A UML Profile for the Specification of System Architecture Variants Supporting Design Space Exploration and Optimization

Alexander Wichmann, Ralph Maschotta, Francesco Bedini, Sven Jäger, and Armin Zimmermann
Systems and Software Engineering Group
Computer Science and Automation Department
Technische Universität Ilmenau
Ilmenau, Germany
Contact: alexander.wichmann@tu-ilmenau.de; see http://www.tu-ilmenau.de/sse

Keywords: System modeling, architecture variant description, UML profile, design space

Abstract: The optimization of complex systems as well as other design methods require a description of the system parameters, or the *design space*. Explicit encoding of all possible variants is practically impossible, thus an implicit method is needed. While this is easy for purely numerical parameters and a fixed number of them as usually assumed in direct or indirect optimization, it is quite hard for systems in which the architecture and thus the structure of the parameters themselves can be varied. This paper introduces an approach to specify system architecture variants in a concise way and proposes a UML profile for this task. Standard UML meta model elements are used for the description of variant-specific stereotypes. An example of a variant specification for a communication network model is presented.

1 Introduction

Model-based systems engineering is helpful in allowing early validation of complex system designs and reduction of costly failure corrections in late development states. The underlying models have to capture all significant information, which often contains both (static) structural as well as (dynamic) behavioral aspects. Numerous design decisions must be made to obtain a hopefully close-to-optimal system. These decisions could be supported or automated by a closed-loop indirect optimization approach (van Leeuwen et al., 2014), in cases where the descriptional power of linear programming is insufficient. The set of valid system variants (or design alternatives), also called the design space (Taghavi and Pimentel, 2010), has to be specified as an input to the optimization heuristic. Such a specification (not its derivation or exploration, though) is comparably simple as long as all parameters are just numerically valued with a given interval. In fact, it is usually described by a vector of n real values for a system with n design parameters, and we may imagine the design space as an n-dimensional geometrical space then. However, this paper addresses the problem of *architectural* optimization, in which the structure of variants and design parameters is typically much more complex, and in which already the number of parameters n is not obvious as it depends on other design parameter value choices: Components may be optional and include a variable attribute. If such an optional component is not used inside a system variant, then their properties as well as the corresponding parameters are also not existing. Thus, the well-understood research area of linear systems with numerical parameters cannot be applied here.

What should be described for the design space about a variable architecture? First of all, the general structure must be defined including system components as well as their properties and relations to other components. Furthermore, variants of the system model have to be specified in order to solve the following questions: How many instances of components are existing? How many instances belong to or comprise another one? What are the limits of the numbers of instances? Which values can be assigned to component properties? Which interface realization should be used? Should an optional feature be used? Where is each component placed? What are fixed components and attributes? How should an attribute be specified, which depends on another attribute? What kind of connections should be used between components? How can attributes be captured that are only necessary for certain structural selections or options?

Apparently the design space includes numerical

parameters, structural, non-numerical parameters and parameters, which are only existing in specific cases. How can the design space be described? A possibility is to list all architecture variants explicitly. This description is practically impossible for complex systems because of the sheer number of variants, which usually scales exponentially (with the size of the cross product of all individual parameters). An implicit description is thus the only viable way to specify a design space, in which decisions and their alternatives are described.

In the existing literature on this subject, system architectures can be specified by using the family of architecture description languages (ADL). There are a variety of languages like xADL (XML-based ADL, (Dashofy et al., 2001)), Acme (Garlan et al., 2010) or μ-ADL (Oquendo, 2004). A survey of such languages is presented in (Clements, 1996). All of these languages can be used to specify a single system architecture, but they do not support techniques to specify variations or the possible design space.

Feature models (Schobbens et al., 2006; Zakál et al., 2011; Acher et al., 2014; Grönninger et al., 2014) are a description language, which is used in product lines engineering. It allows the description to a variety of products, which are based on an identical basis, while differing in features and design details. Feature models provides elements to describe *features* of a system, including possibilities to choose or ignore alternative features or define XOR-relations of features, where exactly one feature has to be chosen. However, feature models lack support for dynamic variability: All alternatives have to be described explicitly. For instance, a component can be existing between 0 and 10 times. Furthermore, each component includes a property, which can be varied. This can be done with XOR-relations, in which each component count is explicitly listed. For each component alternative, the identical attribute variants have to be described separately and thus redundantly. For small systems, this may be usable, but the model will lose clarity and expressiveness in variant specifications of complex systems. Moreover, the possibility to define physical connections between features such as two components communicating or a component including a variable count of another component are completely missing.

Another architecture description language is EAST-ADL (Association, 2013), which is a domain-specific language for the description of automotive systems. EAST-ADL includes a package which provides description elements for variability management. This language is based on the AUTOSAR (AUTOSAR, 2015) meta model, which is developed for use in automotive domain. Here, a domain-independent language is required in order to model variants of system of different domains. A broader view on architectural models in the automotive domain is given in (Broy et al., 2009).

