Dealing with Variability in Architecture Descriptions to Support Automotive Product Lines

Stefan Mann
Fraunhofer Institute for Software and Systems Engineering ISST
Steinplatz 2, 10623 Berlin, Germany
stefan.mann@isst.fraunhofer.de

Georg Rock
PROSTEP IMP GmbH
Dolivostr. 11, 64293 Darmstadt, Germany
georg.rock@prostep.com

Abstract
Architectural description languages (ADLs) are essential means for a system and software design in the large. Their common concepts are components, ports, interfaces and connectors. Some of them already support the representation and management of variance, a prerequisite to support product line engineering, but the support of variability often stops on component level. In this paper, a more detailed view on the integration of variability into architectural models is taken. The focus is set on providing support for product line engineering within the automotive E/E\textsuperscript{1} domain, where functionality and/or its realization is varying according to specific customer needs and hardware topologies. In general, the fundamental question in this domain is not, whether a product line approach should be applied, but what is the best way to integrate it.

1. Introduction

Architecture description languages (ADLs) are widely used to specify systems and software designs in the large. According to nowadays complexity of embedded software systems in the automotive industry (more than 2000 functions in today’s upper class vehicles) architectural models specified in an ADL have to be structured in different layers. We propose to use an appropriate compositional specification formalism that gives us on the one hand the possibility to analyze the described models concerning their variability at each layer separately and on the other hand the possibility to integrate them within a common model in the overall development process. The various layers introduce different perspectives as for example a functional architecture perspective that describes the hierarchical structure and the corresponding interfaces of a system. During the development of complex systems and in a distributed engineering task such a structured and modular approach is indispensable. Besides these aspects it is important to improve the system development process at each stage of such a development, especially with respect to the reuse of certain artifacts within the development process.

In the recent past software product line based development methodologies truly became a mainstream development technique. Although successfully applied in the software development field, product lines are still not state of the art within the automotive domain of embedded software systems. Software and E/E system product lines are build and managed often by explicitly naming and configuring all variants in advance and maintaining all single products through the complete development process.

Product lines have to be engineered within a larger system context of hardware and software systems. The proposed layered approach respects the current development artifacts and processes and introduces variability concepts only where needed. As a reference, the following artifacts depicted in Figure 1 were considered within the VEIA project. Products and their variability are expressed using feature models as they were introduced in [27, 39]. Logical architectures are described using an architecture description language that al-
Variability has to be represented in all architecture models. The relationships between the artifacts regarding variability issues stem from the idea of the \textit{variability pyramid} in [33], and accordingly reflect the \textit{sources of variation} [7] on the right level. In more detail, they have to incorporate the concept of binding times for variability [40]. In [19] the mentioned binding time problem was addressed by specifying a function net for the product line of a system functionality in vehicles (see section 4) as well as two concurring software architectures. By the use of product line metrics the two architectures are delegated to the parent component. The architectural artifacts of the VEIA reference process (cf. Figure 1) are specializations of such a general component model. Because of the automotive domain we concentrate on signal-based communication between components in functional views.

Common variability concepts for product lines, e.g. found in [27, 40, 33, 36], are dealing with \textit{optionality}, \textit{alternatives} (encapsulated by XOR variation points) and \textit{OR variability}. Parameterization is also often provided. Furthermore, \textit{dependencies} (e.g. “needs”, “requires” or other logical statements) are used to constrain possible combinations of variable elements.

In order to represent product lines on an architectural level, variability concepts need to be consistently integrated with the above mentioned architectural concepts. The integration, as sketched in the following, results in new kinds of modeling elements on different levels of granularity:

2The implementation of the demonstrator is still work in progress. All mentioned methods and analysis operations are prototypically realized within \textit{v.control}. 

In this paper we introduce the underlying concepts for the integration of variability modeling aspects into architectural models (see section 2) and a possibility to automatically compute the corresponding feature tree for formal analysis purposes (see section 3). Doing so we are able to analyze the variance of the product line with respect to architectural views and abstraction levels, to assess alternative solution strategies with respect to their variability aspects, and to evaluate them regarding development costs and efforts. We examined the presented method with the help of an example use case described in detail in section 4. The paper ends by mentioning related work (section 5) and some concluding remarks (section 6).

