On the Modularization Provided by Concern-Oriented Reuse

Matthias Schöttle Omar Alam
Jörg Kienzle Gunter Mussbacher
McGill University, Montreal, QC H3A 0E9, Canada
{Matthias.Schoettle | Omar.Alam}@mail.mcgill.ca, {Joerg.Kienzle | Gunter.Mussbacher}@mcgill.ca

Abstract

Reuse is essential in modern software engineering, and hence also in the context of model-driven engineering (MDE). 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 concern, models are further decomposed and modularized by views and features. High-level concerns can reuse lower-level concerns, and models within a concern can extend other models belonging to the same concern, resulting in complex inter- and intra-concern dependencies. To clearly specify what dependencies are allowed between models belonging to the same or to different concerns, CORE advocates a three-part interface to describe each concern (variation, customization, and usage interfaces). This paper presents the CORE metamodel that formalizes the CORE concepts and enables the integration of different modelling languages within the CORE framework.

Categories and Subject Descriptors D.2.10 [*Software Engineering*]: Design; I.6.5 [*Simulation and Modeling*]: Model Development

Keywords model reuse, concern-orientation, model interfaces, model dependencies, variability, composition

1. Introduction

MDE [4] is a unified conceptual framework in which software development is seen as a process of *model production*, *refinement,* and *integration*. To reduce the accidental complexity and the effort needed to move from a problem domain to a software-based solution, MDE advocates the use of different modelling formalisms, i.e., modelling languages, to represent and analyze the system from *multiple points of view*. For each level of abstraction, the modeller uses the best formalism that concisely expresses the properties of the system that are important to that level, and in such a way that the concepts used in the language are close to the problem domain at hand. During development, high-level specification models are refined or combined with other models to include more solution details, such as the chosen architecture, data structures, algorithms, and finally even platform and execution environment-specific properties. The manipulation of models is achieved by means of model transformations. Model refinement and integration continues until a model (or code) is produced that can be executed.

However, MDE on its own is not a silver bullet against the complexity of modern software development. To reduce the complexity of reasoning, analyzing the problem, and constructing a software-based solution, traditional software engineering principles such as decomposition, interfaces, information hiding, levels of abstraction, and reuse are key. Unfortunately, the crosscutting nature of most software development concerns is an obstacle for many classical modularization techniques that apply the aforementioned principles in practice. To be effective, a *flexible notion of modularity* is required, that allows to separate and package concerns in a reusable way, and allows advanced composition mechanisms to introspect modules and compose them to build a usable (i.e., analyzable, simulatable, and/or executable) representation of the solution.

Concern-Oriented Reuse (CORE) [3] is a new software development paradigm inspired by the ideas of multi-dimensional separation of concerns [18]. CORE builds on the disciplines of MDE, software product lines (SPL) [14], goal modelling [6], and advanced modularization techniques offered by aspect-orientation [8, 15] to define flexible software modules that enable broad-scale model-based software reuse. This paper discusses the CORE concepts by presenting the CORE metamodel and reflects on how concerns and concern hierarchies support modular software development.

2. Concerns

A concern (*COREConcern* in Figure 1) is the main unit of modularization, abstraction, construction, and reasoning in the CORE paradigm. It groups models that pertain to any domain of interest to a software engineer. To increase the reuse potential, a concern typically encapsulates not only one specific way of addressing the domain of interest during software development, but *many* relevant *variations*. Furthermore, the models within a concern span multiple phases of software development and levels of abstraction. Each concern has a root phase, where the concern manifests itself for the first time. Some concerns appear in early phases of software development, e.g., broadly scoped system properties with functional, non-functional, or even intentional characteristics. Some concerns appear in later phases of software development, e.g., solution-specific concerns, e.g., specific communication protocols, concrete authentication algorithms, or design patterns.

Models detailing different views of the concern are built for the root phase and all follow-up phases using the most appropriate modelling formalisms to express the properties of the concern that are relevant during each phase. The models can use many modelling notations, which need to offer advanced composition mechanisms in order to handle the crosscutting nature of certain concerns.

