Towards Concern-Oriented Design of Component-Based Systems

Jörg Kienzle
McGill University
Montreal, Canada
Email: joerg.kienzle@mcgill.ca

Anne Koziolek, Axel Busch, Ralf Reussner
Karlsruhe Institute of Technology
Karlsruhe, Germany
Email: {koziolek, busch, reussner}@kit.edu

***Abstract*—Component-based software engineering (CBSE) is based on defining, implementing and composing loosely coupled, independent components, thus increasing modularity, analyzability, separation of concerns and reuse. However, complete separation of concerns is difficult to achieve in CBSE when concerns crosscut several components. Furthermore, in some cases, reuse of components is limited because component developers make certain implementation choices that are incompatible with the non-functional requirements of the application that is being built. In this paper we outline how to integrate CBSE and concern-oriented reuse (CORE), a novel reuse paradigm that extends Model-Driven Engineering (MDE) with best practices from aspect-oriented software composition and Software Product Lines (SPL). Concretely, we outline how to combine the Palladio Component Model (PCM) capable of expressing complex software architectures with CORE class and sequence diagrams for low-level design. As a result, multiple solutions for addressing concerns that might even crosscut component boundaries can be modularized in a reusable way, and integrated with applications that reuse them using aspect-oriented techniques. Additionally, thanks to CORE, component developers can avoid premature decision making when reusing existing libraries during implementation.**

I. Introduction

With the ever increasing complexity of software, traditional approaches of building software from scratch become more and more inefficient in terms of productivity and cost. Component-based software engineering (CBSE) [1] is a reuse-promoting way of developing software that is based on defining, implementing and deploying loosely coupled, independent components. Each component forms a modular and cohesive unit and provides clearly defined functionality, encapsulated behind *provided* interfaces. In case the implementation of the component depends on functionality provided by other components, these dependencies are documented with *required* interfaces. Ideally, when building an application with CBSE, only application-specific components have to be implemented from scratch, and general functionality is provided by reusing existing components from the *component repository*. A running application is obtained by selecting the appropriate components based on their interfaces, assembling them following a well-defined software architecture, and deploying them onto computational, communication and storage resources.

CBSE has multiple advantages. For example, it promotes modularity [2] and separation of concerns [3], as components can be designed by different developers, potentially using different programming languages, and implementation details hidden behind well-defined interfaces. Since components can be developed independently, developers can work on components according to their specific business or technical knowledge, isolating them from the complexity and unrelated details of the remaining part of the application, and allowing them to focus on the development tasks that they are experts in.

Another advantage of CBSE is that thanks to the *provided* and *required* interfaces, replaceability is promoted. A required component can be replaced by other compatible ones that implement the same functionality in different ways. This in turn increases maintainability, since component implementations that are no longer suitable can be substituted easily.

Furthermore, the explicit software architecture present in CBSE forms an additional layer of structuring that makes it possible to reason about software at a higher level of abstraction. In particular, model-driven engineering (MDE) approaches can be applied at the software architecture level to analyze, verify and predict desired properties of applications under development. For example, automated MDE-based approaches such as [4] can exploit the degrees of freedom that component substitution and flexible allocation strategies offer to perform multi-criteria design exploration.

However, although CBSE constitutes an important step forward, reuse in the context of CBSE is in practice not as straightforward. For example, substitutability and reuse of components, in particular COTS components, is sometimes difficult to achieve [1]. Even if the provided functionality and required dependencies match, the non-functional requirements, e.g., performance or memory/power consumption, can prevent reuse in a particular application context. This is most often due to the fact that bottom-up development approaches such as CBSE force developers to make most implementation choices at development-time when the requirements of the reuse context are unknown.

Moreover, separation of concerns is difficult to achieve in CBSE when the concern does not align with the structure imposed by the software architecture [5]. This is predominantly the case for concerns that are inherently distributed, as by their very nature their functionality crosscuts multiple computational resources. Since classic CBSE is designed in such a way that the unit of reuse is the component, concerns that crosscut component boundaries can not easily be packaged. This hinders reuse, and additionally prevents the rigorous application of information hiding principles.