The goal and contribution of this paper are (meta) models for the description of system architecture variants that can later be used for design space analysis, including automatic indirect optimization methods (Wichmann et al., 2015). We propose standard UML class diagrams for this goal and combine them in a UML profile (OMG, 2015) for this task. Profiles include stereotype definitions, which extend standard UML elements and are used to define domain-specific information. In this case, a profile named *variant profile* is specified, which defines several stereotypes for our task in the subsequent Section 2. Technically, the Eclipse modeling project and the Sirius project are used which enable a more effective realization of domain-specific languages than other approaches (Eclipse, 2014; El Kouhen et al., 2012).

This profile is used for the description of system architecture variants for an example in Section 3. The architecture of a sample communication network is modeled with the proposed profile, in which system components, their properties and connections to other components are described. Stereotypes of the profile are available inside system models and can be applied to UML elements in order to specify variants of the system, which results in a system architecture variants model as design space description.

Furthermore, this approach allows an easy integration with the concept of executable system optimization specification, in which the behavior of system optimization is modeled with UML activity diagrams and transformed into executable code using model-to-text generators (Wichmann et al., 2016). The generators are also applicable to system architecture variant models and create executable code, which can be used by optimization processes using an fUML execution engine (Bedini et al., 2017).

2 A UML profile for system architecture variant specification

This section describes the UML profile and its application for the implicit specification of architectural design spaces. A UML profile is an element of the UML and defined inside the UML meta model (OMG, 2015). Profiles are used to extend classes of the UML meta model with additional stereotypes, which allows a more detailed (usually domain- or appplication-specific) specification of system.

A model of a system can be structured as a *class* representing the system. This system class has associations to component classes, which have properties and may have associations to other component classes. For that, two types of structural element are identifiable: class properties and class associations.

The following three UML meta classes are extended with variant-specific stereotypes: *Property*, *Dependency* and *Model*. The following subsections describe the additional stereotypes for these meta classes in the profile.

2.1 Model Variants

The UML meta class *Model* serves as the root element of a system model and includes all elements, which describe structure and behavior of the system architecture. The profile extends *Model* with stereotype *variant*, which implies that the system model includes variants.

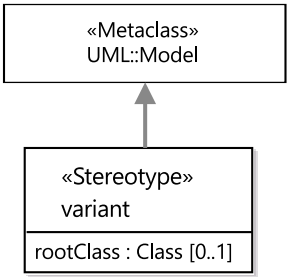

Figure 1: Profile diagram for *Model* stereotype

Variant owns the optional property *rootClass*, which provides a reference to the model class, in which the creation of a system architecture variant should start. If this property is not specified, root classes can be determined automatically by searching for classes which are not referenced by another class and thus are independent. For each found class, the variant creation is executed afterwards.

2.2 Value Variants

The second extended UML meta class is *Property*, which is used to specify properties of classes. Figure 2 presents all stereotypes, which are extending *Property*.

Regarding variant specification, properties can be classified into value-based and instance-based properties. Value-based properties are specified by primitive data types (comparable to simple linear or numerical design parameters) or enumerations and are assigned one or more fixed values. Their properties cannot be modified for variant specification. Thus,

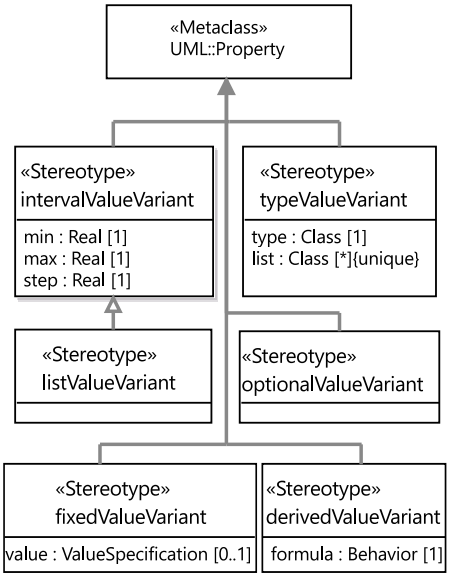

Figure 2: Profile diagram for value variants

the set of stereotypes for *Property* is called *valueVariant*. In contrast to this, instance-based properties are specified by associations and allow hierarchical variant specification of the associated class.

Property is classified into several categories: numerical attributes, optional attributes, fixed attributes or derived attributes. For each category, a separate stereotype is defined, which owns different properties to specify the variants of corresponding *Property* element.

First of all, a system architecture may have component properties, which are important for simulation and evaluation of this architecture, but should not be varied in the design space description. Such properties can apply the stereotype *fixedValueVariant* explicitly, but this is not mandatory. Properties without variant stereotype do not increase the design space and thus are automatically interpreted as fixed. Stereotype *fixedValueVariant* is thus defined as default. *FixedValueVariant* includes property *value* of type *ValueSpecification*, which allows static configuration of *Property* values for system architecture variant models. *Value* is an optional stereotype property allowing to set a value to be assigned to this property.

