Systematische Rekonfiguration eingebetteter softwarebasierter Fahrzeugsysteme auf Grundlage formalisierbarer Kompatibilitätsdokumentation und merkmalbasierter Komponentenmodellierung

Peter Manhart

RD/EDS
Daimler AG
Wilhelm-Runge-Straße 11
89081 Ulm
Peter.Manhart@daimler.com

Abstract: Mit dem steigenden Wertschöpfungsanteil SW-basierter Funktionen in Fahrzeugen und schnelleren Releasezyklen bei wachsender Variantenkomplexität steigt der Bedarf an praxistauglichen Lösungen zum nachträglichen Austausch minimaler Teilreleases der zugrundeliegenden SW-Komponenten. Wir stellen einen merkmalbasierten Modellierungsansatz für die Komponenten von Steuergerätereleases vor, der auf einer formalisierbaren Kompatibilitäts-dokumentation von Release-Komponenten basiert. Durch zudem klar definierte Abbildungen der Kompatibilitätsaussagen in ein merkmalbasiertes Varianten-modell ermöglicht unser Ansatz eine systematische Rekonfiguration SW-basierter Fahrzeugsysteme.

1. Einleitung

Wettbewerbsrelevante Fahrzeugfunktionen mit hohem Wertschöpfungsanteil, wie zum Beispiel Fahrerassistenzsysteme, sind immer häufiger softwarebasierte Funktionen. Sie werden immer schneller weiterentwickelt und unterschiedlichen Baureihen bereitgestellt. Im Falle von Funktionserweiterungen oder der Anpassung an sich wandelnde gesetzliche Anforderungen stellt sich die Aufgabe der Rekonfiguration: Welche minimale Teilkonfiguration eines Systems muss ausgetauscht werden, um eine Änderung konsistent umzusetzen.

Der hier dargestellte Lösungsansatz entstand im Umfeld der Integration von HW- und SW-Komponenten für ein Kombiinstrument in der Fahrzeugindustrie. In dem Bereich werden nur als wettbewerbsrelevant eingestufte oder stark baureihenspezifische Teilfunktionen intern entwickelt. Die Hardware und die Basissoftware kommen vom Zulieferer. Der Quellcode zugelieferter SW-Komponenten ist in der Regel für die interne Funktionsentwicklung nicht verfügbar.

In der Ausgangssituation waren in dem Entwicklungsbereich folgende, für unsere Betrachtung relevante Entwicklungspraktiken etabliert:

1 Eine Menge systematischer *Anforderungsspezifikationen* für die Komponenten und deren Zusammenspiel.

2 Ein *Versionsmanagement*, das die inkrementelle Weiterentwicklung aller relevanten Entwicklungsartefakte wie Spezifikationen, Lösungen in Form von Modellen, HW-Merkmale und SW-Komponenten, Buildprozessen etc. festhält und dokumentiert.

3 Ein *Releasemanagement* im Kontext des Versionsmanagements, das die Nachverfolgbarkeit von sich zeitlich weiterentwickelnden Releases (Freigabekonfigurationen) sicherstellt und Bausteinversionen zu Konfigurationen gruppiert und diesen einen definierten Reifegrad und einen eindeutigen Bezeichner zuordnet.

Diese Praktiken ermöglichen es bereits (1) herauszufinden, welche Komponentenversionen existieren, (2) zu rekonstruieren, welche Komponentenversionen in welche Releases eingeflossen sind und (3) diese Releases nachzubilden.

Zunächst wurde in dem Bereich merkmalbasiertes Variantenmanagement auf Grundlage der bei uns in der Serieentwicklung bewährten Methode und Werkzeugkette eingeführt. Damit können die Eigenschaften der Komponentenvarianten und ihrer Variationspunkte modelliert und die Konfiguration automatisiert werden. Jedoch wird damit noch nicht die minimale Änderung historischer Konfigurationen unterstützt.

1.1 Rekonfiguration

Wir verstehen unter *Rekonfiguration* den Austausch von Teilreleases in historischen Konfigurationen. Eine Darstellung der Problemstellung in Abgrenzung zum Reengineering findet sich in einer früheren Arbeit [Ma05].

Klassische Konfigurations- und Variantenmanagementansätze, z.B. der CM-Prozess im CMMI-Referenzmodell fordern lediglich die Abbildung von Abhängigkeiten „at given points in time“ [CKS06], also innerhalb einer Konfiguration. Auch der Feature Modeling Process in Kapitel 4.9 von [CE00] beschreibt die Merkmalmodellierung der aktuellen Konfiguration. Für Rekonfiguration müssen jedoch auch Abhängigkeiten zwischen Vorgänger- und Nachfolgerversionen dokumentiert werden. Fehlen diese, wird lediglich der Austausch kompletter Releases durch Nachfolger- / Vorgänger-Releases systematisch unterstützt, aber nicht der Austausch minimaler Teilkonfigurationen. Das ist jedoch selten praktikabel, da beispielsweise aufgrund von HW-Einbauabhängigkeiten oder hoher HW-Tauschkosten der Austausch von HW oft unmöglich oder zu teuer ist. Aber auch der Austausch kompletter SW kann unmöglich oder unwirtschaftlich sein, da die Basis-SW oft nicht extern flashbar verbaut ist oder auch im Falle flashbarer SW ein kompletter Flashvorgang zu lange dauert.

Der hier dargestellte Ansatz zum Kompatibilitätsmanagement zielt darauf ab, dasjenige, möglichst minimale Software-Teilrelease zu ermitteln, das für einen bestimmten Anwendungsfall notwendigerweise ersetzt werden muss. Im folgenden Abschnitt werden unsere Anwendungsfälle und Anforderungen dargestellt, die in unserem Kontext die unmittelbare Anwendung bestehender Rekonfigurationsansätze erschweren.

2. Anwendungsfälle, Anwendungskontext und dessen Bedingungen für Praxistauglichkeit

Unter Abstraktion von Sonderfällen, die mit abgedeckt werden, lassen sich im betrachteten Kontext die folgenden Anwendungsfälle unterscheiden:

2.1 Fall Funktionsänderung: Folge von Spezifikationsänderungen

Bei einer Funktionsänderung wird zunächst die Anforderungsspezifikation einer Komponente weiterentwickelt. Als Folge muss die Umsetzung der Komponente angepasst werden. Dies muss nicht, kann aber in einem größeren oder kleineren Funktionsumfang, sowie in Änderungen der Schnittstelle der Komponente resultieren.

Aus dem Blickwinkel der Rekonfiguration von Komponenten stellt sich die Frage, welche Auswirkungen eine Implementierungsänderung einer Komponente für den Fall hat, dass in einer historischen Konfiguration eine Vorgängerversion der Komponente ersetzt werden muss.

