Delaying decisions in variable concern hierarchies

Jörg Kienzle, Gunter Mussbacher, Philippe Collet, Omar Alam

Delaying Decisions in Variable Concern Hierarchies

Jörg Kienzle
Gunter Mussbacher
McGill University, Montreal, Canada
{Joerg.Kienzle | Gunter.Mussbacher}@mcgill.ca

Philippe Collet
Université Côte d'Azur, CNRS, I3S
Sophia Antipolis, France
Philippe.Collet@unice.fr

Omar Alam
Trent University,
Peterborough, Canada
omaralam@trentu.ca

Abstract

Concern-Oriented Reuse (CORE) proposes a new way of structuring model-driven software development, where models of the system are modularized by domains of abstraction within units of reuse called concerns. Within a CORE concern, models are further decomposed and modularized by features. This paper extends CORE with a technique that enables developers of high-level concerns to reuse lower-level concerns without unnecessarily committing to a specific feature selection. The developer can select the functionality that is minimally needed to continue development, and reexpose relevant alternative lower-level features of the reused concern in the reusing concern's interface. This effectively delays decision making about alternative functionality until the higher-level reuse context, where more detailed requirements are known and further decisions can be made. The paper describes the algorithms for composing the variation (i.e., feature and impact models), customization, and usage interfaces of a concern, as well as the concern's realization models and finally an entire concern hierarchy, as is necessary to support delayed decision making in CORE. The approach is evaluated on a crisis management case study.

1. Introduction

Every decision made during software development, from high-level, architectural choices down to platform-specific implementation choices, determines how well the software complies with the requirements it was initially set out to fulfill. This situation is complicated further because requirements often change. Non-functional requirements are often not known until the execution platform has been determined, and can vary drastically depending on last-minute implementation choices. In the context of Software Product Line (SPL) development [23], different products might have to adhere to different non-functional requirements. Finally, for adaptive systems, the changing context in which the software operates imposes different requirements as well.

The problem of incomplete or varying requirements is accentuated further in the context of software reuse [18; 19]. The goal of a reusable entity is to be reused multiple times in different contexts [28]. For reusable entities that are provided as part of a library, the contexts in which they are going to be used are typically not known in advance. Hence the precise requirements for the entity, in particular the non-functional ones, are unknown or variable at best. Therefore, following some previous works on variability and composition [15; 27; 28], we advocate that a versatile reusable entity should be designed in such a way that it encapsulates several functional and implementation variants. This would allow a developer to choose the variant that best fits the specific functional and non-functional requirements of the context in which the reusable entity is being reused.

Unfortunately, even if a reusable entity has been designed to enable adaptation of the encapsulated functionality to any context of reuse imaginable, the entity's reuse potential is still limited if the decision to use a specific variant of the reusable entity needs to be made prematurely. This is, e.g., the case with variable components [27], where a component must be completely configured to be reused. However, determining the best variant of a reusable entity is only possible when all requirements are known. This is far from being the case when developing a piece of software that is itself reusable, since again it is not known in what context the reusable entity will be used in the future. Therefore, not all decisions required to reuse a variant should be made when the entity is reused. To maximize reuse, we need to allow a developer of a reusable entity to delay decision making without holding up development when reusing other entities until additional requirements for decision making are known.

In this paper, we propose an approach for delaying decisions during software development that enables this reuse potential. The approach combines ideas from variability modelling and configuration, model-driven engineering, and advanced separation of concerns, and internally relies heavily on software composition to incrementally generate more specific versions of the reusable entities whenever additional

decisions are made. Concretely, we show how we integrated support for delayed decision making into Concern-Oriented Reuse (CORE) [2], and present, based on well-defined interfaces, the composition algorithms required for incrementally adapting reusable entities as additional decisions are made.

The outline of the paper is as follows. Section 2 illustrates the idea of delayed decision on a running example, while introducing the main concepts of CORE and positioning the proposed approach with respect to the main areas of related work. Section 3 details the different composition algorithms used to support delayed decision making. Section 4 presents reuse/decision delaying metrics gathered when designing a case study crisis management system. Section 5 elaborates on related work, and the last section draws the conclusions.

2. Synopsis of CORE

This section presents an overview of Concern-Oriented Reuse (CORE) [2], a new reuse paradigm that we have extended in this paper with support for delaying decisions, and illustrates the main concepts of CORE and the need for supporting delayed decision making with a running example.

2.1 Background on CORE

In CORE, software development is structured around modules called concerns that provide a variety of reusable solutions for recurring software development issues. Techniques from Model-Driven Engineering (MDE), SPL engineering, and software composition (in particular aspect-orientation) allow concerns to *form modular units of reuse that encapsulate a set of software development artifacts, i.e., models and code, describing all properties of a domain of interest during software development in a versatile, generic way* [2]. The models within a concern can span multiple phases of software development and levels of abstraction (from requirements, analysis, architecture, and design models to code). Concerns decompose software into reusable units according to some points of interest [10; 21] and may have varying scopes, e.g., encapsulating several authentication choices, communication protocols, or design patterns.

The main premise of CORE is that recurring development concerns are made available in a concern library, which eventually should cover most recurring software development needs. Similar to class libraries in modern programming languages, this library should grow as new development concerns emerge, and existing concerns should continuously evolve as alternative architectural, algorithmic, and technological solutions become available. Applications are built by reusing existing concerns from the library whenever possible, following a well-defined reuse process supported by clear interfaces. The same idea is applied to the development of concerns as well: high-level/more specific concerns can reuse low-level/more generic concerns to realize the functionalities they encapsulate. In the end, the software architecture of software developed with CORE takes the form of a concern hierarchy (directed, acyclic graph), thus supporting hierarchical modularity [5].

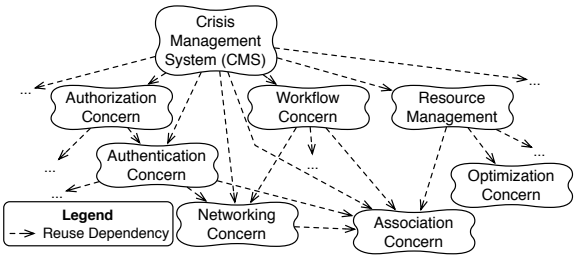

Figure 1. Partial Concern Hierarchy of the CMS

Throughout this paper, we use a product line of crisis management systems (CMS) as proposed in [17] to illustrate concern hierarchies, delaying of decisions, and composition algorithms that support delayed decision making. Crisis management involves identifying, assessing, and handling a crisis situation. A CMS facilitates this process by allocating and manages resources, and providing access to relevant information to authorized users in a timely and reliable manner. Figure 1 depicts parts of the concern hierarchy of the CMS case study that is presented further in Section 4.

