Compositional Variability — Concepts and Patterns*

Mark-Oliver Reiser
Technische Universität Berlin
Fakultät IV – Softwaretechnik – FR5–6
Franklinstraße 28/29
D-10587 Berlin, Germany
mark-oliver.reiser@tu-berlin.de

Ramin Tavakoli Kolagari
Volvo Technology Corporation
Mechatronics and Software
Dept 6260, M2.7
SE-405 08 Gothenburg, Sweden
ramin.tavakoli@volvo.com

Matthias Weber
Carmeq GmbH
Carnotstraße 6
D-10587 Berlin, Germany
matthias.weber@carmeq.com

Abstract

Most software-intensive systems rely on a component-based design and are therefore made up of encapsulated structural units which are hierarchically composed of one another. In this paper, we (1) propose a scheme for rigorously managing variability in the context of such a compositional hierarchy, which consistently extends the paradigm of component-based design to variability management, (2) present several basic patterns of specifying variability when applying this scheme in practice, and (3) show how all this was technically realized in EAST-ADL2, an architecture description language for automotive software development. While the observations and concepts discussed in this paper emerged from an automotive context, they are arguably applicable to many other industrial domains involving software-intensive systems.

1. Introduction

Hierarchical composition is the key organization scheme used in the design of software-intensive systems. This applies both to the top level, where the complete system is divided into a number of broad subsystems, and to the lower hierarchical levels, where system functionalities are composed of many fine-grained, elementary software or hardware functions. Such a component-based design has manifold benefits [4]. Interestingly, when it comes to variability management, traditional industrial approaches usually reject this paradigm, organizing the configuration parameters globally and thus orthogonally to the component hierarchy, as will be detailed later in Section 8. By contrast, in this paper we propose a scheme for managing variability in a strictly hierarchical fashion, called compositional variability management, thus incorporating the core benefits of component-based design within variability management.

Instead of presenting a single, specific technique for compositional variability management, we provide a general description and discussion of this notion in order to show that (a) this is a basic scheme for managing variability within compositional structures which might be realized technically in many different ways and, (b) the observations presented in this paper, in particular the application patterns identified in Section 5, are equally applicable to all these potential realizations. To prevent the discussion from becoming too abstract, numerous examples are provided and our specific, technical realization of the concepts is briefly outlined in Section 7.

The remainder of this article is organized as follows. After introducing several basic terms and concepts in the next section, we investigate in detail the characteristics of a hierarchical structure of components and subcomponents (Section 3). Based on this discussion, a scheme for rigorously managing variability in such a composition hierarchy is then proposed in Section 4, this we term compositional variability. When applying this scheme in practice, several typical patterns of variability specification occur, which are identified in Section 5. Section 6 discusses the applicability and benefits of compositional variability management, and Section 7 gives an overview of our implementation of the concept. Finally, related work is investigated in Section 8.

2. Terminology

Before going into details, we define a number of basic terms in this section.

A first conception required in the following discussion is component-based design [11], which is a software engineering paradigm emphasizing the decomposition of the engineered systems into functional or logical units, called components, with well-defined interfaces used exclusively for communication across these units of composition. Components do not share state and they communicate by exchanging messages or signals carrying data. In most methodolo-
features for component-based design, components have the following constituents: ports that represent anchor points for communication with the outside, subcomponents that can be contained within a higher-level component and communication links connecting ports in order to establish well-defined communication paths between components. An example of a component with these constituents is given below in Figure 2. More details about component-based design, and in particular composition hierarchies of components, are given below as needed for the discussion.

The other important conception this paper relies on is variability and variability management. When dealing with variability there is always some variable entity that exists in a variety of forms, called variants. For example, the car manufacturer Mercedes-Benz deals with the variable entity “Mercedes-Benz Car” with variants such as “A-Class”, “C-Class”, and “E-Class”. The strategic management of commonalities and differences between such a variable entity’s variants is called variability management, sometimes also known as variability and commonality management to emphasize that commonalities are also focused on.

