DERIVING ARCHITECTURE DESIGN VARIANTS FOR SYSTEM OPTIMIZATION FROM DESIGN SPACE DESCRIPTIONS EXPRESSED USING A UML PROFILE

Alexander Wichmann
Francesco Bedini
Ralph Maschotta
Armin Zimmermann

Technische Universität Ilmenau
Department of Computer Science and Automation
System and Software Engineering Group
PO-Box 100 565, 98684 Ilmenau, Germany
Alexander.Wichmann@tu-ilmenau.de

ABSTRACT

In complex (dynamic) systems, models are usually too complex for a direct evaluation, and simulation is the method of choice (indirect optimization). Another aspect is the structure of the design space for complex system. In a recent paper of the authors, an approach is presented for design space description with a UML profile-based description of architectural variations of complex dynamic systems. In order to execute a simulation of a system variant, one has to be chosen based on the used heuristic, and specified in a standardized way to be used for constructing the actual simulation model with the use of a library of template models. This paper approaches a method to automatically generated individual UML object models from the design space specification for a given parameter selection.

Keywords: UML profile, architecture design space description, system optimization, design variants

1 INTRODUCTION

Model-based systems design is an important help in the design process of complex systems and aims at reduced risks and better design decisions without the need to implement costly prototypes. In the end, each engineering task can be viewed as an optimization problem of finding the best design alternative under given constraints. However, automatic optimization methods can often not be used because of the complexity of the problem field. Optimization of linear systems with numerical parameters is a well-understood area of operations research. However, in complex (dynamic) systems, models are usually too complex for a direct evaluation, and simulation is the method of choice (indirect optimization) (van Leeuwen et al. 2014). In order to find the optimal system architectures, heuristic techniques can be used, which is an important and widely covered research area (Liberti and Maculan 2006, Fu 1994, Carson and Maria 1997).

Another aspect is the structure of the design space — as long as it can be described as a multidimensional space with continuous or discrete numerical values, well-known heuristics such as simulated annealing can be applied. This is not sufficient any more in all cases where the architecture of a system or choice of used technology should be decided in an optimal way. Selecting a certain design parameter value (which decides about using a specific communication technology, for instance) may lead to additional parameters that only

SpringSim-Mod4Sim 2017, April 23-26, Virginia Beach, VA, USA

emerge because of this choice. In such less structured settings, it is already unclear how to describe the design space itself without enumerating the set of all variants (which is usually prohibitively large).

A solution for this problem has been proposed recently, with a UML profile-based description of architecture variants of complex dynamic systems (Wichmann et al. 2017), which is depicted in Figure 1. This meta model allows to describe multi-dimensional design spaces which may change their inner structure based on certain parameter settings, among others. 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. Variants of system component properties can be specified by value variant stereotypes, which extends the UML meta class *Property*. Properties can be classified into numerical properties, optional properties, enumeration-based 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. In contrast to this, instance-based properties are specified by associations and allow hierarchical variant specification of the associated class. To model such hierarchical relations between classes, 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). 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. A detailed specification of this meta model extension can be found in (Wichmann et al. 2017).

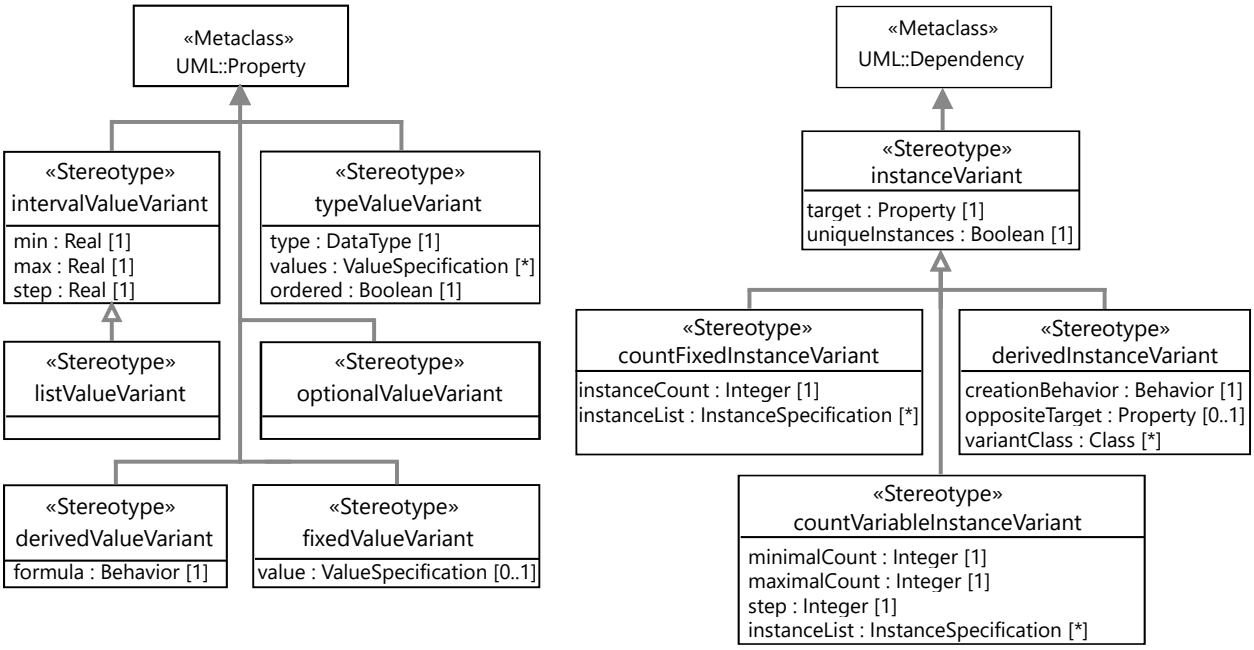

Figure 1: UML Profile diagram for System Architecture Variant Specification (Wichmann et al. 2017).