2.2 Running Example: Resource Management

Resource management (RM) is required in many business applications. For example, in the CMS, *resources* such as vehicles and emergency personnel *need to be* tracked and *assigned to tasks, e.g.,* rescue missions. Because RM is a recurring functionality needed in many applications, a developer might want to design and implement the RM functionality within a reusable artifact, i.e., within a concern. This concern will have to represent the diverse usage of such a functionality and to implement the needed variants internally.

Moreover, to accelerate the development of the concern, the developer should in turn be able to reuse existing design concerns from the concern library while tailoring them for the specific reuse context. For example, the library provides a concern *Association* whose functionality could be configured and used to associate tasks with resources.

The following subsections highlight the need for delaying decisions by means of this use case: *Resource Management* reusing *Association*. We explain the CORE reuse process from the perspective of the developer of *RM*, and in doing so motivate the different composition algorithms that are required to support delaying of decisions. The details of the different algorithms are then presented in Section 3.

2.3 Capturing the Features of RM

A CORE concern organizes all relevant variations/choices that are available for reuse in form of a *feature model* [14], taking this *de facto* standard for variability modelling.

The feature model expresses the *closed* variability that the designer of the concern encapsulated within the concern similar to what is done in software product lines for a specific application domain. This part of the concern can be directly related to what was proposed for variable components [27] or more recently with variable modules [15].

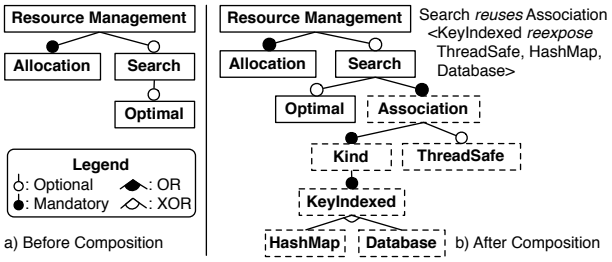

Figure 2. *Resource Management* Feature Model

Figure 2(a) shows the feature model for our *RM* concern. The main functionality of *RM* is to keep track of the availability of *resources*. Resources can be allocated to *tasks*, in which case they are marked as unavailable until they are released again. This functionality is provided in the mandatory feature *Allocation*. When each resource may exhibit different *capabilities*, *RM* offers the possibility to search for available resources with *desired* capabilities in the optional feature *Search*. An *np*-complete algorithm that determines the optimal solution for assigning resources with capabilities to tasks is provided as part of the feature *Optimal*.

2.4 Reusing the *Association* Concern

In the design/implementation (called the *realization* in CORE terminology) of the different features of *RM*, our developer needs to maintain data structures that associate resources with tasks, resources with capabilities, etc. Since dealing with such relationships is a recurring software design concern, our developer browses the concern library and discovers the *Association* concern, and decides to reuse it.

Reusing the solutions encapsulated inside a CORE concern is streamlined by a well-defined reuse process that is based on three steps: 1) choosing the desired solution among the available alternatives (see Section 2.4.1), 2) adapting the provided realization of the chosen solution to the specific reuse context (see Section 2.4.2), and 3) using the chosen, adapted realization for your own purpose (see Section 2.4.3).

2.4.1 Step 1: Choosing the Desired Variant

The developer who realizes the *Allocation* feature of *RM* first needs to know what different solutions the *Association* concern provides. To this aim, the developer consults the feature model of *Association* (left side of Figure 3), which lists the available solutions encapsulated by *Association*. Furthermore, *Association* also provides the list of non-functional goals that differentiate among the possible solutions (right side of Figure 3). To determine the most appropriate solution to be reused in the context of *RM*, the developer *interacts* with the variation interface of *Association*. Whenever the developer selects features in the variation interface, the developer is informed about the non-functional impacts of his choice with the help of relative satisfaction values shown next to the goals (e.g., 100% for *Increase Performance*).

It turns out that for realizing the *Allocation* feature of *RM*, the link between `Task` and `Resource` can be realized by any of the variants of *Association*. Which variant is best,

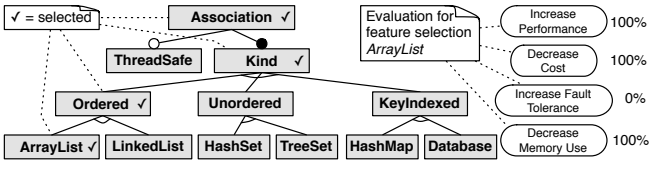

Figure 3. Variation Interface of *Association*

however, depends on the context in which *RM* is eventually used. E.g., the choice of *ArrayList* would yield the best performance and lowest memory usage of the encapsulated variants at no cost (the resulting satisfaction values of this choice are shown in Figure 3). However, in a multithreaded program, a correct association solution that connects tasks and resources must deal with concurrent access (optional feature *ThreadSafe*), which impacts performance negatively, and hence should only be chosen when necessary. Finally, a choice of *Database*, while increasing cost and memory usage and decreasing performance, would store resource allocations persistently and hence increase fault tolerance.

Since our developer does not know whether the applications that are going to reuse *RM* are going to be multithreaded or not, or whether in their context saving cost has higher importance than increasing fault tolerance, it is impossible to make the right decision at this point. With the approach we are proposing in this paper, the developer can now defer decisions that cannot be made at the moment of reuse to a later time. The precise rules for deferring of decisions are explained in Section 3.3. Essentially, the developer makes the decisions that can be made at this point by a) selecting features that are absolutely needed, b) de-selecting features that cannot be applied in this reuse context, and c) deferring any optional or alternative choices to a later point. Without the ability to defer the reuse decision, the developer would be forced to select a variation with limited knowledge, making it likely that the decision turns out to be suboptimal when *RM* is reused. This clearly illustrates that some decisions should be delayed while handling variability between reusable artifacts. This is notably not possible in variable components proposed by van der Storm [27], in which a component must be fully configured before reuse.

After the developer has made his selection (called a *configuration* in CORE), a realization of the chosen variant is created by composing the models encapsulated in the concern that realize the selected features to yield a *user-tailored set of realization models* of the concern for the specified configuration. In our specific example, our developer would simply select the mandatory features (*Association* and *Kind*), and defer the decision for the remaining features. As a result, the realization models of *Association* and *Kind* are composed. The algorithm that composes the realization models of a concern according to a given configuration, taking into account feature interactions and delayed decisions, if any, is explained in Section 3.2 (Composing a Concern).