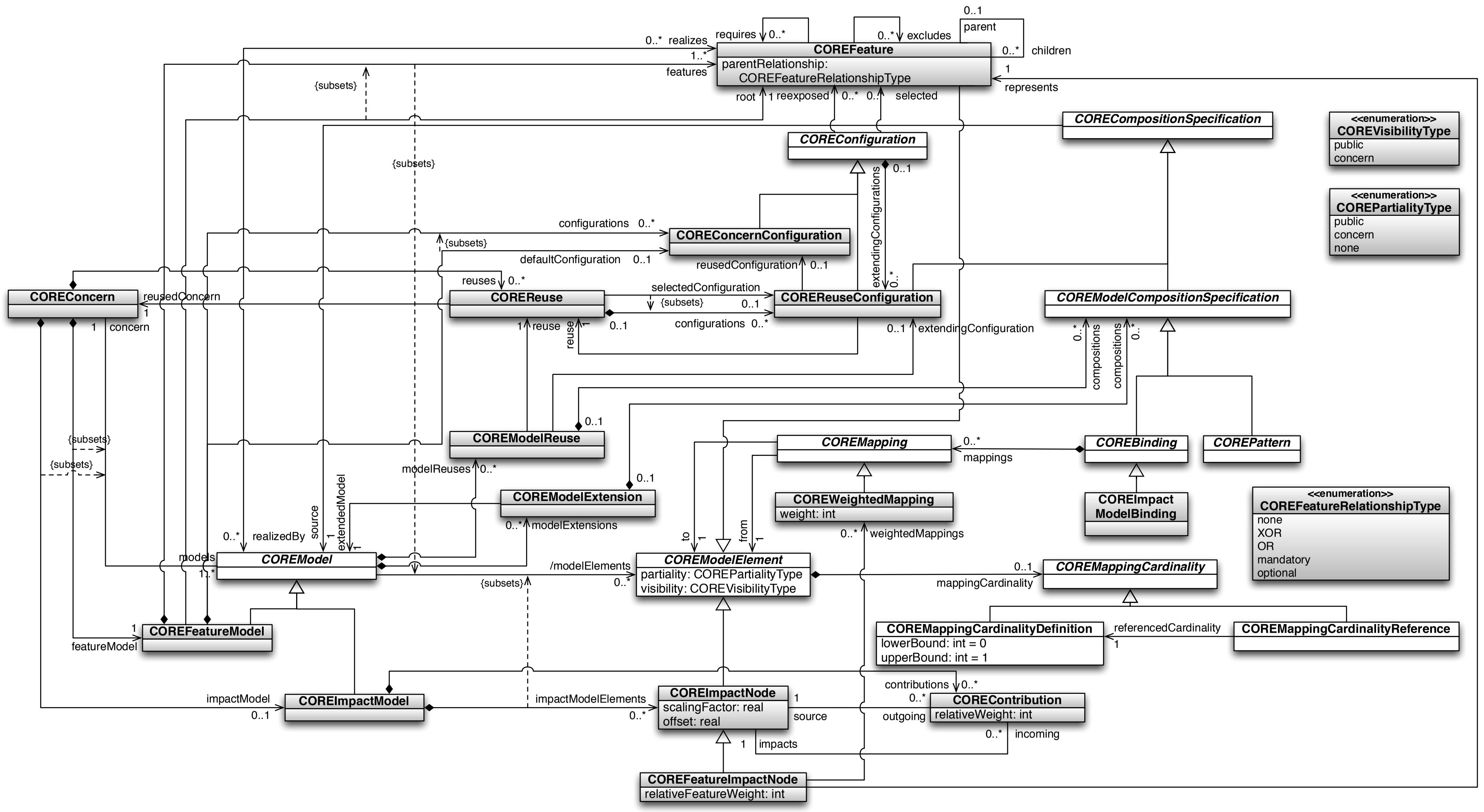

Figure 1. The CORE Metamodel

3. Intra-Concern Modularity

3.1 View-Oriented Decomposition

Some concerns, in particular the ones that address broader development concerns, encapsulate highly complex structure and behaviour. One way of decomposing the functionality of the concern is to follow the MDE philosophy, and describe the properties of interest using multiple models/views (*COREConcern* contains multiple *COREModel* in Figure 1) at multiple levels of abstraction, expressed using different modelling notations. In this case, "standard" MDE practice dictates that *model transformations* and/or *consistency rules* link the models that are expressed in different formalisms to ensure the coherence of the concern. For example, in the current reference implementation of CORE [1, 17, 19], the structure of software design concerns are specified using *class diagrams*, and the behaviour is modelled with *sequence diagrams* and *protocol state machines*. OCL constraints and references between the views ensure that the behaviour in the sequence diagram can only access structural properties (e.g., attributes, associations) that are defined in the class diagrams [16], and only invoke operations on instances according to the protocols specified in the protocol state machines [2]. Ensuring consistency between views is a well-known challenge in MDE-based approaches . The effort it takes to keep views consistent is view and modelling language dependent, and hence not further covered by CORE.

3.2 Feature-Oriented Decomposition

Additional complexity stems from the fact that a concern encapsulates *multiple variants* of addressing a domain of interest, similar to what is done in SPL. To tame the complexity due to variability, CORE provides modularization support within a concern using *features*.

To this aim, each concern has an associated feature model (*COREFeatureModel*), that is composed of features (class *COREFeature* in Figure 1). Currently, CORE supports classic feature trees as defined by Kang et al. [7].

The idea of modularization along features is that instead of creating large, monolithic models/views of a concern for each variation, the model is decomposed into many small models called *realization models*. The realization models are linked with *directed extension dependencies* (*COREModelExtension*). Typically, a realization model containing only model elements pertaining to a single feature extends other realization models whose model elements it builds upon and which are potentially extended. I.e., realization models can contain model elements that are shared by several features. Which realization model belongs to which feature is encoded in the association *realizedBy/realizes* between *COREFeature* and *COREModel*. The extension dependencies between realization models form a hierarchy that is consistent with the parent/child dependencies of the corresponding features.