An approach of automatic indirect optimization method is already presented in (Wichmann et al. 2015), where possible system architecture variants are determined by heuristic optimization methods and evaluated by simulating the system model iteratively. There are several possible approaches to implement an optimization heuristic: One is the classic implementation by using a standard programming language. In our previous work, this has already been done in the programming language C++. To model an optimization process completely, both structure and behavior have to be described. The Eclipse Modeling Project (EMP) can be used for model-based development of domain-specific applications. In (Giese et al. 2009) a special

Story-Diagram, which is an enhancement on an activity diagram, is used to model the behavior of UML class diagrams. The Object Management Group (OMG) defines the fUML (Semantics Of A Foundational Subset For Executable UML Models, (OMG 2013b)) to realize models with executable behavior. The authors of (Lazar et al. 2010) present a special action language for fUML activity diagrams. In another related work, the Action Language for Foundational UML (ALF (OMG 2013a)) is used to describe the behavior inside fUML models of cyber-physical systems (Gerlinger Romero et al. 2013).

An indirect optimization approach is realized using an approach of model-based specification of executable system optimization processes based on activity diagrams in our previous work (Wichmann et al. 2016). UML class diagrams and activity diagrams (OMG 2013b) are used to model structure and behavior of optimization processes of a system, which should be integrated into C++-based applications. A C++ representation of the models are generated based on these models using a UML4CPP generator (Jäger et al. 2016). These classes can be used to execute the defined optimization process by using a C++ fUML-conform execution engine, which is defined in a model-based way and automatically generated as well (Bedini et al. 2017) . (Generators and execution engine are available at sse.tu-ilmenau.de/mde4cpp).

In order to execute a simulation of a system variant, one has to be chosen based on the used heuristic, and specified in a standardized way to be used for constructing the actual simulation model. An open question is now how to describe one selected variant in this way and how to interface the usually numerical parameter descriptions of heuristics with such a less structured variant description.

This paper presents a method to automatically generate individual UML object models from the design space specification. 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, Amine and Dumoulin, Cedric and Gerard, Sébastien and Boulet, Pierre 2012). The paper is structured as follows: The subsequent section specifies how to create different architecture variants depending on heuristic decisions. Section 3 presents an architecture variants model of a communication system and its variation creation as an example.

2 ARCHITECTURE VARIANT CREATION FOR HEURISTIC OPTIMIZATION METHODS

This section describes the approach of generating individual UML object models from a design space description in support of system optimization.

2.1 Workflow for System Architecture Optimization

Figure 2 presents a workflow of system architecture optimization. To optimize a system architecture, the system design has to be modeled first. System components, their properties as well as their connections to other components are specified here. This is done using the widely accepted and standardized UML (OMG 2015) and can be visualized with UML class diagrams. The system architecture optimization process requires information about how such a system design model can be varied in order to select an optimal system architecture among these variants. For that, the previously introduced *variant profile* is applied to the system design model. Variant-specific stereotypes are available inside the model for this purpose and can be used to describe value variants as well as instance variants. The resulting model is called architecture variants model (Wichmann et al. 2017).

An architecture variants model describes the design space of all possible architecture variants, which has to be given by the system designer and is used as an input for the system optimization process. During execution of system architecture optimization, a heuristic is executed iteratively and creates and evaluates

[1] Generators and execution engine are available at the MDE4CPP project home page at sse.tu-ilmenau.de/mde4cpp .

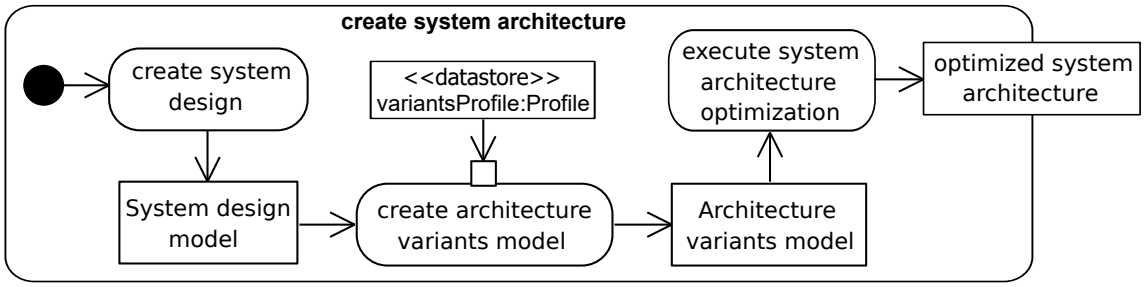

Figure 2: Main Steps in Architecture Optimization.

architecture variants in order to find the best system architecture. The approach of creation system architecture variants is specified in Section 2.4.

In our UML setting, system architecture variants are instances of the architecture variants model (Wichmann et al. 2017). The OMG defines the class *InstanceSpecification* inside the UML specification to describe instances of a modeled system. In general, *InstanceSpecification* represents an instance of a UML *Classifier* like *Class* or *Interface*. *InstanceSpecification* includes the property *classifier*, defining which *Classifier* is represented. Each *Property* of a *Classifier* is assigned an explicit value by using a UML *ValueSpecification* interface. *ValueSpecification* is used to assign an explicit value to a specific property. Values of primitive types are configured using *LiteralSpecification* and its specializations. Enumeration and instances of associated classifiers are described by *InstanceValue*, which includes references to corresponding *InstanceSpecification* elements. Details for the specification of each element can be found in the UML specification *OMG2013b*.