Defining a set of valid values is another possibility to specify value variants of a *Property* element. Each value of this set can be assigned to a class property. We propose the stereotype *typeValueVariant* for this case, which allows the definition of a value list, whose items confirm to the type of its *Property*. For that, the stereotype *typeValueVariant* owns two properties: *type* specifies the class, whose value are set to

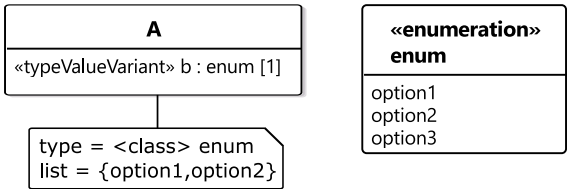

Figure 3: Example for *typeValueVariant*

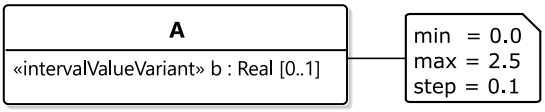

Figure 4: Example for *intervalValueVariant*

the list and thus defines the type of the list, while *list* represents a set of actual values (like an enumeration).

Figure 3 presents a simple example us of *typeValueVariant*. The class *A* has a property *b* with enumeration class *enum* as type. The enumeration includes three literals *option1*, *option2* and *option3*. Property *b* is specified with stereotype *typeValueVariant*. Thus, stereotype properties *type* and *list* should be set. The value of *type* must match with type of *b*. This can be an instance of property type or an instance of a derived class. Here, the enumeration *enum* is used. Property *list* includes all value variants, which can be assigned to *b*. Here, these value variants must be a literal of *enum*. However, *list* does not have to contain all existing literals, but only literals which are relevant for the current variant specification. Thus, *list* is filled with enumeration literals *option1* and *option2*.

Variants of numerical properties may be specified by interval definitions with minimal and maximal value as well as a step definition to specify all intermediate values. For that, stereotype *IntervalValueVariant* is introduced to describe the range of numerical values of *Property*. The range is restricted by a minimal value (*min*) and maximal value (*max*). The values between these limits are calculated using the property *step*.

An example for application of stereotype *intervalValueVariant* is presented in Figure 4. A class *A* owns a real number *b*, which should be varied. For that, the stereotype *intervalValueVariant* is applied to *b* and its properties have to be set. The property *b* should be able to take values from *0* to *2.5* with accuracy of one decimal digit. Thus, the property *min* is set to *0.0* and *max* to *2.5*. The accuracy is specified with property *step*, which is set to *0.1*. Thus, the values *0.0*, *0.1*, ..., *2.4*, *2.5* can be assigned to property *b*.

The length of list properties can be assumed as s special case of numerical attributes. Instead of assigning a value to a *Property*, a value can represent the number of elements inside the *Property*. However, the changed interpretation of variant specification requires the additional stereotype *listValueVariant*, which is derived from *intervalValueVariant* and inherits its properties. This stereotype only specifies the count of values inside a *Property* element, but does not have information about the value specifications. This has to be done by another stereotype.

Furthermore, a system component may have optional features, which are represented by Boolean values specified in *Property* and thus can assume either the value *true* or *false*. Exactly such a variant specification is realized with stereotype *optionalValueVariant*. Properties for *optionalValueVariant* are not specified, because further information for this stereotype is not necessary.

System components may have properties, which depend on other properties and can be calculated explicitly based on the value of such properties. Such variants are specified with the stereotype derivedValueVariant (to simplify their later calculation, which otherwise could be done with constraints in a much less efficient generate-and-test algorithm). Its property *formula* describes the calculation function. *Formula* is a *Behavior*, which is the basic meta class for all behavioral (executable) elements of the UML. Thus, the *formula* can be described by using UML diagrams like *Activity Diagram* or *Sequence Diagram*. Furthermore, *OpaqueBehavior* or *FunctionalBehavior* can be used to specify the behavior with code fragments of programming languages or expressions of OMG's Meta Object Facility Model to Text Transformation Language (MOFM2T (OMG, 2008)). The specified behavior can be executed and the result is assigned to the corresponding *Property* afterwards. The actual behavior for *formula* has to be specified during the variants modeling.

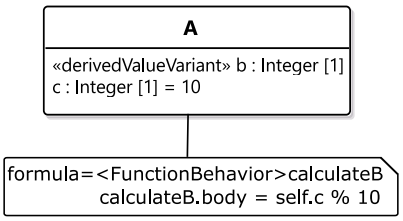

Figure 5: Example for *derivedValueVariant*

Figure 5 shows an example class *A* with property *b* and *c*. The value of *b* should be calculated based on the value of *c*. For that, the stereotype *derivedValueVariant* is applied to *b*. Its calculation formula is specified inside the stereotype property *formula*, which can be described using MOFM2T expressions for instance. All instances and values of the current system architecture variant can be used for the calcu-

lation. Here, MOFM2T expression is specified inside a *FunctionBehavior* and calculates the modulo value of *c*. The result of MOFM2T expression processing is assigned to *b*.

These stereotypes suffice to describe all possibilities for specifying variants of properties that we have encountered so far in our analysis. The more complex description of instance variants is covered next.

2.3 Instance Variants

The UML meta class *Dependency* is extended with variant stereotypes in order to vary instance-based properties, which are specified by associations to other classes. A *Dependency* relation defines that a class depends on a single supplier class or set of supplier classes (OMG, 2015). Figure 6 shows the introduced stereotypes, which extend the *Dependency* class. These stereotypes define, how many instances of the supplier class should be created and how their properties have to be set.

