Konzepte zur Erweiterung des SPES Meta-Modells um Aspekte der Variabilitäts- und Deltamodellierung

Peter Manhart[1], Pedram Mir Seyed Nazari[2], Bernhard Rumpe[2], Ina Schaefer[3], und Christoph Schulze[2]

[1]Software-Variantenmanagement, Daimler AG, http://www.daimler.com
[2]Software Engineering, RWTH Aachen, http://www.se-rwth.de
[3]Software Engineering and Automotive Informatics, TU Braunschweig, http://www.tu-bs.de/isf

Abstract: In diesem Beitrag werden Konzepte zur Erweiterung eines mehrperspektivischen Meta-Modells (am Beispiel des SPES Meta-Modells) um Aspekte der Variabilitätsmodellierung durch das Konzept der Delta-Modellierung vorgestellt. Die Konzepte werden exemplarisch anhand der logischen und der funktionalen Perspektive des SPES Meta-Modells illustriert. Die vorgestellten Konzepte können ohne Weiteres auch auf die Anforderungs- und die technische Perspektive angewendet werden. Eine Besonderheit dabei sind die Cross-Cutting-Deltas. Diese gruppieren Deltas der einzelnen Perspektiven und ermöglichen somit Deltas über mehrere Perspektiven hinweg.

1 Einleitung

Das SPES Meta-Modell [Po12] basiert auf einer schrittweisen Verfeinerung der Systembeschreibung und wurde im Rahmen des vorangegangenen Forschungsprojekts SPES 2020 entwickelt. Ziel dieses Forschungsprojektes ist die durchgängig modulare und modellbasierte Entwicklung von eingebetteten Systemen. Dabei wird das System aus unterschiedlichen Perspektiven (oft auch Viewpoints genannt) - Anforderungs-, funktionale oder logische Perspektive - und auf unterschiedlichen Abstraktionsebenen betrachtet. Diese Aufteilung (Abstraktionslevel und Perspektiven) teilen das SPES Meta-Modell in eine (SPES-)Matrix ein (siehe Abbildung 1). Das Meta-Modell fordert nicht ein, dass auf jedem Granularitätslevel und in jeder Perspektive Systemelemente spezifiziert werden. Das bedeutet, dass ein Modell in Level n beispielsweise auch erst 3 Level tiefer verfeinert werden kann. Auch muss kein Modell einer bestimmten Perspektive in den anderen Perspektiven auf demselben Level existieren. Das SPES Meta-Modell erlaubt es, eine Vielzahl von verschiedenen Sprachen, wie zum Beispiel Sequenzdiagramme, Tabellen, Aktivitätsdiagramme und Zustandsdiagramme, zu verwenden. Des Weiteren sind die Zellen der SPES-Matrix in der Regel modular, d.h., diese können unabhängig voneinander entwickelt werden. Außerdem sind weitere Erweiterungen des Meta-Modells, sowohl um Perspektiven (Viewpoints) als auch um neue Sprachen innerhalb der Perspektiven, im Zuge weiterer Forschungsarbeiten beabsichtigt.

[MMR+13] P. Manhart, P. Mir Seyed Nazari, B. Rumpe, I. Schaefer, C. Schulze
Konzepte zur Erweiterung des SPES Meta-Modells um Aspekte der Variabilitäts- und Deltamodellierung
In: Workshopband Software Engineering 2013, GI-Edition Lecture Notes in Informatics (LNI)
Dritter Workshop zur Zukunft der Entwicklung softwareintensiver eingebetteter Systeme (ENVISION2020),
pp. 283 - 292, 2013.
www.se-rwth.de/publications