2.2 Fall Fehlerbeseitigung (Bugfix): Austausch fehlerhafter Komponenten

Im Rahmen eines Bugfix bleibt normalerweise die Spezifikation der Komponente stabil, aber die Implementierung muss aufgrund eines von der Spezifikation abweichenden Verhaltens angepasst werden.

Aus Sicht der Rekonfiguration stellt sich die Frage, in welcher Vorgängerversion der Komponente der Fehler zuerst vorhanden war und ob sich diese möglicherweise bereits in der Anwendung befindliche Komponente isoliert austauschen lässt oder ob zusätzliche Maßnahmen nötig sind.

2.3 Anwendungskontext Kombisteuergerät in der Fahrzeugindustrie

Die flashbare und damit prinzipiell austauschbare Software des zugrundeliegenden Kombisteuergerätes besteht aus acht Komponenten. Zwei SW-Komponenten werden intern spezifiziert aber extern entwickelt; der Quellcode ist dem OEM daher nicht verfügbar:

- Die Basisanwendung *APPL* implementiert die Basis-SW incl. Kommunikations-SW, sowie Funktionen zur Ansteuerung von akustischen und optischen Signalen.
- Die Signalschnittstelle *DML* implementiert eine Netzwerkabstraktion, so dass die Funktions-SW netzwerkunabhängig aufgebaut werden kann.

Für sechs Komponenten liegt auch der Quellcode vor, da sie intern entwickelt werden:

- Die Warndatenbank *SMF* enthält länderspezifische Warnmeldungen.
- Die Komponente *GDE* implementiert eine Statechart-basierte Ansteuerung der im Kombiinstrument eingebauten Displaymatrix.
- Die Datenbank *IMG* stellt länderspezifische Symbole bereit.
- Die Datenbank *LNG* stellt länderspezifische Texte bereit.

- Die modellbasiert entwickelte Funktionskomponente *TC* implementiert Fahrerinformationen.
- Die modellbasierte Funktionskomponente *EDF* unterstützt CO_2-sparendes Fahren.

Abhängigkeiten der SW-Komponenten nach außen bestehen beispielsweise zu Varianten der Grafikdisplays oder zu der baureihenspezifischen Signalwelt der Fahrzeugnetzwerke. Inkompatible HW- und SW-Versionen, sowie die Dauer eines kompletten Flashvorganges führen zur Notwendigkeit des Austausches von Teilreleases.

Ein konkretes Beispiel für baureihenbedingte Konfigurationsabhängigkeiten wäre:
- APPL implementiert u.a. die Netzwerkschnittstelle und stellt den anderen Komponenten baureihenspezifische Signale zur Verfügung.
- DML implementiert eine Abstraktion von konkreten, baureihenspezifischen Netzwerkssignalen und stellt diese Signale APPL und anderen SW-Komponenten zur Verfügung. DML hängt damit von APPL ab.
- APPL implementiert auch den Zugriff auf HW wie Leuchten und Zeiger auf Basis der von DML abstrahierten Signale. Damit hängt APPL von DML ab.
- Ein neues Netzwerksignal in APPL würde beispielsweise von DML abstrahiert werden. Eine darauf aufbauende Anzeigefunktion von APPL könnte beispielsweise nicht funktionieren, wenn DML das Signal nicht berücksichtigt.

Die Schnittstelle, die APPL für DML bereitstellt, bezeichnen wir als APPL>DML, die von APPL benötigte entsprechend als DML>APPL.

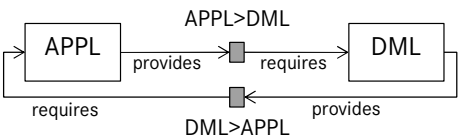

Abbildung 1: Schnittstellen zwischen APPL und DML

Die Schnittstellen fassen wir als *Konfigurationsschnittstellen* auf, sie repräsentieren die äußeren Abhängigkeiten einer Komponente, um diese getrennt vom Funktionsumfang und Semantik ihrer inneren Funktionsimplementierungen zu betrachten.

Der nächste Abschnitt skizziert die in unserem Kontext wesentlichen praktischen Voraussetzungen für Ansätze zum Kompatibilitätsmanagement.

2.4 Bedingungen für Praxistauglichkeit

Über die prinzipielle Lösung der oben genannten Anwendungsfälle hinaus stellt unser Umfeld folgende zusätzliche Anforderungen an eine tragfähige Lösung:

1 Der Austausch einer Teilkonfiguration kann heikel sein, weil bei dem versehentlichen Ersatz einer Komponente durch eine inkompatible Alternative Funktionsstörungen auftreten können. Ein praktikabler Lösungsansatz sollte darum einen zu ersetzenden Teilreleaseumfang hinsichtlich Kompatibilität nachvollziehbar für den Verantwortlichen machen. In unserem Kontext wurde gefordert, die Kompatibilität von Komponentenversionen mit den Eigenschaften der Funktionsänderungen von Komponenten zu begründen.

2 In unseren Fahrzeugsystemen werden viele von Zulieferern entwickelte SW-Komponenten integriert. Da wir keinen Zugriff auf deren Quellcode haben, könnten wir Komponentenabhängigkeiten nur unvollständig aus Quellcodeanalysen ableiten.

3 Die Dokumentation von Komponentenabhängigkeiten ist komplex. Ein praktikabler Lösungsansatz darf darum nicht fordern, globale Aussagen über historische Zusammenhänge zu machen, sondern muss es erlauben, diese zeitnah bei Änderungen im Rahmen der Entwicklung auf Grundlage des momentanen Wissenstandes des Entwicklers zu dokumentieren. Aus der Menge lokal und zeitnah dokumentierter Abhängigkeiten muss sich dann ein global funktionierender Ansatz für das Rekonfigurationsproblem ergeben.

4 Wenige unserer Funktionsentwickler haben Erfahrungen in der Anwendung formaler Softwareentwicklungsmethoden. Ein praktikabler Ansatz muss diesen Funktionsentwicklern ermöglichen, belastbare Ergebnisse mit geringer Fehlerquote zu liefern.

5 Die verfügbare Zeit für Dokumentation ist begrenzt. Der Zusatzaufwand für Kompatibilitätsdokumentation darf deren Nutzen nicht übersteigen.

Diese Bedingungen führen dazu, dass der Schwerpunkt unserer Arbeit nicht der eigentliche Rekonfigurationsalgorithmus ist, sondern die Fragestellung, wie in unserem Umfeld vorab das nötige Abhängigkeitswissen in der Funktionsentwicklung systematisch erfasst und für automatische Rekonfiguration formalisiert werden kann.

