Constraint-basierte Feature-Modellierung

Entwicklung von Werkzeugen zur Unterstützung von produktlinienorientierter Softwareentwicklung

Bakkelaureatearbeit für die LVA Projektprakikum

Johannes Kepler Universtität Linz

Markus Gaisbauer

markus.gaisbauer@students.jku.at

Matr.Nr. 0256634

13. Oktober 2006

Kurzfassung

Die Entwicklung von Software-Produktlinien ist eine Methode des Software Engineerings zur effizienten Entwicklung großer Software-Systeme. Einzelne Produkte einer Produktlinie werden durch die Menge ihrer Features, das heißt die vom Benutzer sichtbaren Leistungen des Systems, charakterisiert. Feature-Modelle dienen dazu Gemeinsamkeiten und Unterschiede zwischen Mitgliedern der selben Produktlinie zu spezifizieren. Aus der abstrakten Beschreibung eines Produktes kann anschließend voll oder teils automatisiert konkrete Software erzeugt werden.
Feature-Modelle müssen sowohl mächtige genug sein, um komplexe Abhängigkeiten zwischen Features darzustellen, als auch einfach genug, um sowohl für Entwickler von Produktlinien, aber auch für Käufer und Verkäufer verständlich zu sein. Komplexe Abhängigkeiten werden erst nach und nach durch neue Ansätze zur Feature-Modellierung ermöglicht. Die ersten Feature-Modelle in FODA konnten nur sehr wenige Abhängigkeiten darstellen. Nachfolgende Modelle wie Cardinality-based Feature-Modelling erweiterten diese zwar schrittweise um neue Abhängigkeiten, ließen aber ein einheitliche Beschreibung dieser vermissen.
In dieser Arbeit wurde ein neuer Ansatz zur Feature-Modellierung entwickelt, der alle Abhängigkeiten zu explizit festgelegten Bestandteilen eines Modells macht. Alle Abhängigkeiten werden durch spezielle Constraints beschrieben. Durch die Erweiterbarkeit um neuen Spezialisierungen der vorhandenen Constraints kann das Modell ohne großen Aufwand an neue Anforderung angepasst werden.
Im Rahmen dieser Arbeit wurde ein XML-Schema für Constraint-basierte Feature-Modelle entwickelt. Zur Unterstützung des Modellierungs-Prozesses wurde ein XML-Editor um einige nützliche Features erweitert. Zusätzlich wurde ein Werkzeug für die Feature-Selektion entwickelt. Dieses unterstützt aufgrund von Constraint Propagation eine inkrementelle, halb-automatisierte Auswahl von Features.

Abstract

Product Line Engineering is a Software Engineering method for efficiently developing large software systems. Individual members of a product line are characterised by feature specifications. Feature models are used to specify commonalities and differences between products of the same product line. This abstract description of a product can be used to automatically generate concrete software products.
Feature models have to be expressive enough to model complex dependencies between features. On the other hand they have to be simple enough to be understood by engineers as well as sales people and eventually even clients. In the last years feature models have evolved to support increasingly complex dependencies between features. The first models in FODA only supported a very limited set of hierarchical constraints. Cardinality-based Feature Modelling added additional modelling concepts but still lacked a uniform way of describing features and constraints for a given model.
This paper describes a slightly modified approach to feature modelling, where all dependencies between features are explicitly defined with constraints. Different kinds of predefined constraints can be used to specify common dependencies. By extending the model with new specialized constraints, it can later be adapted for new requirements.
For the project described in this paper an XML-Schema for Constraint-based Feature Modelling was developed. To support the modelling process an existing XML-Editor was enhanced with a couple of useful features. Finally a tool for feature selection was developed. This tool supports simple constraint propagation and drives an incremental selection process.