One of the commonest concepts for variability management are feature models, introduced by Kang et al. in [5]. They are used to document the characteristics common to all variants of a variable entity and the characteristics that differ from one variant to another. Dependencies between characteristics are also defined in feature models. In this context, a feature is a certain characteristic or trait, in the broadest sense, that an individual variant may or may not have. During configuration, these features can be selected or deselected for an individual variant. Often, feature models are hierarchically structured as feature trees, as in the example of 1. In such a feature tree, children depend on their parent in that they can only be selected if their parent is selected. In the example, Radar is a child of Adaptive and can therefore only be selected if this advanced version of the cruise control is selected.

Features can either be mandatory or optional. Mandatory features, denoted by a filled circle, must always be selected if their parent is selected whereas optional features, denoted by an empty circle, need not necessarily be selected if their parent is selected. For example, all cars have a Wiper because this feature is mandatory, but they need not have a CruiseControl because this is optional; similarly, all CruiseControls have the AcceleratorActuator. Two or more siblings can be marked alternative by connecting their parent/child relations with an arc, as shown in the example for Simple and Adaptive. Then, exactly one of the alternative features must be selected whenever their parent is selected. In the example, this means that whenever the CruiseControl is selected, either Simple or Adaptive has to be chosen. The final concept needed in the discussion below are parameterized features. Such features are not only selected or deselected during configuration, but a value of a certain type has to be provided for them during configuration if and only if they are selected. In the example, the feature Radar was defined to be parameterized by type Int, which means that when selecting the Radar during configuration, an integer value also has to be provided. The meaning of this value—for example the minimal distance to the vehicle ahead before the adaptive cruise control automatically reduces speed—has to be explained in the feature’s textual description (not shown in the figure).

A multitude of variations of the above concepts and of additional concepts are discussed in the literature. In our context, however, further details of feature modeling are not required; they can be found in [2, 7, 10, 8].

### 3. Point of Departure

Let us now return to our initial goal of introducing rigorous variability management in a hierarchical structure of components and subcomponents. As a first step, we begin by clarifying in this section a few important characteristics of such component hierarchies and by introducing a running example. Based on this, we then precisely formulate the concept of compositional variability management in the next section and identify important application patterns in Section 5.

A component hierarchy is made up of several components that are composed of one another, i.e. each component may contain one or more subcomponents. We refer to components that actually contain subcomponents as composite components and to those that do not as elementary components. Figure 2 shows an example of a component definition, where a wiper controller is defined as being composed of a threshold component and an integrator that arbitrates between the driver’s request for wiping and the input from the rain sensor.

As trivial as this may seem, there is an important intricacy to be noted about these component definitions, related to the precise semantics of containment. Con-
Figure 2. A simple composite component.

sider, for example, the nature of the rectangle labeled rsThr:Threshold in Figure 2: this rectangle represents neither a class nor an instance. If it constituted a class definition, all the internal details of the threshold component would have to be defined at this location and only a single such class definition would be allowed per system definition, which is obviously not the case because the threshold component may also be used in components other than WiperController. On the other hand, if the rectangle denoted an instance, this would mean that at the implementation level, exactly one instance of the threshold rsThr would have to be present for each such rectangle, i.e. there would have to be a 1:1 relation between the symbol in the diagram and actual instances at the implementation level. At first sight, this might seem to be the case, but if the WiperController is used several times within a single complete system, e.g. for the front and the rear wiper, then several instances will be created for the single rectangle in the WiperController component diagram, e.g. one threshold instance for the front and one for the rear wiper. This means that the rectangles represent neither classes nor instances but rather denote an intermediate concept between class and instance. In early publications on component modeling, this concept was sometimes illustratively described as a “place-holder for instances”, which insinuates that for each such rectangle (the rsThr in the example), an instance of the corresponding component will be created whenever the containing component (the WiperController in the example) is instantiated.

For this reason, most modeling languages that support component diagrams provide a dedicated term for these rectangles that denote subcomponents within a supercomponent’s definition. In UML2, they are called Parts; in methodologies and modeling languages that emerged from an automotive context, such as AUTOSAR or EAST-ADL, the synonymous term Prototype is more common. This peculiarity of component diagrams must be given careful consideration when introducing variability management in such a context.