However, *InstanceSpecification* is used for the description of an architecture variant. As a standardized element of UML, it is suitable as an interface between heuristic and architecture generator. The task of an architecture generator is to generate a simulation model of a current architecture variant. Thus, this generator requires knowledge about the used simulation tool in order to build valid simulation models. The heuristic is independent of simulation-specific information by using *InstanceSpecification* as the interface, and can be used for several simulation tools without additional effort. Furthermore, the use of *InstanceSpecification* as interface allows easy exchange of architecture generators as well as the used heuristics.

2.2 Top-level Behavior of Heuristic Execution

The action *execute system architecture optimization* of Figure 2 realizes an approach of indirect optimization (Wichmann et al. 2016) in model-based way. This approach uses a heuristic to create architecture variants. The general top-level behavior of a heuristic is described by an activity diagram, which is shown in Figure 3. Three ingoing activity parameter nodes and one outgoing activity parameter are used. The ingoing nodes for termination condition and architecture variants model are placed on top of the activity. This data is not changed during the whole optimization process, but a token has to be put on this node (given by *DataStoreNode*) in order to fulfill UML-conform activity execution behavior. The third ingoing parameter node provides the result of the last loop execution (or invalid result at first execution). An architecture variant is placed on the outgoing parameter node. This variant can be a newly created one, which should be evaluated, or the best found architecture variant, if the overall optimization is finished.

If the heuristic is executed for the first time, an architecture variant is created randomly. How this task is done, is described in Section 2.4. Otherwise, the current evaluation result is compared to the result of the current best variant using an objective function that should be maximized or minimized. A heuristic-specific termination condition is checked afterwards. If this condition is fulfilled, the algorithm is finished and the

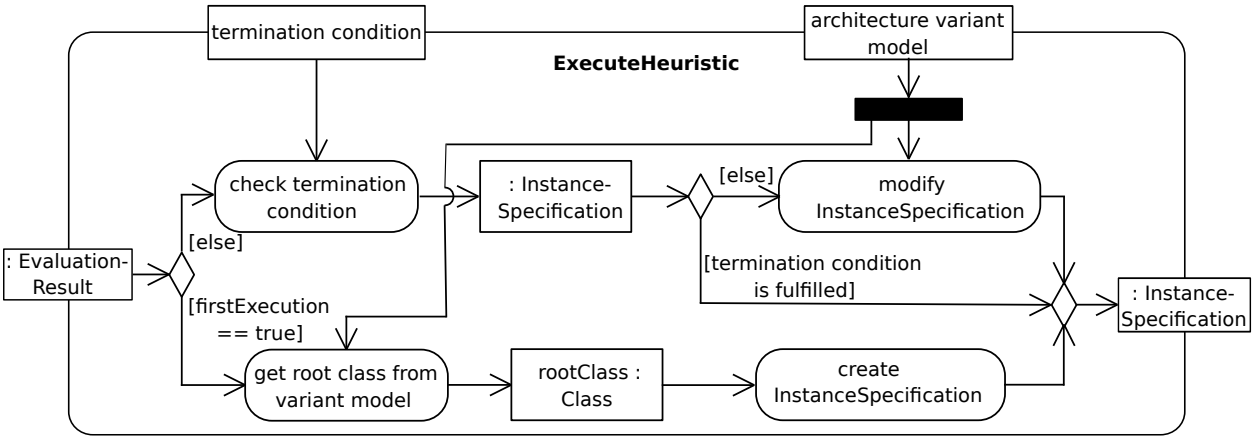

Figure 3: Top level behavior of heuristic.

resulting best variant is placed on the outgoing activity parameter node. Otherwise, the architecture variant is modified, which is described in Section 2.5, and the iteration loop starts over.

2.3 Software Design of Architecture Variant Creation

This section describes the class structure of the architecture variant creation during the presented optimization loop execution. Figure 4 presents the corresponding class diagram of this approach. Architecture variants are calculated by a heuristic, which implements the interface *XHeuristic*. The interface provides the function *execute*, which is executed inside the optimization loop and implemented by class *BaseHeuristic*. This class provides basic functionality for checking termination conditions, comparison of the current architecture variant against previous and best variants, as well as creating the next architecture variant. This class should be inherited from and thus specialized by example heuristics such as simulated annealing, which will use the provided functionality and add their specific behavior or overwrite the defaults of *BaseHeuristic*.