Realization models in the upper levels of the extension hierarchy often also define *partial model elements*. Their role is to define generic properties shared by a set of realization models, but require to be completed with additional feature-dependent properties in order to function correctly. The concern designer can explicitly tag such model elements using the *partiality* attribute of *COREModelElement* and set it to *concern*. For instance, a design realization model for an *Authentication* concern might define a concern-partial *Credential* class and a placeholder for an operation *check* to compare credentials, so that the *AuthenticationManager* can use it to validate an attempted authentication.

For a given realization model, the models related directly and indirectly through extension dependencies form an extension tree. It can be used to generate a complete model/view for any variant by composing all realization models in the tree with each other. By default, this composition is done with a simple name/signature-based *merge*, as it is general practice in MDE (see, for example, UML package merge [11]). If this is not sufficient, each extension dependency can provide composition directives (*COREModelCompositionSpecification*), that use pattern expressions or bindings that map (using *COREMapping*) elements from the current model to model elements in the extended model.

For instance, name/signature-based merge is mostly sufficient to compose design class diagram realization models, except when partial model elements are defined in the extension tree. Then, composition specifications are used to specify which model elements from the lower model complete the partial model elements defined in the upper model. For example, the design realization model for the *Password* feature of the *Authentication* concern would complete the partial *Credential* class introduced above with a class called *Password* that contains an encrypted *String* and defines a comparison operation *checkPassword* to complete the concern-partial operation *check*. This would be specified using a *COREMapping* from *Password* to *Credential*, and from *checkPassword* to *check*.

On the other hand, composing behavioural designs expressed using sequence diagrams requires even more sophisticated composition mechanisms that use patterns to express where the behaviour defined in the current model should be positioned with respect to the behaviour specified in the models that it extends [9].

