

Delta Modeling for Software Architectures

Arne Haber¹, Holger Rendel¹, Bernhard Rumpe¹, Ina Schaefer²

¹Software Engineering
RWTH Aachen University, Germany
<http://www.se-rwth.de/>

²Software Systems Engineering Institut
Technische Universität Braunschweig, Germany
<http://www.tu-braunschweig.de/sse>

Abstract: Architectural modeling is an integral part of modern software development. In particular, diverse systems benefit from precise architectural models since similar components can often be reused between different system variants. However, during all phases of diverse system development, system variability has to be considered and modeled by appropriate means. Delta modeling is a language-independent approach for modeling system variability. A set of diverse systems is represented by a core system and a set of deltas specifying modifications to the core system. In this paper, we give a first sketch of how to apply delta modeling in MontiArc, an existing architecture description language, in order to obtain an integrated modeling language for architectural variability. The developed language, Δ -MontiArc, allows the modular modeling of variable software architectures and supports proactive as well as extractive product line development.

1 Introduction

Today, modeling of software architectures is an integral part of the software development process [MT10]. Dividing a system into small manageable parts provides an overview of the system structure, subcomponent dependencies, and communication paths. Architectural modeling allows reasoning about structural system properties in early development stages. Reusing well-defined modular components reduces development costs and increases product quality. Especially embedded system architectures have to be carefully designed, because resources like bandwidth or memory are restricted, product quantities are high, and faults that are detected late are extremely expensive.

Diversity is prevalent in modern software systems, in particular in the embedded systems domain. Systems exist in many different variants simultaneously in order to adapt to their application context or customer needs. Software product line engineering [PBvdL05] is a commercially successful approach to develop a set of systems with well-defined commonality and variability. Product line engineering benefits from architectural modeling, since common components can be reused in different system variants. However, in order

to develop a software product line, system variability has to be considered in all phases of the development process, including architectural modeling. This means that the variability of the architectures in different system variants has to be modeled by appropriate means.

Delta modeling [CHS10, SBB⁺10, Sch10] is a transformational approach to represent product variability. It combines the modular representation of changes between system variants with expressive means to capture the influence of product features. In delta modeling, a set of diverse systems is represented by a designated core system and a set of deltas describing modifications to the core system to implement further system variants. A particular product configuration is obtained by applying the changes specified in the applicable deltas to the core system. The concepts of delta modeling are language-independent.

In this paper, we apply delta modeling to represent variability in software architectures that are described by the existing architecture description language *MontiArc* [HKRR11]. *MontiArc* is designed to model architectures for asynchronously communicating (logically) distributed systems. In order to express variable *MontiArc* architectures by delta modeling, the modification operations that can be specified in deltas over architecture descriptions will be defined. An architectural delta can add, remove and modify components and alter the communication structure between these components. This reflects the variability that is induced by different product features. By applying a sequence of deltas to the *MontiArc* description of a designated core architecture, *MontiArc* descriptions for architectures of other product variants can be obtained. The resulting modeling language for architectural variability, Δ -*MontiArc*, provides means to modularly specify architectural variability by defining architectural deltas.

Δ -*MontiArc* supports the proactive and extractive development [Kru02] of software product line architectures. Hence, it can be used methodologically as an incremental variability model where components are added or replaced by refined and enhanced versions. This is in contrast to annotative approaches [ZHJ03, Gom04, CA05], where a model containing all possible variants is stripped down for a particular feature configuration. Furthermore, existing legacy applications can be transformed into a product line by specifying the architectural deltas that are required to describe the complete product space.

2 Architectural Modeling with *MontiArc*

Architecture description languages (*ADLs*) [MT00] support the modeling, design, analysis, and evolution of system architectures. In *ADLs*, architectures are generally described in terms of components, connectors and communication relationships. *ADLs* facilitate a high-level description of the system structures in a specific domain and support reasoning about structural system properties [GMW97].