In diesem Artikel wird ein Ansatz vorgestellt, um das SPES Meta-Modell - beispielhaft für mehrperspektivische Meta-Modelle mit beliebig vielen Abstraktionsleveln und Sprachen - um Aspekte der Variabilitätsmodellierung durch das Konzept der Delta-Modellierung zu erweitern. Die Delta-Modellierung [CHS10, Sc10] ist dafür gut geeignet, da sie eine modulare und sprachunabhängige Möglichkeit darstellt, die Variabilität von Softwareartefakten zu beschreiben. Eine Systemfamilie wird dabei durch ein ausgezeichnetes Kernsystem und eine Menge von Deltas beschrieben. Das Kernsystem ist eine vollständige Systemvariante, die mit bewährten Prozessen entwickelt werden kann, um ihre Qualität sicherzustellen. Ein Delta beschreibt Modifikationen des Kernsystems, um weitere Systemvarianten zu realisieren. Modifikationen sind in der Regel das Hinzufügen, Entfernen, Verändern oder Ersetzen von Systemelementen. Durch Anwendung einer Menge von Deltas auf das Kernsystem können diese Systemvarianten automatisch erzeugt werden. Eine konkrete Variante ist dann jeweils durch das Basismodel und einer konkreten Kombination von Deltas gegeben. Dies erlaubt auch die Wiederverwendung einzelner Deltas, um unterschiedliche Varianten zu beschreiben. Zum Beispiel kann die grundsätzliche Funktionalität einer Steuerung für die Innenraumbeleuchtung eines Autos als Basismodell definiert werden. Diese schaltet abhängig vom Zustand des zugehörigen Lichtschalters das Licht an oder aus. Eine Steuerung, die unabhängig vom Zustand des Lichtschalters auch bei Öffnung der Tür das Licht einschaltet, wird dann als Variante des Basismodells definiert. Dazu ist es nur nötig, die zusätzliche Funktionalität und die Modifikationen der Basisfunktionalität in einem Delta zu beschreiben.

Der weitere Aufbau dieses Papiers ist wie folgt: Einen Überblick über verwandte Arbeiten gibt Abschnitt 2. Abschnitt 3 beschreibt kurz die Idee zur Erweiterung des SPES-Meta-Modells um Konzepte der Delta-Modellerierung und wird in Abschnitt 4 exemplarisch für Deltas innerhalb einer Perspektiven als auch über mehrere Perspektiven hinweg angewendet. In Abschnitt 5 wird das Konzept anhand eines konkreten Beispiels veranschaulicht. Abschließend fast Abschnitt 6 den vorgestellten Ansatz zusammen.

2 Verwandte Arbeiten

Variabilitätsmodelle können den Problemraum eines Systems, d.h., die Interessen und Wünsche der Stakeholder, oder den Lösungsraum erfassen. Lösungsraumvariabilität (*solution space variability* [Me07]) behandelt die Variabilität von wiederverwendbaren Artefakten, wie Architekturelementen oder anderen Entwicklungsdokumenten. Im Folgenden werden Variabilitätsmodellierungsansätze, für den Lösungsraum betrachtet, da sich der in diesem Artikel vorgestellte Ansatz auf den Lösungsraum bezieht. Die verschiedenen existierenden Ansätze eignen sich unterschiedlich gut zur Erweiterung der SPES Meta-Modells um Aspekte der Variabilitätsmodellierung. Die zwei wichtigen Einflussfaktoren dabei sind die Sprachunabhängigkeit und Modularität eines Ansatzes.

Annotative Ansätze (*annotative approaches*) fassen alle Varianten in einem einzigen Modell (oft 150%-Modell genannt) zusammen, was eine feingranulare Repräsentation der Unterschiede zwischen den einzelnen Varianten erlaubt [Gr08]. Variantenannotationen, wie zum Beispiel UML-Stereotypen [Go05] oder Presence Conditions [Cz05], definieren

welche Teile des Modells entfernt werden müssen, um ein konkretes Modell eines Produkts zu erhalten. Beim Orthogonal Variability Model (OVM) [PBL05] wird Variabilität in einem Variabilitätsmodell, das von den Artefakten getrennt ist, modelliert. Zwischen dem OVM und Artefakten werden explizit Verbindungen gezogen. Ist eine Variante nicht ausgewählt, werden die dazugehörigen Artefakte entfernt. Annotative Ansätze sind grundsätzlich sprachunabhängig, jedoch wird Variabilität monolithisch und nicht modular beschrieben. Damit sind sie eingeschränkt für das SPES-Meta-Modell verwendbar.

Kompositionale Ansätze (*compositional approaches*) zerlegen die Variabilität eines Systems in geeignete Systemfragmente. Sie verknüpfen diese Fragmente mit Produktfeatures, so dass die Fragmente zu einer bestimmten Featurekonfiguration durch den gewählten Kompositionsmechanismus zusammengesetzt werden können. Dabei werden z.B. aspektorientierte Mechanismen [HW07, NK08, VG07] zur Modellierung von Variabilität verwendet. Im feature-oriented model-driven development (*FOMDD*) [TBD07] werden Modellfragemente, die mit einem Produktfeature verknüpft sind, zu Featuremodulen zusammengefasst. In diesen Featuremodulen können Modellierungselemente hinzugefügt oder verfeinert werden. Für eine bestimmte Featurekonfiguration werden die entsprechenden Featuremodule nach den Prinzipien der schrittweisen Verfeinerung [BSR04] kombiniert. Kompositionale Ansätze beschränken die Modellierung der Variabilität auf den gewählten Kompositionsmechanismus. Sie sind daher nicht sprachunabhängig und ebenfalls nur eingeschränkt zur Erweiterung des SPES-Meta-Modells geeignet.