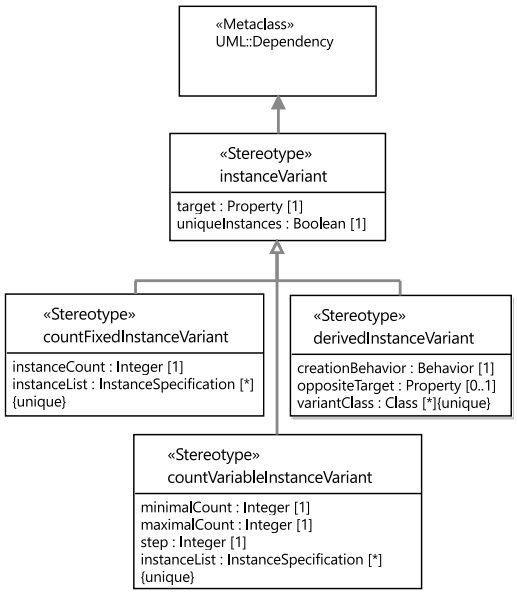

Figure 6: Profile diagram for instance variants

In general, variant specification of instance-based properties defines, that instances of a supplier class should be assigned to a property of the depending class. How many instances should be created and how these instances are configured, should be defined through specializations.

The meta class *Dependency* is extended by the stereotype *instanceVariant*, which implies, that a dependent class includes instances of the supplier class. These instances are assigned to the property of the dependent class, with is configured in stereotype property *target*. The second *Property* of *instanceVariant* is called *uniqueInstances* and implies that the instances are unique w.r.t. their values.

Further stereotypes are derived from this base stereotype and specify the number of instances and their property settings. The properties of *instanceVariant* are inherited and thus configurable by derived stereotypes.

The number of instances can be fixed or variable for variant specification. In the fixed case, a predefined number of supplier class instances has to be assigned to a property of the depending class. Furthermore, these fixed instances have attributes, which may be fixed or variable. For that, the derived stereotype *countFixedInstanceVariant* is introduced. It implies that a fixed number of instances of the supplier class should be created and assigned to the inherited *Property target*. The count of instances is specified with *Property instanceCount*. A created instance has properties, to which a value has to be assigned. These values can be determined by a fixed or variable specification. In order to cover all combinations of variant specifications, three possibilities including value setting behavior are defined, and priorities assigned to them as follows.

The preferred (high priority) option is the configuration of fixed instance specifications inside the stereotype *countFixedInstanceVariant*. For that, the optional *Property instanceList* is defined, which is a set of *InstanceSpecification* elements, which includes slots for values specifications of class properties. It may be that an *InstanceSpecification* is incomplete, because at least one value of an attribute is not preconfigured. In this case, the second option is checked, in which the value should be determined based on stereotype specifications. If the property does not apply a stereotype, specified default values of the property should be used. If none of the options is applicable, the property value stays undefined.

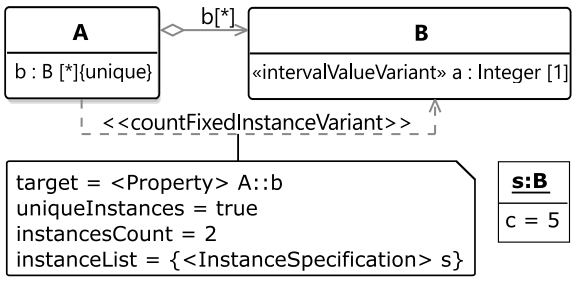

Figure 7: Example for *countFixedInstanceVariant*

An example is shown in Figure 7: Two classes *A* and *B* are connected through a composite association, in which class *A* includes instances of class *B*. Additionally, class *B* has a numeric property *c*. The *variant*

profile should be used to specify that *A* owns two instances of *B*, in which one instance has a fixed value *5* for *c*. The second instance's attribute can be varied between 0 and 10, but has to be different from the first instance's value. To specify that, a *Dependency* connection is created between target class *A* and supplier class *B*, which applies the stereotype *countFixedInstanceVariant*. The stereotype property *target* specifies the property of the target class, to which the created instances should be assigned. Here, the instances are assigned to the property *b* of class *A*. *UniqueInstances* is set to *true*, because the instances inside *b* should be different. Class *A* should own two instances of *B*, thus *instanesCount* is set to *2*. The fixed instance is specified with *InstanceSpecification s*, which assign the value *5* to property *c*. This *InstanceSpecification* is assigned to stereotype property *list*. The stereotype *intervalValueVariant* is applied to property *c* in order to specify the variants of class *B*.

According to the specified value-setting behavior, an instance of class *A* is created and its properties are configured based on the applied stereotypes. Two instances of class *B* are created afterwards. The preconfigured instance specifications of class *B* are used first. The second instance is not specified by *instanceList*. Thus, the remaining instances are created based on variant specification of class *B*, in which the value of property *c* may be set to *9*, for instance.

In contrast to *countFixedInstanceVariant*, the stereotype *countVariableInstanceVariant* is used for variable quantities of class instances. Variants of instance quantities are restricted by the stereotype properties *minimalCount* and *maximalCount*. The values between these limits are calculated based on *steps*. The *Property* values of each class instance are calculated in the same way as for *countFixedInstanceVariant*.