Last but not least, CBSE does not provide support for explicitly expressing the fact that there are often many solutions and possible software architectures to address specific functional or non-functional development issues. As a result, a software architect that wants to put together an application by reusing components from the component repository might directly select a specific solution, i.e., set of components and architecture, and oversee potential alternatives.

In this paper we outline how to integrate CBSE and concern-oriented reuse (CORE), a novel reuse paradigm that extends Model-Driven Engineering (MDE) with best practices from separation of concerns, goal modelling and Software Product Lines (SPL). With its explicit variation, customization and usage interfaces, advanced software composition based on aspect-orientation, and support for delayed decision making, CORE has the potential to enhance CBSE to overcome the shortcomings mentioned above.

The remainder of the paper is structured as follows. Section II illustrates the problem with a motivating example. Section III presents the most important concepts of CORE, compares CORE with CBSE, and outlines the main integration challenges. Section IV explains concretely how we envision to integrate the Palladio Component Model (PCM) [6], a modelling formalism capable of expressing complex software architectures with performance, cost and reliability impacts, into the existing CORE framework reference implementation that supports low-level design with class and sequence diagrams. The last section presents a perspective on the potential benefits that a successful CORE/CBSE integration might provide in a long run.

II. Motivating Example

Let us assume a software architect develops a web store. The web store is comprised of four main components, namely a `store`, a `basket`, `checkout` and `shipping` component. These components operate together when a customer processes an order: `Store` interacts with `basket`, while `basket` forwards its content to `checkout` (for payment). Finally, checkout triggers `shipment`.

Payment is a recurring issue in many commercial applications, and hence it is not surprising that multiple third parties offer payment processing systems that can be integrated into component-based systems. Such systems provide essential payment-related functionality, e.g., *payment processing*, which covers standard payment interactions and connection to multiple payment solutions, i.e., credit cards, PayPal, *payment verification*, which keeps track of payment status and provides reliable confirmation that payment was successful, and *billing*, which creates and distributes customer bills.

Including this functionality into the web shop application affects the software on multiple levels. The software architecture changes, since the payment system is constituted of additional COTS components that need to be considered during deployment. Furthermore, these new components need to be connected to the business components according to the new control flow that includes payment. For instance, after checkout, but before shipping, payment verification needs to occur. Similarly, a bill should only be sent out after successful shipping.

The internal design of some components also needs to be updated. For example, somewhere within the functionality of checkout, the payment processing needs to be invoked. This involves changing the internal behaviour of *checkout*, but also adding a *required* interface to *checkout* that needs to be linked to the component that provides payment processing.

While all third-party solutions for payment offer the main functionalities outlined above, they do so in different ways, with a varying number of components and interfaces, different control flows and dependencies on other components, and with different quality (e.g., performance, reliability, cost).

Integrating CBSE and CORE would make it possible to express the variability of available solutions, and for each solution describe the architectural variations, if any, and design integrations, if any. For each solution, the concern would encapsulate the architectural models and detailed design models specifying the solution, as well as how to apply the solution within an application. The CORE reuse process would assist the software developer in choosing a solution, and ensure that the solution is correctly and consistently integrated with the application architecture and design. Why this is the case is explained in the following section, which presents CORE and how it relates to CBSE in more detail.

III. CORE and CBSE

A. Background on CORE

In concern-oriented reuse (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, 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 relevant properties of a domain of interest during software development in a versatile, generic way [7]. Concerns decompose software into reusable units according to some points of interest [3], [2] and may have varying scopes, e.g., encapsulating several authentication choices, communication protocols, or design patterns. Consequently, the models within a concern can span multiple phases of software development and levels of abstraction (from requirements and analysis models, to design models to code).

The main premise of CORE is that recurring development concerns are made available in a concern library, which covers 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 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 [8]. 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 [9].