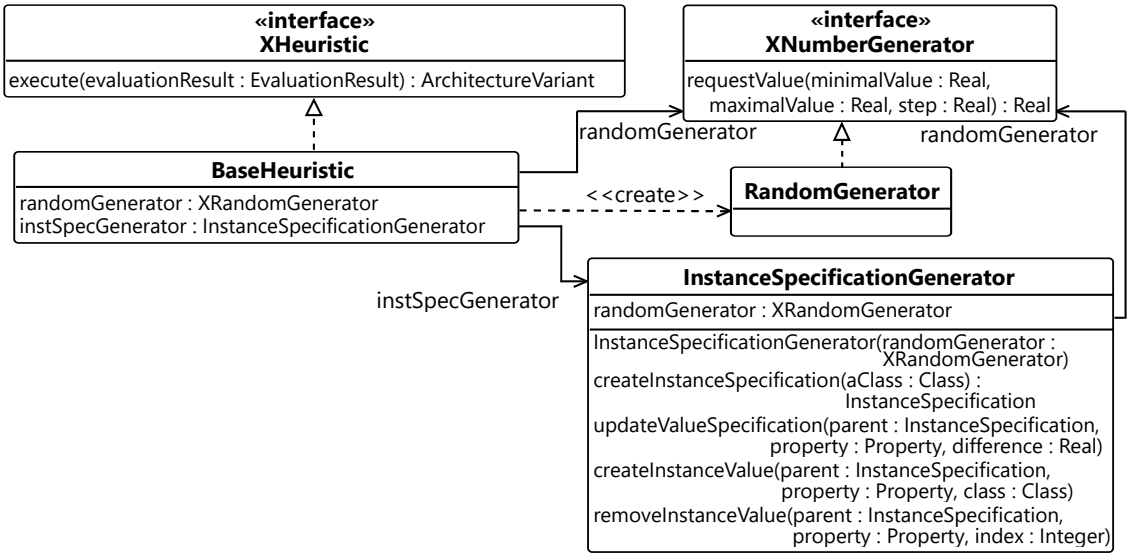

Figure 4: Class structure of architecture variant creation approach.

BaseHeuristic owns a reference to an *XNumberGenerator* interface, which is used to choose a number based on a given interval. The interface is realized by a random generator with uniform distribution. Alternatively, a random number generator with Gaussian distribution or even deterministic generator can be used for instance. The heuristic should specify its required type of distribution. Furthermore, there is an *InstanceSpecificationGenerator*, whose instance is assigned to *BaseHeuristic*. Additionally, *InstanceSpecificationGenerator* includes a reference to an *XNumberGenerator*, which is handed over in the constructor by the heuristic. In addition to the creation of UML conform *InstanceSpecification* instances, the task of *InstanceSpecificationGenerator* is to manipulate existing *InstanceSpecification* instances according to inputs of the heuristic.

2.4 Creation of Architecture Variants

This section specifies the behavior to create an individual architecture variant. The architecture variants model may specify a root class, at which the variant creation should start. If such a root class is not specified, the variants model is searched for a class, which is not owned by another class. If such a class is found, it is used as root class. Otherwise, the algorithm cannot be executed and the optimization fails.

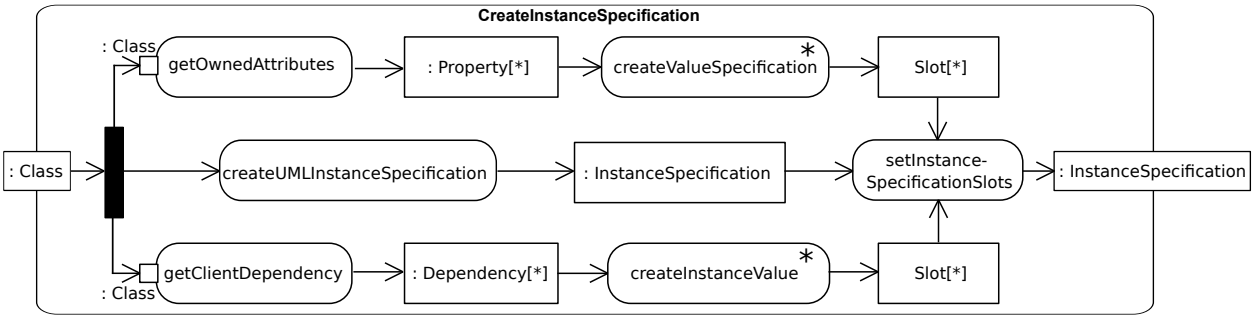

Figure 5: Behavior of Instance Specification Creation.

Figure 5 presents an activity diagram specifying the top-level behavior for variant creation, which is executed with a root class as the input parameter. The variant is created in three steps, which can be carried out in parallel: The first step is to create an instance of UML *InstanceSpecification*, which represents a specific UML class. Thus, *InstanceSpecification* owns property *classifier*, which includes a reference to the class given as ingoing parameter.

Secondly, a list of all owned attributes are selected from *Class* parameter. For each element of resulting *Property* list, action *createValueSpecification* is executed. How does a value, which should be assigned to current *Property* instance, is determined? All stereotypes, which are applied to the *Property* instance, are selected. Variant stereotypes, which extend UML meta class *Property* are analyzed here. If stereotype *intervalValueVariant* is applied, a value is selected by use of *XNumberGenerator*'s function *requestValue(...)*. The required parameters are already defined inside the stereotype. A UML *LiteralReal* is created and the selected value is assigned to it. Furthermore, a slot is created with references to the *Property* instance and *LiteralReal* instance. For stereotype *listValueVariant*, a value is requested in the same way. In difference to the previous stereotype, a list with a size corresponding to the value is created instead of a single literal.

Another stereotype is *optionalValueVariant*, which results in a Boolean literal as its value specification. The decision, which Boolean value should be used, is simply chosen with interval [0,1] with step 1. Thus, the result of *XNumberGenerator* execution can be either '0' or '1'. The last stereotype with variants possibility is called *typeValueVariant*. Since *values* can also include non-numeric values or values with different distances, a value request directly based on the list values is not possible. Alternatively, the value selection is done

indirectly via the list index. An *index* is chosen by the *XNumberGenerator* and the value, which is placed on this index, is assigned to the slot corresponding to the *Property* instance.

Stereotype *derivedValueVariant* includes a behavior, which calculates a value deterministically. This behavior is executable by using fUML execution engine. Thus, the required value can calculated here. If the stereotype *fixedValueVariant* is assigned to the *Property* instance, a *ValueSpecification* is already defined and is assigned to the slot here. If no variant stereotype is applied to a *Property*, but a default value is specified, this value will be used. Otherwise, *LiteralNull* is applied to the corresponding slot instance, meaning that no value is defined explicitly.