Variation Interface Definition: To support this first step of the reuse process, every CORE concern provides a *vari-*

ation interface that *expresses the closed variability offered by the concern* as in an SPL *using a feature model* as explained above. Additionally, *the impacts of selecting a feature on non-functional goals and qualities are specified with an impact model* that is expressed using a variant of the Goal-oriented Requirement Language (GRL) [11]. Supporting delayed decision making in CORE means allowing a developer to define partial configurations when reusing a concern, which in turn has as a consequence that the reused concern still exhibits variability. Hence, the variation interface of the reused concern has to be composed with the variation interface of the reusing concern to allows its users to complete the configuration. To this aim, Section 3.3 explains how delayed decisions of reused concerns are reexposed in the variation interface of a reusing concern by composing the concerns' feature models, and Section 3.4 presents details on how impacts of reused concerns are composed with the impact model of the reusing concern.

2.4.2 Step 2: Adapting the User-Tailored Realization

The realization models encapsulated within a CORE concern are written as generically as possible in order to maximize reuse. In other words, they often only *partially define structural and behavioural model elements* to enable *open variability*, i.e., adaptation to unforeseen contexts. Similar to generic classes in object-orientation, developers can adapt the functionality provided by a realization to a specific context by completing the partial elements with context-specific structure and behaviour. To do this, the developer has to create mappings from the generic model elements of the reused concern to model elements of the reusing concern.

For example, the customization interface of the design class diagram realization for the variant <*Association,Kind*> of *Association* contains two partial classes: `|Data` and `|Associated`. Our developer proceeds by mapping them to classes in the design class diagram that realizes *Allocation* as follows: `|Data` → `Task`, `|Associated` → `Resource`. Based on these mappings (called a *customization* in CORE), the user-tailored generic realization models are adapted by composing them with the reusing concern's realization models. The algorithm that accomplishes this composition is provided by the realization language (Compose Models).

Customization Interface Definition: To support this second step of the CORE reuse process, every CORE realization model provides a *customization interface,* which *designates the generic, partially defined structural/behavioural model elements* that enable open variability *and how many times*, minimally and maximally, *these elements have to be mapped to model elements in the reusing model*. The rules for composing customization interfaces of realization models are presented in Section 3.5.

2.4.3 Step 3: Using the Adapted Realization Models

The third and last step of the CORE reuse process is to access the functionality, i.e., the structure and behaviour, of the chosen, customized variant of the reused concern. In our example, in the class diagram that realizes the feature *Allocation*, thanks to the reuse of *Association*, the `Task` class now has several public operations that can be used to `add`, `remove`, and `iterate` through resource instances associated to a `Task` instance. It is therefore possible for our developer to design and implement, for example, an operation `allocate(Resource r)` for the class `Task`. `Allocate` first checks that the resource `r` is not already allocated. If not, it sets `r` as allocated and then calls `add`, a functionality provided by the reused *Association*, to associate `r` with the task.

Thanks to the support for delaying of decisions, it was possible for our developer to continue development, i.e., in our case continue to design the functionality of *Allocation*, although the decision of which realization eventually will implement the association between tasks and resources has not been made yet.

Usage Interface Definition: To support this third step of the CORE reuse process, every CORE realization model must provide a *usage interface* that *designates the realization model elements that can be accessed by the context in which the concern is reused*. Any realization model element that has its visibility set to *public* is part of the usage interface and can therefore be used. Other model elements remain encapsulated within the concern. The rules for composing usage interfaces are presented in Section 3.6.

2.5 Finalizing Delayed Decisions for Association

At some point during development, there will be additional information available to complete decisions about optional or alternative features that have been delayed at the moment of reuse. For example, in our case, the realization of the *Search* feature of the *RM* concern requires an association between resources and tasks that is *indexed by capabilities*. If selected, the *Search* feature therefore makes the additional decision to select the feature *KeyIndexed* from *Association*, and maps `|Key` → `Capability` (see Figure 4).

Eventually, the *RM* concern is, for example, reused in the context of the CMS product line. The CMS is a system that is highly concurrent, e.g., due to the fact that multiple missions execute concurrently. It therefore is essential that *RM*, which is reused in the *Vehicle & Personnel* feature realization, can handle concurrency, which means that the *Association* realization must be thread safe (which can be achieved by selecting the feature *ThreadSafe*). Also, fault tolerance in the context of a CMS is more important than saving costs, and hence the feature *Database* is the best realization choice. Figure 4 illustrates the incremental decision making for the *Association* concern across features and concern boundaries.

To allow the user of *RM* to make decisions for the features of *Association* that were delayed at the moment of reuse, the remaining variability from the variation interface of *Association* is reexposed as variants in the variation interface of *RM*. The feature model composition algorithm that achieves this is explained in Section 3.3. To enable tradeoff analysis on the impacts that the delayed decisions have, the impact