2. Variability concepts in architecture descriptions

\textit{Components} are the building blocks of an architecture in common ADLs. \textit{Interface descriptions} (ports) specify the ability of combining components. \textit{Connectors} establish the composition by connecting ports. The composition of components results in higher-level components again, see e.g. [29]. As an assumption for the following discussion, a higher level component is just a cluster of its subcomponents and does not add any additional behavior or hides anything. Thus, all ports not connected between the subcomponents are delegated to the parent component. The architectural artifacts of the VEIA reference process (cf. Figure 1) are specializations of such a general component model. Because of the automotive domain we concentrate on signal-based communication between components in functional views.
• Applying the optionality concept on components results in a differentiation of composition, i.e. we get a distinction in mandatory and optional subcomponents.

• The integration of XOR variability into components results in a new hierarchical relationship between components: A component variation point represents a generalization of alternative components as it exhibits their commonalities with respect to the architectural role which the alternatives are able to fulfill.3

• By the application of variability concepts on horizontal connectors, we can distinguish between mandatory and optional connectors. XOR variability (like “switches” in Koala [32]) is not supported, because this situation can be emulated by a variable component fulfilling this task, or by a component variation point.

Delegation connectors are used to relate two hierarchical levels. They cannot exhibit own variability because of our definition of the composition of components.

• The consistent integration of the variability concepts yields components with varying interfaces in the form of optional vs. mandatory ports, and in the form of ports with fixed or varying signal sets.

• Parameterization is applicable on all architectural elements, e.g. parameterized components, ports, connectors, or signals, whereby variability is supported for their attributes.

These basic concepts are not isolated, but interrelated. Our approach allows—and thus deals with—the situation when variability within a component cannot be fully encapsulated and hidden by that component. Such a situation happens when a component has optional subcomponents or subcomponents with optional ports. Consider for instance the optional output port poutDisplayEnhanced of the mandatory component CbsD disp layEnhanced in Figure 5. When this port is available in one product, the corresponding communication is delegated to the environment of the parent component Cbs. Another situation when inner variability cannot be hidden is often introduced by component variation points. Alternative components architecturally play the same role, but they are related to different requirements or solutions. This can cause a different demand on signals they send or receive. The comparison of the alternatives with respect to their ports leads to the distinction of ports which are common to all alternatives (e.g. input port pínKm of the component variation point CbsWdSparkP lugs2 in Figure 5), and variant-specific ports which are not present in all alternatives (e.g. port pínSensorsSp of the same component). The component variation point exhibits the result of this comparison.

<table>
<thead>
<tr>
<th>Origin of variability</th>
<th>Occurrence</th>
<th>Signal set</th>
<th>Kind of ports wrt. variability</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>independent</td>
<td>always</td>
<td>fixed</td>
<td>mandatory, fixed port</td>
<td>![ ]</td>
</tr>
<tr>
<td></td>
<td>varying</td>
<td>varying</td>
<td>mandatory, varying port</td>
<td>![ ]</td>
</tr>
<tr>
<td>sometimes</td>
<td>fixed</td>
<td>optional, fixed port</td>
<td>![ ]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>varying</td>
<td>optional, fixed port</td>
<td>![ ]</td>
<td></td>
</tr>
<tr>
<td>dependent</td>
<td>always</td>
<td>fixed</td>
<td>not applicable</td>
<td>![ ]</td>
</tr>
<tr>
<td></td>
<td>varying</td>
<td>dependently optional, fixed port</td>
<td>![ ]</td>
<td></td>
</tr>
<tr>
<td>sometimes</td>
<td>fixed</td>
<td>dependently optional, fixed port</td>
<td>![ ]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>varying</td>
<td>dependently optional, varying port</td>
<td>![ ]</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Variability of ports.

In general, we provide a minimal/maximal view of the communication needs of a component which has inner variability. The minimal view only comprises the common elements (the invariants). In contrast, the maximal view comprises the complete set of possible elements (i.e. invariant as well as variant-specific elements). Furthermore, it is distinguished whether the variability of the element originates from another element, i.e. if it’s independent or dependent. Dependency is established along the hierarchical structure of components, from lower to higher components. Ports have to represent this differentiation, fully characterized by the following three variability criteria:

1. **Occurrence of a port:** A port can be a mandatory or an optional “feature” of a component, i.e. a port is distinguished whether it always (part of all products) or sometimes (in some products of the product line) occurs.

