VCU: The Three Dimensions of Reuse

Jörg Kienzle, Gunter Mussbacher, Omar Alam, Matthias Schöttle, Nicolas Belloir, Philippe Collet, Benoit Combemale, Julien Deantoni, Jacques Klein, Bernhard Rumpe

VCU: The Three Dimensions of Reuse

Jörg Kienzle[1], Gunter Mussbacher[1], Omar Alam[2], Matthias Schöttle[1], Nicolas Belloir[3], Philippe Collet[4], Benoit Combemale[5], Julien DeAntoni[6], Jacques Klein[7], and Bernhard Rumpe[8]

[1] SOCS/ECE, McGill University, Montréal, QC, Canada,
joerg.kienzle@mcgill.ca, gunter.mussbacher@mgill.ca,
matthias.schoettle@mail.mcgill.ca
[2] Trent University, Canada oalam@acm.org
[3] Université de Pau, France nicolas.belloir@univ-pau.fr
[4] Université Nice Sophia Antipolis, France
philippe.collet@unice.fr, julien.deantoni@polytech.unice.fr
[5] Université de Rennes 1, France benoit.combemale@irisa.fr
[6] Université du Luxembourg, Luxembourg jacques.klein@uni.lu
[7] RWTH Aachen, Germany rumpe@se-rwth.de

Abstract. Reuse, enabled by modularity and interfaces, is one of the most important concepts in software engineering. This is evidenced by an increasingly large number of reusable artifacts, ranging from small units such as classes to larger, more sophisticated units such as components, services, frameworks, software product lines, and concerns. This paper presents evidence that a canonical set of reuse interfaces has emerged over time: the variation, customization, and usage interfaces (VCU). A reusable artifact that provides all three interfaces reaches the highest potential of reuse, as it explicitly exposes how the artifact can be manipulated during the reuse process along these three dimensions. We demonstrate the wide applicability of the VCU interfaces along two axes: across abstraction layers of a system specification and across existing reuse techniques. The former is shown with the help of a comprehensive case study including reusable requirements, software, and hardware models for the authorization domain. The latter is shown with a discussion on how the VCU interfaces relate to existing reuse techniques.

Keywords: Reuse, Interfaces, Variability, Customization, Configuration, Extension, Usage, Concern-Oriented Reuse

1 Introduction

Complex systems are rarely built from scratch, but rather rely on the existence of reusable artifacts for improved productivity and higher quality. Reuse of artifacts comes in very different flavors, and can be investigated by looking at how the reusable artifact is manipulated during the reuse process, and by looking at various reuse techniques.

A long list of reuse techniques exist, each with its own unit of reuse [15]. Many of them are considered success stories, starting from isolated classes managed

in libraries to sophisticated components and services [6] and finally to large reusable entities such as frameworks and Software Product Lines [20]. Recently, concerns have been proposed as variable and generic units of reuse [3]. Successful reuse also includes development artifacts, such as analysis and design models describing interaction, function, data, or architecture. In recent years, it has been shown that even crosscutting elements can be reused with aspect-based merging and weaving techniques. Instead of concrete artifacts, it is also possible to reuse conceptual knowledge, such as design patterns [8], or to encode reuse knowledge in model transformations or code generators.

Dimensions of reuse may be considered by categorizing the manipulations performed on the reusable artifact during the reuse process. These manipulations range from the simple act of *using* an existing artifact to the more mature, coordinated *customization* (also called adaptation or extension) of reusable artifacts to a new reuse context, and thus also include white-box and black-box forms of reuse. As a prerequisite to all previously mentioned activities, a specific reusable artifact must first be *identified* (i.e., *selected* from a set of possibly applicable reusable artifacts). The following paragraph gives some examples of these common activities.

A simple example of repeated *use* of an artifact is the common case of a software application started several times. Often, however, a reusable artifact needs to be adapted to its reuse context. Source code may be reused through *copy/paste* and free adaptation (a common, but bad form of reuse as it is error-prone and difficult to maintain). On the other side of the reuse spectrum, there is the common reuse scenario, where, e.g., an operating system is installed on different computers with different, predefined features required for different forms of use and preferences. This is a case of reuse of an artifact through the *selection from a planned set of variations*, which also is the case for the popular Software Product Lines (SPL) paradigm. Modern applications more and more often have the ability to adapt themselves to their environment, i.e., to automatically *select* the most appropriate variation depending on their context. This requires the consequences of a *selection* on the system to be made explicit, so that it can be reasoned about. Last but not least, a piece of software can also be reused by embedding it in different applications (e.g., generic reusable class libraries, components, and frameworks). However, genericity is hard to achieve. While class libraries provide crisp interfaces describing an intended form of reuse, they may easily be too narrow to be usable. Frameworks often add customization ability through subclassing of their concepts to cover a wider range of supported reuse contexts.