Bevor wir unseren Lösungsansatz und dessen Umsetzung beschreiben, werden im nächsten Kapitel die von uns verwendeten Begriffe und eine Kurznotation für diese eingeführt.

3. Begriffe und Notationen

In diesem Kapitel werden die uns wesentlichen Begriffe und Schreibweisen kurz eingeführt, um das Verständnis der im Folgenden verwendeten kompakten Notationen zu erleichtern. Im Anhang werden diese ausführlicher beschrieben.

- Eine *HW- oder SW-Komponente* wird als Zeichenkette notiert, z.B. Signaldatenbank oder S.
- Eine *Version* wird als Zahl geschrieben, z.B. 20120213 oder 0.
- Eine *Variante* wird nicht direkt bezeichnet, sondern im Rahmen der Variantenmodellierung durch Beziehungen zwischen Komponentenversionen abgebildet.
- Eine *Komponentenversion* und deren *Funktionsumfang* wird als versionierte Komponente notiert, z.B. Signaldatenbank120213 oder D0.
- Eine (versionierte) Konfigurationsschnittstelle wird als versionierter Schnittstellenbezeichner der Form SigDB>APPL0 bzw. S>A0.
- Ein Release wird im Rahmen der Variantenmodellierung als Merkmalkonfiguration modelliert.

- Unter *Kompatibilität* verstehen wir die Möglichkeiten und Folgen des Austausches zweier Komponentenversionen in Bezug auf bestehende Releasekonfigurationen. *Abwärtskompatibilität* bedeutet, dass Nachfolgerversionen für Vorgänger eingesetzt werden können, *Aufwärtskompatibilität*, dass Vorgängerversionen für Nachfolger eingesetzt werden können. Kompatibilitätsaussagen haben die Form „–„ für aufwärtskompatibel, „+“ für abwärtskompatibel, „0“ für keine Änderung oder voll kompatibel, „X“ für inkompatibel.
  Aussagen zur Kompatibilität des Funktionsumfangs von Komponentenversionen oder Komponentenversionsschnittstellen können durch Kombination der Notationen ausgedrückt werden, z.B. „D1-“ für eine abwärtskompatible Funktionsänderung von D1 in Bezug auf D0.

4. Lösungsansatz

Der hier vorgestellte Ansatz zielt auf einen Austausch von Komponenten oder Teilreleases mit Kontrolle über Funktionsumfang und unter Wahrung der Schnittstellenintegrität. Der Ansatz basiert auf einer Beschreibung von Komponentenversionen in Releases durch ihre Funktionsumfänge und Schnittstellen, sowie der Beschreibung von Kompatibilitätsbeziehungen zwischen Funktionsumfängen und zwischen Schnittstellen.

Der eigenständige Beitrag unseres Ansatzes ist die formalisierbare Erfassung von Abhängigkeitswissen und dessen schematische Abbildung in ein Merkmalmodell. Die Konzepte, mit denen im Merkmalmodell Kompatibilitätsabhängigkeiten modelliert werden, unterscheidet sich von denen in etablierten Konfiguratoren wie dem Debian Package Manager hauptsächlich dadurch, dass die Abhängigkeitsbeziehungen über Komponentenabhängigkeiten hinaus um Schnittstellenabhängigkeiten ergänzt werden. Diese Abhängigkeitsbeziehungen von Schnittstellen entsprechen denen von Architekturbeschreibungssprachen wie Koala [Om00].

Im ersten Schritt des Ansatzes wird die Releasedokumentation um Kompatibilitätsinformation ergänzt. Diese Zusatzdokumentation ist so aufgebaut, dass im zweiten Schritt eine schematisierte Abbildung in das bei uns für merkmalbasierte Variantenmodellierung [FO90] etablierte Modellierungswerkzeug möglich ist.

4.1 Kompatibilitätsinformation als Erweiterung bestehender Releasedokumentation

Da in unserem Umfeld Teile der Komponenten von externen Zulieferern kommen, können wir nicht wie in Koala Schnittstellenabhängigkeiten automatisch aus Quellcode extrahieren. In unserem Umfeld müssen wir die Abhängigkeitsinformationen vor der späteren Formalisierung zunächst manuell dokumentieren.

In der Ausgangssituation wurden die Releases der Softwarekomponente in einer Tabelle dokumentiert. Eine Zeile der Tabelle enthielt eine Spalte für das Releasedatum, eine für den Releasebezeichner und darauffolgend je eine Spalte für den Versionsbezeichner jeder im Release enthaltenen Komponentenversion. Dies wurde durch ein Bemerkungsfeld abgeschlossen, in dem textuelle Hinweise zu dem Release untergebracht werden konnten.

Diese Tabelle wurde im Rahmen des hier vorgestellten Ansatzes zur Erfassung kompatibilitätsrelevanter Zusatzinformation um folgende weitere Spalten ergänzt:

- Eine Spalte, in der für jede Komponentenversion des Releases beschrieben wird, welche Änderungen an den Funktionen neuer Komponentenversionen vorgenommen wurde. Diese Dokumentation war bis dahin unvollständig.
- Für jede veränderte Komponentenversion eine Spalte zur Dokumentation der Kompatibilität von Funktionsumfang und Schnittstellen.

Beispiel: Die neue Signaldatenbankversion D1 wurde gegenüber D0 um ein Signal erweitert und in der Applikationsversion A1 wurde eine Funktion hinzugefügt, die bei Eintreffen des Signales einen Warnton ausgibt. Der Entwickler würde die neue Situation folgendermaßen dokumentieren:

Applikation	Signaldatenbank D	Änderungsdetails	A	D	…	D>A	..
…	…	…	…	…	…	…	..
A1	D1	D: neues berechnetes Signal Bsm A: neuer Warnton links: BsmWl	+	+		+	..

Die Kompatibilitätsaussagen und ihre Begründungen sind:

- A1 ist abwärtskompatibel zu A0, weil die neue Version alle Funktionen der alten fehlerfrei erfüllt.
- D1 ist abwärtskompatibel zu D0, da D1 alle Signale von D0 liefert.
- Die Ausgangsschnittstelle D>A1 von S1 ist gleichzeitig die Eingangsschnittstelle von A1 und zudem abwärtskompatibel zu D>A0, weil dort alle für den Vorgänger von A nötigen Schnittstellenfunktionen, die Signalisierungen, geliefert werden und weil A alle Signalisierungen des Vorgängers gleich verarbeitet.