3.2.1 Expressing Feature Interactions and Modularizing Feature Conflict Resolutions

The existence of feature interactions is a well-known fact in the SPL world, and they come in three flavours. In some cases, the functionality of a feature is incompatible with the functionality offered by one or several other features. The concern designer can express such an unresolvable conflict using the reflexive *excludes* relationship of *COREFeature*. In other cases, the functionality of a feature can only be provided using the functionality of one or several other features. This dependency is expressed using the reflexive *requires* relationship of *COREFeature*.

The most elaborate form of feature interactions are resolvable conflicts. They occur when the simultaneous presence of two or more features requires their individual structural and/or behavioural realizations to be adapted. CORE allows the concern designer to modularize this conflict resolution within separate realization models, and link them to the features whose conflict they resolve with the association *realizes* between *COREModel* and *COREFeature*.

4. Inter-Concern Modularity

While the module boundaries between realization models within the same concern are thin, the boundaries between concerns are very rigid. This is intentional in order to simplify reuse for a *concern user*. The user should be shielded from the complexity encapsulated within the reused concern by the *concern designer*—an expert of the domain that the concern addresses—as much as possible.

In CORE, the boundaries between concerns are formalized with three interfaces: the *variation interface*, the *customization interface*, and the *usage interface*. They contain all the information that a *concern user* needs to know in order to reuse a concern created by a *concern designer*.

4.1 Variation Interface

The *Variation Interface* allows the concern designer to expose to the concern user the available functional variants and design choices that the concern offers, together with the impact of the different alternatives on high-level stakeholder goals, qualities, and non-functional requirements. For example, a security concern may offer various means of authentication, from *password-based* to *biometrics-based* solutions, each with differing impacts on the *level of security* as well as *cost* and *end-user convenience*.

The different variants/choices are encoded in the *feature model* introduced above. The structure of the feature tree and the nature of the parent/child relationship (*parentRelationship* attribute of *COREFeature* of type *COREFeatureRelationshipType*, which can take the values *mandatory*, *optional*, *or*, or *xor*), as well as the *requires* and *excludes* relations between features constrain the possible choices that are available to the concern user.

Additionally, the impact of selecting a feature on non-functional goals and qualities is specified with an *impact model* [5]. CORE currently only supports impact models that are expressed using a variant of the Goal-oriented Requirement Language (GRL) [6]. We choose goal models for impact analysis because goal models allow vague, hard-to-measure system qualities to be evaluated, such as *user convenience* or *security*, in addition to more quantifiable qualities, e.g., *cost*. Goal modelling is typically applied in early requirements engineering activities to capture stakeholder and business objectives, alternative ways of satisfying these objectives, and the positive/negative impacts of these alternatives on various high-level goals and quality aspects. The analysis of goal models guides the decision-making process, which seeks to find the best suited alternative for a particular situation. These principles also apply in our context, where an impact model is a type of goal model that describes the advantages and disadvantages of features offered by a concern, and gives an indication of the impact of a selection of features (*COREConfiguration*) on goals that are important to the concern user. The impact model in CORE (*COREImpactModel*) consists of an acyclic goal graph, modelled as impact nodes representing goals (*COREImpactNode*) connected by weighted contributions links (*COREContribution*).

4.1.1 Operationalizing the Variation Interface

While the variation interface concisely exposes the available variations and their impacts to the concern user, the concern designer has to connect the features in the feature model with the impact nodes of the impact model in order to operationalize the interface.

This link is established using feature impact nodes. For each feature in a concern whose associated realization model have impacts on non-functional goals and qualities, the concern designer must create a *COREFeatureImpactNode* and link it to the impacted *COREImpactNode*s using *COREContribution* links with the appropriate weight.