Nowadays, it is generally agreed that reuse of artifacts with explicitly defined, clear boundaries through their interfaces is the most appropriate to reach high levels of reuse maturity. In this way, internal complexity and properties are encapsulated, and thus do not affect the (re-)users. While interfaces have traditionally been mostly employed to formalize the usage of an artifact, we stipulate that all forms of manipulating a reusable artifact should be supported by interfaces in today's complex development processes, starting with *identifi-*

cation, followed by *customization*, and finally the *usage* of a reusable artifact. Consequently we have identified the need for three interfaces that every reusable artifact should consider providing:

- a Variation (V) Interface,
- a Customization (C) Interface, and
- a Usage (U) Interface.

We call them interfaces, because people with different roles interact with the artifact during different activities of the development process through the appropriate interface to achieve a desired result. Each interface targets a different dimension of reuse, and together they streamline the reuse process. However, depending on the reusable artifact, these interfaces may be broader or smaller and explicit or implicit.

In the remainder of this paper, we first introduce the VCU interfaces in more detail in Section 2. The following two sections intend to provide convincing evidence that these three interfaces capture all dimensions required to achieve effective reuse. Section 3 presents several example models from the *Authorization* domain expressed in different modeling notations that were made reusable by adding VCU interfaces. In Section 4, we discuss how the explicit and implicit interfaces of existing units of reuse can be categorized with the VCU approach. Section 5 presents our conclusions and discusses future work.

2 The VCU Approach - Definitions

VCU stands for the three interfaces: *variation*, *customization*, and *usage*. We start with the last and most known.

2.1 Usage Interface

The *Usage Interface* (UI) describes what functionality can be requested by the developer of the application who wants to reuse the artifact, i.e., which structural and behavioral elements within the artifact are accessible. For example, the UI of a software design artifact is typically comprised of the *public* classes and methods made available by the artifact. For a reusable security artifact this might include an *authentication* operation that an administrator can invoke in order to gain access to restricted behavior. Sometimes usage interfaces are explicitly published, which includes the promise of developers that those are stable over evolution steps.

2.2 Customization Interface

The *Customization Interface* (CI) describes how the developer of an application tailors a generic artifact to a specific application. The term customization here has an extended meaning compared to the SPL paradigm. A reusable artifact is described as generically as possible to increase reusability. Therefore,

some elements in the artifact are only *partially* specified and need to be complemented with concrete modeling elements of the application that intends to reuse the artifact. Sometimes, parameters have to be filled in a template. Sometimes, complete new classes have to be provided and injected into the reused artifact, e.g., as hot spots in frameworks or as plug-ins in pluggable applications. The CI is hence used when a reusable artifact is *composed* with the application. For example, a security artifact may define generic *Users* and *Administrators* as partial classes that need to be merged with the concrete application classes that describe the actual users of the system, e.g., *Customer* or *CrisisCoordinator*, respectively. At the implementation/source code level, Java generics, for example, exist, which require the provision of a concrete type when they are reused. Databases are customized with schema definitions. The Eclipse IDE framework became so popular, because it is strongly decoupled and structured as a plug-in system, allowing the IDE to be customized.

2.3 Variation Interface

The *Variation Interface* (VI) exposes the available variants that the artifact encapsulates and from which the developer has to choose. It helps organize possible variations and their impact on goals and system qualities. From this VI the application developer selects one concrete variant of the artifact that fits the stakeholders needs best. Variations are typically described by a *feature model* [11] that specifies the individual features of the artifact, as well as their mandatory, optional, alternative, requires, and excludes relationships. The impact of choosing a feature can be specified with goal models [10] when relationships among goals are more complex or otherwise with attributed feature models [4]. For example, a reusable authentication artifact may offer various alternatives for authentication, from *key-based* to *biometrics-based solutions*, each with differing impacts on the *level of security* as well as *cost* and *end-user convenience*.

2.4 VCU Approach to Reuse

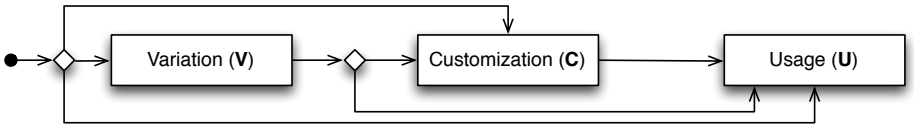

Fig. 1. The VCU Reuse Approach

Variant selection and customization typically happen during development time. Use of an artifact in terms of connecting it to the rest of the application also happens at development time, while actually using its functionality happens

at runtime when the application is executed. Some kinds of artifacts allow deferring the variant selection and customization at least partially to installation or runtime. Modern operating systems allow users to customize or at least adapt customization during installation, prior to or even while executing it. Plug-in systems allow extending an application and thus building new variants partially even at runtime. Recently, adaptive systems have started to automate the selection process, switching among variations at runtime. Still, the three interfaces should be methodically distinguished to simplify understanding of reusability techniques.

Existing techniques (see Section 4) may only use some of the three interfaces from the least mature to the most mature levels of reuse: only U, V&U, C&U, and all three interfaces as indicated in Fig. 1. Actually, a developer that wants to reuse an artifact is typically exposed to the VCU interfaces of the artifact in the opposite order (V, then C, then U), roughly following these *methodical guidelines*:

1. The developer determines the variant of the artifact that best suits her needs. This is done by *selecting* the feature(s) with the best impact on relevant stakeholder goals and system qualities from the VI of the artifact based on provided impact analysis.
2. The developer *customizes* the resulting artifact by filling all parameterizations (generics, partial elements), connecting the resulting artifact to the application under development with the help of the CI.
3. The developer actually *uses* the UI of the artifact in the rest of the application under development, such that the artifacts structural and behavioral properties are integrated at the desired locations.