But before doing so, let us quickly extend the component definition from Figure 2 to give a more complete example which can be used for illustration purposes throughout the remainder of this paper. This is shown in Figure 3. Here, the WiperController is employed twice to form the complete wiper system. Also, the wiper motor is reused for front and rear wiping. The switches for controlling the wipers, however, are modeled in two distinct forms for the front and rear switch. Finally, a single rain sensor is used for both the front and rear wiper.

4. Compositional Variability

Using feature modeling, we now wish to introduce variability in such a hierarchical composition structure. As already indicated above, we must extend the component model for this purpose. The key extensions can be summarized in three steps:

1. Means for defining the internal structure of composite components in a variable form,
2. Components are supplied with a public feature model,
3. Each component defines a mapping from its public feature model to (a) the variant of its internal structure and (b) the public feature models of its contained subcomponents.

Step 1 is fairly obvious: when seeking to introduce variability in a component hierarchy, we must also support variabilities in the component structure, e.g. subcomponents may disappear in certain variants or the graph of communication links between subcomponents may differ from one variant to another. To allow such structural variabilities to be defined, an appropriate means of expression must be provided. At this point, a wide range of different solutions are conceivable, but their technical details have no impact on the fundamental idea behind compositional variability, which is the focus of this discussion. We therefore do not elaborate on this variability mechanism here; it is sufficient to assume that we have some technical concept at hand to express such variabilities in a component’s internal structure.

Steps 2 and 3 are more significant in our context because this is where variability management is actually designated to become compositional. As one of their key characteristics, all component models identify in detail the information that constitutes the public interface of a component. They all provide something like ports, through which information is transmitted to or from the component; some provide means for specifying precise communication protocols using state machines, for example. To deal with variability, we extend this interface specification by a feature model, called the public feature model of the component. The purpose of this feature model is to specify the entire variability...
Besides this extension to the public interface, we also introduce, in step 3, an extension to the internal structure: in each variably defined component, i.e. each component that provides a feature model in its public interface, a mapping is defined from this public feature model to the variants of its internal structure (step 1) and the public feature models of the directly contained subcomponents (step 2 w.r.t. the subcomponents). “Mapping” in this sense means that by applying this information, it is possible to derive a configuration of the component’s internal structure and the public feature models of the contained lower-level components whenever a configuration of the component’s public feature model is provided. In other words, the mapping states how to internally configure a component depending on the configuration of its public feature model.

Consider, for example, a cruise control system provided in a simple and an adaptive form, the latter offering a mode of operation in which the vehicle’s speed is automatically reduced as soon as the distance to the vehicle ahead falls below a certain threshold. The definition of the cruise control’s internal structure will somehow state that the radar subcomponent is optional (step 1), its public feature model will specify that two alternative variants Simple and Advanced are available (step 2) and, finally, the cruise control will be supplied with a mapping stating that when Simple is selected, the optional radar is to be removed, and when Advanced is chosen, it is to be retained (step 3).

The fact that the mapping introduced in step 3 is provided as part of a component’s internal structure already underlines that it is not revealed to the outside world. This means that the precise internal realization of some variability presented in the public feature model (e.g. as an optional feature), in particular whether it is realized by structural variability or by a variable subcomponent, is completely concealed from clients. In addition, how this internal variability is actually configured when an individual variant of the containing component is chosen is also hidden. This is perfectly in line with information hiding, the basic idea behind all public interface specifications. The public feature model can therefore justifiably be viewed as an equal constituent of a component’s public interface and the term configuration hiding can be used for this concealment of variability and configuration-related information.

In addition to the basic fact that this information is hidden from the outside world, another important analogy to interfaces and information hiding in general should be noted here. Just as the method signatures of a class in object-oriented programming need not correspond to the internal implementation details of this class, the public feature model of a component need not be in line with how the variability is internally structured. The public interface of a class may suggest that this class has three fields, e.g. by providing three pairs of getter and setter methods, whereas at the implementation level only two fields are actually used, while the third property is being derived from these. This is also the case with the public feature model: the variability of contained subcomponents as defined by their public feature models can be packaged and presented completely orthogonally in the containing component’s public feature model. This is of particular significance in practice, as we shall see in detail when investigating patterns of compositional variability below.