4.2 Customization Interface

Each realization model within a concern is described as generally as possible to increase reusability. Therefore, some elements in the models are only *partially* specified and need to be related or complemented with concrete modelling elements of the *application* that intends to reuse the concern. This is in contrast to partial elements that are defined in a high-level realization model that have to be completed by other realization models *within* the concern as introduced in subsection 3.2. Partial elements defined in the customization interface cannot be completed at design-time by the concern designer, since they are placeholders for application-specific structure and behaviour. As such, they are completed at reuse-time by the application designer. For example, a *Security* concern may define a generic *User* as a partial class that needs to be merged with the concrete application classes that describe the actual users of the system, e.g., *Administrator* or *Employee*.

The *Customization Interface* in CORE lists all model elements in the realization models that *have to be adapted* to the context of a specific application in order to be useable. Again, the *partiality* attribute of type *COREPartialityType* in *COREModelElement* is used for that purpose: every model element whose partiality attribute has the value *public* is part of the customization interface.

As a result, there is no single customization interface for a concern. On the contrary, for each possible configuration of a concern, i.e., for each set of selected features, the corresponding customization interface is defined by the *union of all public partial model elements of all realization models* that realize the features of the configuration and those that they extend. Given a configuration, it can be generated by following the *realizedBy* and *extendedModel* links.

4.3 Usage Interface

The *Usage Interface* is the classic kind of interface. It specifies which model elements are accessible/visible to the outside world, i.e., to the models that the concern user creates. In other words, it defines how the concern user can access the functionality, i.e., the structure and behaviour, provided by the concern. For example, the usage interface of the design model of a concern is typically comprised of all *public* classes and methods made available by the concern. For a *Security* concern this might include an *authentication* operation that an administrator can invoke in order to gain access to restricted behaviour, or a *Password* class that can be instantiated.

To designate those usable model elements, the CORE metamodel defines a *visibility* attribute of type *COREVisibilityType* for *COREModelElement*. By setting it to *public*, the concern designer can expose any model element of a realization model to the outside world.

Similarly to the customization interface, there is no single usage interface for a concern. This is not surprising, since different features may offer different functionality. Therefore again, for each possible configuration of a concern, the corresponding usage interface is defined by the *union of all publicly visible model elements of all realization models* that realize the features of the configuration and those that they extend. Given a configuration, it can be generated following the *realizedBy* and *extendedModel* links.

5. Concern Hierarchies

Complex applications consist of many intertwined, interacting concerns, and CORE advocates developing an application by reusing as many already existing concerns as possible. The same principle applies to the development of a concern itself: in order to realize its functionality, a high-level concern (called from now on a *reusing concern*) is able to reuse the functionality of a lower-level concern when appropriate. To this aim, a reusing concern can create a *COREReuse* that refers to the concern that is to be reused (*reused concern*), hence creating a *concern hierarchy*.

Typically, concerns that encapsulate domains that are at a higher level of abstraction reuse concerns at a lower level of abstraction. For example, an *Authentication* concern might reuse an *Observer* design concern in its realization to notify the authentication servers when a user updates his credentials, or within a *Database* concern to store the user credentials. Similarly, a more domain-specific or solution-specific concern can reuse other more general concerns. For example, a *UniversityAccounting* concern might reuse a more general *Accounting* concern to implement some of its functionality.

5.1 Reuse Process

While building a concern is a non-trivial, time consuming task, typically done by or in consultation with a domain expert, reusing an existing concern is extremely simple, and essentially involves three steps:

1. The concern user must first select the feature(s) with the best impact on relevant stakeholder goals and system qualities from

the variation interface of the concern based on tool-provided impact analysis. Based on this configuration, the modelling tool then merges the models that realize the selected features to yield new models of the concern corresponding to the desired configuration. Depending on the root phase of the concern, the merging may involve requirement models and/or design models. For composition at the requirements level with goal and workflow/scenario models, the interested reader is referred to [10] for more details. For details on how this composition is done for design concern models with structural and behavioural descriptions based on class, sequence, and state diagrams, see [3].

