concept

In the adaption concept frames are stepwise specialized. From general frames more special frames will be assembled. Higher level frames *adapt* lower level frames. The more general frames are customized to the circumstances and requirements of the more special frames [3, p. 88]. A frame will be changed on its slots, which are the variation points of the frame hierachy. A frame has default values for variation points, so an adapting frame must only change the slots if there are special requirements which differ from the defaults [3, p. 89]. Furthermore a frame at the adaption concept did not have any mutable state. So its processing compares to the processing of variables at the functional programming paradigm.

2.1.2 Abstraction Concept

Within the abstraction concept frames are not part of an other frame, rather they are instantiated. The created instances will be referenced from other frame instances. In this way a hierarchy of frame instances is constructed. It is possible to instantiate a frame more than once. Every instance gets its own set of slot-values and references to other frames. This feature of the abstraction concept resembles the class/instance-scheme of object-oriented programming. Thereby the abstraction concept is similar to object-oriented programming.

2.2 Templates

The MDSD aims at automatic generation of executable software out of a formal model [24, p. 11]. A formal model represents rules which make a statement about the models meaning. In this context a model could be a class diagram in the *Unified Modeling Language* (UML). Moreover also other types of models e.g. textual models can be used. Automatic generation means that the source code is generated without manual intervention. No modifications may be applied to the generated source code because the model adopts the role of code.

Underlying every concrete model is a meta-model. A meta-model is a formal description of a domain in which a system is generated. It defines the abstract syntax of the models. The abstract syntax consists of the meta-model elements and their relation amongst each other [24, p. 29]. A transformation is described by means of a template language. A template consists of static text which contains tags on specific spots. During the generation process these spots will be assigned text according to a input model [24, p. 146]. The tags can contain additional instructions which are executed while the model is processed, e.g. to modify an input string. While templates are defined on the basis of a meta-model, the generated text will be created according to a concrete model. In this approach the templates are the reusable components of the solution space.

MDSD aims at generating code artifacts. Because code is generated mainly as text, it is also possible to apply MDSD for generating textual non-code artifacts.

2.3 Techniques not Examined

The techniques mentioned before are not the only options to generate text. Subsequently some other techniques are listed. Every technique has its own weakness which renders it not suitable for the

[1] XML Hypertext Markup Language

[2] Cascading Style Sheets

[3] Extensible Markup Language

[4] Extensible Stylesheet Language-Formating Objects

specific purpose of text generation. In [24, p. 149ff] it is remarked that common programming languages, such as Java or C# can be used to create generators. However this idea is rejected, because of language related problems such as processing of strings which requires to use escaped sequences for special characters. Moreover, language structures which are not designed for text generation have to be used for the generator. Thereby it is hardly recognizable, how the result will be structured. In [24, 147ff] another alternative is discussed, namely the application of XSLT[5] to build a generator. The availability of *XPath* as a powerful navigation language is pointed out as a special advantage. *XPath* supports the navigation of basically every object structure. But an obvious disadvantage is the hardly readable syntax of XML [6], which has to be used to describe the transformation in XSL files. Preprocessors like those used in C/C++, can be used independently from the compiler as well [24, p. 143f]. Therefore it is basically possible to realize a text generator. A preprocessor is often used for small replacements of texts, for instance the value of a constant. A frequent consequence is that preprocessors are mostly not Turing-complete, they lack especially control structures for iterations. These are required in order to implement more complex generators. For this reason it is not practical to realize the generation of textual non-code artifacts with a preprocessor.

3. A Use Case for Text Generation

As use case to gain experience with the tools a family of documents is implemented. A document of these family accompanies a pizza and contains the following elements:

- The address and the title of the customer
- The price of the ordered pizza
- A list describing the ingredients. The list also contains a warning if an ingredient could be allergenic.
- A complimentary close customized to the order. Friends of chili get the spanish wish *¡buen provecho!*, garlic fans get the italian wish *Buon appetito* and if no or both extras are chosen, the close will be the english phrase *enjoy your meal*.

The document is generated to be suitable to the corresponding pizza. The example covers a wide range of imaginable non-code artifact generation scenarios. So one can gain experience with the utilized tools. First the price contains no static contents: the generator is able to compute all values. Second the list of ingredients consists of distinct text blocks which will be assembled with respect to a certain order. The third part, the complementary close, is a mixture of the both preceding variants. The closings will not be changed but a generator must choose the right text block with respect to the chosen extras.

4. Tools for Text Generation

In the last section, techniques for textual non-code artifact generation were examined. This section focuses on tools for implementing these techniques. There is more than on implementation for each of the preceding introduced techniques. Tools to realize one of the two frame concepts are presented in [11, p. 56]. The template approach is realized by MDSD tools. An overview of available tools is provided in [7, p. 622]. The evaluation of these techniques will be based on open source or freely available tools so that the following descriptions can be easily reproduced[7]. However it is important that the used tools adequately support the examined technique. This requirements are fulfilled by the following tools: For the abstraction concept *XFramer* is chosen and for the adaption concept *XVCL* is selected. To examine the template approach *openArchitectureWare* will be applied.

