Präzise Syntaxdefinition einer Modellierungstechnik für Infotainment-Systeme

Antje Jud

Diplomarbeit am Fachbereich Informatik der Technischen Universität Berlin

In Kooperation mit IAV GmbH

Januar 2007

Die selbstständige und eigenhändige Anfertigung versichere ich an Eides Statt.

_____

Berlin, den / Unterschrift

Inhaltsverzeichnis

1 Einleitung

Die Infotainment Markup Language (IML) dient zur Spezifikation des Aussehens und des Verhaltens von Infotainmentsystemen wie sie zum Beispiel in Automobilen ihren Einsatz finden. Die unterschiedlichen Aspekte solcher Systeme (z. B. Bedienkonzepte, Oberflächengestaltung, Funktionalität) werden von verschiedenen Berufsgruppen bearbeitet und festgelegt (z. B. Psychologen, Designern, Technikern). Die Beschreibungen der jeweiligen Aspekte eines Systems werden bisher in den unterschiedlichsten Formaten festgehalten und meistens ausgedruckt, als physikalische Dokumente, von einem zum anderen weitergereicht. IML stellt eine Möglichkeit dar alle Aspekte eines komplexen Systems in einem einzigen Format zu beschreiben. Ihr Nachteil ist, dass sie mit ihrer textuellen Darstellungsform und ihrer komplexen Struktur auf den ersten Blick nur wenig mit den bisherigen Beschreibungsformen gemeinsam hat und für Menschen schwer zu verwenden ist.

Ziel dieser Diplomarbeit ist es nun aufzuzeigen, wie IML benutzerfreundlich durch graphische, der jeweiligen Aufgabe angepasste Beschreibungssprachen erzeugt werden könnte. Diese Beschreibungssprachen sollten durch passende Editoren erzeugt werden und sich automatisch in IML überführen lassen. Zudem sollte es möglich sein, aus den in IML spezifizierten Daten die jeweilige grafische Darstellung zu erzeugen.

Um dieses Ziel erreichen zu können, wird zuerst ein Metamodell für IML erstellt. Die bisherige Beschreibung von IML besteht aus XML-Schemata mit eingebundenen Constraints, einer natürlich-sprachlichen Beschreibung ihrer einzelnen Elemente und mündlich vereinbarten Richtlinien. Für die Beschreibung von Metamodellen gibt es zwei verbreitete Methoden, die Meta-Object Facility (MOF) und die Graphtransformation. Da nicht das Erstellen von direktem IML, sondern die Beschreibung der schon existenten Struktur im Vordergrund steht, wird in dieser Arbeit zur formellen Beschreibung die Meta-Object Facility gewählt. Die vorhandenen Constraints, sowohl die in den Schemata als auch die informell existierenden, werden in der Object Constraint Language (OCL) beschrieben.

Nach der Beschreibung des Metamodells für IML wird beispielhaft für eine graphische Beschreibungssprache ebenfalls ein Metamodell in MOF erstellt. Dafür werden die zurzeit verwendeten Statecharts ausgewählt. Die Statecharts haben im momentanen Prozessablauf die Aufgabe die Menüwechsel in Abhängigkeit von System- oder Userevents zu beschreiben. Sie enthalten außerdem informelle Hinweise zur Funktionalität und andere Informationen, die in dieser Form nicht automatisch auswertbar sind. Das Metamodell der Teile des Statecharts, die automatisch erfasst werden können, hat also nur eine partielle Überdeckung mit dem IML-Metamodell. Wenn zukünftig die toolbasierte graphische Erstellung der Spezifikation aller Aspekte des Infotainmentsystems angestrebt wird, werden weitere Beschreibungssprachen und deren Metamodelle für die noch nicht beschriebenen Aspekte benötigt. Durch die Kombination aller dieser Modelle sollte eine vollständige Überdeckung mit dem Metamodell von IML erreichbar sein.

Solange diese anderen Metamodelle noch nicht existieren, werden alle Aspekte des IML-Metamodells, die durch das Metamodell der Statecharts nicht abgedeckt werden, direkt aus dem IML-Modell übernommen. Dadurch wird eine vollständige Überdeckung beider Modelle

sichergestellt. Diese Überdeckung ist nötig, um die beiden Modelle später ohne Informationsverlust ineinander überführen zu können. Diese Überführung muss verlustfrei sein, da sie bidirektional durchgeführt werden soll. So kann auf der einen Seite die textuelle Beschreibung leicht verwendet werden, um daraus automatisiert Code zu erzeugen und auf der anderen Seite können die für Menschen leichter lesbaren, graphischen Beschreibungen zur Erstellung und Änderung des Systems genutzt werden.

2 Grundlagen

2.1 Die Anforderungen an ein Embedded Infotainmentsystem

Die Entwicklung von Software für Embeddedsysteme wird stark von den meist sehr engen Grenzen der Systeme beeinflusst. Ein System, das sich gerade in der Entwicklung befindet, stellt zum Beispiel zur Umsetzung der gesamten Spezifikation 16KByte RAM und 100 KByte ROM zur Verfügung. Zur Spezifikation zählen in diesem Fall zwar weder die Grafiken noch das programmierte Verhalten der Widgets, aber alle Informationen, die in IML festgelegt werden, müssen dort Platz finden. Ein anderes, in seinem Funktionsumfang etwas aufwendigeres System ist auf 32MByte für die gesamte Software inklusive aller Grafiken beschränkt.

Zu der geringen Größe des zur Verfügung stehenden Speichers kommt das Fehlen einer dynamischen Speicherverwaltung hinzu, das die Softwareentwicklung erschwert. Es muss immer klar sein wie viel Speicher benötigt wird, da dieser nicht dynamisch hinzugefügt werden, man aber auch nicht zu großzügig mit ihm umgehen kann. Da häufig Speicher mehrfach verwendet werden muss, ist es sehr wichtig zu gewährleisten, dass zwei Objekte, die den gleichen Speicher verwenden, entweder die gleichen Daten erwarten oder nie zur gleichen Zeit aktiv sein können. Deshalb werden Menüs aus dem Speicher entfernt, wenn sie nicht mehr angezeigt werden. Dies führt dazu, dass sie über die Anzeigezeit hinaus keine Daten in sich selber speichern können.

Desweiteren ist auch die Leistung der Prozessoren weit von den Möglichkeiten eines durchschnittlichen Computers entfernt. Ein Grund dafür sind die Anforderungen, die an die Hardware gestellt werden. Jede Hardware, die in ein Fahrzeug eingebaut wird, muss zum Beispiel im Temperaturbereich von -40 °C bis +80°C betrieben werden können. Übliche CPUs, Festplatten und Speicherelemente haben Betriebstemperaturbereiche, die viel kleiner sind. Hardware, welche die Anforderungen des Fahrzeugeinsatzes erfüllt, ist teurer oder leistungsschwächer als normale. Um dennoch die erforderliche Leistung zu erhalten, werden häufig die verschiedenen Aufgaben auf viele unterschiedliche, stark spezialisierte CPUs verteilt.

Da das gleiche Infotainmentsystem zum Teil von mehreren Herstellern gleichzeitig produziert wird und somit auf unterschiedlicher Hardware laufen muss, kann die Spezifikation des Systems nicht mit Blick auf eine spezielle Hardware erfolgen. Es wird vielmehr versucht die Spezifikation so weit wie möglich von der Hardware zu trennen. Aus diesem Grund sind zum Beispiel alle angegebenen Funktionen als ein abstrakter Platzhalter für eine Funktion zu verstehen, die mit der Hardware kommunizieren kann. Er muss letztendlich nicht einmal einem direkten Funktionsaufruf entsprechen, sondern kann auch als Protokoll-Kommunikation über eine Busschnittstelle implementiert sein.

Die Kommunikation und Funktionalität der Hardware darf nicht durch die Benutzerinteraktion verzögert oder gar verhindert werden. Das gesamte System muss eine hohe Ausfallsicherheit haben und nach einem Ausfall möglichst unauffällig und schnell wieder bereit stehen. [Kreu 1]

All diese Anforderungen müssen von Spezialisten für das jeweilige Zielsystem in der Software beachtet werden. Deshalb sollte die Portierung der Spezifikation bzw. die mögliche Codegenerierung aus der Spezifikation von ihnen vorgenommen werden. Die Spezifikateure hingegen müssen diese Grenzen nur in Ausnahmefällen betrachten. Wenn ein System zu grafiklastig wird oder auf den Versand von zu vielen Nachrichten gleichzeitig angewiesen ist, ist eine Optimierung eventuell nicht mehr in der Lage, das System innerhalb der gegebenen Grenzen zu halten. Dann ist es nötig, dass Designer oder Spezifikateure ihre Arbeit so verändern, dass eine Anpassung auf die Hardware wieder möglich wird. Eine verwendete Möglichkeit den Speicherbedarf von Bildern zu reduzieren ist zum Beispiel, sie in neun Teile aufzuteilen. Dadurch wird es möglich Bilder unterschiedlicher Größe aus den gleichen gespeicherten Daten zu erzeugen. Dafür werden die Ecken jeweils einmal und die Mittelteile mehrfach verwendet. In diesem Fall müssen die Designer ihre Arbeitsweise auf die Beschränkungen des Systems anpassen.

2.2 Die Intention von IML

Heutzutage werden elektronische Geräte entwickelt und hergestellt, indem sie in elementare Komponenten zerlegt werden, die später zum Endprodukt kombiniert werden. Die Komponenten werden jeweils von Experten entwickelt, die mit den Problemen in diesem Teilstück des Gerätes vertraut sind. An der Entwicklung eines Gerätes arbeitet eine große Anzahl von Entwicklern, deren Wissensstand, selbst bezogen auf eine einzige Komponente, sehr inhomogen sein kann. Kommunikation ist in einer solchen Situation essentiell, wird jedoch durch die oftmals globale Verteilung der Entwicklergruppen und die nicht einheitliche Sprache erschwert. Selbst zwei Entwicklergruppen, die eigentlich die selbe Sprache sprechen, haben oft Probleme sich gegenseitig zu verstehen, da Begriffe, die eine Gruppe als klar definiert empfindet, bei den Anderen in einer anderen Form verwendet werden.

Auf Grund der Zielgruppe der in dieser Arbeit beschriebenen Spezifikationssprache wird als Beispielentwicklung ein Infotainmentsystem im Sinne der Automobilindustrie verwendet. "Infotainment" ist eine Wortschöpfung, die das Zusammenspiel von Unterhaltung (Entertainment) und Information in einem System verdeutlichen soll. Einfache Infotainmentsysteme unterscheiden sich in ihrem Funktionsumfang kaum bis gar nicht von dem gewohnter Autoradios. Ihre komplexen Geschwister enthalten jedoch viele weitere Funktionen. Neben der Steuerung eines oder mehrere Radiotuner und dem Abspielen von Audio-CDs können sie auch Funktionen für eine Navigation, die Steuerung einer eingebauten Freisprecheinrichtung für Handys, die Steuerung von TV-Tunern, das Abspielen von DVDs, eine erweiterte Nutzung von Verkehrsinformationen, das Betrachten von digitalen Bildern von unterschiedlichsten Speichermedien, die Speicherung von Medieninhalten auf eingebauten Festplatten, die Steuerung von CD-Wechslern, die Steuerung der Klimaanlage und eine Adressspeicherung enthalten. Die Liste der möglichen Funktionen wird in Zukunft noch anwachsen. [Bar 1]

Die Integration all dieser Funktionen in ein System hat für den Anwender viele Vorteile. Er kann zum Beispiel die gleichen Adressdaten nutzen, um Anrufe zu tätigen und Routen berechnen zu lassen. Außerdem können so Informationen aus den Verkehrsmeldungen des Radioprogramms direkt im Navigationssystem berücksichtigt werden. Für die Entwickler der Systeme stellt die Komplexität jedoch eine große Herausforderung dar. Um die Komplexität für den Anwender überschaubar zu halten, ist es von großer Bedeutung einheitliche Bedienkonzepte bei allen Teilsystemen einzuhalten. Diese dient außerdem zur Widererkennung einer Marke. Auch die grafische Darstellung der Bedienoberfläche und die Wahl von Bezeichnungen für Befehle müssen über alle Teilsysteme koordiniert werden. All diese Konzepte müssen so unter den Entwicklern kommuniziert werden, dass es nicht zu

so unter den Entwicklern kommuniziert werden, dass es nicht zu Missverständnissen kommt, während sie gleichzeitig an die Anforderungen des Systems angepasst werden müssen. Es hat sich gezeigt, dass die natürlich-sprachliche Beschreibung der Konzepte nicht mehr ausreichend ist.

Als Versuch diese Problematik zu lösen, wurde zum Beispiel angefangen UML als Spezifikationssprache zu verwenden. Allerdings waren die erzielten Ergebnisse unbefriedigend, was jedoch nicht auf Schwächen von UML zurückzuführen ist, sondern auf ihre ungenaue oder gar falsche Anwendung. Es wurde nur ein kleiner Teil der gesamten Spezifikation umgestellt und dieser Teil auch nur auf eine angepasste Form der Statecharts. Diese sind an sich ausreichend, um die Bedienabläufe darzustellen, jedoch fehlten Möglichkeiten zu spezifizieren, aus welchen Elementen ein Menü aufgebaut ist oder welche Events im System auftreten können. Ein weiteres Problem mit dieser Spezifikationsmethode ist, dass keine UML-Editoren zur Erstellung der Statecharts verwendet wurden, sondern dieselben in Grafikprogrammen 'gemalt' wurden. Dies hatte nicht nur den Nachteil, dass den Entwicklern zu viele Freiheiten eingeräumt wurden, sondern die Ergebnisse konnten auch nur als Grafik weitergegeben werden und waren somit nicht maschinell zu verarbeiten. Die oben genannten Freiheiten führten darüber hinaus dazu, dass die entstandenen Statecharts nur noch bedingt mit der eigentlich beabsichtigten Form übereinstimmten. Es wurden zum Beispiel Informationen über aufzurufende Funktionen nicht durch Transitionen mit entsprechender Aktion beschrieben, sondern als Inhalt eines Kommentars oder einer Notiz. Der grundsätzliche Aufbau eines Grafikelementes wurde ebenso wenig spezifiziert, wie seine Platzierung auf dem Bildschirm oder seine Möglichkeiten mit anderen Elementen zu kommunizieren.

Ein geeignetes Tool oder Tool-Kette muss in der Lage sein, den Anwender zu Vollständigkeit zu zwingen. Gleichzeitig darf die gewohnte Arbeitsweise nicht zu sehr eingeschränkt werden. Entwickler, die es gewohnt sind Modelle zuerst skizzenhaft zu entwerfen und sie erst nach Absprachen mit Kollegen und Zuarbeiten von anderen Entwicklungsbereichen zu verfeinern, können nicht plötzlich vollständige Modelle aus dem Stehgreif erzeugen. Sie brauchen weiterhin die Möglichkeit, unvollständige oder ungenaue Modelle zu entwerfen, sollten jedoch zu geeigneten Zeiten auf die Mängel in ihrem Modell hingewiesen werden.

2.3 Das XML-Schema

Das XML-Schema ist eine Methode zur Beschreibung der gewünschten Struktur eines XML-Dokuments, bei der XML wiederum als Beschreibungssprache verwendet wird. Ein XML-Schema legt fest, welche Elemente in einem XML-Dokument enthalten sein können, welche Attribute ein Element haben kann und an welchen Stellen des Dokuments das Element erscheinen darf. Es bestimmt die Abhängigkeiten zwischen den Elementen, die Anzahl und Reihenfolge der Kind-Elemente eines Elements und die Datentypen seiner Attribute.

Ein Schema kann enthalten:

Verweise auf andere Schemata: In einem Schema können mehrere Schemata zusammengefasst werden. Dadurch kann ein Projekt beschrieben werden, das aus einer Reihe von Dateien mit unterschiedlichem Aufbau besteht.

Einfache Elemente (simple elements): Einfache Element enthalten keine weiteren Elemente oder Attribute. Sie enthalten lediglich Text, der durch einen zugewiesenen Datentyp und andere Bedingungen beschrieben werden kann.

Attribute: Für Attribute werden ihr Name und ihr Datentyp definiert. Sie können einen Default-Wert oder einen festen Wert zugeordnet bekommen und sie können required (notwendig) oder optional sein.

Attributgruppen (attribute groups): In Attributgruppen werden Attribute zusammengefasst, damit sie später in den komplexen Elementen gemeinsam verwendet werden können. Bei häufig verwendeten Kombinationen von Attributen kann dies die Definition erleichtern.

Eigene Datentypen: Der Wertebereich für den Inhalt von einfachen Elementen und Attributen kann über den Datentyp hinaus eingeschränkt werden. Dabei können entweder die Grenzen des Wertebereichs festgelegt werden (minInclusive, maxInclusive), die Werte direkt aufgezählt werden (enumeration) oder eine Schablone für die Daten angegeben werden (pattern). Durch die Einschränkung von Datentypen können eigene Datentypen definiert werden.

Komplexe Elemente (complex elements): Komplexe Elemente können andere Elemente und Attribute enthalten. Es gibt vier verschieden Arten von komplexen Elementen:

- Leere Elemente
- Elemente, die nur Text enthalten
- Elemente, die nur andere Elemente enthalten
- Elemente, die Text und andere Elemente enthalten.

Jede Art von komplexen Elementen kann außerdem Attribute enthalten.

Ein komplexes Element, das andere Elemente enthält, kann definiert werden, indem die ihm untergeordneten Elemente direkt bei seiner Definition aufgeführt werden oder indem ein Complextype definiert wird, der bei der Definition des komplexen Elements als sein Typ angegeben wird. Im zweiten Fall kann derselbe Complextype in verschiedenen Elementen verwendet werden. [Vli 1] [W3C 1]

Im Metamodell wurden die Klassen der Complextypes mit dem Präfix "ct_" versehen, während die komplexen Elemente, die ihre Kinder direkt definieren, mit dem Präfix "el_" versehen wurden.

Choice & Sequence: Durch Choice oder Sequence werden Elemente mit ihren Kind-Elementen verbunden oder Complextypes aufgebaut. Bei der Choice kann von den angegebenen Kind-Elementen nur ein Typ gleichzeitig, bei der Sequence müssen alle Kind-Elemente in der angegebenen Reihenfolge auftreten. Die Anzahl der jeweiligen Elemente (keines, eines oder viele) kann dabei angegeben werden.

2.4 Meta-Object Facility

Die Meta-Object Facility (MOF) [OMG 1] wurde von der OMG entwickelt und standardisiert. Sie dient zur Spezifikation von Metamodellen von Sprachen. Die durch MOF erzeugten Metamodelle verschiedener Sprachen können in Beziehung gesetzt und teilweise oder vollständig ineinander überführt werden. Ein Beispiel für eine Sammlung solcher Metamodelle ist UML (Unified Modeling Language), die ebenfalls von der OMG standardisiert wurde. Ein

weiteres Beispiel ist MOF selber, was dazu führt, dass MOF in der Sprache spezifizierbar ist, die sie spezifiziert.

Um die maschinelle Verarbeitung der Metamodelle oder der in den durch die Metamodelle spezifizierten Sprachen zu gewährleisten, bietet XMI (XML Metadata Interchange) die Möglichkeit diese in XML (Extensible Markup Language) zu schreiben, wobei die Struktur des XML durch eine erzeugte DTD (Document Type Definition) bestimmt wird.

Visuell teilt sich MOF ihre Darstellung mit den UML Objektdiagrammen. Dadurch ist es möglich, MOF-Modelle in UML Editoren zu erzeugen. Dabei sind nur die engeren Beschränkungen der MOF gegenüber der UML zu beachten.

Die Hauptbestandteile von MOF sind Klassen, Assoziationen, Datentypen und Pakete. Im Folgenden werden die Elemente der MOF, die für das in dieser Arbeit erstellte Modell verwendeten wurden, beschrieben.

Die **Klassen** in MOF definieren die Typen von Objekten. Sie können Attribute und Operationen enthalten und durch Assoziationen miteinander verbunden werden. Die Klassen des folgenden Modells enthalten zum Teil Attribute, aber keine Operationen. MOF Klassen können außerdem noch Exceptions, Konstanten, Datentypen und Constraints enthalten. Von diesen Möglichkeiten wurden nur die Constraints verwendet. Die Datentypen wurden global für das gesamte Modell definiert. Die Constraints wurden zur besseren Lesbarkeit außerhalb der Klassen dargestellt.

Klassen können Generalisierungen von anderen Klassen sein. Die generalisierte Klasse vererbt dann ihre Eigenschaften (Attribute, Operationen, Assoziationen usw.) an ihre Spezialisierungen. Wenn die generalisierte Klasse abstrakt ist, bedeutet dies, dass nur ihre Spezialisierungen instanziiert werden können.

Die **Attribute** haben einen Typ, der entweder ein Datentyp oder eine Klasse sein kann, einen Namen, der innerhalb einer Klasse eindeutig sein muss, und einen oder mehrere Werte.

Datentypen sind Typen, die keinen Klassen entsprechen. Sie entsprechen primitiven Typen, setzen sich aus anderen Typen zusammen oder schränken andere Typen ein. Ein Datentyp kann einer Aufzählung von Werten entsprechen (Enumeration).

Die **Assoziationen** beschreiben die Beziehungen zwischen den Klassen. In MOF hat eine Assoziation genau zwei **Assoziationsenden**, die jeweils mit einer Klasse verbunden sind. Die Assoziationsenden haben eine Kardinalität, die bestimmt, wie viele Instanzen an dieser Beziehung teilnehmen können. Eine besondere Form der Assoziation ist die **Komposition.** Sie darstellt, dass eine Instanz einer Klasse nur beim Vorhandensein einer Instanz einer anderen Klasse existieren darf.

Die **Constraints** im folgenden Beispiel werden in OCL (Object Constraint Language) [OMG 2] geschrieben.

2.5 Begriffsklärung

Während der Entwicklung und der anschließenden Verwendung von IML hat sich unter den Anwendern eine gemeinsame Sprache über IML ausgeprägt. Die folgenden Beschreibungen

und Erklärungen zu IML verwenden diese Sprache. Damit alle Begriffe, die außerhalb des Kontextes von IML abweichende Bedeutungen haben können, allgemein verständlich werden, sind sie hier erklärt.

IML - Infotainment Markup Language (IML) ist eine Sprache zur Spezifikation von Infotainmentsystemen.

Widget - Ein Widget ist ein Anzeigeobjekt beliebiger Komplexität, wobei ein Widget nicht zwingend eine grafische Repräsentation besitzen muss. Vereinfacht gesagt entspricht alles, was man auf dem Bildschirm eines Infotainmentsystems sehen kann, einem Widget. Da Widgets jedoch hierarchisch geordnet sind und sich gegenseitig beinhalten können, ist auch jede Zusammenfassung von Widgets wieder ein Widget.

Base-Widget – Base-Widgets sind die einfachsten grafischen Elemente wie Linie, Image oder Textfeld. Base-Widgets können keine anderen Widgets enthalten. Eine ausführliche Erklärung der Base-Widgets wird in der Beschreibung von IML gegeben. (siehe Kapitel 4.6.3)

Widget-Baum - Alle Widgets eines Systems sind baumförmig angeordnet. Das Menü-Netz, das alle anderen Menüs und Menü-Netze enthält, ist die Wurzel des Baumes. Die Base-Widgets bilden seine Blätter. In einem Widget-Baum können Widgets auch mehrfach, auf beliebigen Ebenen vorkommen.

Menü - Das Menü in IML bezeichnet einen gesamten Bildschirminhalt. Menüs sind im Widget-Baum die höchsten Widgets, die eine grafische Repräsentation besitzen. Über ihnen befinden sich nur noch die Menü-Netze.

Overlays - Overlays werden manchmal auch als Popup bezeichnet und sind eine Sonderform der Menüs. Sie werden zur Anzeige von Status- oder Fehlermeldungen verwendet. Overlays werden über den Menüs dargestellt. Obwohl sie dabei scheinbar nur einen Teil des Bildschirms bedecken, habe sie in Wirklichkeit die gleichen Ausmaße wie andere Menüs auch.

Menü-Netz - Ein Menü-Netz kann Menüs und andere Menü-Netze als Elemente enthalten und fasst diese zu bestimmten Themengebieten zusammen. Menü-Netze dienen der Steuerung, welches Menü zu einem bestimmten Zeitpunkt angezeigt werden soll oder wann ein Menüwechsel statt zu finden hat.

Property – Ein Widget kann Propertys habe.Propertys stellen die Eigenschaften eines Widgets dar. Durch sie können das Aussehen und das Verhalten des Widgets verändert werden.

Überschreibung - Widgets entstehen durch die Kombination von bereits bestehenden Widgets, die durch ihre Propertys in ihrem Aussehen und Verhalten geändert werden können. Da viele Widgets an vielen verschiedenen Stellen im System verwendet werden, müssen sie an viele unterschiedliche Anforderungen angepasst werden. Diese Anpassung geschieht durch Überschreibung der Propertys des Kind-Widgets. Dabei wird der Wert des Propertys nur für das Widget, das die Überschreibung vornimmt, und alle seine Eltern-Widgets geändert. Die Verwendung des Kind-Widgets in anderen Zweigen des Widget-Baumes wird dadurch nicht beeinflusst.

Punktnotation - Eine Art Scriptsprache, die es ermöglicht Operationen in IML auszuführen und Querverweise anzulegen.

HardKey - Als HardKeys werden alle physikalisch existierenden Bedienelemente am Infotainmentsystem bezeichnet. Die Events, die ihre Bedienung auslöst, sind im gesamten System gleich, auch wenn die Reaktion darauf jeweils menü- oder kontextabhängig sein kann.

SoftKey - Als SoftKey wird ein Bedienelement bezeichnet, das nur als grafische Repräsentation auf dem Bildschirm existiert. In Systemen ohne Touchscreen bezeichnen SoftKeys auch Bedienelemente, die physikalisch direkt neben dem Bildschirm existieren und deren aktuelle Funktion jeweils durch eine Beschriftung auf dem Bildschirm angezeigt wird.

Target-Hardware - Gesamtheit der Hardwarekomponenten, auf denen das fertige System laufen soll.

Simulationskomponente - Software, welche die Funktionalität einer Hardwarekomponente, z. B. des Radiotuners, simuliert. Durch die Simulationskomponenten kann das Verhalten des Systems und vor allem seine Bedienbarkeit getestet werden, bevor es in Hardware existiert.

2.6 Überschreibungsregeln

Überschreibungen sind eine der Grundlagen von IML. Ohne sie wäre die Wiederverwendung von Widgets nur in beschränktem Umfang möglich. Da es in einem Ast des Widget-Baumes zu mehrfachen Überschreibungen desselben Propertys kommen kann, ist es von großer Bedeutung eine eindeutige Wichtung der Überschreibungen festzulegen. Jeder Entwickler, der ein System mit IML spezifizieren möchte, muss diese Wichtung kennen und verstehen. Für das Metamodell von IML hingegen ist sie relativ unbedeutend. Aus diesem Grund werden die Überschreibungsregeln hier nur kurz angerissen und an einem stark vereinfachten Beispiel dargelegt.

In Widgets können Property-Werte zustandsabhängig oder zustandsübergreifend definiert werden. Wenn sich das Widget im Zustand A befindet, hat ein Property entweder den Wert, der ihm abhängig vom Zustand A zugewiesen wurde, oder, wenn abhängig von diesem Zustand kein Wert zugewiesen wurde, nimmt es den zustandsübergreifenden Wert an. Der zustandsübergreifende Wert wird üblicher Weise als Default-Wert bezeichnet, was aber leider leicht zu Verwechselungen mit dem Default-Value führen kann. Default-Value ist eine der drei möglichen Arten von Values, die im Element "Values" enthalten sein können.

Wenn ein Widget als Element in einem anderen Widget verwendet wird, können seine Propertys vom Eltern-Widget überschrieben werden. Bei der Überschreibung gelten grundsätzlich zwei Regeln:

- Zustandsabhängige Werte sind wichtiger als zustandunabhängige.
- Die Werte der Eltern sind wichtiger als die Werte des Kindes.

Dabei ist zu beachten, dass ein im Kind zustandsabhängig definierter Wert bei den Eltern auch in Abhängigkeit der Zustände der Kinder geändert werden muss, gleichgültig, ob der Wert im Eltern-Widget selbst zustandsabhängig oder -unabhängig gesetzt wird. Wenn ein Designer entscheidet, dass ein Widget in seinen States ein unterschiedliches Aussehen haben soll, so kann ein anderer Designer, der das Widget des Ersten verwendet, diese Entscheidung so nur durch eine bewusste Handlung ändern.

Als kleines Beispiel wird ein Widget 'K' angenommen, das die Zustände 'A' und 'B' hat und das in einem Widget 'P' verwendet wird, das die Zustände 'C' und 'D' besitzt. In der folgenden Tabelle wird nun gezeigt, welchen Wert ein Property annehmen würde, wenn die angegebenen Überschreibungen in den angegebenen Zuständen vorgenommen wurden.

Das '-' bedeutet, dass an dieser Stelle keine Überschreibung vorgenommen wird. 'K(A)' bedeutet, dass das Property für den Zustand 'A' des Widgets 'K' gesetzt wird. Dieser Wert gilt also nur dann, wenn 'A' der aktive Zustand von 'K' ist. 'K( )' hingegen beschreibt das Setzen des Propertys außerhalb der möglichen Zustände. Dieser Wert wird nur dann verwendet, wenn das Property in einem Zustand nicht gesetzt wird.

'P( ) für K(A)' überschreibt den Wert des Propertys in 'P' für den Fall, dass 'A' der aktive Zustand von 'K' ist. Dabei ist es gleichgültig, welcher der Zustände von 'P' gerade aktive ist. Soll auch dieser aktive Zustand eine Bedeutung bekommen, so muss er mit angegeben werden. 'P(C) für K(A)' zum Beispiel überschreibt das Property ebenso wie im ersten Fall, allerdings nur, wenn sich das Widget 'P' im Zustand 'C' befindet. Schließlich ist es noch möglich die Überschreibung abhängig vom aktiven Zustand von 'P', aber unabhängig von aktiven Zustand von 'K' vorzunehmen: 'P(C) für K( )'.

Damit sich ein Wert in einem Property ändert, das explizit für einen bestimmten Zustand gesetzt worden ist, muss die Überschreibung in diesem Zustand stattfinden.

in K ( )	K(A)	P( ) für K( )	P( ) für K(A)	P(C) für K( )	P(C) für K(A)	aktive Zustände	Wert
rot	gelb	-	-	-	-	K(A)	gelb
rot	gelb	-	-	-	-	K(B)	rot
rot	gelb	grün	-	-	-	K(A) P(C)	gelb
rot	gelb	grün	-	-	-	K(B) P(C)	grün
rot	gelb	grün	-	-	-	K(A) P(D)	gelb
rot	gelb	grün	-	-	-	K(B) P(D)	grün
rot	gelb	grün	blau	-	-	K(A) P(C)	blau
rot	gelb	grün	blau	-	-	K(B) P(C)	grün
rot	gelb	grün	blau	-	-	K(A) P(D)	blau
rot	gelb	grün	blau	-	-	K(B) P(D)	grün
rot	gelb	grün	blau	lila	-	K(A) P(C)	blau
rot	gelb	grün	blau	lila	-	K(B) P(C)	lila
rot	gelb	grün	blau	lila	-	K(A) P(D)	blau
rot	gelb	grün	blau	lila	-	K(B) P(D)	grün
rot	gelb	grün	blau	lila	orange	K(A) P(C)	orange
rot	gelb	grün	blau	lila	orange	K(B) P(C)	lila
rot	gelb	grün	blau	lila	orange	K(A) P(D)	blau
rot	gelb	grün	blau	lila	orange	K(B) P(D)	grün

Die Reihenfolge der Werte ist:

- Der Wert für den aktiven Zustand des Kindes im aktiven Zustand der Eltern,
- der Wert für den aktiven Zustand des Kindes im Default der Eltern,

- der Wert im aktiven Zustand des Kindes,
- der Wert für den Default des Kindes im aktiven Zustand der Eltern,
- der Wert für den Default des Kindes im Default des Elters,
- der Wert im Default des Kindes.

Der erste Wert, der vorhanden ist,wird verwendet.

2.7 Das verwendete Beispiel

Um das Modell der Infotainment Markup Language leichter verständlich erklären zu können, werden zu den einzelnen Modellteilen die entsprechenden Darstellungen in IML aufgezeigt. Dazu wird ein kleines Beispielprojekt verwendet, das für diese Arbeit nochmals vereinfacht wurde, um die Übersichtlichkeit zu erhöhen. Das ursprüngliche Projekt hatte nicht vordringlich das Ziel die Möglichkeiten von IML darzustellen, sondern wurde zur Verdeutlichung der Funktionsweise eines Erstellungstools verwendet. Die übernommenen Teile des Projekts wurden deshalb so überarbeitet, dass sie die größt möglichen Variationen in IML beinhalten.

Das beschriebene Beispielsystem beinhaltet vier verschiedene Menüs, 34 weitere Widgets, 5 Dateien für Sprache, Definition von Konstanten, Funktionen und Events und eine Vielzahl von Bildern. Reales Projekte, die in der IAV GmbH bearbeitet werden, haben zum Beispiel viel größere Ausmaße. So enthält ein aktuelles Projekt ca. 350 Menüs und mehrere hundert Overlays, die System- oder Fehlermeldung darstellen. Insgesamt sind ca. 1760 Widgets nötig, um den gesamten Funktionsumfang in der gewünschten Art darzustellen. Diese Zahlen sind hier nur zur Veranschaulichung der Dimensionen von realen Projekten gedacht, damit ein besserer Eindruck davon entsteht, wie wichtig eine Werkzeugunterstützung für die Spezifikateure ist.

In unserem Beispiel wird im ausgeschalteten Zustand bzw. nach dem Start der Simulation das Menü "PCM_Off" angezeigt.

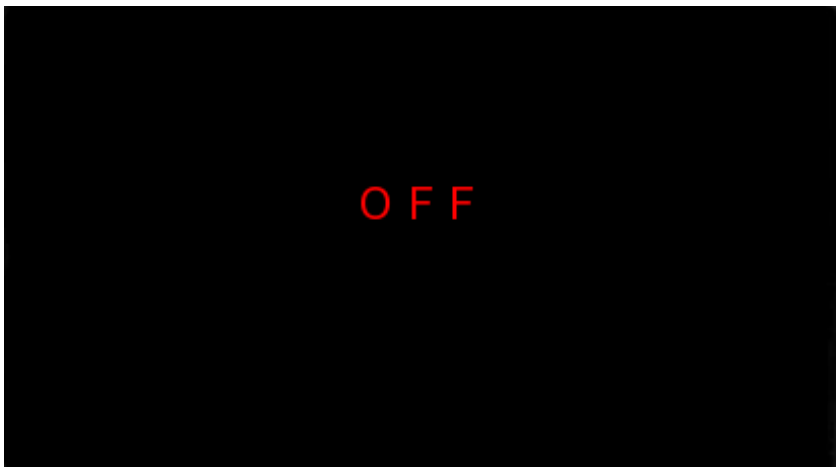

Abbildung 1 - Menu "PCM_Off "

Nach dem Einschalten wird zum Menü "PCM_On" gewechselt, das eine Animation ausgeführt, die zum Schluss wie folgt aussieht.

Abbildung 2 - Menü "PCM_ON"

Nach der Animation wird automatisch zum Menü "PCM_TUN_MAIN" oder "PCM_MEDIA_MAIN" gewechselt. Welches der Menüs jeweils ausgewählt wird, wird durch Bedingungen bestimmt. Im Menü "PCM_TUN_MAIN" werden sechs gespeicherte Radiosender angezeigt. Der momentan ausgewählte Sender wird dabei grafisch hervorgehoben.

Abbildung 3 - Menü "PCM_TUN_MAIN"

Durch die Betätigung von Hardkeys bzw. Tastatureingaben in der Simulation kann zwischen den Menüs gewechselt werden. Im Media-Kontext existiert hier nur ein Menü, welches das Abspielen einer CD ermöglicht.

Abbildung 4 - Menü "PCM_MEDIA_MAIN"

Bei den Menüs "PCM_TUN_MAIN" und "PCM_MEDIA_MAIN" steht der untere Rand des Bildschirms der so genannten Statuszeile zur Verfügung. Dort werden immer die Temperatur, die Uhrzeit und Informationen zur aktuellen Medienquelle angezeigt. Die Anzeige für die Medienquelle wird allerdings erst wirklich interessant, wenn in einem System auch Menüs existieren, die diese Informationen sonst nicht darstellen würden.

3 Der Modellierungsprozess

Das bevorzugte Vorgehen bei der Entwicklung einer neuen Sprache ist es, zuerst das Metamodell zu erstellen und daraus die konkrete Sprache abzuleiten. Im Falle von IML war dies leider nicht der Fall. IML ist eine Weiterentwicklung einer XML-basierten Sprache, die in einem vorhergehenden Projekt eingesetzt wurde. Diese "Urform" von IML war sehr begrenzt in ihren Ausdrucksmöglichkeiten. Sie verwendete Formulierungen, welche die maschinelle Verarbeitung für das resultierende XML praktisch unmöglich machte. In den Bezeichnern der XML-Elemente waren zum Beispiel die Namen der Widgets enthalten. Das Parsen war nur möglich, da spezielle Präfixe in den verschiedenen Tags verwendet wurden.

Aus der Erfahrung mit diesem Vorläufer wurde IML entwickelt. Dabei wurde entschieden, die bis dahin verwendeten DTDs zu Gunsten von XML-Schemata aufzugeben. Durch die Schemata wurden die möglichen Ausprägungen von XML so eingeschränkt, dass den Spezifikateuren eine Anleitung zur Erstellung einer verwendbaren Spezifikation zur Verfügung stand. Das XML-Schema enthält jedoch nur die Repräsentation der Daten, wie sie in XML erscheinen. Das zu Grunde liegende Datenmodell existierte bisher nur in den Köpfen der Schema-Entwickler und in einer textuellen Beschreibung der XML-Elemente. [Ang 1]

Im Laufe dieser Arbeit wurde ein Datenmodell entwickelt, das die in IML enthaltenen Daten beschreibt. Bei dieser Entwicklung wurde darauf geachtet, dass sich die Elemente der beiden Modelle, des Schemas und des MOF Models, möglichst leicht zuordnen lassen. Es wurden aber auch Teile, die für das Datenmodell keine Bedeutung haben, entfernt. Dabei handelt es sich meistens um Elemente, die in XML nur die Aufgabe haben, die ihnen untergeordneten Elemente zu bündeln, sonst aber keine weiteren Informationen enthalten. Da IML bisher von Hand geschrieben werden muss, ist diese Klammerung von Elementen gleichen Typs in XML sinnvoll, im Datenmodell wird sie jedoch nicht benötigt.

An anderer Stelle wurde das MOF-Modell gegenüber dem Schema vereinfacht, da in IML redundante Informationen enthalten sind. Das Schema unterscheiden zum Beispiel zwischen Werten in Propertys, die ohne Bedingung gelten und solchen, die zwar selbst keine Bedingung haben, denen jedoch bedingte Werte voran stehen.

```
<Values>

    <FixedValue>
        <Value Value=" "/>          <-- Ein Wert ohne Bedingungen
    </FixedValue>

</Values>
```

```
<Values>

    <ConditionalValue>
        <Condition Name=" ">
            [...]
        </Condition>
        <Value Value=" "/>
    </ConditionalValue>

    <DefaultValue>
        <Value Value=" "/>      <-- Ein Wert ohne Bedingungen, dem jedoch bedingte Werte vorangehen
    </DefaultValue>

</Values>
```

Da es für den unbedingten Wert keinen inhaltlichen Unterschied macht, ob ihm bedingte Werte voran stehen oder nicht, wurde diese Unterscheidung für das MOF-Model entfernt.