B. Comparing CORE and CBSE

While the description of CORE presented above shows many similarities with component-based development, one of the fundamental differences is that the unit of reuse in CORE – the *CORE concern* – is broader than the unit of reuse in CBSE – the *component*. Similar to SPLs, a CORE concern encapsulates a variety of solutions for a specific domain of interest, and expresses this variability explicitly in a *variation interface* (VI). The VI comprises a feature model [10], which describes the available functional- and implementation alternatives that the concern encapsulates as well as their dependencies, if any. The feature model expresses the *closed variability*, i.e., the set of solutions that the designers of the concern have realized. Standard CBSE does not provide a means to group a set of functionally equivalent components together, but extensions have been proposed to express variability within components, e.g., [11], and modules, e.g., [12].

The CORE VI also describes the impacts that the different solutions offered by the concern have on high-level stakeholder goals, system qualities, and non-functional properties using a variant of the Goal-oriented Requirement Language (GRL) [13]. Again, standard CBSE does not address non-functional properties, but extensions have been defined to address specific quality properties [14]. In particular, the Palladio Component Model (PCM) [6] that is discussed in more detail in the next section supports detailed performance, reliability and cost analysis for software architectures.

In CORE, each solution within a concern must define a *customization interface* (CI) that describes how the solution can be adapted to a specific reuse context. Since each solution is described as generally as possible to increase reusability, some structural and behavioural elements are only partially specified and need to be related or complemented with concrete elements from the reuse context. This enables *open variability*, similar to what is achieved at the programming level with generic or template classes. Standard components support coarse-grained open variability, as they can specify *required* interfaces for functionality that must be provided by the reuse context. Application-specific functionality can then be integrated during assembly by connecting the *required* interface to an application component with a compatible provided interface. Internal customization of components, while provided by extended component models such as Fractal [15] and BlueArX [16], is not supported by standard CBSE. In this case, object-oriented customization techniques, e.g., generics, inheritance and call-backs, can be used for customization purpose.

Last but not least, one important advantage of CORE is that it does not require a concern user to commit to a specific solution variant at the moment of reuse. When reusing a concern, a developer only needs to decide on the reusable functionality that is *minimally needed* to continue development, and can re-expose relevant alternatives of the reused concern in the reusing concern's interface. This delays decision making to when more detailed requirements are known and further decisions can be made. To operationalize delayed decision making, CORE relies on additive, aspect-oriented software composition techniques that (re)compose concern realizations whenever additional decisions are made [17].

Table I summarizes the CBSE and CORE comparison.

	Standard CBSE	Palladio PCM	CORE
Interface for expressing variability	no	no	variation interface feature model
Support for expressing impacts	no	performance reliability cost	variation interface impact model
Interface for customization	goarse grained *required* interface	goarse grained *required* interface	customization interface
Interface for functionality	*provided* interface	*provided* interface	usage interface
Support for delaying decisions	no	no	yes

TABLE I
COMPARISON OF CBSE AND CORE

C. Integrating CORE and CBSE

The CORE paradigm is general, and in theory any modelling or programming language can be used to describe properties of interest for a concern. Concretely, though, a modelling language that wants to be usable as a realization language within CORE must, for each model type it provides:

1) clearly identify customization and usage interfaces, and
2) offer a homogeneous model composition operator.

Furthermore, if the modelling language is supposed to be used in combination with other modelling languages to describe the realization of a concern following a multi-view approach, then consistency rules have to be specified between the views, and compatibility of the composition operators must be ensured.

IV. PALLADIO INTEGRATION

Palladio [6] is a model-driven approach to CBSE comprised of the Palladio Component Model (PCM), a metamodel for describing component-based software architectures, and the Palladio Bench, an Eclipse-based modelling environment. What sets Palladio aside from other component-based approaches is an extensive set of analysis tools for performance, reliability, and costs evaluation, with includes PerOpteryx, a tool for multi-criteria design space exploration.