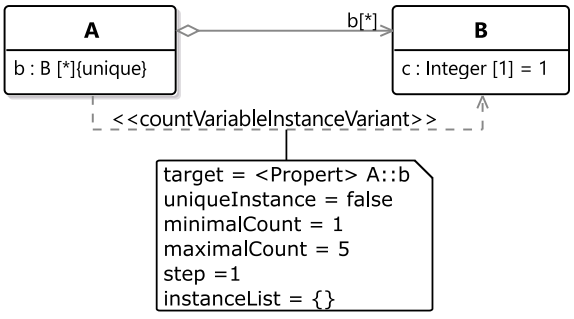

Figure 8: Example for *countVaribleInstanceVariant*

Figure 8 presents an example for the application of stereotype *countVaribleInstanceVariant* to the same system as in Figure 7. Class *A* should include one to five instances of class *B*. Instances of class *B* are not predefined, but the property of *B* is fixed to *1*. Thus, a *Dependency* connection with stereotype *countVariableInstanceVariant* is created between classes *A* and *B*. The stereotype *target* is set again to *A::b*. Class *B* should not be varied; thus all instances have the same value for *c* and the property *uniqueInstances* must be set to *false*. At least one instance of *B* has to exist, but not more than five of them together. For this, *minimalCount* is set to *1*, *maximalCount* to *5*, and *step* to *1*. The list of preconfigured instances remains empty, because instance specifications are not used here.

Creation of class instances may depend on already existing instances of other classes. This can be the creation of a communication connection between two component classes, for instance. If a communication link can be created, it may depend on properties of the connected components (modeled as class instances), which cannot be evaluated during variant specification. This has to be done at run time of the later variant creation. The required behavior to create an instance of the supplier class has to be defined inside the system architecture variants model.

For that, the stereotype *derivedInstanceVariant* has to be applied. *DerivedInstanceVariant* implies that the supplier class is created based on information from the depending class. How the supplier instances should be created, is specified in stereotype property *creationBehavior*, which is a *Behavior*. *CreationBehavior* can be specified by any UML behavioral element like *Activity Diagram* or *FunctionBehavior*. The specified behavior is executed for each unique combination of instance specifications of the dependent class.

For the special case of bidirectional associations between classes, the optional *Property oppositeTarget* is introduced, which specifies a property of the supplier class. It expresses that the bidirectional association should be created, in which instances of a depending class should be assigned to the property *oppositeTarget* and the created supplier class instance should be assigned to the target class property *target*. Furthermore, *oppositeTarget* allows the restriction of instance combinations as input for the behavior. For instance, a *Dependency* is linked to a class with an opposite target-property, which required exactly two instances of the class. Thus, only combinations with two different class instances have to be investigated.

A *derivedInstanceVariant*-applied *Dependency* is presented in Figure 9. Class *A* and *B* are associated bidirectionally, in which class *B* depends on class *A*. Class *A* owns several instances of *B*, but class *B* includes only one instance of *A*. Thus, the *Dependency* is created from *A* to *B*. The stereotype *derivedInstanceVariant* is applied. The property *target* is set

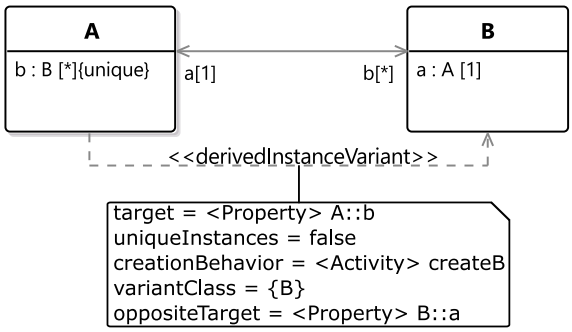

Figure 9: Example for *derivedInstanceVariant*

to *A::b* and *oppositeTarget* to *B::a* in order to realize the bidirectional association. The *creationBehavior* is specified by *Activity createB*, which creates an instance of *B*. The property *variantClass* defines possible classes, which could be created during the optimization process. Here, only instances of class *B* can be created and thus, *B* is assigned to *variantClass*.

These stereotypes form the *variant profile* and allow the design space specification of system architectures.

3 An Application Example

The presented profile for modeling system architecture variants from Section 2 is applied to a communication network, where nodes shall send and receive data using certain communication protocols.

The class diagram of the system design is presented in Figure 10. The communication network is represented by the class *Network*. Network nodes are modeled as interfaces to provide basic properties, which are necessary for all nodes. A network node has a name and a position in three-dimensional space as well as nonrecurring and ongoing costs. Communication with other nodes is realized over a *connection* interface, which is implemented by an Ethernet or WLAN connection and can transfer data according to its *Property dataRate*. Network nodes may provide Ethernet slots or WLAN technique to be able to communicate. The connection realization *WLANConnection* has additional properties to specify the maximal communication range and stores the actual distance between two network nodes. Each connection describes communication between two *NetworkNode* objects.

Network nodes can have more than one connection to different nodes depending on their Ethernet slots and WLAN features. Connections between two nodes are limited to one. The *NetworkNode* interface is realized by classes *EndNode* and *AccessPoint*.