Das notwendige Detailwissen für Kompatibilitätsaussagen dieser Qualität ist nur zum Zeitpunkt der Änderung vorhanden. Darum muss die Dokumentation zeitnah zur Änderung erfolgen. Im Sinne optimaler Nutzung von Komponentenvariabilität für die Minimierung von zu ersetzenden Teilreleases ist es sinnvoll, zusätzliche Methoden für die Absicherung von Kompatibilitätsaussagen einzuführen. Beispiele dafür wären eigene Tests auf Kompatibilität oder der Einsatz von geeigneten Analysewerkzeugen.

4.2 Konzept für Kompatibilitätsmodellierung auf Grundlage merkmalbasierter Variantenmodellierung

Die Dokumentation von Kompatibilität, wie sie im letzten Abschnitt dargestellt wurde, eignet sich jedoch in der Form nicht direkt für eine maschinelle Auswertung. Darum sieht der dargestellte Ansatz als nächsten Schritt eine Formalisierung der

Kompatibilitätsinformation als merkmalbasiertes Variantenmodell vor. Im Falle der Rekonfiguration ist ein Modell erforderlich, das die Konfigurationshistorie der Produktevolution inklusive der Kompatibilität der Komponentenversionen abbildet. Dazu müssen die einzelnen Evolutionsschritte unmissverständlich auf Änderungen des Merkmalmodells abgebildet werden. In unserem Modellierungsschema sind neun Hauptanwendungsfälle abstrahiert dargestellt. Die Sonderfälle ergeben sich durch Weglassen der für Kompatibilität eingesetzten Relationen `provides` und `requires` an Merkmalen für Funktionsumfänge und Schnittstellen. Die Abbildung von Kompatibilitätsaussagen in ein Merkmalmodell sieht folgendermaßen aus:

- Komponenten und deren Versionen, Schnittstellen und deren Versionen werden als Merkmale abgebildet.
- Abhängigkeiten zwischen Komponenten und Schnittstellen werden als Beziehungen abgebildet. Beispiele:
  A0 hat Eingangsschnittstelle D>A0: `A0 requires D>A0`
  D0 stellt Ausgangsschnittstelle D>A0 bereit: `D0 provides D>A0`
- Abhängigkeiten zwischen Komponenten und Abhängigkeiten zwischen Schnittstellen werden als Merkmalbeziehungen abgebildet. Beispiele:
  Funktionsumfang von A1 ist abwärtskompatibel zu A0: `A1 requires A0`
  Schnittstelle S>A1 ist abwärtskompatibel zu S>A0: `S>A1 requires S>A0`
  Diese Beziehungen werden bei Aufwärtskompatibilität umgekehrt und bei Inkompatibilität weggelassen.

Wir setzen für die folgenden zwei wichtigen Fälle der Evolution ohne Einschränkung der Allgemeinheit voraus, dass die jeweiligen Komponenten- und Schnittstellen in der Version 0 existieren, also dass beispielsweise A0, D0 oder etwa D>A0 der aktuelle Stand ist.

Fall 1: Ein typischer Anwendungsfall wäre eine gleichzeitige abwärtskompatible Funktionserweiterung einer Applikationskomponente von A0 nach A1. Die folgenden Änderungen bilden den Vorgang im Merkmalmodell ab:

1. Im Applikations-Merkmalteilbaum neue Komponentenversion A1 hinzufügen.
2. Der Komponentenversion A1 die Relation „`requires A0`“ hinzufügen.

Fall 2: Ein weiterer möglicher Fall wäre eine neue Komponentenversion einer Signaldatenbankkomponente mit abwärtskompatibler Schnittstellenänderung zur Applikation. Die Abbildung eines abwärtskompatiblen Funktionsumfangs würde analog Fall 1 umgesetzt werden. Die Schnittstellenänderung von D>A0 nach D>A1 würde nach folgendem Schema modelliert:

1. Im DML-Merkmalteilbaum neue Komponentenversion D1 hinzufügen.
2. Im Schnittstellen-Merkmalteilbaum die neue Schnittstellenversion D>A1 hinzufügen.
3. Der Komponentenversion D1 die Relation „`provides D>A1`“ hinzufügen.
4. Der Schnittstellenversion D>A1 die Relation „`requires D>A0`“ hinzufügen.

Mit entsprechenden Schemata ließen sich alle Anwendungsfälle für die Kompatibilitäts-modellierung der Evolution von Funktionsumfang und Schnittstellen abbilden. Die folgenden Abbildungen veranschaulichen den Modellierungsverlauf eines einfachen

Beispiels von Weiterentwicklung. In der Ausgangssituation liegen die Komponenten und ihre Funktionsumfänge in der Version 0 vor (Abbildung 2).

Releases werden als Merkmalkonfiguration abgebildet. Abbildung 2 zeigt auf der rechten Seite die initiale SW-Konfiguration. Verstöße gegen Kompatibilitätsaussagen resultieren in verletzten Beziehungen zwischen den ausgewählten Merkmalen der Konfiguration. Sind alle Beziehungen befriedigt, ist die ausgewählte Konfiguration bei fehlerfreier Modellierung unter Kompatibilitätsgesichtspunkten valide.

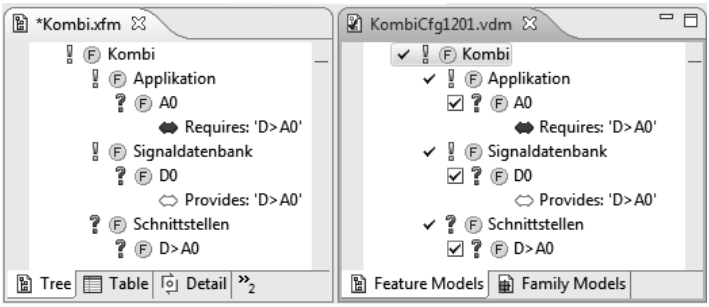

Abbildung 2: Ausgangssituation Modellierung A und D

Die Modellierung der Kompatibilitätsinformation eines Evolutionsschrittes wie im letzten Abschnitt beschrieben, resultiert in einer neuen Version des Merkmalmodelles. Dieses ist in Abbildung 3 auf der linken Seite dargestellt.