In practice, however, this is an evolutionary process, e.g., changing the chosen variant due to adapted goals and therefore switching between variant selection, customization, and usage.

3 VCU Interfaces Across Levels of Abstraction

This section illustrates the applicability of the VCU approach by building a unit that encapsulates reusable requirements, design, and hardware models that describe general structural and behavioral properties of *Authorization*. The approach that was followed to create these models is called concern-orientated reuse (CORE), and described in more detail in [3].

Among authorization models, the most used ones are based on access control policy. The main idea is that access to a resource is controlled by some rules, such as, e.g., in the widely used Role-Based Access Control (RBAC) [7, 21]. In RBAC, the access of a user to a resource is based on the role of the user in the system to which RBAC rules are applied (e.g., in a banking institution, the role of a user can be customer or teller). Access to a resource is usually defined as a set of actions that the user can perform on the resource (e.g., a customer can withdraw or deposit money from or in an account).

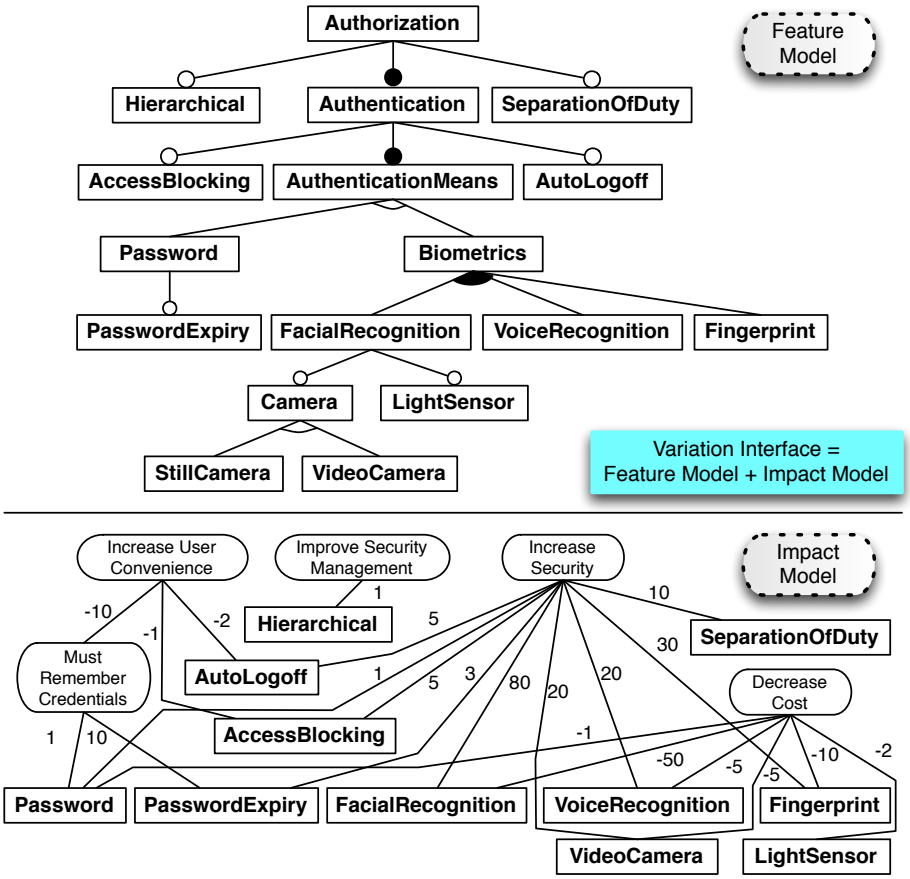

Fig. 2. Authorization Variation Interface

We first model the VI of *Authorization*, i.e., the different RBAC and Authentication features and their impacts using feature diagrams and impact models (Section 3.1). We then present the interaction workflow of *Authorization* using Aspect-oriented Use Case Maps (Section 3.2), the structural and behavioral design models using class and sequence diagrams with Reusable Aspect Models (RAM) (Section 3.3), as well as the hardware configurations using enhanced SysML block diagrams (Section 3.4). For space reasons, the descriptions focus mostly on the CI and UI of each model. Then, we show how to reuse the *Authorization* concern in a simple bank application (Section 3.5).

3.1 Variation Interface Models

Inspired by the RBAC specification in the NIST standard [7] and an RBAC feature model [14], we created a feature model [11] for the *Authorization* concern as shown at the top of Fig. 2.