Inhaltsverzeichnis

1 Grundlagen **1**

1.1 Problemstellung 1

1.1.1 VAI-PLE 2

1.1.2 FeatureKing 3

1.2 Feature-Modellierung 4

1.2.1 Grundbegriffe 4

1.2.2 FODA 5

1.2.3 Kardinalitäts-basierte Feature-Modellierung 6

1.2.4 Constraints 8

2 Theoretische Überlegungen **10**

2.1 Constraint-basierte Feature-Modellierung 10

2.1.1 Modellierungsansatz 10

2.1.2 Modellierungs-Baukasten 10

2.1.3 Constraints 11

2.1.4 Notation 13

2.1.5 Umsetzung 14

2.2 Constraint Propagation 17

3 Werkzeugunterstützung **21**

3.1 Eclipse 21

3.2 Bisherige Ansätze 22

3.3 Modellierung mit XML 22

3.3.1 XML-Author 23

3.3.2 Vereinfachte Erzeugung von XML 24

3.3.3 Umsetzung für XML-Author 25

3.4 FeatureKing 30

3.4.1 Allgemeines 31

3.4.2 Darstellung 31

3.4.3 Konsistenzprüfung 32

3.4.4 Constraint Propagation 34
3.4.5 Historie und Undo 35
3.5 Beispiel 37
3.5.1 XML-Author 37
3.5.2 FeatureKing 38

4 Zusammenfassung **42**

Literaturverzeichnis **43**

Anhang A **45**

Anhang B **46**

Kapitel 1

Grundlagen

1.1 Problemstellung

Eines der größten Probleme der Software-Industrie ist die rapide wachsende Komplexität der zu entwickelnden Systeme. Software ist Teil von mehr und mehr Produkten für Wirtschaft, Industrie und Alltag. Da ein immer größerer Teil der Entwicklungskosten von Software verschlungen wird, ist es von zunehmender Bedeutung, sie möglichst schnell und kosteneffizient herzustellen.
Neue Ansätze des Software Engineerings versuchen diesem Problem durch systematisches und automatisiertes Lösen von immer wiederkehrenden Problemen zu begegnen. *Product Line Engineering* (PLE) [19] ist ein spezieller Ansatz, um das Erstellen von Software für einen eingeschränkten Anwendungsbereich zu beschleunigen. In der Literatur wurden Software-Produktlinien einmal folgendermaßen definiert:

> „A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the needs of a particular market segment or mission and that are developed from a common set of core software assets in a prescribed way“ [17]

Ziel des PLE ist es, bisher manuell durchgeführte Arbeiten zu automatisieren und damit die Produktivität und Qualität zu erhöhen. In der Praxis führt dies zu zusätzlichen Vorteilen, da Entwicklungszeiten verkürzt und Kosten reduziert werden können. Es ist für die Entwicklung einer Produktlinie entscheidend, möglichst viele Gemeinsamkeiten der zu entwickelnden Produkte zu identifizieren und Variabilität nach vorgeschriebenen Mustern zu beschreiben.
Die Menge aller möglichen Produkte in einer Produktlinie wird mit einem Produktlinienmodell definiert. Dieses Modell beschreibt Variabilität der Produkte im „Problemraum“ (zum Beispiel geforderte Features) und im „Lösungsraum“ (zum Bei-

spiel Architektur und Komponenten des Endproduktes), sowie erlaubte Abbildungen zwischen diesen. Wie für die Beschreibung von Anforderungen mittels Feature-Diagrammen wurden auch für die Beschreibung der Variabilität von Software-Architekturen und Komponenten eigene Methoden und Werkzeuge entwickelt.
Die Entwicklung von Produkten mit Produktlinien kann in zwei Phasen eingeteilt werden. Zunächst müssen die zentralen Bestandteile aller Produkte der Produktlinie, zum Beispiel Software-Komponenten, identifiziert, entworfen und erstellt werden (*core asset development*). In einer zweiten Phase werden konkrete Produkte aus den zuvor entwickelten Bestandteilen zusammengebaut. Dazu werden Anforderungen des Kunden zunächst auf Features der Produktlinie abgebildet und diese anschließend von konkreten Komponenten umgesetzt. Feature-Modelle dienen dazu Gemeinsamkeiten und Unterschiede zwischen Mitgliedern der selben Produktlinie zu spezifizieren. Aus der abstrakten Beschreibung eines Produktes kann anschließend, voll oder teils automatisiert, konkrete Software erzeugt werden.
Feature-Modelle müssen sowohl mächtige genug sein, um komplexe Abhängigkeiten zwischen Features darzustellen, als auch einfach genug, um sowohl für Entwickler von Produktlinien, aber auch für Käufer und Verkäufer verständlich zu sein. Komplexe Abhängigkeiten werden erst nach und nach durch neue Ansätze zur Feature-Modellierung ermöglicht. Die ersten Feature-Modelle in FODA konnten nur sehr wenige Abhängigkeiten darstellen. Nachfolgende Modelle wie Cardinality-based Feature-Modelling erweiterten diese zwar schrittweise um neue Abhängigkeiten, ließen aber ein einheitliche Beschreibung dieser vermissen.
In dieser Arbeit wurde ein neuer Ansatz zur Feature-Modellierung entwickelt, der alle Abhängigkeiten zu explizit festgelegten Bestandteilen eines Modells macht. Alle Abhängigkeiten werden durch spezielle Constraints beschrieben. Durch die Erweiterbarkeit um neuen Spezialisierungen der vorhandenen Constraints kann das Modell ohne großen Aufwand an neue Anforderung angepasst werden.
Im Rahmen dieser Arbeit wurde ein XML-Schema für Constraint-basierte Feature-Modelle entwickelt. Zur Unterstützung des Modellierungs-Prozesses wurde ein XML-Editor um einige nützliche Features erweitert. Zusätzlich wurde ein Werkzeug für die Feature-Selektion entwickelt. Dieses unterstützt aufgrund von Constraint Propagation eine inkrementelle, halb-automatisierte Auswahl von Features.

1.1.1 VAI-PLE

Die im Laufe dieses Projekt durchgeführten Arbeiten konzentrieren sich auf die Produktlinie der Firma Siemens VAI (Voest Alpine Industrieanlagenbau), die im Bereich Analyse- und Steuer-Software für Stahlwerke angesiedelt ist.