Wenn man dem Schema folgt, ist es möglich das Verhalten von Kind-Widgets ebenso zu überschreiben, wie die Werte in ihren Propertys. Die Spezifikateure des ersten Projekts haben sich allerdings schnell darauf verständigt, diese Möglichkeit nicht zu verwenden. Die Konsequenzen dieser Überschreibungen wurden bereits nach kurzer Zeit unüberschaubar. Da aus der Erfahrung des erstellten Projekts klar wurde, dass diese Art der Überschreibung auch in komplexen Situationen nicht zwingend benötigt wird, wurde sie für das MOF-Model vollständig verworfen.

Zu weiteren Vereinfachungen hat die grundsätzlich andere Struktur von MOF und die mit ihr gemeinsam genutzte Object Constraint Language (OCL) geführt. Das XML-Schema verbindet seine Elemente in einer hierarchischen, baumförmigen Struktur. Querverweise zwischen Elementen sind nicht möglich. Um im Schema sicherzustellen, das alle Elemente in einem Widget unterschiedliche Namen haben, muss es einen ausgezeichneten Typ bzw. eine ausgezeichnete Attributguppe geben, der nur im Attribut *Name* des Elements *Widget* verwendet wird. Aus diesem Grund gibt es im Schema Attributgruppen, die alle den gleichen Aufbau haben, die aber jeweils nur in einem bestimmten Element verwendet werden.

Durch OCL können in MOF Bedingungen wie Eindeutigkeit an Attribute in einem bestimmten Elementtyp geknüpft werden. Es ist daher möglich, den Typ eines Attributs in verschiedenen Elementen zu verwenden, ohne diese Einschränkungen zu verlieren. Ein Beispiel für diese Vereinfachung ist die Attributguppe für Name und Id, die im Schema für beinahe jedes benannte Element existierte und im MOF Modell zu einer einzigen Klasse zusammengefasst wurde. Dass die Eindeutigkeit innerhalb der typgleichen Elemente bestehen muss, wird durch OCL Ausdrücke wie dem Folgenden gewährleistet.

```
context el_Elements_Definition inv:
self.ct_Element_Definition→forAll(e1:ct_Element_Definition|
    self.ct_Element_Definition→forAll(e2:ct_Element_Definition|
    e1<>e2 implies e1.ElementName.Name <> e2.ElementName.Name))
```

Die Möglichkeit von MOF, beliebige Klassen in Relation zueinander zu setzen, wurde in diesem Modell genutzt, um die Definitionen der Funktionen und Nachrichten zentral, an einer Stelle zu sammeln. Das Schema gibt an, dass sowohl die Funktionen als auch die Nachrichten in jedem Widget selbst definiert werden müssen. Bei den Funktionen hat sich bei der praktischen Anwendung schnell herausgestellt, dass dieses Vorgehen zu Fehlern und Inkonsistenzen führt. Es ist kaum nachzuvollziehen, ob eine benötigte Funktion schon in einem anderen Widget eingeführt wurde oder ob sie neu erzeugt werden muss. Bei der Programmierung oder der maschinellen Umwandlung der Funktionen aus XML in Funktionsrümpfe oder Interfaces

in einer Programmiersprache käme es zu großen Redundanzen, die auf Grund von Ressourcenknappheit vermieden werden sollten.

Mit diesen Erkenntnissen wurde bereits für IML eine zentrale Datei mit allen bereits definierten Funktionen erzeugt. Die Funktionsbeschreibungen innerhalb der Widgets blieben jedoch erhalten. Dadurch wurden redundante Daten im System erlaubt, die leicht zu Fehlern führen können. Im MOF-Modell wurde daher in den Widgets nur ein Element für die Funktionen eingefügt, das jeweils die widgetspezifischen Informationen enthält und eine Relation zur Definition der gewünschten Funktion hat. Über diese Relation kann auf alle sonstigen Informationen der Funktion, z. B. die Signatur, zugegriffen werden.

Auch die Nachrichten werden im Schema dezentral, in den Widgets definiert. Bei ihnen wurde bisher aber noch keine Abhilfe der auftretenden Probleme geschaffen. Im MOF-Modell werden sie wie die Funktionen modelliert. Sie werden in einer eigenen Definitionsdatei definiert und in den Widgets über ein internes Element mit einer Relation zur gewünschten Nachricht verwaltet.

4 Das Metamodell von IML

Die Infotainment Markup Language basiert auf XML und wurde bisher durch ein XML-Schema beschrieben. Dieses Schema wurde als Grundlage für das Metamodell verwendet. Dadurch finden sich im Metamodell Konstrukte und Bezeichnungen, die sich von der Verwendung im Schema ableiten und deren Abwandlungen im Metamodell nicht zu einer Verbesserung geführt hätten.

Einige Attributgruppen des Schemas wurden als eigene Klassen übernommen, um den engen Zusammenhang von bestimmten Attributen zu verdeutlichen. Um sie von den Elementen des Schemas abzugrenzen, wurden sie in den Diagrammen nicht als Klassen verwendet, sondern als Attribute, deren Typ die Klasse ist. Die Klassen, die Attributgruppen repräsentieren, haben den Präfix 'ag_' in ihrem Namen.

Im Falle von *ag_Name_ID* wurden verschieden Attributgruppen, die alle ein Attribut für den Namen und ein Attribut für eine ID enthalten, zusammengefasst. Die verschiedenen Attributgruppen wurden im Schema benötigt, um Regeln für Namenseindeutigkeiten und -konventionen in Abhängigkeit von den verschiedenen Elementen beschreiben zu können. MOF in Zusammenarbeit mit OCL hat andere, zum Teil einfachere Möglichkeiten, das Gleiche auszudrücken. Diese Möglichkeiten wurden an diesen Stellen im Metamodell verwendet.

Bei zwei Attributgruppen, die sich nur durch die Multiplizität ihrer Elemente unterscheiden, wurden die Gruppen mit aussagekräftigen Namen versehen, die gleichzeitig ihre Verbindung verdeutlichen und einen Hinweis auf die Multiplizität geben. Aus diesem Grund wurden die Postfixe "_Optional" und "_Required" eingesetzt.

Um die Struktur des Schemas im Metamodell erhalten zu können, war die Verwendung von Hilfsklassen erforderlich. Die Selektion bzw. Sequenz werden an den Stellen, an denen ihr Erhalt sinnvoll war durch Klassen mit den Präfixen "sel_" bzw. "seq_" modelliert.

Bei der Erstellung des Schemas für IML wurden einige Elemente als Complextype modelliert, um sie mehrfach verwenden zu können. Die entsprechenden Klassen dieser Elemente tragen den Präfix "ct_", wenn sie bei der Metamodellierung übernommen wurden. Bei Elementen, die als Complextype modelliert, die jedoch im Schema nur an einer Stelle verwendet wurden, wurde im Metamodell zu direkt definierten Elementen übergegangen.

Die Elemente von IML, die kein Complextype sind, die also ihren Aufbau direkt definieren, wurden als Klassen mit dem Präfix "el_" modelliert.

Da die Elemente "WidgetFunctions" und "WidgetNotification" im Schema in seiner momentanen Version nur als Platzhalter für eine noch nicht ausgearbeitete Idee dienen, werden sie im Metamodell nicht übernommen. Wenn sie bei einer Überarbeitung des Schemas eine verwendbare Bedeutung bekommen, müsste das Modell und alle darauf basierenden Tools, die diesen Teil von IML bearbeiten, ebenfalls angepasst werden.

Die Definition von Nachrichten und Funktionen wurde im Modell aus den Widgets an eine globale Stelle verlagert. Sie werden als eigenständiger Teil des Modells behandelt, auf den aus den Widgets heraus nur verwiesen wird. Die Definition von Funktionen und Nachrichten in jedem Widget hat schnell dazu geführt, dass es keine eindeutigen Definitionen für sie gab. Deshalb wurde bereits das Schema erweitert, um wenigstens eine zusätzlich globale Definition der Funktionen zu ermöglichen. Die dadurch redundanten Definitionen in den Widgets wurden aber nicht entfernt. Im Modell wurde auf die Definition von Funktionen und Nachricht im Widget vollständig verzichtet.

Die Infotainment Markup Language besteht aus mehreren, unterschiedlich strukturierten Informationssammlungen, die zusammen eine nahezu vollständige Beschreibung eines Infotainmentsystems erlauben. Diesen Sammlungen enthalten Konstanten, die für das Aussehen verwendet werden (DataDictionary, Ressourcen), Funktionen (HMI_API), Events (Message), sprachabhängige Konstanten (Language) und die Widgets.

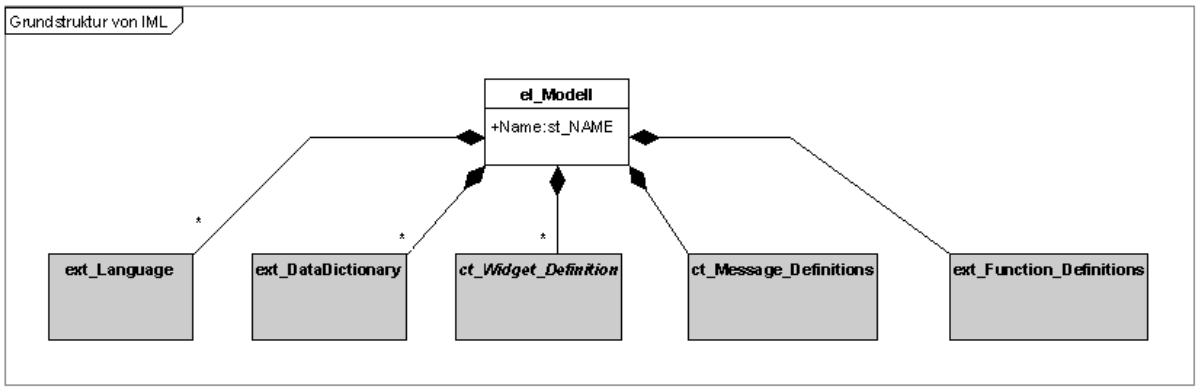

Abbildung 5 - Elemente eines Modells eines Infotainmentsystems

4.1 Übersetzungsdateien

Infotainmentsysteme werden in der Regel international vertrieben. Daraus ergeben sich eine Anzahl von Aufgaben, da zum Beispiel die Nutzer des Systems in einer Region einen bestimmten Benutzungsablauf gewohnt sind, der für Benutzer in einer anderen Region unverständlich wirkt. Diese Probleme sind nur durch verschieden spezifizierte Systeme, die im Allgemeinen Varianten genannt werden, zu lösen.

Doch auch wenn alle Benutzer einer Region die gleichen Bedienabläufe akzeptieren, möchte jeder von ihnen sein System auch in seiner Sprache verwenden können. Die Spezifikation eines Systems in allen angebotenen Sprachen als Varianten wäre nicht nur übermäßig arbeitsintensiv, sondern würde den Nutzer auch dazu zwingen, sich beim Kauf des Systems auf eine Sprache festzulegen.

Statt also unzählige Varianten in verschiedenen Sprachen zu spezifizieren, werden für alle Texte, die vom System angezeigt werden sollen, so genannte Text-Ids vergeben. Diese Text-Ids werden dem jeweils passenden Text in einer Sprache zugeordnet. Das System wird mit mehreren oder allen Sprach-Dateien ausgeliefert und es wird während der Laufzeit, abhängig von der eingestellten Sprache der passende Text auf dem Bildschirm angezeigt.

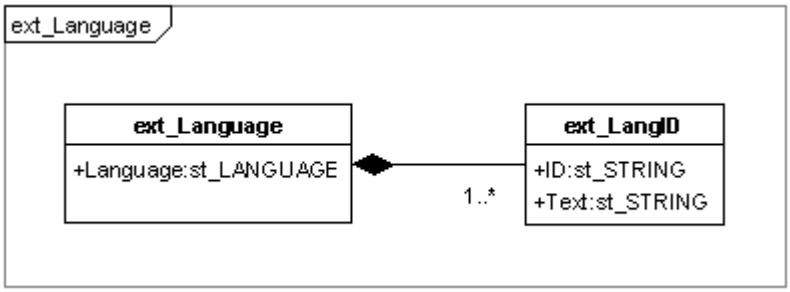

Abbildung 6 - Sprachdatei

Das Beispiel enthält nur eine sehr kleine Sprachdatei, da es nicht viele statische Texte gibt, die angezeigt werden können. Bei einem großen Projekt wird von den IML-Entwicklern ebenfalls nur eine Sprachdatei erstellt, die dann an Übersetzer weitergegeben wird. Die Übersetzer haben nicht nur die Aufgabe inhaltlich entsprechende Begriffe in einer anderen Sprache zu finden, sondern müssen auch dafür sorgen, dass die ausgewählten Begriffe in ihrer Länge nicht zu weit vom Original abweichen, da immer nur ein begrenzter Platz für sie zur Verfügung steht. Diese Arbeit kann ihnen erleichtert werden, wenn sie sich die Verwendung der Texte in einer Simulation ansehen können.

```
<!--=============================================-->
<!--Language Specific Texts: Texts ENGLISH-->
<!--=============================================-->
<lang_0409>
    <LangID ID="txt_NO_CONTENT"                Text="---"/>
    <LangID ID="txt_TEMPERATURE_STR"           Text="%1 °C "/>
    <LangID ID="txt_TRACK_STR"                 Text="%1 - %2"/>
    <LangID ID="txt_AUDIO_PLAY"                Text="Play"/>
    <LangID ID="txt_AUDIO_PAUSE"               Text="Pause"/>
    <LangID ID="txt_AUDIO_STOP"                Text="Stop"/>
    <LangID ID="txt_AUDIO_NEXT_TRACK"          Text="Next"/>
    <LangID ID="txt_AUDIO_PREV_TRACK"          Text="Prev"/>
    <LangID ID="txt_MENU_TITLE_TUN_MAIN"       Text="Radio"/>
    <LangID ID="txt_MENU_TITLE_MEDIA_MAIN"     Text="Media Context"/>
</lang_0409>
```

In den Text-Ids können Platzhalter (z. B. %1) verwendet werden, die während der Laufzeit des Systems ersetzt werden. Diese Ersetzung wird durch das .FORMAT der Punktnotation spezifiziert. Die eingesetzten Werte können dabei andere Text-Ids, feste Werte oder Ergebnisse von Funktionsaufrufen sein.

4.2 DataDictionary und Ressourcendatei

Die Verwendung von sprachabhängigen Text-Ids verringert die Menge der Varianten eines Systems deutlich. Trotzdem ist auch die nahezu gleichzeitige Entwicklung von fünf oder mehr Systemen mit begrenzten Ressourcen problematisch. Da die Varianten sich im Allgemeinen stark ähneln, ist die Wiederverwendung von bereits fertig gestellten Teilen nicht nur erstrebenswert, sondern auch zu einem großen Teil erreichbar. Solange sich die Änderungen zwischen den Varianten auf Farben, Dimensionen und Positionen der Elemente beschränken, können diese durch die Verwendung von Konstanten anstelle fester Werte erreicht werden.

Das DataDictionary enthält solche Konstanten und die ihnen zugeordnete Werte, die nach ihren Datentypen strukturiert werden. Es existiert für jede Variante ein eigenes DataDictionary, das die entsprechend angepassten Werte enthält. Die größte Schwierigkeit beim Anlegen

von Konstanten ist die Entscheidung, welche Informationen zu einer Konstante zusammengefasst werden können und für welche man individuelle Konstanten benötigt. Wenn es zum Beispiel nur eine Konstante für die Breite aller Button im gesamten System gibt, steht man vor einem Problem, wenn man später eine Variante spezifizieren muss, die Buttons mit verschiedenen Breiten enthält. Gleichzeitig jedoch ist die Arbeitserleichterung, die das DataDictionary gewährt, nur gering, wenn jeder Button eine eigene Konstante für seine individuelle Breite besitzt. Die Verwendung von individuellen Konstanten ist der von festen Werten zwar immer noch vorzuziehen, da so wenigstens alle zu ändernden Werte an einer Stelle gesammelt sind. Der wirkliche Nutzen des DataDictionary kann aber nur durch gute Planung zum Tragen kommen.

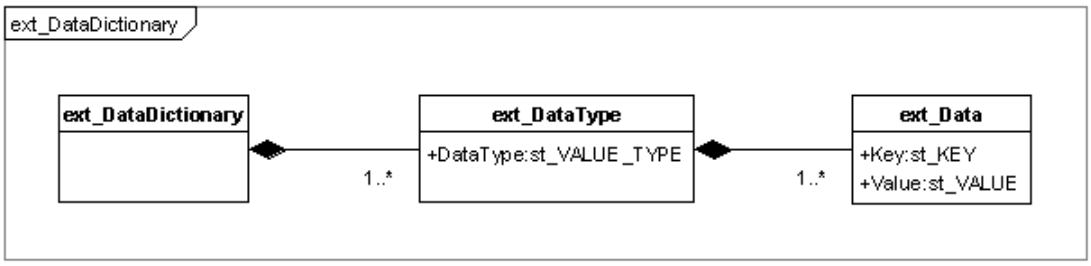

Abbildung 7 - Aufbau eines DataDictionarys

Neben der Anpassung der Systeme an die regionsbedingten Bedienabläufe und die verschiedenen Sprachen, wird es oftmals auch gewünscht, dass der Kunde das Aussehen der Oberflächen seinem individuellen Geschmack anpassen kann. Dazu werden für das System mehrere Oberflächen gestaltet. Diese werde Skins genannt. Der Kunde kann dann festlegen, welcher der vorgefertigten Skins ihm am meisten zusagt. Da sich Skins in den in ihnen verwendeten Farben stark unterscheiden können, werden unterschiedliche Definitionsdateien für die darzustellenden Farben benötigt. Für jeden Skin wird daher eine so genannte Ressourcendatei erstellt, in der die Farbwerte festgelegt werden, die über Ids in den Widgets verwendet werden.

Die Ressourcendateien entsprechen in ihrem Aufbau dem DataDictionary. Im Gegensatz zum DataDictionary, gibt es in einem System jedoch mehrere Ressourcendateien.

Dies ist ein Teil des DataDictionarys des Beispielprojekts, da die gesamte Datei zu umfangreich gewesen wäre sie hier darzustellen. Im DataDictionary können im Value auch Verweise zu anderen DataDictionary- oder Sprachdatei-Einträgen verwendet werden. (siehe Markierung 1 und 2 im Beispiel)

```
<HMI Name="PCM">
    <DataDictionary>
        <DataTypes>
            <DATA_TYPE_BOOL Key="bool_TRUE" Value="true"/>
            <DATA_TYPE_BOOL Key="bool_FALSE" Value="false"/>
            […]
            <DATA_TYPE_ALIGNMENT Key="alignment_CENTER" Value="Center"/>
            […]
            <DATA_TYPE_LIFETIME Key="lifetime_ONCE" Value="ONCE"/>
            […]
            <DATA_TYPE_PATH Key="path_BASE_WIDGET" Value="%DataWidgets%//Base//"/>
            […]
            <DATA_TYPE_UINT Key="uint_AUTOMATIC" Value="-1"/>
            <DATA_TYPE_UINT Key="uint_LIST_NUMBER_OF_ITEMS_DEFAULT" Value="5"/>
            <DATA_TYPE_UINT Key="uint_MEDIA" Value="2"/>
            […]
            <DATA_TYPE_STRING Key="str_EMPTY" Value=""/>
            <DATA_TYPE_STRING Key="str_MENU_TITLE_DELIMITER" Value=" - "/>
            <DATA_TYPE_STRING Key="MP_I1" Value=".LANG{txt_AUDIO_PLAY}"/>                    <-- ! 1 !
            […]
```

```
            <DATA_TYPE_MESSAGE_ID Key="hmi_msg_MENU_LEAVE" Value="MSG_MENU_LEAVE"/>
            <DATA_TYPE_MESSAGE_ID Key="msg_HARDKEY_PRESSED" Value="MSG_HARDKEY_PRESSED"/>
            [...]
            <DATA_TYPE_MESSAGE_TYPE Key="msg_type_NORMAL" Value="MESSAGE_TYPE_NORMAL"/>
            [...]
            <DATA_TYPE_UINT Key="sint_NULL" Value="0"/>
            [...]
            <DATA_TYPE_DYN_ARRAY_SINT Key="DynArray_Default" Value="{.DICT{sint_NULL}}"/>                <-- ! 2 !
            <DATA_TYPE_DYN_ARRAY_SINT Key="Array_ListAnchors_Default_Y" Value="{0;32;64;96;128;160}"/>
        </DataTypes>
    </DataDictionary>
</HMI>
```

In der Ressourcendatei werden die Farben, Größen und verwendeten Fonts für einen Skin festgelegt. Dadurch kann auch bei sehr unterschiedlichen Farben der Hintergrundbilder der Kontrast für die Schrift erhalten werden.

```
<HMI Name="PCM">
    <DataDictionary>
        <DataTypes>
            <DATA_TYPE_UINT Key="uint_FontSize_8" Value="8" />
            <DATA_TYPE_UINT Key="uint_FontSize_12" Value="12" />
            <DATA_TYPE_UINT Key="uint_FontSize_16" Value="16" />
            <DATA_TYPE_UINT Key="uint_FontSize_36" Value="36" />
            [...]
            <DATA_TYPE_COLOR Key="color_IAV_BLUE" Value="0xFF7D3700" />
            <DATA_TYPE_COLOR Key="color_TEXTFIELD_BG_TRANSPARENT_DEFAULT" Value="0x00000000" />
            <DATA_TYPE_COLOR Key="color_TEXTFIELD_FG_TRANSPARENT_DEFAULT" Value="0x00FFFFFF" />
            <DATA_TYPE_COLOR Key="color_TEXTFIELD_BG_DEFAULT" Value="0xFF000000" />
            <DATA_TYPE_COLOR Key="color_TEXTFIELD_FG_DEFAULT" Value="0xFFFFFFFF" />
            <DATA_TYPE_COLOR Key="color_TEXTFIELD_HIGHLIGHTED_DEFAULT" Value="0xFFC0C0C0" />
            <DATA_TYPE_COLOR Key="color_LIST_ITEM_SELECTED" Value="0xFF0093FF" />
            <DATA_TYPE_COLOR Key="color_LIST_ITEM_NORMAL" Value="0xFF00B0FF" />
            <DATA_TYPE_COLOR Key="color_HARDKEY_RECTANGLE" Value="0xFF8080" />
            <DATA_TYPE_COLOR Key="color_HARDKEY_TEXT" Value="0x80FFFFFF" />
            <DATA_TYPE_COLOR Key="color_MENUSHOW_TEXTFIELD" Value="0x80FFFFFF" />
            [...]
            <DATA_TYPE_FONT Key="font_DEFAULT" Value="Arial" />
            <DATA_TYPE_FONT Key="font_VERDANA" Value="Verdana.ttf" />
            <DATA_TYPE_FONT Key="font_VERDANA_BOLD" Value="VerdanaB.ttf" />
            <DATA_TYPE_FONT Key="font_VERDANA_ITALIC" Value="VerdanaI.ttf" />
            <DATA_TYPE_FONT Key="font_VERDANA_BOLD_ITALIC" Value="VerdanaZ.ttf" />
    </DataTypes>
    </DataDictionary>
</HMI>
```

4.3 Definition von Funktionen für das Gesamtsystem

Um die Spezifikation von der späteren Target-Hardware unabhängig zu halten und Simulationen des Systems zu ermöglichen, werden in IML nur abstrakte Funktionsaufrufe verwendet. Die abstrakten Funktionen werden unter *ct_Function_Definitions* definiert. Funktionalitäten, die zu einer bestimmten Hardware-Komponente gehören, werden auch bei der Definition der zugehörigen Funktionen zusammengefasst. Dabei entspricht *el_Device* bzw. *el_SubDevice* der Hardware-Komponente, zum Beispiel dem Radiotuner oder dem CD-Spieler. Jedes *el_Device* und *el_SubDevice* muss einen systemweit eindeutigen Namen besitzen. Alle Elemente in diesem Abschnitt der IML müssen zwingend eine Beschreibung (Documentation.Description) haben. Sie dient dazu Unklarheiten, die von der Namensgebung herrühren können, vorzubeugen und enthält z. B. bei *el_DeviceFunction* eine informelle Beschreibung der Funktionalität.

Unter dem *el_Device* werden die eigentlichen Funktionen aufgeführt. Diese haben einen für das Device/Subdevice eindeutigen Namen. Es besteht die Konvention, die Funktion mit DeviceName.FunctionName aufzurufen. Da die verschiedenen Komponenten häufig von unter-

schiedlichen Entwicklern bearbeitet werden, kann anders nur schwer für die absolute Eindeutigkeit der Funktionsnamen gesorgt werden.

Neben einem Namen hat eine Funktion (*el_DeviceFunction*) auch noch die Attribute *ResponseFrequency* und *ResponseType*. *ResponseFrequency* kann die Werte *NONE*, *ONCE* oder *PERMANENT* annehmen, die angeben, wie oft die Funktion auf eine Anfrage antwortet. Den Wert *NONE* bekommen Funktionen, die einen Wert in der Komponente ändern sollen und von denen keine Antwort erwartet wird. *ONCE* wird für Funktionen verwendet, die als Statusabfrage gedacht sind, wenn eine spätere Änderung des Wertes für die Abfrage keine Rolle spielt oder eine Änderung nur durch eine Aktion des Anwenders herbeigeführt werden kann (z. B. die Abfrage der Lautstärke). Wenn ein Wert sich durch äußere Einflüsse, unabhängig vom Anwender ändern kann und diese Änderungen im HMI angezeigt werden sollen, muss die passende Funktion eine *ResponseFrequency* mit dem Wert *PERMANENT* besitzen (z. B. aktuelle Frequenz beim Sendersuchlauf, Empfang eines TP-Signals).

Der *ResponseType* gibt Aufschluss darüber, ob die Funktion synchron oder asynchron auf eine Anfrage antworten wird. Dies gilt aber nur als Information für den abstrakten Platzhalter der Funktion. Die konkrete Umsetzung in Hardware kann durchaus davon abweichen.

Zu einer *el_DeviceFunction* können beliebig viele Parameter (*el_Parameter*) und ein Rückgabewert (*el_ReturnValue*) gehören. Wenn die Funktion eine Antwort liefern soll, muss der Rückgabewert definiert sein. Sowohl der Rückgabewert als auch die Parameter benötigen Angaben über ihren jeweiligen Datentyp und eine informelle Beschreibung. Sie können beide darüber hinaus auch noch Angaben über ihre möglichen Wertebereiche enthalten. Parameter benötigen außerdem eine *Direction* (IN, OUT, INOUT), welche die Richtung des Parameters angibt, einen für die Funktion eindeutigen Namen und ein Default-Value.

Um ein System lauffähig zu machen, müssen die abstrakten Funktionen durch eine jeweils angepasste Zwischenschicht (Abstraction-Layer) auf die konkreten Funktionen der Target-Hardware oder der Simulationskomponenten abgebildet werden.

Constraints:

1. Bei einer DeviceFunction muss der ResponseType angegeben werden, wenn die ResponseFrequency nicht "NONE" ist.

context el_DeviceFunction **inv**:
self.ResponseFrequency <> 'NONE' implies self.ResponseType->notEmpty()

2. Bei einer DeviceFunction muss ein el_ReturnValue existieren, wenn die ResponseFrequency nicht "NONE" ist.

context el_DeviceFunction **inv**:
self.ResponseFrequency <> 'NONE' implies self.elReturnValue->notEmpty()

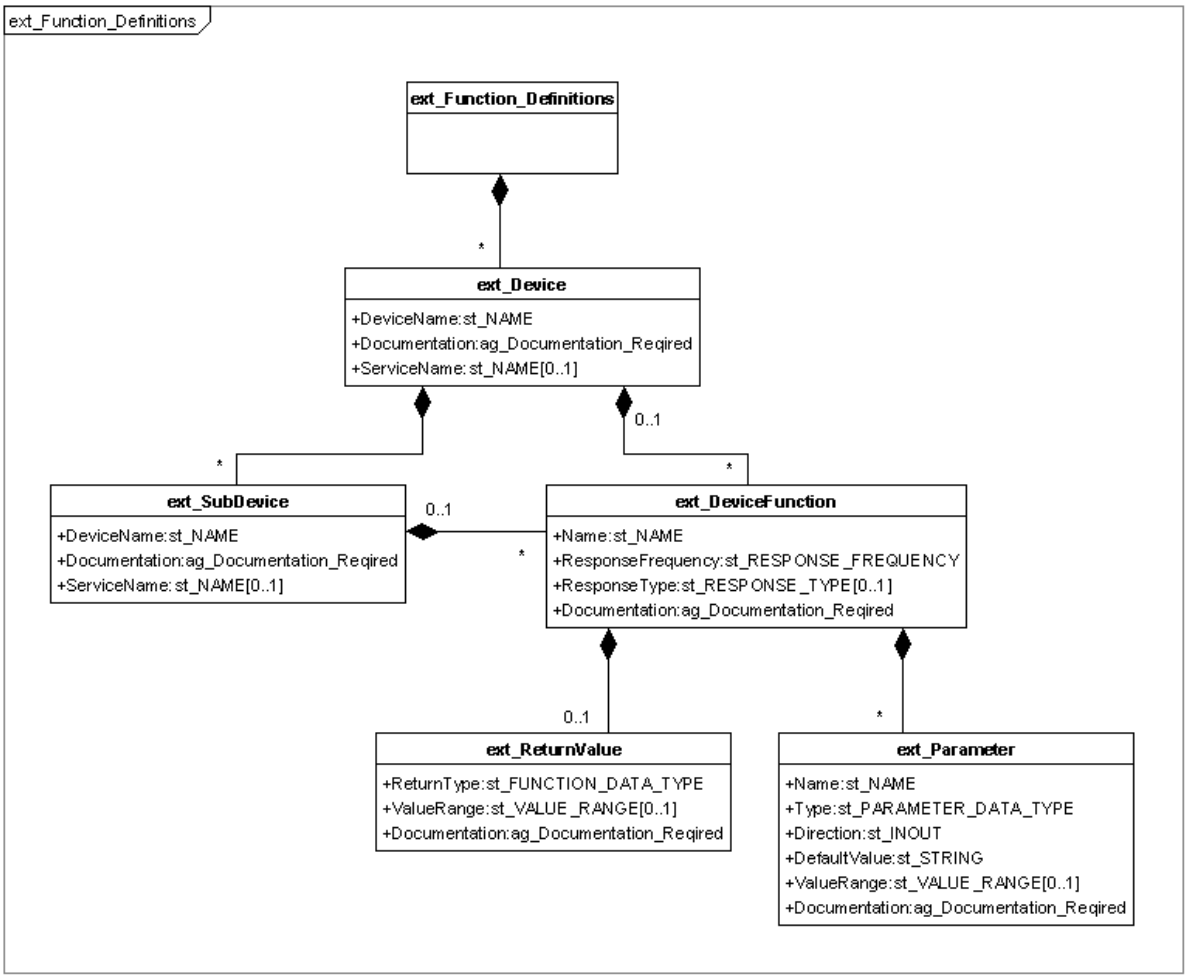

Abbildung 8 - Definition von Funktionen

Um den Aufbau einer API-Beschreibung zu verdeutlichen, sind im Beispiel unten ein paar Funktionen und zwei Hardwarekomponenten ausgewählt worden, die einen guten Überblick geben sollten.

"ActivateNextTrack" wird vom Media-Player verwendet, um den nächsten Track zu spielen. Es ist eine Funktion ohne Rückgabewert und ohne Parameter. "GetCurrentTrackInfo" liefert alle Angaben über den Track, der entweder gerade gespielt wird oder beim Betätigen von 'Play' gestartet würde. Die Daten werden aktuell gehalten, dass heißt es werden neue gesendet, wenn sich etwas ändert, z. B. beim Wechsel zum nächsten Track nach Ablauf des Aktuellen. Der Rückgabewert dieser Funktion ist ein String-Array. Die Kennzeichnung von Arrays und der Zugriff auf ihre Werte werden bei der Beschreibung der Punktnotation erklärt (siehe Kapitel 4.5). "GetElapsedTimeOfTrack" dagegen liefert einen einfachen String als Rückgabewert, der direkt verwendet werden kann. "ActivateAudioComponent" schließlich hat keinen Rückgabewert, benötigt aber einen Parameter, um aufgerufen werden zu können.

```
<HMI Name="PCM">
    <AbstractionLayer>
        <HMI_API>

            // Funktionen, die für den CD-Spieler verwendet werden
            <Device DeviceName="Audio" Description="controls the audio player" ServiceName="AUDIO">
                <DeviceFunctions>
                    <DeviceFunction FunctionName="ActivateNextTrack" FunctionResponse="NONE"
                            Description="Starts playing the next track. ">
                        <ReturnValue ReturnType="DATA_TYPE_VOID" Description="%"/>
```

```
                </DeviceFunction>
                <DeviceFunction FunctionName="GetCurrentTrackInfo" FunctionResponse="PERMANENT"
                        Description="Returns information about the current track. If there is no track name available,
                            an empty string is returned. ">
                    <ReturnValue ReturnType="DATA_TYPE_DYN_ARRAY_STRING" Description="track information"
                            ValueRange="{TrackName, ArtistName, AlbumName, TrackNumber}"/>
                </DeviceFunction>
                <DeviceFunction FunctionName="GetElapsedTimeOfTrack" FunctionResponse="PERMANENT"
                        Description="Returns the elapsed time of the current track as a string in the format 'mm:ss'.">
                    <ReturnValue ReturnType="DATA_TYPE_STRING" Description="elapsed time"
                            ValueRange="'min:sec'"/>
                </DeviceFunction>
            </DeviceFunctions>
        </Device>

        // Funktionen, die für die Regelung der Lautstärke und Ähnliches verwendet werden.
        <Device DeviceName="Sound" Description="Functions for tone, volume and other sound specific settings"
                ServiceName="SOUND">
            <DeviceFunctions>
                <DeviceFunction FunctionName="ActivateAudioComponent" FunctionResponse="NONE"
                        Description="Sets the active audio component to the selected one">
                    <ReturnValue ReturnType="DATA_TYPE_VOID" Description="%"/>
                    <Parameters>
                        <Parameter
                            Name="AudioComponent"
                            Type="DATA_TYPE_UINT"
                            Direction="IN"
                            DefaultValue="1"
                            Description="audio component"
                            ValueRange="1 - 2"
                            Comment="1 = Radio, 2 = Media"/>
                    </Parameters>
                </DeviceFunction>
            </DeviceFunctions>
        </Device>

    </HMI_API>
  </AbstractionLayer>
</HMI>
```

4.4 Nachrichten

Widgets sind abgeschlossene Einheiten, die durch Überschreibungen in ihrem Aussehen und programmierten Verhalten geändert werden können. Damit Widgets im System zusammenarbeiten können, benötigen sie eine Möglichkeit zur Kommunikation. Diese Kommunikation wird durch das Empfangen und Versenden von Nachrichten realisiert.

Jedes Widget definiert, welche Nachrichten es empfangen und welche es versenden kann. Die Nachrichten selber haben eine eindeutige ID, die auch ein sprechender, aber eindeutiger Name sein kann. Der Typ einer Nachricht wird verwendet, um ihre Wichtigkeit zu bewerten. Somit kann zum Beispiel eine kritische Systemmeldung Vorrang vor einer Benutzereingabe bekommen. Nachrichten können nicht nur von Widgets versendet werden, sondern sie sind auch eine Möglichkeit für die Systemkomponenten mit dem Benutzer zu kommunizieren, ohne auf eine Anfrage zu warten. Ein Anwendungsbeispiel ist das Eintreffen einer Verkehrsmeldung. Der Radiotuner sendet dem HMI eine Nachricht, dass eine neue Verkehrsmeldung eingetroffen ist, und das HMI aktiviert ein entsprechendes Menu, wenn dies im aktuellen Kontext gewünscht ist.

Da die Nachrichten auch für die Kommunikation zwischen dem HMI und den Systemkomponenten verwendet werden, ist eine entsprechende Dokumentation von entscheidender Bedeutung. Jeder Systemkomponenten-Entwickler muss wissen, wann seine Komponente welche Nachricht versenden muss, damit die gewünschten Darstellungen von Informationen auf dem

Bildschirm erfolgen. Aus diesem Grund ist die Angabe einer Dokumentation beim Spezifizieren einer Nachricht zwingend nötig.

Um die Anzahl der spezifizierten Nachrichten in einem überschaubaren Rahmen zu halten und die Wiederverwendbarkeit der Widgets zu sichern, können Nachrichten parametrisiert werden. In diesen Parametern können entweder direkt die darzustellenden Informationen mit versendet werden oder sie können zur genaueren Identifikation des Senders dienen. Wenn in einem Menü mehrere Buttons existieren und einer davon betätigt wird, muss das Menü unterscheiden können von welchem eine eingehende "Pressed" Nachricht stammt, da davon die Reaktion des Menüs abhängt.

Da an dieser Stelle nur das Gerüst der Nachrichten definiert wird, hat ein Parameter zwar einen für die Nachricht eindeutigen Namen, einen Typ und eventuell eine Beschreibung welche Daten der Parameter aufnehmen soll, aber noch keinen konkreten Wert. Werte werden den Nachrichten erst in den Widgets oder beim Versenden durch die System-Komponenten zugeordnet.

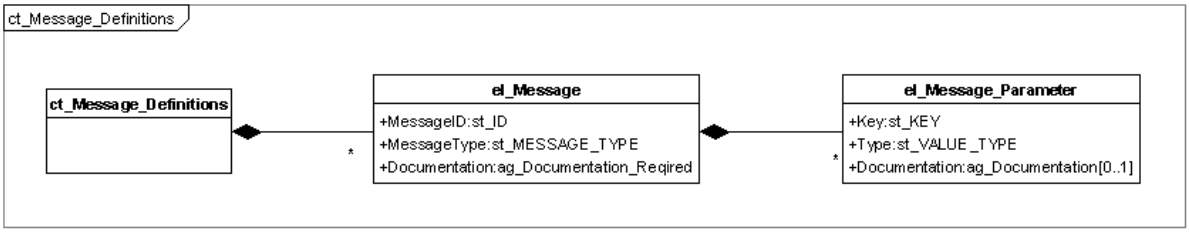

Abbildung 9 - Definition von Nachrichten

4.5 Punktnotation

Die Attribute "Value" von *el_Value* und "Term" von *el_Term* können nicht nur feste Werte enthalten, sondern auch Konstanten, Referenzen auf dynamische Quellen wie zum Beispiel Funktionsaufrufe oder auf Werte, die in anderen Widgets spezifiziert sind. Um all dies beschreiben zu können, wurde die Punktnotation entwickelt. Da die Punktnotation eine rein textuelle Darstellungsform hat, wird sie hier durch ihre EBNF Spezifikation erklärt. Die hier angegebene EBNF-Spezifikation ist eine überarbeitete Version derjenigen, die für die Programmierung eines Parsers in der IAV GmbH erstellt wurde. [M&K 1]

Die Bedeutung der verwendeten Formulierungen sind folgende:

- A B –> Ausdruck A wird gefolgt von Ausdruck B,
- A | B –> Ausdruck A oder Ausdruck B können auftreten,
- [A] –> Ausdruck A kann ein- oder keinmal auftreten,
- {A} –> Ausdruck A kann beliebig oft, auch keinmal, auftreten,
- "A" –> Das Zeichen 'A' oder eine feste Zeichenfolge,
- A - B –> Alles, was durch A erlaubt wird, außer allem, das durch B erlaubt wird,
- ; –> Bezeichnet das Ende einer Produktionsregel,

- A = B C; –> Grundstruktur einer Produktionsregel. Wenn A als Ausdruck vorliegt, kann er durch den Ausdruck B gefolgt von Ausdruck C ersetz werden.