The base functionality that any RBAC system must provide is encapsulated in the root feature *Authorization* and the mandatory *Authentication* child feature. The optional feature *Hierarchical* adds the ability for role inheritance, whereas *SeparationOfDuty* (SoD) adds the ability to restrict permissions based on constraints. Furthermore, the child features of *Authentication* provide different means for performing authentication (*Password* and *Biometrics* with its three sub-options), as well as the optional features *Access Blocking*, *Auto Logoff*, and *Password Expiry*. Hardware variability is also depicted by different *Camera* configurations and an optional *LightSensor* for *FacialRecognition*.

The impact model of the *Authorization* concern is shown at the bottom of Fig. 2. Four high-level goals are defined: *Increase Security*, *Decrease Cost*, *Increase User Convenience*, and *Improve Security Management*. The impact of variable features on these goals are indicated with weighted contributions in a relative way, e.g., the *Facial Recognition* feature impacts security sixteen times more than the *Auto Logoff* feature (80 vs. 5).

The Variation Interface (VI) for the *Authorization* concern is comprised of the feature and impact models. The feature model presents all encapsulated variants of the concern to the developer, and the impact model helps the developer to determine the best solution for a specific reuse context by enabling impact analysis on high-level system qualities. It is the VI that all other requirements, design, and hardware models presented in the remainder of this section have in common, i.e., the other models are realizations of the features defined in the feature model and the impact model relates the impact of these realization models to system qualities.

3.2 Requirements Models

The workflow model describes the two main user-system interactions of the *Authorization* concern in Fig. 3. First, the |*Administrator* may choose to define roles at any time (define start point), possibly using hierarchies (Hierarchical) and constraints (*SeparationOfDuty*). Second, the |*User* may have to authenticate herself (*authenticate* start point), but the authentication behavior must be combined with application-specific behavior of the system reusing *Authorization*. Therefore, a pointcut stub (dashed diamond shape with *P*) represents all those locations in the application that require authentication. Those locations are identified with a pattern, stating that authentication is needed when the |*User* interacts with a |*ProtectedResource* by attempting a |*protectedAction*.

The vertical bar | in the model highlights generic model elements that need to be customized to the actual application under development, i.e., these model elements constitute the Customization Interface (CI) of the workflow model. In Fig. 3 the CI elements are highlighted in orange. For example, |*ProtectedResource* may have to be matched against *Account* and |*protectedAction* against *withdraw* and *transfer*. Given these customizations, the *authenticate* behavior would be composed with the *withdraw* and *transfer* actions, resulting in an authentication check before performing these actions (because the authenticate behavior occurs before the *requiresAuthentication* pointcut stub).

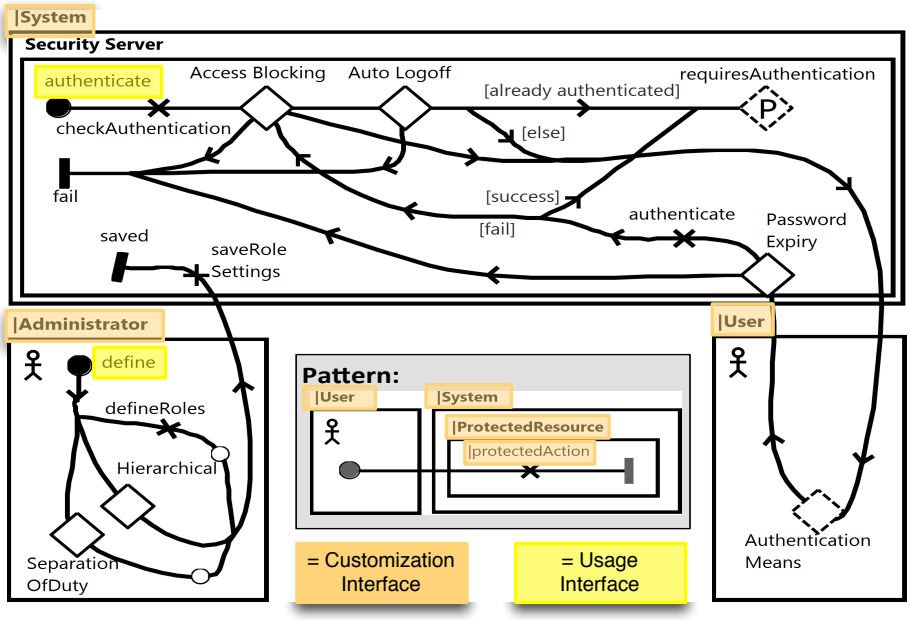

Fig. 3. Authorization Requirements Models

The UI is defined by the start points (i.e., *define*, *authenticate*, and all start points of lower-level workflow models of the variable features depicted by stubs (diamonds)). In Fig. 3 the UI elements are highlighted in yellow.

We used Aspect-oriented Use Case Maps (AoUCM) to represent the workflow of *Authorization*. However, the approach is not AoUCM-specific and could have considered other languages like activity diagrams or BPMN models and their aspect-oriented extensions.

3.3 Design Models

To illustrate reusable software design models, we design realization models for each feature of *Authorization* using RAM [13]. For space reasons, only the RAM model realizing the root feature of *Authorization* is shown in Fig. 4. The RAM model comprises two compartments, the structural view showing the class diagram and the message view defined using sequence diagrams. The partial structural entities such as the class |*User* and the operation |*execute* again designate the CI. The UI is comprised of all public classes and operations.