The textual *ADL* *MontiArc* [HKRR11] is designed for modeling distributed information flow architectures in which communication is based on asynchronous messages. It is developed using the DSL framework *MontiCore* [GKR⁺08] that supports the agile development of textual domain-specific languages. Following [MT00], architectural components in *MontiArc* are units of computation or storage defining their computational commitments

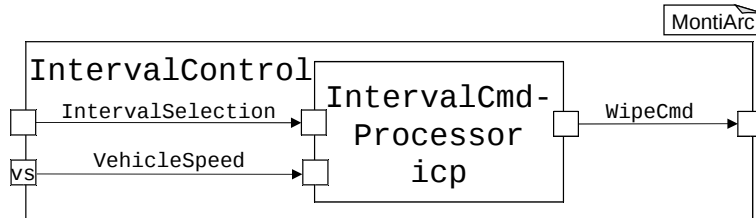


Figure 1: Architecture of the IntervalControl

via interfaces. These interfaces are the only interaction points of components to provide clear concepts of interaction between entities of computation [BS01].

As an example of a system architecture description in MontiArc, we use an excerpt of an embedded windshield wiper system. The architecture is graphically represented in Figure 1. The component `IntervalControl` receives an interval selection from the driver and emits a command `WipeCmd` to control a connected wiper actuator. It is hierarchically decomposed into a subcomponent `IntervalCmdProcessor` named `icp` that calculates the desired wiper behavior based on the selection and the vehicle speed. In MontiArc, a subcomponent is either an instance of a component referenced from a separate model, or a local component definition (similar to a private inner classes in object-oriented programming languages). The interface definition of the component `IntervalControl` contains an incoming port `IntervalSelection`, an incoming port `vs`, and an outgoing port `WipeCmd`. All ports are connected to the subcomponent `icp` using connectors. Ports and subcomponents obey an implicit naming rule. If the used type is unique in the current component definition, the usage of an explicit name is optional and the artifact is implicitly named after its type.

Listing 1 shows the textual description of the above described architecture (cf. Figure 1) in MontiArc syntax. Components are organized in packages (l. 1). A component defini-

```

1 package wipe;
2
3 component IntervalControl {
4     autoconnect port;
5
6     port
7         in IntervalSelection,
8         in VehicleSpeed vs,
9         out WipeCmd;
10
11     component IntervalCmdProcessor icp;
12
13     connect vs -> icp.VehicleSpeed;
14 }

```

Listing 1: Structural component `IntervalControl` in MontiArc syntax

tion consists of the keyword `component` followed by its name and curly brackets that surround the component's architectural elements. As in the graphical notation, the component's interfaces are described by ports (ll. 6-9). The keyword `port` and the direction (in or out) are followed by a communication data type and the port's name (optional). Subcomponents that instantiate other component definitions are created with the keyword `component` followed by the subcomponent's type and an optional name (l. 11). Subcomponents that define an inner component are created with the same component definition syntax as described above.

MontiArc offers several mechanisms to connect subcomponents: the `autoconnect port` (l. 4) statement automatically connects all type-compatible ports with the same name. If the `autoconnect` statement is parametrized with the keyword `type`, ports with the same type are connected. Using this mechanism, the connector from the outgoing port `WipeCmd` of component `icp` to the outgoing port `WipeCmd` is created in the example. If it is not possible to create all connections automatically as uniquely identifying names may not be always given, explicit connections can be created using the `connect` statement (l. 13) connecting one source port with one or more target ports. If a target or source port belongs to a subcomponent, it is qualified with the subcomponent's name. Implicit connectors can always be redefined by explicit connector definitions.