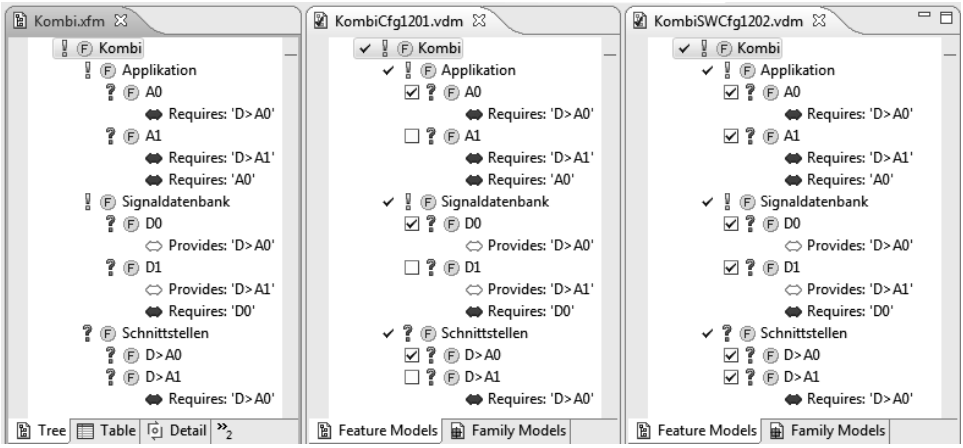

Abbildung 3: Modellierung eines Evolutionsschrittes und einer Nachfolgekonfiguration

Wie man in den Variantenmodellen in Abbildung 3 sieht, verhält sich die Modellierung, wie man es von der Kompatibilitätssituation erwarten würde. Die bisher bestehende Konfiguration `KombiSWCfg1201` ist nach wie vor gültig, die neu mögliche Konfiguration `KombiSWCfg1202` aus `A1` und `D1` ist zusätzlich gültig.

Für diese Ergebnisse hätte eine der bei uns etablierten Variantenmodellierungstechniken ausgereicht. Nicht aber für die Frage, welche der SW-Komponenten in der jetzt

historischen Konfiguration KombiSWCfg1201 ersetzt werden darf. Ein Austausch von D0 durch D1, der zu KombiSWCfg1201_UpgradeD führt, ist kompatibel. Nicht jedoch der isolierte Austausch von A0 gegen A1; in diesem Fall meldet der Variantenmodellierer in KombiSWCfg1201_UpgradeA das Fehlen der Schnittstelle D>A1 (siehe Abbildung 4).

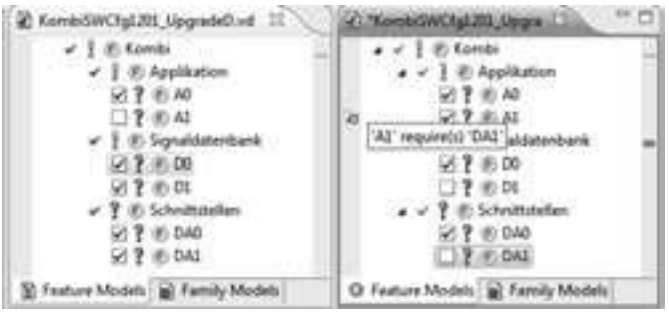

Abbildung 4: Upgradekonfigurationen

5. Prototypische Evaluierung des Konzeptes

Der hier vorgestellte Ansatz wurde für die Software-Rekonfiguration eines Kombiinstrumentes prototypisch umgesetzt. Dieser Anwendungskontext wurde bereits in Kapitel 2 beschrieben.

Während der 12-monatigen Erprobung entstand ein Merkmalmodell für etwa 4 Hauptreleases mit jeweils zwei bis fünf Zwischenreleases. In jedem Release wurden etwa fünf Funktionsänderungen vorgenommen, dokumentiert und modelliert. Dadurch entstand ein merkmalbasiertes Kompatibilitätsmodell mit 62 Merkmalen und 95 Relationen.

Der Dokumentations- und Modellierungsaufwand für die Evaluierung beschränkte sich auf wenige Stunden je Release und wurde als handhabbar akzeptiert. Darum wurde beschlossen, den Ansatz in der Serienentwicklung zu pilotieren und dann im Entwicklungsbereich als Standard zu etablieren.

6. Bestehende Ansätze, Implementierungen und deren Beziehung zu unserem Ansatz

Der Kern unserer Lösung besteht aus drei Bausteinen: Einer Methode zur formalisierbaren Kompatibilitätsdokumentation, einer Struktur zur merkmalbasierten Modellierung von Kompatibilität, sowie einer Systematik der Umsetzung der Dokumentation in die Modellierung. Dabei werden neben den skizzierten Anwendungsfällen alle Use-Cases unter den Bedingungen der Praxistauglichkeit umgesetzt. Im Folgenden werden aus unserer Sicht alternative Herangehensweisen und Arbeiten im Umfeld unseres Ansatzes dargestellt und abgegrenzt.

6.1 Pragmatische Ansätze

Die in der Praxis wahrscheinlich am weitesten verbreitete Herangehensweise ist die textuelle Dokumentation in Texten oder Tabellen. Dabei sichern die Entwickler im Laufe der Entwicklung kompatibilitätsrelevantes Wissen für die spätere Unterscheidung von Komponentenvarianten zur Konfiguration von Systemen. Dieses Wissen enthält auch Anteile, die hilfreich bei der Rekonfiguration historischer Konfigurationen sein können. Da jedoch dieser Vorgehensweise kein formales Modell der Rekonfiguration zugrunde liegt, ist dieses Wissen für einen Fall, den man nicht vorhergesehen hat, regelmäßig unvollständig, so dass nachermittelt oder getestet werden muss. Darüber hinaus sind pragmatische Ansätze sehr stark von individueller Terminologie und individuellem Hintergrundwissen geprägt, so dass die Notizen selbst für Mitglieder des gleichen Teams oder Einsteigern nicht mehr ohne weiteres verständlich sein können. Eine maschinelle Auswertung dieser Informationen ist nicht umsetzbar.

Im Gegensatz dazu liefert der hier vorgestellte Ansatz einen klaren und maschinell auswertbaren Rahmen für die Rekonfiguration. Der Entwickler weiß im Detail, welche Aussagen er im Falle einer neuen Komponentenversion dokumentieren muss und der Modellierer, wie er diese Information in ein maschinelles Rekonfigurationsmodell überführen kann.

6.2 Wissensbasierte Systeme

Es existiert eine Vielzahl von Arbeiten mit dem Ziel der wissensbasierten Konfiguration komplexer strukturierter Produkte. Als einer der ersten praktizierten Ansätze für wissensbasiertes Konfigurieren gilt der regelbasierte Konfigurator XCON, der 1978 für die Konfiguration von DEC-Computern aufgebaut wurde. John stellt in seiner Dissertation [Jo02] einen Ansatz auf Grundlage Constraint-basierter Modellierung über endlichen Domänen vor, der neben Konfigurations-, auch Rekonfigurationsaufgaben adressiert. Mannistö et al. beschreiben [Ma99] eine Methode zur Rekonfiguration, die auf expliziten Rekonfigurationsoperationen und -invarianten beruht. Diese können zum Beispiel aus vergangenen Rekonfigurationen durch fallbasiertes Schließen abgeleitet werden, womit der Vorteil einer gewissermaßen empirischen Absicherung gegeben wäre. Das Problem der technischen Begründbarkeit der Gültigkeit wird allerdings nicht explizit abgedeckt.