3.4 Hardware Models

Often, software is connected to specific hardware elements with which it tightly interacts. In the context of *Authorization*, this is the case for specific *Authenti-*

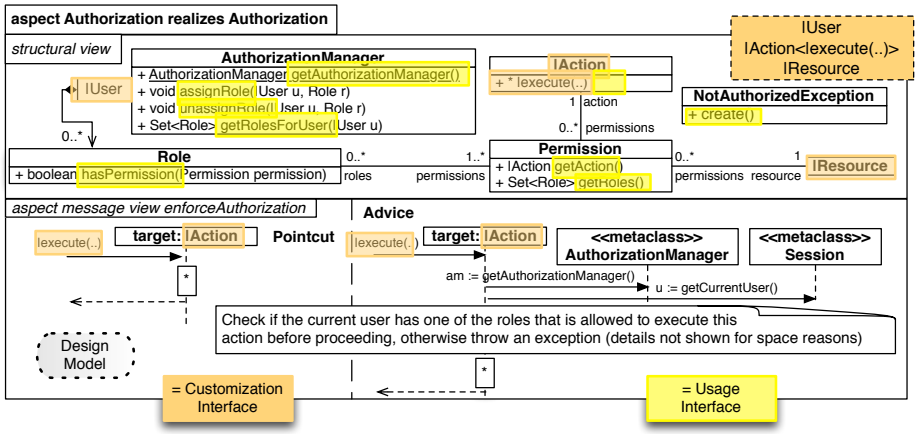

Fig. 4. Authorization Design Models

cation Means like *Fingerprint* or *Facial Recognition*. To illustrate that our interfaces are also capable of dealing with hardware, we present hardware models realizing the *Facial Recognition* feature in the System Model in Fig. 5.

We used SysML to represent the execution platform of *Authorization* because a *Block* in SysML can be realized by hardware or physical elements. However, the approach is not SysML-specific and could have considered other suitable modeling languages like MARTE or AADL. Features are therefore realized not only by workflow models and UML design models, but also by a SysML block diagram specifying the hardware and by a SysML allocation model specifying how the software is linked to the hardware.

Our facial recognition artifact could contain lots of hardware variability (such as different quality cameras, optional light sensors) as shown in the feature model, but for space reasons we are only illustrating one hardware model using a video camera and a luminosity sensor.

The SysML internal block diagram describes the hardware elements that *FacialRecognition* provides to measure physical data: a *VideoCamera* and a *LightSensor*. It also depicts required hardware elements, such as a *PowerSource*, a *LightSource*, a *USB plug*, and at least one *CPU*, and specifies how they are connected to the provided hardware. These elements constitute the CI of our hardware model, highlighted again with the vertical bar. Allocations of drivers to ≪part≫ model elements show how the software relates to the hardware.

The remainder of this subsection summarizes the VCU interfaces for a reusable artifact for system modeling, i.e., when software and hardware are to be made reusable. In this case, the VI also includes the hardware variability offered by the unit of reuse, and the resulting impacts on high-level goals such as cost, power consumption, impact on environment, and noise. The CI also includes hardware elements or physical elements in addition to software elements (e.g., the temperature in the environment), and constraints on their properties (e.g.,

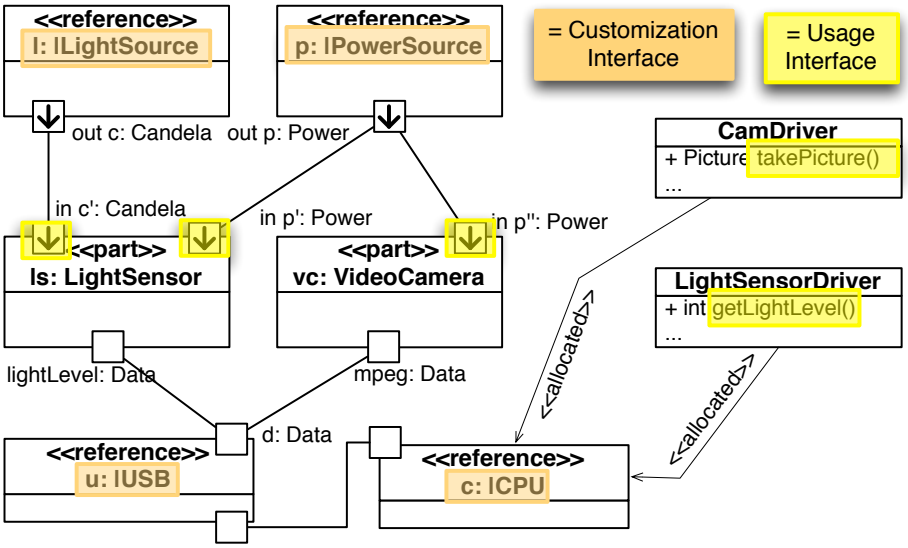

Fig. 5. Authorization System Models

greater than 100 Watt). The UI includes the functionality needed by the user of the hardware artefact during execution/simulation of the model, i.e., interfaces offered by the drivers (e.g., takePicture()) but it also includes the physical data flow ports needed to make the system operational (e.g., a specific amount of electricity through a power connection, a minimum and maximum quantity of lumens into a lens). This hardware model highlights that even if the nature of the UI is different from the one in a software model (i.e., it is not based only on method declarations), the notion of UI is still valid and correctly encompasses these different notions.