This section outlines the steps involved in integrating Palladio with the CORE framework. For each Palladio model type, we discuss what would constitute the customization and usage interface, and explain how we envision the design of a homogeneous composition operator.

A. Structural View: Component Repository and Assembly

Just like other component-based approaches, behaviours (i.e., operations) are modularized structurally in the PCM with *signatures*, which have a name, parameters and return type. Signatures are logically grouped within *interfaces*. The

unit of reuse in the PCM is the *component*. There are three increasingly detailed ways of specifying a component: the *Provides Type* defines interfaces comprised of the signatures a component offers to be used by other components; the *Complete Type* extends the *Provides Type* with the interfaces that a component requires to realize the provided services. The *Implementation Type* goes further and provides an abstract behavioural description on the implementation of the signatures of a component. Signatures, interfaces and components are specified in the system-independent *component repository* model.

The *assembly model* describes the inner structure, i.e., the conceptual architecture, of composite components and systems. In an assembly model, an *assembly contexts* stands for a component, from which it inherits the provided and required interfaces as defined in the component repository. *Assembly connectors* link a required interface of one assembly context to a provided interface of another assembly context.

Integration of Structural View with CORE

Structural Customization Interface: As explained in section III, CORE concerns are broader units of reuse than components. It is therefore natural that a concern can encompass *multiple components*, i.e., an entire, though maybe partial, software architecture. Some of the components encapsulated in the concern will be basic Palladio components, i.e., they have *provided* and *required* interfaces, and their complete implementation is also contained inside the concern. However, in order to be able to capture architectural elements that crosscut component boundaries, it should be possible to define *partial* components, interfaces and signatures that can later be composed with the architecture of the reuse context. These elements comprise the architectural customization interface.

For example, imagine a simple *Logging* concern. From an architecture point of view, it would be comprised of a standard *Logger* component that provides a *Log* interface that would allow any other component in the system to write log entries to a common storage. Having a *Logger*, though, is not enough. There must also be at least one component that make use of the *Logger*. Structurally, we don't know much about the components that use the *Logger*, apart from the fact that they should require the *Log* interface, and that they need to be connected to the logger. From an architectural point of view, the *Logging* concern should therefore also comprise a *Logged* partial component that must be composed with components in the system in order to augment their structure with the required *Log* interface and link them to the *Logger* component in the assembly. The customization interface for the *Logging* concern therefore consists of the *Logged* partial component.

Structural Usage Interface: The *provided* interfaces of Palladio components are the key to executing the component's functionality. It therefore makes sense to use them as the architectural CORE usage interface. However, as some *provided* interfaces of components within a concern should never be directly connected to components in the reuse context, only specifically selected *provided* interfaces should be part of the usage interface.

For example, assume that the *Logger* component in our example above uses a *Database* component to store the logs. The *provided* interface of the *Database* component should not be part of the usage interface for the *Logging* concern. Information hiding principles [2] dictate that internal design decisions, in particular those that might change in the future, should not be visible to the outside world to reduce complexity and increase maintainability by avoiding unnecessary dependencies. Hence, we propose that the architectural CORE usage interface of a concern should be constituted of a subset of the *provided* interfaces of the components encapsulated within the concern.

Structural Composition Operator: We envision the definition of the composition algorithm that combines two structural architecture models to be straightforward. It essentially constitutes a symmetric merge of two repositories and two assembly contexts. The specifics of the merge are highly similar to what is done in the current CORE reference implementation for merging class diagrams. Components need to be treated just like classes, and assembly connectors just like associations. For details, the interested reader is referred to [18] for CORE class diagram composition.