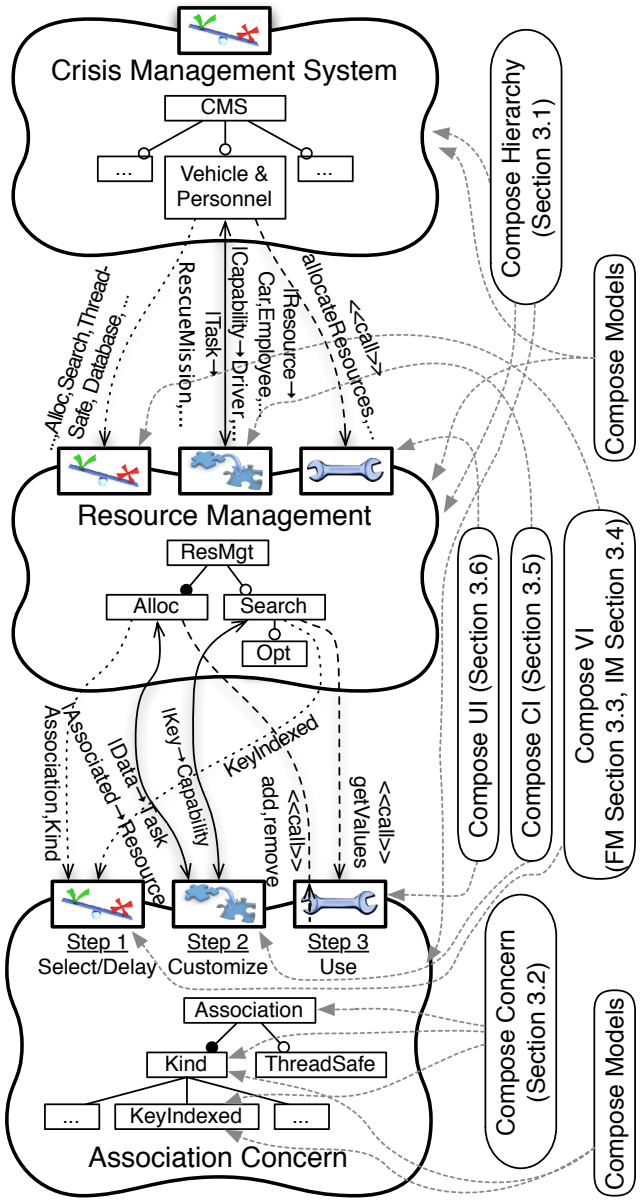

Figure 4. Overview of Composition in CORE

model of *Association* is composed with the impact model of *RM* as explained in Section 3.4.

With these interface composition mechanisms in place, *variable concern hierarchies* can be built, where one concern reuses another concern, which reuses another concern, etc... Delayed decisions from a low level can be made available at a higher level. At any level, a concern user can make additional decisions about lower-level delayed decisions, i.e., make optional features mandatory when required by the current level, or remove optional features entirely in case the corresponding realization is incompatible with the realization of the current level. This goes on until at some point all delayed decisions for a given reuse have been made. In our example, this is the case for the *Association* reuse

Algorithm 1 Composing a Concern Hierarchy

1: **function** COMPOSEHIERARCHY(c,conf,cust)
2: $|rm_c$: COREModel = composeConcern(c,$conf|_c$)
3: **for all** $r \in rm_c.mR$ **do**
4: $||conf_r$ = composeConfigurations($conf|_{r.reuse.rC}$, $r.selConf$)
5: $||cust_r$ = composeCustomizations($cust|_{r.reuse.rC}$, $r.comps$)
6: $||rm_r$: COREModel = composeHierarchy($r.rC$, $conf_r$, $cust_r$)
7: $||$updateUsageInterface(rm_r, $cust_r$)
8: $||rm_c$ = composeModels(rm_c,rm_r,$cust_r$)
9: $|$**return** rm_c

when building a specific application such as the CMS. Now, a complete realization can be generated by re-composing the *Association* concern according to the complete configuration. The same is done for all other concern reuses. Eventually, the final application is built by composing the application models, the final *RM*, final *Association*, and any other *finalized* realization models of reused concerns as explained in detail in Section 3.1 (Composing a Concern Hierarchy).

The composition algorithms that come into play when composing concern hierarchies with support for delayed decision making are highlighted on the right hand side of Figure 4, and are explained in detail in the following section.

3. CORE Algorithms for Delaying Decisions

3.1 Composing a Concern Hierarchy

This subsection details the overarching composition algorithm that flattens an entire concern hierarchy according to a given configuration to generate a final realization model where all concerns have been combined.

Figure 5 presents a simplified view of the CORE metamodel. It clarifies the essential structure and properties of a CORE concern related to composition, which in turn helps to understand the description of the recursive algorithm that flattens a concern hierarchy using a bottom-up traversal of the reuse dependencies. For simplicity reasons, the pseudocode in Algorithm 1 describes how to perform the composition for a single type of realization model. In reality, the composition algorithm is executed once for each type of realization model used in the concern hierarchy.

We illustrate the execution of the algorithm with the situation in Figure 4 when the CMS wants to generate a final realization model for the *RM* concern by calling COMPOSEHIERARCHY(RM,<Alloc, Search, ThreadSafe, Database>, {|Resource→Car,...}). As a first step (line 2), a user-tailored realization model rm_c corresponding to the selected features of concern c, denoted by $conf|_c$, is created by calling COMPOSECONCERN(RM,<Alloc,Search>) (see Algorithm 2 in Section 3.2). The resulting model still contains reuses of other concerns, e.g., *Association*. For each of the reuses, the part of $conf$ that relates to features of the reused concern ($conf|_{Association}$ = <ThreadSafe,Database>) is composed with the configuration of the reuse (<Association, Kind, KeyIndexed>) in line 4 (see Algorithm 3 in Section 3.3). Similarly, the customizations passed as a parameter and the customizations of the reuse are composed (line 5)

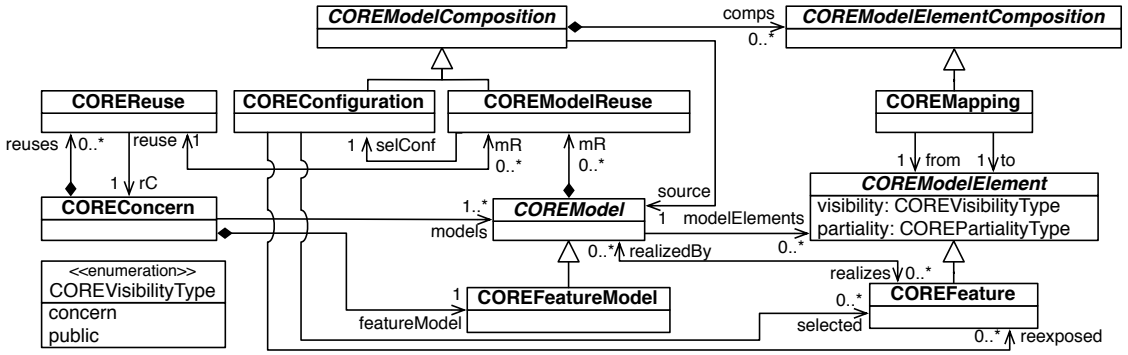

Figure 5. Composition in the CORE Metamodel

Algorithm 2 Composing a Concern

```
1:  function composeModelReuses(r1,r2)
2:  | r.selConf ← composeConfigurations(r1.selConf, r2.selConf)
3:  | r.comps ← composeCustomizations(r1.comps, r2.comps)
4:  | return r
5:  function composeModelReuseSets(ℝ1,ℝ2)
6:  | for all r2 ∈ ℝ2 do
7:  | | if ∃r1 ∈ ℝ1 | r1.reuse = r2.reuse then
8:  | | | r1 ← composeModelReuses(r1,r2)
9:  | | else
10: | | | ℝ1 ← ℝ1 ∪ r2
11: | return ℝ1
12: function composeConcern(c,conf)
13: | ℝℳ: Set{COREModel} ← findRealizations(conf|c)
14: | rmc: COREModel ← ∅
15: | for all rm ∈ ℝℳ do
16: | | rmc ← composeModels(rmc,rm,∅)
17: | | if rm.mR ≠ ∅ then
18: | | | rmc.mR ← composeModelReuseSets (rmc.mR, rm.mR)
19: | return rmc
```