Stahlwerke folgen einem modularen Aufbau und sind äußerst komplex. Wie in allen technischen Bereichen gewinnt Software auch in Industrieanlagen zunehmend an Bedeutung. Der modulare physische Aufbau von Maschinen, Steuer- und Analyse-Hardware kann zwar auf entsprechende Software-Komponenten und eine entsprechende Architektur abgebildete werden, die Komplexität moderner Software für Stahlwerke ist aber trotzdem enorm.
Alle Stahlwerke folgen prinzipiell dem gleichen Schema. Da die Größe, Funktionalität und verwendete Hardware stark von Werk zu Werk schwankt, gleicht dennoch keines dem anderen. Somit ist es nachvollziehbar, dass auch die verwendete Software flexibel genug sein muss, um auf ein bestimmtes Werk ohne großen Aufwand abgestimmt werden zu können.
Im Rahmen mehrere Forschungsprojekte bei Siemens VAI wird versucht, diesen derzeit großteils manuell durchgeführten Vorgang weitgehend zu automatisieren [18]. Die Software-Architektur, die der Produktlinie zu Grunde liegt, basiert auf Java und dem Spring-Komponentenframework [3]. Die Konfiguration der Komponenten erfolgt über XML und lässt sich zwar sehr flexibel an die Anforderungen der Kunden anpassen, erfordert aber derzeit noch erheblichen manuellen Aufwand. Weiterer großer manueller Aufwand ist mit dem Erstellen von speziellen Adaptern für das Zusammenspiel zwischen Komponenten innerhalb der Produktlinie, aber auch mit bereits vorhandenen Komponenten der Kunden.
Das größte Problem besteht derzeit allerdings darin, dass Software-Architekten und Verkäufer eine stark unterschiedliche Sicht auf ihr gemeinsames Produkt haben. So führt eine unglückliche Auswahl von Features durch Kunde und Verkäufer oft zu erheblichem Mehraufwand in der Konfiguration und Anpassung des Produktes für den Kunden. Diese fehlende Verbindung zwischen Verkäufer und Architekt soll mithilfe von PLE geschlossen werden.

1.1.2 FeatureKing

Zur Unterstützung der Produktlinienentwicklung in der VAI wurde ein erster Prototyp erstellt, der eine computer-unterstützte Konfiguration und Dokumentation von Produkten bestimmter Produktlinien ermöglicht.
Der typische Ablauf bei der Verwendung des FeatureKings ist wie folgt: In einer Vorbereitungsphase wird zunächst ein Modell für eine bestimmte Produktlinie erstellt. Dieses Model beschreibt Features des Produktes, konkrete Software-Komponenten, sowie Abhängigkeiten zwischen diesen.
Das für eine Produktlinie erstellte Modell kann danach in mehreren Konfigurationsphasen für das Erstellen konkreter Produkte verwendet werden.

1. Auswahl von Features
2. Konfiguration von Komponenten
3. Konfiguration von Projektdaten
4. Erstellen von Dokumentation

All diese Schritte dienen dazu, bisher manuell durchgeführte Arbeiten zu automatisieren. Die im Rahmen meines Projekts durchgeführte Arbeit am FeatureKing betrifft vor allem die Vorbereitungsphase und Schritt 1 der Konfigurationsphase. Diese Arbeiten werden im weiteren Verlauf dieser Arbeit beschrieben.

1.2 Feature-Modellierung

Im folgenden Abschnitt werden die grundlegenden Konzepte von Feature-Modellierung eingeführt. Für eine ausführlichere Einführung in die Thematik siehe [11].

1.2.1 Grundbegriffe

Feature

Features sind entscheidende, für den Benutzer sichtbare Leistungen eines Systems. Features können sowohl funktionale Anforderungen, wie „Syntax-Highlighting“, als auch nicht-funktionale Anforderungen, wie Antwortzeiten von maximal einer Sekunde, beschreiben.

Feature-Modell

Ein Feature-Modell umfasst alle Features eines Systems und Abhängigkeiten zwischen diesen. Es dient dazu Gemeinsamkeiten und Variabilität verschiedener Produkt-Konfigurationen auf strukturierte und übersichtliche Weise zu modellieren.

Feature-Diagramm

Viele Feature-Modelle lassen sich durch eine hierarchische, baumartige Struktur beschreiben. Deshalb können sie ähnlich wie Klassenhierarchien in UML in Form von Diagrammen dargestellt werden. Dies ist für kleine Modelle die übersichtlichste und anschaulichste Form, jedoch bei großen Modellen schwer zu handhaben. Eine interaktive Darstellung als Baum, wie sie z.B. bei der Navigation durch Dateisysteme verwendet wird, ist bei großen Modellen handlicher. Querverweise können nicht direkt dargestellt werden, dennoch kann mit Hilfe von Verknüpfungen leicht zwischen

Teilen des Baums hin und hergesprungen werden.
Zur internen Repräsentation und zum Erstellen und Editieren von umfangreichen Modellen eignen sich Textformate, da diese sowohl von Mensch als auch Maschine leicht verständlich und mit geeigneten Editoren effizient zu bearbeiten sind.