2. Next, the concern user has to adapt the generated detailed models to the application context by mapping customization interface elements to application-specific model elements. Again, depending on the root phase of the concern, this step might require customizing requirement models and/or design models.
3. Finally, a software engineer can use the functionality provided by the selected concern features which are exposed in the usage interface within his own application models. In requirements models, this may mean including workflow segments exposed in the concern's usage interface in the application's workflow models. In design models expressed using sequence diagrams, for instance, using a concern may involve instantiating a class exposed in the concern's usage interface and/or calling one of its public operations.

5.2 Variation and Impact Transparency

For each reuse, the concern designer of the reusing concern decides what features of the reused concern are needed. To be precise, all realization models of the reusing concern that want to use the functionality provided by the reused concern state which features of the reused concern they want to rely on. They do that by creating a *COREModelReuse* referring to the *COREReuse* with a *COREReuseConfiguration* that designates the desired features (association *selected*) of the reused concern. To increase future reuse potential, the concern user (who is the concern designer of the reusing concern) should select only the *minimal set of features required for realization*. Decisions between multiple alternative features that provide the desired functionality, or about whether or not to include optional features that provide additional functionality, should not be made. Any such features that do not conflict with the ones that were selected should be marked as reexposed (association *reexposed*).

Using this information, the actual set of deferred decisions, i.e., the set of reexposed features from the reuse, is defined as the intersection of all reexposed features of all *COREModelReuses* made by realization models. These deferred decisions are *automatically propagated* to the variation interface of the reusing concern, and are as a result now available variants of the reusing concern.

Likewise, the non-functional qualities of the reusing concern are heavily influenced by the qualities of the reused concern. Therefore, all impacts of reused concerns are propagated to the variation interface of the reusing concern. Since the concern designer of the realization models of the reusing concern knows how the functionality of the reused concern is being used, she can create a *COREModelReuse* for the impact model of the reusing concern with a *COREImpactModelBinding* that specifies *COREWeightedMappings* to connect the impacts of the reused concern to the impacts of the reusing concern.

5.3 Customization Translucency

Since the structure and behaviour of most lower-level concerns crosscuts the upper levels (and the final application), any model element of a reused concern *can* be customized. However, as explained in subsection 4.2, the partially defined model elements of a reused concern that are exposed in the customization interface *must be customized* in order to yield a runnable realization. Therefore, any uncustomized model elements from the customization interface of a reused concern are propagated to the customization interface of the reusing concern.

5.4 Usage Opacity

In order to successfully reduce complexity, the concerns should allow for *separate reasoning*. They should *hide the complexity* of the lower levels from the upper levels, following the information hiding principles advocated by Parnas [12, 13]. For this reason, the usage interface of a reused concern is not visible in the interface of a reusing concern, i.e., the value of the *visibility* attribute of all model elements in the reused concern that are *public* are changed to *concern* by the TouchCORE tool after the model composition is complete.

6. Conclusion

This paper discussed the concepts of CORE that relate to modularity by means of the CORE metamodel. Concern hierarchies allow the developer to organize software development into different spheres of abstraction, where each sphere is a concern. Internally, the concern can reuse other concerns, just like a big sphere can encapsulate smaller spheres. All the information that a concern user needs to reuse a concern is available through the three interfaces, i.e., at the shell of the sphere. However, the image of a sphere breaks down when it comes to considering the properties of the interfaces. While a shell is typically opaque and hard, the modularization provided by concerns is flexible. The three interfaces are opaque from a usage point of view in order to support information hiding. They are translucent from a configuration point of view to ensure complete adaptation to potentially unknown reuse context and enable crosscutting composition with other concerns. Finally, they are transparent from a variation and impact point of view in order to expose impacts of reused concerns at the reusing concern's interface, and to propagate any undecided realization alternatives of the reused concern to the reusing concern's feature model interface.

Intra-concern modularity is provided by means of view- and feature-oriented decomposition. Realization models within a concern have complete visibility on each other if they need to, but must declare such extension dependencies explicitly.