// CHARACTERS

Um die Produktionsregeln leichter beschreiben zu können werden bestimmte Gruppen von Zeichen zusammengefasst und benannt. 'ANY' bezeichnet dabei alle überhaupt möglichen Zeichen. Als nötige Untergruppen werden die Ziffern (digit), für Namen verwendbare Zeichen (tchar) und die weiter eingeschränkte Menge der Zeichen, die in einem Namen an erster Stelle stehen dürfen (tchar1st). Außerdem werden alle Zeichen zusammengefasst, die in der Punktnotation als Sonderzeichen verwendet werden und nicht in Namen auftauchen dürfen (xchar) und besondere Steuerzeichen (whitespace).

```
digit       = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";

xchar       = "/" | "." | "{" | "}" | "[" | "]" | "+" | "-" | "*" | "(" | ")" | "," | "" | "\" | ";";

whitespace  = " " | "\n" | "\r" | "\t";

tchar       = ANY - xchar - whitespace;

tchar1st    = tchar - digit - whitespace;
```

// TOKEN

Token stellen einen elementaren Ausdruck einer Produktionsregel dar. Sie bestehen aus einem oder mehreren Zeichen (Characters).

```
nametoken   = tchar1st { tchar };

ws          = whitespace { whitespace };
```

// PRODUCTIONS

Das *ImlValue* beschreibt den möglichen Inhalt von *Value* oder *Term*. Ein ImlValue beinhaltet entweder einen festen Wert (Content), der keiner weiteren Einschränkung unterliegt, ein Feld aus mehreren Werten (Array) oder einen "PunktAusdruck" (IdentExpression). Die Bezeichnung "Punktausdruck" haben die so benannten Ausdrücke dem führenden Punkt in ihren Schlüsselwörtern zu verdanken.

```
ImlValue    = Array [ws]
            | IdentExpression [ws]
            | Content;

Content     = {ANY};
```

Arrays werden hauptsächlich dann verwendet, wenn eine unbekannte Anzahl von gleichen Angaben benötigt wird. Das ist vor allem in statischen Listitems der Fall. Um das Definieren von Listen zu erleichtern, wurde beschlossen, dass Listitems mit gleichem Aufbau zusam-

mengefasst werden können. Das bedeutet, dass zum Beispiel eine Liste mit Radiosendern bei der Definition nur ein Listitem beinhaltet, bei dem dann die Anzahl der anzuzeigenden Items spezifiziert wird. Wenn der Inhalt eines solchen Listitems nicht dynamisch ist, wird er als Array im Property *StaticContent* definiert. Ein Array steht zwischen geschweiften Klammern und die einzelnen Werte werden durch Semikolon voneinander getrennt. Feste Werte in einem Array werden in einfachen (Quote1Content) oder doppelten Hochkomata (Quote2Content) geschrieben, außer es handelt sich bei den Werten um Zahlen (NumContent). Diese werden ohne Hochkommata geschrieben. Außer den festen Werten können auch Punkt-Ausdrücke (IdentExpression) als Elemente auftreten.

```
Array               = "{" ArrayContent "}";

ArrayContent        = [ QuoteNumExpression [ws] { ";" [ws] [ QuoteNumExpression [ws] ] } ];

QuoteNumExpression = IdentExpression
                    | NumContent
                    | Quote1Content
                    | Quote2Content;

QuoteExpression     = IdentExpression
                    | Quote1Content
                    | Quote2Content;

Quote1Content       = "'" {ANY} "'";

Quote2Content       = "\"" {ANY} "\"";

NumExpression       = IdentExpression
                    | NumContent;

NumContent          = ["-"] digit {digit};

PosNumExpression = IdentExpression
                    | PosNumContent;

PosNumContent       = digit {digit};
```

IdentExpressions sind die Kernelemente der Punktnotation. Sie zeigen an, dass es sich bei einem Ausdruck um eine Punktnotation handelt und sie bestimmen welche Art Punktnotation in einem bestimmten Fall vorliegt.

```
IdentExpression     = LinkExpression
                    | CalcExpression
                    | DictExpression
                    | LangExpression
                    | ResExpression
                    | RepExpression
                    | FormatExpression
                    | StrExpression;
```

LinkExpressions werden verwendet, um auf Elemente eines Widgets zugreifen zu können. Dabei müssen das Widget, in dem sich die LinkExpression befindet, und das Ziel-Widget nicht übereinstimmen. Sie müssen jedoch in einem hierarchischen Zusammenhang stehen. Das Ziel-Widget muss ein direktes oder weiter entferntes Kind-Widget des Widgets sein, in dessen Datei sich die LinkExpression befindet. Wenn die LinkExpression als Überschreibung

in einem gemeinsamen Eltern-Widget verwendet wird, so können auch Elemente von benachbarten Widgets verbunden werden.

```
LinkExpression      = ".LINK" "{" LinkTarget "}";

LinkTarget          = Path TargetElement;

Path                = ["/" | (["./"] {"../"})] {nametoken "/"};

TargetElement       = ".PROPERTY" "[" PropertyName "]" Index
                    | ".DATA" "[" nametoken "]" Index
                    | ".ATTRIBUTE" "[" AttributeName "]"
                    | ".STATE"
                    | ".ANCHOR" "[" AnchorName "]"
                    | ".FUNCTION" "[" nametoken "]" Index;

AttributeName       = ("Position" "." ("X" | "Y")) | ("Dimension" "." ("Width" | "Height"));

AnchorName          = ("Position" "." ("X" | "Y")) | ("Alignment" "." ("Horizontal" | "Vertical"));

Index               = ["[" PosNumExpression "]"];

PropertyName        = nametoken {"." nametoken};

RepositoryKey       = "/" nametoken {"/" nametoken};
```

Mit der CalcExpression ist es möglich, in Widgets einfache Rechnungen auszuführen. Viele dieser Rechnungen könnten auch durch feste Werte in Überschreibungen ersetzt werden und dies geschieht auch als Optimierung bei der Portierung auf das Target-System. Für den Spezifikateur ist es jedoch eine große Erleichterung, wenn eingefügte Element zum Beispiel ihre Größe selber berechnen und sie nicht explizit für jede Instanz gesetzt werden muss.

```
CalcExpression      = ".CALC" "{" [ws] CalcTerm "}";

CalcTerm            = (NumExpression | BracketTerm) [ws]
                    {Operator [ws] (NumExpression | BracketTerm) [ws]};

Operator            = "+" | "-" | "*" | "/";

BracketTerm         = "(" [ws] CalcTerm ")";
```

FormatExpression und StringExpression werden beide zur Modifikation von Strings verwendet. Die StringExpression kann mehrere Strings kombinieren, indem sie sie in der angegebenen Reihenfolge aneinander fügt. Bei der FormatExpression werden die in der ersten QuoteExpression enthaltenen Platzhalter mit dem Inhalt der folgenden ersetzt. So ist es möglich zu berücksichtigen, dass Informationen nicht in allen Sprachen in der gleichen Reihenfolge dargestellt werden.

```
FormatExpression    = ".FORMAT" "{" [ws] QuoteExpression [ws] "," [ws] FormatContent "}";

FormatContent       = QuoteNumExpression [ws] {"," [ws] QuoteNumExpression [ws]};

StrExpression       = ".STR" "{" [ws] StrContent "}";

StrContent          = QuoteExpression [ws] {"+" [ws] QuoteExpression [ws]};
```

Die verbleibenden Expression werden verwendet, um Werte aus dem DataDictionary, der Sprachdatei, aus der Ressourcendatei oder dem Repository zu verwenden. Das Repository ist dabei die einzige Quelle, die laufzeitabhängige Daten enthalten kann. Es entspricht einem shared Memory und wird bei Systemen verwendet, bei denen Speicher gegenüber Rechenleistung das kleinere Problem darstellt.

```
DictExpression      = ".DICT" "{" nametoken "}";

LangExpression      = ".LANG" "{" nametoken "}";

ResExpression       = ".RES" "{" nametoken "}";

RepExpression       = ".REP" "{" RepositoryKey "}";
```

Im verwendeten Beispiel soll in der Statuszeile die Temperatur im Innenraum des Fahrzeugs angezeigt werden. Die Funktion GetTemperature liefert die aktuelle Temperatur als numerischen Wert. Dieser wird durch die .FORMAT Anweisung mit der passenden Einheit kombiniert.

```
.FORMAT{.LANG{txt_TEMPERATURE_STR}, .LINK{../.FUNCTION[GetTemperature]}}
```

In diesem Beispiel der Punktnotation befinden sich drei unterschiedliche Anwendungen.
".LANG{txt_TEMPERATURE_STR}" liefert den Inhalte der Text-Id
"txt_TEMPERATURE_STR": " %1 ℃" .

".LINK{../.FUNCTION[GetTemperature]}" stellt eine Verbindung zur Funktion GetTemperature her, deren Rückgabewert dann durch das .FORMAT in einen String umgewandelt und die Stelle des Platzhalters "%1" gesetzt wird.

4.6 Widgets

IML betrachtet die grafische Beschreibung von Gesamtsystemen ebenso wie ihre Unterkomponenten als so genannte Widgets. Ein Widget spezifiziert dabei das Aussehen und das Verhalten eines Objekts gleichgültig, ob es sich dabei um etwas Komplexes wie einen Schieberegler oder eine einfache Linie handelt. Ein Widget wird aufgeteilt in seine Eigenschaften (Properties), Kind-Widgets (Elements), aus denen es zusammengesetzt ist, seine Zuständen (States) und sein Verhalten (Behavior). Es gibt strukturell betrachtet zwei unterschiedliche Arten von Widgets: die elementaren Base-Widgets und die komplexen Widgets.

Alle Widgets haben einen projektweit eindeutigen Namen. Um die maschinelle Verarbeitung zu erleichtern, kann neben dem Namen auch noch eine ID angegeben werden. Die Verwendung eines sprechenden Namens und einer ID erfüllt sowohl die Ansprüche maschineller Verarbeitung als auch die von menschlichen Entwicklern.

Ebenso, wie alle Widget einen Name haben müssen, haben sie auch einen Typ. Ein Widget kann vier verschieden Typen annehmen, die ebenfalls für die maschinelle Verarbeitung von Bedeutung sind. Der Typ "WIDGET_TYPE_NORMAL" ist der am häufigsten verwendete. Widgets dieses Typs haben kein spezielles, ausprogrammiertes Verhalten, sondern beschreiben hauptsächlich das Aussehen von Objekten. Sie haben dennoch ein in IML beschriebenes Verhalten.

"WIDGET_TYPE_BASE" ist der Typ aller Base-Widgets. Sie stellen die Grundelemente aller anderer Widgets dar. Blätter im Widget-Baum sind meist Base-Widgets. Diese haben einen anderen Aufbau als komplexe Widgets und werden später noch näher beschrieben (siehe Kapitel 4.6.3).

Die Widgets vom Typ "WIDGET_TYPE_TARGET_LIBRARY" haben sowohl eine grafische Repräsentation und ein Verhalten in IML, als auch ein ausprogrammiertes Verhalten. Diese Widgets sind hauptsächlich in den tieferen Schichten des Widget-Baumes zu finden.

Der letzte Type, den ein Widget annehmen kann ist der "WIDGET_TYPE_FUNCTION". Widgets dieses Typs haben keine grafische Repräsentation. Ein häufig verwendetes Beispiel wäre ein Timer. Auch diese Widgets können als Blätter im Widget-Baum auftreten.

Um die Zusammengehörigkeit von *el_Definition_Root* und *el_Implementation_Root* zu verdeutlichen, wurden sie *seq_Complex_Root* untergeordnet. *seq_Complex_Root* ist ein Beispiel für ein Element des Modells, das keine direkte Entsprechung im Schema besitzt. Um zum gleichen Ergebnis zu kommen, hätte auch ein entsprechendes Constraint eingeführt werden können. Die Einführung des Elements schien jedoch anschaulicher. Das Element ct_Widget_Definition wurde als abstrakte Klasse spezifiziert, da sowohl ct_Base_Definition als Wurzelelement des Base-Widgets als auch seq_Complex_Root als Wurzelelement des komplexen Widgets dieselben Daten enthalten und diese von ct_Widget_Definition erben können.

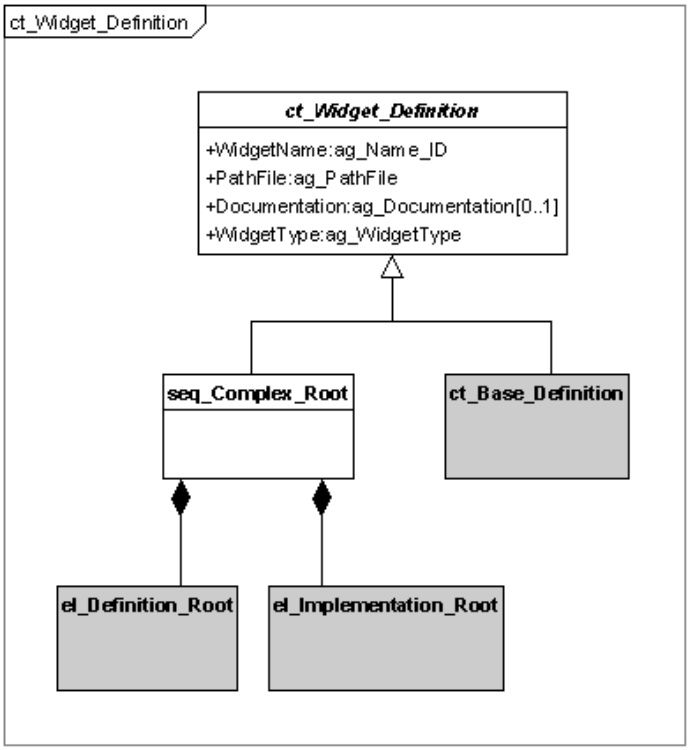

Abbildung 10 - Grundstruktur von Widgets

Zur Veranschaulichung seines Aufbaues ist an dieser Stelle die IML Darstellung eines vollständigen Widgets aufgeführt. Bei der Erläuterung der einzelnen Elemente werden dann nur noch Ausschnitte verwendet. Wenn es möglich ist, werden die Abschnitte aus diesem Widget

stammen. Einige Aspekte von IML wurden in diesem Menü nicht verwendet. Für diese Aspekte werden Beispiele aus anderen Widgets aufgeführt.

```
<Widget Name="PCM_OFF" Type="WIDGET_TYPE_MENU" >
    // Der Definitionsteil spezifiziert, welches Kind-Widgets verwendet werden und welche Zustände
    das Widget annehmen kann
    <Definition>

        <Elements>
            <Element Name="Background">
                <Widget Name="Image" Path=".DICT{path_BASE_WIDGET}" File="Image.xml"/>
            </Element>
            <Element Name="OFF_Message">
                <Widget Name="TextField" Path=".DICT{path_BASE_WIDGET}" File="Textfield.xml"/>
            </Element>
        </Elements>

        <States>
            <State Name="BlackRed"/>
            <State Name="Beauty"/>
        </States>

    </Definition>

    // Im Implementationsteil werden die Werte der eigenen Propertys gesetzt und die Werte der Propertys der Kinder
    abhängig oder unabhängig vom aktuellen Zustand überschrieben. Außerdem wird dort auch das Verhalten des Widgets
    beschrieben.
    <Implementation>

        // Die eigenen Propertys
        <Properties>
            <StateProperty>
                <State Name="BlackRed"/>
            </StateProperty>
        </Properties>

        // Das Setzen der Propertys der Kinder unabhängig von den eigenen States
        <Defaults>
            <Elements>

                // Der Hintergrund des Widgets besteht aus einem Bild, dessen Dateiname hier angegeben wird.
                <Element Name="Background">
                    <Region>
                        <Position>
                            <Absolute X=".DICT{pos_NULL_X}" Y=".DICT{pos_NULL_Y}"/>
                        </Position>
                        <Dimension Height=".DICT{height_MENU}" Width=".DICT{width_MENU}"/>
                    </Region>
                    <Existence Load="ONCE" Unload="ONCE" Visible="true" Active="true"/>
                    <Widget Name="Image">
                        <Base Name="Image">
                            <Properties>
                                <Property Name="ImageProperties">
                                    <Property Name="Left">
                                        <Property Name="File">
                                            <Values>
                                                <FixedValue>
                                                    <Value Value=".DICT{file_OFF_BACKGROUND}"/>
                                                </FixedValue>
                                            </Values>
                                        </Property>
                                    </Property>
                                </Property>
                            </Properties>
                        </Base>
                    </Widget>
                </Element>

                // Außer dem Hintergrundbild existiert in diesem Menü noch ein Textfeld,
                dessen Inhalt und Aussehen hier gesetzt werden.
                <Element Name="OFF_Message">
                    <Region>
                        <Position>
                            <Absolute X=".DICT{pos_NULL_X}" Y="80"/>
                        </Position>
                        <Dimension Height="34" Width="400"/>
```

```
                </Region>
                <Existence Load="ONCE" Unload="ONCE" Visible="true" Active="true"/>
                <Widget Name="TextField">
                    <Base Name="TextField">
                        <Properties>
                            <Property Name="TextFieldProperties">

                                // Der Inhalte des Textfeldes
                                <Property Name="Content">
                                    <Values>
                                        <FixedValue>
                                            <Value Value="O F F"/>
                                        </FixedValue>
                                    </Values>
                                </Property>

                                // Die verwendete Schriftfarbe
                                <Property Name="Color">
                                    <Property Name="Foreground">
                                        <Values>
                                            <FixedValue>
                                                <Value Value=".RES{ color_MENUSHOW_TEXTFIELD_1}"/>
                                            </FixedValue>
                                        </Values>
                                    </Property>
                                </Property>

                                // Die verwendete Schriftgröße
                                <Property Name="Font">
                                    <Property Name="Size">
                                        <Values>
                                            <FixedValue>
                                                <Value Value=".RES{uint_FontSize_16}"/>
                                            </FixedValue>
                                        </Values>
                                    </Property>
                                </Property>
                            </Property>
                        </Properties>
                    </Base>
                </Widget>
            </Element>
        </Elements>
    </Defaults>

    // Das Setzen der Propertys der Kinder abhängig von den eigenen States
    <States>
        <ElementStates>
            <State Name="Beauty">

                // Im State "Beauty" wird ein anderes Hintergrundbild verwendet.
                <Element Name="Background">
                    <Widget Name="Image">
                        <Base Name="Image">
                            <Properties>
                                <Property Name="ImageProperties">
                                    <Property Name="Left">
                                        <Property Name="File">
                                            <Values>
                                                <FixedValue>
                                                    <Value Value=".DICT{file_OFF_BACKGROUND_BEAUTY}"/>
                                                </FixedValue>
                                            </Values>
                                        </Property>
                                    </Property>
                                </Property>
                            </Properties>
                        </Base>
                    </Widget>
                </Element>

                // Auch der Inhalt des Textfeldes und die Schriftfarbe werden angepasst
                <Element Name="OFF_Message">
                    <Widget Name="TextField">
                        <Base Name="TextField">
                            <Properties>
                                <Property Name="TextFieldProperties">
```

```
                                    <Property Name="Content">
                                        <Values>
                                            <FixedValue>
                                                <Value Value="PCM Ausgeschaltet"/>
                                            </FixedValue>
                                        </Values>
                                    </Property>
                                    <Property Name="Color">
                                        <Property Name="Foreground">
                                            <Values>
                                                <FixedValue>
                                                    <Value Value=".RES{ color_MENUSHOW_TEXTFIELD_2}"/>
                                                </FixedValue>
                                            </Values>
                                        </Property>
                                    </Property>
                                </Property>
                            </Properties>
                        </Base>
                    </Widget>
                </Element>
            </State>
        </ElementStates>
    </States>

    // Das Verhalten des Widgets ist aufgeteilt in die verwendeten Funktionen (hier keine), die angelegten Variablen zum
    Zwischenspeichern von Informationen (hier ebenfalls nicht vorhanden), den möglichen eingehenden und ausgehenden
    Nachrichten und der StateTable, die sie miteinander verbindet.
    <Behavior>
        <InOut>

            // Die eingehenden Nachrichten. Das Widget kann nur auf die hier aufgeführten Nachrichten reagieren.
            <InMessages>
                <Message Name="In_ON" Type="MESSAGE_TYPE_NORMAL"
                        MessageID=".DICT{hmi_msg_KEY_PRESSED}">
                    <Parameters>
                        <ParameterInMessage Key="Name" ValueType="DATA_TYPE_STRING">
                            <Value Value=".DICT{param_KEY_PRESSED_a}"/>
                        </ParameterInMessage>
                    </Parameters>
                </Message>
                <Message Name="In_StartScreen" Type="MESSAGE_TYPE_NORMAL"
                        MessageID=".DICT{hmi_msg_KEY_PRESSED}">
                    <Parameters>
                        <ParameterInMessage Key="Name" ValueType="DATA_TYPE_STRING">
                            <Value Value=".DICT{param_KEY_PRESSED_q}"/>
                        </ParameterInMessage>
                    </Parameters>
                </Message>
            </InMessages>

            // Die ausgehenden Nachrichten. Das Widget verwendet diese Nachrichten, um andere Widget über Ereignisse
            zu informieren.
            <OutMessages>
                <Message Name="Out_ON" Type="MESSAGE_TYPE_NORMAL"
                        MessageID=".DICT{hmi_msg_MENU_LEAVE}">
                    <Parameters>
                        <ParameterOutMessage Key="Event" ValueType="DATA_TYPE_STRING">
                            <Value Value=".DICT{param_KEY_PRESSED_a}"/>
                        </ParameterOutMessage>
                    </Parameters>
                </Message>
                <Message Name="Out_StartScreen" Type="MESSAGE_TYPE_NORMAL"
                        MessageID=".DICT{hmi_msg_MENU_LEAVE}">
                    <Parameters>
                        <ParameterOutMessage Key="Event" ValueType="DATA_TYPE_STRING">
                            <Value Value=".DICT{param_KEY_PRESSED_q}"/>
                        </ParameterOutMessage>
                    </Parameters>
                </Message>
            </OutMessages>

            // In der StateTable wird festgelegt, wie ein Widgt auf eine eingehende Nachricht reagiert.
            <StateTable>
                <Transitions>

                    <Transition Name="ON">
```

```
                        <InMessage MessageNameReference="In_ON"/>
                        <OutMessages>
                            <OutMessage MessageNameReference="Out_ON"
                                    Propagation="PROPAGATION_TYPE_PARENT"/>
                        </OutMessages>
                    </Transition>

                    <Transition Name="StartScreen">
                        <InMessage MessageNameReference="In_StartScreen"/>
                        <OutMessages>
                            <OutMessage MessageNameReference="Out_StartScreen"
                                    Propagation="PROPAGATION_TYPE_PARENT"/>
                        </OutMessages>
                    </Transition>

                </Transitions>
            </StateTable>
        </InOut>
    </Behavior>
</Implementation>
</Widget>
```

4.6.1 Wiederkehrende Strukturen

In diesem Abschnitt werden Strukturen von IML beschrieben, die mehrfach, an verschiedenen Stellen des Modells vorkommen. An den jeweiligen Stellen wird dann nur noch auf diese Beschreibung verwiesen.

4.6.1.1 Die Attributgruppen

Im Gegensatz zu den normalen Klassen des Metamodells, werden die Attributgruppen in IML nicht durch XML-Elemente sondern durch XML-Attribute repräsentiert. Bei der Metamodellierung wurden sie durch das Präfix "ag_" gekennzeichnet und werden nicht durch Aggregation mit ihrer Eltern-Klasse verbunden, sondern als ein Attribut der Eltern-Klasse angelegt.

Die am häufigsten verwendete Attributgruppe ist *ag_Name_ID*. Sie fast die beiden Attribute zusammen, die für die Identifikation von Elementen in IML verwendet werden, *Name* und *ID*. Das Attribut *Name* ist zwingend erforderlich, während das Attribut *ID* nicht gesetzt werden muss. Dass *Name* als erforderliches Attribut gewählt wurde, ist mit der vorherrschenden Bearbeitungsmethode der IML-Spezifikationen zu erklären. Da es noch kaum Werkzeugunterstützung zur Erstellung gibt, sind sprechende, für Menschen verständliche Namen wichtig. *ID* wurde als zusätzliche Eigenschaft eingeführt, um nebenher auch eine maschinenfreundliche Identifikation zu ermöglichen. Da beide Attribute mit beinahe beliebigen Zeichenketten gefüllt werden können, kann bei einem werkzeugunterstütztem System eine eindeutige Ziffer oder sonstige Kodierung anstelle des sprechenden Namens gewählt werden. Die verwendeten Zeichenketten unterliegen lediglich den Einschränkungen, denen das Nametoken der Punktnotation unterworfen ist, da über das Attribut Name in der Punktnotation auf diese benannten Elemente zugegriffen wird.

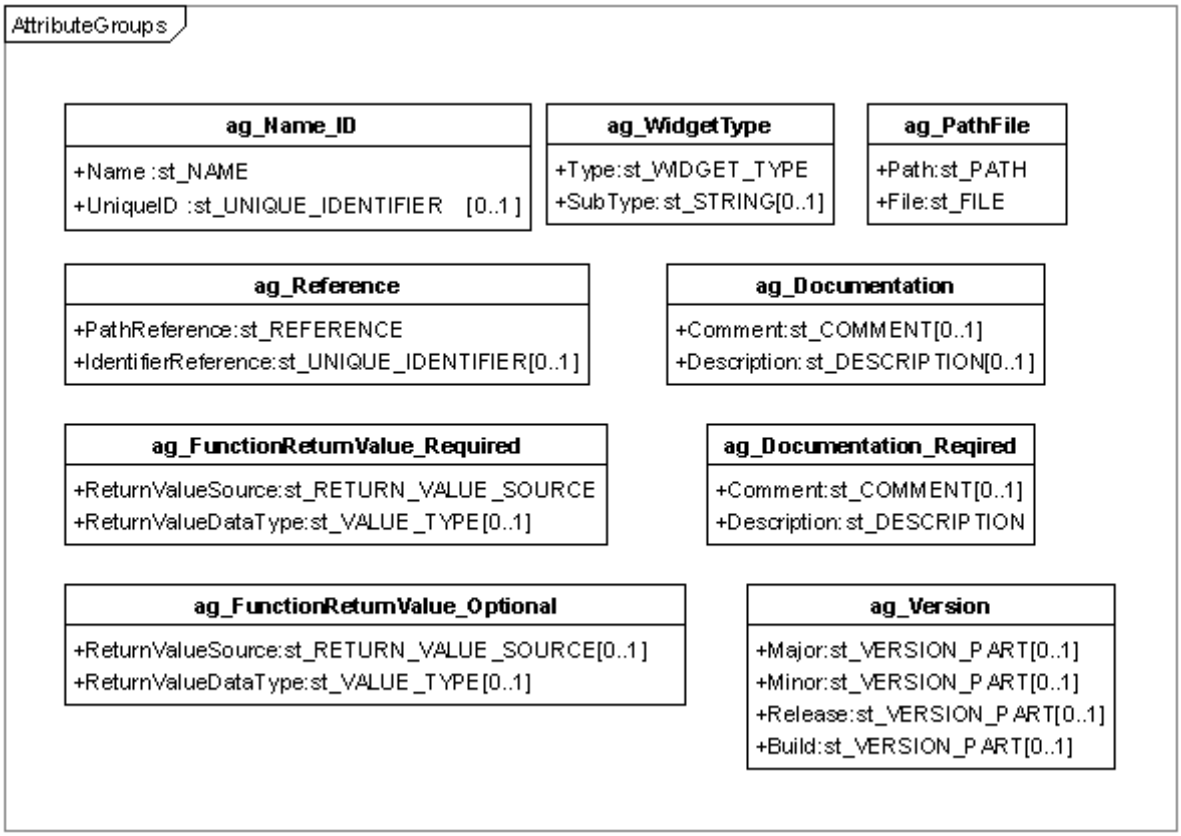

Abbildung 11 - Die Attributgruppen

4.6.1.2 Bedingungen

Um die Darstellung des Infotainmentsystems möglichst variabel zu gestalten, ist es häufig nötig, aktuelle Systemeinstellungen oder externe Gegebenheiten zu beachten. Dies kann man in IML erreichen, in dem man Bedingungen (*el_Condition*) verwendet. Bedingt werden können dabei die Wertebelegung von Propertys und Variablen, Transitionen und Funktionsaufrufe.

Bei Propertys und Variablen wird durch die Bedingung festgelegt, welcher der angegebenen Werte unter den jeweiligen Umständen zu verwenden ist. Bei Transitionen und Funktionsaufrufen dienen sie zur Entscheidung, ob diese jeweils ausgeführt werden oder nicht. Dadurch sind bedingte Zustandsübergänge möglich.

Man könnte argumentieren, dass die bedingten Werte bei Propertys auch durch die Einführung von neuen Zuständen und bedingten Transitionen zu realisieren sind. Dies ist tatsächlich möglich. Allerdings hat ein solches Vorgehen auch einige Nachteile. IML kennt keine parallelen Zustände. Das heißt, dass anstatt n verschiedener Propertys mit jeweils zwei möglichen Werten, würden n^2 Zustände benötigt. Da es oft auch mehr als zwei mögliche Werte bei einem Property gibt, wird die Anzahl der Zustände schnell extrem groß und damit unübersichtlich. Abgesehen von der Anzahl der Zustände würde gleichzeitig auch die Anzahl der Transitionen stark ansteigen. Deshalb werden Zustände hauptsächlich dann verwendet, wenn mehrere Propertys die gleiche Bedingung verwendet. Die endgültige Entscheidung für oder gegen die Verwendung von Bedingungen wird von den Entwicklern getroffen.

Eine Bedingung hat einen Namen (*ConditionName*), der als Typ die Klasse ag_Name_ID hat. Diese Attributgruppe ist der Typ, der für die Namen aller Elemente in IML verwendet wird. Er beinhaltet zwei Elemente, *Name* und *UniqueID*.

Der Name muss in einer Gruppe von Bedingungen eindeutig sein. Es ist von Vorteil eine Beschreibung der Bedingung als Namen zu wählen. Die Bedingung kann zwei verschiedene Elemente enthalten. Das eine ist eine informelle Bedingung (*el_InformalCondition*), die einen Ausdruck, der nicht IML-konform ist, enthält, oder eine Expression (*el_Expression*). Die informelle Bedingung kann mit hoher Wahrscheinlichkeit nicht maschinell verarbeitet werden. Sie sollte daher nur zur vorübergehenden Definition verwendet werden, zum Beispiel dann, wenn das Interface einer benötigten Funktion noch nicht feststeht.

Eine Expression enthält bis zu vier Elemente: einen UnaryOperator (*el_UnaryOperator*), der den gesamten Ausdruck beeinflusst, ihn zum Beispiel negiert. Mindestens eine, höchsten aber zwei weiter Expression oder zwei Term(e) (*el_Term*) und einem BinaryOperator (*el_BinaryOperator*), der eine Verbindung zwischen zwei Expressions darstellt (AND, OR) oder einen Vergleich zwischen zwei Termen ermöglicht (EQUAL, GREATER usw.). Ein Term kann einen festen Wert enthalten oder über die Punktnotation mit dem Wert eines Propertys oder einer Variablen oder mit einer Funktion verbunden sein.

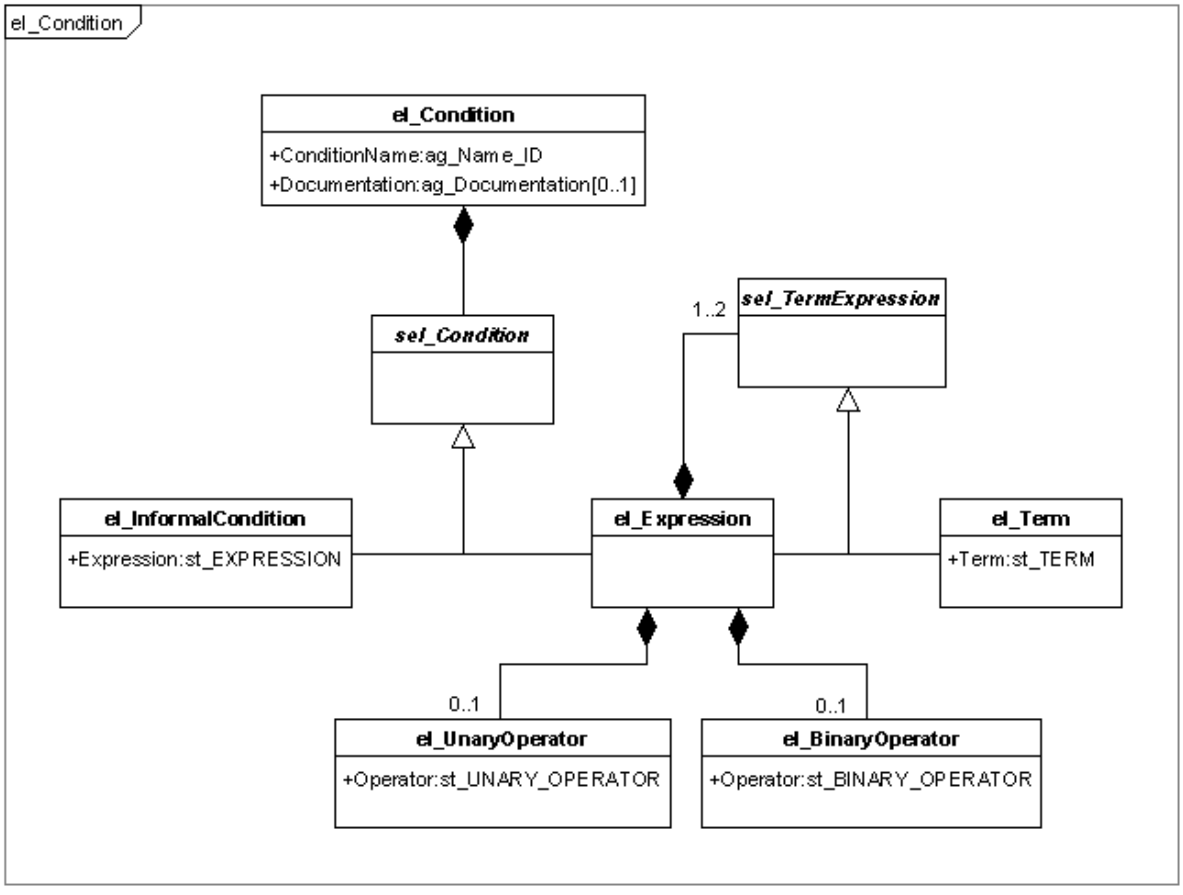

Abbildung 12 - Aufbau einer Bedingung

Im Beispielsystem existiert in der Statuszeile ein Textfeld, das den Benutzer darüber informiert, welche Audioquelle gerade verwendet wird. Dazu muss entschieden werden, ob gerade das Radio spielt, damit dann der Name des aktiven Senders angezeigt wird, oder ob eine CD

abgespielt wird. Dann wird der Name des Musikstücks angezeigt. Der Inhalt des Textfeldes wird durch die Auswertung von Bedingungen bestimmt.

```
<Property Name="Content">
    <Values>

        <ConditionalValue>
            <Condition Name="Radio_Active">
                <Expression>
                    <Term Term=".LINK{../.FUNCTION[GetMediaSource]}"/>
                    <BinaryOperator Operator="EQUAL"/>
                    <Term Term=".DICT{uint_RADIO}"/>
                </Expression>
            </Condition>
            <Value Value=".LINK{../.FUNCTION[GetActiveStationInfo][0]}"/>
        </ConditionalValue>

        <ConditionalValue>
            <Condition Name="Media_Active">
                <Expression>
                    <Term Term=".LINK{../.FUNCTION[GetMediaSource]}"/>
                    <BinaryOperator Operator="EQUAL"/>
                    <Term Term=".DICT{uint_MEDIA}"/>
                </Expression>
            </Condition>
            <Value Value=".LINK{../.FUNCTION[GetTrackInfo][0]}"/>
        </ConditionalValue>

        <DefaultValue>
            <Value Value=".LANG{txt_NO_CONTENT}"/>
        </DefaultValue>
    </Values>
</Property>
```

Das Modell der ersten Bedingung ("Radio_Active") sieht wie folgt aus:

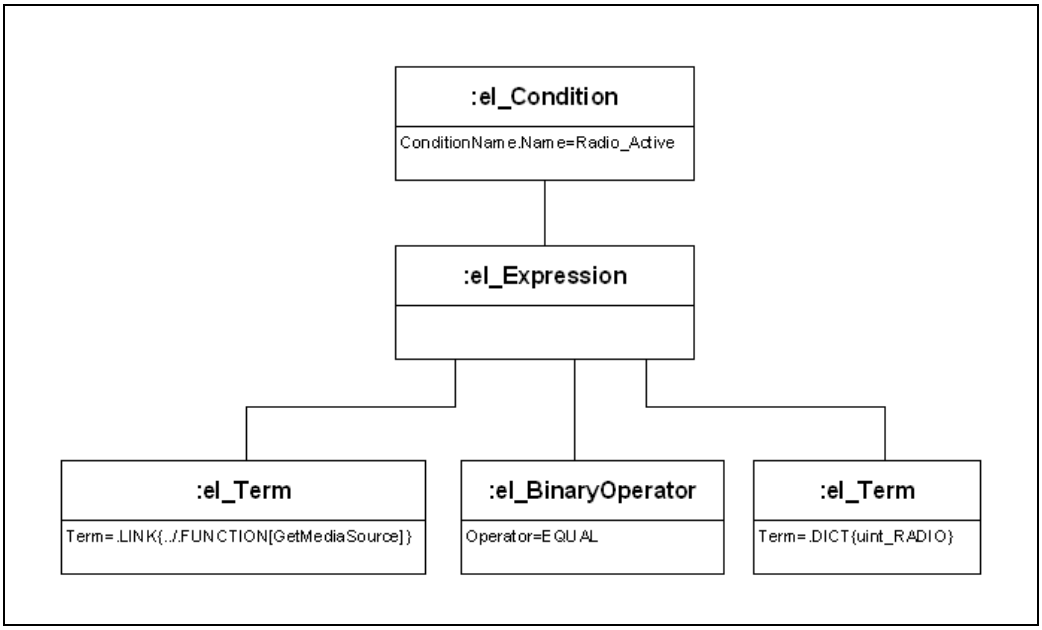

Abbildung 13 - Modell einer Bedingung

4.6.1.3 Values und Referenzen

Die Beschreibung von Value und Reference ist schwierig, da ihre gewollte und ihre wirkliche Verwendung voneinander abweichen. Hier wird die Verwendung dieser Elemente beschrieben, da dies im Sinne des Verständnisses von existierenden Projekten geeigneter erscheint. Durch die veränderte Bedeutung der Elemente sind ihre Namen und ihre Attribute zum Teil ungenau bzw. unpassend. Um das Modell und IML nicht zu weit auseinander laufen zu lassen, wurden sie dennoch nicht geändert.