(see Algorithm 5 in Section 3.5), and then COMPOSEHIERARCHY is invoked recursively (line 6). The exact call in our case would be COMPOSEHIERARCHY(*Association*, <Association, Kind, KeyIndexed, ThreadSafe, Database>, {|Data→Task, |Key→Capability, |Associated→Resource}). Then, the usage interface of the resulting realization model $rm_{Association}$ is updated according to information hiding principles in line 7 (see Algorithm 6 in Section 3.6). Finally, the updated $rm_{Association}$ is composed with the user-tailored realization model of *RM* rm_c by calling COMPOSEMODELS, i.e., the composition algorithm provided by the realization language (line 8).

3.2 Composing a Concern

The COMPOSECONCERN algorithm shown in Algorithm 2 generates a user-tailored realization model of a given concern according to a given configuration. The algorithm first determines the realization models that need to be composed. In CORE, feature realizations that have no interactions with other feature realizations are associated with a single realization model (see `realizedBy` link in Figure 5). To handle resolvable feature interactions, additional realization models can be defined by the concern designer that realize multiple features, i.e., the ones for which they resolve the interaction. In line 13, the FINDREALIZATIONS function returns the set of realization models $\mathbb{RM}$ that realize the selected features in *conf*, prioritizing feature interaction models, if any.

Next, an empty realization model rm_c is created (line 14). Successively, all chosen feature realization models *rm* in $\mathbb{RM}$ are composed with rm_c using the composition algorithm provided by the realization language (line 16). The model reuses of rm_c are then merged with the model reuses of *rm*, if any (COMPOSEMODELREUSESETS in line 18).

3.3 Composing Configurations and Feature Models

When defining a CORE configuration, we took the following considerations into account:

C1 Choosing the best variant of a reused concern is only possible once all desired system qualities are known. These may not be known at intermediate levels in a concern hierarchy.

C2 A concern encapsulates all possible variants that can be useful in any context. When reused in a specific context, some variants may not be applicable.

Configuration Rules: To support C1, we allow a concern user to *specify partial configurations, i.e., to select only the minimally required features that the reusing concern needs from the variable features of the reused concern.* To support C2, we allow a concern user to *reexpose in the variation interface of the reusing concern the features of the reused concern that are also alternative realizations or potentially useful optional features in the context of the reusing concern*. Any features of a reused concern that are neither selected nor reexposed are deemed inapplicable realizations in the context of the reusing concern.

Configuration Verification: The following rules ensure that the *union of the selected and reexposed features* constitutes a valid feature selection in the classical sense:

- General Rule: All ancestor features (from the parent of a feature to the root) of a selected or reexposed feature must either be mandatory, selected, or reexposed.

Algorithm 3 Composing a Configuration

```
1: function COMPOSECONFIGURATIONS(c1,c2)
2: | conf.selected ← c1.selected ∪ c2.selected
3: | conf.reexposed ← c1.reexposed ∩ c2.reexposed
4: | return conf
```

Algorithm 4 Feature Model Composition Algorithm

```
1: procedure COMPOSEFEATUREMODELS(f_reusing,FM,conf)
2: | for all f_i ∈ FM | f_i ∉ (conf.selected ∪ conf.reexposed) do
3: | | remove f_i from FM
4: | for all f_i ∈ FM | f_i in XOR group do
5: | | if f_i ∈ conf.selected then
6: | | | set fi to mandatory
7: | for all f_i ∈ FM | f_i in OR group do
8: | | if f_i ∈ conf.selected then
9: | | | set fi to mandatory
10: | | else if f_i ∈ conf.reexposed then
11: | | | if ∃f | f ∈ same OR group ∧ f ∈ conf.selected then
12: | | | | set f_i to optional
13: | add root of FM as mandatory subfeature of f_reusing
```

- XOR Group Rule: If the parent of an XOR group is mandatory, selected, or reexposed, then either exactly one feature of the XOR group must be selected, or at least 2 features of the XOR group reexposed.
- OR Group Rule: If the parent of an OR group is mandatory, selected, or reexposed, then either at least one of the features of the OR group must be selected (other features may be additionally reexposed), or at least two reexposed.
- *Requires* and *Excludes* Rules: A feature that is required by a selected feature has to also be selected. A feature that is required by a reexposed feature has to be either selected or reexposed. A feature that is excluded by a selected feature is neither allowed to be selected nor reexposed. A feature that is excluded by a reexposed feature is not allowed to be selected.

Configuration Composition Algorithm: Algorithm 3 shows the simple algorithm that composes two configurations by calculating the union of the selected features (line 2) and the intersection (line 3) of the reexposed features.

Feature Model Composition Algorithm: Algorithm 4 describes how to attach to the feature $f_{reusing}$ the feature model of a reused concern configured with *conf*, eliminating features that are neither selected nor reexposed and adjusting the XOR and OR groups of the feature model, if necessary. Figure 2(b) shows the result of running the feature model composition algorithm for *RM* and *Association*, where the feature *Search* selects *KeyIndexed* and reexposes *ThreadSafe*, *HashMap* and *Database*.

3.4 Composing Impact Models

Impact models make it possible to compare configurations of a concern with respect to a set of qualities. For instance, Figure 6 shows the impact model of the *Association* concern dealing with the *Performance* quality. Comparison against

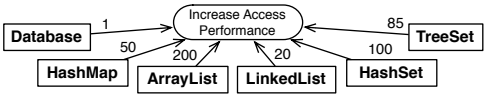

Figure 6. *Association* Impact Model

Figure 7. *Resource Management* Impacts

other features is *relative*, e.g., the *HashSet* feature with contribution 100 increases performance twice as much as the *HashMap* feature with contribution 50.

Impact Model Composition Specification: The quality properties of a concern depend on the design of the models realizing its features, but also on the qualities of the realization of reused concerns, if any. Therefore, the impact models of the reusing concern need to be composed with those of the reused concern. How much the qualities of the reused concern affect those of the reusing concern, relatively speaking, depends on the details of how the concern is reused. Each quality is addressed individually.