Wir adressieren in unserem Ansatz in Ergänzung zu dieser Arbeit die aus unserer Sicht wichtige Problemstellung der Erhebung und Formalisierung von Abhängigkeitswissen.

6.3 LINUX Kernel Configurator LKC und Debian Package Manager

Das LINUX-System enthält einen Kernel-Rekonfigurator für seine Software-Bausteine, der auf einer expliziten Modellierung der Abhängigkeiten zwischen den Konfigurationsoptionen in der KConfig Abhängigkeitssprache basiert. Der Ansatz bewährt sich über eine Periode, in der sich die Anzahl der Merkmale mehr als verdoppelt hat. Die Praxis zeigt jedoch, dass es im Falle neuer Komponentenversionen regelmäßig zu Konfigurationsproblemen kommt, weil nicht alle oder falsche Abhängigkeiten modelliert wurden oder weil die Abhängigkeiten nicht mehr nachvollziehbar sind. Das liegt daran, dass die Abhängigkeiten nicht formal abgeleitet, sondern aus der

Entwicklungserfahrung direkt umgesetzt werden. Eine wesentliche Erkenntnis aus einer Analyse des LINUX-Konfigurators [Lo10] ist ein Mangel an Unterstützung hinsichtlich Nachvollziehbarkeit und Wartbarkeit der resultierenden Repräsentation. Die Autoren extrahierten aus den Commit-Logs auf Seite 12 ihrer Arbeit Aussagen wie "After carefully examining the code...", "As far as I can tell, selecting ... is redundant", "we do a select of SPARSEMEM_VMEMMAP ... because ... without SPARSEMEM_VMEMMAP gives us a hell of broken dependencies that I don't want to fix " und "its a nightmare working out why CONFIG_PM keeps getting set" "(emphasis added)". Bei einer großen Gemeinschaft der Nutzer und falls keine erheblichen Schäden entstehen können ist dies möglicherweise nicht weiter tragisch, weil Fehler schnell auffallen und korrigiert werden. In einem kleinen Entwicklerteam und bei geringer Nutzungsrate oder im Falle kritischer Systemfunktionen ist eine Reifungsphase jedoch kritisch. Im Debian Package Manager werden Paketabhängigkeiten direkt zwischen Komponentenversionen mit den hier im engeren Sinne relevanten Relationen wie *suggests*, *replaces*, *conflicts* und *provides* abgebildet. Eine Herausforderung ist hier wie im LKC die vollständige und konsistente Beschreibung der Abhängigkeiten zwischen den Paketen.

Im Gegensatz dazu leiten sich die Abhängigkeiten in dem hier dargestellten Ansatz aus feineren Aussagen über Unterkomponenten und deren schematischen Umsetzung ab. Derzeit wird zwar die Modellierung noch manuell vorgenommen. Jedoch planen wir die Umsetzung von Kompatibilitätsaussagen zu automatisieren, um das Risiko von Fehlmodellierungen zu verringern und den Modellierungsaufwand zu minimieren.

6.4 Dokumentationsmethodiken

Die Herausforderung der Rekonfiguration eingebetteter Systeme stellt sich spätestens, wenn in der Werkstatt Erweiterungen, Funktionsoptimierungen oder Fehlerbehebungsmaßnahmen umgesetzt werden müssen. In der Dissertation von Köhler [Kö11] wird eine Erweiterung etablierter stücklistenorientierter Dokumentationsmethodiken beschrieben. Dieser Ansatz leitet aus einer Modellierung von Austauschketten für SW-Flashdateien und flashbaren HW-Komponenten unter den Rahmenbedingungen von Rekonfigurationsinvarianten ab, auf welche Weise eine bestimmte Menge von Rekonfigurationszielen erreicht werden kann. Eine Herausforderung dabei ist die Ermittlung funktionierender Austauschketten. Diese werden in dem Ansatz aus freigegebenen Nachfolgekonfigurationen oder auf Grundlage expliziter Tests ermittelt.

Der hier vorgestellte Ansatz basiert im Gegensatz dazu auf einer expliziten Modellierung von Funktionsumfängen und Komponentenschnittstellen von SW-Bausteinen der flashbaren Software zur Zeit der Entstehung der Variabilität. Dadurch werden die technischen Grundlagen der Austauschbarkeit abgebildet und nicht nur die Austauschbarkeit dokumentiert. Zweitens erfolgt die Dokumentation zeitnah und nicht erst vergleichsweise spät im Entwicklungs- oder sogar Dokumentationsprozess. Der dritte Unterschied ist die feinere Auflösung der Modellierung auf die SW-Komponenten, aus denen später die Flash-SW zusammengebunden wird.

6.5 Formale Methoden

Es existieren eine Reihe von Theorien für formales Software Engineering [ET01], die bis auf wenige Ausnahmen noch nicht in unserer Serienentwicklung angekommen sind. Wir verwenden beispielsweise derzeit statische Analysewerkzeuge zum Auffinden von Programmfehlern, aber nicht zur Unterstützung von Konfigurationsaufgaben. In [Tr07] wird beschrieben, dass SAT Solver und Logical Truth Maintenance Ansätze zu Analyse von Featureinteraktionen eingesetzt wurden. Dies würde für den Fall, dass der Quellcode vorliegt, das wichtige Teilproblem der Abhängigkeitsanalyse lösen. Eine weitere relevante Arbeit ist die Beschreibung einer expliziten, formalen Back-Box Spezifikation von Funktionalität auf Grundlage von Timed-Streams im Kontext eines service-orientierten Ansatzes zu Spezifikation von Softwareproduktlinien [HH07]. Der Ansatz sollte prinzipiell die formale Absicherung von Aussagen über Komponentenabhängigkeiten ermöglichen. Wir sehen jedoch vor dem Hintergrund vorhandenen Kenntnisse, Erfahrung, zeitlichen Rahmenbedingungen und der Problemgröße eine große Herausforderung darin, theoretisch fundierte Ansätze dieser Art in unseren Entwicklungskontexten einzuführen.