3 Delta Modeling

Delta modeling [CHS10, SBB⁺10, Sch10] is a transformational approach for modular variability modeling. The concepts of delta modeling are language-independent. A *delta-oriented product line* is split into a core model and a set of model deltas that are developed during domain engineering [PBvdL05]. The *core model* corresponds to a product for some valid feature configuration. The core model can be developed according to well-established single application engineering principles or derived from an existing legacy system. The variability of the product line is handled by model deltas. The *model deltas* specify modifications to the core model in order to integrate other product features. The modifications comprise additions of modeling entities, removals of modeling entities, and modifications of modeling entities by changing the internal structure of these entities. The model deltas contain *application conditions* determining under which feature configurations the specified modifications have to be carried out. These application conditions are Boolean constraints over the features in the feature model and build the connection between the feature model and the variability of the product artifacts. A model delta does not necessarily refer to exactly one feature, but potentially to a combination of features which allows handling the influence of feature combinations to the artifacts individually. The number of model deltas that have to be created to cover the complete product space depends on the desired granularity of the application conditions.

A model for a particular feature configuration is obtained by *delta application*. The modifications specified by the model deltas which have a valid application condition for the respective feature configuration are applied to the core model. To avoid conflicts between modifications targeting the same model entities, a partial order between the model deltas

can be defined that determines in which order the model deltas have to be applied if applied together. The partial order captures the necessary dependencies in order to guarantee that for every feature configuration a unique model can be generated. Furthermore, the partial order ensures that a model delta is applicable to the core model or an intermediate model during delta application. *Applicability* requires that added model entities do not exist and removed and modified entities exist. An intermediate model during delta application may not be well-formed according to the well-formedness rules of the underlying modeling language. However, after all applicable model deltas are applied, the resulting product model must be well-formed.

4 Architectural Variability Modeling in Δ -MontiArc

In order to make the ADL MontiArc presented in Section 2 more amenable to product line engineering, we apply delta modeling presented in Section 3 to MontiArc. This results in the architectural variability modeling language Δ -MontiArc. The modification operations that can be specified in model deltas in Δ -MontiArc allow the addition, removal and modification of all architectural elements of MontiArc, i.e., subcomponents, associated ports and connectors. The replace operation substitutes a subcomponent by another subcomponent with the same interface. This allows changing the internal realization of a subcomponent.

A MontiCore [GKR⁺08] grammar defining the syntax of Δ -MontiArc is shown in Listing 2. Based on MontiCore's reuse mechanisms, the existing MontiArc grammar can be extended by the `MADelta` production defining model deltas for MontiArc descriptions. The production `ArcElement` for representing architectural elements, like components, ports, or connectors, in MontiArc is also reused. A model delta in Δ -MontiArc starts with the keyword `delta` (l. 2) followed by the name of the delta. Using the optional `after` clause (l. 3), a partial order on the application of model deltas can be defined. The keyword `after` is followed by a list of model delta names. The defined model delta has to be applied after the listed deltas during delta application, if the deltas are applied together. The keyword `when` is followed by a logical constraint (l. 5) which defines the application condition of the model delta. The `Constraint` production represents Boolean constraints over the feature model and is omitted for space reasons. The changes of a MontiArc architecture specified by a model delta are defined using the production `MADeltaBody` (l. 7) which may contain an optional (?) `ExpandAutoConStatement` and an arbitrary number (*) of statements defining modifications of architectural elements. The `ExpandAutoConStatement` (ll. 20-21) triggers the MontiArc autoconnect mechanism after the application of delta modifications. The interface `Statement` is implemented by several productions that allow modifying, adding, or removing `ArcElements` from the target model (ll. 11-16). The `ReplaceStatement` (ll. 17-19) removes a subcomponent `oldComp` and replaces it with a new subcomponent `newComp`.