A final remark on step 2 is in place here. When discussing the notion of compositional variability with practitioners—especially the idea of attaching feature models to components—we often encounter a certain reservation. Among engineers, feature modeling sometimes has
the reputation of being highly abstract, vague, and thus inappropriate for detailed specification of low-level implementation aspects. This need not be the case. A feature modeling technique can—and should—be defined soundly with formal syntax and semantics and it is then suitable for precisely specifying implementation details. In more concrete terms, a set of precise, implementation-specific configuration parameters can be straightforwardly expressed as a feature model by simply defining a flat model containing only parameterized root features, which proves that feature models can—but need not—be a highly abstract modeling means. Such “degenerate”, but still perfectly legal feature models are of particular relevance in the context of compositional variability, especially at the lower levels of system design.

5. Patterns

This scheme for defining variability in compositional structures now provides us with a flexible instrument that can be used in a variety of ways. In this section, we look at such application patterns and attempt to show that they are quite well suited for typical real-world application scenarios.

Pattern #1: Plain Propagation

First of all, some variability published by a contained lower-level component may simply be added in identical form to the containing component’s public feature model, together with a one-to-one mapping from there to the subcomponent’s feature model. This way, the variability is actually propagated up to the next level in the containment hierarchy; the decision on when and how this variability is to be bound is simply deferred during the design of this component and thus delegated to the engineer who will employ it within some other, higher-level component.

An example of this scenario is given in Figure 4, where the wiper controller introduced in Section 3 is augmented with variability specifications. The Threshold component now provides a public feature model with a single parameterized feature called thresholdValue of type float, depicted as a dashed rectangle in the top-left corner of the rsThr prototype. Next, the containing WiperController component is also supplied with its own public feature model, again depicted as a dashed rectangle in the top-left corner. As described above for this application pattern, it contains an identical copy of the variability specification to be propagated upward, i.e. also a single parameterized feature of type float with the same meaning. Finally, a one-to-one mapping is provided. This way, the variability related to the threshold value is simply propagated up to the next level of component containment.

Note that the public feature model of Threshold appears, but is not defined, in Figure 4; it is, instead, defined in the Threshold component definition (not shown). This is exactly the same as for the ports of Threshold, which also only appear here but are defined elsewhere, providing further evidence that the public feature models can properly be viewed as a constituent of a component’s public interface. The public feature model of WiperController, on the other hand, is actually defined—and could therefore be modified—in Figure 4, because the figure actually shows the component definition of the WiperController.

Pattern #2: Direct Binding

Instead of being propagated up, lower-level variability introduced by subcomponents may also be directly bound within the definition of a containing, higher-level component. In the example of Figure 4, for instance, the threshold value of the rsThr prototype could have been set to a certain, invariable, value, instead of providing a corresponding feature in the public feature model of WiperController. Such a course of action would make sense, if the wiper controller were intended for reuse only in cases where always the same threshold value is required; the Threshold component could thus be defined variably without having to make WiperController a variable component, even though it contains Threshold. Thus, the lower-level variability is completely eliminated by the design at this point.

In steps 1, 2 and 3 in the previous section, we did not provide a dedicated technique for specifying the configuration of lower-level components within a higher-level component definition. This was unnecessary because such an invariable configuration of lower-level variability can be realized technically as a special case of the mapping from step 3: as a “constant” or invariable mapping which is completely independent of the configuration of the containing compo-
Pattern #3: Orthogonal Propagation

More interestingly, a third option regarding the propagation of lower-level variability is conceivable. Instead of variability being propagated without change or not at all, it can be propagated in a different form. This includes both a partial propagation as well as a propagation in a diversely structured and packaged form, illustrated in Figure 5. It shows a situation similar to that in Figure 4, but instead of directly propagating the variability of Threshold to the public feature model of WiperController, we chose to present this variability in a different form. A small feature model with two alternative features now defines the variability of WiperController, i.e. there now remain only two variants Front and Rear, and the mapping specifies how to configure the thresholdValue in rsThr depending on the configuration of this feature model. As trivial as this example may be, it shows three extremely important use cases of orthogonal propagation: the lower-level variability introduced by the contained Threshold component is