Value (*el_Value*) und Referenzen (*el_Reference*) können in Propertys, Messages und Datas verwendet werden. In allen existierenden Projekten werden Referenzen jedoch nur in InMessages verwendet, überall sonst erscheinen nur Values.

In der Bedeutung, in der sie verwendet werden, enthalten Values entweder einen festen Wert oder einen Ausdruck der Punktnotation, der auf ein anderes Value oder einem Wert aus einer der Definitionsdateien verweist oder den Aufruf einer Funktion enthält. Die Werte, die der Link zurück liefert, müssen einen Datentyp haben, der mit dem spezifizierten Datentyp des Propertys, Datas oder des Messageparameters übereinstimmt oder in ihn überführt werden kann. Values, die mit der Punktnotation kombiniert wurden, haben die Eigenschaft sich die Werte am Ende des Pfades zu Eigen zu machen. Dass heißt, sie übernehmen den Wert vom Ziel des Pfades. Eine Änderung des Ursprünglichen Wertes hat auch Auswirkungen auf die mit ihm verlinkten Elemente.

Referenzen haben eine entgegengesetzte Richtung. Auch bei ihnen wird eine Verbindung zu einem anderen Element mit Hilfe der Punktnotation angegeben, aber sie werden dazu verwende, ihren Wert, an das Ziel des Pfades zu schreiben. Dies würde natürlich bei gleichzeitiger Verwendung unweigerlich zu Mehrdeutigkeiten und Fehlern führen. Eben deshalb ist die Anwendung von Referenzen auf InMessages beschränkt. An dieser Stelle wird dieser Push-Mechanismus benötigt, da er in der Umgebung, in der sich IML entwickelt hat, die einzige Möglichkeit ist, den Inhalt einer Message vorübergehen zu speichern.

Die vollständige Bedeutung der Referenzen für InMessage wird bei der Beschreibung derselben ausgeführt. (siehe Kapitel 4.4.6.2.6) Hier sei nur noch angemerkt, dass die Punktnotation in einer Referenz immer auf ein Data-Value verweist und an keiner anderen Stelle im Widget-Baum geändert werden kann.

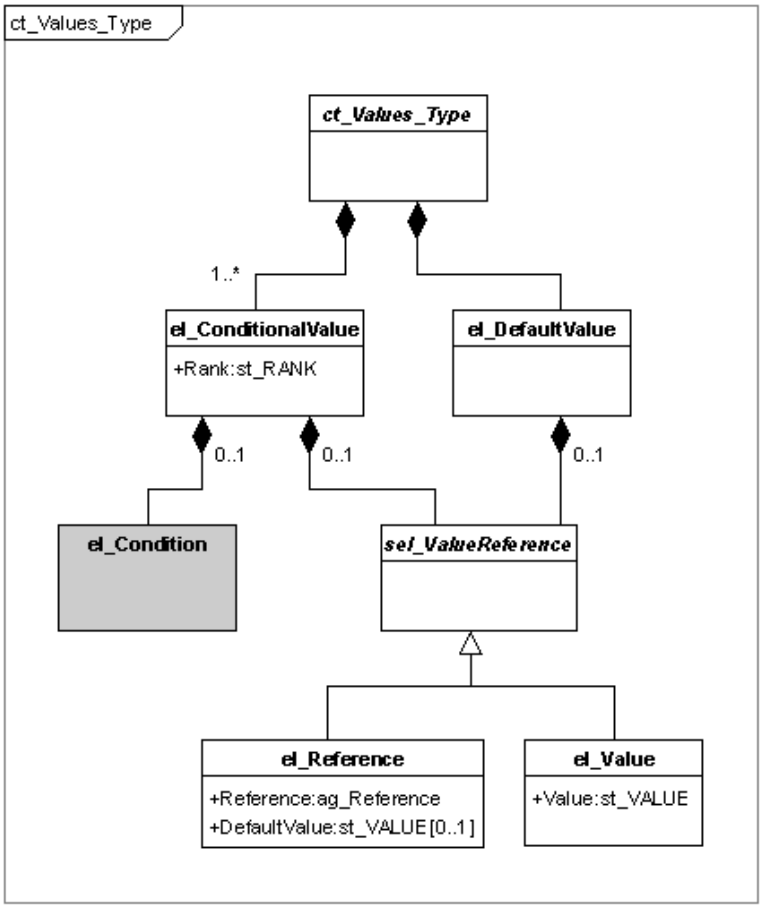

Abbildung 14 - Die verschiedenen Arten von Values

4.6.2 Das Komplexe Widget

Ein komplexes Widget ist aufgeteilt in einen Definitions- und einen Implementationsteil. Im Definitionsteil werden die Informationen festlegt, die bei der späteren Verwendung des Widgets als Kind-Widget nicht verändert werden können. Hier werden die Namen und Datentypen der eigenen Propertys festgelegt, definiert, welche anderen Widget als Kinder verwendet werden (Name, Speicherort) und Zustände benannt, die das Widget annehmen kann.

Constraints:

1. Im Definitionsteil muss dem Widget ein eindeutiger Name gegeben werden.

context el_Modell **inv**:
self.ct_Widget_Definition->forAll(w1:ct_Widget_Definition|
self.ct_Widget_Definition->forAll(w2:ct_Widget_Definition|
w1<>w2 implies w1.WidgetName.Name <> w2.WidgetName.Name))

2. Jeder State im Definitionsteil muss einen eindeutigen Namen haben

context el_States_Definition **inv**:
self.el_State_Definition->forAll(s1:el_State_Definition|
self.el_State_Definition->forAll(s2:el_State_Definition|
s1<>s2 implies s1.StateName.Name <> s2.StateName.Name))

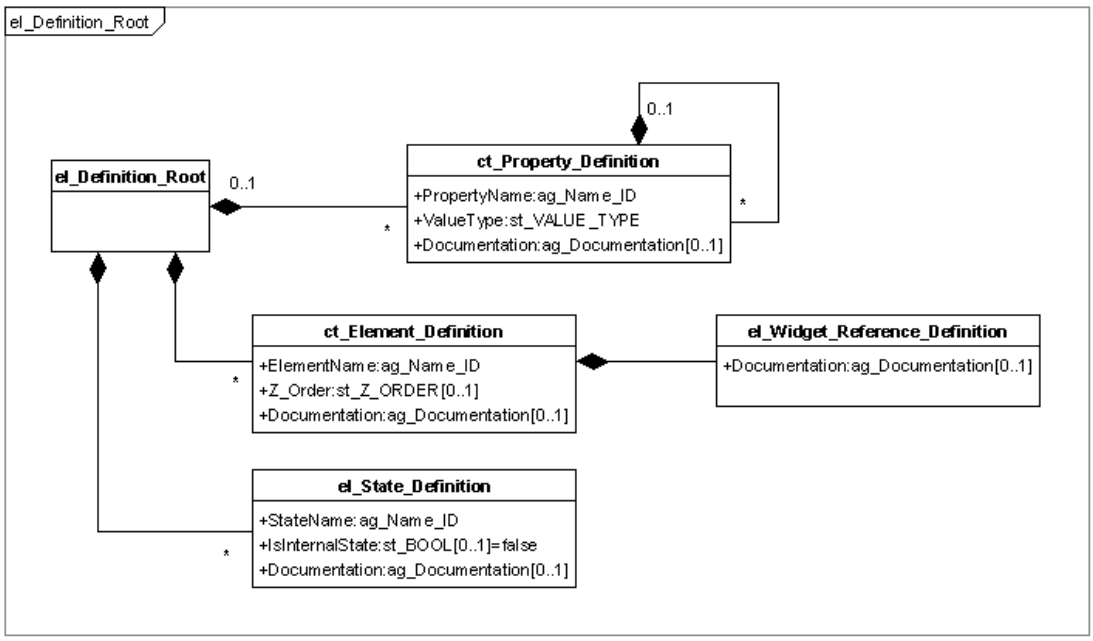

Abbildung 15 - Der Definitionsteil

Als Beispiel dient hier erneut der Definitionsteil des Widgets "PCM_OFF", das schon als vollständiges Widget gezeigt wurde. "PCM_OFF" ist ein Menü, das zwei Kind-Widgets enthält. Wenn man Widgets als Objekte in einer Programmiersprache betrachtet, dann entsprechen die Namen der Widgets den Objekttypen. Die Kind-Widgets in diesem Beispiel haben die Typen "Image" and "Textfield". Da ein Widget mehrere Kind-Widgets eines Typs enthalten kann, erhält die Beziehung zum Kind-Widget einen Namen, der innerhalb des Eltern-Widgets eindeutig sein muss. In der Programmiersprache würde dieser Name dem Instanznamen entsprechen.

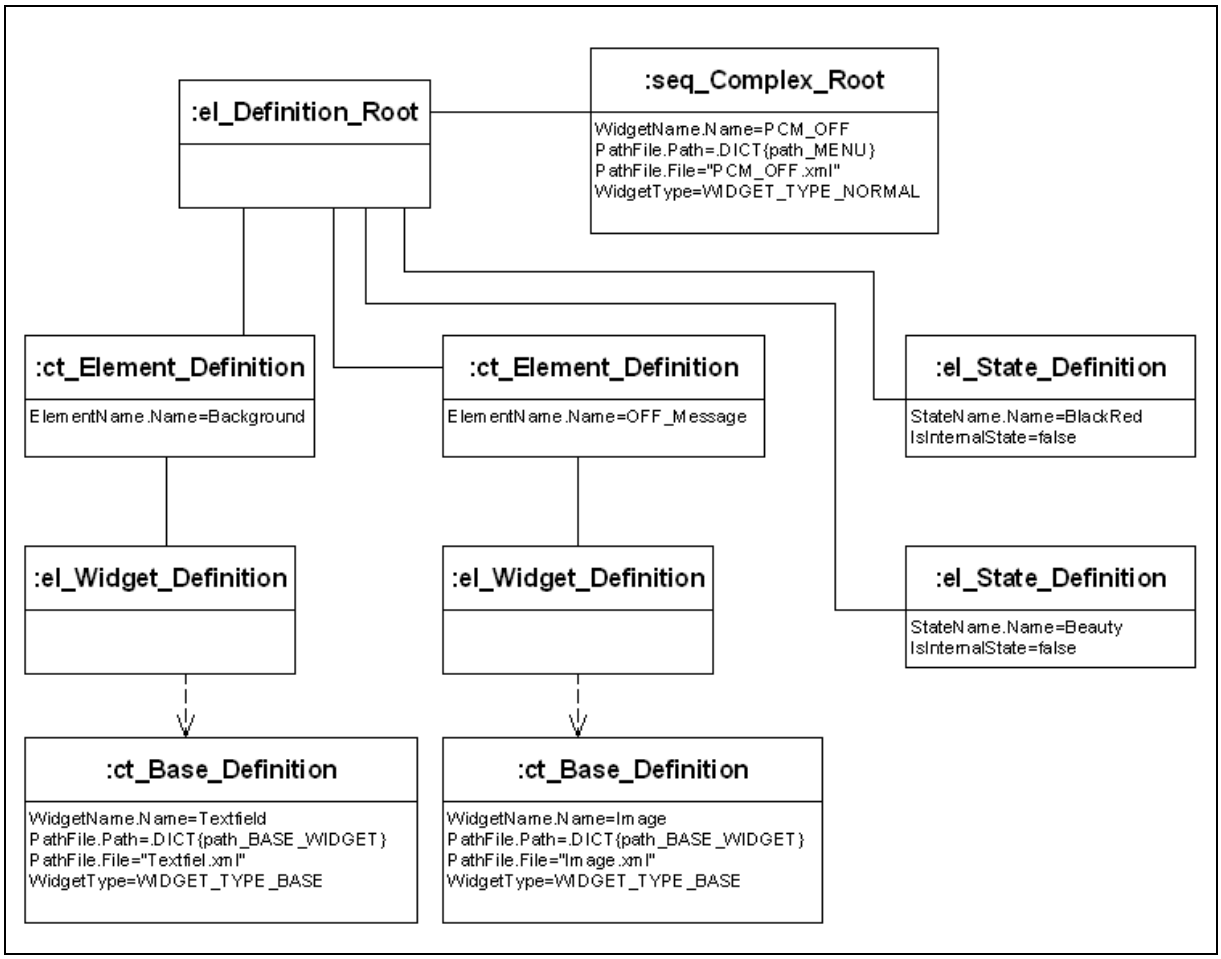

Abbildung 16 - Modell des Definitionsteils

Am Beispiel des Definitionsteils wird hier die Darstellung des Modells in XML gezeigt. Auf der linken Seite der unteren Abbildung (Abbildung 17) sind die Instanzen der Modellklassen zu sehen. Auf der rechten Seite erscheinen die XML-Elemente mit ihren Attributen. Die Pfeile deuten an, welche Instanz zur Erzeugung welches XML-Elements führt. Es existieren einige XML-Elemente, die keine Verbindung zu einer Klasseninstanz des Modells haben. Dies sind XML-Elemente, die lediglich die Lesbarkeit des XML erleichtern sollen. Sie werden unter bestimmten Bedingungen eingefügt. Die jeweilige Bedingung ist an das Element angefügt. Die Unterelemente von "Element" wurden aus Gründen der Übersichtlichkeit weggelassen.

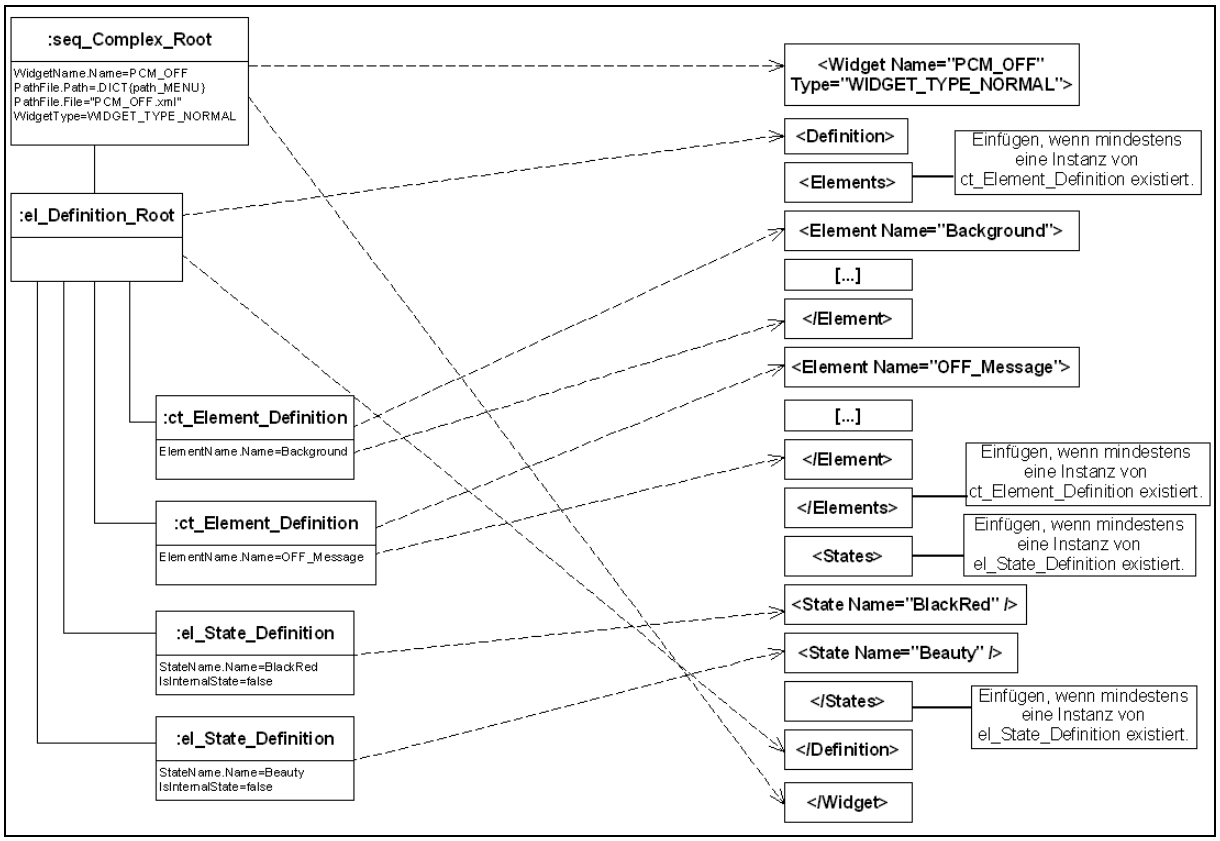

Abbildung 17 - Umsetzung des Definitionsteils zu XML

```
<Widget Name="PCM_OFF" Type="WIDGET_TYPE_MENU" >
    <Definition>

        <Elements>

            <Element Name="Background">
                <Widget Name="Image" Path=".DICT{path_BASE_WIDGET}" File="Image.xml"/>
            </Element>

            <Element Name="OFF_Message">
                <Widget Name="TextField" Path=".DICT{path_BASE_WIDGET}" File="Textfield.xml"/>
            </Element>

        </Elements>

        <States>
            <State Name="BlackRed"/>
            <State Name="Beauty"/>
        </States>

    </Definition>
    [...]
</Widget>
```

Im Implementationsteil werden den im Definitionsteil erstellten Propertys Werte zugewiesen (*el_Propertes_Implementation*). Weiterhin können die Werte der Propertys von Kind-Widgets geändert werden. Diese Änderungen können entweder allgemein oder in Abhängigkeit beliebiger Kombinationen der eigenen und der Kindzustände gelten. So kann zum Beispiel in einem Widget, das zwei Elemente des Typs "TextField" enthält, die Farbe des Textes eines der Elemente auf den Wert "rot" gesetzt werden, während die Textfarbe des anderen Elements ihren ursprünglichen Wert behält. Es können in gleicher Form auch die Propertys von den Kindern der Kinder in beliebiger Tiefe verändert werden, so lange dies in dem jeweiligen Widget gestattet ist (siehe FinalProperty Kapitel 4.1.6.2.1).

Eine weitere wichtige Aufgabe des Implementationsteils ist die Spezifikation des eigenen Verhaltens unterhalb des Behavior-Knotens (*el_Behavior*). Dort werden sowohl die ein- und ausgehenden Nachrichten, die verwendbaren Funktionen als auch die Zustandsübergänge beschrieben (*el_InOut*). Außerdem können dort Variablen (*ct_DataSet*) angelegt werden, die zur kurzfristigen Speicherung von Daten während der Laufzeit verwendet werden.

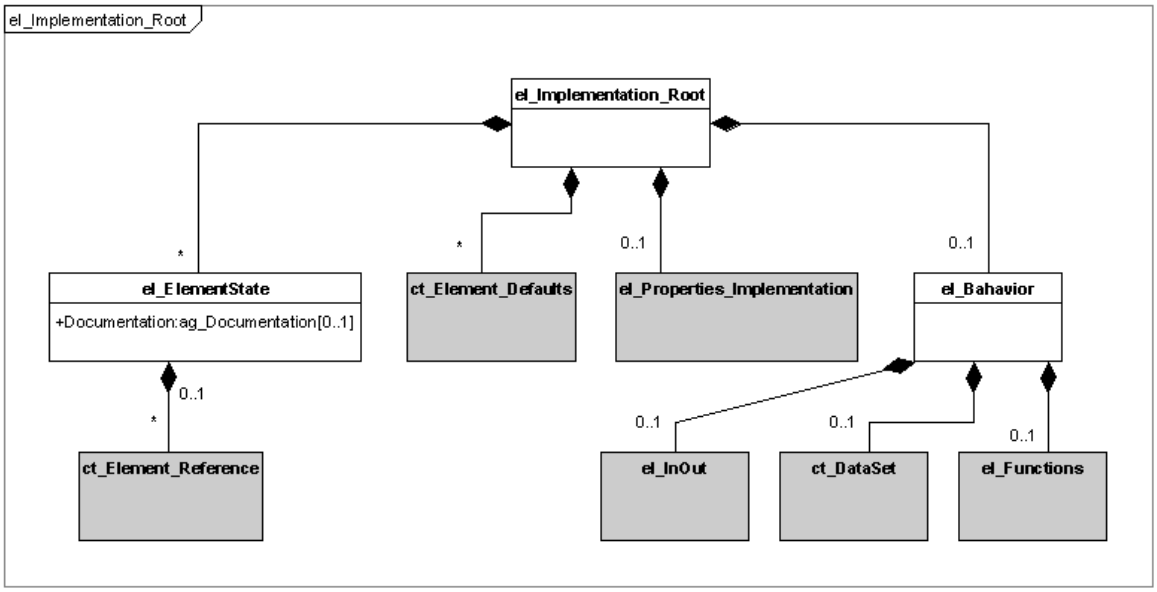

Abbildung 18 - Aufbau des Implementationsteils

Da der Implementationsteil der Widgets schon bei einfachen Widgets sehr komplex ist, wird hier nur die generelle Struktur dargestellt. Der genauere Aufbau wird bei der Beschreibung der einzelnen Komponenten mit Beispielen belegt.

```
<Implementation>
    <Properties>
        […]
    </Properties>
    <Defaults>
        <Elements>
            <Element Name="Background">
                […]
            </Element>
            <Element Name="OFF_Message">
                […]
            </Element>
        </Elements>
    </Defaults>
    <States>
        <ElementStates>
            <State Name="Beauty">
                <Element Name="Background">
                    […]
                </Element>
                <Element Name="OFF_Message">
                    […]
                </Element>
            </State>
        </ElementStates>
    </States>
```

```
    <Behavior>
        <InOut>
            <InMessages>
                [...]
            </InMessages>
            <OutMessages>
                [...]
            </OutMessages>
            <StateTable>
                [...]
            </StateTable>
        </InOut>
    </Behavior>
</Implementation>
```

4.6.2.1 Properties

Es gibt zwei Arten von Propertys für Widgets: die einen sind die generellen Propertys, die allen Widgets gemein sind, die aber nicht in jedem Widget einen Wert haben müssen. Die andere sind die spezifischen Propertys, die individuell für ein bestimmtes Widget angelegt werden.

Um die Wiederverwendbarkeit von Widgets möglichst groß zu halten, werden Widgets entwickelt, die möglicht generisch sind. Diese generischen Widgets werden dann bei der Anwendung der Situation angepasst. Die Möglichkeit der Anpassung erhalten Widgets durch ihre Propertys (Eigenschaften). Über die Propertys kann sowohl das ausprogrammierte Verhalten von Widgets beeinflusst als auch Teile ihres Aussehens verändert werden.

Es ist zum Beispiel für die Grundfunktionalität des bei der Lautstärkeeinstellung verwendeten Sliders kein Unterschied, ob diese waagerecht oder senkrecht angeordnet werden. Für die Berechnung der Position des Anzeigeelements ist diese Information jedoch erforderlich. Da das Widget "Slider" ein Property "Direction" erhalten hat, kann diese Information bei der Berechnung abgefragt werden und man muss bei der Änderung der Richtung nur den Wert dieses Propertys anpassen.

Da die Widgets, die höher im Widget-Baum stehen, immer spezieller werden, nimmt auch die Anzahl ihrer Eigenschaften ab. Es ist nicht ungewöhnlich, wenn Menüs nur generelle Propertys besitzen.

4.1.6.2.1 Generelle Propertys

Generelle Propertys sind Teil jedes Widgets, müssen jedoch nicht in jedem Widget mit einem Wert versehen sein.

Das *FinalProperty* hat vier Attribute: *Properties*, *Elements*, *States* und *StateProperty*. Diese Attribute können die Werte "TRUE" und "FALSE" annehmen und bestimmen ob die entsprechenden Teile des Widgets von Widgets oberhalb seines direkten Eltern-Widgets überschrieben werden dürfen. Der Wert "FALSE" beim Attribut *Properties* verbietet es also jedem Widget oberhalb der Eltern die Propertys dieses Widgets zu verändern. Im Allgemeinen haben alle Attribute dieses Propertys den Wert "TRUE" und behalten ihn auch.

Das StateProperty wird verwendet, um den State festzulegen, in dem sich das Widget befinden soll, wenn es aufgerufen wird. Dabei können auch mehrere States mit unterschiedlichen Bedingungen angegeben werden. Über das StateProperty kann außerdem über das Attribut "CurrentState" immer der aktuelle Zustand eines Widgets abgefragt werden.

Ob die Angaben vom StateProperty nur beim ersten Aufruf des Widgets oder bei jedem Aufruf ausgewertet werden, wird über das HistoryProperty gesteuert. Wenn das Attribut *History-Type* des HistoryPropertys den Wert "NONE" annimmt, bedeutet dies, dass das StateProperty bei jedem Aufruf ausgewertet wird, außer eine direktes oder indirektes Eltern-Widget hat ein HistoryProperty mit dem Wert "DEEP ". Der Wert "SHALLOW" bedeutet, dass ein Widget bei erneutem Aufruf in den zuletzt aktiven Zustand zurückkehrt. "DEEP" hat für das Widget, in dem es gesetzt wird die gleiche Auswirkung wie "SHALLOW", aber hier wirkt sich diese Angabe auch auf alle Kinder aus.

Das ImplementationProperty bietet die Möglichkeit ein Widget mit einem ausprogrammierten Verhalten zu verbinden, um zum Beispiel Simulationen zu ermöglichen. Dazu enthält es eine API-Beschreibung (*el_API*), die die Attribute "Type" und "Target" besitzt. Das Behavior eines Widgets ist durchaus hinreichend, um zu beschreiben, wie es auf Ereignisse reagiert oder welche Informationen von Komponenten zur Anzeige abgerufen werden sollen, aber es ist nicht geeignet das innere Verhalten eines komplexen Widgets zu spezifizieren.

Die Widgets, die ein solchermaßen komplexes Verhalten vorweisen sind von WidgetType "FUNCTION", "TARGET_LIBRARY" oder "BASE". Durch die Verwendung von Systemfunktionen und der Punktnotation wäre es theoretisch auch möglich, das ganze Verhalten im Widget selbst zu spezifizieren. Dies ist jedoch gar nicht erwünscht. Da die Anforderungen eines Infotainmentsystems in Hinsicht auf Speicherbedarf und Geschwindigkeit hoch sind, ist es nötig, den Programmcode stark auf die Hardwaregegebenheiten zu optimieren. Da das Widgetmodell von der späteren Target-Hardware unabhängig spezifiziert werden muss, können solche Optimierungen dort nicht beachtet werden.

Das VersionProperty ermöglicht es, eine Versionskontrolle auf Ebene der einzelnen Widgets durchzuführen. Dazu kann man die Attribute "Major", "Minor", "Release" und "Build" verwenden.

Da die meisten Widgets grafische Elemente sind, ist ihre Positionierung auf dem Bildschirm sehr wichtig. Bedientechnische oder funktionale Fehler des Systems sind für den Benutzer nur selten klar als Fehler zu erkennen. Bei grafischen Fehlern durch die Verwendung falscher Farben, Bilder oder einer fehlerhaften Positionierung, ist dies anders. Solche Fehler sind leicht als Fehler zu erkennen und senken damit die allgemeine Zufriedenheit der Kunden rasch. Aus diesem Grund wird auf die korrekte, meistens pixelgenaue Positionierung aller grafischen Elemente viel Wert gelegt.

Das AnchorProperty erleichtert den Designern die genaue Positionierung von Widgets. Mit seiner Hilfe kann festgelegt werden, welcher Punkt des Kind-Widgets als Anker der Positionierung verwendet wird. In der Regel werden dazu neun Punkte auf einem das Widget umhüllenden Rechteck verwendet. Diese Punkte werden mit jeweils zwei Koordinaten, ihrer jeweiligen horizontalen und vertikalen Position, definiert. Wenn dies noch nicht ausreichend ist, können die Koordinaten des Ankerpunktes auch durch feste Pixelwerte oder durch Offsets zu den neun Punkten bestimmt werden. Die angegeben absoluten Pixelpositionen des Widgets werden dann relativ zur Null-Position (TOP; LEFT) des Eltern-Widget ausgewertet. Die relativen Positionen richten sich nach den Positionen der Geschwister-Widgets.

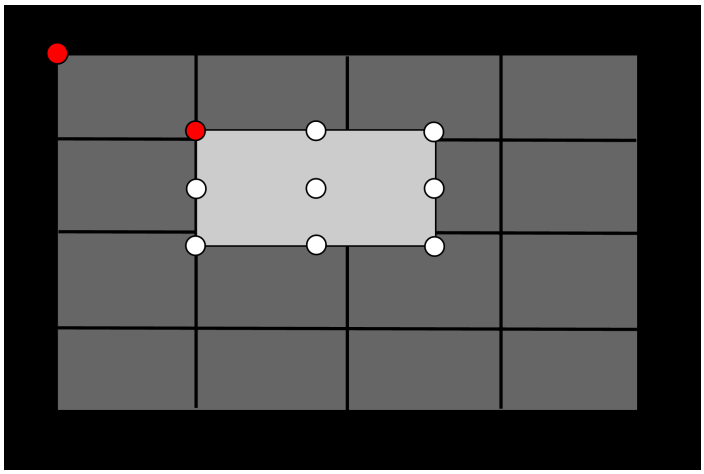

Abbildung 19 - Die Ankerpunkte des Widgets

Das DescriptionProperty dient zum automatisierten Erzeugen einer Dokumentation. In ihm können die Propertys genauer beschrieben werden als es in ihren Documentation-Attributen möglich ist.

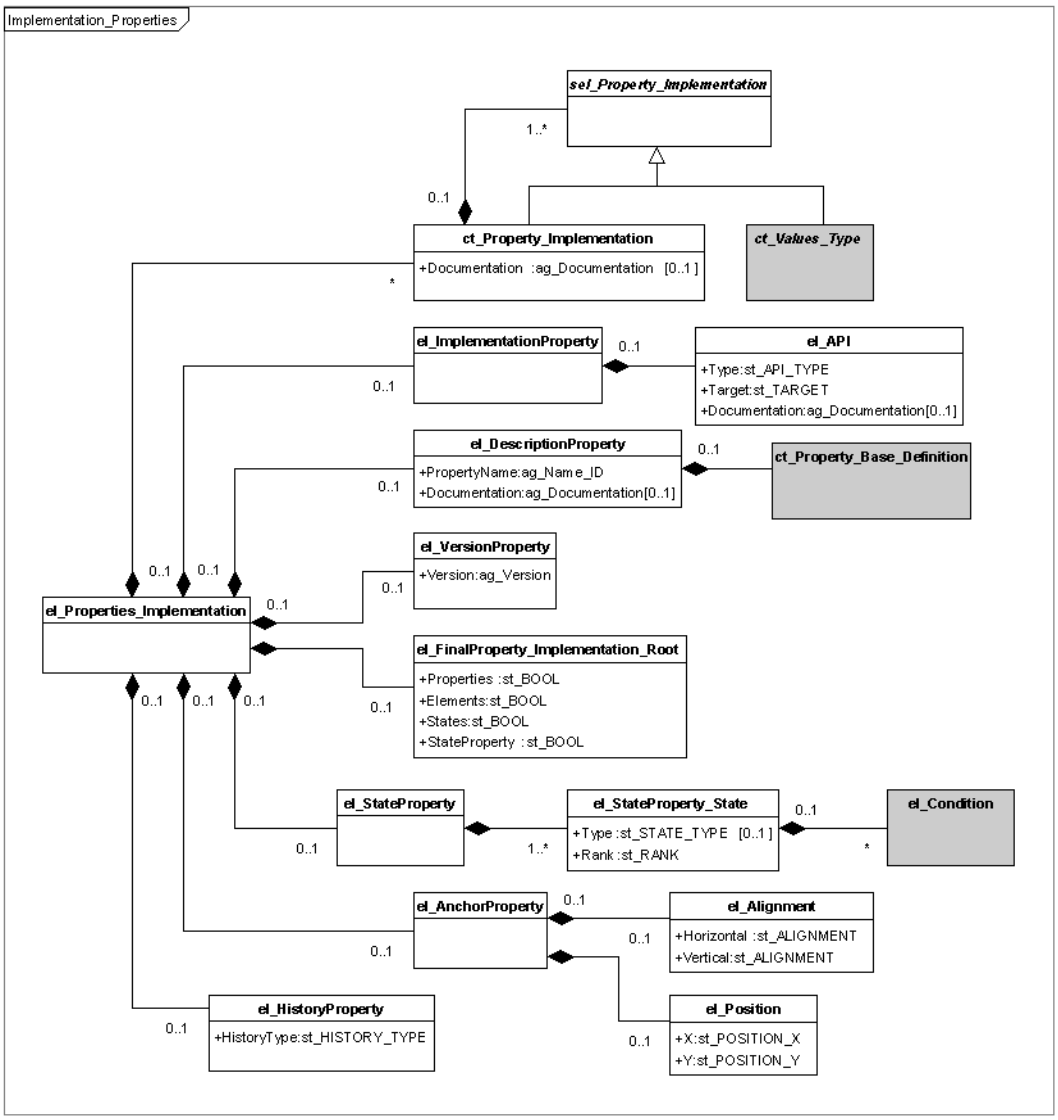

Abbildung 20 - Propertys im Implementationsteil

Im verwendeten Beispiel werden nur einige der vorhandenen generellen Propertys gesetzt, da die anderen nicht benötigt wurden. Daher wird nur für diese Propertys ein XML-Beispiel aufgeführt.

Das StateProperty des Menüs "PCM_Off" entscheidet über den aktiven Zustand für das Widget durch die Abfrage einer Systemvariablen. Wenn diese Variable den Wert "true" enthält, wird der Zustand "Beauty", ansonsten der Zustand "BlackRed" aktive. An Stelle der Abfrage einer Systemvariablen wäre auch der Aufruf einer Funktion möglich.

```
<StateProperty>
    <State Name="Beauty">
        <Conditions>
            <Condition Name="Is Plus Version">
                <Expression>
                    <Term Term=".REP{.DICT{repkey_HIGHEND_VERSION}}"/>
                    <BinaryOperator Operator="EQUAL"/>
                    <Term Term=".DICT{bool_TRUE}"/>
                </Expression>
            </Condition>
        </Conditions>
    </State>
    <State Name="BlackRed"/>
</StateProperty>
```

Das Setzen von AnchorProperty und FinalProperty ist bei der Größe dieses Bespielsystem nicht unbedingt nötig und wurde nur zu Anschauungszwecken hinzugefügt. Die beiden Beispiele für AnchorProperty legen den Ankerpunkt des Widgets auf die linke, obere Ecke. Dass dies oder eine andere Position die Null-Position aller Widget ist, wird projektabhängig festgelegt.

```
<AnchorProperty>
    <Position X="0" Y="0"/>
</AnchorProperty>
```

```
<AnchorProperty>
    <Alignment Horizontal="alignment_TOP" Vertical="alignment_LEFT"/>
</AnchorProperty>
```

Wenn das FinalProperty nicht angegeben wird, wird der Default-Wert "false" für alle seine Attribute angenommen. Das bedeutet, dass jedes nicht angegebene FinalProperty äquivalent durch die Angaben des Beispiels ersetzt werden könnte.

```
<FinalProperty Elements="false" States="false" Properties="false" StateProperty="false"/>
```

4.1.6.2.2 Spezifische Propertys

Spezifische Propertys sind speziell für die verschiedenen Widget definierbar. Sie dienen dazu die Wiederverwendbarkeit der Widgets zu erleichtern, indem sie Informationen über die konkrete Instanz eines Widgets aufnehmen können. Durch sie kann zum Beispiel die Aufschrift von Buttons oder die Schriftfarbe einer Textbox den jeweiligen Anforderungen angepasst werden. Auch Einstellungen wie die Laufzeit eines Timers oder die Laufrichtung einer Fortschrittsanzeige können so gesteuert werden.

Je allgemeiner ein Widget sein soll, desto mehr Propertys benötigt es. Wie veränderbar ein Widget zu sein hat bzw. wie genau es schon auf eine spezielle Verwendung im System zugeschnitten werden soll, ist eine Designfrage, die genaues Abwägen nötig macht. Ein Widget, das viele verschiedene Anwendungsmöglichkeiten abdeckt, sorgt für eine insgesamt kleine Anzahl benötigter Widgets. Da es jedoch so flexibel ist, hat es für jede Situation Eigenschaften und zum Teil sogar Elemente, die nicht benötigt werden. Einerseits kann es also bei der Spezifikation mit generelleren Widgets leichter zu Fehlern kommen, da sie komplexer und damit schwerer zu überschauen sind. Zudem werden Ressourcen verwendet, die nicht benötigt werden und die bei einer Übertragung auf ein Target-System optimiert werden sollten. Andererseits können grundsätzliche Änderungen leichter durchgeführt werden, wenn nur wenige Widgets, in diesem Fall die generellen, davon betroffen sind.

Constraints:

1. Ein ct_Property_Implementation enthält entweder genau ein ct_Values_Type oder eine oder mehrere ct_Property_Implementation

```
context ct_Property_Implementation inv:
self.select(sel_Property_Implementation)→size() > 1 implies
    self.sel_Property_Implementation→forAll(s.oclIsTypeOf(ct_Property_Implementation))
```

Propertys können ineinander verschachtelt sein. Dabei kann jedes Property jedoch nur entweder eine "Klammer" für andere Propertys sein oder einen Wert enthalten, niemals beides.