Bei transformationalen Ansätzen (*transformational approaches*) wird die Variabilität eines Systems durch die Transformation eines Basismodells repräsentiert, um eine Produktvariante zu erhalten, wie z.B. in der Common Variability Language (*CVL*) [Ha08]. Der in diesem Beitrag verwendete Ansatz der Delta-Modellierung ist ebenfalls ein transformationaler Ansatz. Ein Beispiel für die Delta-Modellierung von Softwarearchitekturen ist die Architekturbeschreibungssprache Δ-MontiArc [Ha11a, Ha11b]. Transformationale Ansätze, wie die Delta-Modellierung, sind modular und sprachunabhängig, so dass diese sich ausgezeichnet für die Erweiterung des SPES-Meta-Modells eignen.

3 Delta-Modellierung im SPES Meta-Modell

Die Delta-Modellierung zur Variantenbeschreibung lässt sich aufgrund ihrer Sprachunabhängigkeit und ihrer Modularität konzeptuell sehr gut in das SPES Meta-Modell integrieren. Dabei haben wir das Meta-Modell so erweitert, dass Deltas genutzt werden können, um Variabilität innerhalb einer Perspektive und über Perspektiven hinweg zu beschreiben (siehe Abbildung 1). Da innerhalb der SPES-Matrix zwischen einzelnen Artefakten unterschiedlicher Abstraktionsebenen und auch zwischen Artefakten unterschiedlicher Perspektiven Abhängigkeiten bestehen können, ist es notwendig, diese nach Anwendung von Modifikationen durch Deltas konsistent zu halten. Dies bedeutet, dass für jede Modifikation, die Elemente verändert, entfernt oder einführt, die in Abhängigkeit zu Elementen anderer Artefakte stehen, entsprechende Modifikationen auch an den jeweils anderen Artefakten durchgeführt werden müssen, um das gesamte System konsistent zu halten.

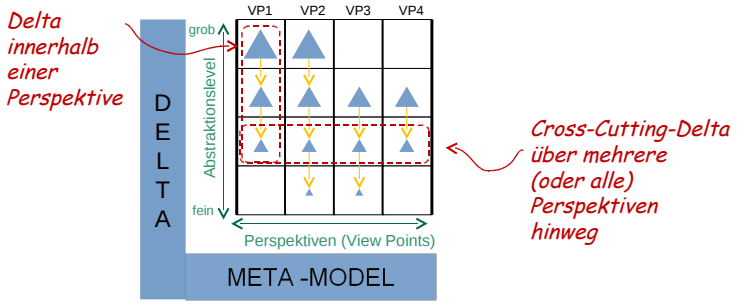

Abbildung 1: Delta-Meta-Modell als Querschnitt im SPES-Meta-Modell

Jede Perspektive der SPES-Matrix kann durch Konzepte der Delta-Modellierung erweitert werden, um Deltas für die jeweiligen Artefakte der Perspektive, aber auch Deltas, welche sich auf unterschiedliche Artefakte einer oder mehrerer Perspektiven gleichzeitig auswirken, zu definieren. Dies erlaubt auch Referenzen zwischen Artefakten und zwischen Perspektiven zu modifizieren. Dabei wird die Variabilität für alle Perspektiven konzeptionell gleichartig beschrieben. Die Variabilität unterschiedlicher Perspektiven kann in einem gemeinsamen Delta (siehe Abschnitt 4) zusammengefasst werden, um gleichzeitig Systemvarianten für unterschiedliche Perspektiven zu erzeugen.

Abbildung 2 zeigt das Delta-Meta-Modell (weiße Klassen, links), welches die syntaktische Struktur von Deltas vorgibt. Jedes Delta wird in einer der beiden Unterklassen der abstrakten Klasse `DeltaModel` definiert.