3.5 Reusing Authorization

This section illustrates how the *Authorization* concern is reused in a simple ATM machine. The ATM developer selects from the *Authorization* VI the desired features with the best impact (step 1 in 2.4). Based on this selection, the workflow, design, and system models linked to the selected features are composed by the reuse tool to create workflow, design, and system models for *Authorization* that only contain the selected features.

The next step (step 2 in 2.4) is to customize each kind of model by establishing mappings from the *Authorization* CI to the ATM models. To customize *Authorization* to the ATM context at the workflow and design level, $|$*User* of *Authorization* is mapped to the *Customer* component in the ATM workflow model and the *Customer* class in the ATM design model. Similarly, $|$*ProtectedResource* in the workflow as well as $|$*Action* and $|$*Resource* in the design are mapped to *Account*. Finally, the $|$*protectedAction* responsibility in

the workflow and the |*execute* operation of *Action* are mapped to the *withdraw* and *deposit* responsibilities and operations of *Account*, respectively: |`User` → `Customer`; |`ProtectedResource` → `Account`; |`protectedAction` → `withdraw`, `deposit`; |`Action`, |`Resource` → `Account`; |`execute` → `withdraw`, `deposit`.

The internal block diagram model of the ATM machine contains parts representing the specific CPU that was chosen, the memory used, the USB ports, the specific power source chosen, etc. To customize the *Authentication* hardware model, mappings must be established that link the CI model elements to model elements in the ATM machine that satisfy the property constraints, e.g.: |`PowerSource` → `SeaSonicSS;` |`USB` → `MediasonicHP1-U34F;` ...

4 VCU Interfaces Across Reuse Techniques

While the last section demonstrated the use of the VCU interfaces in requirements, design, and hardware models across various levels of abstraction, this section focuses on VCU interfaces in existing reuse techniques.

Usage Interface: The prototypical example of a UI is the API of a class in an object-oriented programming language. Standard classes do not have a customization interface, as all of their public operations and attributes are fully specified and defined (as opposed to generic classes described in the next paragraph). UIs can also be of considerable size, such as the API of an entire library. Libraries, even if comprised by several classes that offer alternative functionality, typically do not have an explicit VI. Information about variants encapsulated in the library, and impacts of the different variants on non-functional requirements and qualities are informally described in textual documentation, if at all.

Customization and Usage Interfaces: Generic classes are a popular reuse mechanism in programming languages, such as Ada, Java, and C++ (where they are called template classes). In essence, a generic class provides a crisp set of functionalities, and for that purpose encapsulates some structure and behavior that is generically applicable for all parameter types. The CI is defined by the parameters, which are classes, types, and often also operations that define what the generic class needs from the reuse context. The programmer must provide the correct parameters at development time, when instantiating the generic class, to customize the class to a particular reuse context and to access its tailored UI.

In modeling, the CI is defined in similar form. The UML template parameters provide the mechanism to tailor models to different reuse contexts. They can be applied to a class as in programming languages. However, template parameters can also be applied to UML packages, thus effectively parameterizing the entire model contained in the package. Many aspect-oriented modeling techniques offer UML template parameters or similar CIs to adapt aspect models that encapsulate reusable structure and behavior to specific reuse contexts (e.g., [3]).

Application frameworks are also composed of classes, but usually focus on providing reusable structure and behavior related to a specific domain (e.g.,

Graphical User Interfaces, Persistence, Banking). By definition, frameworks impose an application architecture, drive the execution control flow, and require the programmer to tailor the framework to their needs and integrate the application's behavior by implementing interfaces or extending classes provided by the framework. The CI of a framework is defined by the interfaces and abstract classes that need to be subclassed by the programmer to reuse the framework. The UI of the framework consists of the public (or published) operations defined in its API.

Components are broad units of reuse that encapsulate a set of classes whose instances collaborate to provide a reusable service. The required interface of a component is a form of CI, since it allows the component to list the services it needs from the reuse context in order to be operational. The provided interface describes the service(s) that the component offers, and hence is equivalent to the UI.

All four discussed mechanisms have clearly defined CIs and UIs. Customization is in all forms applied by binding open holes, namely generic parameters or super classes, with concrete types or subclasses. Usage in all forms is defined by the public (published) interfaces. None of them however explicitly provide mechanisms for expressing the variation they encapsulate, if any. It would be very interesting to add such explicit variation mechanisms into the respective programming or modeling techniques, in order to allow documenting and understanding possible variation and selecting variants at design time. Currently variation can only implicitly be achieved by using the UI, i.e., calling mode-setting functions to adapt behavior, or using the CI by defining several subclasses for hot spots in frameworks. In the latter case, creational design patterns such as *Factory* [8] can be used to select variations encapsulated by the framework at initialization time or at run time.