In a similar way, the third step creates variants, which are described by instance variant stereotypes. For that, all client dependencies are selected from *Class* parameter and *InstanceValue* instances are created based on applied stereotype. Stereotype *countFixedInstanceVariant* defines a fixed count of instances of a supplier class, which should be created. It is possible, that *InstanceSpecification* instances are already preconfigured, which are used instead of creating new instances. Each preconfigured *InstanceSpecification* is checked, if all class properties are existing and assigned to a *ValueSpecification*. If a property is missing, it is created by the *InstanceSpecificationGenerator*. Similarly, *countVariableInstanceVariant* is processed, in which the instance count is chosen by *XNumberGenerator*. In a similar way to *derivedValueVariant*, the stereotype *derivedInstanceVariant* owns a creation behavior, which is executed for all combinations of input instances. Finally, action *setInstanceSpecificationSlots* inserts the results of the second and third step into the created *InstanceSpecification*.

2.5 Modifications of Architecture Variants

After creating the first architecture variant, further variants may be computed by changing the previous variant in at least one property of an *InstanceSpecification* or even adding or removing an *InstanceSpecification*. For some optimization heuristics a notion of distance between parameter values is assumed (temperature-based selection of next parameter value in simulated annealing, for instance). This assumption cannot easily be transferred to our complex architecture variants, where there may not be any relation depending on the order of settings. This problem will be tackled in future work; for the moment, we assume that the encoding sequence of values is exploited.

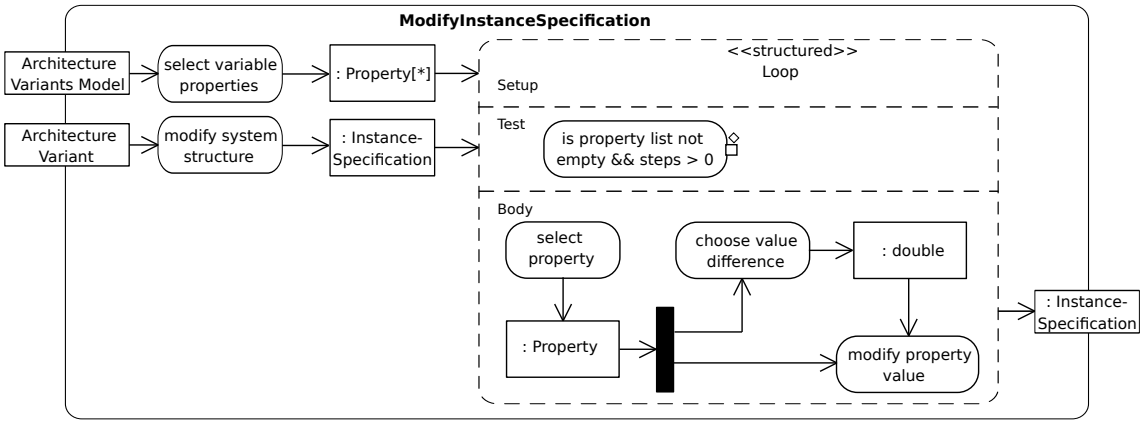

Figure 6: Behavior of Architecture Variant Modification.

To make matters worse, such a distance may be needed also for the complex multi-dimensional design space, and it would not make sense to use space-geometrical (Pythagorean) assumptions on parameters. Instead, we assume here that one step in one parameter is 'as important' as any other parameter step, and

thus propose to apply Manhattan distance as an approximation of 'how distant' two parameter settings are (i.e., variants, or points on the design space). One step is defined as the minimal difference of property values, which is specified also for real-valued parameters. A minimal difference for *intervalValueVariant* is defined by its property *step* for instance. One value step of *typeValueVariant*-properties is defined as an index increment or decrement. For *countVariableInstanceVariant* stereotypes, the minimal step is defined by adding or removing one instance. For each heuristic execution, a calculated step count is available, which can be used to modify the architecture variant. The actual count calculation has to be defined adaptively by specialized heuristics if necessary.

Figure 6 proposes the behavior for modifying an architecture variant using an activity diagram in two steps. Structural modifications on *InstanceSpecification* instances are performed first. For that, all combinations of *countVariableInstanceVariant*-based instance counts are calculated. *XNumberGenerator* is used to select one out of this combination set. *InstanceSpecification* instances are created or deleted depending on the selection result. The step count is decreased by the number of *InstanceSpecification* creations and deletions. The remaining step count is used to change property values of existing *InstanceSpecification* instances. A list of all properties with value variant stereotype is received by *InstanceSpecificationGenerator*. While the step count is greater than zero and there are still unmodified properties, a property and a difference value as well as the direction of change is selected randomly using *XNumberGenerator*. The value setting is done by using *InstanceSpecificationGenerator*.

All presented methods are realized model-based using fUML and is completely executable.

3 AN APPLICATION EXAMPLE

This section describes the derivation of concrete system variant instances for a simplified communication network, in which network nodes communicate using wired and wireless communication protocols. This communication network and the following communication network variants model has been presented in (Wichmann et al. 2017). An *EndNode* can be a server, personal computer or other gadgets and produces data, which should be sent to another *EndNode*. For that, *EndNode* instances can provide WLAN technique or Ethernet slots. Additionally, *AccessPoint* instances could be used to cover large distance between *EndNode* or as connection of WLAN-based and Ethernet-based communication.

3.1 Communication network variants

Figure 7 presents an architecture variants model, which is used by a system optimization process in order to find the best architecture. A *Network* includes instances of *Interface NetworkNode*, which is realized by classes *EndNode* and *AccessPoint*. To specify the count of class instances, which should be owned by *Network*, *Dependency* connections are created between *Network* and *EndNode* as well as *AccessPoint*. *EndNode* is preconfigured with instance specifications and should not be varied. The first *EndNode* represents a smart phone providing WLAN features, but not having an Ethernet port. The other *EndNode* instance is a personal computer with one Ethernet port and without WLAN. This information is specified by using the variant stereotype *countFixedInstanceVariant*.