2. **Signal set of a port:** The information items communicated via a port can be fixed or varying. A port can have different signal sets because of the introduction of alternative components. The alternatives could exhibit that they all need information from a specific component, but they differ in the specific signal set. In this case, the

3We do not focus on an explicit support of OR variability (i.e. selecting any number of elements of a set) in architectures, although it can be a useful construct. In case that OR variance is incorporated additional constraints on the underlying levels that describe not only the architecture but also the behavior of different development artifacts have to be incorporated. These constraints are concerned with communication or synchronization issues that are inherited from the variance modeling at the upper layer.

4A component variation point is represented by a rounded rectangle in our graphical notation, its alternatives are the direct subcomponents. Dashed rectangles mark components as optional.
Table 2. Components and their ports wrt. to variability.

<table>
<thead>
<tr>
<th>Component</th>
<th>Port occurrence</th>
<th>Port’s signal set</th>
<th>Possible kinds of ports on the component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic component</td>
<td>always fixed</td>
<td>invariant port</td>
<td>[ ]</td>
</tr>
<tr>
<td></td>
<td>sometimes fixed</td>
<td>optional, fixed</td>
<td>[ ]</td>
</tr>
<tr>
<td>Hierarchical composed component</td>
<td>always fixed</td>
<td>invariant port</td>
<td>[ ]</td>
</tr>
<tr>
<td></td>
<td>varying varying</td>
<td>dependently varying port</td>
<td>[ ]</td>
</tr>
<tr>
<td></td>
<td>sometimes fixed</td>
<td>dependently optional, fixed port</td>
<td>[ ]</td>
</tr>
<tr>
<td>Component variation point</td>
<td>always fixed</td>
<td>invariant port</td>
<td>[ ]</td>
</tr>
<tr>
<td></td>
<td>varying varying</td>
<td>port variation point</td>
<td>[ ]</td>
</tr>
<tr>
<td></td>
<td>sometimes fixed</td>
<td>optional, fixed</td>
<td>[ ]</td>
</tr>
<tr>
<td></td>
<td>varying varying</td>
<td>dependently optional, varying port</td>
<td>[ ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>optional, varying port</td>
<td>[ ]</td>
</tr>
</tbody>
</table>

The corresponding port at the component variation point is represented as a varying port.

3. **Origin of variability of a port:** The variability can originate from lower level components, thus a port can be *independent* or *dependent* with respect to its occurrence or its signal set. Reconsider the output port `poutDisplayEnhanced` of function `CbsDisplay` in Figure 5. The corresponding port of function `Cbs` is dependent. The same effect applies to a delegated mandatory port of an optional subcomponent (e.g. port `pinSensorsPf` of the optional component `CbsWdParticleFilter`). In general, there is architectural independence of ports at atomic components, because their inner variability is not represented on the architectural level.

The combination of the three variability criteria results in different kinds of ports as summarized in Table 1. Mandatory, fixed ports represent ports already known from common architectural models, where no variability concept is explicitly integrated. Because mandatory, fixed ports are always present (i.e. invariant), when its component is present, they need not be configured. Thus, there is no dependent version of them.

Along the hierarchical composition relation between components, the variability of a component is propagated to its upper level components. The possible combinations between components and ports are listed in Table 2. How variability caused by alternative components is propagated to upper levels is illustrated in Figure 2. Consider, for instance, how mandatory or optional ports of the alternative components A and B, which are encapsulated by the component variation point C, are delegated to the upper levels. The invariant port `pC1` states that the alternatives A and B share a common port with the same signal set. In contrast, the varying port `pC7` states, that both components have a common port (with respect to its architectural role), but with different signal sets.

The integration of variability concepts also introduces the need to consistently configure the variable elements occurring within an architecture description. The notions *dependent* and *independent* as above introduced with respect to ports, represent an additional specification means to classify occurring variability. The information whether a specification element is dependent can be automatically computed, thus supporting the user for example by determining the minimal set of configuration points to generate a complete configuration.\(^5\)

To which extend the described concepts are utilized during the analysis and how far the implementation of the mentioned concepts is realized within the demonstrator `v.control` of the VEIA project is sketched in the next section.