6.6 Architekturbeschreibungssprachen

Architekturbeschreibungssprachen (ADL) stellen Lösungsansätze bereit, um Komponentenarchitekturen und Komponentenschnittstellen zu modellieren. Beispielsweise ist in [Om00] beschrieben, wie für die ADL Koala aus Quellcodeanalysen Interfacebeschreibungen generiert werden können, wobei auch die Evolution von Systemen adressiert wird. Wir untersuchen derzeit in einer Studie, welche der existierenden Sprachen in unserem Umfeld welchen zusätzlichen Nutzen stiften könnten.

7. Zusammenfassung und zukünftige Arbeiten

Wir haben in diesem Papier einen neuen und praxistauglichen Ansatz zur Erfassung und formalisierbaren Dokumentation von Abhängigkeitswissens für die Rekonfiguration eingebetteter Systeme vorgestellt und eine systematische Abbildung in eine Implementierung auf Grundlage merkmalbasierter Variantenmodellierung dargestellt. Der Ansatz hat sich in der Praxis bewährt und wird daher in der Serienentwicklung eingesetzt. Er basiert auf einer bei uns erprobten Methodik und Werkzeugkette für merkmalbasiertes Variantenmanagement. Mit dem Ansatz sind wir der Lage, minimale Teilreleases zur Lösung einer Rekonfigurationsaufgabe zu bestimmen.

Eine kurzfristig lösbare und bereits in der Weiterentwicklung befindliche Verbesserung betrifft die Automatisierung der schematischen, aber dennoch teilweise umfänglichen, und fehlerträchtigen Änderungssequenzen in der Merkmalmodellierung, die sich regelmäßig im Laufe der Systemevolution als Folge neuer Komponentenversionen ergeben. Hier konzipieren und implementieren wir eine Schnittstelle vom Versionsmanagement in die Modellierung, die auf Grundlage der Kompatibilitäts-information darauf abzielt, die zugehörigen Änderungen automatisch und fehlerfrei durchzuführen.
Ein mögliches Weiterentwicklungspotenzial des Ansatzes ergibt sich aus der heute schwachen systematischen Absicherung von Aussagen über Kompatibilität: Aufgrund

welcher Kriterien entsteht eine Aussage über die Austauschbarkeit von Komponenten? Derzeit auf Basis von Überlegungen zum Funktionsumfang und den Konfigurationsschnittstellen von Komponenten und deren formalisierbarer Dokumentation. Wir haben begonnen, zur Systematisierung der Dokumentation komponentenspezifische Kriterienkataloge zu erstellen, um die Vollständigkeit und Widerspruchsfreiheit von Kompatibilitätsaussagen zu erhöhen. Erstrebenswert wäre hier darüber hinaus eine Erhärtung dieser Aussagen durch Kompatibilitätstests und durch analytische Verfahren, wie der statischen SW-Analyse oder aufgrund der automatischen Extraktion von Abhängigkeitsinformation. Ideal wären verifizierte Kompatibilitätsbeziehungen. Wir schätzen es allerdings als große Herausforderung ein, diese Verfahren in unserem Serienprozess umzusetzen.

Nicht zuletzt ist für uns als Forschungs- und Vorentwicklungsbereich auch interessant, wie sich der Ansatz in anderen Entwicklungsbereichen bewährt und wie er über die Zeit und hinsichtlich Größe skaliert. Entsprechende Pilotierung sind nach Serienübergabe in unserem jetzigen Anwendungsbereich geplant.

Literaturverzeichnis

[CKS06] Mary Beth Crissis, Mike Konrad, Sandy Shrum, CMMI: Guidelines for process integration and product improvement SEI series 2006

[CE00] K. Czarnecki, U. Eisenecker, Generative Programming, Addison-Wesley 2000

[FO90] Kang et al, Feature oriented domain analysis (FODA), Technical Report CMU/SEI-90-TR-21, CMU/SEI, 1990

[HH07] Harhurin A., Hartmann J.: A Formal Approach to Specifying the Functionalities of Software System Families, TUM Report, München 2007

[Jo11] John, U..: Konfiguration und Rekonfiguration mittels Constraint-basierter Modellierung Dissertation zur künstlichen Intelligenz, Aka 2002

[Kö11] Köhler, M.: Dokumentationsmethodik zur Rekonfiguration von Softwarekomponenten im Automobil-Service“, Dissertation der Mathematisch-Naturwissenschaftlichen Fakultät der Eberhard Karls Universität Tübingen, 2011

[Lo10] Lotufo R., She S., Berger, T., Czarnecki K., Wąsowski, A.: Evolution of the Linux Kernel Variability Model, SPLC 2010

[Ma05] Manhart P., Reconfiguration – A Problem in Search of Solutions, Configuration Workshop at IJCAI'05

[Ma99] Mannistö T., Soininen T., Toohonen J., Sulonen R. Framework and Conceptual Model für Reconfiguration, Proc. Of WS on Configuration at AAAI’99, Orlando, 1999.

[Om00] Ommering et. al. The Koala Component Model for Consumer Electronics Software, IEEE Computer 2000.

[Tr07] Trinidad et. al, Mapping Feature Models onto Component Models to Build Dynamic Software Product Lines, SPLC 2007

Anhang

Begriffe und Notationen

In diesem Abschnitt werden die uns wesentlichen Begriffe ausführlicher beschrieben, als im Papier, um Missverständnissen vorzubeugen. Für die in der Arbeit relevanten Begriffe wird jeweils eine Notation mit Beispielen angegeben, um kompakte Aussagen zu ermöglichen. Da Syntax in gewissem Umfang beliebig ist, haben wir versucht die im Entwicklungsbereich bereits vorhandenen Schreibweisen um die für hier notwendigen Elemente zu erweitern.

Komponente

In der Automobilindustrie wird der Begriff oft für nur für ein Steuergerät inklusive Software in einem vernetzten Fahrzeugsystem aufgefasst. Hier jedoch wird der Begriff im Sinne der Systemmodellierung hierarchisch und rekursiv verwendet: Komponenten bestehen aus Komponenten, die wiederum aus Komponenten bestehen können und so weiter. Wir unterscheiden HW- und SW-Komponenten.
Generell konfigurationsrelevant bei *HW-Komponenten* sind deren mechanische Eigenschaften wie Bauraum, deren Ein- und Ausgangsschnittstellen, die Netzwerke, die Platinen inklusive deren Speicherausbau (ROM, RAM, NVRAM), Prozessoren und deren Benutzerschnittstellen wie Tester, Drehsteller, Leuchten, Zeigerwerke und Displays.
Konfigurationsrelevante Arten von *SW-Komponenten* sind Basis-SW-Komponenten, welche Betriebssystemfunktionen und eine statisch gebundene Middleware enthalten und die teilweise nicht einfach nachträglich austauschbar sind, z.B. durch Flashen, und Funktionssoftwarekomponenten, die oft aber nicht immer flashbar sind.
Der Einfachheit halber, jedoch ohne Einschränkung der Allgemeinheit betrachten wir hier den einfachsten Fall einer Komponente, die aus einer Menge atomarer HW- und SW-Komponenten besteht. Wenn die Tatsache betont werden soll, dass es sich um eine nichtatomare Komponente handelt, verwenden wir den Begriff *Zusammenbau*.