AccessPoint instances can be varied in their number as well as their properties. For that, stereotype *countVariableInstanceVariant* is applied to the *Dependency* connection. At least one and not more than three *AccessPoint* instances may exist. Additionally, several properties of *AccessPoint* are varied, which is specified by value variant stereotypes. Position of *AccessPoint* and count of Ethernet ports are specified by *intervalValueVariant*. Furthermore, WLAN feature is defined as optional. Connections are established between two nodes, in which a connection can be realized by *WLANConnection* or *LANConnection*. Which connection is to be created, is defined by stereotype *derivedInstanceSpecification*. This stereotype owns a

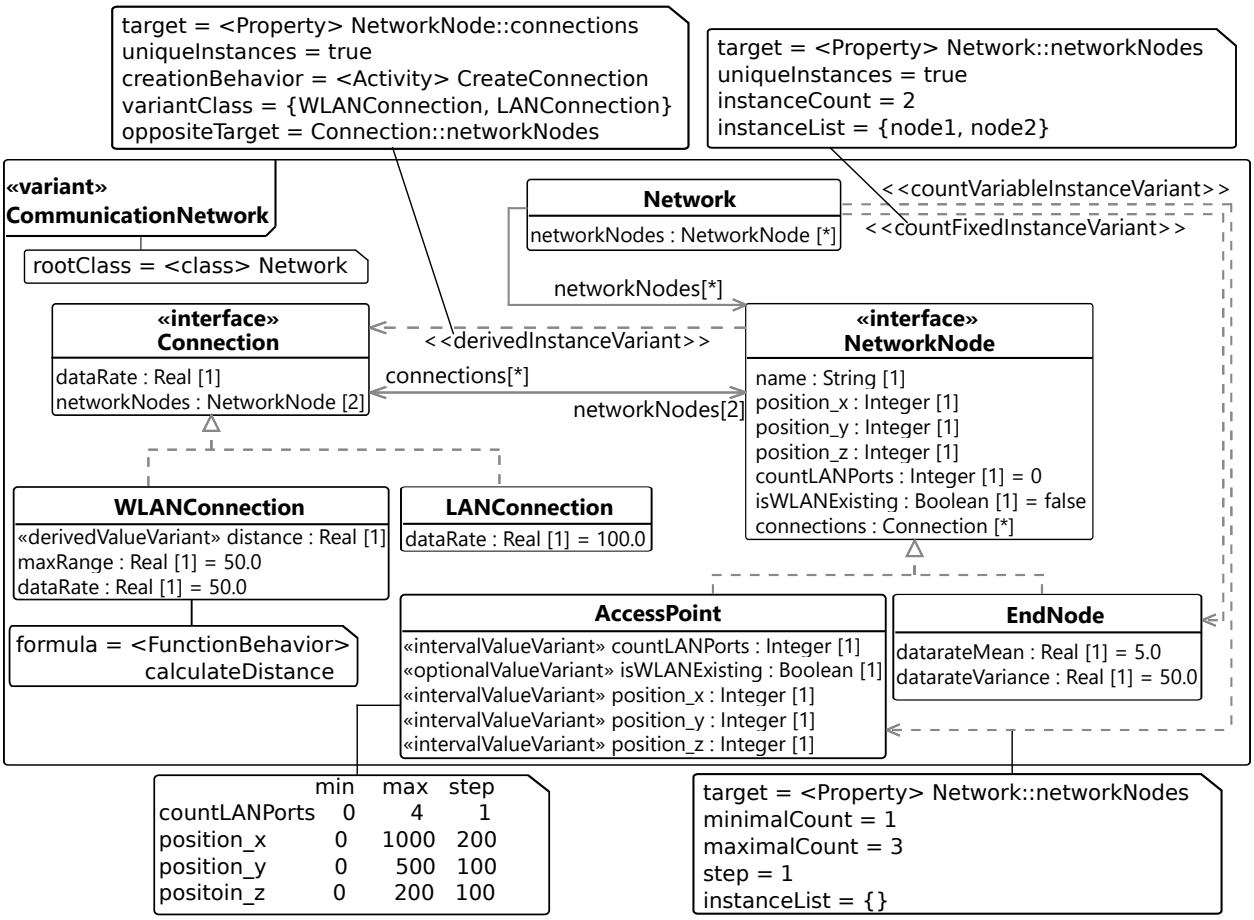

Figure 7: Architecture Variants Model for a Communication Network (Wichmann et al. 2017).

creation behavior, which is executed for each combination of two *NetworkNode* instances. This behavior specifies that a *WLANConnection* can only be created if both nodes provide WLAN. Similarly, a *LANConnection* requires a free Ethernet slot on each node. If both connections are possible, the decision can be influenced by the heuristic.

3.2 Resulting InstanceSpecification instances

The presented approach for deriving concrete system variant instances is specified in a UML-conform way in our system optimization model. Similarly, the architecture variants model is set up as a UML model using the *variant profile*. In order to execute this, our UML4CPP generator (Systems and Software Engineering Group 2016) is used to transform the models into executable C++ code, which is compilable without further implementation efforts. Finally, the resulting system optimization application is executable through the use of our fUML-conform execution engine. The optimization process is executed and several architecture variants are generated based on the architecture variants model of the communication network.