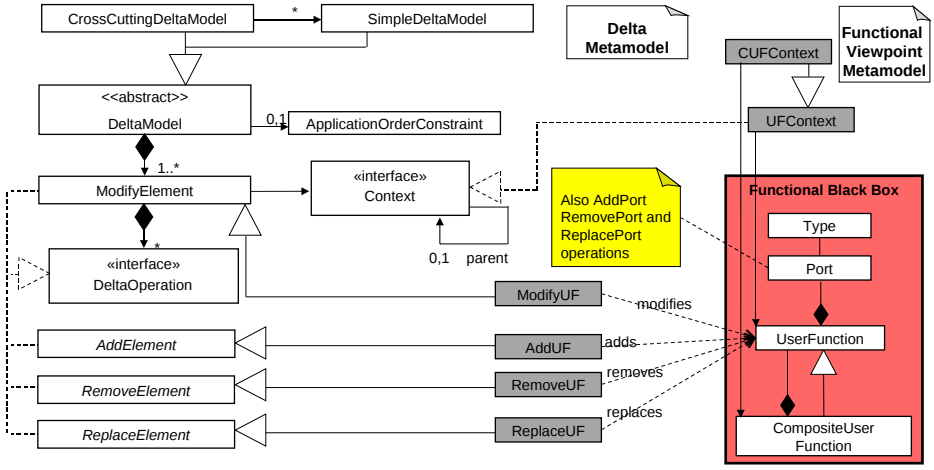

Abbildung 2: Delta-Meta-Modell (l.) und beispielhafte Erweiterung des funktionalen VPs

Deltas innerhalb einer Perspektive werden in `SimpleDeltaModel` und Deltas über

mehrere Perspektiven hinweg in `CrossCuttingDeltaModel` spezifiziert. Jedes Delta enthält mindestens ein `ModifyElement`, welches den zu modifizierenden Kontext (`Context`) referenziert. Ein Kontext gehört zu Elementen, die innere Elemente enthalten können, und kann ebenfalls hierarchisch dekomponiert sein. Auf diese Weise ist es möglich, innere Elemente einer Hierarchie zu modifizieren (zum Beispiel durch Löschen oder Hinzufügen). Modifikationen des Kontexts sind als `DeltaOperations` spezifiziert und beinhalten das Hinzufügen (`AddElement`), Löschen (`RemoveElement`), Ersetzen (`ReplaceElement`) und Modifizieren (`ModifyElement`) von Elementen. Des Weiteren kann ein Delta einen `ApplicationOrderConstraint` (AOC) enthalten, um explizit Abhängigkeiten zwischen Deltas zu modellieren. Dieser wird in Fällen benötigt, in denen Deltas dasselbe (Kern-)Modell modifizieren oder Produktfeatures realisieren, die voneinander abhängen. Ein `CrossCuttingDeltaModel` enthält zusätzlich `SimpleDeltaModels` (siehe Abschnitt 4).

4 Exemplarische Erweiterung von ausgewählten SPES-Perspektiven

Deltas innerhalb eines Viewpoints Die Erweiterung des Meta-Modells um Konzepte der Delta-Modellierung innerhalb einer Perspektive wird anhand der funktionalen Perspektive beispielhaft am (vereinfachten) Functional-Black-Box Modell [Po12] beschrieben (siehe 2). Diese Konzepte können ebenfalls für andere Modelle innerhalb der funktionalen und auch anderer Perspektiven angewendet werden. Solche Deltas werden durch `SimpleDeltaModel`-Elemente modelliert (siehe Abbildung 2).

Wir schlagen folgende (grobe) Vorgehensweise zur Erweiterung einer Modellsprache *M* um Konzepte der Delta-Modellierung vor:

1. Für jedes konzeptuelle Sprachelement *E* einer Modellsprache *M* ist zu bestimmen, ob es modifizierbar ist bzw. sein soll (weiter mit *(2)*) oder nicht (weiter mit *(4)*). Beispiel: In Abbildung 2 ist das Element `Type` nicht modifizierbar[1]. Ein `Type` kann somit nicht über Delta-Operationen ersetzt, gelöscht oder hinzugefügt werden, zumindest nicht direkt (siehe unten).

2. Soll *E* modifizierbar sein, müssen die Operationen (typischerweise Einfügen, Entfernen oder Ersetzen) festgelegt werden, die auf *E* anwendbar sind. Soll *E* ersetzbar sein, ist eine Unterklasse von `ReplaceElement` zu erstellen, welche die Ersetzen-Operation für *E* darstellt. Analog gilt dies für die Hinzufügen- (`AddElement`) und Löschen- (`RemoveElement`) Operationen. Beispiel: Das Element `Port` aus Abbildung 2 ist beispielsweise ersetzbar, weshalb es ein entsprechendes Element `ReplacePort` gibt. Handelt es sich bei *E* um ein komplexer aufgebautes Element, das heißt, ein Element, welches weitere Elemente enthalten kann, weiter mit *(3)*, ansonsten weiter mit *(4)*.