1. Reduced (the infinite variants of Threshold, i.e. one per float value, were reduced to only two variants, i.e. Front and Rear),
2. Diversely Structured (the single, parameterized feature was turned into two alternative features),
3. Semantically Enriched (we not only chose the two variants 140 and 80 from the infinite number of variants—covered in item 1 above—but we also stated that one is intended to be used for the front and the other for the rear wiper and that 140 is to be used for the front and 80 for the rear wiper).

To further clarify item 2, let us consider an example: we might define a wrapper component WiperController2, which contains only a single prototype of WiperController, called wc. In the public feature model of WiperController2 we could present the variability of the wiper controller as a feature model in which the parent feature Wiper has only a single, optional(!) child feature, called IsRear, and define a mapping which selects wc.Rear if IsRear is selected and wc.Front otherwise. Then we have not changed the number of variants (the single optional feature also defines two variants) and have not changed the semantic meaning of the variants (if IsRear is unselected, then the standard case of the front wiper is chosen), but we have changed the structure of the variability definition (a single optional instead of two alternative features). When dealing with larger feature models, especially when they are maintained by different personnel, it is extremely useful to be able to change the structure of variability definitions in this way.

In principle, these three patterns completely cover the different ways of coping with variability in composition hierarchies. However, during our case studies at Daimler AG and in the context of the ATESSST project, two other important patterns showed up. We found that the above three patterns all employ the notion of compositional variability in a very strict, forceful way and that sometimes a more pragmatic procedure is desirable. This led to the following two patterns.

Pattern #4: Top-Level Propagation

It is sometimes desirable, or even indispensable, to state at a very low hierarchical level that a certain variability, represented by one or more features, is directly propagated up to the top level, called top-level propagation here. According to compositional variability as delineated in Section 4, variability is usually propagated up in a step-by-step manner, one hierarchy level at a time, but sometimes it has to be propagated up to the top level in a single step. The key point here is that, in the standard case, the decision to propagate to the top level is distributed among all higher levels: at each level of the hierarchy the propagation may be broken by actually binding the variability (pattern # 2 above). In the case of an immediate top-level propagation, however, the decision to make the variability appear at the top level is taken and fixed at the lower level where the variability occurred and designers reusing this component at intermediate levels cannot avoid this—except by refraining from using this component, of course.
Pattern #5: Global Features / Reverse Propagation

Another important observation, also related to the use case of compositional variability, is that sometimes global features are required at the lower hierarchical levels. The difference to the standard case of features defined “locally” within the public feature models of lower-level components is that such global features can be used in different subtrees of the system’s composition hierarchy and will still eventually be configured identically. Since such global features can be available within the containment hierarchy only below the point where they are defined, and since they are therefore usually defined near the top level, these global features can be perceived as the counterpart of features which were top-level propagated; to emphasize this, we may also speak of reverse propagation.

It is important to note that patterns #4 and #5 violate basic rules and assumptions of compositional variability and component-oriented software engineering. Or, to put it more kindly, they represent pragmatic compromises that are often useful in the design of real-world systems. In the case of top-level propagation, a lower-level component restricts the design choices at higher levels (i.e. whether or not variability is propagated further up), and in the case of global features, lower-level components make—and thus rely on—assumptions on the higher levels (i.e. which global features are defined there and what their precise meaning is). Interestingly, this pragmatic deviation from the basic principles of component-oriented system design has an analogy among traditional concepts of component modeling. In particular, many methodologies for component modeling provide special “global” ports which are immediately available in all components. For example, in UML/RT, one of the precursors of the component diagrams in UML2, this concept was called system port. Such ports are intended for direct communication with the component infrastructure, the middleware, hardware drivers or the operating system. When using such system ports, a lower-level component makes and then relies on assumptions about the surrounding system in much the same way as described above for global features. Global features can therefore be perceived as the equivalent of system ports for variability specification.