The heuristic is executed for the first time. Thus, a first variant has to be created by executing the behavior presented in Section 2.4. Figure 8 shows the resulting architecture variant as an example. A *network* includes two preconfigured *EndNode* instances as well as two *AccessPoint* instances. An *AccessPoint* supports WLAN and two Ethernet ports. The other *AccessPoint* has three Ethernet ports, but no WLAN. Connections

between these*NetworkNode* instances is realized as follows: *LANConnection* is realized between *node2* and *accessPoint1* as well as between *accessPoint0* and *accessPoint1*. *Node1* is connected to *accessPoint0* by a *WLANConnection*. This architecture variant is evaluated by executing the simulation loop.

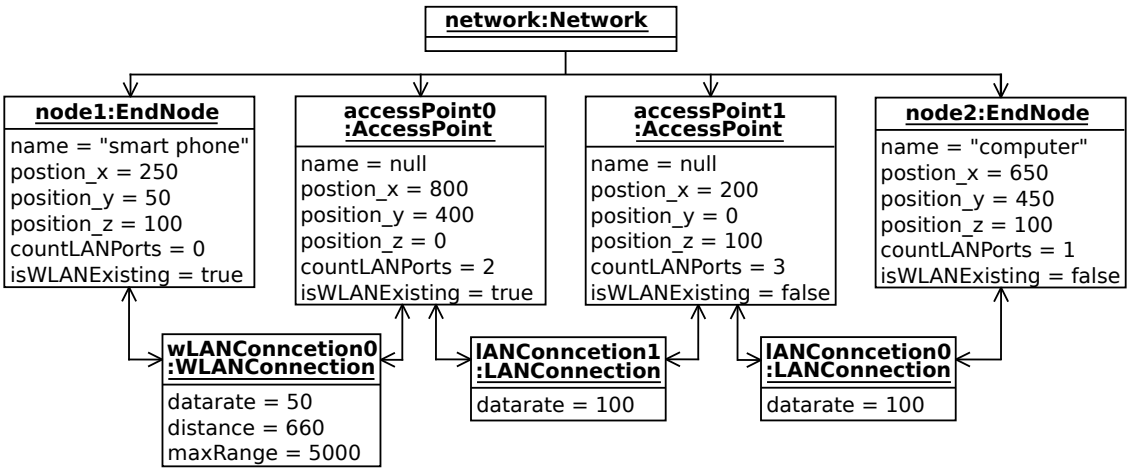

Figure 8: Example for a Created Architecture Variant.

The heuristic is executed a second time afterwards. Now, the existing variant is modified by the behavior specified in Section 2.5. Figure 9 presents an *InstanceSpecification* of communication network model after execution the heuristic for the second time. The count of *AccessPoint* instances has not changed, but their properties. Ethernet ports of *accessPoint0* is increased to '3', while *accessPoint1* decreased to '1'. Furthermore, *accessPoint1* provides WLAN connections now. Thus, the connection between *accessPoint0* and *accessPoint1* must be changed to *WLANConnection*, because the only port is occupied by the connection to *node2*.

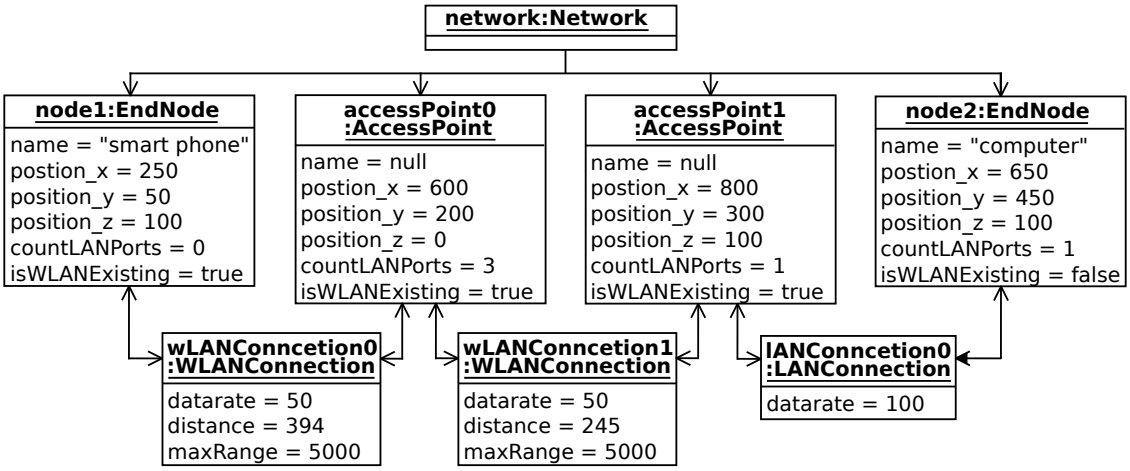

Figure 9: Example for Modified Architecture Variant.

The system optimization process is continued until the termination criterion is fulfilled and the best variant is returned in the end.

4 CONCLUSION

The paper presents an approach for deriving concrete design variant instances from architecture design space descriptions based on a UML profile, with the goal of supporting automatic system architecture optimization. The behavior for creating architecture variants is specified by standard UML meta model elements, which is executable by the application of our UML4CPP generator and C++ fUML-conform execution engine. The derivation of design variants is shown with a simplified communication network model.

Future steps include the investigation of optimization methods suitable for system architectures. The optimization loop will be further refined, in particular for variant evaluations with complex objective functions defined on the model. Furthermore, constraints should be added to the variant model in order to allow formal validity checks for system variants.

REFERENCES

Bedini, F., R. Maschotta, A. Wichmann, S. Jäger, and A. Zimmermann. 2017. "A Model-Driven C++-fUML Execution Engine". In *5th Int. Conference on Model-Driven Engineering and Software Development, MODELSWARD 2017*. Technische Universität Ilmenau. accepted for publication.