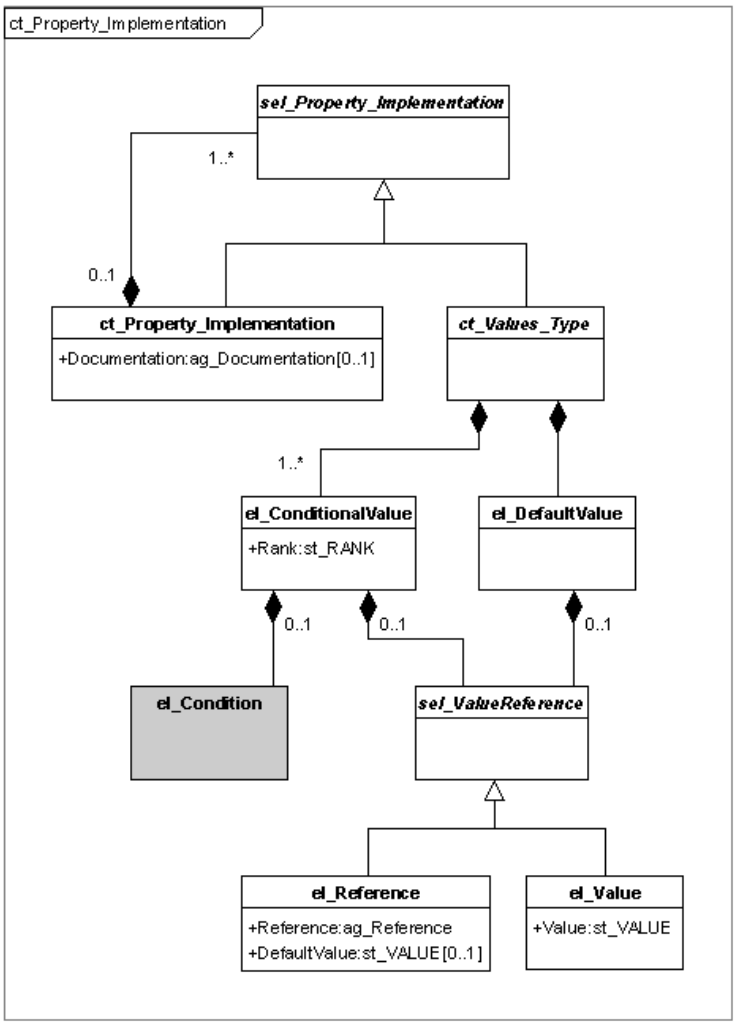

Abbildung 21 - Aufbau eines spezifischen Propertys

Im Definitionsteil werden die Namen und der Datentypen der spezifischen Propertys festgelegt. Diese können bei späteren Überschreibungen nicht geändert werden. Das verwendete Beispiel zeigt die speziellen Propertys des Widgets "Single_List_Item". Die Propertys sind ineinander verschachtelt, um Zusammenhänge zwischen ihnen darstellen zu können. Das Property "ItemTag" enthält zum Beispiel die Propertys "DynamicIndex" und "Static". Durch den Inhalt dieser beiden Propertys kann jedes Element einer Liste genau identifiziert werden. Propertys, die zum Zusammenfassen von anderen Propertys verwendet werden, haben den Datentyp "DATA_TYPE_STRUCT".

```
<Properties>
    <Property Name="ItemProperties" ValueType="DATA_TYPE_STRUCT">
        <Property Name="ItemTag" ValueType="DATA_TYPE_STRUCT">
            <Property Name="DynamicIndex" ValueType="DATA_TYPE_UINT"/>
            <Property Name="Static" ValueType="DATA_TYPE_STRING"/>
        </Property>
        <Property Name="Dynamic" ValueType="DATA_TYPE_BOOL"/>
        <Property Name="ScrollType" ValueType="DATA_TYPE_UINT"/>
        <Property Name="ItemCount" ValueType="DATA_TYPE_UINT"/>
        <Property Name="StaticContent" ValueType="DATA_TYPE_STRUCT">
            <Property Name="Label" ValueType="DATA_TYPE_DYN_ARRAY_STRING"/>
        </Property>
        <Property Name="DataSource" ValueType="DATA_TYPE_STRUCT">
            <Property Name="Label" ValueType="DATA_TYPE_STRING"/>
        <Property>
    </Property>
</Properties>
```

Im Implementationsteil werden die Default-Werte für die Propertys gesetzt. Im aufgeführten Beispiel wurden nicht alle Propertys übernommen. Die Aufgeführten zeigen jedoch eine große Anzahl der möglichen Arten der Wertezuweisung.

```
<Properties>
    <Property Name="ItemProperties">
        <Property Name="ItemTag">
            <Property Name="DynamicIndex">
                <Values>
                    <FixedValue>
                        <Value Value="0"/>
                    </FixedValue>
                </Values>
            </Property>
            <Property Name="Static">
                <Values>
                    <FixedValue>
                        <Value Value=".DICT{txt_EMPTY}"/>
                    </FixedValue>
                </Values>
            </Property>
        </Property>
        [...]
        <Property Name="StaticContent">
            <Property Name="Label">
                <Values>
                    <FixedValue>
                        <Value Value="{'testtt'}"/>
                    </FixedValue>
                </Values>
            </Property>
        </Property>
        <Property Name="DataSource">
            <Property Name="Label">
                <Values>
                    <FixedValue>
                        <Value Value=".LINK{.PROPERTY[ItemProperties.StaticContent.Label]
                                [.LINK{.PROPERTY[ItemProperties.ItemTag.DynamicIndex]}]}"/>
                    </FixedValue>
                </Values>
            </Property>
        </Property>
    </Property>
    [...]
</Properties>
```

4.6.2.2 Elemente

Der Widget-Baum wird durch Verwendung von Widgets in Widgets aufgebaut. Im Definitionsteil des Eltern-Widgets wird angegeben, welche Kind-Widgets verwendet werden sollen (siehe Abbildung 15 - Der DefinitionsteilAbbildung 15). Um ein Widget verwenden zu können, wird im Eltern-Widget ein Element eingefügt. Wenn man ein Widget als Klasse betrachtet, entspricht ein Element einer Instanz. Jedes Element hat einen, in seinem Eltern-Widget, eindeutigen Namen. Dies ist besonders wichtig, da ein Eltern-Widget mehrere Instanzen eines Widgets verwenden kann. Eine Bildschirmtastatur besteht zum Beispiel aus einer Anzahl von Buttons, die alle Instanzen des gleichen Widgets sind.

Im Implementationsteil können die Kind-Widgets den Ansprüchen des Eltern-Widgets angepasst werden. Dort müssen bei direkten Kindern auch die Position und die Existenz-Attribute gesetzt werden.

Constraints:

1. Jedes Element in einem Widget muss einen eindeutigen Namen haben.

context el_Elements_Definition **inv**:
self.ct_Element_Definition→forAll(e1:ct_Element_Definition|
self.ct_Element_Definition→forAll(e2:ct_Element_Definition|
e1<>e2 implies e1.ElementName.Name <> e2.ElementName.Name))

2. Jedes Element aus Definition, das nicht in Defaults existiert, muss in allen States eine Region und Existence haben.

context ct_Element_Definition **inv:**
self.Implementation→isEmpty() implies self.StateImplementations→forAll(s:el_ElementState|
s.ct_Element_Reference→select(e:ct_Element_Reference|
e.ElementName.Name = self.ElementName.Name).el_Region_Reference→notEmpty() and
s.ct_Element_Reference→select(e:ct_Element_Reference|
e.ElementName.Name = self.ElementName.Name).el_Existence_Reference→notEmpty())

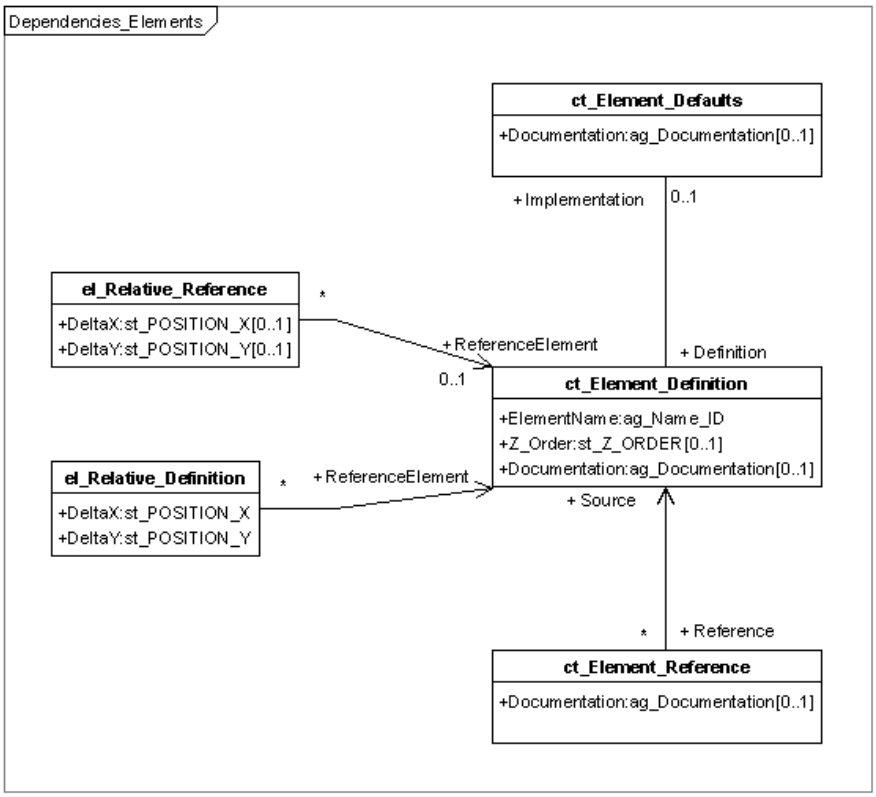

Abbildung 22 - Relation zwischen Elementen und ihren Referenzen.

Wie schon bei den Erklärungen zum Definitionsteil erwähnt, enthält das Menu "PCM_Off" zwei Elemente, das Hintergrundbild und ein Textfeld.

```
<Elements>
    <Element Name="Background">
        <Widget Name="Image" Path=".DICT{path_BASE_WIDGET}" File="Image.xml"/>
    </Element>
    <Element Name="OFF_Message">
        <Widget Name="TextField" Path=".DICT{path_BASE_WIDGET}" File="Textfield.xml"/>
    </Element>
</Elements>
```

Diese werden im Defaults-Element des Implementationsteils an die Anforderungen von PCM_Off angepasst. Das Hintergrundbild erhält die Größe eines Standardmenüs, wodurch es den gesamten Bildschirm ausfüllt. Außerdem wird im Property "File" der Name der konkreten

Bilddatei angegeben. Die Widgets vom Typ "Image" besitzen zusätzlich Propertys, welche die Art der verwendeten Bilddatei und ihren Speicherort definieren. In diesem Beispiel werden nur eine Dateiart und ein Speicherort verwendet. Deswegen werden diese Werte nicht überschrieben.

Für das Textfeld wird ebenfalls eine Position und Größe angegeben. Außerdem werden der Inhalt, die Textfarbe und die Schriftart festgelegt.

```
<Defaults>
    <Elements>

        // Das erste Kind-Widget
        <Element Name="Background">
            <Region>
                <Position>
                    <Absolute X=".DICT{pos_NULL_X}" Y=".DICT{pos_NULL_Y}"/>
                </Position>
                <Dimension Height=".DICT{height_MENU}" Width=".DICT{width_MENU}"/>
            </Region>
            <Existence Load="ONCE" Unload="ONCE" Visible="true" Active="true"/>
            <Widget Name="Image">
                <Base Name="Image">
                    <Properties>
                        <Property Name="ImageProperties">
                            <Property Name="Left">
                                <Property Name="File">
                                    <Values>
                                        <FixedValue>
                                            <Value Value=".DICT{file_OFF_BACKGROUND}"/>
                                        </FixedValue>
                                    </Values>
                                </Property>
                            </Property>
                        </Property>
                    </Properties>
                </Base>
            </Widget>
        </Element>

        // Das zweite Kind-Widget
        <Element Name="OFF_Message">
            <Region>
                <Position>
                    <Absolute X=".DICT{pos_NULL_X}" Y="80"/>
                </Position>
                <Dimension Height="34" Width="400"/>
            </Region>
            <Existence Load="ONCE" Unload="ONCE" Visible="true" Active="true"/>
            <Widget Name="TextField">
                <Base Name="TextField">
                    <Properties>
                        <Property Name="TextFieldProperties">

                            //Setzen des Inhalts des Textfelsdes
                            <Property Name="Content">
                                <Values>
                                    <FixedValue>
                                        <Value Value="O F F"/>
                                    </FixedValue>
                                </Values>
                            </Property>

                            //Setzen der Farbe der Schrift
                            <Property Name="Color">
                                <Property Name="Foreground">
                                    <Values>
                                        <FixedValue>
                                            <Value Value=".RES{color_RED}"/>
                                        </FixedValue>
                                    </Values>
                                </Property>
                            </Property>
```

```
                        // Setzen der Schriftgröße
                        <Property Name="Font">
                            <Property Name="Size">
                                <Values>
                                    <FixedValue>
                                        <Value Value=".RES{uint_FontSize_16}"/>
                                    </FixedValue>
                                </Values>
                            </Property>
                        </Property>
                    </Property>
                </Properties>
            </Base>
        </Widget>
    </Element>
</Elements>
</Defaults>
```

4.2.6.2.1 Verwendung von Kind-Widgets

Jedes Widget hat außer seinen generellen und spezifischen Propertys noch weitere Eigenschaften. Auch diese Eigenschaften sind allen Widget gemein, allerdings können sie anders als generelle Propertys nicht in den Widgets selbst gesetzt werden. Um sie im internen Sprachgebrauch von den Propertys abzusetzen, werden sie als Attribute bezeichnet. Das Attribut "Position" muss im direkten Eltern-Widget ebenso gesetzt werden wie die Attribute "Active" und "Visible". Das Attribut "Dimension" dagegen kann, muss aber nicht zwingend, gesetzt werden. Wenn es nicht gesetzt wird, erhält das Kind die Dimension seines Eltern-Widgets. Alle Attribute können wie Propertys auch von Widgets oberhalb der Eltern sowohl zustandsabhängig als auch zustandsunabhängig überschrieben werden.

Das Attribut "Position" definiert die Position der grafischen Repräsentation des Widgets auf dem Bildschirm. Dabei kann die Position absolut aber mit Bezug auf seine Eltern-Widget oder relativ zu seinen Geschwistern gesetzt werden. Beide Positionsangaben beziehen sich für das Widget selbst immer auf seinen definierten Anker (AnchorProperty). Bei der Angabe von relativen Positionen muss darauf geachtet werden, dass mindestens ein Kind-Widget eine absolute Position erhält. Die absoluten Positionen beziehen sich auf die linke obere Ecke des Eltern-Widget.

Das Attribut "Dimension" bestimmt die Größe eines Widgets. Sie wird durch die Höhe und Breite angegeben. Die Dimensionen eines Kind-Widgets sollten die Dimensionen seines Eltern-Widgets nicht überschreiten und es sollte vollständig innerhalb seines Eltern-Widgets liegen. Dies ist nicht immer möglich, Ausnahmen sollten aber möglichst selten gemacht werden.

Die Attribute des Existence-Elements dienen dazu die Lebensdauer, die Sichtbarkeit und die Verwendbarkeit des Kindes festzulegen. Die Attribute "Load" und "Unload" bestimmen die Lebenszeit, indem sie angeben, wann das Widget geladen bzw. entladen wird. Wenn bei diesen Attributen keine Angaben gemacht werden, wird angenommen, dass das Kind-Widget zusammen mit dem Eltern-Widget geladen und entladen wird. Das spätere Nachladen oder frühzeitige Entladen von Widgets kann zur Speicheroptimierung genutzt werden. In einer Liste, die hundert Einträge enthält, können jeweils nur fünf oder sechs Einträge dargestellt werden. Da jeder Eintrag in einer Liste einer Instanz eines Widgets entspricht, würde der Speicherbedarf unnötig hoch sein, wenn alle Einträge geladen im Speicher gehalten werden würden. Resourcenschonender ist es, wenn nur die sichtbaren Einträge und eventuell einige vorherige und nachfolgende im Speicher gehalten werden. Wann oder unter welchen Umstän-

den die Widgets nachgeladen werden, wird nicht in IML spezifiziert, sondern ist Teil der Umsetzung auf der Target-Hardware.

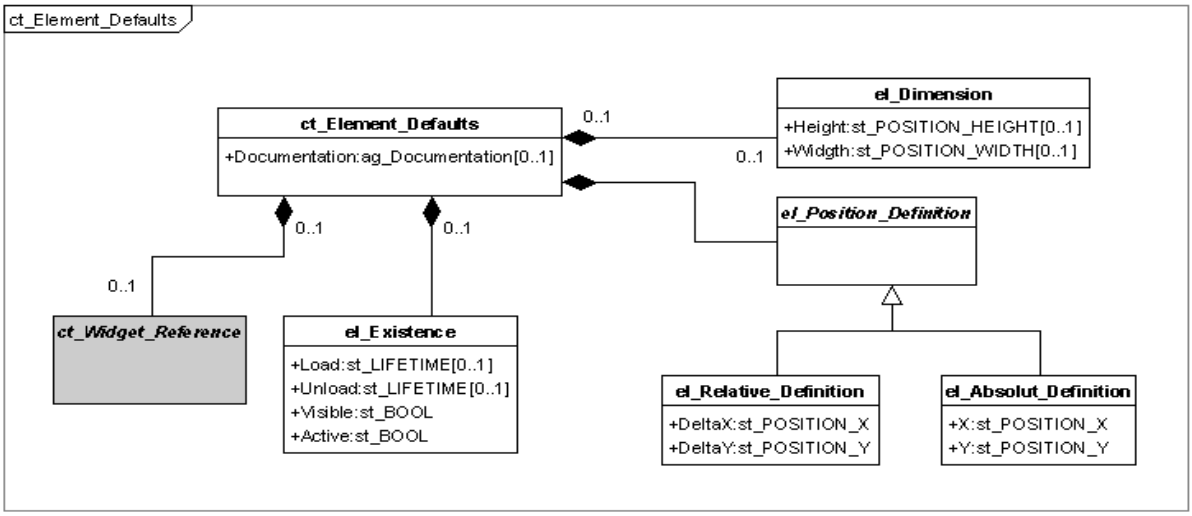

Abbildung 23 - Angaben zu direkten Kind-Widgets

Widgets, die als Elemente in anderen Widgets verwendet werden, müssen vorher selber definiert worden sein. Dass heißt sowohl jede *el_Widget_Reference_Definition* als auch jede *ct_Widget_Reference* bezieht sich auf eine existierende *ct_Widget_Definition*.

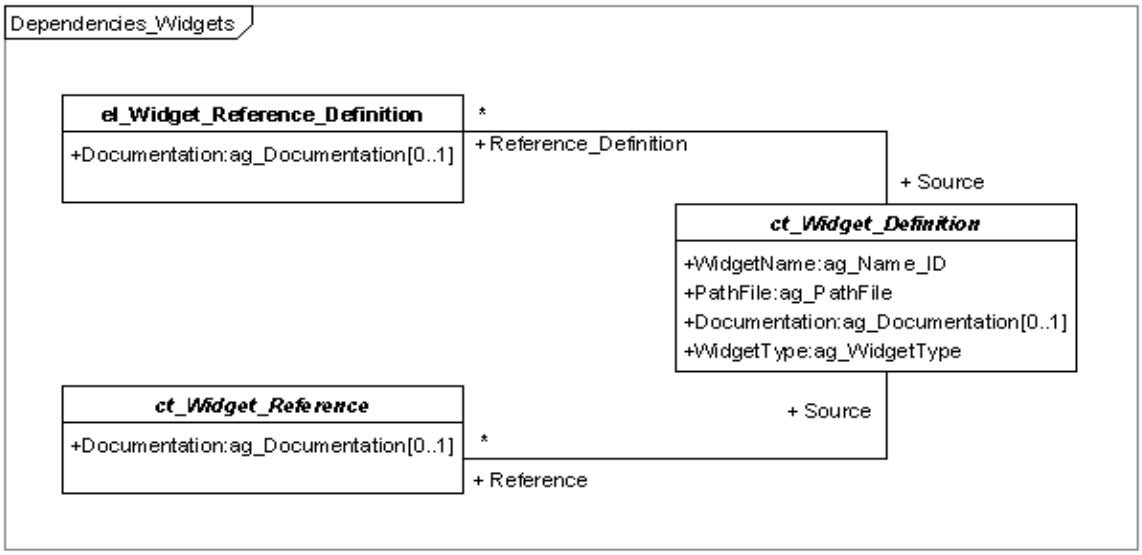

Abbildung 24 - Relationen zwischen der Definition eines Widgets und seinen Referenzen im Definition- bzw. Implementationsteil eines Eltern-Widget.

Die Attribute des Kind-Widgets müssen in seinem direkten Eltern-Widget gesetzt werden. In diesem Beispiel wird eine absolute Position verwendet, die ebenso wie die Dimension mit festen Werten gefüllt wird. Das Element ist sichtbar (Visible), bedienbar (Active) und wird zusammen (ONCE) mit dem Menü geladen bzw. entladen.

```
<Element Name="Background">
    <Region>
        <Position>
            <Absolute X=".DICT{pos_NULL_X}" Y=".DICT{pos_NULL_Y}"/>
```

```
        </Position>
        <Dimension Height=".DICT{height_MENU}" Width=".DICT{width_MENU}"/>
    </Region>
    <Existence Load="ONCE" Unload="ONCE" Visible="true" Active="true"/>
    <Widget Name="Image">
        [...]
    </Widget>
</Element>
```

4.2.6.2.2 Anpassung von Kind-Widgets

Neben den Informationen, die man nur der Instanz eines Widgets zuordnen kann, wie zum Beispiel seine Existence, kann man auch die Werte der Propertys des Kind-Widgets ändern. Wenn keine Werte für die Propertys der Instanz angegeben werden, verwendet man die Werte, die im Widget selbst definiert sind oder die Werte, die durch eine höherwertige Überschreibung entstehen (siehe auch Überschreibungsregeln).

Bei der Überschreibung von Widgets ist es nicht möglich, auf den Definitionsteil des Kindes zuzugreifen. Im Implementationsteil können die Werte der spezifischen Propertys des Kind-Widgets und sein AnchorProperty und StateProperty geändert werden. Außerdem können dort abhängig oder unabhängig von den Zuständen des Kindes die Propertys der Enkel-Widgets überschrieben werden. Auf diese Weise kann man alle Widgets eines Astes des Widget-Baumes bis zum untersten Base-Widget verändern. Wenn eine Überschreibung der eigenen spezifischen Propertys, des eigenen StatePropertys oder der Kind-Widgets in States oder im Allgemeinen nicht erlaubt werden soll, kann dies über das FinalProperty spezifiziert werden.

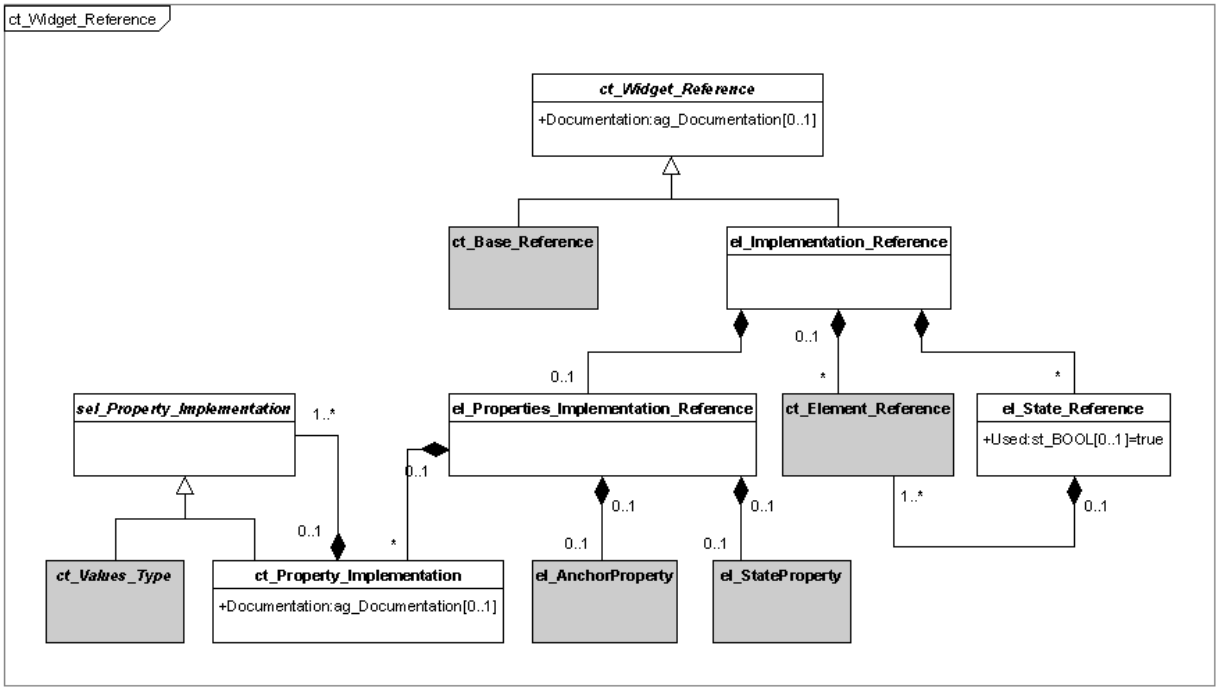

Abbildung 25 - Überschreibbare Elemente eines Widgets

Die Attribute der Widgets können entweder einmal im Defaults-Teil oder in jedem State des Eltern-Widgets gesetzt werden.

```
<Defaults>
    <Elements>
        <Element Name="Background">
            <Region>
```

```
                <Position>
                    <Absolute X=".DICT{pos_NULL_X}" Y=".DICT{pos_NULL_Y}"/>
                </Position>
                <Dimension Height=".DICT{height_MENU}" Width=".DICT{width_MENU}"/>
            </Region>
            <Existence Load="ONCE" Unload="ONCE" Visible="true" Active="true"/>
            <Widget Name="Image">
                <Base Name="Image">
                    <Properties>
                        <Property Name="ImageProperties">
                            [...]
                        </Property>
                    </Properties>
                </Base>
            </Widget>
        </Element>
        <Element Name="OFF_Message">
            <Region>
                <Position>
                    <Absolute X=".DICT{pos_NULL_X}" Y="80"/>
                </Position>
                <Dimension Height="34" Width="400"/>
            </Region>
            <Existence Load="ONCE" Unload="ONCE" Visible="true" Active="true"/>
            <Widget Name="TextField">
                <Base Name="TextField">
                    <Properties>
                        <Property Name="TextFieldProperties">
                            [...]
                        </Property>
                    </Properties>
                </Base>
            </Widget>
        </Element>
    </Elements>
</Defaults>
```

Es ist ebenfalls möglich, einzelnen Attributen oder Propertys in den States des Eltern-Widgets andere Werte zuzuweisen, als sie ihm im Defaults-Teil bekommen haben. Wenn nur die Attribute überschrieben werden sollen, ist es nicht erforderlich das "Widget"-Element mit aufzuführen (siehe Markierung 1).

```
<States>
    <ElementStates>
        <State Name="Beauty">
            <Element Name="Background">
                <Widget Name="Image">
                    <Base Name="Image">
                        <Properties>
                            <Property Name="ImageProperties">
                                [...]
                            </Property>
                        </Properties>
                    </Base>
                </Widget>
            </Element>
            <Element Name="OFF_Message">
                <Existence Visible="false" Active="false"/>          <---------------------------------- !! 1 !!
            </Element>
        </State>
    </ElementStates>
</States>
```

Wenn es sich bei dem Kind-Widget um ein Base-Widget handelt, können dort natürlich nur die dort vorhandenen spezifischen Propertys und das AnchorProperty überschreiben werden. Ein StateProperty sowie Zustände oder weitere Kind-Widgets existieren in diesem Fall nicht.

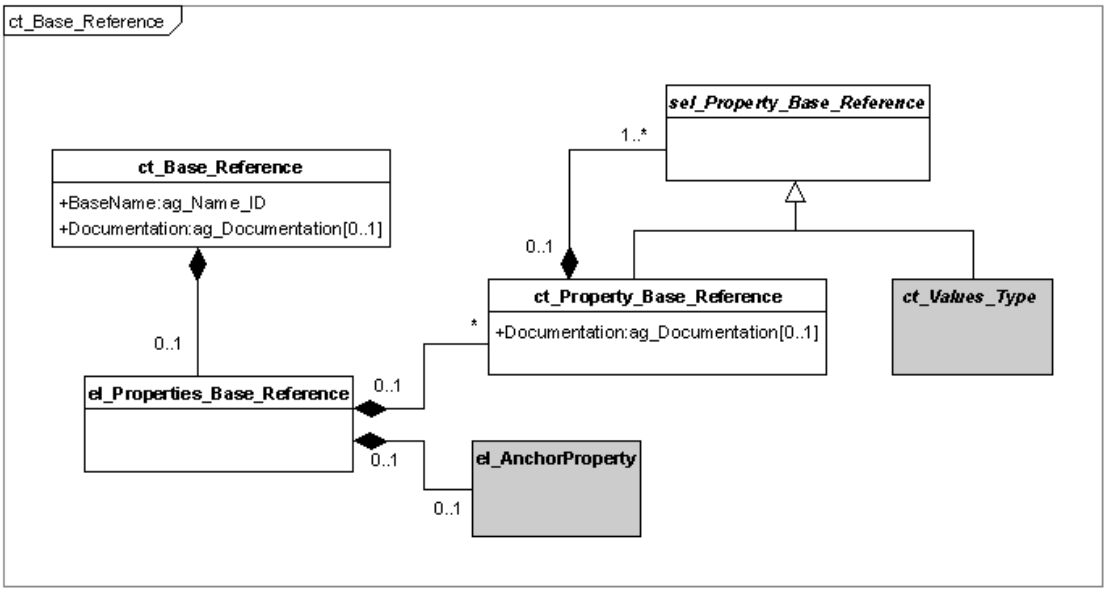

Abbildung 26 - Überschreibbare Elemente eines Base-Widgets

4.2.6.2.3 Anpassung von Widgets weiter entfernter Generationen

Die Überschreibung von Widgets der Enkel- oder weiter entfernter Generationen geschieht analog zur Überschreibung der direkten Kind-Widgets. Jedoch ist die Angabe von Position, Existenz und Dimension des Widgets an dieser Stelle fakultativ.

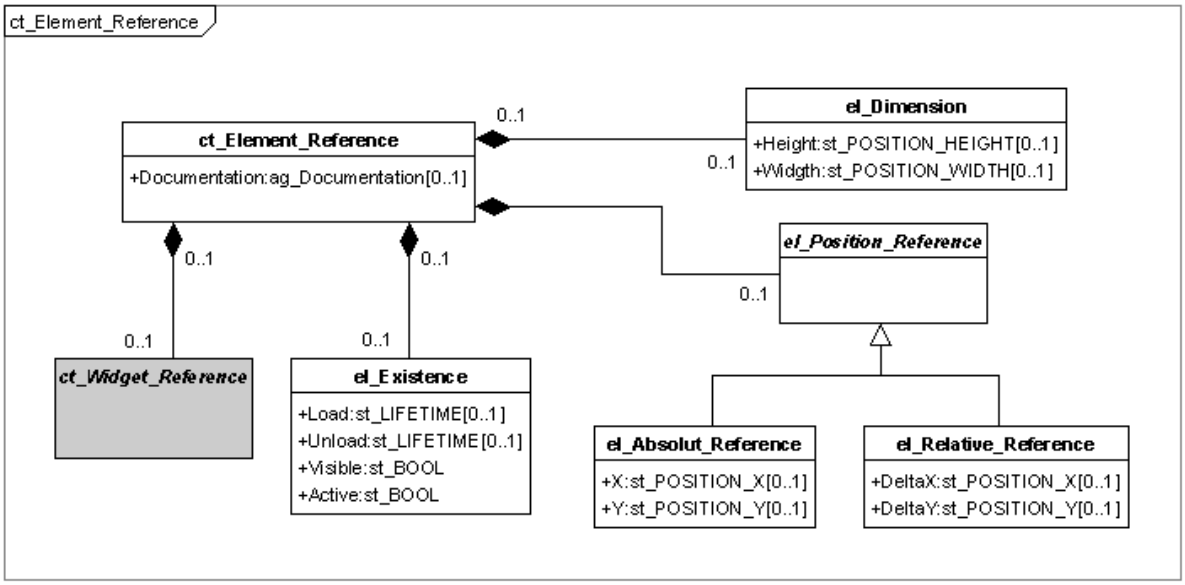

Abbildung 27 - Mögliche Angaben zu Widgets in Überschreibungstiefen von 2 oder höher

Im Beispielsystem wird der Inhalt des Textfeldes "Title" im "PCM_MEDIA_MAIN" überschrieben. Das Setzen des Propertys erfolgt in diesem Fall parallel zu allen anderen Fällen, in denen Propertys gesetzt werden. Es wäre außerdem möglich, gleichzeitig Propertys auf den Zwischenstufen zu überschreiben.

```
<Element Name="ScreenType_PCM_Menu">
    [...]
    <Widget Name="ScreenType_PCM_Menu">
        <Implementation>
            <Defaults>
                <Elements>
                    <Element Name="Title">
                        <Widget Name="PCM_Menu_Title">
                            <Implementation>
                                <Defaults>
                                    <Elements>
                                        <Element Name="Title">
                                            <Widget Name="TextField">
                                                <Base Name="TextField">
                                                    <Properties>
                                                        <Property Name="TextFieldProperties">
                                                            [...]
                                                        </Property>
                                                    </Properties>
                                                </Base>
                                            </Widget>
                                        </Element>
                                    </Elements>
                                </Defaults>
                            </Implementation>
                        </Widget>
                    </Element>
                </Elements>
            </Defaults>
        </Implementation>
    </Widget>
</Element>
```

4.6.2.3 Zustände

Das Anlegen von verschiedenen Zuständen in einem Widget erlaubt es, Informationen durch unterschiedliche Darstellungen von Kind-Widgets zu visualisieren. Zustände können außerdem verwendet werden um die Bedienabläufe des Systems den äußeren Umständen anzupassen.

In jedem definierten State ist es möglich, alle Einstellungen für die Elemente des Widgets, die auch allgemein überschrieben oder gesetzt werden können, speziell für diesen State zu ändern. Dabei ist zu beachten, dass diese Änderungen nur dann wirksam werden, wenn sich das Eltern-Widget in eben diesem State befindet. Ebenfalls ist es wichtig zu wissen, dass Werte, die in States gesetzt wurden, nur durch das Überschreiben in diesen States von übergeordneten Widgets aus geändert werden können. (siehe Überschreibungsregeln)

Über das StateProperty kann einerseits bestimmt werden, in welchem State sich ein Widget beim ersten Anzeigen befindet. Andererseits kann dort auch immer der aktuelle State eines Widgets abgefragt werden.

Constraints:

1. Der CurrentState aus der Transition muss ein State des korrespondierenden Widget sein.

   context el_Transition **inv**:
   self.CurrentState.el_Definition_Root.ct_Widget_Definition =
   self.el_InOut.el_Behavior.el_Implementation_Root.ct_Widget_Definition

2. Der NextState aus der Transition muss ein State des korrespondierenden Widgets sein.

   context el_Transition **inv**:
   self.NextState.el_Definition_Root.ct_Widget_Definition =

self.el_InOut.el_Behavior.el_Implementation_Root.ct_Widget_Definition

3. Der CurrentState aus dem StateProperty muss im korrespondierenden Widget definiert sein.

context el_StateProperty **inv**:
self.CurrentState.el_Definition_Root.ct_Widget_Definition =
self.el_Properties_Implementation.el_Implementation_Root.ct_Widget_Definition

4. Jeder State aus dem StateProperty muss im korrespondierenden Widget definiert sein.

context el_StateProperty **inv**:
self→forAll(s: el_StateProperty_State | s.UsedState.el_Definition_Root.ct_Widget_Definition
= self.el_Properties_Implementation.el_Implementation_Root.ct_Widget_Definition)

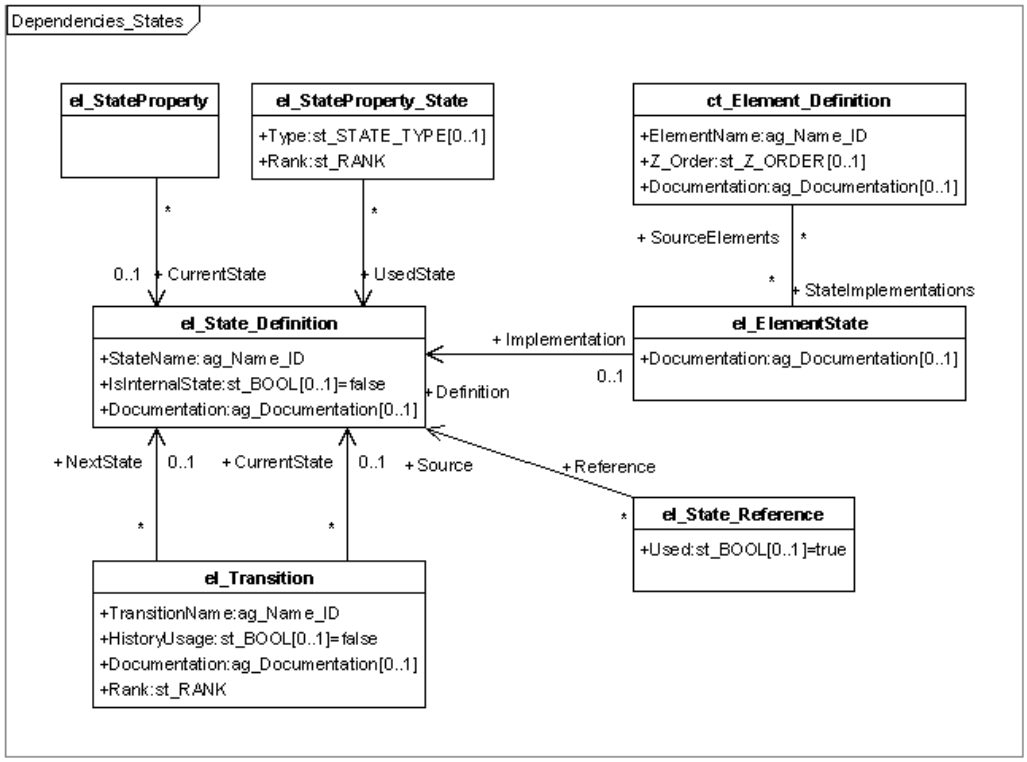

Abbildung 28 - Relationen zwischen der Definition eines States und seinen Referenzen

Das Menü "PCM_Off" definiert in seinem Definitionsteil zwei mögliche Zustände "BlackRed" und "Beauty".

```
<Definition>
    <Elements>
        [...]
    </Elements>
    <States>
        <State Name="BlackRed"/>
        <State Name="Beauty"/>
    </States>
</Definition>
```

4.6.2.4 Das Verhalten

Die Klasse *el_Behavior* leitet die Beschreibung des Verhaltens eines Widgets ein. Dadurch kann sich die Darstellung des Widgets abhängig von äußeren Einflüssen ändern. So könnte zum Beispiel der Wechsel in einen anderen Zustand als Reaktion auf ein Ereignis (*el_Message_In*) eine Änderung hervorrufen. Auch die Verwendung von Funktionen zur Anzeige von Systeminformationen kann Änderungen bewirken.

Die Grundstruktur des Verhaltens sieht wie folgt aus.

```
<Behavior>
    <Functions>
        <Function Name=" ">
        [...]
        </Function>
    </Functions>
    <DataSet>
        <Static>
            <Data Name="" AccessType="" ValueType="">
                [...]
            </Data>
        </Static>
    </DataSet>
    <InOut>
        <InMessages>
            <Message Name="" Type="" MessageID="">
                [...]
            </Message>
        </InMessages>
        <OutMessages>
            <Message Name="" Type="" MessageID="">
                [...]
            </Message>
        </OutMessages>
        <StateTable>
            <Transitions>
                <Transition Name="">
                    [...]
                </Transition>
            </Transitions>
        </StateTable>
    </InOut>
</Behavior>
```

4.4.6.2.1 Verwendung von Funktionen im Behavior

Im Verhalten eines Widgets werden alle Methoden (*ct_Function*) aufgeführt, die im Widget verwendet werden, um z. B. Bedingungen auszuwerten, dynamische Werte anzuzeigen oder Einstellungen, die der Anwender getätigt hat, an die Komponenten weiterzuleiten.

Das Element *el_Behavior* enthält dafür das Element *el_Functions*, das wiederum die Elemente *ct_Function*, *el_Entry* und *el_Leave* enthält.