A Δ -MontiArc product line model consists of the core model which is a standard MontiArc architecture for the core feature configuration and a set of model deltas specified using the above syntax. A MontiArc architecture for a particular feature configuration is

```

1 MADelta =
2   "delta" Name
3   ("after" predecessors:QualifiedName
4     ("," predecessors:QualifiedName)? )?
5   "when" Constraint MADeltaBody;
6
7 MADeltaBody = "{" ExpandAutoConStatement? Statement* "}";
8
9 interface Statement;
10
11 ModifyStatement implements Statement =
12   "modify" ArcElement ";";
13 AddStatement implements Statement =
14   "add" ArcElement ";";
15 RemoveStatement implements Statement =
16   "remove" ArcElement ";";
17 ReplaceStatement implements Statement =
18   "replace" oldComp:ArcReference
19   "with" newComp:ArcReference ";";
20 ExpandAutoConStatement =
21   "expand autoconnect;";

```

Listing 2: MontiCore grammar for the Δ -MontiArc language

obtained from a Δ -MontiArc product line model as follows: First, determine the model deltas that have a valid application condition in the `when` clause for the given feature configuration. Second, find a linear order of the applicable deltas that is compatible with the partial order of delta application specified in the `after` clauses. Third, apply the modifications specified in the model deltas one by one in the given order to the core model. If the `expand autoconnect` statement is provided, matching ports are connected using the autoconnect mechanism, after the architectural modifications specified in the delta.

In order to ensure that the application of every model delta is defined, the following conditions must hold for each model delta during delta application. In the current stage of language definition, we use a rather restrictive approach. (1) A subcomponent named *sc* must not be added to component *c*, if *c* already contains a subcomponent named *sc*. (2) A port named *p* must not be added to component *c*, if *c* already contains a port named *p*. (3) A connector with the target port *tp* must not be added to component *c*, if *c* already contains a connector with the target *tp*, or if *tp* does not exist (as a port in *c* or a port in a subcomponent of *c*). (4) An architectural element named *ae* must not be removed from component *c*, if *c* does not contain an architectural element named *ae*. A port named *p* must not be removed from component *c*, if *c* contains a connector with *p* as its source or target. (5) A subcomponent named *sc* must not be removed from component *c*, if *c* contains a connector that has a port that belongs to *sc* as its source or target. (6) An architectural element named *ae* must not be modified, if an architectural element named *ae* does not exist. (7) The operator `replace` can only substitute a subcomponent *sc*₁ of component *c* with a subcomponent *sc*₂ (specified by the `Target` construct), if *sc*₂ has the same interface like *sc*₁. The new component *sc*₂ will be connected as the component *sc*₁.

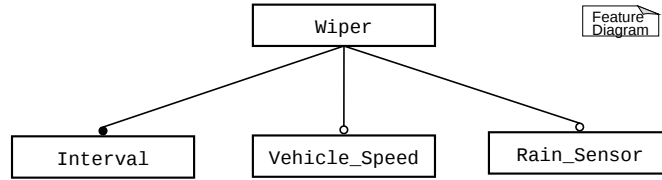


Figure 2: Feature model for the IntervalControl

```

1 delta DRainSensor when Rain_Sensor {
2
3   expand autoconnect;
4
5   modify component IntervalControl {
6     add port in RainSensorStat;
7     add component RainEval;
8   };
9
10  modify component IntervalCmdProcessor {
11    add port in RainIntensity;
12  };
13
14 }
  
```

Listing 3: Model delta adding the Rain_Sensor feature

Example To illustrate architectural variability modeling using Δ -MontiArc, we transform the component `IntervalControl` presented in Section 2 into a product line of similar components. The product line of similar components can be described by the feature model shown in Figure 2. All components have a mandatory feature `Interval` which allows selecting the interval for the windshield wipers. Optional features are `Vehicle_Speed` which adjusts the wiping speed based on the vehicle speed and a `Rain_Sensor` which determines the wiping based on the current rain intensity. As core model for the delta-oriented product line, we choose the component `IntervalControl` shown in Listing 1 implementing the the features `Interval` and `Vehicle_Speed`.

Listing 3 shows the model delta `DRainSensor` adding the `Rain_Sensor` feature. The model delta adds a new component `RainEval` and an incoming port `RainSensorStat` for the input from a rain sensor. Furthermore, an incoming port `RainIntensity` is added to the component `IntervalCmdProcessor` receiving the output from the component `RainEval`. The connections between the added components and ports are created by the `expand autoconnect` statement. Applying the delta `DRainSensor` to the core model results in a system with all three features.