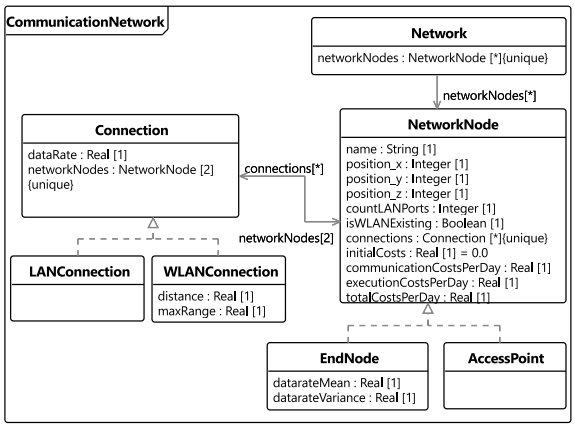

Figure 10: Design of communication network

node1:EndNode	node2:EndNode
name = "smart phone"	name = "computer"
isWLANExisting = true	countLANPorts = false
postion_x = 250	postion_x = 650
position_y = 50	position_y = 450
position_z = 100	position_z = 100

Figure 11: Fixed instance specifications of communication network model

EndNodes represent machines like server, personal computer or smart phones. They produce data with a Gaussian distributed data rate and send this to connected nodes, while also receiving data. *AccessPoints* serve as transmission nodes and transfer received data to target nodes or another *AccessPoint*.

The design space of this system should be described in order to be used by a system architecture optimization process. The presented variants profile is assigned to the communication network model in order to define which elements of this model can be varied.

Figure 12 presents the resulting architecture variant model of the communication network. First of all, the model *CommunicationNetwork* applies stereotype *variant*. Its property *rootClass* is set to *Class CommunicationNetwork*, because creation of an architecture variant should start with this class.

Dependency connections are defined and specified with stereotypes afterwards. *EndNodes* should not be varied. The example considers one personal computer and one smart phone. The personal computer has one Ethernet port but no WLAN option, while the smart phone provides WLAN connections, but has no Ethernet ports. Both are placed in different positions. For that, a *Dependency* is created from *Network* to *EndNode* and stereotype *countFixedInstanceVariant* is assigned to this connection. The specification of this two *EndNode* instances is presented in Figure 11.

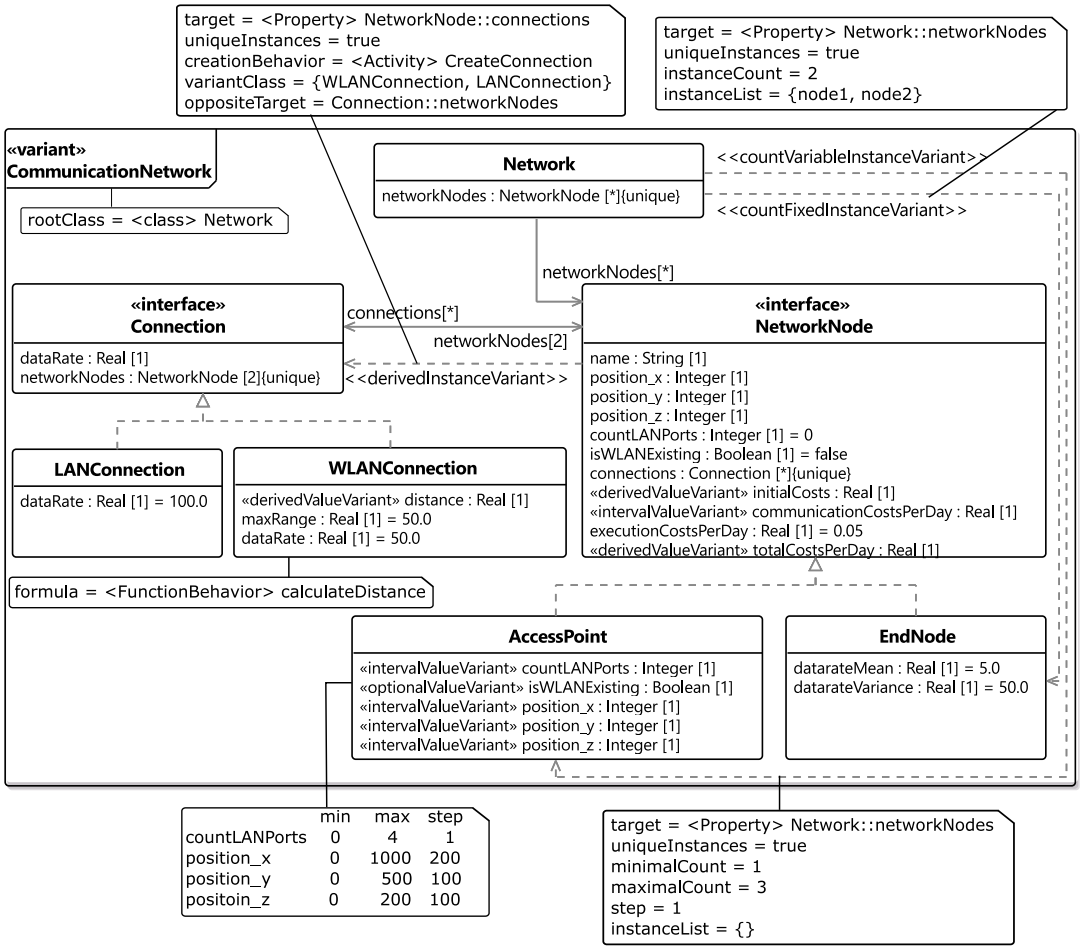

Figure 12: Architecture variants model of communication network

*InstanceSpecification*s are used to define properties of existing nodes. They are assigned to stereotype property *instanceList*. Furthermore, *objectCount* is set to value *2*, all instances should be unique. *Network::networkNodes* is specified as *target*.