Carson, Y., and A. Maria. 1997. "Simulation Optimization: Methods And Applications". In *Proc. of the 29th Winter Simulation Conference*, WSC '97, pp. 118–126.

Eclipse 2014. "Sirius". http://www.eclipse.org/sirius/.

El Kouhen, Amine and Dumoulin, Cedric and Gerard, Sébastien and Boulet, Pierre 2012. "Evaluation of Modeling Tools Adaptation". Available: https://hal.archives-ouvertes.fr/hal-00706701.

Fu, M. C. 1994. "A Tutorial Overview of Optimization via Discrete-Event Simulation". In *11th Int. Conf. on Analysis and Optimization of Systems*, edited by G. Cohen and J.-P. Quadrat, Volume 199 of *Lecture Notes in Control and Information Sciences*, pp. 409–418. Sophia-Antipolis, Springer-Verlag.

Gerlinger Romero, A., K. Schneider, and M. Goncalves Vieira Ferreira. 2013, Sept. "Towards the applicability of alf to model Cyber-Physical Systems". In *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on*, pp. 1427–1434.

Giese, H., S. Hildebrandt, and A. Seibel. 2009, 0. "Improved Flexibility and Scalability by Interpreting Story Diagrams". In *Proceedings of the Eighth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2009)*, edited by T. Magaria, J. Padberg, and G. Taentzer, Volume 18, Electronic Communications of the EASST.

Jäger, S., R. Maschotta, T. Jungebloud, A. Wichmann, and A. Zimmermann. 2016. "An EMF-like UML Generator for C++". In *4th Int. Conference on Model-Driven Engineering and Software Development, MODELSWARD 2016*. Technische Universität Ilmenau. submitted for publication.

Lazar, C.-L., I. Lazar, B. Parv, S. Motogna, and I.-G. Czibula. 2010. "Tool Support for fUML Models". In *Int. J. of Computers, Communications & Control*.

Liberti, L., and N. Maculan. 2006. *Global Optimization: From Theory to Implementation*. Springer Verlag.

OMG 2013a. "Concrete Syntax for a UML Action Language: Action Language for Foundational UML". Technical report, Object Management Group.

OMG 2013b. "Semantics of a Foundational Subset for Executable UML Models". Technical report, Object Management Group.

OMG 2015. "Unified Modeling Language (OMG UML), Version 2.5". Technical report, Object Management Group.

Systems and Software Engineering Group 2016. "Model Driven Engineering for C++ (MDE4CPP), sse.tu-ilmenau.de/mde4cpp".

van Leeuwen, C., J. de Gier, J. O. de Filho, and Z. Papp. 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., S. Jäger, T. Jungebloud, R. Maschotta, and A. Zimmermann. 2015. "System Architecture Optimization With Runtime Reconfiguration of Simulation Models". In *IEEE International Systems Conference (SYSCON'15)*. Technische Universität Ilmenau.

Wichmann, A., S. Jäger, T. Jungebloud, R. Maschotta, and A. Zimmermann. 2016. "Specification and Execution of System Otimization Processes with UML Activity Diagrams". In *IEEE International Systems Conference (SYSCON'16)*. Technische Universität Ilmenau.

Wichmann, A., R. Maschotta, F. Bedini, S. Jäger, and A. Zimmermann. 2017. "A UML Profile for the Specification of System Architecture Variants Supporting Design Space Exploration and Optimization". In *5th Int. Conference on Model-Driven Engineering and Software Development, MODELSWARD 2017*. Technische Universität Ilmenau. accepted for publication.

AUTHOR BIOGRAPHIES

ALEXANDER WICHMANN received the Master's degree in Computer Science from the Technische Universität Ilmenau, Germany, in 2013. He is currently working towards the Ph.D. degree in the Systems and Software Engineering Group of Technische Universität Ilmenau. His main research interests include model-based specification and execution of system optimization processes. His email address is alexander.wichmann@tu-ilmenau.de.

FRANCESCO BEDINI received the Master's degree in Research in Computer and Systems Engineering with distinction from Technische Universität Ilmenau in 2016. He is currently working towards the Ph.D. degree in the Systems and Software Engineering Group of TU Ilmenau. His main research interests include efficient generation of code from UML and fUML models and their validation. His email address is francesco.bedini@tu-ilmenau.de.

RALPH MASCHOTTA received the Diploma degree in technical computer science from the University of Applied Sciences Schmalkalden, Germany, in 1999 and the Ph.D. degree from Technische Universität Ilmenau, Germany, in 2008. Since 2011, he has been a senior scientist and lecturer with the Systems and Software Engineering Group, Faculty of Computer Science and Automation, TU Ilmenau. His main research interests include object-oriented programming, modeling, and design of software systems, as well as image processing and image recognition in medical and industrial applications. His email address is ralph.maschotta@tu-ilmenau.de.

ARMIN ZIMMERMANN received the Diploma, Ph.D., and Habilitation degrees from Technische Universität Berlin, Germany, in 1993, 1997, and 2006, respectively. He has been a full Professor of systems and software engineering since 2008 and the director of the Institute for Computer and Systems Engineering since 2012 with Technische Universität Ilmenau, Germany. His research interests include discrete event system modeling and performance evaluation and their tool support with embedded systems applications. He is a member of the Industrial Automated Systems and Controls Subcommittee of the IEEE IES Technical Committee on Factory Automation. His email address is armin.zimmermann@tu-ilmenau.de.

This work was supported by the *Federal Ministry of Economic Affairs and Energy* of Germany [20K1306D] and Federal Ministry for Education and Research of Germany [01S13031A].