B. Behavioural View: SEFFs and Usage Scenarios

Palladio uses an abstract behavioural description, called service effect specification (SEFF), for describing the internal behaviour of a component at a high level of abstraction. SEFFs model the abstract control flow of the service provided by a signature of a component in terms of internal actions (i.e., resource demands accessing the underlying hardware) and external calls (i.e., accessing functionality by invoking signatures in *provided* interfaces of connected components). All control flow elements of activity diagrams, e.g., conditional execution, iterative execution, parallel execution and synchronization, are supported. In order to support quality prediction, additional information is attached to SEFFs. For example, for performance prediction, resource demands for internal actions (in terms of CPU instructions to be executed) can be specified. Cost and reliability prediction are supported as well.

In addition to the intra-component behaviour specified using SEFFs, processing rates of hardware nodes can be specified. Furthermore, domain experts can specify the interaction between certain user types and the system under development with a *usage model*, typically expressed using a variant of activity diagrams. Workload annotations can be supplied for each scenario definition that defines the number of occurrences of a scenario within a certain time period in terms of probability. The combined information from resource demands, processing rates and usage models makes performance prediction of architectures possible.

Since usage scenarios focus on describing user interaction and workload with entire systems, we currently believe that it does not make sense to include usage scenarios in concerns. Concerns are meant to be reused in many systems, each subject to their own usage scenarios. Therefore, the remainder of this subsection focuses solely on integrating SEFFs with CORE.

Integration of Behavioural View with CORE

In the current reference implementation of CORE, the behavioural design of a concern is modelled using a variant of sequence diagrams that can describe control flow at the same level of detail as code, if desired. One single sequence diagram specifies an interaction that can involve many operations provided by multiple objects.

When integrating CORE and Palladio, a component implementor would use CORE sequence diagrams to specify the functionality of the component with an object-oriented design. In this case, SEFFs are simply a more abstract representation of the behaviour of a component that adequately represents the low-level detailed design described using CORE sequence diagrams. In other words, the SEFF view and the sequence diagram view should be consistent. If they are not consistent, then the Palladio quality estimation tools would not be able to accurately predict performance, reliability and cost of a design.

Integration Strategy 1 (IS1):: One way of keeping SEFFs and sequence diagrams consistent is to define consistency rules or model checking algorithms that the CORE tool would use to verify consistency periodically. Whenever the consistency check fails, the modeller would be prompted to adjust the models. Additionally, model transformations should be defined to create a SEFF skeleton from an existing CORE sequence diagram, and vice versa. The former transformation would support *bottom-up* development, since it creates an abstract SEFF representation that is consistent with an already existing design. The later transformation would support *top-down* development, since it generates a skeleton design that corresponds to the high-level behavioural description envisioned by a software architect and specified in a SEFF.

Integration Strategy 2 (IS2):: Another way to ensure that SEFFs and sequence diagrams are consistent is to combine the information from SEFFs and sequence diagrams into one model. To this aim, we need to identify the information encoded in the SEFF metamodel that is not currently present in the CORE sequence diagram metamodel. The control flow structures in CORE sequence diagrams are at least as expressive as the ones in SEFFs, so they can replace the SEFF control flow structures. Similarly, whether an invocation of an operation of a class is a internal or external action can be determined by checking in the architecture model whether the operation is part of a class that belongs to the same component or not. On the other hand, performance, reliability and cost information, e.g., resource demands and failure rates, are only present in SEFFs. The sequence diagram metamodel would therefore need to be augmented so that the quality information from SEFFs can be attached to the right model elements of the sequence diagram.

SEFF Customization and Usage Interface: We believe that the customization and usage interface of SEFFs, in analogy with what is done for CORE sequence diagrams, should be inherited from the customization and usage interface of the structural view.

SEFF Composition Operator: If SEFFs and sequence diagrams were integrated by combining them into one model (IS2), then the current sequence diagram composition operator can be used as is to also compose the sequence diagrams with additional quality estimation information. This is true because the composition is solely based on the control flow.