In order to obtain a system which only realizes the mandatory feature, we have to provide a model delta that removes the feature `Vehicle_Speed`. This delta `DRemSpeed` is shown in Listing 4. It removes the ports `VehicleSpeed` from the sub- and supercomponents of the core model. The associated connector is also removed due to the usage of

```

1 delta DRemSpeed when !Vehicle_Speed {
2
3   expand autoconnect;
4
5   modify component IntervalCmdProcessor {
6     remove port in VehicleSpeed;
7   };
8
9   modify component IntervalControl {
10    remove port in VehicleSpeed;
11  };
12
13 }

```

Listing 4: Model delta removing `Vehicle_Speed` feature

`expand autoconnect`. In order to obtain the fourth possible system variant, realizing the feature `Rain_Sensor`, but not the feature `Vehicle_Speed`, both model deltas have to be applied the core model.

5 Related Work

Existing approaches to express variability in modeling languages can be classified in two main directions [VG07]: annotative and compositional approaches. Annotative approaches consider one model of all products of the product line. Variant annotations, e.g., UML stereotypes [ZHJ03, Gom04] or presence conditions [CA05], define which parts of the model have to be removed to derive a concrete product model. The orthogonal variability model (OVM) [PBvdL05] captures the variability of product line artifacts in a separate variability model where artifact dependencies take the place of annotations. In the Koala component model [vO05], the variability of a component architecture that contains all possible components is expressed by explicit components, called switches. Switches select between component variants in different system configurations playing the role of annotations. Compositional approaches associate model fragments with product features that are composed for a particular feature configuration. In [HW07, VG07, NK08], models are constructed by aspect-oriented composition. In [AJTK09], model fragments are composed by model superposition. In feature-oriented model-driven development [SBD07], a product model is composed from a base module and a sequence of feature modules. Additionally, model transformations [HMPO⁺08, JWEG07] can represent variations of models, e.g., in [TM10, WF02], architectural variability is captured by graph transformation rules. Delta modeling [CHS10, SBB⁺10, Sch10] can be seen as a transformational approach which provides a modular definition of transformations dependent on feature configurations.

For architectural variability modeling in [MKM06], a resemblance operator is provided that allows creating a component that is a variant of an existing component by adding,

deleting, renaming or replacing component elements. The old and the new component can be used together to build further components. Instead, in delta modeling, the existing component is destroyed since a complete model of the resulting architecture will be generated by delta application. In [HvdH07], architectural variability is represented by change sets containing additions and removals of components and component connections that are applied to a base line architecture. Relationships between change sets specify which change sets may be applied together similar to application conditions of model deltas. However, the order in which change sets are applied cannot be explicitly specified. Conflicts between change sets have to be resolved by excluding the conflicting combination using a relationship and providing a new change set covering the combination which may lead to a combinatorial explosion of change sets to represent all possible variants.

6 Conclusion

In this paper, we have presented a first version of Δ -MontiArc to support the flexible modular modeling of architectural variability by extending the existing ADL MontiArc with the concepts of delta modeling. In order to guarantee the uniqueness of the resulting artifacts we plan to apply the criteria given in [CHS10] to Δ -MontiArc. We expect to get more experience from case studies to investigate how to optimize this integration and compare it with other similar approaches. We are in particular interested in restricted forms of modifications that add and refine architectures. An appropriate refinement calculus for this is given in [PR99]. Additionally, we aim at developing a tool infrastructure for Δ -MontiArc. Common language artifacts, like symbol tables, context condition checkers and model editors, as well as a generator that creates a MontiArc model for a particular feature configuration by delta application have to be developed. Δ -MontiArc is a promising language for modeling architecture evolution since delta modeling is flexible enough to deal with anticipated and unanticipated variability.