\(^5\)This feature is currently not realized within `v.control`.  

---

**Figure 2. Port dependency because of component variation points.**
3. Analysis of feature models

There are a lot of proposals in literature to formally analyze feature models. Most of them translate a feature model into a representation suitable for a formal analysis. Prominent examples are Feature Modeling Plugin (FMP) [39], pure::variants [34] or the FAMA Framework [10].

The analysis engine of FMP is based on binary decision diagrams (BDD). pure::variants uses the logical programming language Prolog to reason about feature models. These analysis techniques are used to prove certain properties of feature models that were related to the following questions or operations:

1. Does the feature model has at least one valid configuration (satisfiability) and if so, how many models (configurations) are represented by that feature model?
2. Does the feature model contain features that were never part of a valid configuration (dead feature detection)?
3. The operation to derive a (partly) configured product out of a feature model is most important during the development process and strongly connected to the binding of variance.
4. If a property such as satisfiability cannot be established, then the user should not simply get the answer “no”, but should get a helpful explanation on the basis of his feature model to be able to analyze the fault and repair it.
5. The ability to prove arbitrary properties about a feature model is concerned with nearly all the before mentioned operations and questions. It gives the user the possibility to formally examine his feature model to ensure the desired feature model behavior on a formal, objectifiable and repeatable basis.

Within the VEIA project we aim at providing a proof of concept in terms of a prototypical implementation of a feature modeling tool that is able to answer all the enumerated questions and is not limited to Horn-formulæ for example. The technique that we propose is based on the same idea as the tools mentioned above. We use a transformation approach that translates a given feature model into a propositional logic or first-order logic formula. This approach allows us to define arbitrary conditions on features that are expressible in the respective logic. These conditions represent constraints on the corresponding feature model that have to be respected if satisfiability is checked or within a configuration process. We decided to use the automatic theorem prover SPASS [30] as our reasoning engine. Such a theorem prover is able to formally analyze problems with a large number of features and it can be used to solve the constraints arising during the configuration process. Thus, all the above mentioned questions can be answered using this approach that is completely implemented within v.control.

We further expect to scale up with large feature models, since many techniques used to implement for example a constraint propagation mechanism are already successfully used as proving strategies within such theorem provers. Nevertheless, the integration into the complete development process that is concerned with different refinement levels (see Figure 1) is still not solved completely. In the following we sketch a proposal how to connect feature models with the underlying architectural models especially with respect to the configuration of feature models.

As mentioned before we propose a layered approach for the separation of concerns on the different levels. Within such a structured approach a connection between the various layers has to be established that maps the feature model to the architectural artifacts. This mapping gives us on the one hand the possibility to trace changes over the complete development process and on the other hand allows for an automatic computation of model configurations. In the following we substantiate the notion of configuration models within the help of a simple example taken from the CBS scenario described in section 4.

In our approach we use feature models and the corresponding operations defined on them as the central variability management engine. The integration of this engine into a system development process is one of the major tasks for an enterprise wide consistent and non-isolated variability management approach. To this end we use a translation process that integrates the development artifacts from different layers into one single feature model6. Within the computation of this model the variability analysis presented in section 2 is used to formulate the respective constraints on functions, ports and their connections.

The translation algorithm is based on translation rules that constitute the mapping from architectural elements to feature model elements (propositional formulae) as illustrated by the following selection of rules.7

R1: An atomic function \( F \) may have mandatory ports \( P_1, \ldots, P_n \) which are part of a configuration if and only if the function itself is part of that configuration, expressed by: \( F \leftrightarrow (P_1 \land \ldots \land P_n) \).

R2: An atomic function \( F \) may have optional ports \( P_1, \ldots, P_n \). If one of the optional ports is part of a configuration, then the function \( F \) is part of that configuration, expressed by: \( P_1 \lor \ldots \lor P_n \Rightarrow F \). Note that in this case the translation excludes configurations where ports exist with no associated function.