If SEFFs are kept separately from the sequence diagrams (IS1), then a separate composition operator for SEFFs needs to be defined. Just like for any other behavioural modelling notation, the order and sequencing of model elements that represent internal and external actions in SEFFs is important. Hence, the composition operator for SEFFs would have to support the composition of behaviour from one SEFF *before*, *after*, *around* or *in parallel* of some behaviour in the other SEFF. Since CORE is based on additive composition, complete replacement of behaviour does not need to be supported. Instead, a limited form of substitution has to be provided, where the original behaviour is moved to some place within the substituting behaviour. The details of how this can be accomplished with CORE sequence diagrams are presented in [19], and for general sequence diagrams in [20].

When implementing the SEFF composition operator, care must be taken that the composition is compatible with the sequence diagram composition already present in the CORE reference implementation. Otherwise, when composing two concerns one could end up with an output that has inconsistent SEFFs and sequence diagrams, even in the case where the SEFFs and sequence diagrams of each input concern are initially consistent. Since the SEFF control flow structures are a subset of the control flow structures available in sequence diagrams, it should be fairly easy to create a compatible SEFF composition operator by adapting the algorithm of the existing sequence diagram composition operator.

SEFF Integration Strategy Discussion: From the considerations above it seems like it would be easier to choose IS1, i.e., combine sequence diagrams and SEFFs into one model. First, there would be no need to define a new composition operator, since the existing sequence diagram composition operator suffices. Second, concern realization models would be simpler, as there would not be two separate views describing the same behaviour at different levels of abstraction that must be kept consistent.

However, one of the advantages of Palladio is that it enables rapid design exploration. During development, SEFFs can be used to capture intended behaviour of architectural components at a high level of abstraction even before they exist. In doing so, it is possible to predict quality properties of the solution architecture under development before the detailed design of each individual component has been elaborated. If we want to continue to use Palladio for this purpose, then SEFFs have to be supported in CORE independently of CORE sequence diagrams.

C. Deployment View

The Palladio deployment view covers the specification of the execution environment and the allocation of software components on resources of this execution environment, e.g., processors, servers, hard disks, communication links. Resources are annotated with information used for quality prediction, e.g., processing and failure rates. At the current state of

research we believe that it is not necessary to integrate the Palladio deployment view into CORE. The reason for this is that currently, concerns only encapsulate *software* components.

Integrating the structural and behavioural views of Palladio into CORE as described above allows developers to define reusable software architectures and implementations that can be composed with an application architecture to yield a final system with an architecture that includes the components defined in the reused concern. Developers would then create a deployment view for this final system architecture, i.e., define resources and deploy the components onto them.

Adding deployment models to CORE concerns could make sense if a concern would also encapsulate reusable *hardware*. In that case, resource specifications for these hardware components should be provided in the concern, and different ways of allocating provided software components and functionality to these resources could be expressed with deployment models.

V. Conclusion

This paper presented a road map on how we are planning to integrate two highly complementary modelling approaches, Palladio and CORE. Palladio and the Palladio bench tools enable modellers to specify component-based software architectures and predict their quality by means of abstract behavioural specifications annotated with performance, reliability and cost information. CORE, in particular the current CORE reference implementation, provides facilities for encapsulating a variety of design solutions, and modelling the detailed designs of the solutions with class and sequence diagrams.

By integrating the Palladio architecture model with the CORE reuse mechanisms and low-level design models as described in this paper, reuse of existing architectural solutions is greatly simplified. First, different architectural solutions to the same development issue are easier to find, as they are grouped within a concern and their variability expressed with CORE feature models. Furthermore, the Palladio quality estimation tools for performance, reliability and cost, and the CORE impact models for any other relevant non-functional qualities enable informed decision making and even tradeoff analysis between the proposed solutions. Once a specific solution is chosen, the CORE customization interface clearly designates the architectural and design elements of the solution that need to be connected to application-specific elements. Once the developer specifies the mapping, the aspect-oriented composition operators of CORE take care of composing the chosen concern with the application. This includes adding interfaces to application components and linking them to components provided by the reused solution, and modifying the internal application design to invoke the services provided by the solution at the appropriate places. Finally, thanks to CORE's support for delayed decision making, the developers of reusable architectural concerns can internally reuse other architectural and design concerns without having to make premature or default decisions regarding non-functional implementation alternatives. Ultimately this will increase reusability of the solutions, as the commitment to a specific implementation can be made once the requirements of the reuse context are known.