The CORE metamodel presented here has already been used in practice to integrate the modularization capabilities offered by CORE into two modelling notations, namely the *User Requirements Notation* and *Reusable Aspect Models*.

References

[1] TouchCORE Tool. `http://touchcore.cs.mcgill.ca/`.

[2] AL ABED, W., SCHÖTTLE, M., AYED, A., AND KIENZLE, J. *Behavior Modeling – Foundations and Applications: International Workshops, BM-FA 2009-2014, Revised Selected Papers.* Springer International Publishing, Cham, 2015, ch. Concern-Oriented Behaviour Modelling with Sequence Diagrams and Protocol Models, pp. 250–278.

[3] ALAM, O., KIENZLE, J., AND MUSSBACHER, G. Concern-oriented software design. In *International Conference on Model-Driven Engineering Languages and Systems - MODELS 2013* (2013), vol. 8107 of *LNCS*, Springer, pp. 604–621.

[4] DOUGLAS C. SCHMIDT. Model-Driven Engineering. *IEEE Computer 39* (2006), 41–47.

[5] DURAN, M., MUSSBACHER, G., THIMMEGOWDA, N., AND KIENZLE, J. On the reuse of goal models. In *SDL 2015* (2015), Springer, pp. 1–18.

[6] ITU. User Requirements Notation (URN), 2012.

[7] KANG, K., COHEN, S., HESS, J., NOVAK, W., AND PETERSON, S. Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, Software Engineering Institute, CMU, 1990.

[8] KIENZLE, J., Ed. *Transactions on Aspect-Oriented Development (TAOSD VII), Special Issue on a Common Case Study for Aspect-Oriented Modeling*, vol. 6210 of *LNCS*. Springer, 2010.

[9] KIENZLE, J., AL ABED, W., AND KLEIN, J. Aspect-Oriented Multi-View Modeling. In *Aspect-Oriented Software Development – AOSD 2009* (March 2009), ACM Press, pp. 87 – 98.

[10] MUSSBACHER, G., AMYOT, D., AND WHITTLE, J. Composing goal and scenario models with the aspect-oriented user requirements notation based on syntax and semantics. In *Aspect-Oriented Requirements Engineering*. Springer Berlin Heidelberg, 2013, pp. 77–99.

[11] OBJECT MANAGEMENT GROUP. *Unified Modeling Language: Superstructure (v2.4.1)*, December 2011.

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

[13] PARNAS, D. L. A technique for software module specification with examples. *Communications of the Association of Computing Machinery 15*, 5 (May 1972), 330–336.

[14] POHL, K., BÖCKLE, G., AND VAN DER LINDEN, F. J. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[15] R. FILMAN, T. ELRAD, S. CLARKE, M. AKŞIT. *Aspect-Oriented Software Development*. Addison-Wesley, 2004.

[16] SCHÖTTLE, M., AND KIENZLE, J. On the Challenges of Composing Multi-View Models. *In the GEMOC'13 Workshop co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013)* (October 2013).

[17] SCHÖTTLE, M., THIMMEGOWDA, N., ALAM, O., KIENZLE, J., AND MUSSBACHER, G. Feature modelling and traceability for concern-driven software development with TouchCORE. In *Companion Proceedings of MODULARITY 2015* (March 2015), pp. 11–14.

[18] TARR, P., OSSHER, H., HARRISON, W., AND SUTTON, JR., S. M. N degrees of separation: Multi-dimensional separation of concerns. In *ICSE'1999* (May 1999), IEEE CS, pp. 107 – 119.

[19] THIMMEGOWDA, N., ALAM, O., SCHÖTTLE, M., ABED, W. A., DI'MECO, T., MARTELLOTTO, L., MUSSBACHER, G., AND KIENZLE, J. Concern-Driven Software Development with jUCMNav and TouchRAM. In *Proceedings of the Demonstrations Track of the ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014), Valencia, Spain, October 1st and 2nd* (2014), CEUR-WS.org.