When defining the impact model composition specification, we took in addition to C1 and C2 mentioned in the previous subsection the following considerations into account:

C3 The qualities of the reusing concern are affected by the qualities of the reused concerns.

C4 It must be possible to compare the results of impact models (*comparability*). If the satisfaction results for a quality in the impact models of two reused concerns are the same value *n*, then the relative degree of satisfaction for each of the two reused concerns must be the same.

C5 It must be possible to determine for a particular quality of a reused concern whether a given set of selected features results in the optimal or worst solution with respect to this quality (*determinability*).

C4 and C5 are satisfied by following the relative comparison scheme for contributions and by normalizing the satisfaction results of each node of each impact model to the [0,100] range [11]. In other words, the unifying feature across all impact models is the fact that the best possible solution results in a satisfaction value of 100 and the worst in 0. If this is the case, then the designer again only needs to think about the relative comparison of features of the reusing concern and the respective quality goals of the reused concerns as shown in Figure 7. In this figure, it is determined that the realization of *Search* of *RM* contributes twice as much to increase performance as *increase access performance* from the reused *Association* (10 vs. 5)[1]. In other words, *Search* contributes two thirds, and the reused concern potentially

[1] Of course, when the impact model is evaluated during impact analysis, a goal from a reused concern only contributes to the goals of the reusing concern if the feature that reuses the concern is actually selected.

Algorithm 5 Composing Customizations

```
1: function COMPOSECUSTOMIZATIONS(ℂ1,ℂ2)
2: | return ℂ1 ∪ ℂ2
```

one third. If the reused concern contributes the best possible performance result (i.e., its satisfaction value is 100 because the best option is selected, i.e., *ArrayList*), it will contribute the full third to the performance goal of the reusing concern (i.e., 33 after normalization to the [0,100] range). If the satisfaction value of the reused concern is less than 100 (e.g., because *Database* is selected instead), then the contribution is proportionally less. Hence, continuity from the reused to the reusing impact model is ensured.

The composition of a reused impact model with the impact model of the reusing concern is hence the set of contribution links including their weights that have to be added from the reused impact model to the reusing one. In the example in Figure 7, this set contains only one link for the reuse of the *Association* concern (i.e., the link between *Increase Access Performance [Association]* and *Increase Performance*).

Impact Model Composition: The composition algorithm for impact models is quite straightforward. The contribution links specified in the impact model composition specification are created between the reused and the reusing impact model. In turn, this causes the satisfaction results to be recalculated to ensure normalization [11].

3.5 Customization Interface Composition

When defining customization interface composition, the following consideration is important:

C6 In order to obtain an executable application, all customization elements , i.e., any partially defined structural and behavioural elements, must be finalized.

To satisfy C6, the customization interface of the reusing concern is a union of the new customization elements introduced by the reusing concern and the customization elements of the reused concerns that have not been customized, i.e., that were not mapped to specific elements in the reusing concern. This makes it possible to grow or shrink the customization interface within concern hierarchies, depending on the intent of the developer. A "more specific" concern, for instance, would abstain from introducing new customization elements, and map some of the lower-level customization elements to specific elements. The same rule also ensures that customization elements that are part of reexposed features of the reused concern are incorporated into the customization interface of the reusing concern.

Therefore, the composition algorithm for composing customizations, presented in Algorithm 5, simply creates a union of the sets of mappings that are to be composed.

3.6 Usage Interface Composition

C7 In alignment with information hiding principles [21], internal details of reused concerns that are irrelevant for the user should be encapsulated and hidden to reduce complexity and minimize unnecessary dependencies.

Algorithm 6 Updating the Usage Interface

```
1: procedure UPDATEUSAGEINTERFACE(rm, ℂ)
2: | for all e ∈ rm.modelElements do
3: | | if (e.visibility = COREVisibilityType::public ∧
       ∄m | (m ∈ ℂ ∧ m.from = e)) then
4: | | | e.visibility = COREVisibilityType::concern
```

In order to satisfy C7, information hiding principles are applied by default when composing a concern hierarchy. In other words, the accessible structure and behaviour of the reused concern is *not* automatically included in the usage interface of the reusing concern, unless explicitly requested by the developer. To do so only makes sense when the reused concern provides functionality that the reusing concern wants to offer, potentially under a different (more specific) name. For example, the `getValues` functionality of the *Association* concern can be used as is to implement the functionality `getAssignedResources` of the class `Task`. Such a reexposition and renaming can be specified by the developer using a COREMapping that maps the model element from the reused concern to a model element in the reusing concern of the same type with a different name.

The UPDATEUSAGEINTERFACE algorithm shown in Algorithm 6 implements this idea: to apply information hiding principles, it finds all model elements in the usage interface of a realization model (i.e., the model elements that have a *public* visibility), and switches their visibility to *concern*, unless the developer has provided a mapping for it.

4. bCMS Validation

To validate the proposed approach, we implemented the algorithms for delaying decisions in our TouchCORE tool [24] and applied them to the design of the *bCMS* case study [7], a product line of crisis management systems. In the models realizing the features required in [7] we made use of Aspect-Oriented Use Case Maps (AoUCM) [20] to describe the interactions of the system with its environment, and we used Reusable Aspect Models (RAM) [16] – class diagrams and sequence diagrams – to describe the internal design of the *bCMS* backend. Within the realization, we reused a substantial number of pre-existing concerns that had been defined in previous CORE projects. Some of these concerns themselves reuse other concerns, which creates a *variable concern hierarchy* with complex concern dependencies[2].

Table 1 shows reuse metrics collected during the case study. The case study involves 102 reuses of 12 existing concerns. The reused concerns contain a total number of 61 features that are realized by 59 workflow and design models. The maximum depth of the concern hierarchy was 4.

Reuse of previously developed concerns suggests the effectiveness of the CORE paradigm and the proposed com-

[2] The resulting *bCMS* models including models of all reused concerns can be downloaded from http://www.ece.mcgill.ca/~gmussb1/bCMS/.