1.2.2 FODA

Eingeführt wurde Feature-Modellierung erstmals im Rahmen von FODA (Feature Oriented Domain Analysis) [12]. Die wesentlichen Bestandteile von FODA liegen auch allen aktuellen Ansätzen zu Grunde und werden deshalb kurz erläutert.
Feature-Modelle werden in FODA in Form von sogenannten *Feature-Bäumen* beschrieben, wobei die Knoten des Baumes den Features des Modells entsprechen. Abhängigkeiten werden durch Hierarchie und speziellen Typen für Knoten und Kanten beschrieben. Die Hierarchie des Baumes legt fest, welche Features grundsätzlich selektiert werden können. Features können nur dann selektiert werden, wenn auch deren Vater-Feature im Baum selektiert ist. Davon ausgenommen ist das Wurzel-Feature, das immer selektiert sein muss.
Mit speziellen Typen für Knoten und Kanten lassen sich weitere Abhängigkeiten definieren. *Verpflichtende Features* müssen in einer Instanz eines Feature-Modells genau dann gewählt werden, wenn deren übergeordnetes Feature (Vater) gewählt wird. *Optionale Features* können nur dann, müssen aber nicht gewählt werden, wenn deren Vater gewählt wurde. Aus einer Gruppe von *Alternativen Gruppe* muss genau eines gewählt werden und aus einer *Oder-Gruppe* muss mindestens eine Feature gewählt werden.

Notation

Die mit FODA eingeführte Notation für Feature-Modelle, genannt *Feature-Diagramme*, wurde bis heute weitgehend beibehalten. Die exakte Darstellung schwankt zwar von Ansatz zu Ansatz, aber das Grundkonzept, Features durch Knoten und Abhängigkeiten durch dekorierte Kanten darzustellen, wurde beibehalten.

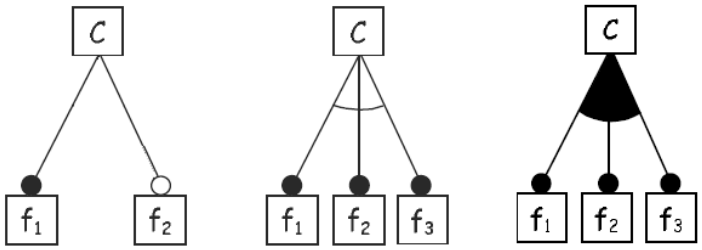


Abbildung 1.1: Feature-Diagramm: Verpflichtend, Optional, Alternative, Oder [11]

Abbildung 1.1 zeigt drei Diagramme. Links ist f1 verpflichtend und f2 optional, in der Mitte sind f1, f2 und f3 eine Gruppe von alternativen Features, von denen genau eines gewählt werden muss und rechts bilden f1, f2 und f3 eine Gruppe von denen mindestens eines gewählt werden muss.

Beispiel

Ein in der Literatur oft verwendetes Beispiel zur Veranschaulichung von Feature-Modellen und Feature-Diagrammen ist in Abbildung 1.2 zu sehen [11].

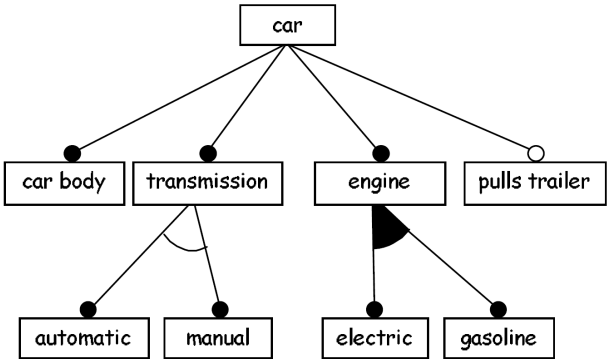


Abbildung 1.2: Feature-Diagramm für ein Auto [11]

In diesem einfachen Beispiel wird ein Auto auf sehr abstrakte Weise modelliert. Das Auto besteht aus Karosserie, Schaltungsgetriebe und Motor. Das Schaltungsgetriebe existiert in manueller und in automatischer Ausführung, wobei nur eines der beiden wählbar ist. Weiters stehen sowohl Benzin-Motor als auch Elektro-Motor zur Auswahl, im Gegensatz zum Schaltgetriebe können allerdings auch beide gewählt werden (hybrid). Optional kann das Auto noch einen Anhänger nachziehen.

1.2.3 Kardinalitäts-basierte Feature-Modellierung

Kardinalitäts-basierte Feature-Modellierung [13] erweitert FODA um allgemeine Kardinalitäten für einzelne Features und Gruppen von Features. Kardinalitäten für Features bestimmen die minimale und maximale Anzahl von Ausprägungen, die von diesem einen Feature gewählt werden können. Kardinalitäten für Gruppen bestimmen wie viele der Features in einer bestimmten Gruppe ausgewählt werden können. Durch diese flexiblere Limitierungen sind die entstehenden Modelle oft kompakter als FODA-Modelle mit vergleichbarer Semantik. Die in FODA beschriebenen Bausteine sind nur noch Spezialfälle des allgemeineren Konzeptes. Verpflichtende Features sind Features mit Kardinalität [1..1], sprich diese Features müssen mindestens einmal und maximal einmal ausgewählt werden. Optionale Features sind Features

mit Kardinalität [0..1]. Gruppen von alternativen Features sind Gruppen mit der Kardinalität [1..1], sprich aus der Menge der Features in der Gruppe muss genau eines gewählt werden.