3. Sollen die inneren Elemente eines komplexeren Sprachelements *E* nicht explizit modifiziert werden (ohne dabei *E* komplett austauschen zu müssen), weiter mit *(4)*. Bei-

[1] Dies ist lediglich eine Designentscheidung und soll als Beispiel zum Verständnis des Ansatzes beitragen.

spiel: Dies kann in Abbildung 2 bei `Port` der Fall sein. Ein `Port` hat zwar einen `Type`, allerdings wurde aus methodischen Überlegungen entschieden, dass dieser nicht ausgetauscht werden können soll. Um den Typ eines Ports auszutauschen, muss somit der ganze Port über `ReplacePort` ausgetauscht werden. Ist eine Modifikation innerer Elemente möglich, wird für *E* ein Kontext *C* erstellt, welcher das Interface `Context` aus Abbildung 2 implementiert. Besitzt eine Unterklasse *E_Sub* von *E* weitere - nicht in *E* enthaltene - modifizierbare Elemente, muss ebenfalls ein Kontext *C_Sub* für *E_Sub* erstellt werden, welcher eine Unterklasse von *C* ist. Zusätzlich ist die Erstellung einer Unterklasse von `ModifyElement` notwendig, welche die Modifikations-Operation für *E* darstellt. Zusätzlich kann, beispielsweise über OCL-Constraints, festgelegt werden, welche Delta-Operationen innerhalb des Kontextes *C* erlaubt sind.

Beispiel: In Abbildung 2 sind solche hierarchische Elemente `UserFunction` und `CompositeUserFunction`. Letzteres ist eine Unterklasse des Ersteren und kann zusätzlich zu Ports andere (`Composite-`)`UserFunctions` enthalten. Diese Vererbungsstruktur wird ebenfalls in den beiden Kontexten `UFContext` und `CUF-Context` widergespiegelt. Die Modifikations-Operation `ModifyUF` eignet sich sowohl für `UserFunction` als auch für `CompositeUserFunction`. Eine eigene Modifikations-Operation für `CompositeUserFunction` ist nicht notwendig, da abhängig vom aktuellen Kontext (`UFContext` oder `CUFContext`) mittels Constraints bestimmt werden kann, ob Delta-Operationen zum Hinzufügen (`Add-UF`), Löschen (`RemoveUF`) und Ersetzen (`ReplaceUF`) von `UserFunctions` erlaubt sind (wenn `CUFContext`) oder nicht. Entsprechende Delta-Operationen gibt es für die Ports einer `UserFunction`.

4. Gibt es weitere Sprachelemente in *M*, welche noch nicht betrachtet wurden, weiter mit *(1)*, ansonsten ist die Spracherweiterung fertiggestellt.

Zu beachten ist, dass dieses grobe Verfahren unbedingt ergänzt werden sollte um spezifische Überlegungen, die in weiteren Operatoren münden. Zum Beispiel sind Refactoring-artige Operatoren sinnvoll, die an mehreren Stellen gleichzeitig Typen ersetzen und so per Konstruktion Konsistenz sichern. Weitere Operatoren könnten die Expansion oder Kapselung von Teilstrukturen ermöglichen, etc.

Im obigem Ansatz bleibt die Basissprache *M* bei der Erweiterung unverändert, stattdessen werden für die Spracherweiterung im Meta-Modell Adaptoren (zum Beispiel `UFContext`) verwendet, um eine Verknüpfung zwischen dem Delta-Meta-Modell und dem Basis-Meta-Modell herzustellen. Das hat unter anderem den Vorteil, dass für die Basismodelle keine Abhängigkeiten zu den Delta-Modellen entsteht.