In addition to the aforementioned requirements, there are also the following properties of the tools expected, to realize a family of non-code artifacts:

- It must be possible to define a domain specific language.
- It must be possible to preserve a certain locality of the parts of the GDM to achieve a sufficient maintainability
- Control structures for processing text modules must be available.
- Debugging facilities must be available.
- It should be possible to externalize huge text blocks in order to improve the readability of the generator modules.
- It must be possible to exactly control the output of white spaces to get the expected output by the TEX-processor.
- Tools, i.e. editor, must be available for creating the text modules.

4.1 XFramer

XFramer is available from [23] as freeware for Linux and Windows for non-commercial purpose. It is used to extend the programming languages C++, C# and Java with the capability to process frames [11, p. 55] Nevertheless the compiler is still required to use it as a full frame processor. *XFramer* works as a preprocessor, which translates the frame definitions to valid source code of the language of the used compiler. Even if one of the languages C++, C# or Java is used to process the frames, it does not imply that only one of this languages can be used exclusively. In fact all textual representation—any programming language, HTML, XML or natural language are imaginable—of information can be processed by the tool [11]. The tool extends the supported languages with new elements.

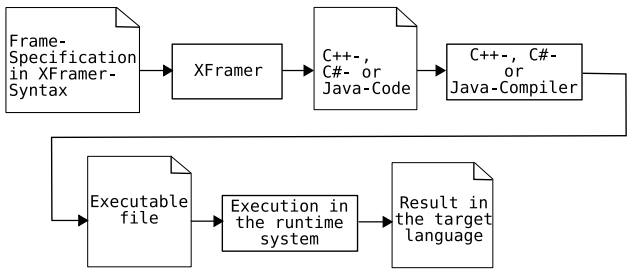


Figure 1. Workflow when applying XFramer [11, p. 57]

Figure 1 depicts the flow of a XFramer run. The XFramer preprocessor reads frame specifications and transforms them into source code modules—classes in the case of the chosen host programming language. The output of the preprocessor is processed by the compiler of the used host language. In this step the real generator is created. This generator is executed in the runtime system and creates the result in the target language. XFramer creates the generator in the selected programming language, so it is fully amenable to the debugger of this language [10, p. 80]. In addition all libraries available in the host language can be used to build the generator. So it is e.g. imaginable to make the generator configurable by XML using a XML processing library like Xerces [10, p 80].

[5] Extensible Stylesheet Language Transformations

[6] Extensible Markup Language

[7] The complete source-files of all three implementations can be downloaded as a zip archive from [15]

4.2 XVCL

XVCL is an acronym for *XML-based Variant Configuration Language* and names a language to define frames according to Bassetts adaption concept. Frame definitions in XVCL are processed by the tool *XVCL processor* [12, p. 1]. Language, tool as well as a methodology for domain analysis are developed by the *National University of Singapore*. XVCL processor can be freely downloaded from [1] under the permission of the *GNU Lesser General Public License* (LGPL) version 2.1.

XVCL is a XML-based language and is shipped with a *Document Type Definition* (DTD). It is written in Java so it executes on many host systems. The tool is invoked with the specification frame and starts the generation process in which all frames directly or indirectly referenced by the specification frame are stepwise processed and assembled to one or more output documents. There lies one key difference to *XFramer*, which instantiates the frames, so the content can be changed during the generation process.

If the parameter `-B` is passed to the execution of *XVCL* superfluous white spaces are removed. Otherwise all blank lines of the frames are written into the output documents. Lines with XVCL commands become blank lines. An exemplary invocation of the tool is `java -jar xvcl.jar -B pizza-document.xvcl`. A detailed description of the XVCL-tool can be found in [18].

4.3 OpenArchitectureWare

OpenArchitectureWare (oAW) is a generator framework from the area of model driven software development. It covers a language family to check and transform models. The language *Xpand* is used to describe model to text transformations. The framework offers editors and plug-ins for the eclipse platform but it is also possible to use it independently [17]. It is available under the terms of the *Eclipse Public License* (EPL).

Within the subproject *Xtext* a tool is developed which allows to define the syntax of a DSL with a sort of *Extended Backus Naur Form* (EBNF). With this definition a model is generated which represents the abstract syntax tree (AST) of the language. In addition an editor for the eclipse platform is generated which assists the user with error checking, syntax highlighting and so on.

It is not possible to describe all features of the oAW project here. For details the reader is referred to the manual of oAW [8].