References

- [AJTK09] Sven Apel, Florian Janda, Salvador Trujillo, and Christian Kästner. Model Superimposition in Software Product Lines. In *International Conference on Model Transformation (ICMT)*, 2009.
- [BS01] Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement*. Springer Verlag Heidelberg, 2001.
- [CA05] Krzysztof Czarnecki and Michal Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *GPCE*. 2005.
- [CHS10] D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract Delta Modeling. In *Proc. of GPCE*, 2010.
- [GKR⁺08] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. MontiCore: a Framework for the Development of Textual Domain Specific Languages. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, 2008.

- [GMW97] David Garlan, Robert T. Monroe, and David Wile. Acme: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, 1997.
- [Gom04] H. Gomma. *Designing Software Product Lines with UML*. Addison Wesley, 2004.
- [HKRR11] Arne Haber, Thomas Kutz, Jan Oliver Ringert, and Bernhard Rumpe. MontiArc 1.1.3 - Architectural Modeling Of Interactive Distributed Systems. Technical report, RWTH Aachen University, 2011. (to appear).
- [HMPO⁺08] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. Olsen, and A. Svendsen. Adding Standardized Variability to Domain Specific Languages. In *SPLC*, 2008.
- [HvdH07] Scott A. Hendrickson and André van der Hoek. Modeling Product Line Architectures through Change Sets and Relationships. In *ICSE*, pages 189–198, 2007.
- [HW07] F. Heidenreich and Chr. Wende. Bridging the Gap Between Features and Models. In *Aspect-Oriented Product Line Engineering (AOPLE'07)*, 2007.
- [JWEG07] Praveen K. Jayaraman, Jon Whittle, Ahmed M. Elkhodary, and Hassan Gomma. Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In *MoDELS*, pages 151–165, 2007.
- [Kru02] Charles Krueger. Eliminating the Adoption Barrier. *IEEE Software*, 19(4):29–31, 2002.
- [MKM06] Andrew McVeigh, Jeff Kramer, and Jeff Magee. Using resemblance to support component reuse and evolution. In *SAVCBS*, pages 49–56, 2006.
- [MT00] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 2000.
- [MT10] Nenad Medvidovic and Richard N. Taylor. Software architecture: foundations, theory, and practice. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE '10*, pages 471–472, New York, NY, USA, 2010. ACM.
- [NK08] N. Noda and T. Kishi. Aspect-Oriented Modeling for Variability Management. In *SPLC*, 2008.
- [PBvdL05] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer, 2005.
- [PR99] Jan Philipps and Bernhard Rumpe. Refinement of Pipe And Filter Architectures. In *FM'99, LNCS 1708*, pages 96–115, 1999.
- [SBB⁺10] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented Programming of Software Product Lines. In *SPLC*, 2010.
- [SBD07] S. Trujillo, D. Batory, and O. Díaz. Feature Oriented Model Driven Development: A Case Study for Portlets. In *ICSE*, 2007.
- [Sch10] Ina Schaefer. Variability Modelling for Model-Driven Development of Software Product Lines. In *VaMoS*, 2010.
- [TM10] D. Tamzalit and T. Mens. Guiding Architectural Restructuring through Architectural Styles. In *Proc. ECBS 2010*, pages 69–78. IEEE, 2010.
- [VG07] Markus Völter and Iris Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *SPLC*, pages 233–242, 2007.
- [vO05] Rob C. van Ommering. Software Reuse in Product Populations. *IEEE Trans. Software Eng.*, 31(7):537–550, 2005.
- [WF02] Michel Wermelinger and Jos Luiz Fiadeiro. A graph transformation approach to software architecture reconfiguration. *Science of Computer Programming*, 44(2):133 – 155, 2002.
- [ZHJ03] Tewfik Ziadi, Loïc Hérouët, and Jean-Marc Jézéquel. Towards a UML Profile for Software Product Lines. In *Product Family Engineering (PFE)*, pages 129–139, 2003.