Concern	Features	Realizations (AoUCMIRAM)	Possible Confs without Reexposing	Possible Confs with Reexposing	Reuses	Unique Configurations	Reused Features (Unique)	Reexposed Features (Unique)
Association	12	0 | 23	224	–	38	8	76 (4)	125 (11)
Named	2	0 | 2	2	–	32	1	32 (1)	32 (1)
Singleton	2	0 | 2	2	–	11	1	9 (1)	–
Copyable	3	0 | 2	2	–	3	2	6 (3)	–
Networking	5	0 | 1*	4	–	7	1	7 (1)	28 (4)
Encryption	5	0 | 0*	5	–	1	1	1 (1)	4 (4)
KeyCounter	1	0 | 1	1	–	2	1	2 (1)	–
Command	2	0 | 2	2	5	3	2	4 (2)	–
Authentication	10	8 | 6*	20	80	1	1	2 (2)	8 (8)
RM	4	5 | 4	3	232	1	1	3 (3)	1 (1)
Authorization	3	3 | 2*	2	160	1	2	1(1)	1 (1)
Workflow	12	0 | 11	64	1040	2	2	15 (12)	–
Total	60	16 | 43	–	–	102	23	158 (32)	199 (27)
bCMS	19	8 | 15	384	57,016,320	–	–	–	–

Table 1. *bCMS* Reuse Metrics (* ⇔ under construction)

position algorithms. All reusable concerns apart from *RM* had been developed when building other applications. Some concerns were reused multiple times, possibly with multiple configurations resulting in a total of 158 feature reuses. The 158 feature reuses involve 32 unique reused features (i.e., features that are reused at least once). The most reused concern is the *Association* concern with 38 reuses. The *Named* concern allows to set and modify names and is reused 32 times. The largest concern in terms of features and realization models is *Workflow* (12 features and 11 realization models), which provides an implementation of a workflow engine for reactive systems.

Support for delayed decision making was very helpful for the *bCMS* development. First, the concerns that reused other concerns did not have to prematurely commit on a specific provided solution, and we able to reexpose alternative solutions in their variation interface. This was the case for *Command*, *Authentication*, *RM*, *Authorization*, and *Workflow*, which as a result increased their variability significantly (see the two columns "Possible Confs without / with Reexposing" in Table 1).

Second, it was possible to make correct/optimal realization choices even across several levels of reuse dependencies. For example, the *bCMS* was able to ensure correct concurrent behaviour for *RM* by selecting *ThreadSafe* as shown in Figure 4. Similarly, it was possible to choose optimal configurations for *Networking*, *Association*, *Copyable*, and *Named*, that are reused in the *Workflow* concern, when *Workflow* in turn was reused in the context of the *bCMS*. The *Reexposed Features* column in Table 1 provides further quantitative evidence for this fact. For example, it shows that in the 38 reuses of the *Association* concern a total of 125 features were reexposed (and all of the 11 features that *Association* offers were reexposed at least once), allowing for delayed/optimized configuration at a later point.

Third, it was possible to realize the features that support different communication protocols as requested in the *bCMS* requirements document simply by reusing the *Networking* concern and reexposing the provided protocols at the *bCMS* level. Finally, although not explicitly requested in the *bCMS* requirements document, reexposing some features of reused concerns at the *bCMS* level automatically created further features for the *bCMS* product line. For example, it made sense to delay the decisions about which authentication means to reuse (*Authentication* concern), and which encryption technology to reuse (*Encryption* concern). As a result, when configuring a specific *bCMS*, the developer can now choose, e.g., which authentication methods / encryption scheme to use by selecting any of the reexposed features from *Authentication* / *Encryption*. By delaying all unnecessary decisions, the resulting *bCMS* can in the end be configured in 57,016,320 ways instead of 384.

While the TouchCORE implementation validates our algorithms and the *bCMS* models provide evidence that delaying of decisions can be useful and facilitate reuse, we have not yet performed usability studies to evaluate whether or not delaying of decisions is practical in real-world software development.

5. Related Work

Variability and Modularity. Our approach is combining variability management techniques [8] and hierarchical modularity [5]. Contrary to classic SPL engineering principles where variability and composability have to be managed as a whole in the architecture of the SPL [6], our concerns – which are forms of variable components – handles variability in predefined domains like in product populations with software components [28]. Similar but less expressive approaches have been used when seeking more flexibility in a classic SPL setting [22], or when handling reuse in open-source communities [13]. Our contribution differs from these propositions by being dedicated to concerns and by handling definition and composition of open variable parts in the reusable units. Our work can also been seed as an extension of van der Storm's proposal for variable components [27] as his component dependencies need to be fully configured to be reused, thus not supporting delayed decision making as we do.

In [15], Kästner et al. propose a core calculus and C-based implementation for variability-aware modules with variability handling capabilities inside modules and on module interfaces. Our approach differs as it supports hierarchical modularity [5] with concern hierarchies and as it considers the impact of expressed variability on system qualities while guiding configurations.

Variability Composition. Composition algorithms for feature models have already been proposed. Acher et al. introduce FAMILIAR [1], a domain-specific language that allows the developer to manage several feature models, compose them with different operators, and reason about their validity. Our approach is different because the composition of feature models of different concerns is driven by concern reuse. Depending on the specific reuse, some features are

kept, some reexposed, and some removed. In addition, our composition of feature models may take into account trade-off analysis of stakeholders of the concerns when applying more advanced goal modelling concepts to impact models.

Software Product Lines. There has been work to use aspect-orientation in the context of SPLs [12; 29]. Voelter et al [29], e.g., use aspect-oriented (AO) modelling techniques to compose realizations of optional features with the realization of the root feature using AO programming techniques. While this approach resembles the way we compose a concern's realization models, it applies AO techniques *within a product line*, and is therefore not applicable to developing partial products with open variability, and allowing re-exposing of features. Furthermore, none of the AO/SPL approaches consider the specification of feature impacts.

Within concern hierarchies, the process of delaying decisions on configurations can be seen as similar to the notion of staged configuration in SPL [9], which breaks a global and closed variability model into multilevel or specialized configurable models. Our approach spreads variability in the concern hierarchy but it remains open as it is attached to a concern or a composition of concern. We focus on variable reusable units while staged configuration aims more at organizing configuration times, roles, and contexts. We see the two approaches as being complementary.

Research in Multi Product Lines (MPL) use composition models and model interfaces to compose different interdependent SPLs together. Schröter et. al. [25] provide interfaces for variability, modelling the syntactic, behavioural, and non-functional levels of the development process. However, they do not provide means to develop artifacts that are partially defined, which is supported by our approach and identified by our customization interface.

From a different perspective, Bak et al. [3] propose Clafer, a metamodeling language that supports expressing feature models and metamodels, but does not explore dependencies between feature models, which is important to express concern dependencies in our approach.