Notation

Die Notation von FODA wurde großteils übernommen, auch für die dort beschriebenen Spezialfälle werden weiterhin eigene Symbole verwendet.

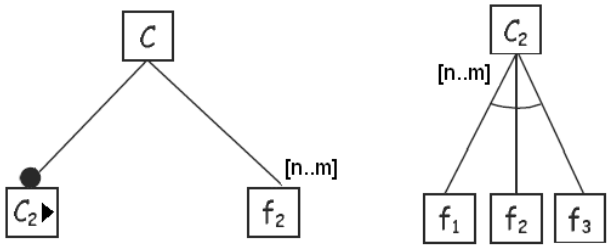


Abbildung 1.3: Verweis, Feature- und Gruppen-Kardinalität [2]

Abbildung 1.3 zeigt wieder 2 Diagramme. Links ist ein Feature mit Kardinalität [n..m], f2 muss also mindestes n mal und maximal m mal ausgewählt werden. Der kleine Pfeil neben C2 deutet an, dass es sich um eine Referenz auf ein Feature handelt, das in einem eigenen Baum dargestellt wird. Rechts ist das referenzierte Feature C2 und eine Gruppe mit der Kardinalität [n..m]. Aus der Gruppe der Features müssen also mindestes n und maximal m gewählt werden.

Beispiel

Der Einsatz von Kardinalitäten bei der Feature-Modellierung soll am Beispiel verdeutlicht werden. Abbildung 1.5 zeigt ein Feature-Diagramm zur Beschreibung der Innenausstattung eines Autos.

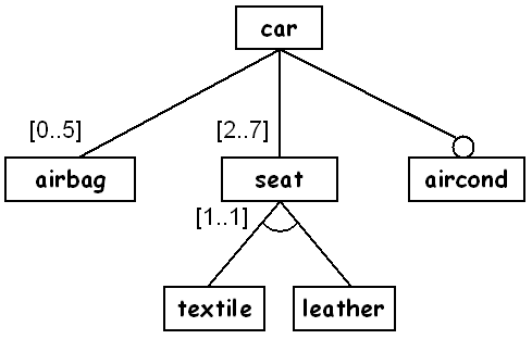


Abbildung 1.4: Feature-Diagramm für die Innenausstattung eines Autos

Die Innenausstattung besteht aus 2 bis 7 Sitzen, 0 bis 5 Airbags und einer optionalen Klimaanlage (Kardinalität [0..1]. Sitze können entweder mit Stoff oder mit Lederbezug ausgestattet werden.

1.2.4 Constraints

Constraints (Einschränkungen) sind Regeln die bei der Selektion von Features zu befolgen sind. Durch die Hierarchie von Features und deren Kardinalitäten werden bereits implizite Constraints definiert. Explizite Constraints sind notwendig, wenn sich Abhängigkeiten nicht oder nur umständlich durch Feature-Hierarchien darstellen lassen. Ein typisches Szenario sieht folgendermaßen aus: Ein Features f1 in einem Teil des Feature-Baums benötigt ein Feature f2 in einem anderen Teil des Feature-Baumes, bevor es selektiert werden kann. Dies kann durch den Constraint *f1 requires f2* beschrieben werden. Constraints müssen in realistischen Modellen oft komplexe Zusammenhänge beschreiben, es ist also wichtig, dafür eine mächtige Beschreibungssprache zur Verfügung zu stellen.

XPath

XPath ist eine mächtige Sprache mit der große XML-Dateien auf spezielle Punkte von Interesse reduziert werden können. Wie mit Pfaden in Dateisystemen oder URLs im World Wide Web, lassen sich mit XPath-Ausdrücken Teile des ganzen referenzieren. Zusätzlich erlaubt XPath 2.0 logische Operationen, Mengen-Operationen und komplexe Berechnungen mit Attributen. Es folgen ein paar Beispiele für Constraints mittels XPath.

```
1  if(//f1) then //f2
2  some x in //f1 satisfies $x[@value >= 5]
```

Zeile 1 zeigt einen einfachen *requires* Constraint. Zeile 2 zeigt wie logische Operationen dazu verwendet werden können, komplexe Einschränkungen zu spezifizieren. Wie in [1] gezeigt wird, kann XPath für in XML vorliegende Feature-Modelle als allgemeine Constraint-Sprache verwendet werden.