---

6 used as an internal computation and analysis model
7 The complete set of rules realized within v.control covers all syntactical possibilities used during the specification of a functional architecture.
R3 A hierarchical (non-atomic) function $F$ is decomposed into sub-functions $F_1, \ldots, F_n$. These sub-functions may be mandatory, i.e. if their parent is present, then the sub-functions are present, too. A sub-function can be an atomic function, a hierarchical function, or a function variation point. The corresponding translation is given by the following formula: $\forall i : 1 \leq i \leq n : F_i \Rightarrow F$

R4 As described in R3 a hierarchical function $F$ can be decomposed into sub-functions $F_1, \ldots, F_n$. These sub-functions may be optional. Thus, their presence within a configuration depends on the presence of their parent function and on the fact whether they are selected during that configuration. The simple formal translation is given by the following formula: $\forall i : 1 \leq i \leq n : F_i \Rightarrow F$

R5 A variation point is a function $F$ which encapsulates the logical XOR-relation between its sub-functions $F_1, \ldots, F_n$. If a function variation point is present then exactly one of its sub-functions is present.\(^9\) Which of the sub-functions $F_1, \ldots, F_n$ is taken depends on the configuration of the function variation point. The alternatives may again be a function variation point, or an atomic or hierarchical function. The formal translation is reflected by the following formula:

$$
(\forall i : 1 \leq i \leq n : F_i \Rightarrow F) \land (F \Rightarrow (F_1 \lor \ldots \lor F_n)) \land (\forall i,j : 1 \leq i,j \leq n : i \neq j \Rightarrow F_i \Rightarrow \neg F_j)
$$

The given rules can be applied recursively to architectural specifications resulting in a feature tree with the corresponding constraints that at first represents the variability occurring within the functional architecture. Note that this representation does not represent the functional architecture itself. It simply exploits the architecture description elements in order to unambiguously represent their variability. The given formal representation can then be used to formally reason about the variability and to prove for example whether a configuration is consistent.

To illustrate the mentioned approach let us assume we have finished the description of the feature model that represents the product line description in Figure 1 and is shown in Figure 4. Within the scenario described in section 4 we have specified the corresponding function net as illustrated in Figure 5. The function CbsWdSparkPlugs is described as a variation point introducing two alternatives. These alternatives are mirrored within the corresponding feature tree. Let us assume for illustration issues that the alternative within the feature tree between adaptive and linear computing has not been made before (so we cut the feature tree after the WdSparkPlugs node). This leads us to the problem of introducing a new variability within the function net that is not reflected within the feature tree. Although it is possible in such a situation to let the user repair this model inconsistency, we think that such an approach is error prone and it should be possible to automatically compute a new feature tree that incorporates the newly introduced variability. In our example we assume that the feature node WdSparkPlugs is mapped to the function CbsWdSparkPlugs. From this we can conclude that there is a new variability since the CbsWdSparkPlugs represents a variation point. Now the computation is easy as illustrated in Figure 3. The general idea is to represent functions as features and their sub-functions as children of these features. The ports associated to a function are collected under a feature Ports. The dependencies between ports\(^9\) are expressed using needs-Links as depicted in Figure 3. For the sake of readability Figure 3 shows only one needs link between the port pinSensorsSp of the function CbsWdSparkPlugsAdaptive and the port pinSensorsSp of the function CbsWdSparkPlugs. The connection between the mentioned ports is only a delegating connection which is also expressed by the established equivalence relation. For the underlying development artifacts, which have to be connected to the corresponding features in the feature tree, it means that both ports can be identified. The complete list of ports can in general be avoided or hidden since most of them are mandatory and not part of some variability. Besides this we suggest to introduce a general hiding concept that lets the user choose the view (level of detail) of the feature tree.

Based on this automatically computed feature tree the user is able to configure its development artifacts on each

---

\(^9\)This holds for a completely configured product.

\(^9\)occurring as delegation or horizontal connectors
4. Case study

The presented concepts are evaluated by a case study from our project partner BMW about a distributed supporting functionality in vehicles: “Condition-based service” (CBS) [19]. Hereby, necessary inspections are not any longer terminated at regular and fixed durations of time or driven distances, but on the real wearout of relevant wearing parts of the vehicle like motor oil quality, brakes, spark-plugs etc. Variability within CBS is primarily caused by different business service concepts for specific vehicle types, different infrastructures of supported vehicle types, and by organizational and development process-related aspects. It is reflected in a varying information basis for CBS, different algorithms to estimate the wearout, and different solution strategies to implement the functionality.