Cross-Cutting-Deltas Im vorherigen Abschnitt wurde beschrieben, wie das SPES Meta-Modell für einen Modelltyp innerhalb einer Perspektive um Delta-Konzepte erweitert werden kann. Dieser Abschnitt beschäftigt sich mit der Modellierung von Cross-Cutting-Deltas. Charakteristisch für solche Deltas ist, dass sie sich in der Regel auf zwei oder mehrere Artefakte unterschiedlichen Modelltyps beziehen. Somit eignen sie sich sowohl für Artefakte unterschiedlichen Typs innerhalb derselben Perspektive, aber auch für die

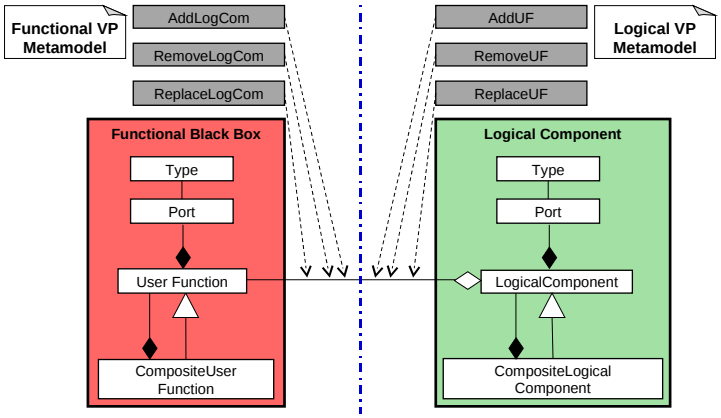

Abbildung 3: Cross-Cutting Delta am Beispiel zweier Perspektiven

Modellierung von Abhängigkeiten in unterschiedlichen Perspektiven. Im Folgenden werden Cross-Cutting-Deltas über mehrere Perspektiven hinweg am Beispiel der funktionalen und logischen Perspektive des SPES Meta-Modells beschrieben.

In Abbildung 3 existiert zwischen `LogicalComponent` aus der logischen Perspektive und `UserFunction` aus der funktionalen Perspektive eine Aggregationsverbindung. Die Bedeutung der gestrichelten Pfeile ist analog zu Abbildung 2. So stellt `AddLogCom` die Verbindung seitens der `UserFunction` her. Die Modifikation der Aggregation betrifft beide Perspektiven. Wird beispielsweise die Aggregation aus `LogicalComponent` entfernt, so muss diese auch aus `UserFunction` entfernt werden, um beide Artefakte konsistent zu halten. Dies wird durch ein `CrossCuttingDeltaModel` (siehe Abbildung 2) mittels entsprechenden Delta-Operationen für die einzelnen Perspektiven (beziehungsweise Artefakte) sichergestellt. Das heißt, ein `CrossCuttingDeltaModel` bündelt (neben den `SimpleDeltaModel`s, siehe Abbildung 2) für jede Perspektive die notwendigen Delta-Operationen, um ein Delta über mehrere Perspektiven umzusetzen. In Abbildung 3 wird zum Entfernen der Aggregation die Delta-Operation `RemoveLogCom` auf der funktionalen Perspektive und `RemoveUF` auf der logischen Perspektive benötigt. Die Delta-Operationen `Add-`, `Remove-` und `ReplaceLogCom`, sowie `Add-`, `Remove-` und `ReplaceUF` erben entsprechend der Meta-Modelle in den vorherigen Abschnitten von den Klassen `Add-`, `Remove-` und `ReplaceElement` des Delta-Meta-Modells (aus Platzgründen nicht abgebildet). Ein Cross-Cutting-Delta sorgt dafür, dass diese beiden Deltas auch existieren, da der Zustand inkonsistent wird, wenn nur in einer Perspektive die Aggregation entfernt wird.

Bevor die Cross-Cutting-Deltas bei der Variantengenerierung angewendet werden, werden zunächst alle Deltas, die nur eine Perspektive betreffen (`SimpleDeltaModel`), angewendet, so dass die einzelnen Perspektiven für sich in einem konsistenten Zustand sind. Anschließend folgen die Cross-Cutting-Deltas, um die Konsistenz zwischen den Perspektiven sicherzustellen.

5 Beispiel

Im Folgenden wird das Erstellen einer neuen Variante über mehrere Perspektiven hinweg exemplarisch dargestellt. Das zugrunde liegende Szenario ist die Modellierung zweier Varianten einer Steuerung der Innenraum-Beleuchtung für Autos. Dabei ist die Komplexität der Anforderungen gering gehalten, um die Anwendung von Cross-Cutting-Deltas, bzw. SimpleDeltas, anschaulich darzustellen. Das Kernmodell beschreibt die einfachste Variante der Steuerung, welche es nur ermöglicht, das Licht innerhalb des Autos mittels Schalter ein- und auszuschalten. Die zweite Variante, welche das Licht auch einschaltet, wenn eine Autotür geöffnet ist, wird durch entsprechende (Cross-Cutting-)Deltas modelliert. Sowohl das Kernmodell, als auch die Variante, welche durch Anwendung der Deltas entsteht, sind in Abbildung 4 dargestellt. Dabei sind alle hinzugefügten Elemente blau gekennzeichnet. Elemente, die durch ein Delta entfernt wurden, sind rot gekennzeichnet.