If this is deemed insufficient to justify such a serious deviation from the principles of component-oriented design, or if the downsides of applying them are expected to outweigh their benefits in a particular application context, then the use of global features can easily be forbidden or restricted by appropriate tool support. They should be seen as an option that is highly versatile in some circumstances but should be employed with caution and only when actually needed (interestingly, the same was said of system ports in the early publications on UML/RT).

6. Discussion

The various patterns of compositional variability management identified in the previous section already show that such a hierarchical approach to variability provides a great degree of flexibility. At each level of composition, additional variability can be introduced, existing variability of contained lower-level components can be bound or retained and all the remaining variability can be freely organized and presented in a suitable form, which may be completely orthogonal to the internal variability specification and can therefore be tailored freely to suit the application context and purpose of the containing component. Equally important, this will often be useful to reduce the complexity of system variability. In the example above, the enormous variability of the threshold component (i.e. the infinite number of variants) was reduced in the wiper controller component to the choice between a front and a rear variant. This reduction in complexity may not be too impressive when applied in a toy example as above, but when used consistently throughout a complex system design, this may enormously abbreviate the system’s overall variability specification.

Beyond flexibility and reduction of complexity, compositional variability has another important benefit: it can be used to gradually shift the perspective of variability specifications from a detailed technical viewpoint to a more abstract, maybe even client-oriented viewpoint. At the lower hierarchy levels, the variability specifications usually directly refer to technical aspects such as hardware-related parameters, signals, and calibration values. Near the top, complete system level, the components represent large coarse-grained subsystems of the car, or even the entire car, and their public feature models therefore usually reflect the structure of the company’s overall model range. This is illustrated in Figure 6, which shows the changing viewpoint of a cruise control’s variability specification. At the lowest level, we again encounter our Threshold component, which is used to decide whether the actual distance to the next car has fallen below the minimum distance allowed\(^1\).

Its thresholdValue parameter is highly detailed and implementation-specific because the actual value used here must be compliant with the distance values received from the Radar, e.g. it must be based on the same physical unit. At the next hierarchy level, this is enriched with some semantic meaning, i.e. the minimum distance allowed, which already constitutes a first slight change in perspective. At the level of the body electronics system—comprising wiper, lights, climate control, cruise control, etc.—the variability specification is changed further: instead of providing the option of an adaptive cruise control, a comfort version of the

\(^1\)As this is a control engineering problem, it would be solved differently in real-world applications; the simplified example given here is intended for illustration purposes only.
cruise control (i.e. ComfortCC) is provided, which comprises and adaptive cruise control but may also comprise other optional features of the cruise control not shown in the sample excerpt of Figure 6. This means that several fine-grained options related to the cruise control were packaged into a single coarse-grained variant of a comfort version; the variability was thus diversely packaged based on a shift in viewpoint at this level of hierarchy. Finally, the top level carried this to extremes. Here, the variability specification is structured along completely diverse categories: the model, i.e. A-, C-, and E-Class, and the market, i.e. either U.S. or Europe / Middle-East / Asia (EMEA). The example assumes that in the A-Class the comfort version of the cruise control is completely unavailable, while in the E-Class it is a standard equipment and in the C-Class it is offered as part of the comfort version of the C-Class (mappings not shown in the figure); in the variability specification, this is reflected by the fact that only the C-Class feature has an optional child to select or deselect the ComfortCC. Note that, again, the Comfort feature below C-Class need not only comprise the comfort cruise control but may also include other advanced functionalities, e.g. a wiper with a rain sensor. The minimum distance allowed, which is required to configure the low-level adaptive cruise control component, could depend on the market. In the EMEA market, for example, a shorter distance might be required than in the U.S. owing to different local legislation. In that case, this parameter would not appear at all in the public feature model of the Car component; instead an appropriate mapping (step 3 above) would be defined in Car to configure the cruise control appropriately, depending on the selected market. Whether or not this is desirable is a design choice at the level of the Car component.