AccessPoints are variable in quantity and property configuration. They are used to implement communication between computer and smart phone. For that, *Network* and *AccessPoint* are connected by a *Dependency* with applied stereotype *countVariableInstanceVariant*. *Network* includes at least zero and not more than three *AccessPoints* objects. A *instanceList* is not set. Thus, values of instances are set based on value variant specification or default value of properties.

A *Dependency* with applied stereotype *includeDerivedObjects* is created between interfaces *NetworkNode* and *Connection*. References of this connection are stored inside *NetworkNode::connections* and references of nodes should be deposited in *Connection::networkNode*. A connection can be realized by classes *WLANConnection* and *LANConnection*. The property *variantClass* is specified with both classes. Thus, an optimization process can decide which should be created, if both are possible.

Furthermore, stereotype *includeDerivedObjects* requires a specification of a *Behavior* element, which should be processed to create an instance of *Connection* class. For that, the *Activity CreateConnection* is specified. Figure 13 presents the corresponding Activity Diagram. It has three ingoing and one outgoing *Activity Parameter*s. The two ingoings are used for *NetworkNode* instances, because the interface *Connection* owns *Property networkNode*, which should exactly include two *NetworkNode* instances. The third parameter is called *Class* and allows to influence this activity by a calling optimization process.

The activity *CreateConnection* checks which connections are possible between two ingoing *NetworkNode* instances. A *LANConnection* can be created, if both *NetworkNode* instances have free Ethernet ports. A prerequisite for *WLANConnections* is the support of the WLAN feature by both *NetworkNode* instances.

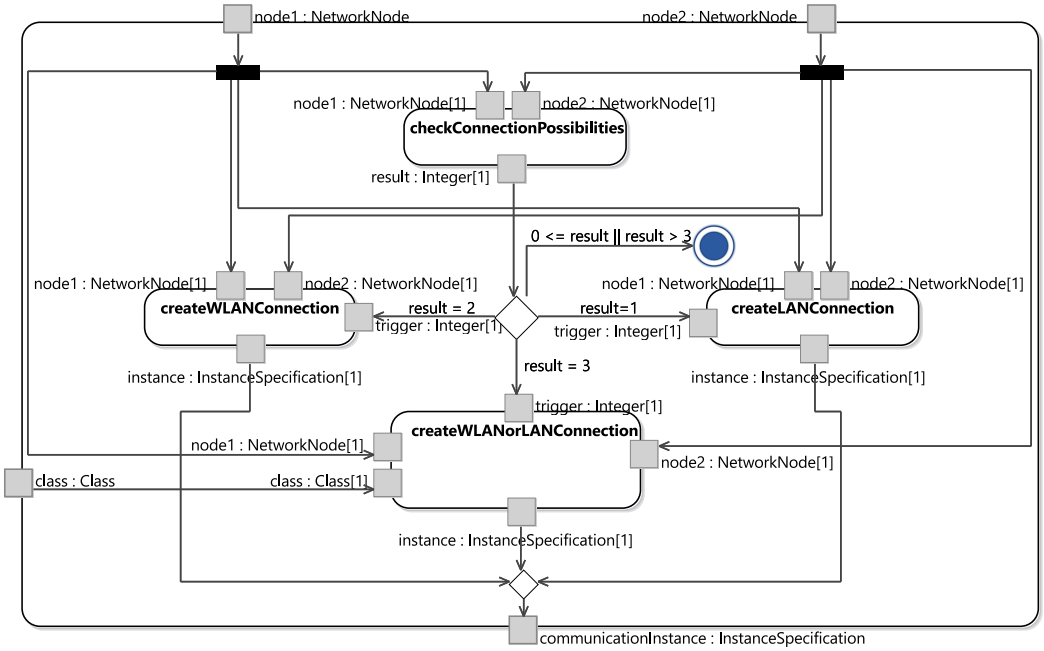

Figure 13: Activity diagram for connection creation behavior

If exactly one check passes, then an instance of the corresponding class is created. The class specified by *Activity Parameter Class* is instantiated, if both checks are successful. Otherwise, no connection instance is created. The activity ends with an *InstanceSpecification* of a *Connection* or without a result, if no connection is creatable. This behavior is executed for all combinations with two different *NetworkNode* instances.

Value variant stereotypes are applied to properties of interface *NetworkNode* as well as of classes *AccessPoint* and *WLANConnection*. *InitialCosts*, *communicationCostPerDay* and *totalCostsPerDay* of interface *NetworkNode* are derived properties and apply stereotype *derived*. For each *Property*, a formula is specified with MOF Model to Text Transformation Language expressions. The formula for *initialCosts* depends on the quantity of Ethernet ports and enabled WLAN features. Daily communication costs are calculated based on the enabled WLAN feature. Furthermore, the sum of fixed daily execution costs and daily communication costs results in total daily costs. Another derived *Property* exists in *WLANConnection*. *Distance* calculates the distance between two *NetworkNode* instances, which are set in *networkNodes*.