Innerhalb der Anforderungsperspektive wurde jeder Variante ein `HardGoal` zugeordnet, welches durch jeweils eine `UserFunction` realisiert ist. Hierbei entsteht schon die erste Beziehung zwischen zwei Perspektiven: ein `HardGoal` ist in der funktionalen Perspektive als eine `UserFunction` realisiert. Eine weitere Beziehung zwischen zwei Perspektiven entsteht durch die Realisierung einer `UserFunction` durch entsprechende logische Komponenten. Um die Variante `InLightCtrl_DS` einzuführen, ist es wegen dieser Beziehungen zwischen den Perspektiven notwendig, ein Cross-Cutting-Delta zu definieren. Dieses führt in einem ersten Schritt die zugehörigen Modifikationen in jeder Perspektive durch (einfache Deltas, grau unterlegt), um anschließend Deltas anzuwenden, welche die Beziehungen zwischen den Perspektiven modifizieren (Cross-Cutting-Deltas, hellgrau unterlegt). Das Hinzufügen des Goals `InLightCtrl_DS`, welches die angesprochene Anforderung definiert, und dessen Abhängigkeit zum Goal `InLightCtrl`, führt zu einer notwendigen Umstrukturierung innerhalb der funktionalen Perspektive, da die gegebene Funktionalität nun durch eine Komposition zweier Funktionen definiert wird.

Innerhalb der logischen Perspektive wird die definierte Funktionalität durch eine Modifikation der `InLightCtrl` Komponente realisiert. Zur Veranschaulichung ist in Listing 1 der Pseudocode des zugehörigen Deltas angegeben. In Zeile 2 wird die zu modifizierende logische Komponente angegeben und in den folgenden Zeilen werden zuerst notwendige Ports und Komponenten hinzugefügt, um dann abschließend zugehörige Verbindungen

Delta-MontiArc

```
1  delta DoorState {
2      modify component InLightCtrl {
3          add port in Boolean doorStatus;
4          add component Or or;
5
6          connect SwitchStatus -> or.inputA;
7          connect DoorStatus -> or.inputB;
8          connect or.output -> LightCmd;
9      }
10 }
```

Listing 1: Exemplarische Modifikation der Komponente `InLightCtrl` durch ein Delta.

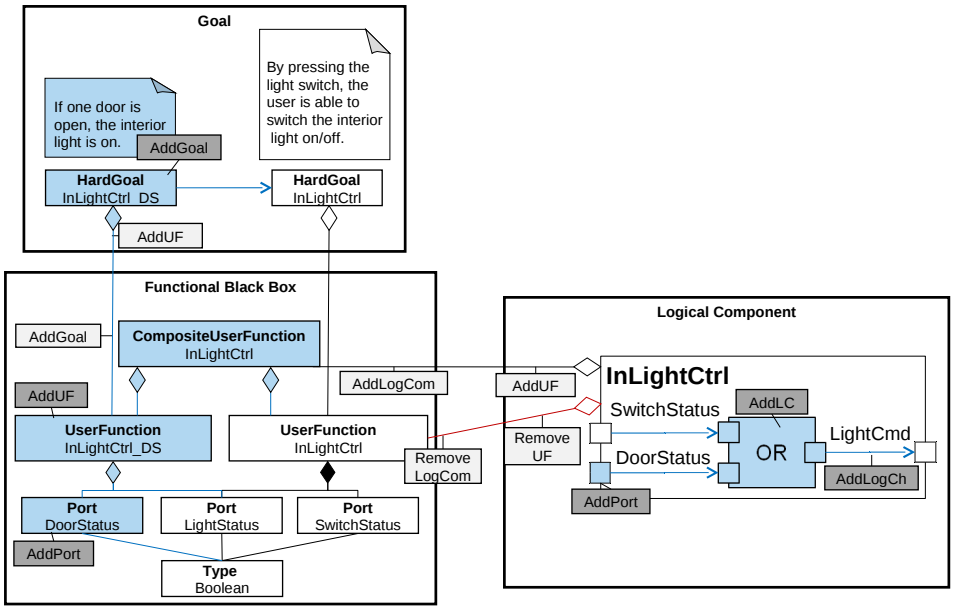

Abbildung 4: Ein Cross-Cutting-Delta über drei Perspektiven.

zu etablieren. Dieses Delta kann grundsätzlich losgelöst von den jeweils anderen Deltas definiert und auch in anderem Kontext verwendet werden.