Object Constraint Language

Die Object Constraint Language (OCL) ist eine Constraint-Sprache, die speziell für UML-Modelle entworfen wurde. Wie in Feature-Modellen können in UML nicht beliebig komplexe Abhängigkeiten und Einschränkungen mit den normalen Sprach- und Modellierungs-Elementen ausgedrückt werden. In [2] wird gezeigt, dass sich die OCL auch für Feature-Modellierung eignet, und aufgrund einer geeigneteren Syntax

meist leichter verständlich ist. Das Beispiel zeigt die gleichen Constraints wie zuvor für XPath:

```
1  f1.isSelected() implies f2.isSelected()
2  f1.value.att->exists(x | x >= 5)
```

Um typische Constraints auszudrücken, besitzen alle Objekte des Feature-Modells zusätzliche Methoden und Attribute (wie isSelected() oder subFeatures()).

Kapitel 2

Theoretische Überlegungen

2.1 Constraint-basierte Feature-Modellierung

In diesem Abschnitt wird der für diese Arbeit entwickelte Ansatz zur Feature-Modellierung beschrieben. Nach der Beschreibung der allgemeinen Idee, werden Schritt für Schritt die einzelnen Bestandteile beschrieben, und gezeigt wie diese zu komplexen Feature-Modellen zusammengesetzt werden können.

2.1.1 Modellierungsansatz

Die Idee hinter Constraint-basierter Feature-Modellierung besteht darin, Constraints nicht implizit durch verschiedene Feature-Typen, wie Mandatory-Feature, Group-Feature und Grouped-Feature zu beschreiben, sondern diese explizit, als eigene Knoten im Feature-Graph, zu definieren. Features existieren somit nur mehr in einer einzigen, einheitlichen Forum und stehen mit anderen Features nur mehr indirekt über Constraints in Beziehung.

2.1.2 Modellierungs-Baukasten

Unsere Feature-Modelle bestehen im wesentlichen aus 5 verschiedenen Bauteilen. *Features* beschreiben sichtbare Leistungen eines Systems, *Properties* beschreiben spezielle, konfigurierbare Eigenschaften einzelner Features, *Constraints* beschreiben Abhängigkeiten zwischen Features und Properties, *Komponenten* beschreiben konkrete Software-Bausteine und *Annotationen* dienen letztlich zur Dokumentation des Feature-Modells.

Features

Features bestehen aus einer eindeutigen ID, einem Namen und beliebig vielen Annotationen, Properties, Komponenten und Constraints. Es wird nicht zwischen Mandatory, Optional u.s.w. unterschieden. Jedes Feature besitzt die gleiche Struktur und unterscheidet sich von anderen nur durch seine Bestandteile.

Properties

Properties dienen der zusätzlichen Beschreibung der Eigenschaften von Features. Dadurch entstehen Features, die eine flexible Anpassung an die Bedürfnisse des Benutzers ermöglichen. Properties haben eine eindeutige ID, einen Namen, einen bestimmten Datentyp mit optionalem Wertebereich und einen Standardwert. Obwohl beliebige Datentypen denkbar sind, wurden im Rahmen dieser Arbeit nur 4 besonders häufig benötigte Datentypen berücksichtigt: Diskrete Werte in einem vorgegebenen Wertebereich, kontinuierliche Werte, Text und Aufzählungen.

Komponenten

Komponenten sind jene Software-Bausteine, die ein Feature später im Endprodukt ermöglichen. Nicht jedes Feature kann von einer einzigen Komponenten realisiert werden und manche Komponenten realisieren gleich mehrere Features. Wie zwischen Features bestehen auch zwischen Komponenten komplexe Abhängigkeiten. Diese werden jedoch nicht vom Feature-Modell, sondern unabhängig davon in Spring [3] XML-Dateien beschrieben. Die Komponenten im Feature-Modell bestehen also lediglich aus Referenzen auf diese Dateien.

Annotationen

Annotationen dienen zur Dokumentation und können an allen anderen Knoten angebracht werden. Sie bestehen aus einer Kurzbeschreibung, einer ausführlichen Beschreibung, beliebig vielen Kommentaren mit Verfasser und Datum, so wie aus Hyperlinks zu weiterer Dokumentation.

2.1.3 Constraints

Constraints werden dazu verwendet, sämtliche Abhängigkeiten zwischen Features zu beschreiben. Es wird nicht zwischen impliziten und expliziten Constraints unterschieden. Alle Constraints werden einem Feature zugeordnet und sind genau dann verletzt, wenn das zugehörige Feature selektiert und die Bedingung des Constraints

verletzt ist. Constraints sind auch dann verletzt, wenn direkt untergeordnete Features selektiert sind, ohne das übergeordnete Feature zu selektieren. Im folgenden werden die Bedingungen der einzelnen Constraints spezifiziert:

Mandatory

Die Bedingung des Mandatory Constraint ist: Alle Subfeatures müssen selektiert sein.

Optional

Die Bedingung des Optional Constraint ist: Alle Subfeatures können selektiert sein.

Solitary

Die Bedingung des Solitary Constraint ist: Alle Subfeatures müssen mindestens *min* und maximal *max* mal selektiert sein.

XorGroup

Die Bedingung des XorGroup Constraint ist: Es muss genau eines der Subfeatures selektiert sein.