Notation: Eine Komponentenbezeichner hat die Form `Komp := [A-Z,a-z]*.`

Beispiele: Signaldatenbank, DML, D

Version

Entwicklungsartefakte werden meist in einem inkrementellen, evolutionären Entwicklungsprozess entwickelt. Dadurch ergeben sich aufeinanderfolgende Ausprägungen. Diese bezeichnen wir als *Versionen*.

Notation: Eine Version hat die Form `Ver := [0-9]*`

Beispiele: 20120213, 12001, 1

Variante

Ausprägungen von Entwicklungsartefakten mit ähnlichen, aber nicht gleichen Eigenschaften, die in einem gemeinsamen Betrachtungskontext nebeneinander existieren, bezeichnen wir als *Varianten*. Mit dem Begriff Betrachtungskontext ist ein den Artefakten gemeinsamer Gesichtspunkt, z.B. Releases oder deren Historie, gemeint, der festlegt, warum die Artefakte gemeinsam betrachtet werden.

Notation: *keine*, Varianten werden im Rahmen der Variantenmodellierung durch Konfigurationen im Variantenmodell abgebildet.

Funktionsumfang einer Komponentenversion, eines Softwarebausteins

Den Funktionsumfang oder die Funktionalität einer Version einer HW- oder SW-Komponente oder eines Zusammenbaus fassen wir als Menge der durch sie realisierten Funktionen auf. Damit man Funktionsgleichheit und Funktionsänderungen genau beschreiben und bei letzteren Erweiterungen von Einschränkungen unterscheiden.

Notation: Eine Komponentenversion und deren Funktionsumfang hat die Form `Kv := KompVer`

Beispiele: Signaldatenbank120213, DML0, D0

Bemerkung: Der hier vorgestellte Ansatz kommt in seiner jetzigen Form ohne eine weitere Verfeinerung der Beschreibung des Funktionsumfanges aus. Darum bezeichnet die Komponentenversion gleichzeitig den Funktionsumfang. Eine Erweiterung um die explizite Modellierung der einzelnen Funktionsversionen und -varianten erscheint reizvoll; derzeit existiert jedoch in unserem Umfeld kein wichtiger Use-Case dafür.

Konfigurationsschnittstellen einer Komponentenversion

Damit eine Version einer vernetzten Komponente ihren Funktionsumfang in ihrem Kontext erbringen kann, müssen ihre *Schnittstellen* aus Konfigurationssicht, das heißt die Menge ihrer funktionalen und nichtfunktionalen Beziehungen zu anderen Komponenten abgedeckt sein. Um das sperrige Wort Konfigurationsschnittstelle abzukürzen, verwenden wir im Folgenden den Begriff Schnittstelle.
Es ist aus Konfigurationssicht sinnvoll, die von ihr benötigten *Eingangsschnittstellen*, von den von ihr bereitgestellten *Ausgangsschnittstellen* zu unterscheiden. Analog zum Verzicht auf eine Zergliederung des Funktionsumfanges kommt der hier vorgestellte Ansatz in seiner jetzigen Form ohne eine weitere Verfeinerung der Schnittstellenbeschreibung etwa in Ports, Datagramme etc. aus.

Notation: Die Schnittstellen zwischen den Komponente A und B haben folgende Form:

Komponentenausgangsschnittstelle von A zu B:	`Ko := A>B`
Komponentenausgangsschnittstellenversion von A zu B:	`Kov := A>BVer`
Komponenteneingangsschnittstelle von B zu A:	`Ko := B>A`
Komponenteneingangsschnittstellenversion von B zu A:	`Kov := B>AVer`

Beispiele: SigDB>APPL, DML>A120213, A>D001

Release

Unter einem Release verstehen wir eine identifizierbare Menge miteinander freigegebener Komponentenversionen. Releases existieren auf unterschiedlichen Integrationsstufen wie Gesamtfahrzeug, Fahrzeugsystem oder Komponente und in unterschiedlichen Reifegraden von frühen Test- und Erprobungsreleases bis hin zu serienreifen Freigaben oder Aftersales-Releases.

Notation `RelVer`

Beispiel: Rel120213, Rel1

Kompatibilität von Komponentenversionen

Mit *Kompatibilität* bezeichnen wir die Möglichkeiten und Folgen des Austausches zweier Komponentenversionen in Bezug auf bestehende Releasekonfigurationen. Um Aussagen zu den Folgen eines Tausches von Komponenten machen zu können, müssen darum deren kompatibilitätsrelevanten Eigenschaften beschrieben werden.

Dabei muss zum einen die *funktionale Kompatibilität* gewährleistet werden: Zwei Komponenten mit gleichem Funktionsumfang und gleichen Schnittstellen sind *austauschbar* oder *kompatibel*. *Abwärtskompatibilität* bedeutet, dass Nachfolgerversionen für Vorgänger eingesetzt werden können, *Aufwärtskompatibilität*, dass Vorgängerversionen für Nachfolger eingesetzt werden können. *Inkompatibilität* bedeutet, dass nach einem isolierten Austausch unakzeptable Änderungen im Funktionsumfang oder Funktionsstörungen durch nicht zusammenpassende Schnittstellen zu erwarten sind.

Notation: Kompatibilitätsaussagen haben die Form:

aufwärtskompatibel:	`Auf := -`
abwärtskompatibel:	`Ab := +`
keine Änderung oder voll kompatibel:	`Voll := 0`
inkompatibel:	`Ink := X`

Um die Kompatibilität des Funktionsumfangs von Komponentenversionen oder Komponentenversionsschnittstellen zu machen, können die entsprechenden Notationen kombiniert werden, wobei hier der Einfachheit des Sachverhalts entsprechend die Kombinationen zusammengefasst dargestellt werden. Die Aussagen und damit die Notationen beziehen sich implizit auf die Vorgängerversion:

Notation: Kompatibilitätsaussagen mit Bezug haben die Form:
auf-/abwärts/voll/inkompatible Funktionsänderung einer Komponente: `Komp-, Komp+, Komp0, KompX`
Komponentenversionenkompatibilität analog: `KompVer- KompVer+, KompVer0, KompVerX`
Schnittstellenversionskompatibilität analog: `A>BVer+ A>BVer-, A>BVer0, A>BVerX`