While the approach proposed in this paper has multiple advantages, it is also rather heavyweight, mainly because MDE and/or aspect-oriented programming has to be used for the detailed design of all solutions encapsulated within a concern. If MDE technology is going to be used exclusively at the architecture level for design exploration and quality prediction purpose, a more lightweight approach consisting of introducing a new concern-like grouping unit into the Palladio component model could be envisioned as outlined in [21].

References

[1] C. Szyperski, D. Gruntz, and S. Murer, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. New York, NY: ACM Press and Addison-Wesley, 2002.

[2] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.

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

[4] A. Koziolek, H. Koziolek, and R. Reussner, "PerOpteryx: automated application of tactics in multi-objective software architecture optimization," in *QoSA-ISARCS 2011*. ACM, 2011, pp. 33–42.

[5] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr., "N degrees of separation: Multi-dimensional separation of concerns," in *ICSE 1999*. IEEE, 1999, pp. 107 – 119.

[6] R. Reussner *et al.*, *Modeling and Simulating Software Architectures - The Palladio Approach*. MIT Press, 2016, to appear.

[7] O. Alam, J. Kienzle, and G. Mussbacher, "Concern-oriented software design," in *MODELS 2013*. Springer, 2013, pp. 604–621.

[8] J. Kienzle, G. Mussbacher, O. Alam, M. Schöttle, N. Belloir, P. Collet, B. Combemale, J. DeAntoni, J. Klein, and B. Rumpe, "VCU: The Three Dimensions of Reuse," in *ICSR 2016*, ser. LNCS, no. 9679. Springer, June 2016, pp. 122–137.

[9] M. Blume and A. W. Appel, "Hierarchical modularity," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 4, pp. 813–847, Jul. 1999.

[10] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-90-TR-21, November 1990.

[11] T. van der Storm, "Variability and component composition," in *Software Reuse: Methods, Techn., and Tools*. Springer, 2004, pp. 157–166.

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

[13] M. B. Duran, G. Mussbacher, N. Thimmegowda, and J. Kienzle, "On the reuse of goal models," in *SDL 2015*, ser. LNCS. Springer, 2015, vol. 9369, pp. 141–158.

[14] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R. V. Chaudron, "A classification framework for software component models," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 593–615, 2011.

[15] J. Muskens and M. Chaudron, *Prediction of Run-Time Resource Consumption in Multi-task Component-Based SoftwareSystems*. Springer, 2004, pp. 162–177.

[16] J. E. Kim, O. Rogalla, S. Kramer, and A. Hamann, "Extracting, specifying and predicting software system properties in component based real-time embedded software development," in *ICSE-Companion 2009*, May 2009, pp. 28–38.

[17] J. Kienzle, G. Mussbacher, P. Collet, and O. Alam, "Delaying decisions in variable concern hierarchies," in *GPCE 2016*. ACM, 2016.

[18] J. Kienzle, W. Al Abed, and J. Klein, "Aspect-Oriented Multi-View Modeling," in *AOSD 2009*. ACM Press, March 2009, pp. 87 – 98.

[19] M. Schöttle and J. Kienzle, "On the challenges of composing multi-view models," in *GeMOC 2013*, ser. CEUR Workshop Proceedings, vol. 1102, October 2013, pp. 1 – 6.

[20] J. Klein, F. Fleurey, and J. M. Jézéquel, "Weaving multiple aspects in sequence diagrams," *Transactions on Aspect-Oriented Software Development*, vol. LNCS 4620, pp. 167–199, 2007.

[21] A. Busch, Y. Schneider, A. Koziolek, K. Rostami, and J. Kienzle, "Modelling the structure of reusable solutions for architecture-based quality evaluation," accepted at CloudSPD 2016.