Group

Die Bedingung des Group Constraint ist: Es müssen mindestens *min* und maximal *max* der Subfeatures selektiert sein.

Requires

Die Bedingung des Requires Constraint ist: Alle *referenzierten* Features müssen selektiert sein.

Excludes

Die Bedingung des Excludes Constraint ist: Alle *referenzierten* Features müssen deselektiert sein.

Expression

Die Bedingung des Expression Constraint ist: Der benutzerdefinierte Ausdruck muss wahr sein.

2.1.4 Notation

Da das vorgestellte Konzept zur Feature-Modellierung hierarchische Strukturen erzeugt, kann es einfach als Diagramm dargestellt und in XML spezifiziert werden.

Diagramm

Abbildung 2.1 zeigt ein einfaches Feature-Modell mit den Features *Car*, *Tires*, *Unleaded*, *Diesel* und *HiFi*. Das Feature *Car* besitzt drei Constraints, Reifen werden unbedingt benötigt (Mandatory), es wird entweder ein Benzin oder Diesel Motor benötigt (XorGroup) und optional kann eine HiFi Anlage integriert werden.

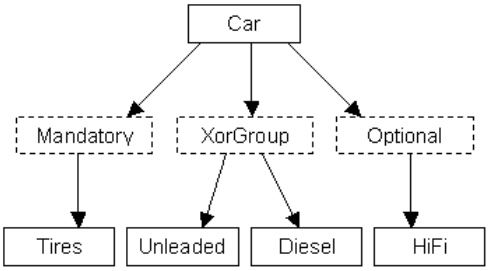


Abbildung 2.1: Modell als Diagramm

XML

Das gleiche Beispiel wie im Diagramm kann in XML folgendermaßen repräsentiert werden:

```
 1  <?xml version="1.0" encoding="UTF-8"?>
 2  <FeatureModel id="car" name="Car">
 3    <Mandatory>
 4      <Feature id="tires" name="Tires"></Feature>
 5    </Mandatory>
 6    <XorGroup>
 7      <Feature id="unleaded" name="Unleaded"></Feature>
 8      <Feature id="diesel" name="Diesel"></Feature>
 9    </XorGroup>
10    <Optional>
11      <Feature id="hifi" name="HiFi"></Feature>
12    </Optional>
13  </FeatureModel>
```

Dieses Model ist, analog zum Diagramm, frei von Annotationen, Properties und Komponenten. Das Wurzelfeature wird als `FeatureModel` bezeichnet, da es alle anderen Features und Constraints umspannt.

2.1.5 Umsetzung

Der Lebenszyklus von Feature-Modellen ist wie folgt: Die Modelle werden zunächst in XML-Dateien mit einem vorgegebenen Schema erstellt. Diese Dateien werden eingelesen und in einen Objektgraphen übersetzt. Das entwickelte Werkzeug für die Feature-Selektion verwendet den Objektgraphen als internes Modell. Zunächst wollen wir das persistente XML-Format betrachten.

XML-Schema

XML-Schema [14] ist eine Sprache zur Strukturbeschreibung von XML-Dateien. Im Schema wird definiert, wo und wie oft bestimmte Elemente im Dokument vorkommen dürfen und welche Attribute sie besitzen müssen oder können. Zusätzlich können primitive Datentypen für die Werte von Elementen oder Attributen festgelegt werden. In Abbildung 2.2 zeigt ein Diagramm eine vereinfachte Version des entworfenen XML-Schemas für Feature-Modelle. Das Schema wird als Baum, der von links nach rechts aufgespannt wird, dargestellt.

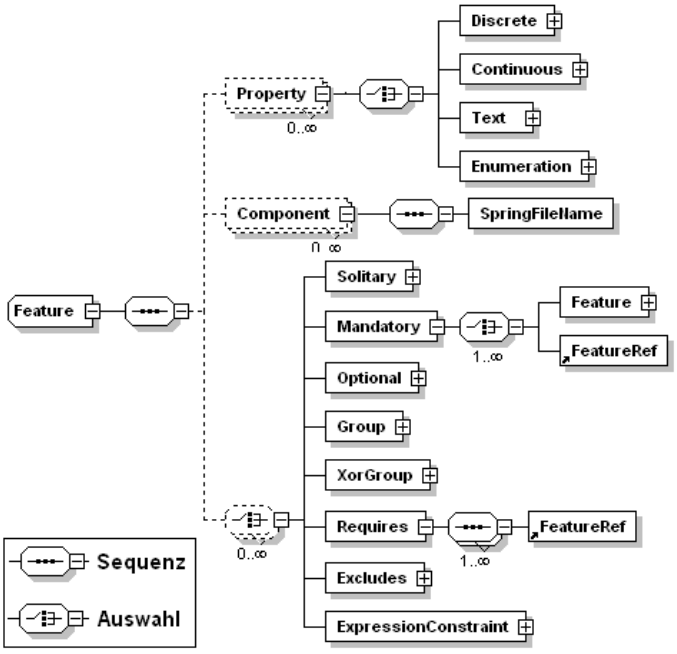


Abbildung 2.2: XML Schema für Feature-Modelle

Das Wurzelelement, *Feature*, besteht aus einer Sequenz von Unterelementen. 0 oder mehr *Property*-Elemente, gefolgt von 0 oder mehr *Component*-Elementen, gefolgt von 0 oder mehr Constraints (*Solitary*, *Mandatory*,...) in beliebiger Reihenfolge. Analog wird die Struktur der anderen Elemente beschrieben, wobei aus Gründen der

besserer Übersichtlichkeit manche Details ausgeblendet wurden (erkennbar durch ein `+` am rechten Rand der Elemente).