When now comparing the public feature model of the top level to those at the lowest levels, we observe a significant change in viewpoint: from a purely technical, highly implementation-specific perspective to an abstract, high-level perspective that divides the configuration space of the complete system into broad areas based rather on management- and marketing-motivated criteria. Thanks to compositional variability, this significant shift in viewpoint can be realized gradually, in a step-by-step fashion from one hierarchical level to another.

Regarding the practical application of compositional variability, two minor reservations should be made here. First, in many application scenarios, not a complete composition hierarchy from the lowest levels to the complete system level will be covered. For example, automotive suppliers do not normally consider the integration level of the complete system, while automotive manufacturers focus on precisely that integration level without covering the lower implementation levels of most subsystems. But even if only some of the hierarchy levels of a complete system are covered by a company, and the shift in viewpoint will therefore be less significant, the ability to shift the perspective of variability specification—and thereby abstract away parts of the lower level’s configuration complexities—will be of great value.

Second, even if all levels of hierarchy are covered, the top level will often be slightly more technically oriented than shown in the example in Figure 6. We provided an extreme case here to illustrate the idea of shifting viewpoints more clearly. The fact that the change in perspective from bottom to top level will be slightly smaller in practice than in the example does not reduce the value of this mechanism in practical applications.

7. Realization in EAST-ADL2

As emphasized in the introduction above, this paper does not aim to present a single technique for compositional variability in complete detail. Instead the approach is presented and discussed at an abstract, fundamental level. Many different technical realizations of this approach are conceivable. This applies in particular to the technique for defining variability related to a component’s internal structure (step 1 above). Nevertheless, to give an idea of how the approach can be realized in general, in this section we briefly sketch the related concepts of EAST-ADL2, an architecture description language specifically designed for the automotive domain in the European research project ATESST [1].

First of all, one of the main constituents of EAST-ADL2
is a component model, provided with a diagramming technique. Components are called software or hardware functions, depending on whether the represented functionality is to be implemented in software or hardware. The traditional constituents of component diagrams are provided—e.g. prototypes, ports and signals—together with some additions specifically tailored to the needs of the automotive domain, such as precise semantics of communication entities.

In addition, EAST-ADL2 offers a complete feature modeling technique, supporting advanced feature-modeling concepts such as cloned features and feature parameterization. This feature-modeling technique is then used to augment software and hardware functions with public feature models, exactly as postulated in step 2 in Section 4. To fulfill step 1, modeling means are provided to variably define a function’s internal structure, based on a simple variation point mechanism: a prototype within a function definition can be marked as a variation point and is then just a place-holder for some variable content; what content can be put inside is specified by one or more variants per variation point. Step 3 is then realized by providing a technical concept called configuration links. They establish a link from one or more source to one or more target feature models with respect to configuration, i.e. whenever a set of configurations of all source feature models is given, configurations of the target feature models can be derived by applying this configuration link. Such a configuration link consists of a set of so-called configuration decisions, each capturing a single consideration affecting the configuration of the target feature models depending on the source feature models. The gray arrows in Figures 4 and 5 provide examples of such configuration decisions. The feature-modeling technique and the concepts of configuration links and configuration decisions are defined on a formal basis, including formal syntax and semantics, this being necessary to precisely settle all the details of such a mapping, which is of considerable complexity, especially when it comes to configuring feature models with cloned features.

All these modeling concepts are implemented in the form of a prototypical research tool for editing EAST-ADL2 models. The following list summarizes only the most important functionalities which are of particular interest in the context of this paper:

- Editing EAST-ADL2 component diagrams,
- Editing public feature models of software and hardware functions,
- Editing the mapping from a function’s public feature model to its internal variability,
- Deriving lower-level configurations from higher-level configurations.

With the functionality mentioned in the last item, it would be possible, for example, to derive a configuration of the CruiseControl component from Figure 6 from a given configuration of the Car component. For this purpose, a configuration of the BodyElectronics-System would first be derived from the Car-configuration and this configuration would then be used to derive the CruiseControl-configuration.