Goal Modelling and SPL. Our approach uses goal models to analyze the impact of selecting features. Research in goal-oriented SPL investigates the mapping between features and goals in different ways. Benavides et al. [4], e.g., extend feature models to include non-functional qualities by expressing relations among feature attributes. They then map the extended feature models onto a Constraint Satisfaction Problem (CSP) that allows some automated reasoning. However, their approach does not allow for stakeholder analysis, expressing dependencies between non-functional features of different feature models, or expressing relative satisfaction values, which are possible in our impact models.

Siegmund et al. [26] present SPL Conqueror, another approach with support for reasoning about non-functional properties in SPLs. Their variant-wise properties require generating variants, which is costly. SPL Conqueror is useful to present the variability of fully developed products. A concern modeller can reason about variability while developing partially-developed products. In addition, concern dependencies are not possible in the SPL Conqueror approach.

Finally, the MPL approaches described earlier [25] provide textual descriptions of non-functional impacts. Our impacts are described using goal-models, which are more formal and provide a concise way to express dependencies among non-functional properties which makes tool-based evaluation possible.

6. Conclusion and Future Work

This paper discusses how *variable concern hierarchies* support delaying of decisions in the context of Concern-Oriented Reuse (CORE). When reusing a concern, a developer decides on the reusable functionality that is minimally needed to continue development, and reexposes relevant alternatives of the reused concern in the reusing concern's interface, hence delaying decision making to when more detailed requirements are known and further decisions can be made. Variable concern hierarchies require sophisticated software composition to incrementally generate more specific versions of the reusable concern whenever additional decisions are made. The variation interface of a concern needs to be composed in the concern hierarchy, along with the models describing a concern and their customization and usage interfaces. The algorithms required for these compositions are detailed in this paper, and evaluated with the help of a crisis management case study, which shows the occurrence of delayed decision making across concern hierarchies and the feasibility of the proposed composition algorithms in support of delayed decision making. In contrast to other approaches, CORE advocates the use of concerns that are reuse-centric, variable, open for adaptation, and quality impact-aware, while employing advanced separation of concern techniques for model composition.

In future work, we will address the coordinated evaluation of combinations of default configurations for improved trade-off analysis, as well as the addition of quantitative, absolute real-world measurements in impact models. We also plan to evaluate further the practicality of composition operators and decision delaying together with CORE with the aim of defining a rigorous software development approach. As a complementary research line, we will explore how our form of variable component can be used in combination with the definition of the architecture of software product lines, mixing global closed variability elements with open variable concerns.

References

[1] M. Acher, P. Collet, P. Lahire, and R. B. France. Familiar: A domain-specific language for large scale management of feature models. *Science of Computer Programming*, 78(6):657–681, 2013.

[2] O. Alam, J. Kienzle, and G. Mussbacher. Concern-oriented software design. In *MODELS 2013*, pages 604–621. Springer, 2013.

[3] K. Bak, K. Czarnecki, and A. Wasowski. Feature and meta-models in clafer: Mixed, specialized, and coupled. In *SLE'10*, pages 102–122. Springer, 2011.

[4] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. In *CAiSE'05*, pages 491–503. Springer, 2005.

[5] M. Blume and A. W. Appel. Hierarchical modularity. *ACM TOPLAS*, 21(4):813–847, July 1999.

[6] J. Bosch. *Design and use of software architectures: adopting and evolving a product-line approach*. Pearson Education, 2000.

[7] A. Capozucca, B. H. Cheng, G. Georg, N. Guelfi, P. Istoan, and G. Mussbacher. Requirements Definition Document for a Software Product Line of Car Crash Management Systems, 2011.

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

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

[10] E. W. Dijkstra. *A discipline of programming*, volume 1. Prentice-Hall Englewood Cliffs, 1976.

[11] M. B. Duran, G. Mussbacher, N. Thimmegowda, and J. Kienzle. On the reuse of goal models. In *SDL 2015*, volume 9369 of *LNCS*, pages 141–158. Springer, 2015.

[12] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor Filho, and F. Dantas. Evolving software product lines with aspects: An empirical study on design stability. In *ICSE'08*, pages 261–270. ACM, 2008.

[13] J. V. Gurp and C. Prehofer. From SPLs to open, compositional platforms. In *Combining the Advantages of Product Lines and Open Source*. Dagstuhl, 2008.

[14] 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, CMU, 1990.

[15] C. Kästner, K. Ostermann, and S. Erdweg. A variability-aware module system. In *OOPSLA '12*, pages 773–792. ACM, 2012.

[16] J. Kienzle, W. Al Abed, and J. Klein. Aspect-Oriented Multi-View Modeling. In *AOSD'09*, pages 87 – 98. ACM Press, 2009.

[17] J. Kienzle, N. Guelfi, and S. Mustafiz. Crisis Management Systems: A Case Study for Aspect-Oriented Modeling. *Transactions on AOSD*, 7:1 – 22, 2010.

[18] C. W. Krueger. Software reuse. *ACM Comput. Surv.*, 24:131–183, June 1992.

[19] M. D. McIlroy. Mass-produced software components. *Proc. NATO Conf. on Software Engineering, Garmisch, Germany*, 1968.

[20] G. Mussbacher, D. Amyot, and J. Whittle. Composing goal and scenario models with the aspect-oriented user requirements notation based on syntax and semantics. In *Aspect-Oriented Requirements Engineering*, pages 77–99. Springer, 2013.

[21] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.

[22] G. Perrouin, J. Klein, N. Guelfi, and J.-M. Jezequel. Reconciling automation and flexibility in product derivation. In *SPLC'08*, pages 339–348, 2008.

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

[24] M. Schöttle, N. Thimmegowda, O. Alam, J. Kienzle, and G. Mussbacher. Feature modelling and traceability for concern-driven software development with TouchCORE. In *MODULARITY Companion*, pages 11–14. ACM, 2015.

[25] R. Schröter, N. Siegmund, and T. Thüm. Towards modular analysis of multi product lines. In *SPLC'13 Workshops*, pages 96–99. ACM, 2013.

[26] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, and G. Saake. Spl conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Control*, 20(3-4):487–517, Sept. 2012.

[27] T. van der Storm. Variability and component composition. In *Software Reuse: Methods, Techniques and Tools*, pages 157–166. Springer, 2004.

[28] R. van Ommering. Building product populations with software components. In *ICSE'02*, pages 255–265. ACM, 2002.

[29] M. Voelter and I. Groher. Product line implementation using aspect-oriented and model-driven software development. In *SPLC'07*, pages 233–242, 2007.