![Figure 4. CBS product line description by features (incl. configuration for a product).](image-url)

The definition of a (simplified) CBS product line is shown in Figure 4. This feature model was configured for a typical product: a vehicle with Otto engine, no additional displays, but where the wearout of spark plugs is detected by an adaptive computing method. The corresponding function net is shown in Figure 5. Each wearing part is represented by a function to compute the wearout. Since the product line supports two ways of computing the wearout of spark plugs, the alternative functions CbsWdSparkPlugsLinear and CbsWdSparkPlugsAdaptive are introduced. The adaptive variant needs additional input from the spark plugs sensors for the computation. This is represented by an additional, variant-specific port at the CbsWdSparkPlugsAdaptive function, which becomes a dependent port at the functional variation point CbsWdSparkPlugs as well as at the (top-level) function Cbs (port pinSensorsSp). Furthermore, the wearout detection function for spark plugs as well as for the particle filter are optional because of hardware dependence: vehicles equipped with an Otto engine have spark plugs only, and vehicles with Diesel engines have particle filters only. Both wearout detection functions could have been modeled as alternatives in the function net, but we’ve decided to model them just as options, because they do not represent “functional” alternatives. Generally, the definition of XOR variation points or the usage of options is always a design decision dependent on specific objectives and situations.

5. Related work

The work presented here is related to a number of different efforts. Various ADLs in the literature incorporate the hierarchical composition of components, cf. [29] for an overview and a taxonomy for comparison of ADLs. ADLs are also object of research for automotive or embedded systems like [11, 12, 32, 14, 5, 6].

Variability management and product line engineering is in the scope of many efforts. ADLs for product lines often introduce structural variability for components, but only deal with constant, maybe optional, interfaces for components. A comparison of those ADLs is given in [1, 36]. The notation we used for ports was inspired by [32]. In contrast to our approach, this ADL supports alternative components by alternative connections (“switches”). By “diversity interfaces” a hierarchical parameterization of components and its elements is supported.
An overview about binding times of variability and variability mechanisms is given in [40, 7, 38, 8]. General notions about commonality and differences are published in e.g. [27, 33]. Exploiting feature models for variability management is done by e.g. [39, 34], a configurator tool according to [8], uses a generic family model as abstraction of blocks of a target language description (e.g. source code). By configuration of a feature model, the family model is used to concatenate the blocks to build a product. In this way, our architectural models can be regarded as specializations of the family model. Thus, our concepts could be integrated with such tools.

6. Conclusion

The presented method introduces the possibility to connect different layers within a structured development process in a generic way resulting in an effective approach to configure and to compute products according to predefined measures. It supports the user in finding valid configurations and guarantees that the constraints are not violated. The proposal is flexible in the sense that it allows to incorporate variance that is introduced at later stages of the development without changing the before specified development artifacts. Those artifacts that are specified and defined below the level of the functional architecture can be integrated analogously resulting in a pervasive development of variability throughout the complete product development process.

Although the concepts for the handling of variability are not yet stable within the AUTOSAR considerations [2], our method provides a mean to realize a pervasive handling of variability throughout a product development process that starts with the requirements specification and ends with an AUTOSAR compliant software development [3, 4].

As proof of concept the work is prototypically implemented in a case tool which we called v.control. It demonstrates different aspects of our methodology: the specification of product lines in form of function and software component architectures including abstraction and reuse issues, the assessment of a product line by the evaluation of the architecture specification using metrics, and the configuration of a product line architecture in order to consistently derive the specifications of each of its products. For the latter we use common feature modeling techniques for a centralized and continuous variability management. The screenshots of the VEIA demonstrator v.control in Figure 6 and Figure 7 illustrate the implemented functionality with respect to the method presented in this paper. The screenshot depicted in Figure 6 shows how the linkage between features and functions is presented to the user. A simultaneous configuration of both the feature model and the connected functional architecture is shown in Figure 7.

The concepts realized within the demonstrator have to be completed in the future with respect to data management issues as for example the change management of configurations, feature models and function models. Furthermore, it is planned to allow for more flexibility within the current fixed three tier development methodology.

References

Figure 6. Screenshot of the VEIA prototype “v.control” wrt. configuration: Link viewer.

Figure 7. Screenshot of the VEIA prototype “v.control” wrt. configuration: Configuration view.