The ATESSST project was also the context in which compositional variability was evaluated. In particular, the EAST-ADL2 was applied in an extensive case study with two automotive subsystems, an anti-blocking break system and parts of an engine management system, which were reengineered from previously existing case studies provided mainly in the form of Matlab Simulink models by the project’s industrial partners. While the basic conception of compositional variability, as delineated in this paper, can now be regarded as fairly well consolidated and reasonably well evaluated, the precise details of the corresponding technical concepts in EAST-ADL2 are still under investigation and finalization.

8. State of the Art & Related Work

Traditional techniques for configuration management commonly used in the automotive domain [9], which, of course, also constitute a form of variability management, usually deal with configuration parameters on a global scale only. Configuration parameters are defined at a global level and are immediately available at all levels of the component hierarchy in identical form. It is not possible to newly introduce a parameter in the context of an individual component—e.g. a wiper controller—which is then differentiated for each use of this component in the complete system—e.g. for the front and rear wiper controller—and it is not normally possible to hierarchically organize the binding of these parameters or to change their structuring and presentation. In other words, such “flat” variability management with plain configuration parameters takes no account whatsoever of the hierarchical structuring of the design. Obviously, most of the benefits of compositional variability management are therefore not available when following such an approach. On the other hand, compositional variability management as defined above contains this global scenario as a special case by way of global features (cf. pattern #5 above), thus supporting it for application contexts where this is deemed desirable; it is even possible to freely combine the global style of variability management with the hierarchical one within a single system.

Some development techniques from the automotive domain with support for configuration management (e.g. the TITUS design technique [3]) enable the values of the configuration parameters to be set in a hierarchical fashion, thus
extending traditional configuration management. However, in these cases only flat lists of configuration parameters are supported (instead of feature trees in our case) and, more importantly, the structuring and packaging of these parameters or their semantic connotation cannot be changed from one hierarchy level to the next. Such approaches to hierarchical configuration can also be seen as a special case or as one constituent of compositional variability management.

The basic idea of managing variability in a hierarchical manner is not new. Van Ommering et al. propose to organize the variation within a component hierarchy by defining and binding variability at each hierarchical level, similar as described above [12]. In fact, their KOALA component model can arguably be seen as a technical realization of compositional variability as defined above: step 1 is realized with function binding, switches, and optional interfaces; step 2 with diversity interfaces; and step 3 with diversity spreadsheets. However, these means of expression are tightly coupled with the other elements of the component model, e.g. the communication ports, which means that even with respect to steps 2 and 3 variation specification is tightly coupled with the component model. The procedure proposed here is different in that it takes up the established technique of feature modeling for this purpose. With their tree structure and clear modeling means for expressing variability, feature models are an ideal tool for publicly specifying a component’s inner variability (step 2), especially on the higher levels in the system design. In addition, such a standardized, more abstract technique provides advantages in more heterogeneous application scenarios, i.e. when the variability management covers other types of artifacts as well, such as requirement specifications on analysis level or test case definitions. The notion of compositional variability thus aims to generalize the elements and benefits of KOALA’s variation management and reconcile them with the well-established technique of feature modeling, thus strengthening the ability for a shift in viewpoint from lower to higher levels and retaining variability management as an orthogonal dimension of development.

Finally, Krueger has described in [6] how modularization and hierarchical decomposition can be used to cope with combinatoric complexity in product lines of software-intensive systems. The hierarchical organization outlined above in Section 4 is different in that it only relies on feature models and some mapping between them, while in [6] the additional concept of matrices is introduced to allow the hierarchical composition of modules. In addition, configuration links are designed to allow variability to be presented in a completely orthogonal manner in order to provide the benefits discussed in Section 5 for pattern #3 and in Section 6, which is not the case for matrices.

9. Conclusion

In this paper we discussed variability management for systems with a component-based design. Based on a brief discussion of the characteristics of component hierarchies a rigorous scheme for compositional variability management was proposed in Section 4. This scheme consistently extends the paradigm of component-based engineering to variability management, thus allowing variability to be organized and bound in a hierarchical manner. Several patterns of specifying variability were identified and discussed (Sections 5 and 6) and it was shown that compositional variability management comprises—but also goes beyond—traditional schemes for managing variability in composite structures (Section 8).

References