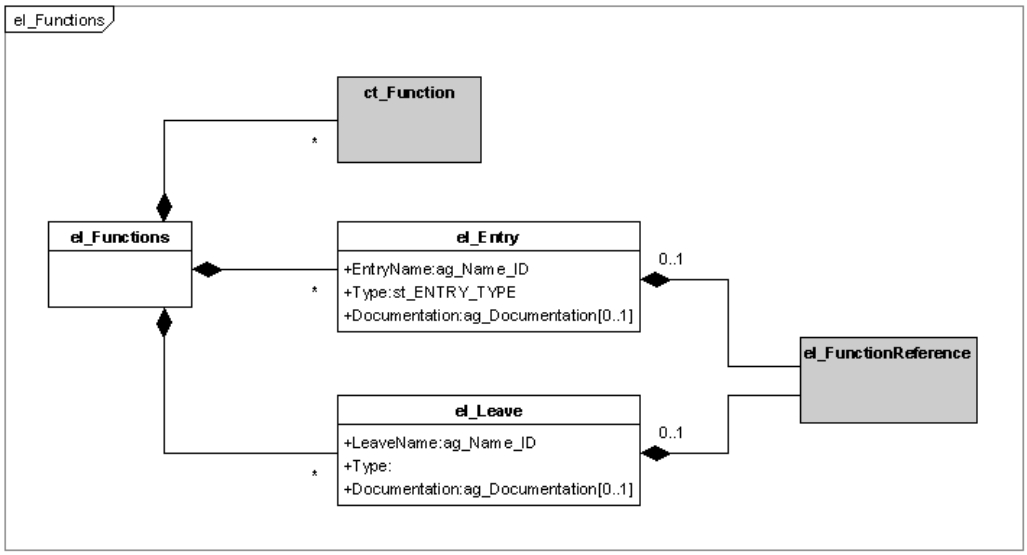

Abbildung 29 - Grundstruktur von Funktionen im Implementationsteil

Die Klasse *ct_Function* entspricht einer mehr oder weniger komplexen Zusammenfassung von Funktionsaufrufen, die aus *el_FunctionReference*s und *el_Call*s aufgebaut sind.

Constraints:

1. Alle Funktionen eines Widgets müssen eindeutige Namen haben.

```
context el_Functions inv:
self.ct_Function->forAll(f1:ct_Function|
    self.ct_Function->forAll(f2:ct_Function|
            f1<>f2 implies f1.FunctionName.Name <> f2.FunctiontName.Name))
```

2. Eine Funktion enthält mindestens entweder einen Call oder eine Funktionsreferenz.

```
context ct_Function inv:
self.el_FunctionReference->isEmpty() implies self.el_Call->notEmpty() and
    self.el_Call->isEmpty() implies self.el_FunctionReference->notEmpty()
```

3. Eine Funktion, welche die ReturnValueSource "FUNCTION_REFERENCE" hat, ist mit einer FunctionReference, aber nicht mit einem Call, einem Command oder einem Parameter verbunden.

```
context ct_Function inv:
self.ReturnValueSource = 'FUNCTION_REFERENCE' implies
    (self.Function->notEmpty() and self.RetVal_Call->isEmpty() and
     self.RetVal_Command->isEmpty() and self.RetVal_Parameter->isEmpty())
```

4. Eine Funktion, welche die ReturnValueSource "COMMAND" hat, ist mit einem Call und einem Command, aber nicht mit einem Parameter verbunden.

```
context ct_Function inv:
self.ReturnValueSource = 'COMMAND' implies
    (self.RetVal_Call->notEmpty() and self.RetVal_Command->notEmpty() and
```

self.RetVal_Parameter->isEmpty())

5. Eine Funktion, welche die ReturnValueSource "PARAMETER" hat, ist mit einem Call, einem Command und einem Parameter verbunden.

context ct_Function **inv**:
self.ReturnValueSource = 'PARAMETER' implies
(self.RetVal_Call->notEmpty() and self.RetVal_Command->notEmpty()
and self.RetVal_Parameter->notEmpty())

Die Methode "GetTemperature" aus dem Widget "PCM_StatusLine" ist ein Beispiel für eine einfache Funktion mit nur jeweils einem Call und Command.

```
<Functions>
    [...]
    <Function
        Name="GetTemperature"
        FunctionReturnValueType="FUNCTION_RETURN_VALUE_COMMAND"
        CallReference_Name="General.GetTemperature"
        CommandReference_Name="General.GetTemperature"
        FunctionReturnValueDataType="DATA_TYPE_SINT">

        <Call Name="General.GetTemperature">
            <API Type="API_TYPE_FUNCTION" Target="ABSTRACTION_LAYER"/>
            <Commands>
                <Command
                    Name="General.GetTemperature"
                    Type="COMMAND_TYPE_ASYNCHRON"
                    CommandReturnValueDataType="DATA_TYPE_SINT"/>
            </Commands>
        </Call>
    </Function>
</Functions>
```

4.4.6.2.2 Calls

Ein Call (*el_Call*) besteht aus dem Element *el_API*, das eine Verbindung zu einem unter *ct_Function_Definition* definiertem Device oder Subdevice darstellt, eventuell einer ct_Condition, welche die Ausführung des Calls bedingt, und den eigentlichen Funktionsaufrufen (*ct_Commands*). *el_Call* besitzt außerdem ein Attribut *Rank*, mit dem man die Abarbeitungsreihenfolge der Calls festlegen kann.

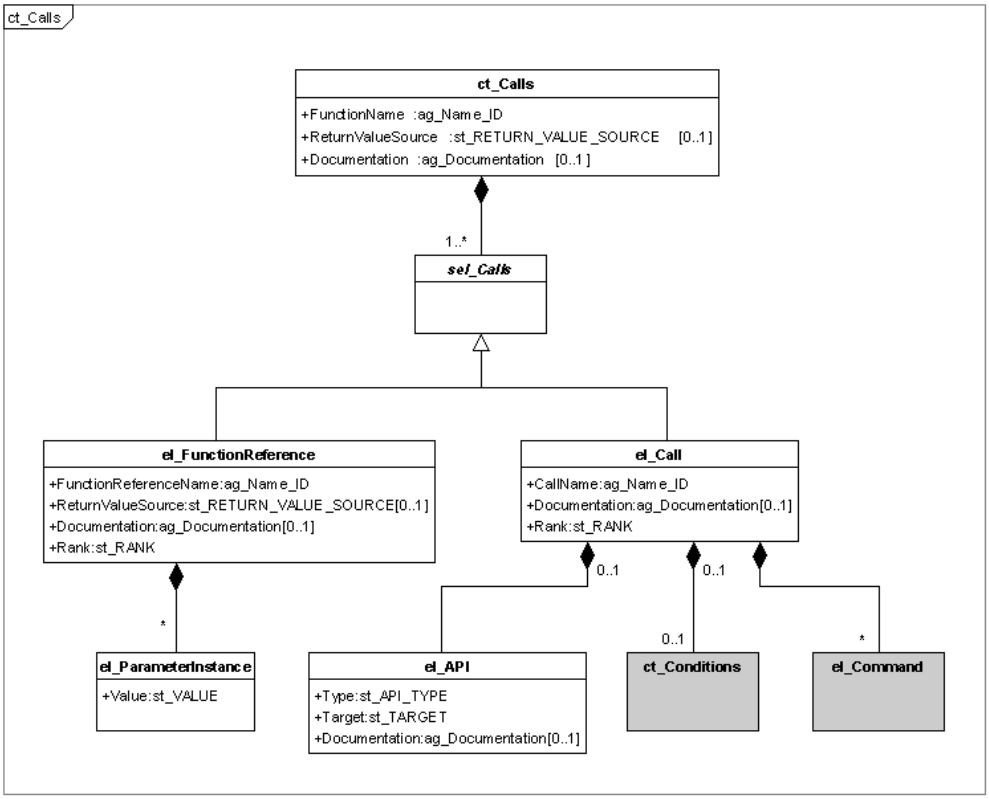

Abbildung 30 - Aufbau von ct_Function

4.4.6.2.3 Kommandos

Ein Kommando (*el_Command*) ist ein Aufruf einer Device-Funktion, die in *ct_Function_Definition* definiert worden ist. Über *el_ParameterCommand* werden die Parameter der Funktion mit den gewünschten Werten gefüllt. Da in einem Call mehrere Kommandos aufgerufen werden können, haben auch Kommandos ein Attribut Rank, das ihre Abarbeitungsreihenfolge festlegt. Die Kommandos, die unter einem Call zusammengefasst sind, müssen sich alle auf Funktionen des gleichen Devices bzw. Subdevices beziehen.

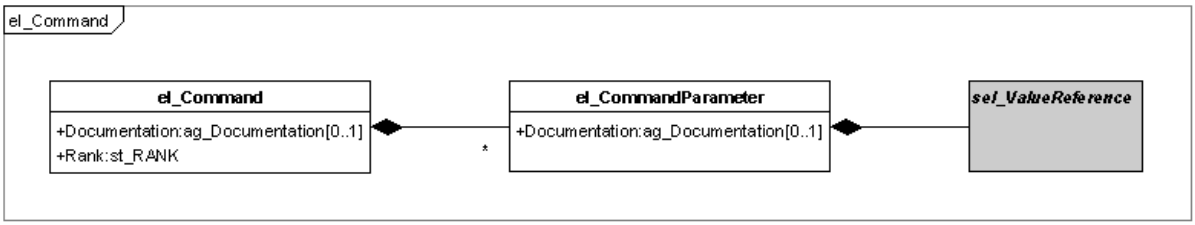

Abbildung 31 - Struktur eines Kommandos

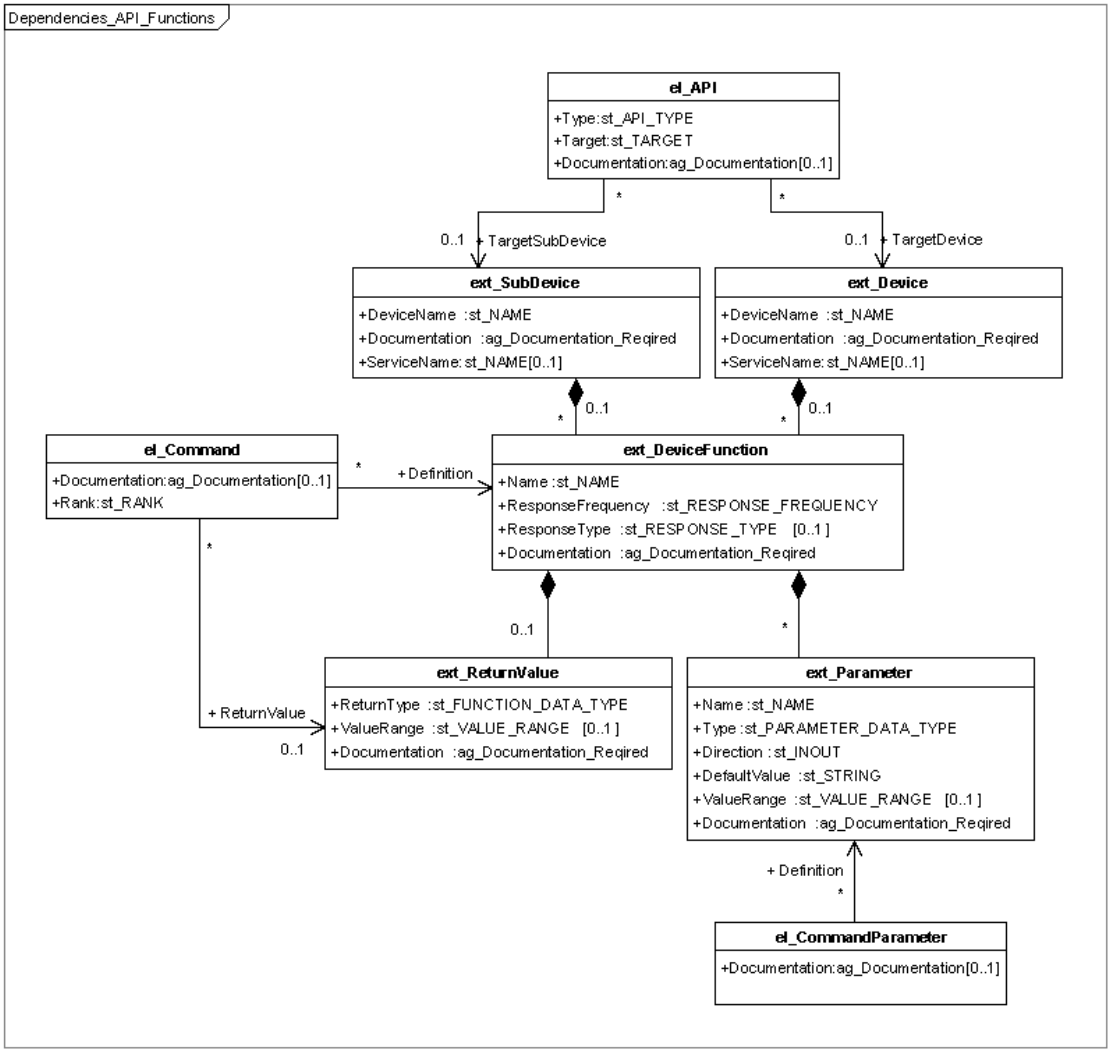

Abbildung 32 - Relation zwischen API-Funktionen und den Widget-Commands

4.4.6.2.4 Referenzen auf Funktionen

Eine Funktionsreferenz (*el_FunctionReference*) ist eine Referenz auf einen schon bestehenden Call, wobei die Möglichkeit besteht, die Belegung der jeweiligen Parameter durch die Verwendung von *el_ParameterInstance*s zu verändern. Funktionsreferenzen werden nicht nur dazu verwendet für komplexe Aufgaben Funktionen zusammenzufassen, sondern auch um an anderen Stellen des Widget von Funktionen Gebrauch machen zu können (Transition, *el_Function_Data*).

Constraints:

1. Wenn eine FunctionReference die ReturnValueSource "FUNCTION_REFERENCE" hat, ist sie weder mit einem Call, einem Command noch einem Parameter verbunden.

context el_FunctionReference **inv**:
self.ReturnValueSource = 'FUNCTION_REFERENCE' implies
(self.RetVal_Call->isEmpty() and self.RetVal_Command->isEmpty()
and self.RetVal_Parameter->isEmpty())

2. Wenn eine FunctionReference die ReturnValueSource "COMMAND" hat, ist sie mit einem Call und einem Command, aber nicht mit einem Parameter verbunden.

context el_FunctionReference **inv**:
self.ReturnValueSource = 'COMMAND' implies
(self.RetVal_Call->notEmpty() and self.RetVal_Command->notEmpty()
and self.RetVal_Parameter->isEmpty())

3. Wenn eine FunctionReference die ReturnValueSource "PARAMETER" hat, ist sie mit einem Call, einem Command und einem Parameter verbunden.

context el_FunctionReference **inv**:
self.ReturnValueSource = 'PARAMETER' implies
(self.RetVal_Call->notEmpty() and self.RetVal_Command->notEmpty()
and self.RetVal_Parameter->notEmpty())

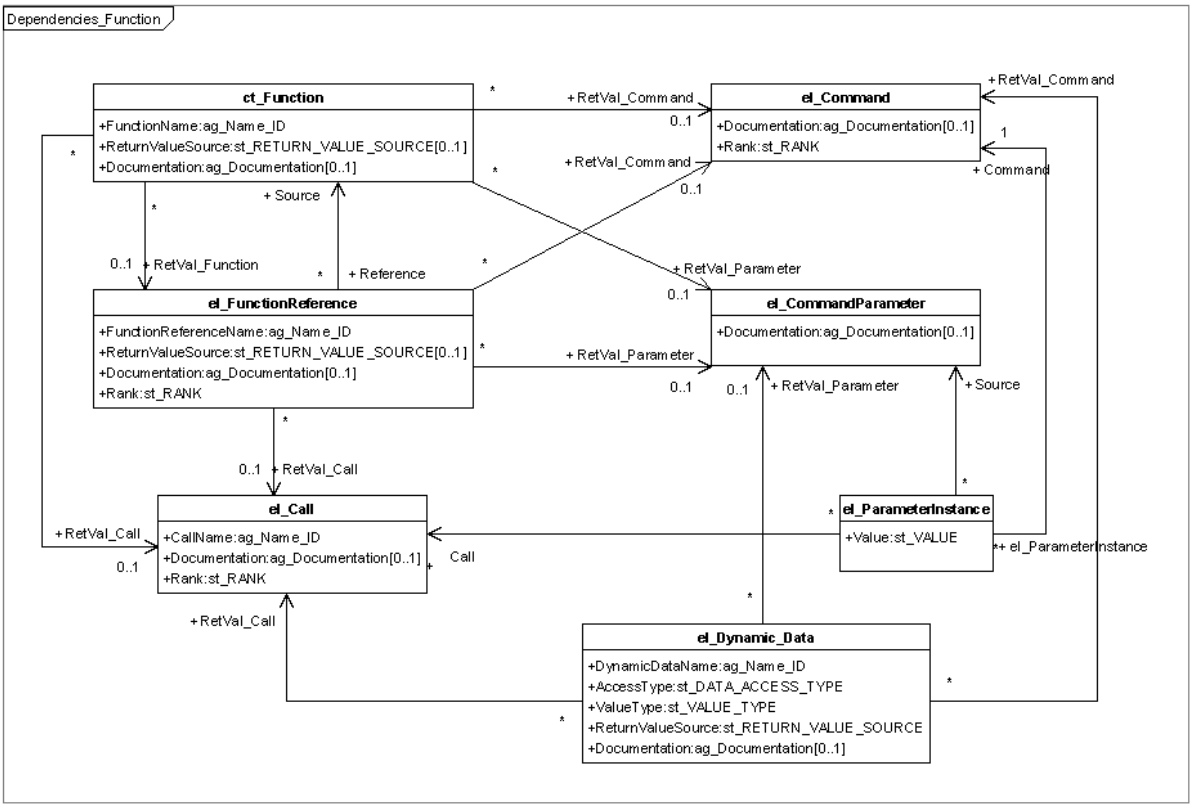

Abbildung 33 - Definition der Rückgabewerte bei Funktionen und Funktionsreferenzen

4.4.6.2.5 Variablen

Die Variablen werden in IML verwendet, um Informationen zu speichern, die nur während der Laufzeit von Bedeutung sind. Es gibt statische und dynamische Variablen. Die statischen Variablen (*el_Static_Data*) werden hauptsächlich dafür verwendet, um Inhalte von empfangenen Nachrichten zu speichern. Diese gespeicherten Daten können dann für Vergleiche in Bedingungen und als Datenlieferanten für ausgehende Nachrichten verwendet werden.

Über die statischen Variablen ist es auch möglich, verschiedene Widgets zu synchronisieren. Es wird beispielsweise gewünscht, dass sich gewisse Fehlermeldungen bei Eingang einer Systemmeldung schließen, wenn vorher eine gewisse Zeitspanne vergangen ist. Das Widget, das diese Fehlermeldung darstellt, kann zwei Nachrichten empfangen. Die eine ist die Nachricht eines Timers, dass dieser abgelaufen ist. Die Zweite ist die erwähnte Systemmeldung. Bei Eingang jeder dieser beiden Nachrichten, wird eine Variable gesetzt und die Variable der anderen Nachricht abgefragt. Wenn die andere Variable noch nicht gesetzt ist, geschieht nichts. Ist sie jedoch gesetzt, wird die Fehlermeldung geschlossen.

Die dynamischen Variablen (*el_Dynamic_Data*) werden mit einer FunctionReference verbunden, die sie mit Daten füllt. Dadurch kann zum Beispiel der Inhalt einer ausgehenden Nachricht auch mit aktuellen, laufzeitabhängigen Daten gefüllt werden. Die dynamischen Variablen haben ansonsten das gleiche Verhalten, dass man auch durch die Verwendung der Punktnotation und des Schlüsselworts .FUNCTION erreichen kann.

Sowohl die dynamischen als auch die statischen Variablen haben einen jeweils widgetweit eindeutigen Namen. Für beide kann jeweils angegeben werden, wer sie setzen und lesen kann (*Accesstype*), wobei die zugelassen Werte "Private", "Public" und "Protected" sind. Auf Variablen mit dem Accesstype "Private" kann nur das Widget selber zugreifen, beim Accesstype "Protected" können dies auch die direkten Kinder und Variablen mit dem Accesstype "Public" sind für jeden verwendbar.

Alle Variablen haben des Weiteren einen festegelegten Datentyp (*ValueType*). Wenn der Datentyp der Variablen von dem Datentyp des Wertes, der in ihr gespeichert werden soll abweicht, muss ein Cast durchgeführt werden.

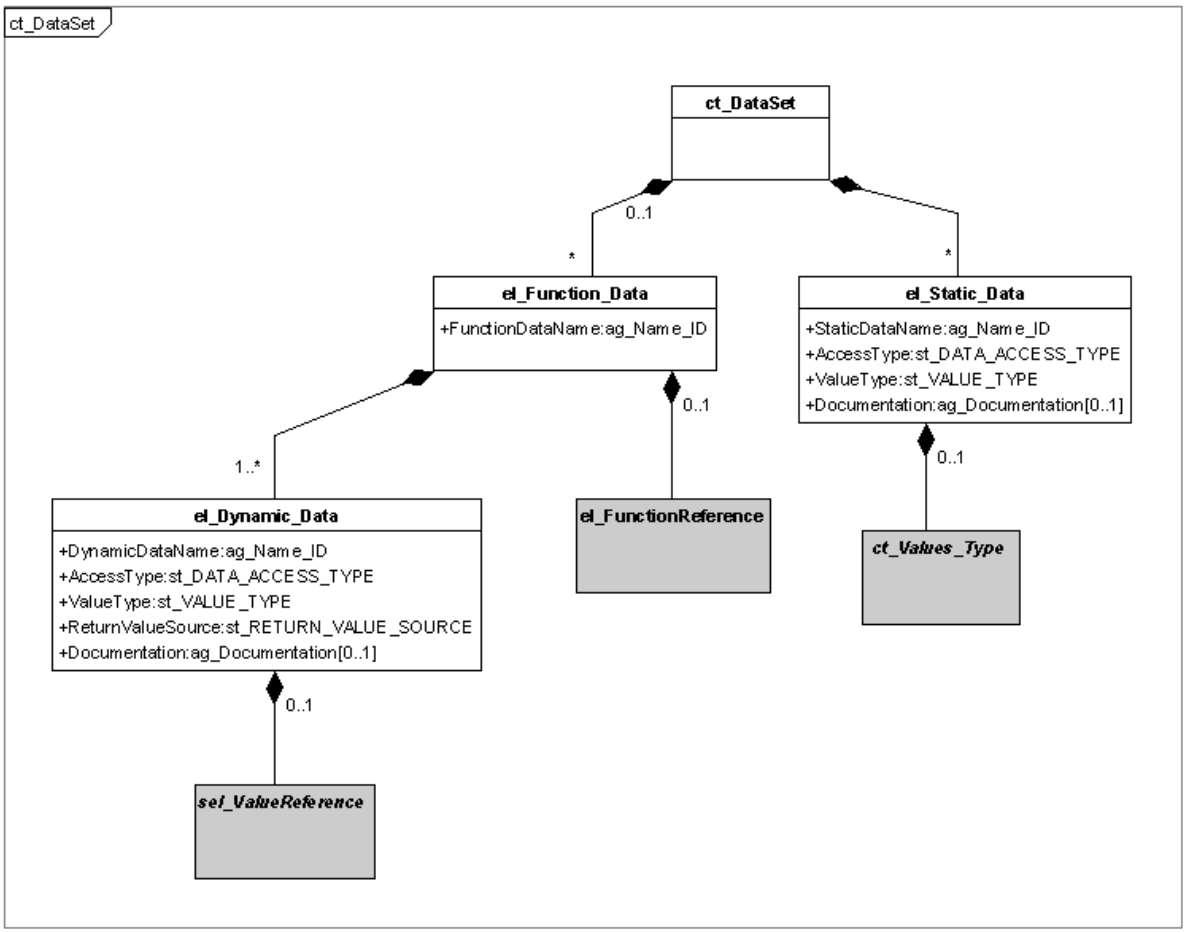

Abbildung 34 - Die statischen und dynamischen Variablen

Constraints:

1. Alle statischen Variablen müssen einen Widget weit eindeutigen Namen haben.

context ct_Data_Set **inv**:
self.el_Static_Data->forAll(sd1:el_Static_Data|
self.el_Static_Data->forAll(sd2:el_Static_Data|
sd1<>sd2 implies sd1.StaticDataName.Name <> sd2.StaticDataName.Name))

2. Alle Function_Data müssen einen Widget weit eindeutigen Namen haben.

context ct_Data_Set **inv**:
self.el_Function_Data->forAll(fd1:el_Function_Data|
self.el_Function_Data->forAll(fd2:el_Function_Data|
fd1<>fd2 implies fd1.FunctionDataName.Name <> fd2.FunctionDataName.Name))

3. Alle dynamischen Variablen müssen einen eindeutigen Namen innerhalb ihres Function_Data haben.

context el_Function_Data **inv**:
self.el_Dynamic_Data->forAll(d1:el_Dynamic_Data|
self.el_Dynamic_Data->forAll(d2:el_Dynamic_Data|
d1<>d2 implies d1.DynamicDataName.Name <> d2.DynamicDataName.Name))

4. Wenn eine dynamisch Variable die ReturnValueSource "FUNCTION_REFERENCE" hat, ist sie weder mit einem Call, einem Command noch einem Parameter verbunden.

```
context el_Dynamic_Data inv:
self.ReturnValueSource = 'FUNCTION_REFERENCE' implies
    (self.RetVal_Call->isEmpty() and self.RetVal_Command->isEmpty()
    and self.RetVal_Parameter->isEmpty())
```

5. Wenn eine dynamische Variable die ReturnValueSource "COMMAND" hat, ist sie mit einem Call und einem Command, aber nicht mit einem Parameter verbunden.

```
context el_Dynamic_Data inv:
self.ReturnValueSource = 'COMMAND' implies
    (self.RetVal_Call->notEmpty() and self.RetVal_Command->notEmpty()
    and self.RetVal_Parameter->isEmpty())
```

6. Wenn eine dynamische Variable die ReturnValueSource "PARAMETER" hat, ist sie mit einem Call, einem Command und einem Parameter verbunden.

```
context el_Dynamic_Data inv:
self.ReturnValueSource = 'PARAMETER' implies
    (self.RetVal_Call->notEmpty() and self.RetVal_Command->notEmpty()
    and self.RetVal_Parameter->notEmpty())
```

4.4.6.2.6 Kommunikation und Zustandswechsel

Durch Überschreibungen können die zusammenarbeitenden Widgets an die Gegebenheiten der jeweiligen Situation angepasst werden. Durch die Verwendung der Punktnotation können Elemente von Widgets verschiedener Ebenen verbunden werden. Durch das Schlüsselwort ".FUNCTION" ist auch die Verwendung von laufzeitabhängigen Werten möglich. Damit jedoch das eingebaute Verhalten zweier Widgets zusammen das gewünscht Ergebnis liefert, müssen sie miteinander kommunizieren.

Die Kommunikation zwischen den Widgets erfolg über Nachrichten. Jedes Widget spezifiziert, welche Nachrichten es empfangen (*el_Message_In*) und welche es versenden kann (*el_Message_Out*). Alle eingehenden und ausgehenden Nachrichten (InMessages und OutMessages) verweisen auf Nachrichten aus der externen Definitionsdatei.

Mehrere In- bzw. OutMessages eines Widgets können auf die gleiche Nachrichtendefinition verweisen. Deshalb werden alle Nachrichten innerhalb ihrer Gruppe mit eindeutigen Namen versehen. Obwohl immer möglich, sollten eine In- und OutMessage nur dann den gleichen Namen tragen, wenn sie auch auf dieselbe Nachrichtendefinition verweisen. Da die Mehrfachverwendung einer Nachrichtendefinition innerhalb der InMessages zu Mehrdeutigkeiten bei der Auswahl der relevanten Nachricht führen kann, benötigen sie ein "Rank" Attribut. Bei den OutMessages ist dies nicht notwendig, da sie in den Transitionen eindeutig ausgewählt werden müssen.

Wenn eine Nachricht ein Widget erreicht, wird zuerst geprüft, ob die Id der Nachricht zu einer der InMessages passt. Wenn dies für mehr als eine InMessage der Fall ist, werden die mögli-

chen Kandidaten nach dem Wert ihres Rank-Attributs sortiert. Danach werden die Übereinstimmungen mit den Parametern der Nachricht überprüft. An dieser Stelle wird das Sonderverhalten der InMessages bezüglich der Verwendung von Value und Referenzen bedeutsam. Wenn in der InMessage für einen Parameter ein Value angegeben ist, dann wird dieser Wert mit dem Inhalt des Parameters der Nachricht verglichen. Die InMessage kann nur dann relevant sein, wenn alle angegeben Parameter, die ein Value enthalten, mit ihren Konterparts übereinstimmen. Wenn ein Parameter, der in der Nachricht enthalten ist, in der InMessage nicht angegeben wurde, so gilt der Vergleich als bestanden.

Wenn die Parameter der InMessage nicht als Value sondern als Referenz angegeben werden, erfolgt kein Vergleich. Die Werte, die in der Nachricht enthalten sind, werden dann in den angegebenen Variablen gespeichert. Der nicht erfolgte Vergleich gilt trotzdem als bestanden. Nach dieser Methode wird die Liste der in Frage kommenden InMessages überprüft, bis eine passende gefunden wurde.

OutMessages können nur Values in ihren Parametern enthalten. Sie können dort zum Beispiel auch die gespeicherten Werte aus den Variablen verwenden, um sie weiter zu transportieren.

Constraints:

1. InMessages müssen einen Widget weit eindeutigen Namen haben.

```
context el_InMessages inv:
self.el_Message_In->forAll(m1:el_Message_In|
    self.el_Message_In->forAll(m2:el_Message_In|
            m1<>m2 implies m1.MessageName.Name <> m2.MessageName.Name))
```

2. Die Parameter einer InMessage müssen eindeutige Namen haben.

```
context el_Message_In inv:
self.ct_ParameterInMessage->forAll(p1:ct_ParameterInMessage|
    self.ct_ParameterInMessage_In->forAll(p2:ct_ParameterInMessage|
            p1<>p2 implies p1.Key <> p2.Key))
```

3. OutMessages müssen einen Widget weit eindeutigen Namen haben.

```
context el_OutMessages inv:
self.el_Message_Out->forAll(m1:el_Message_Out|
    self.el_Message_Out->forAll(m2:el_Message_Out|
            m1<>m2 implies m1.MessageName.Name <> m2.MessageName.Name))
```

4. Die Parameter einer OutMessage müssen eindeutige Namen haben.

```
context el_Message_Out inv:
self.ct_ParameterOutMessage->forAll(p1:ct_ParameterOutMessage|
    self.ct_ParameterOutMessage->forAll(p2:ct_ParameterOutMessage|
            p1<>p2 implies p1.Key <> p2.Key))
```

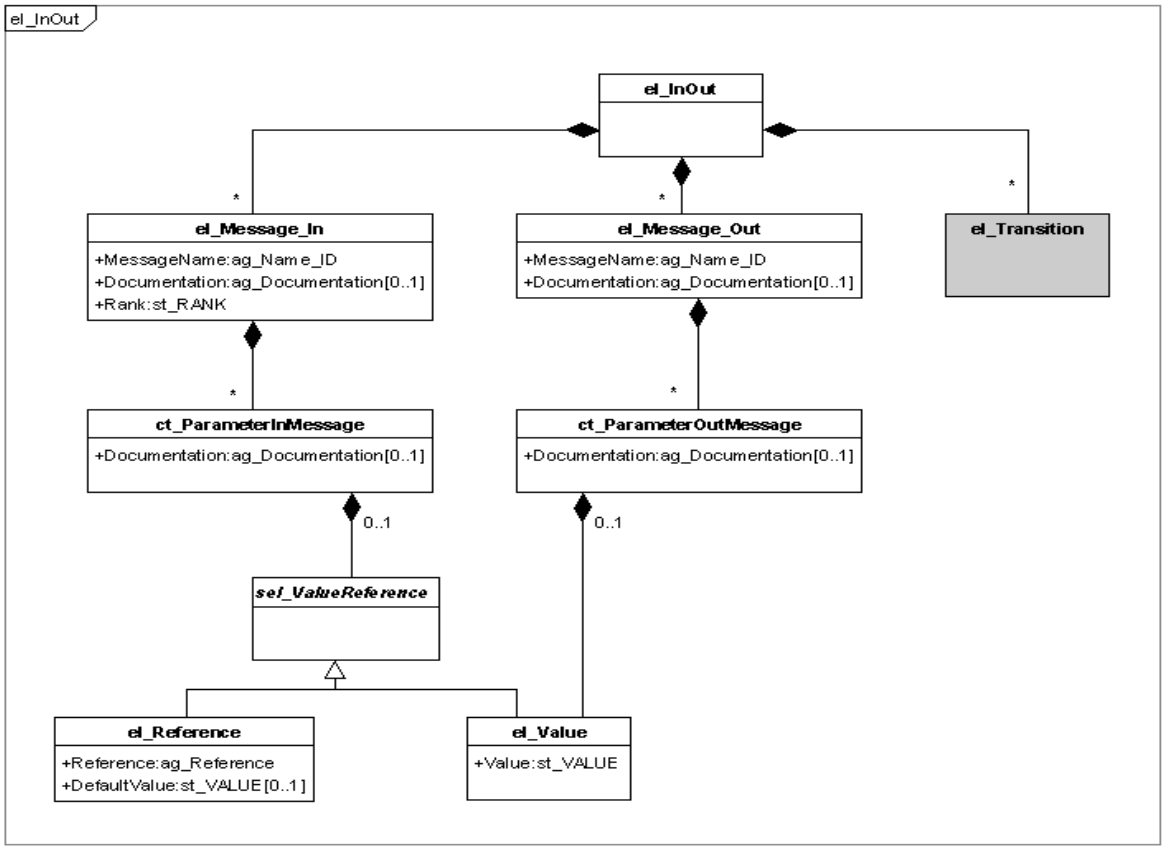

Abbildung 35 - Definition der Nachrichten im Widget

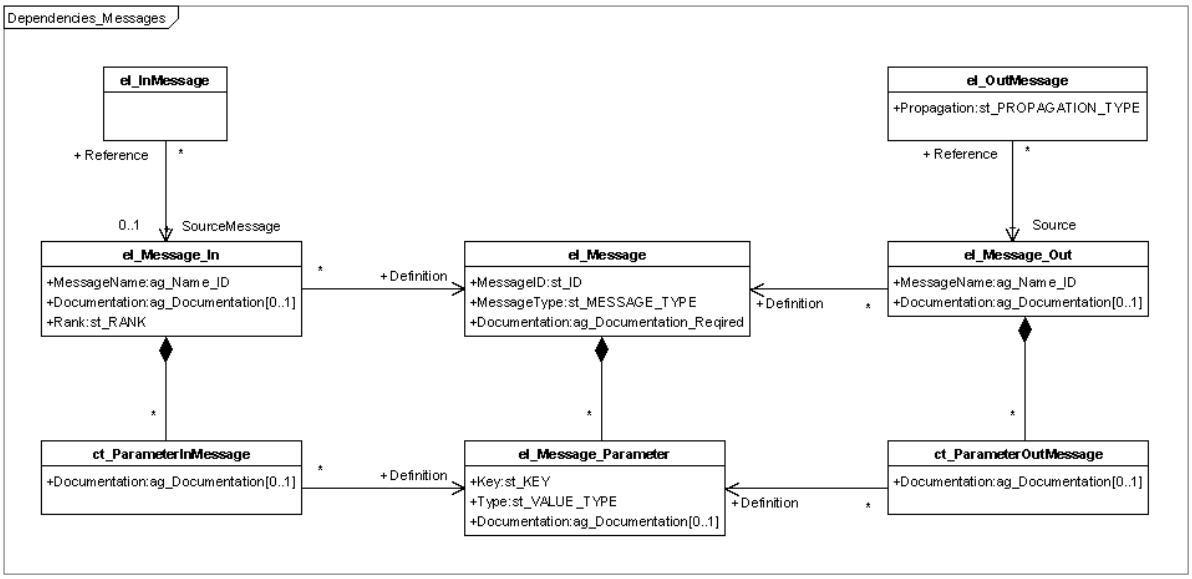

Abbildung 36 - Verbindung zwischen Nachrichten im Widget und ihrer Definition im System

Um die Reaktion auf empfangende Nachrichten zu spezifizieren, werden Transitionen (*el_Transition*) angelegt. Der Begriff Transition wurde gewählt, weil diese Struktur zur Beschreibung von Zustandsübergängen gedacht war. Bei der Entwicklung von IML hat sich aber schnell herausgestellt, dass auch andere Reaktionen beschrieben werden müssen. Deshalb müssen Transitionen in der aktuellen Version von IML nicht mehr unbedingt einen Zustandsübergang beschreiben. Übergänge, die im gleichen Zustand enden in dem sie anfangen, even-

tuell auch unabhängig vom aktuellen Zustand sind zulässig. Wenn für eine Transition kein CurrentState angegeben wird, bedeutet dies, dass diese Transition von jedem State ausgehen kann. Das Fehlen des NextState bedeutet, dass der aktuelle State nicht verlassen wird.

Für eine Transition zwingend sind dagegen eine eindeutiger Name, ein Rang und die Verbindung zu einer der möglichen eingehenden Nachrichten. Es kann mehrere Transitionen geben, die dieselbe Nachricht als Auslöser besitzen. Um in einem solchen Fall zu entscheiden, welche Transition ausgeführt werden soll, werden einerseits der Rang der Transition und andererseits die Bedingungen, die man für sie definieren kann, beachtet. Dabei entscheidet der Rang zwischen den Transitionen, die nach Auswertung der Bedingungen gültig sind. Die OutMessages enthalten *Propagation* als Attribut. Dort wird angegeben, an wen diese Nachricht zu senden ist. In den momentanen Projekten werden die Propagationtypen "Parent", "Children" und "This" verwendet. "Parent" bedeutet, dass die Nachricht an das Eltern-Widget gesendet wird. Der Propagationtype "Children" sorgt für eine Verteilung der Nachricht an alle direkten Kind-Widgets und "This" sendet die Nachricht an sich selbst.

Es wurde in den Projekten festgelegt, dass Nachrichten immer nur eine Widgetebene überwinden können. Das bedeutet, dass Nachrichten explizit durch Widgets hindurchwandern müssen, wenn zwei Widgets, die nicht in einer direkten Eltern-Kind-Relation stehen, kommunizieren wollen. Dies bedeuten zwar einerseits, dass das Verhalten von Widgets an die Bedürfnisse ihrer Eltern- und Kind-Widgets angepasst werden müssen, andererseits erlaubt es eine klare Nachverfolgung von Nachrichten und die explizite Unterbrechung von Kommunikation, wenn dies nötig ist.