Variation and Usage Interfaces: A common approach to handle variability at the domain level is to follow a Software Product Lines (SPL) approach [20]. SPL engineering focuses on how to organize similar software products as a family within a closed domain, exploiting commonalities, and managing variabilities among them. Many implementation techniques have been proposed for SPLs, but when the variability is explicitly represented, feature models [11] are then widely used. In this context, feature models are a perfect mechanism for the VI, as they express the (closed) variability of an SPL.

The UI is usually obtained by a derivation process on the SPL assets. Assets can be code, models, or other software artifacts. Two main groups can be distinguished by their way to derive a product, i.e., annotative and compositional. At the model level, annotative approaches [5] normally use annotations on model elements and prune them during derivation. On the other hand, compositional approaches rely on several models or fragments corresponding to the selected features that then need to be well integrated. For structural models, such as class diagrams composition, techniques can notably rely on aspect-oriented modeling [17], model merging [19] or delta modeling techniques [9]. Related to our authorization case study, a recently proposed compositional approach [14]

captures the variability of RBAC models in a feature model to configure an associated UML model.

In the SPL field, researchers have also proposed extensions to feature models so that the VI is enriched with properties on features. This can be done with attributes on features [4], typically representing non-functional properties within the SPL that can be reasoned about. By the scoped and closed nature of a SPL, the CI is not explicitly present, but it is of course possible that the selected and composed assets provide individual customization mechanisms.

The Service-Oriented Architecture (SOA) is a software architecture style that views the system as set of services that are self-contained, loosely coupled, and can be easily composed. SOA provides guidelines that govern how services are represented and used. Services are designed to address business-related behaviour and logic, and are meant to be assembled to build enterprise solutions [16]. Connections between services are flexible, as services are dynamically invoked at run time through a UI that is described, for instance, by means of the Web Service Description Language (WSDL).

There exist SOA approaches that provide a sort of variability interface, which is helpful in choosing the most appropriate service during run-time. Service Level Agreements (SLAs) specify non-functional properties of services, which is a way of specifying the impacts of services that allows for a limited form of trade-off analysis when multiple services providing similar functionality are available.

Table 1. Summary of Common Units of Reuse

Units of Reuse	Usage Interface	Customization Interface	VI Variation	VI Impact
Classes	Yes	No	No	No
Generic Classes	Yes	Yes	No	No
Components	Yes	Yes	No	No
Frameworks	Yes	Yes	Informal	Informal
SPL Features	Yes	No	Yes	No
SPL Features with Attributes	Yes	No	Yes	Yes
Services	Yes	No	Limited	Limited

Variation, Customization, and Usage Interfaces: A summary of the analysis of the most common units of reuse and their support for usage, customization and variation interfaces is shown in Table 1. Although none of the units provides out-of-the-box support for all three interfaces, there has been lots of research extending their reuse potential.

Perrouin et al. [19] have proposed an approach to provide some flexibility by broadening the scope of the captured variability. In a first reuse step, variability is resolved from a feature model selection and a product is generated by automatically merging model elements associated to the selected features. A second

reuse step involves a customization process implemented by a model transformation and validated by OCL constraints defined on the model elements. This can be seen as a first, but not very explicit, form of the CI.

Handling variability while being able to consider unplanned contexts is a problem that has already been tackled in other works, mainly by introducing variability management in reusable units such as components [18, 22] or modules [12]. van der Storm defines variable components [22] and uses solving techniques for checking compatibility among them. In a similar way, Plastic Partial Components [18] are components equipped with several variable interfaces and implemented internally with aspect-oriented techniques in model-driven software architectures. These approaches are more flexible than common SPL techniques, as they are handling variability at the component level, providing VIs (although with limited support for specifying impacts) and UIs. Nevertheless, the customization part is not as fine-grained as in our definition, as it is obtained by the different compositions of components, and not at the level of each component.

Recently, Kästner et al. [12] proposed a core calculus for variability-aware modules, complemented by a C-based implementation. Variability is handled on module interfaces and inside modules, providing a solution that covers all three VCU interfaces. Modular type checking of internal variability is supported and the composition of two compatible modules yields a well-typed module with combined variabilities. However, the notion of impact model in the VI is not covered.

Finally, there have also been efforts to provide CI for services through parameterization and personalization [1] and using templates [23]. [2] proposes an approach that allows customized use of web services in XML documents. The approach uses an XML schema that allows to specify elements/subelements of the XML document that can be specified/replaced dynamically. They provide an example of a schema for news exchange, where the element <item> can be given by a service call that matches the news service call pattern, which allows to use any service call that returns an element (news <item>) of the correct type.

5 Conclusion

Reuse is one of the most important concepts in software engineering to improve system quality, product reliability and in particular developer efficiency, thus reducing development costs. This paper argues that while there is a huge variety in the kinds of reusable artifacts, almost all forms of reuse have in common the need to provide some or all of three key interfaces, i.e., the variation, customization, and usage interface, summarized as VCU interfaces. We have discussed the commonalities and potential consequences of different kinds of reuse dimensions: a) selecting a variant based on information about impacts, b) adapting the generic artifact to a specific context, and c) using the functionality. Furthermore, we discussed how the three interfaces explicitly pinpoint down "where" this reuse

happens. The variation interface is needed to select from a set of choices offered by a reusable artifact while being informed about the impact of the selection. The customization interface is required to adapt a generic reusable artifact to a specific reuse context. The usage interface is needed to define how the services of a reusable artifact may be accessed.