4.4 Comparison of the Tools

Essential properties of techniques and of tools for generating non-code artifacts in natural language are enumerated in section 2 and 4 respectively. These properties will be studied more closely with respect to the selected tools now.

Definition of the DSL *XFramer* uses a host language. Hence the only restriction implementing a DSL is given by the selected host language. As it is the case in the described example the DSL is embedded into the host language. If this approach is not powerful enough, it is imaginable to externalize the DSL and include a parser which reads the specification. *XVCL* on the other hand only allows to define a rudimentary DSL in the specification frame. In the implementation of the use case with *XVCL* multi-valued and single-valued variables are used to realize the DSL. Another approach could be to define a DSL as an XML lanugage and use XSLT[8] to transform a specification into a valid XVCL specification frame. The *oAW* solution in this paper is showing the realization of a textual DSL with the Xtext tool. It has the ability to create expressive DSLs. A specification in the DSL can be written in its own editor which has the capability to check the specification and highlight keywords.

[8] Extensible Stylesheet Language Transformations

Locality of the parts of the GDM With *XFramer* it is possible to define one configuration frame which contains all the knowledge to configure the elements of the solution space given a specification from the problem space. Another possible approach with *XFramer* is to define intelligent frames with each containing some specific part of the configuration knowledge and elements of the solution space. Thus some elements of the configuration knowledge can be reused but the locality of the components of the configuration knowledge degrades. Because of the stepwise adoption of the frames in *XVCL* only the second approach is feasible. The framework *oAW* is able to use model to model transformations. So the configuration knowledge is localized in the transformation rules. If the direct model to text generation approach is used the maintainability degrades.

Available control structures *XFramer* can use all the facilities of the host language, so very expressive solutions are possible. In contrast *XVCL* provides only a few basic commands which permit the definition of the logic, but they are much harder to use—also because of the XML syntax—than the control structures of a general purpose language. The Xpand template language provides also just rudimentary support for control structures. But it is possible to define the logic in a functional sub-language of the oAW project called Xtend which has the ability to call statically defined Java-methods. Thereby all the power of the Java language is available.

Debugging facilities If errors are detected during the generation process it would be helpful to use a debugger to understand the generation process. In fact *XFramer* uses a host language, hence the debugging facilities of this language are available. If there is any error at the generation process with the *XVCL* tool it produces error messages. Another approach to support debugging is to produce messages which the generator displays during the generation process. At present, other debugging facilities do not seem to be available. The *oAW* project contains debugging facilities to debug templates, workflows and transformations.

Externalize and modularize text blocks If huge text blocks are used, it would be useful to define them externally and reference them by a generator module. This is directly supported only by *XFramer*. Using *XVCL* there is no possibility to do so. Basically the same restriction applies to *oAW* but this functionality can be realized with the capability to use Java-methods. Related to this problem is the ability to modularize text blocks. All tools allow to distribute the modules over several files.

Control white spaces Even if the final document is typeset by LaTeX it is nevertheless important that the resulting document does not contain superfluous white spaces because they could have a meaning. At LaTeX e.g. an empty line results in a new paragraph in the target document. So the techniques must provide a facility to control the output of white spaces. Using *XFramer* all blanks are outputted as defined in the frame. So the output of the blanks can not be well controlled. The *XVCL* tool has the ability to remove superfluous white spaces but if there are intended blank lines they are also removed. The lines in the templates of *Xpand* which contains escaped commands will be removed if a minus is noted at the end of the escaped sequence.

Tool support The creation of text modules for one of the tool could be made more convenient by having any tool support, e.g. a text editor with syntax highlighting. At present, it seems that there is no editor available which assists the creation of frames for *XFramer*. But perhaps it is more adequate to use a common text editor which supports the selected host language. *XVCL* is a XML dialect so any XML editor can be used. Because a DTD is given, some editors can even perform syntax highlighting and code completion. *oAW* is well integrated into the eclipse platform. There

are editors, wizards and other plug-ins available, which make the usage of the different languages convenient.

Table 1. summary of the comparison of the tools

Criteria	XFramer	XVCL	oAW
Defining DSLs	−	−−	++
Locality of parts of the GDM	++	−−	++/−[a]
Available Control structures	++	−−	++/−−[b]
Debugging facilities	++	+	+
Externalize text blocks	++	−−	+[c]
Modularize text blocks	++	++	++
Control white spaces	−−	+	+
Tool support	−−	++	++
Overall effort to implement the use case	−	++	+

(−−) bad to (++) good

[a] Good, if model to model transformation are used, otherwise bad.

[b] By using Xtend all possibilities of Java are available.

[c] By using Xtend and Java.