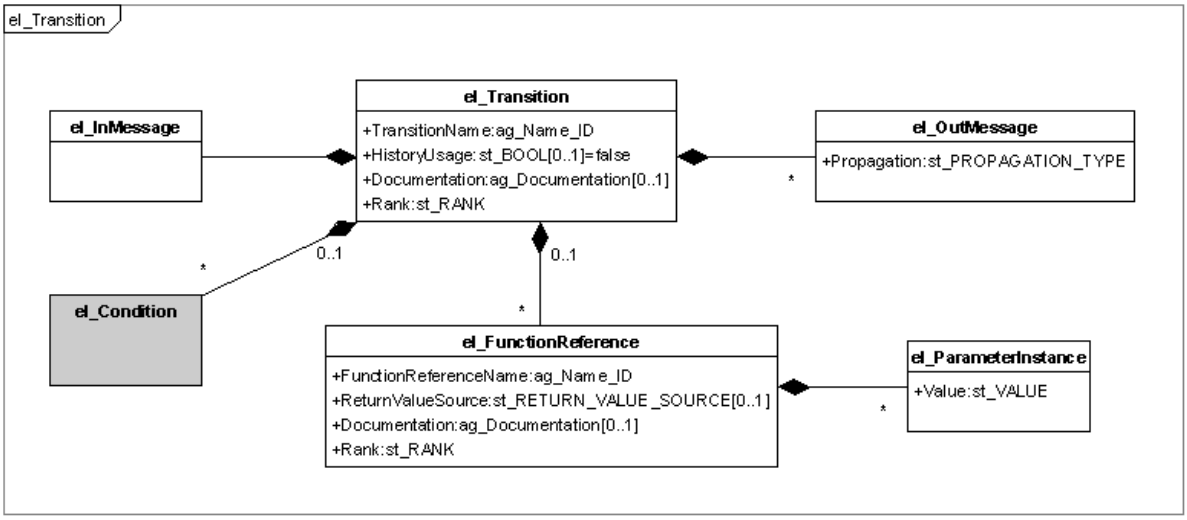

Abbildung 37 - Struktur eines Zustandsübergangs

4.6.3 Das Base-Widget

Base-Widget stellen die Grundelemente des Systems dar, zum Beispiel Image, SensitiveArea, Line, Canvas und Textfield. Sie werden nicht in Definition- und Implementationsteil unterteilt. Sie besitzen nur einen Teil, in dem ihre Propertys gleichzeitig definiert und mit Standartwerten gefüllt werden. Sie haben weder untergeordnete Kind-Widgets noch können sie verschiedenen Zustände annehmen. Deswegen benötigen sie keinen Ort um Überschreibungen zu erlauben. Base-Widgets werden nur über ihre Eigenschaften beschrieben. Sie haben immer den WidgetTyp "WIDGET_TYPE_BASE", damit sie auch für die maschinelle Verarbeitung erkennbar sind. Das Verhalten eines Base-Widgets ist nicht Teil der Spezifikation, sondern wird vor oder zum Projektbeginn zwischen den beteiligten Parteien abgestimmt.

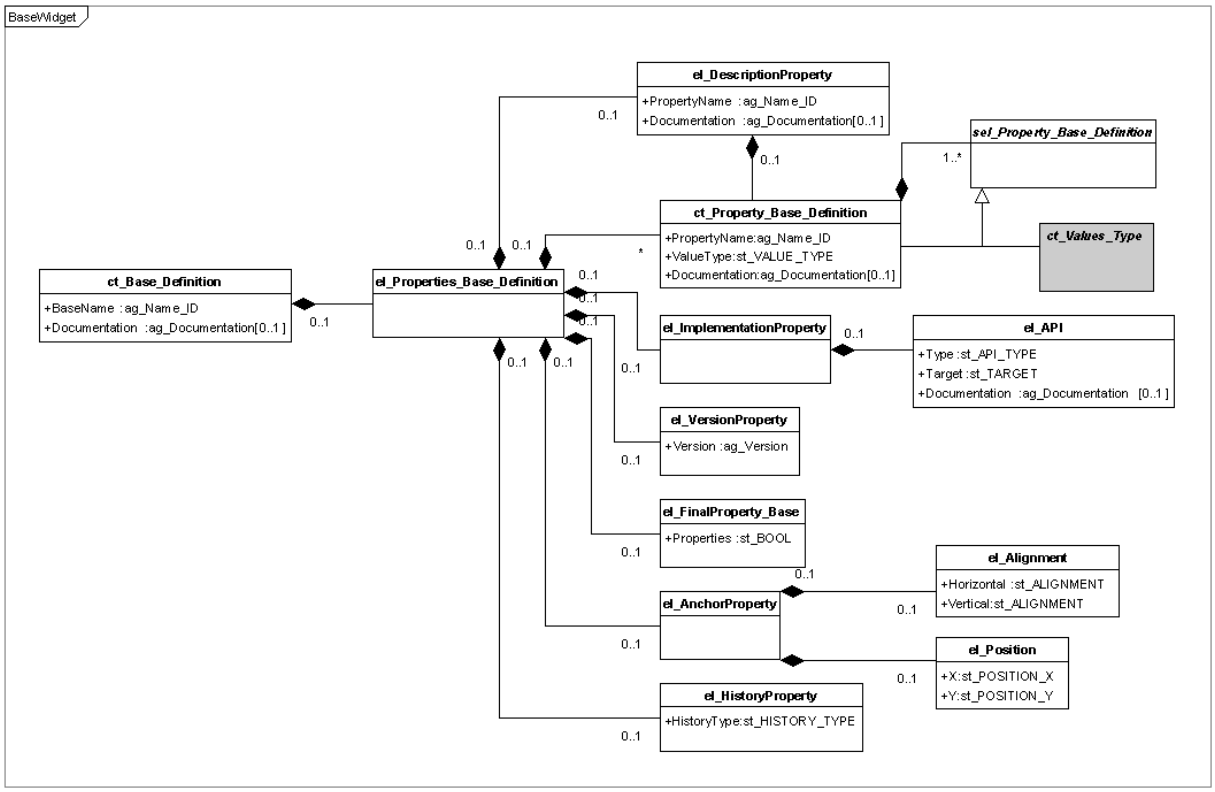

Abbildung 38 - Struktur eines Base-Widgets

4.6.3.1 Propertys im Base-Widget

Es gibt einige Unterschiede zwischen den Propertys der Base-Widgets und denen von komplexen Widgets. Beide besitzen jedoch generelle und spezifische Propertys.

Die generellen Propertys des Base-Widgets entsprechen denen der komplexen Widget bis auf zwei Ausnahmen. Da das Base-Widget keine States besitzen kann, benötigt es auch das StateProperty nicht. Dieses ist also im Base-Widget nicht existent.

Das andere generelle Property, das sich von seinem Gegenpart im komplexen Widgets unterscheidet, ist das FinalProperty. Wie schon bei der Beschreibung des komplexen Widgets festgestellt, dient das FinalProperty dazu, Änderungen zu verbieten. Dies ist auch seine Aufgabe im Base-Widget. Allerdings können bei einem Base-Widget, bedingt durch seinen Aufbau, nur Propertys verändert werden. Aus diesem Grund kann das FinalProperty auch nur Änderungen an denselben verbieten.

Die spezifischen Propertys der Base-Widgets enthalten die gleichen Informationen, wie die der komplexen Widgets. Allerdings werden alle Informationen, die beim komplexen Widget auf die zwei möglichen Stellen der Spezifikation aufgeteilt sind, an einer Stelle gebündelt. Der Name, Typ und Standardwert des Propertys wird also an der gleichen Stelle spezifiziert. Trotzdem ist es nicht möglich, den Typ eines spezifischen Propertys durch eine Überschreibung zu ändern.

Die spezifischen Eigenschaften eines Base-Widgets dienen in erster Linie dazu, den darunter liegenden Programmcode an die vorliegenden Bedingungen anzupassen. Je variabler das Gesamtsystem sein soll, desto größer muss die Parametrisierbarkeit der Base-Widgets sein. Wenn versucht würde, das gleiche Resultat mit einer höheren Anzahl von Base-Widgets zu erreichen, würde sich die Anzahl der Widgets des Gesamtsystems nahezu exponential erhöhen, da für jede Ausprägung, die ein Base-Widget annehmen kann, ein entsprechendes Eltern-Widget entstehen würde, das diese Ausprägung verwendet.

Es könnte zum Beispiel angenommen werden, dass ein System Bilder in zwei verschiedenen Formaten anzeigen kann, .bmp und .png. In einem System, das auf einer Vielzahl von Base-Widgets aufbaut, würden diese Formate durch zwei unterschiedliche Base-Widgets repräsentiert: Image_Bmp und Image_Png. Eines der meist verwendeten, komplexeren Elemente des Systems sind Buttons. Sie bestehen bei Touchscreensystemen aus einer SensitiveArea, die ermittelt, ob auf oder neben den Button gedrückt wurde, einem Bild, das den Button visuell repräsentiert und einem Label. Das Label kann je nach Aufgabe ein Textfeld oder ein Bild oder eine Kombination aus beiden sein. Wenn man die anderen Elemente des Buttons erst einmal außer Acht lässt, werden im System mit mehreren Base-Widgets für Bilder zwei Buttontypen benötigt: Button_Bmp und Button_Png.

In den meisten Infotainmentsystemen werden zwei wesentliche Buttontypen verwendet. Die so genannten Push- bzw. Toggle-Buttons. Push-Buttons wechseln beim einmaligen Betätigen ihren Zustand, um ihre Aktivierung anzuzeigen, am Ende der Aktion kehren sie aber in ihren Ausgangszustand zurück. Sie haben also einen festen und einen flüchtigen Zustand. Toggle-Button besitzen zwei feste Zustände, zwischen denen mit einem Druck auf ihre SensitiveArea hin- und hergeschaltet werden kann. Diese Eigenschaft der Button kann man ebenfalls durch ein Widget mit einem entsprechenden Property oder durch zwei unterschiedliche Widgets spezifizieren. Wenn man sich für die unterschiedlichen Widgets entscheiden würde, hätte man in diesem Beispielsystem bereits jetzt vier verschiedene Button: Button_Bmp_Push, Button_Bmp_Toggle, Button_Png_Push und Button_Png_Toggle.

Schon an diesem kleinen Beispiel ist klar zu erkennen, dass die Anzahl der benötigten Widgets schnell kaum noch zu handhaben sein würde. Immerhin wurde noch nicht betrachtet, dass auch die anderen Elemente eines Buttons in mehreren Ausprägungen vorliegen könnten oder dass Buttons an sich im Widget-Baum oft in sehr tiefen Hierarchiestufen verwendet werden. Dies bedeutet vor allem, dass alle Widgets aller Hierarchiestufen oberhalb der Buttons, die beide Buttonausprägungen verwenden könnten, jeweils doppelt vorhanden sein müssten.

Da das Konzept der Propertys dazu eingeführt wurde, die Wiederverwendbarkeit zu erhöhen, sollte besonders bei der Definition der Grundelemente, der Base-Widgets, diese Architektur verwendet werden.

Constraints:

1. Ein ct_Property_Base_Definition enthält entweder genau ein ct_Values_Type oder eine oder mehrere ct_Property_Base_Definition

```
context ct_Property_Base_Definition inv:
self.select(sel_Property_Base_Definition)->size() > 1 implies
    self.sel_Property_Base_Definition->forAll(s.oclIsTypeOf(ct_Property_Base_Definition))
```

2. Ein ct_Property_Base_Reference enthält entweder genau ein ct_Values_Type oder eine oder mehrere ct_Property_Base_Reference

```
context ct_Property_Base_Reference inv:
self.select(sel_Property_Base_Reference)->size() > 1 implies
    self.sel_Property_Base_Reference->forAll(s.oclIsTypeOf(ct_Property_Base_Reference))
```

5 Statecharts

Aus den Versuchen UML-Modelle als Spezifikationswerkzeuge zu nutzen, ist bekannt, dass Statecharts gut geeignet sind, bestimmte Teile von Widgets visuell zu spezifizieren. Genauer gesagt, handelt es sich bei diesem Teil um die Transitionen. Dabei ist wichtig anzumerken, dass in jedem Statediagramm immer nur die Transitionen eines Widgets betrachtet werden.

Transitionen bestehen in IML aus Zustandsübergängen, die durch ein Ereignis (InMessage) ausgelöst werden, die Aktionen (Transitionfunction) oder weitere Ereignisse (OutMessage) auslösen und durch Bedingungen in ihrer Durchführung eingeschränkt werden können. Alle diese Eigenschaften besitzen Transitionen in etablierten Statechartmodellen ebenfalls. Ihnen fehlt jedoch die Eigenschaft des "Rank", der Wichtung von Transitionen. Diese wird in dieser speziellen Form der Statecharts hinzugefügt. Auf der anderen Seite werden einige Eigenschaften, die zum Beispiel UML-Statecharts bieten, entfernt. Dazu gehören vor allem Parallelitäten und Hierarchien von Zuständen. Eine weitere Erweiterung gegenüber UML-Statecharts sind die Any-States, eine besondere Form von Pseudo-States, auf die später noch genauer eingegangen wird.

5.1 Die Bedeutung der Elemente

Die offensichtlichsten Elemente eines Statechartmodells sind Zustände und Transitionen. Eine Transition beginnt immer an einem Zustand und endet immer an einem Zustand. Dabei kann sie am gleichen Zustand beginnen und enden. Transitionen haben immer einen Rang, durch den sie geordnet werden können und einen Namen. Der Zwang einen Namen für eine Transition angeben zu müssen, wurde aus Kompatibilitätsgründen aus IML übernommen. Da in IML nicht gefordert wird, dass die Namen der Transitionen eindeutig sind, ist es möglich, bei der Erzeugung der Statecharts den Wert einer anderen Eigenschaft zu übernehmen.

Im Statechartmodell gibt es nur eine Klasse *State*, die jedoch unterschiedliche Arten von Zuständen umfassen. Welchen besonderen Zustand eine Instanz von *State* verkörpert, ist an dem Attribut *Type* zu erkennen. Type stellt eine Enumeration dar, deren Mitglieder in der unteren Grafik aufgeführt werden. Die wichtigsten sind st_State, st_Final und st_Initial. St_State repräsentiert die Zustände, in denen sich die Statemachine aufhalten kann, bis ein Event eintrifft. Der Initial-State wird aktiviert, wenn die Statemachine geladen wird. Er wird ohne ein Event verlassen. Der Final-State wird aktiviert, wenn die Statemachine entladen wird. Als Besonderheit in diesem speziellen Statechartmodell können auch Final-States nur durch Transitionen ohne Event erreicht werden. Da es einem Widget nicht möglich ist, sich selber zu deaktivieren, ist diese Möglichkeit auch in den Statecharts entfernt worden.

Ein anderer Typ von Zuständen ist der History-State (*st_History*), der den Initial-State ersetzen kann und den aktiven Zustand beim Entladen der Statemachine speichert. Außerdem existieren noch der Pseudo-State (*st_Pseudo*) und Zustände des Typs *st_Any*. Der Pseudo-State soll das Verständnis eines Statediagramms erleichtern. Er kann Transitionen, die auf das gleiche Event reagieren, zusammenfassen, auch wenn sie in unterschiedlichen Zuständen enden.

Dazu führt eine Transition mit dem gewünschten Event zum Pseudo-State und mehrere Transitionen ohne Event, aber mit Bedingungen, führen zu den Zuständen. In IML gibt es diese Art der Zusammenfassung für Transitionen nicht. Bei der Überführung zu IML müssen das Event der Transition zum Pseudo-State und die Informationen der Transitionen, die den Pseudo-State verlassen, vereinigt werden. Die Umwandlung des Statechartmodells zu IML ist in dieser Hinsicht also relativ leicht zu lösen. Die entgegengesetzte Richtung ist ohne weitere, zwischengespeicherten Informationen nicht zu erreichen.

Der Any-State ist auch eine Besonderheit dieses Statechartmodells. In IML-Modellen kommt es häufig vor, dass Transitionen unabhängig von dem aktiven Zustand ausgeführt werden. Um diese Transitionen nicht durch Transitionen ausgehend von jedem Zustand darstellen zu müssen, wurde der Any-State eingeführt. Eine Transition, die von einem Any-State ausgeht, wird für das aufgeführte Event ausgeführt, unabhängig welcher Zustand gerade aktive ist. Wenn eine Transition aus einem Any-State endet, bleibt der aktive Zustand aktiv. Dabei können nur Transitionen, die an einem Any-State beginnen, auch dort enden.

Die Elemente *TriggerEvent*, *SendEvent*, und *Action* sind die Ankerpunkte für Verweise auf die entsprechenden Elemente in IML. Die *Condition* ist kein Verweis, aber sie ist genau so strukturiert, wie ein el_Condition in IML.

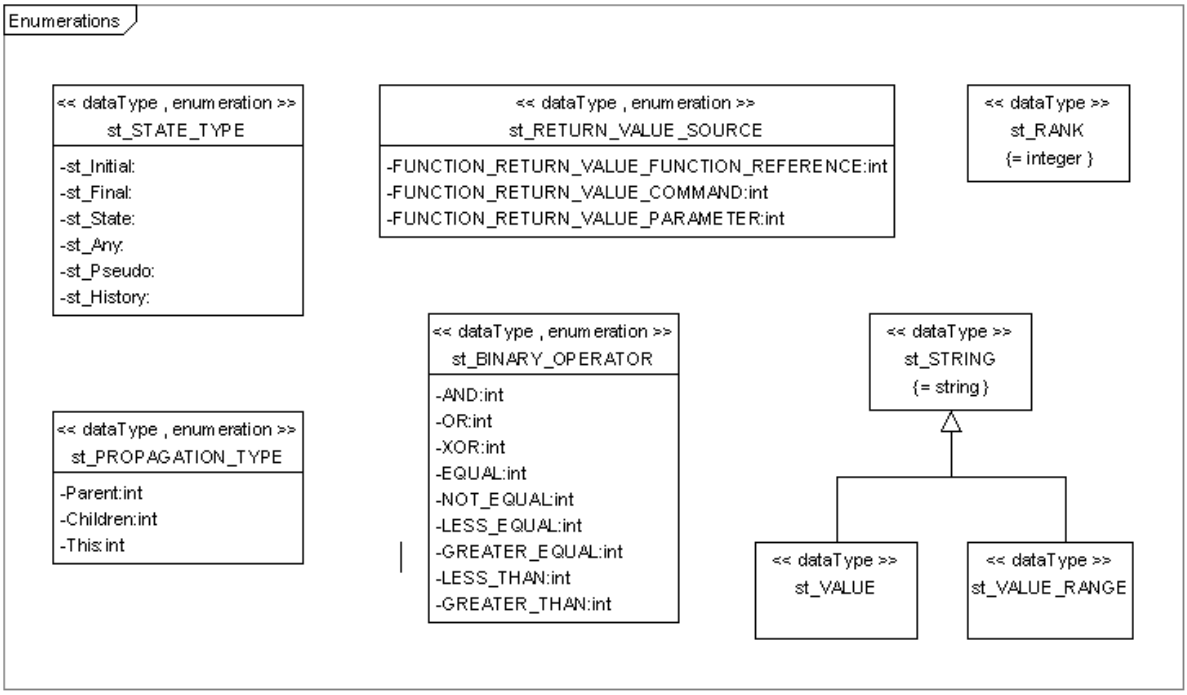

Abbildung 39 - Enumerationen und Datentypen

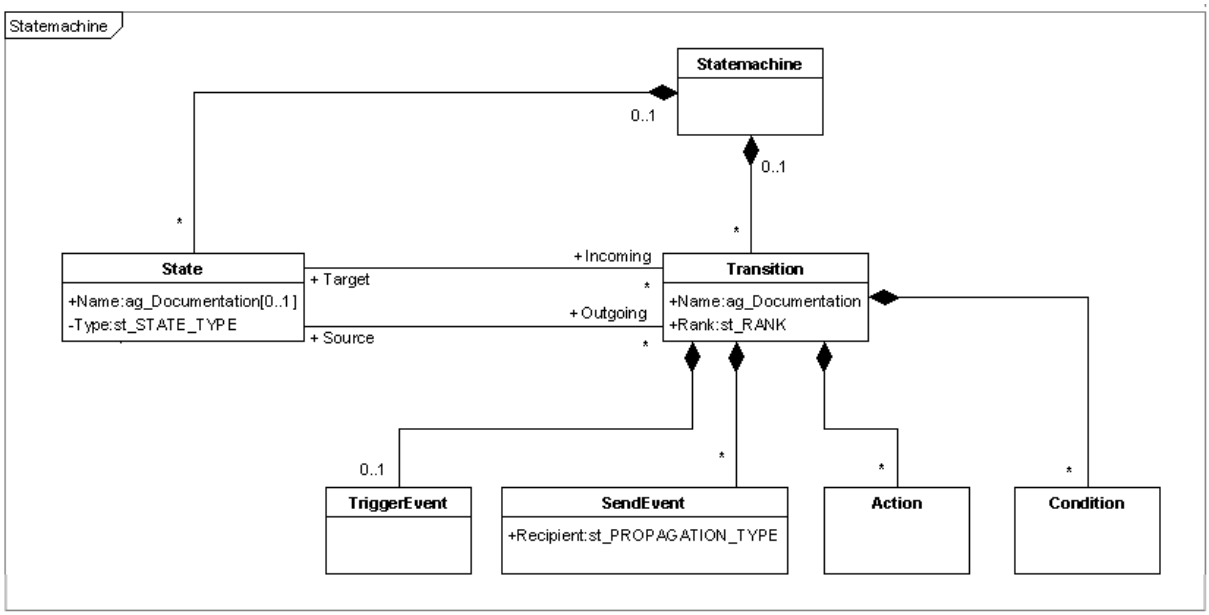

Abbildung 40 - Das Statechartmodell

Constraints:

1. Final-States haben mindestens eine eingehende Transitionen und keine ausgehenden.

   context State **inv**:
   self.Type ="st_Final" implies (self.Incoming->notEmpty() and self.Outgoing->isEmpty())

2. Transitionen, die in Final-States enden, haben kein Triggerevent

   context State **inv**:
   self.Type ="st_Final" implies (self.Incoming.TriggerEvent->notEmpty())

3. Initial-State besitzen mindestens eine ausgehende Transition und keine eingehenden.

   context State **inv**:
   self.Type = "st_Initial" implies (self.Outgoing->notEmpty() and self.Incoming->isEmpty())

4. Transitionen, die von Initial-States ausgehen, haben keine expliziten TriggerEvents.

   context State **inv**:
   self.Type = "st_Initial" implies self.Outgoing.forAll(t : Transition| t.TriggerEvent->isEmpty())

5. History-State besitzen mindestens eine ausgehende Transition und keine eingehenden.

   context State **inv**:
   self.Type = "st_History" implies (self.Outgoing->notEmpty() and self.Incoming->isEmpty())

6. Transitionen, die von History-States ausgehen, haben keine expliziten TriggerEvents.

   context State **inv**:
   self.Type = "st_History" implies self.Outgoing.forAll(t : Transition| t.TriggerEvent->isEmpty())

7. Es existiert entweder ein Initial-State oder ein History-State

   context Statemachine **inv**:

self->State->select(s:State| s.Type = "st_History")->notEmpty implies
self->State->select(s:State| s.Type = "st_Initial")->isEmpty and
self->State->select(s:State| s.Type = "st_Initial")->notEmpty implies
self->State->select(s:State| s.Type = "st_History")->isEmpty

8. Transitionen, die an Pseudo-States enden, haben immer ein TriggerEvent

context State **inv**:
self.Type ="st_Pseudo" implies (self.Incoming->TriggerEvent->notEmpty())

9. Transitionen, die an Pseudo-States beginnen, haben kein TriggerEvent

context State **inv**:
self.Type ="st_Pseudo" implies (self.Outgoing->TriggerEvent->isEmpty())

10. Transitionen, die an Any-States enden, beginnen auch an einem Any-State

context Transition **inv**:
self.Target.Type ="st_Any" implies (self.Source.Type="st_Any")

5.2 Einbettung des Statechartmodells in IML

Um die Verbindung zwischen den Objekten des Statechartmodells in dem Modell von IML klar zu machen, sind die Objekte des IML-Modells grau dargestellt. Wenn das Statechartmodell in das IML-Modell eingebunden wird, entspricht das *Statemachine* Objekt der Sammlung von *el_Transition*, die in IML durch den StateTable-Tag dargestellt wird. (siehe Abbildung 41) Die Statemachine enthält States, die zum Teil den Zuständen des Widgets entsprechen. States, die keine Verbindung zu *el_State_Definition* besitzen, sind zum Beispiel der Initial-State und der Final-State.

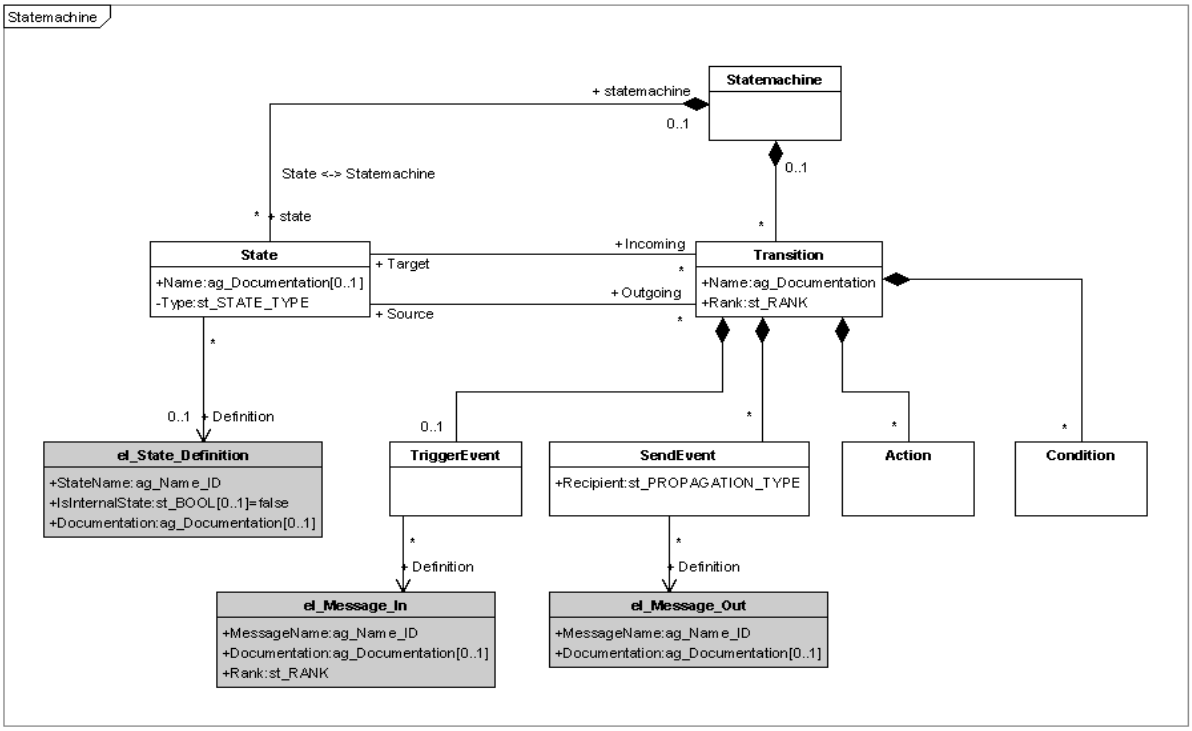

Abbildung 41 - Das Statechartmodell mit den Verbindungen zu IML

Die im Statechartmodell verwendeten Triggerevents verweisen auf die ausgewählten InMessages des Widgets und die SendEvents verweisen auf die OutMessages. In- und OutMessages wurden nicht in das eigentliche Statechartmodell übernommen, da sich diese Darstellungsform nicht ausgesprochen gut dafür eignet sie zu definieren oder aus der Liste der globalen Nachrichten auszuwählen. Für diese Arbeiten wäre es besser, eine eigene, auf sie abgestimmte Darstellungsform zu finden. Diese könnte dann, wie das Statechartmodell es für die Transitionen tut, Teile von IML ersetzen oder erweitern.

Alle States, die den StateType *st_State* besitzen, beziehen sich auf einen Zustand, der im Widget definiert worden ist.

Bei den Funktionen (*Action*), die durch Transitionen ausgeführt werden können, verhält es sich wie bei den Nachrichten. Sie verweisen auf die Funktionen, die für das Widget aus den globalen Funktionen ausgewählt wurden. In ihrem Aufbau gleicht eine Action einer FunctionReference.

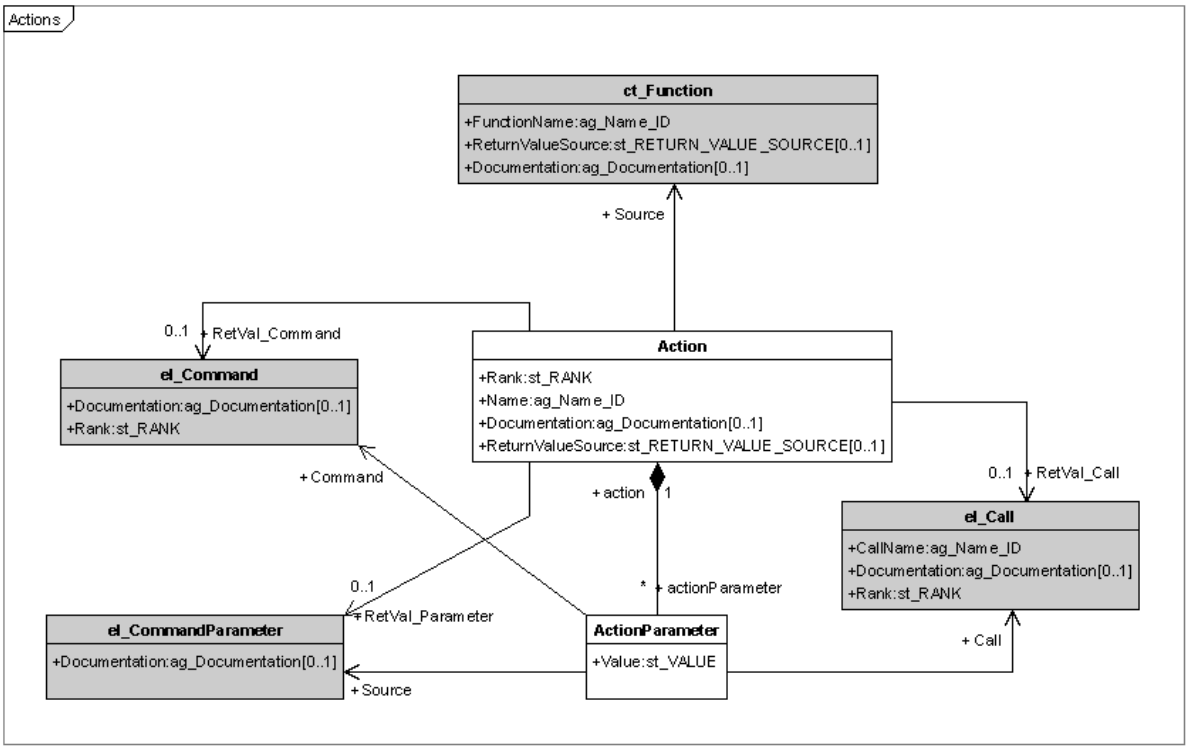

Abbildung 42 - Die Action und ihre Verbindungen zu IML

Der Aufbau der Bedingungen im Statechartmodell entspricht dem Aufbau der Bedingungen in IML. An beiden Stellen wäre es wünschenswert, eine eigene Methode für ihre Definition zu besitzen.

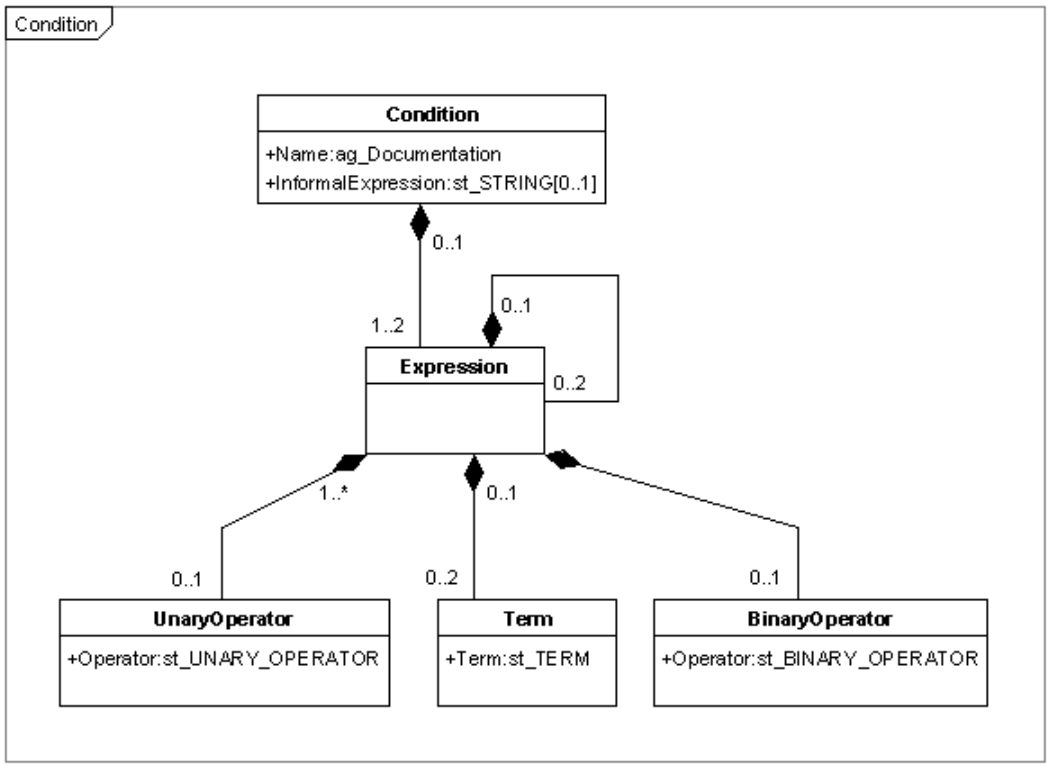

Abbildung 43 - Bedingungen im Statechartmodell

Die hauptsächliche Verbindung zwischen dem IML-Modell und dem Statechartmodell befindet sich im Behavior. Wenn die Transitionen eines Widgets mit Hilfe der Statecharts definiert oder geändert werden sollen, werden die el_Transitions von el_InOut entfernt und stattdessen die Statemachine eingefügt. Dabei werden aller Informationen und Verbindungen für die Statemachine aus den Informationen des Widgets extrahiert. Wenn alle Änderungen abgeschlossen sind, werden die Informationen zurückgewandelt und der Statemachine-Knoten entfernt.

Um die Statecharts sinnvoll darstellen zu können, benötigen diese grafische Informationen, die nicht in IML gespeichert werden. Diese sind auch im Statechartmodell nicht berücksichtigt worden, da gezeigt werden sollte, dass alle für IML relevanten Daten hin- und zurück konvertiert werden können. Damit das Layout eines Statediagramms nicht jedes Mal neu erzeugt werden muss, sollten die grafischen Informationen über das Diagramm parallel gespeichert werden. Bei Elementen, die in IML eingefügt wurden und die noch keine grafischen Informationen enthalten, sollten vernünftige Default-Werte verwendet werden.

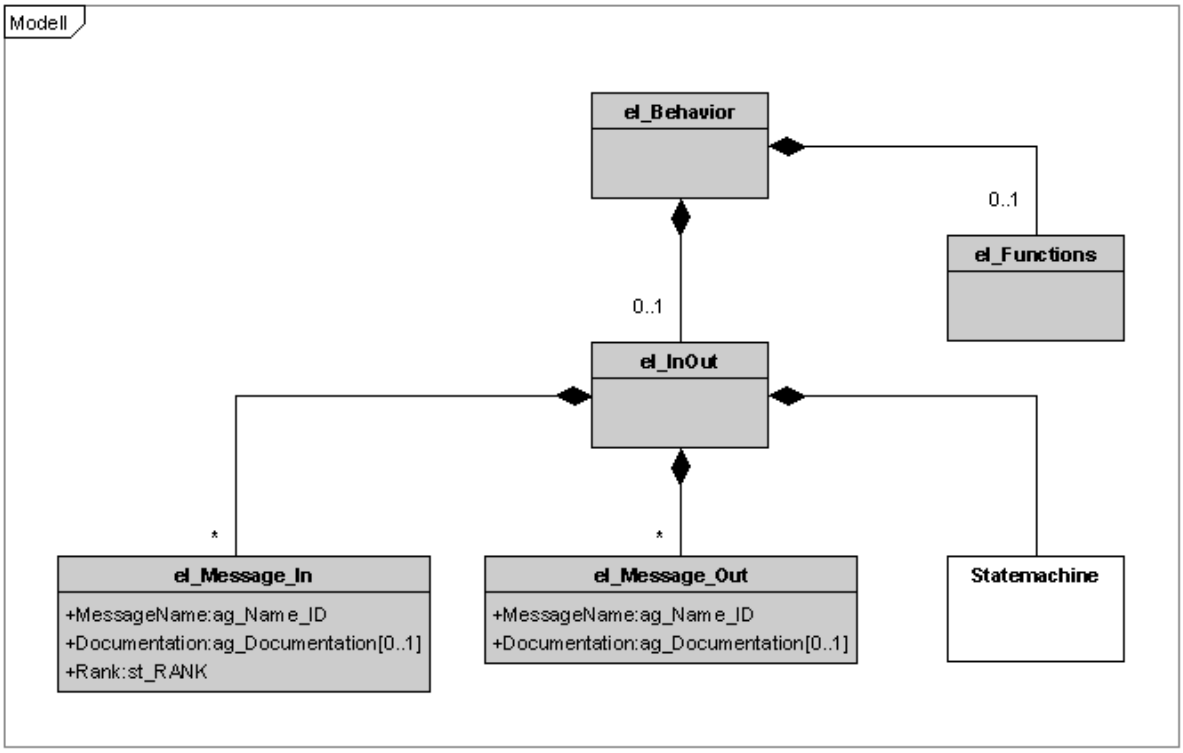

Abbildung 44 - Position des Statechartmodells in IML

5.3 Auflösung von Mehrdeutigkeiten

In allen Statecharts besteht die Möglichkeit, dass zu einem bestimmten Zeitpunkt mehr als eine Transition auf ein Ereignis reagieren kann. Die Entscheidung, welche Transition zum Beispiel bei einer Simulation tatsächlich ausgeführt wird, muss von der Simulationskomponente entschieden werden. Oft werden die Ebenen der Start- und Zielzustände ermittelt und mit ihrer Hilfe eine Reihenfolge festgelegt. Um die Reihenfolge von Transitionen auf gleicher Ebene zu ermitteln, werden zum Beispiel grafische Eigenschaften, wie die Position der Transition auf einem gedachten Zifferblatt, verwendet. [Cra 1]

In IML gibt es nur eine gemeinsame Ebene für alle Zustände, dass heißt diese Eigenschaft steht zur Festlegung von Reihenfolgen nicht zur Verfügung. Auch eine Priorisierung anhand ihrer Lage im oder gegen den Uhrzeigersinn ist problematisch. Die Systementwickler, die mit Hilfe der Statecharts das Verhalten des Infotainmentsystems spezifizieren sollen, sind nur zu einem geringen Anteil mit Softwareentwicklung vertraut. Die Übersichtlichkeit oder Ästhetik der Diagramme könnten so leicht zu Entscheidungen führen, die gleichzeitig die Bedienbarkeit des Systems bestimmen.

Alle Bedienabläufe in einem Infotainment-System werden nach Kriterien des Wiedererkennens und der Handlungspsychologie festgelegt. Dabei sind indirekte Priorisierungen von Transitionen durch ihre Zeichenreihenfolge zu ungenau. Deshalb wird in diesem angepassten Statechartmodell die Priorisierung durch das Attribut "Rank" explizit vorgenommen. Alle Transitionen, die auf das gleiche Ereignis reagieren, müssen unterschiedlich Werte im Attribut "Rank" besitzen (Constraint 1).