For a better understanding of these interfaces, we have examined their concrete appearance across levels of abstraction (i.e., from requirements to software design and hardware design model) and across reuse techniques (from classes and components to software product lines and services). Based on these findings, we have found that all examined reusable artifacts indeed make use of and only of the VCU interfaces. Today we do not know of situations, where the VCU modeling approach will not hold, but these are preliminary findings. We invite the software reuse community to challenge the sufficiency of the VCU interfaces in the context of reuse. In the future, we plan to make generic support for the VCU interfaces available to several mainstream modeling notations.

References

1. Amazon: Amazon web services
2. Abiteboul, S., Amann, B., Baumgarten, J., Benjelloun, O., Ngoc, F.D., Milo, T.: Schema-driven customization of web services. In: Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29. pp. 1093–1096. VLDB '03, VLDB Endowment (2003)
3. Alam, O., Kienzle, J., Mussbacher, G.: Concern-oriented software design. In: Proceedings of the 16th International Conference on Model-Driven Engineering Languages and Systems - MODELS 2013. Lecture Notes in Computer Science, vol. 8107, pp. 604–621. Springer Berlin Heidelberg (2013)
4. Benavides, D., Trinidad, P., Ruiz-Cortés, A.: Automated reasoning on feature models. In: CAiSE'05. pp. 491–503. Springer (2005)
5. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: Glück, R., Lowry, M.R. (eds.) GPCE. Lecture Notes in Computer Science, vol. 3676, pp. 422–437. Springer (2005)
6. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR, Upper Saddle River, NJ, USA (2005)
7. Ferraiolo, D.F., Sandhu, R.S., Gavrila, S.I., Kuhn, D.R., Chandramouli, R.: Proposed NIST standard for role-based access control. ACM Transactions on Information and System Security 4(3), 224–274 (2001), http://doi.acm.org/10.1145/501978.501980
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison Wesley, Reading, MA, USA (1995)
9. Haber, A., Kutz, T., Rendel, H., Rumpe, B., Schaefer, I.: Delta-oriented architectural variability using monticore. CoRR abs/1409.2317 (2014)
10. International Telecommunication Union (ITU-T): Recommendation Z.151 (10/12): User Requirements Notation (URN) - Language Definition (approved October 2012)
11. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, SEI, CMU (November 1990)

12. Kästner, C., Ostermann, K., Erdweg, S.: A variability-aware module system. In: Leavens, G.T., Dwyer, M.B. (eds.) Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012. pp. 773–792. ACM (2012)
13. Kienzle, J., Al Abed, W., Klein, J.: Aspect-Oriented Multi-View Modeling. In: Proceedings of the 8th International Conference on Aspect-Oriented Software Development - AOSD 2009, March 1 - 6, 2009. pp. 87 – 98. ACM Press (March 2009)
14. Kim, S., Kim, D.K., Lu, L., Kim, S., Park, S.: A feature-based approach for modeling role-based access control systems. Journal of Systems and Software 84(12), 2035–2052 (2011), http://dx.doi.org/10.1016/j.jss.2011.03.084
15. Krueger: Software reuse. CSURV: Computing Surveys 24 (1992)
16. Krut, R., Cohen, S.: Service-oriented architectures and software product lines - putting both together. In: 12th International Software Product Line Conference - SPLC '08. pp. 383–383 (Sept 2008)
17. Morin, B., Vanwormhoudt, G., Lahire, P., Gaignard, A., Barais, O., Jézéquel, J.M.: Managing variability complexity in aspect-oriented modeling. In: Czarnecki, K., Ober, I., Bruel, J.M., Uhl, A., Völter, M. (eds.) Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5301, pp. 797–812. Springer (2008)
18. Pérez, J., Díaz, J., Soria, C.C., Garbajosa, J.: Plastic partial components: A solution to support variability in architectural components. In: WICSA/ECSA. pp. 221–230. IEEE (2009)
19. Perrouin, G., Klein, J., Guelfi, N., Jézéquel, J.M.: Reconciling automation and flexibility in product derivation. In: SPLC. pp. 339–348. IEEE Computer Society (2008)
20. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2005)
21. Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. IEEE Computer 29(2), 38–47 (Feb 1996)
22. van der Storm, T.: Variability and component composition. In: ICSR. Lecture Notes in Computer Science, vol. 3107, pp. 157–166. Springer (2004)
23. ten Teije, A., van Harmelen, F., Wielinga, B.: Configuration of web services as parametric design. In: Motta, E., Shadbolt, N., Stutt, A., Gibbins, N. (eds.) Engineering Knowledge in the Age of the Semantic Web, Lecture Notes in Computer Science, vol. 3257, pp. 321–336. Springer (2004)