The class *AccessPoint* inherits all properties of the interface *NetworkNode*. The following properties are overwritten and stereotypes are applied to them: *Property countLANPorts* applies the stereotype *intervalValueVariant*. An *AccessPoint* can have no Ethernet port up to four Ethernet ports. Accordingly, the stereotype properties are set. The *optionalValueVariant* stereotype is applied to *isWLANExisting* indicating if an *AccessPoint* can provide WLAN connections or not. The position attributes of *AccessPoint* applies the *intervalValueVariant* stereotype and the valid coordinates are specified.

Finally, all value-based properties without variant stereotypes are assigned a default value. This completes the system architecture variants model, and an optimization process could use this model of the design space to find an optimal system architecture.

4 Conclusion

This paper presented an approach for the model-based specification of system architecture variants, thus allowing the concise specification of the complex design space of a system with architectural variations. A UML profile is introduced for this task, which extends standard UML meta model elements with variant-specific stereotypes. This allows variant specifications of system architectures, which are necessary to execute system architecture optimizations automatically or to apply other methods which require an implicit design space description.

Future steps include the specification and imple-

mentation of a generator, which creates architectures based on the architecture variants model affected by decisions made during an optimization heuristic execution.

Acknowledgment

The work has been supported by the Federal Ministry of Economic Affairs and Energy of Germany under grant number 20K1306D.

REFERENCES

Acher, M., Cleve, A., Collet, P., Merle, P., Duchien, L., and Lahire, P. (2014). Extraction and evolution of architectural variability models in plugin-based systems. *Software & Systems Modeling*, 13(4):1367–1394.

Association, E.-A. (2013). EAST-ADL domain model specification version V2.1.12. Technical report, EAST-ADL Association.

AUTOSAR (2015). AUTOSAR specification. Online. Release 4.2.

Bedini, F., Maschotta, R., Wichmann, A., Jäger, S., and Zimmermann, A. (2017). A model-driven C++-fUML execution engine. submitted.

Broy, M., Gleirscher, M., Merenda, S., Wild, D., Kluge, P., and Krenzer, W. (2009). Toward a Holistic and Standardized Automotive Architecture Description. *COMPUTER*, 42(12):98–101.

Clements, P. C. (1996). A Survey of Architecture Description Languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, IWSSD '96, pages 16–, Washington, DC, USA. IEEE Computer Society.

Dashofy, E., van der Hoek, A., and Taylor, R. (2001). A highly-extensible, XML-based architecture description language. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 103–112.

Eclipse (2014). Sirius. http://www.eclipse.org/sirius/.

El Kouhen, A., Dumoulin, C., Gerard, S., and Boulet, P. (2012). Evaluation of modeling tools adaptation. Available: https://hal.archives-ouvertes.fr/hal-00706701.

Garlan, D., Monroe, R., and Wile, D. (2010). Acme: An architecture description interchange language. In *CASCON First Decade High Impact Papers*, CASCON '10, pages 159–173, Riverton, NJ, USA. IBM Corp.

Grönninger, H., Krahn, H., Pinkernell, C., and Rumpe, B. (2014). Modeling variants of automotive systems using views. In *Tagungsband Modellierungs-Workshop MBEFF: Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen*.

OMG (2008). MOF model to text transformation language 1.0. Technical report, Object Management Group.

OMG (2015). Unified modeling language (OMG UML), version 2.5. Technical report, Object Management Group.

Oquendo, F. (2004). μADL: An architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *SIGSOFT Softw. Eng. Notes*, 29(3):1–14.

Schobbens, P.-Y., Heymans, P., and Trigaux, J.-C. (2006). Feature diagrams: A survey and a formal semantics. In *14th IEEE International Requirements Engineering Conference (RE'06)*, pages 139–148. IEEE.

Taghavi, T. and Pimentel, A. (2010). Visualization of multi-objective design space exploration for embedded systems. In *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, pages 11–20.

van Leeuwen, C., de Gier, J., de Filho, J. O., and Papp, Z. (2014). Model-based architecture optimization for self-adaptive networked signal processing systems. In *SASO 2014 - 8th IEEE International Conference on Self-Adaptive and Self-Organizing Systems*.

Wichmann, A., Jäger, S., Jungebloud, T., Maschotta, R., and Zimmermann, A. (2015). System architecture optimization with runtime reconfiguration of simulation models. In *IEEE International Systems Conference (SYSCON15)*.

Wichmann, A., Jäger, S., Jungebloud, T., Maschotta, R., and Zimmermann, A. (2016). Specification and execution of system otimization processes with UML activity diagrams. In *IEEE International Systems Conference (SYSCON16)*.

Zakál, D., Lengyel, L., and Charaf, H. (2011). Software Product Lines-based development. In *Applied Machine Intelligence and Informatics (SAMI), 2011 IEEE 9th International Symposium on*, pages 79–81.