Das "Rank" Attribut ist nicht die einzige Abwandlung der klassischen Statecharts. Das hier beschriebene Statechartmodell ist an die Verwendung im Zusammenspiel mit IML angepasst. Dabei geht es vor allem um die Verwendung der in IML beschriebenen Funktionen und Nachrichten. Sie werden in IML für das Gesamtsystem definiert und in den einzelnen Widgets nur verwendet. Da jedes Statediagramm nur jeweils ein Widget enthält, würde dort definierten Funktionen und Nachrichten diese globale Bedeutung verloren gehen. Da jedoch die Definition von Nachrichten und Funktionen in Statecharts in der Regel nur sehr ungenau vorgenommen wird, ist dies keine sehr große Einschränkung. Um das Statechartmodell eigenständig verwenden zu können, würde es genügen, den Trigger- und SendEvents einen eigenen Namen zu geben. Dann könnten alle Informationen, die ein klassisches Statediagramm enthält, übernommen werden.

Constraints:

1. Alle Transitionen, die auf das gleiche Ereignis reagieren, haben einen unterschiedlichen Rang.

```
context Statemachine inv:
self.Transition->forAll(t1:Transition| self.Transition->forAll(t2:Transition|
    t1.TriggerEvent.Definition = t2.TriggerEvent.Definition and t1<>t2
    implies t1.Rank <>t2.Rank))
```

5.4 Vom Statechartmodell zu XML

In diesem Abschnitt soll anhand eines Beispiels gezeigt werden, wie ein Statechartmodell nach der Umwandlung in ein IML-Modell und schließlich in XML aussehen würde. Als Beispiel dient auch hier das in Kapitel 2.7 beschriebene System. Es besteht aus vier Menüs, die in einem Menü-Netz zusammengefasst sind. Eines dieser Menüs ist der CD-Spieler (PCM_MEDIA_MAIN). Das Menü hat sieben verschiedene Zustände, in denen jeweils eine der möglichen Aktionen ausgewählt ist. Die Aktionen für "Play" und "Pause" hängen weiterhin davon ab, ob die CD gerade gespielt wird oder nicht. Deshalb existieren für sie jeweils zwei Zustände.

Abbildung 45 - PCM_MEDIA_MAIN

Die untere Grafik zeigt das Statediagramm, welches das Verhalten der Menüs "PCM_MEDIA_MAIN" beschreibt. Das Diagramm enthält keinen Final-State, da das Menü keine Aktionen ausführen soll, wenn es entladen wird. Es enthält allerdings einen Any-State. Da das Verhalten für des Eintreffen der Nachrichten SE_Muted und SE_UnMuted immer gleich sind, müssen sie so nicht für jeden Zustand beschrieben werden. Der doppelte Aufruf von PauseTrack muss in der Applikation abgefangen werden. Der Zustand Paus_While_Paused verhindert zwar einen solchen doppelten Aufruf, aber sein eigentlicher Zweck ist die Visualisierung des Zustandes des Players für den Benutzer.

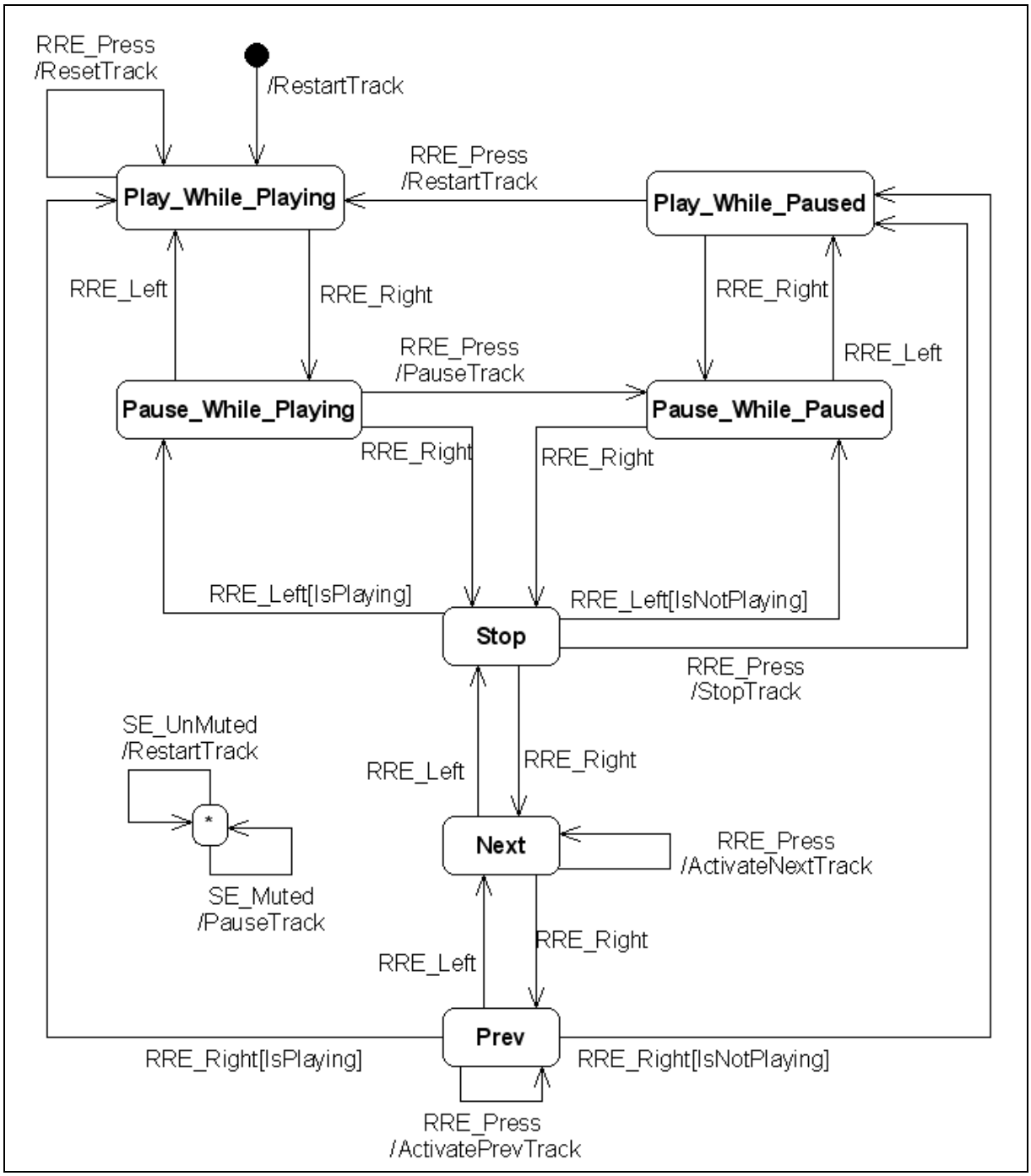

Abbildung 46 - Das Statediagramm für PCM_MEDIA_MAIN

Der Right Rotary Encoder (RRE) ist der rechte von zwei Knöpfen, die sich sowohl drehen als auch drücken lassen. Der linke dieser Zwei ist der Left Rotary Encoder (LRE). Sie können

jeweils die Ereignisse Left (drehen gegen den Uhrzeigersinn), Right (drehen mit dem Uhrzeigersinn) und Press (drücken) auslösen. SE_Muted und SE_UnMuted werden ausgelöst, wenn sich die Lautstärke von Eins auf Null verringert bzw. wenn sie sich von Null auf Eins erhöht. Dies ist im Statediagramm des Menü-Netzes beschrieben.

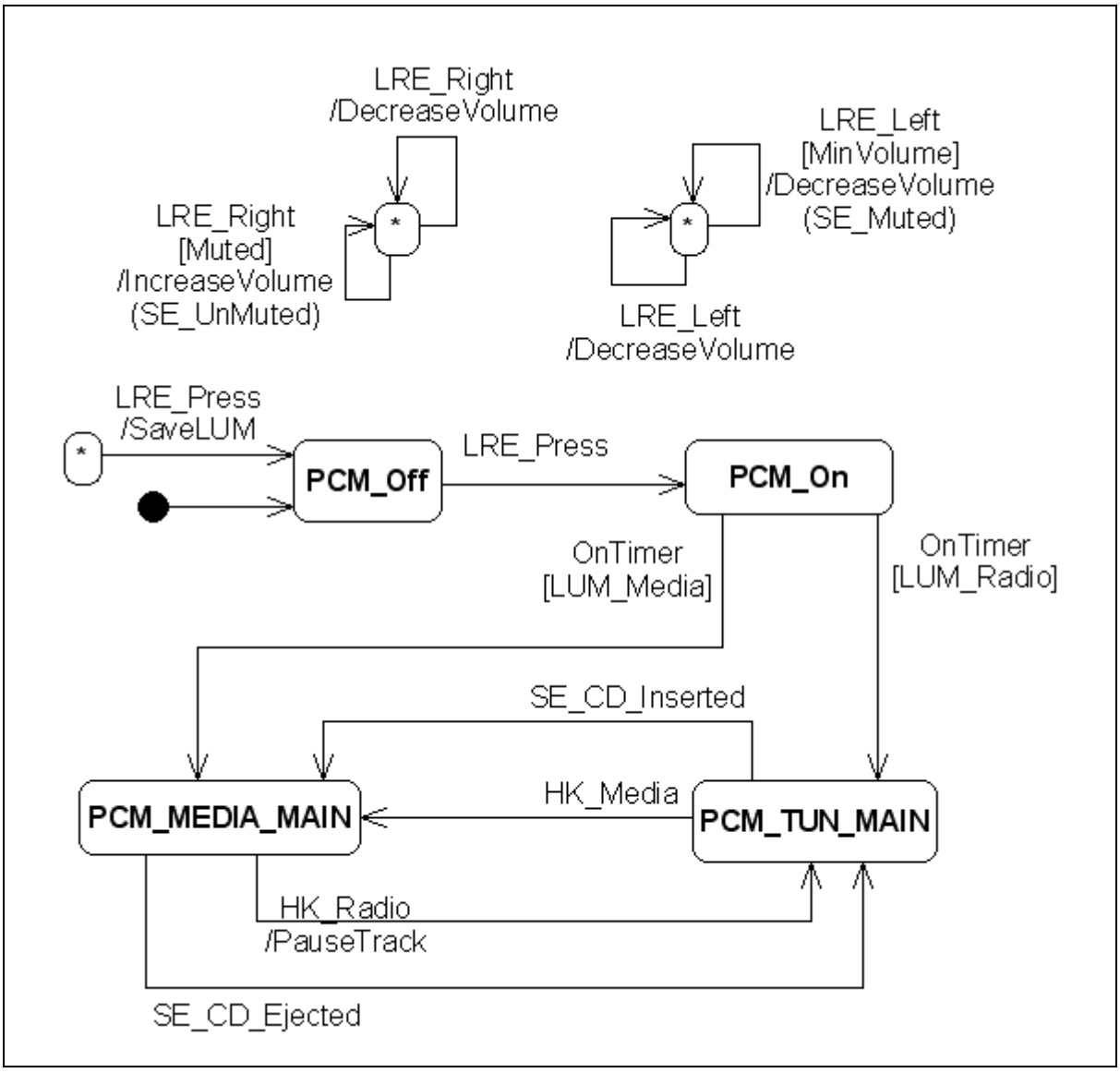

Abbildung 47 - Das Statediagramm des Menünetzes

Das Menü-Netz enthält für jedes Menü einen State. Das Statediagramm des Netzes spezifiziert welches Menü angezeigt werden soll. Bei der Lautstärkeregelung wird davon ausgegangen, dass es keine Rolle spielt, welches Menü angezeigt wird. In den Menüs "PCM_Off" und "PCM_On" werden die Auswirkungen allerdings nicht erkennbar sein. Ein solches System wäre nicht für den Einsatz in Fahrzeugen geeignet. Der Fahrer wäre dabei in der Lage die Lautstärke zu erhöhen während nichts zu hören ist. Beim erneuten Einschalten bestünde dann das Risiko, dass der Fahrer sich erschreckt und einen Unfall verursacht. Um das Beispiel überschaubar zu halten, wird diese Möglichkeit außer Acht gelassen.

Das Instanzenmodell für das Statediagramm des Netzes ist zu unübersichtlich, um es als Beispiel verwenden zu können. Deshalb wurden zwei Transitionen aus dem Diagramm ausgewählt, für die das Instanzendiagramm gezeigt wird. Die erste Transition bestimmt den Wech-

sel von "PCM_MEDIA_MAIN" zu "PCM_TUN_MAIN", wenn der HardKey Radio betätigt wird. Während dieser Transition wird die Funktion PauseTrack ausgeführt und auch hier spielt es keine Rolle, ob die CD gerade spielt oder nicht.

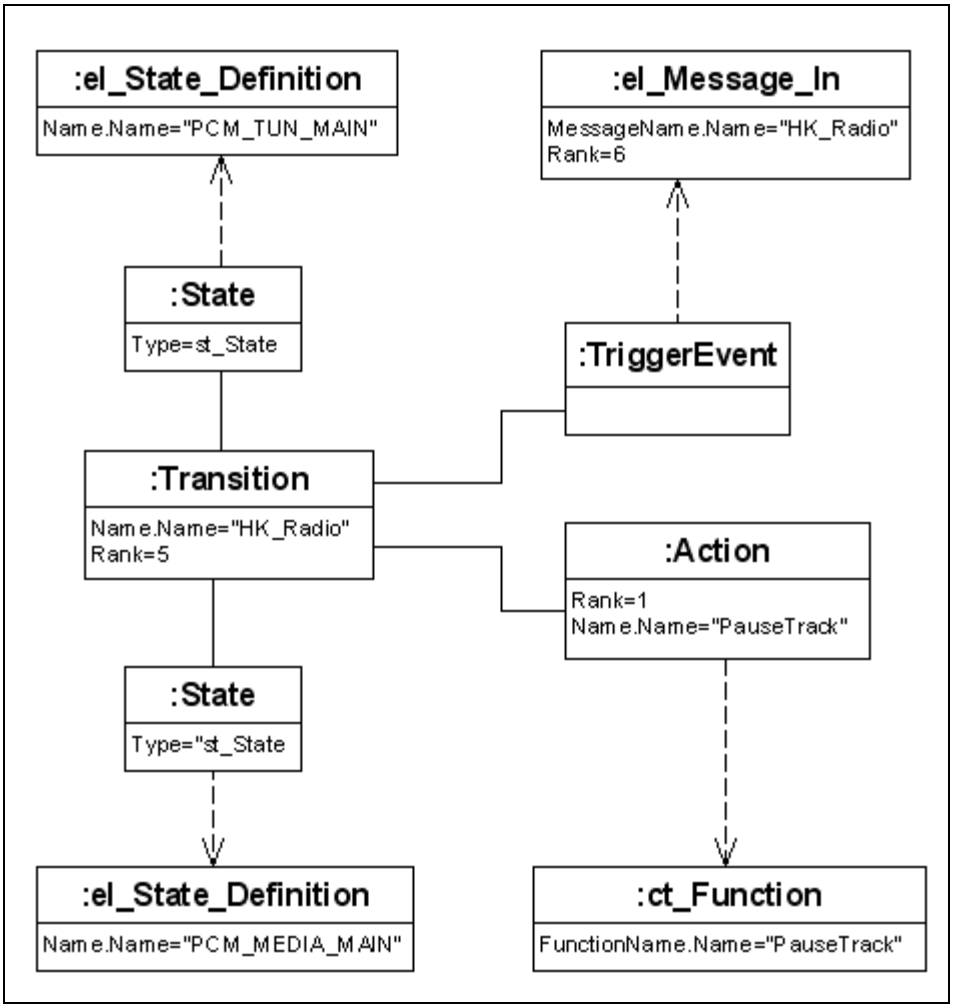

Abbildung 48 - Das Instanzendiagramm für die Transition HK_Radio

Die zweite Transition geht vom rechten Any-State aus und enthält die Bedingung "MinVolume". In dieser Transition sind alle möglichen Elemente vertreten, die eine Transition enthalten kann.

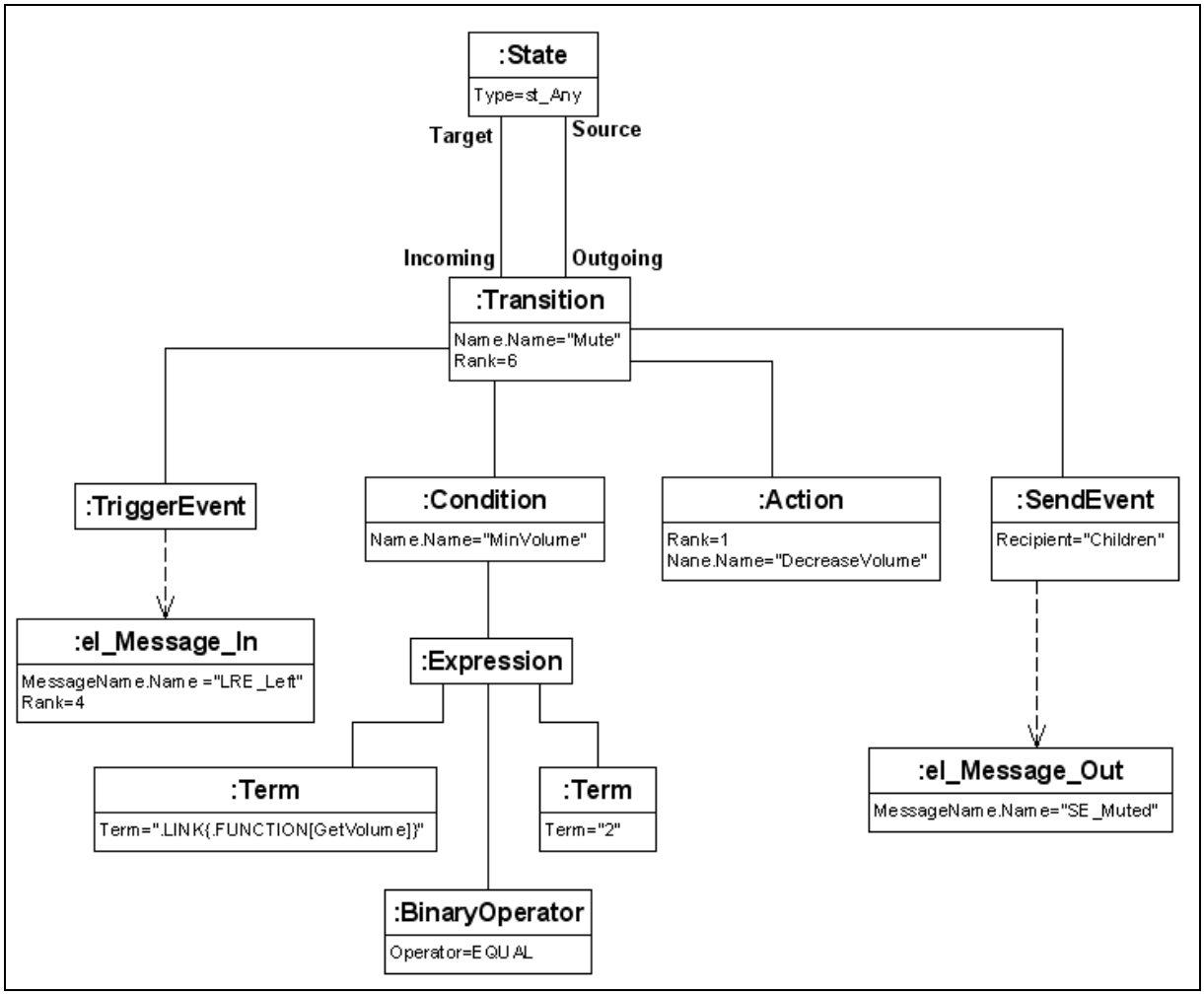

Abbildung 49 - Das Instanzendiagramm für die Transition Mute

Die entsprechenden Instanzendiagramme für diese Transitionen sehen im IML-Modell dann wie folgt aus.

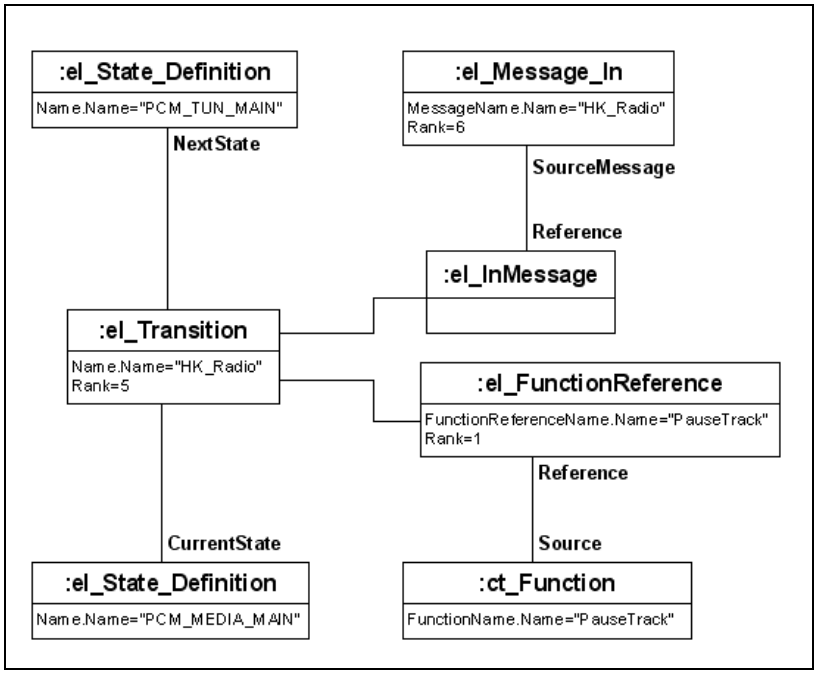

Abbildung 50 - IML Instanzdiagramm für Transition HK_Radio

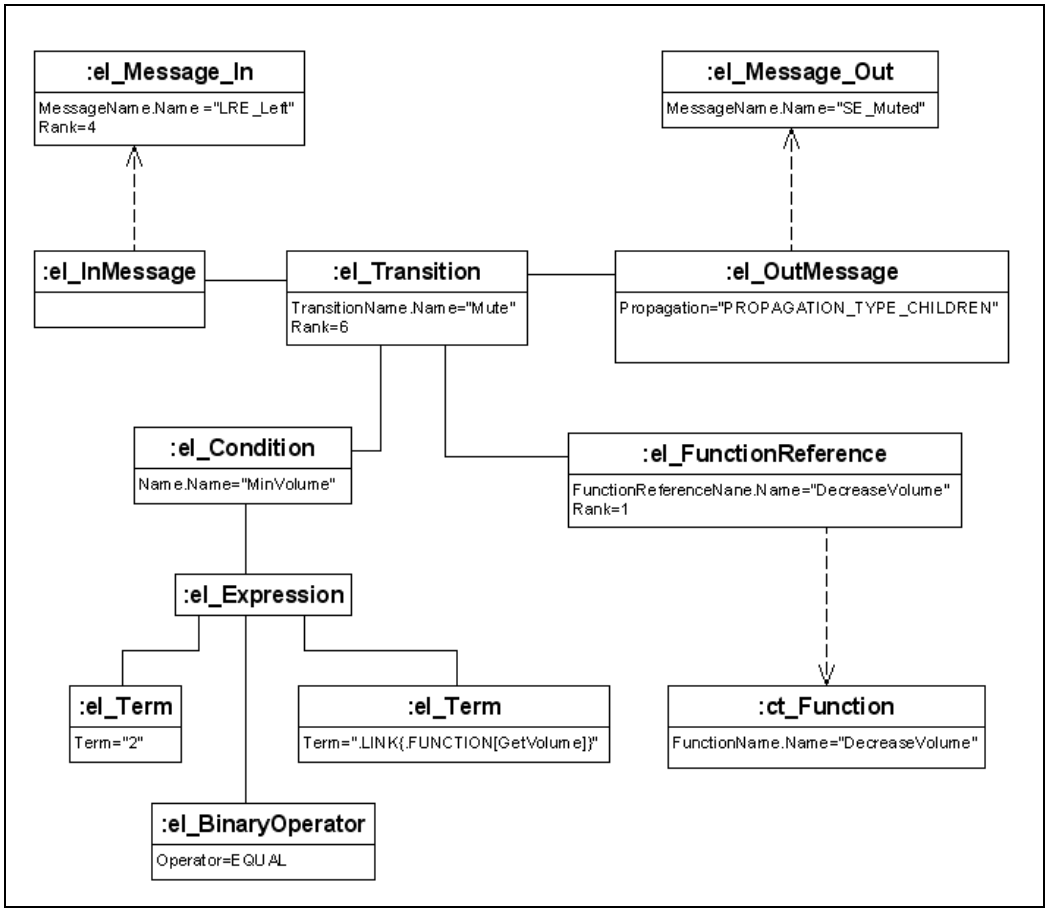

Abbildung 51 - IML Instanzdiagramm für Transition Mute

In der XML-Darstellung des Menü-Netzes sind nur die relevanten Teile ausführlich dargestellt, alle anderen sind nur angedeutet. Alle Elemente, die in den Instanzdiagrammen zu sehen sind, außer der Definition der States, sind noch enthalten.

```
<Widget Name="MenuNet_General" Type="WIDGET_TYPE_NORMAL">
    [...]
    <Implementation>
        <Properties>
            <StateProperty>
                <State Name="PCM_OFF"/>
            </StateProperty>
            [...]
        </Properties>
        [...]
        <Behavior>
            <Functions>
                [...]
                <Function Name="DecreaseVolume">
                    <Call Name="Sound.IncreaseVolume">
                        <API Type="API_TYPE_FUNCTION" Target="ABSTRACTION_LAYER"/>
                        <Commands>
                            <Command Name="Sound.IncreaseVolume" Type="COMMAND_TYPE_SYNCHRON"/>
                        </Commands>
                    </Call>
                </Function>
                [...]
                <Function Name="PauseTrack">
                    <Call Name="Audio.PauseTrack">
                        <API Type="API_TYPE_FUNCTION" Target="ABSTRACTION_LAYER"/>
                        <Commands>
                            <Command Name="Audio.PauseTrack" Type="COMMAND_TYPE_SYNCHRON"/>
                        </Commands>
                    </Call>
                </Function>
                [...]
            </Functions>
            <InOut>
                <InMessages>
                    [...]
                    <Message Name="LRE_Left" Type="MESSAGE_TYPE_NORMAL"
                            MessageID=".DICT{msg_HARDKEY_EVENT}">
                        <Parameters>
                            <ParameterInMessage Key="Event" ValueType="DATA_TYPE_STRING">
                                <Value Value=".DICT{LRE_Left}"/>
                            </ParameterInMessage>
                        </Parameters>
                    </Message>
                    [...]
                    <Message Name="HK_Radio" Type="MESSAGE_TYPE_NORMAL"
                            MessageID=".DICT{msg_HARDKEY_EVENT}">
                        <Parameters>
                            <ParameterInMessage Key="Event" ValueType="DATA_TYPE_STRING">
                                <Value Value=".DICT{HardKey_Radio}"/>
                            </ParameterInMessage>
                        </Parameters>
                    </Message>
                    [...]
                </InMessages>
                <OutMessages>
                    <Message Name="SE_Muted" Type="MESSAGE_TYPE_NORMAL"
                            MessageID=".DICT{msg_SYSTEM_EVENT}">
                        <Parameters>
                            <ParameterOutMessage Key="Event" ValueType="DATA_TYPE_STRING">
                                <Value Value=".DICT{SYS_MUTE_ON}"/>
                            </ParameterOutMessage>
                        </Parameters>
                    </Message>
                    [...]
                </OutMessages>
                <StateTable>
                    <Transitions>
                        [...]
                        <Transition Name="LRE_Left_MinVolume">
                            <InMessage MessageNameReference="LRE_Left"/>
                            <Conditions>
```

```
                                <Condition Name="MinVolume">
                                    <Expression>
                                        <Term Term=".LINK{.FUNCTION[GetVolume]}"/>
                                        <BinaryOperator Operator="EQUAL"/>
                                        <Term Term="1"/>
                                    </Expression>
                                </Condition>
                            </Conditions>
                            <OutMessages>
                                <OutMessage MessageNameReference="SE_Muted"
                                        Propagation="PROPAGATION_TYPE_CHILDREN"/>
                            </OutMessages>
                            <TransitionFunctions>
                                <FunctionReference Name="DecreaseVolume" FunctionName="DecreaseVolume"/>
                            </TransitionFunctions>
                        </Transition>
                        [...]
                        <Transition Name="HK_Radio" CurrentState="PCM_MEDIA_MAIN" NextState="PCM_TUN_MAIN">
                            <InMessage MessageNameReference="HK_Radio"/>
                            <TransitionFunctions>
                                <FunctionReference Name="PauseTrack" FunctionName="PauseTrack"/>
                            </TransitionFunctions>
                        </Transition>
                        [...]
                    </Transitions>
                </StateTable>
            </InOut>
        </Behavior>
    </Implementation>
</Widget>
```

Wie aus dem Beispiel ersichtlich wird, hat der Initial-State oder die Transition, die von ihm ausgeht, eine besondere Auswirkung auf das XML. Die Festlegung des ersten Zustandes wird im StateProperty abgelegt. Dabei werden auch die Bedingungen und das Rank Attribut beachtet, wenn mehrere Transitionen vom Initial-State ausgehen. Die Funktionen, die ausgeführt werden sollen, und die Nachrichten, die versendet werden sollen, werden in einer Transition in der StateTable abgelegt, der als eingehende Nachricht das Event "OnLoad" dient. Dieses Event wird an alle Widgets gesendet, wenn sie gerade geladen worden sind. Um die doppelte Verwendung der Bedingungen und die aufwendige Zuordnung von StateProperty-Ausdrücken und Transitionen, die auf OnLoad reagieren, zu vermeiden, könnten auch alle Informationen in den Transitionen abgelegt werden. Dann müsste aber gesichert sein, dass das StateProperty nicht existiert und das HistoryProperty muss den Wert "false" enthalten.

Transitionen, die zum Final-State führen, haben immer das Event "OnUnload", das ebenfalls vom System an jedes Widget gesendet wird, um ihm mitzuteilen, dass es entladen wird.

Um ein Statechartmodell in ein IML-Modell umzuwandeln sind folgende Überführungen nötig:

Statemachine: Das Statemachine Objekt im Statechartmodell hat keine Entsprechung im IML-Modell. Es dient nur zur Bündelung der States und Transitions des Statechartmodells. Es wird nach der erfolgten Umwandlung entfern und seine Verbindung zu el_InOut wird aufgehoben.

Transition: Eine Transition entspricht einem el_Transition in IML. Eine Ausnahme dazu sind Transitionen, die vom Initial-State ausgehen. Teile ihrer Information werden zu StateProperty.

State: Da alle el_State_Definition, auf die States im Statechartmodell verweisen können, bereits existieren müssen, findet an ihnen keine Änderung statt. Allerdings werden die referenzierten States bei el_Transition zu CurrentState und NextState. Bei der umgekehrten Umwandlungsrichtung, kann es vorkommen, dass ein neu erzeugter State noch keine Angaben zu seiner grafischen Repräsentation besitzt. Es müssen dann klug gewählte Standartwerte verwendet werden.

Action: Eine Action entspricht einer el_FunctionReference. Alle Attribute werden in die der FunctionReference, die gleich benannt sind, überführt. Ebenso werden die Verbindungen zu anderen Objekten im IML-Modell übernommen.

ActionParameter: Der Parameter der Action hat seine Entsprechung im IML-Objekt el_CommandParameter. Die Verbindungen zu IML-Objekten werden übernommen, ausgenommen die Verbindung zu Action, die auf die entsprechende el_FunctionReference umgeleitet wird.

TriggerEvent: Die el_InMessage übernimmt die Verbindungen vom TriggerEvent. Die Verbindung zum Objekt Transition wird auf el_Transition umgelenkt.

SendEvent: Das SendEvent übergibt seine Attributinhalte und Verbindungen an el_OutMessage. Die Verbindung zum Objekt Transition wird auf el_Transition umgelenkt.

Condition: Der Aufbau der Bedingungen in den beiden Modellen ist gleich. Um die entsprechenden Objekte zu identifizieren, müssen bei den Namen nur die Präfixe entfernt oder hinzugefügt werden. (el_UnaryOperator <--> UnaryOperator)

6 Ausblick

Die heutigen Infotainmentsysteme sind bereits sehr komplex, die zukünftigen werden mit aller Wahrscheinlichkeit noch komplexer werden. IML soll nicht als Beschreibungssprache eines solchen Systems verwendet werden, um die Komplexität zu mindern oder zu verschleiern. Im Gegenteil, dem Entwickler soll die volle Komplexität seiner Aufgabe bewusst gemacht werden. Zu viele Fehler oder Inkonsistenzen in existierenden Systemen sind auf mehrdeutige Beschreibungen der Entwickler zurückzuführen, die nicht bedacht haben, dass auch andere Lösungen, als die in ihren Köpfen, möglich sind. Da die meisten Entwicklerteams schon Erfahrung mit früheren Systemen haben, sind sie in der Lage fehlende Informationen zu antizipieren. Natürlich ist das durch "Raten" entstandene Ergebnis nicht immer genau das, was der ursprüngliche Spezifikateur wollte. Eine vollständige Beschreibung, die zum gewünschten Ergebnis führt, muss ihn also zwingen alle Informationen, welche die anderen Entwickler von ihm benötigen, festzulegen.

Einerseits wird die Vollständigkeit der Spezifikation von den Entwicklern gewünscht, andererseits kann die Komplexität von IML leicht dazu führen, dass diese Art der Beschreibung abgelehnt wird, da sie zu schwierig ist. Um in diesem Zwiespalt Abhilfe zu schaffen, werden auf kleine Teilaspekte der Spezifikation angepasste Werkzeuge benötigt. Diese Werkzeuge sollten den Entwickler durch die Erstellung seines Spezifikationsteils führen. Dabei sollte bei ihrer Entwicklung darauf geachtet werden, bestehende Abläufe bei der Spezifikation möglichst beizubehalten, um den Entwicklern die Eingewöhnung so leicht wie möglich zu machen.

Außer dem Wissen über die Abläufe bei der Spezifikation bestimmter Teilbereiche benötigt man auch das Datenmodell für IML. IML wurde mit Blick auf die Darstellung in Texteditoren entwickelt. Die Werkzeuge, die die Arbeit wirklich erleichtern können, werden wahrscheinlich häufig auf grafischen Sprachen beruhen. Für sie ist es unbedeutend, ob Transitions als Unterknoten des Knotens "Transitions" gebündelt werden oder ob sie wild mit In- und OutMessages gemischt werden. Für einen Menschen, der die IML-Beschreibung eines Widgets lesen und dabei verstehen soll, wie es aussieht und sich verhält, spielt es jedoch eine große Rolle. Das in dieser Arbeit vorgestellte Modell für IML soll die Grundlage bieten, um angepasste Editoren für Teileaspekte von IML zu entwickeln. Mit der Beschreibung des Modells der Statecharts wurde versucht, einen dieser Teilaspekte hervorzuheben und an seinem Beispiel zu zeigen, wie das Datenmodell eines Editors aussehen könnte. Neben den Transitionen, die sich gut durch Statecharts darstellen lassen würden, gibt es noch viele andere Teile, die durch eine grafische Sprache leichter zu spezifizieren wären. Der Aufbau eines Widgets aus anderen Widgets könnte sicher durch ein Klassendiagramm verständlicher gemacht werden. Die Punktnotation würde wesendlich leichter zu erzeugen sein, wenn die Ziele der Links ausgewählt werden könnten. Die In- und OutMessages eines Widgets stellen sein Interface zur Kommunikation mit anderen Widgets dar, ihre grafische Darstellung würde es leichter machen zu erkennen, ob zwei Widget miteinander kommunizieren können. Es ist sicher auch möglich, eine Vereinfachung bei der Erstellung von Conditions zu finden.

All dies sind nur Möglichkeiten, die vor der Umsetzung genau geprüft werden müssen. Es gibt sicher noch andere Möglichkeiten die oben genannten oder andere Teile von IML zu spe-

zifizieren. Das vorgestellte Modell kann hoffentlich helfen, dass einige von ihnen realisiert werden. IML könnte dann das Austauschformat sein, in das die Werkzeuge ihre Daten exportieren und das zwischen den Entwicklerteams weiter gereicht wird.

I. Literaturverzeichnis

[Ang 1] C. Angelski, "Infotainment Markup Language – Allgemeiner Aufbau und Struktur", Auf Anfrage unter: http://www.teledrive.de/content/german/products/software/iml/main.html, [Stand Jan 2007].

[Bar 1] Thomas Barmeyer, "Beschreibungssprachen für HMIs von Infotainmentsystemen im Automobil", Diplomarbeit Technische Fachhochschule Berlin 2006.

[Cra 1] M. L. Crane, J. Dingel, "UML vs. Classical vs. Rhapsody Statecharts - Not All Models are Created Equal", Online unter: http://www.cs.queensu.ca/~stl/papers/MoDELS2005.pdf, [Stand Jan 2007].

[Har 1] D. Harel, "Statecharts, a visual formalism for complex systems", Science of Computer Programming 8, 1987, S. 231-274, Online unter: http://www.wisdom.weizmann.ac.il/~dharel/SCANNED.PAPERS/Statecharts.pdf, [Stand Jan 2007].

[IAV 1] G. Wegner, C. Angelski, P. Endt, "Das elektronische Lastenheft als Mittel zur Kostenreduktion bei der Emtwicklung der Mensch-Maschine-Schnittstellevon Infotainment-Systemen im Fahrzeug", aus C. Müller-Bagehl, P. Endt (Hrsg.), "Infotainment / Telematik im Fahrzeug – Trends für die Serienentwicklung", 2004, S 38 ff.

[Kreu 1] J. Kreuzinger, "Softwareentwicklung im Automobilbau – Anforderungen, Standarts und Vorbilder", VDI Berichte 1907, 2005, S. 69 ff.

[Nie 1] D. Niederkorn, J.Zehentbauer, "Steht die steigende Infotainment Funktionalität im Widerspruch zur Stabilität und Qualität?", VDI Berichte 1907, 2005, S 829 ff.

[OMG 1] OMG, "Meta Object Facility (MOF) Specification Version 1.4.1" Online unter: http://www.omg.org/docs/formal/05-05-05.pdf, [Stand Jan 2007].

[OMG 2] OMG, "Object Constraint Language OMG - Available Specification Version 2.0", Online unter: http://www.omg.org/docs/formal/06-05-01.pdf, [Stand Jan 2007].

[Vli 1] Eric von der Vlist, "XML Schema", O'REILLEY, 2003.

[W3C 1] W3C, "XML Schema 1.1 Part 1: Structures", W3C Working Draft 31 August 2006, Online unter: http://www.w3.org/TR/xmlschema11-1/, [Stand Jan 2007].

[M&K 1] H. Mössenböck, J.Kepler, " The Compiler Generator Coco/R - User Manual", 1990, Online unter: http://www.ssw.uni-linz.ac.at/Research /Projects/Coco/Doc/UserManual.pdf, [Stand Jan 2007].

II. Abbildungsverzeichnis