6 Zusammenfassung

In diesem Artikel wurde ein Ansatz vorgestellt, das mehrperspektivische SPES Meta-Modell um Aspekte der Variabilitätsmodellierung durch das Konzept der Delta-Modellierung zu erweitern. Dabei wurde ein allgemeines Delta-Meta-Modell eingeführt, welches zwischen Deltas innerhalb einer Perspektive (bzw. Modelltyps) und Deltas über mehrere Perspektiven (bzw. Modelltypen) hinweg unterscheidet. Das allgemeine Delta-Meta-Modell kann durch Hinzufügen von Adaptoren für die einzelnen Sprachen des SPES Meta-Modells angepasst werden. Das Konzept wurde anhand eines kurzen Beispiels veranschaulicht. Es ist geplant, dass Konzept an einem größeren Beispiel zu evaluieren, um die Komplexität beim Variantenmanagement mittels Delta-Modellierung zu untersuchen.

Literatur

[BSR04] Don S. Batory, Jacob Neal Sarvela und Axel Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Software Eng.*, 30(6):355–371, 2004.

[CHS10] D. Clarke, M. Helvensteijn und I. Schaefer. Abstract Delta Modeling. In *GPCE*. Springer, 2010.

[Cz05] Krzysztof Czarnecki. Mapping features to models: A template approach based on superimposed variants. In *GPCE 2005*, Seiten 422–437. Springer, 2005.

[Go05] Hassan Gomaa. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading, 2005.

[Gr08] Hans Grönniger, Jochen Hartmann, Holger Krahn, Stefan Kriebel, Lutz Rothhardt und Bernhard Rumpe. View-Centric Modeling of Automotive Logical Architectures. In *Tagungsband des Dagstuhl-Workshops MBEES: Modellbasierte Entwicklung eingebetteter Systeme IV*, 2008.

[Ha08] O. Haugen, B. Moller-Pedersen, J. Oldevik, G.K. Olsen und A. Svendsen. Adding Standardized Variability to Domain Specific Languages. In *Software Product Line Conference, 2008. SPLC '08. 12th International*, Seiten 139 –148, sept. 2008.

[Ha11a] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe und Ina Schaefer. Delta-oriented Architectural Variability Using MontiCore. In *ECSA '11 5th European Conference on Software Architecture: Companion Volume*, New York, NY, USA, September 2011. ACM New York.

[Ha11b] Arne Haber, Holger Rendel, Bernhard Rumpe und Ina Schaefer. Delta Modeling for Software Architectures. In *Tagungsband des Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteterSysteme VII*, Seiten 1 – 10, Munich, Germany, February 2011. fortiss GmbH.

[HW07] F. Heidenreich und C. Wende. Bridging the gap between features and models. *Aspect-oriented Product Line Engineering*, Seite 38, 2007.

[Me07] A. Metzger, P. Heymans, K. Pohl, P.-Y. Schobbens und G. Saval. Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis. In *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*, Seiten 243 –253, oct. 2007.

[NK08] N. Noda und T. Kishi. Aspect-Oriented Modeling for Variability Management. In *Software Product Line Conference, 2008. SPLC '08. 12th International*, Seiten 213 –222, sept. 2008.

[PBL05] Klaus Pohl, Günter Böckle und Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[Po12] Klaus Pohl, Harald Hönninger, Reinhold Achatz und Manfred Broy. *Model-Based Engineering of Embedded Systems: The SPES 2020 Methodology*. Springer, 2012.

[Sc10] I. Schaefer. Variability Modelling for Model-Driven Development of Software Product Lines. In *Intl. Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010)*, 2010.

[TBD07] S. Trujillo, D. Batory und O. Diaz. Feature Oriented Model Driven Development: A Case Study for Portlets. In *ICSE 2007*, Seiten 44 –53, may 2007.

[VG07] Markus Voelter und Iris Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *Proceedings of the 11th International Software Product Line Conference*, SPLC '07, Seiten 233–242, Washington, DC, USA, 2007. IEEE Computer Society.