5. Conclusions

This work has analyzed three technology projections for their applicability to generate textual non-code artifacts in natural language. The techniques and tools which realize them were presented. The three tools are used to implement a use case. The three tools were compared with respect to the aforementioned criteria. As table 1 suggests, the toolset of oAW is well suited to realize the text generator. But it needs some effort to set up the environment to get the generator run (define a grammar, install the plugin and so on). The fastest way to implement the text generator of the use case provides XVCL. If a simple DSL suffices or there is an other way to configure the specification frame XVCL is a lightweight alternative to oAW. XFramer was somewhat harder to use then the other two approaches. But if complex decision logic is to be implemented, it can be an alternative because of the integration in a host language. To decide which tool is best suited for a given environment one must check the ease of integration in a present tool chain to generate system family members. Therewith it is possible to use one and the same DSL to specify the software system and the adequate textual non-code artifacts in natural language.

This paper only examines textual non-code artifacts in natural language. Another survey should reveal other relevant types of non-code artifacts. The result of this survey could be organized in a taxonomy of non-code artifacts. With such a taxonomy it would be possible to examine generation tools for all other types of non-code artifacts.

Real life use cases probably contain much harder requirements to textual non-code artifacts in natural language than the presented toy use case. To implement such requirements some ideas can be found in work related to NLG ([20], [21] and [22]).

As the implementation of the use case demonstrates, the conceptual framework of the generative software development [6] is also well-suited to generate textual non-code artifacts in natural language. With this aspect in mind, further technology projections specialized for generating non-code artifacts could be developed. As the usage of a multistage generation process in the use case demonstrates (e.g. XVCL and TEX processor), the usage of more than one tool to realize the generation is a promising approach. A further example would be to generate graphics by producing svg-files (which are text-based) and then use one of the *svg*-tools to convert it to a required file type.

This paper has shown that existing techniques and tools are applicable to generate (textual) non-code artifacts. To get a deeper insight the aforementioned next steps should be pursued.

References

[1] Xvcl download: `http://fxvcl.sourceforge.net`.

[2] P. G. Bassett. Frame-based software engineering. *IEEE Software*, 4(4):9–16, 1987.

[3] P. G. Bassett. *Framing Software Reuse: Lessons from the Real World*, volume 1 of *Yourdon Press Computing Series*. Yourdon Press, Prentice Hall, 1997.

[4] D. Batory. The road to utopia: A future for generative programming.

[5] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, 1992.

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

[7] K. Czarnecki and S. Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Syst. J.*, 45(3):621–645, 2006.

[8] S. Efftinge, P. Friese, A. Haase, C. Kadura, B. Kolb, D. Moroff, K. Thoms, and M. Völter. *openArchitectureWare User Guide*. n.p., 1 edition, September 2007.

[9] U. W. Eisenecker and R. Schilling. Zum Entwickeln entwickelt. *iX*, 10:114 – 121, 2002.

[10] M. Emrich. Generative Programming Using Frame Technology. Diploma thesis, University of Applied Sciences Kaiserslautern, 2003.

[11] M. Emrich and M. Schlee. Codegenerierung mit XFramer und Programmiertechniken für Frames. *Objektspektrum*, 5:55 – 61, 2003.

[12] S. Jarzabek, P. Bassett, H. Zhang, and W. Zhang. Xvcl: Xml-based Variant Configuration Language.

[13] M. Minsky. A Framework for Representing Knowledge. Technical report, MIT, Juni 1974.

[14] J. Müller. Technikprojektion zur generativen Programmierung mit openArchitectureWare. Diploma thesis, University of Leipzig, Marschner Straße 31, 04109 Leipzig, Germany, May 2008.

[15] J. Müller and U. W. Eisenecker. Zip-archive with the sample code. `http://w3l.wifa.uni-leipzig.de/sw/public/ncag.zip`.

[16] N.A. Domain Specific Language. online, Oktober 2007. `http://c2.com/cgi/wiki?DomainSpecificLanguage`.

[17] N.A. openArchitectureWare. homepage, nov 2007. `http://www.openarchitectureware.org/`.

[18] N.A. Xml-based Variant Configuration Language (xvcl), 2007. `http://sourceforge.net/project/showfiles.php?group_id=58966&package_id=54953&release_id=305328`.

[19] N.A. Creative Commons. online, 2008. `http://creativecommons.org/license/`.

[20] E. Reiter and R. Dale. Building Applied Natural Language Generation Systems. *Journal of Natural Language Engineering*, 3(1):57–87, 1997.

[21] E. Reiter and R. Dale. *Building Natural Language Generation Systems*. Cambridge University Press, 2000.

[22] E. Reiter, S. Sripada, and R. Robertson. Acquiring correct knowledge for natural language generation, 2003.

[23] M. Schlee. Xframer 1.45 download: `http://www.geocities.com/mslerm/downloads.html`.

[24] T. Stahl, M. Völter, S. Efftinge, and A. Haase. *Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management*. Dpunkt Verlag, Heidelberg, 2 edition, 2007.