Objektgraph

Statische XML-Dateien, die dem vorgegebenen Schema für Feature-Modelle folgen, können in einen Objektgraph übersetzt werden. Dazu wird zunächst das Document Object Model (DOM) für eine bestimmte Datei erzeugt und anschließend der entsprechende Objektgraph Schritt für Schritt aufgebaut.
Der Objektgraph entspricht dabei einer Instanz des in Abbildung 2.3 gezeigten Klassendiagramms. Dieses entspricht im wesentlichen dem vorgestellten XML-Schema. Features können Properties, Components und Constraints enthalten. Constraints enthalten wiederum Features oder Referenzen auf Features in anderen Teilen des Baums. Das oberste Wurzel-Feature ist ein Objekt der Klasse `FeatureModel` und bietet als solches zusätzliche Funktionalität für verarbeitende Werkzeuge an.

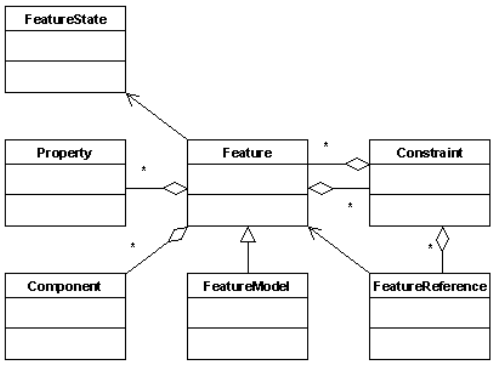


Abbildung 2.3: Klassendiagramm - Feature-Modell

Alle unterstützten Constraints erben von der abstrakten Klassen `Constraint` und implementieren individuelles Verhalten zum Validieren und Propagieren. In der in Abbildung 2.4 gezeigten Klassenhierarchie für Constraints ist zu erkennen, wie manche Constraints als Spezialfälle von anderen implementiert werden konnten. `XorGroup` entspricht zum Beispiel einer Gruppe mit Kardinalität 1..1.

Expression-Constraints

Expression-Constraints erlauben dem Benutzer Constraints mit beliebigen, benutzerdefinierten Ausdrücken zu erstellen. Da Ausdrücke mittels strukturiertem XML sehr umständlich zu lesen, zu erstellen und zu bearbeiten sind, wurde eine kompaktere Text-Repräsentation gewählt. Diese besteht im wesentlichen aus einem logischen Ausdruck in einer Java-ähnlichen Syntax. Optional können vor dem Ausdruck

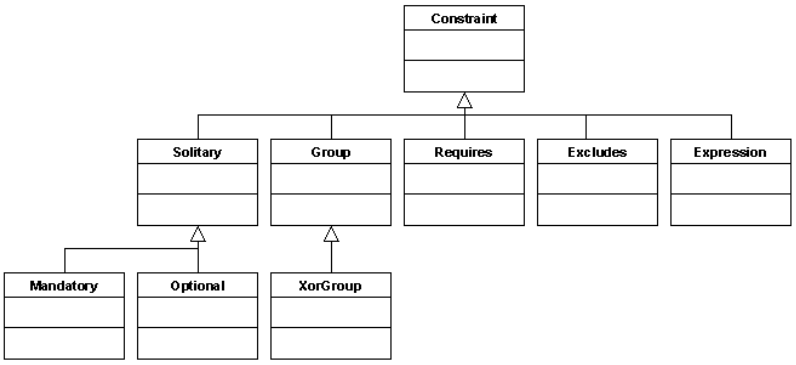


Abbildung 2.4: Klassendiagramm - Constraints

Kurzbezeichnungen (Aliase) für Features oder Properties definiert werden. Am einfachsten wird dies mit einem Beispiel verdeutlicht:

```
1  <ExpressionConstraint>
2    <Code>
3      <![CDATA[
4        ALIAS hp = car.horsepower;
5        50 <= hp && hp <= 200
6      ]]>
7    </Code>
8  </ExpressionConstraint>
```

Zunächst wird für den Ausdruck `car.horsepower` der Alias `hp` definiert (Zeile 4). Dann folgt der logische Ausdruck, der wahr sein muss, um den Constraint zu erfüllen. `50 <= hp && hp <= 200` gibt an, dass zwischen 50 und 200 Pferdestärken für dieses Feature benötigt werden.
Für die Umsetzung von Expression-Constraints wurde ein eigener kleiner Interpreter entwickelt. Die kurzen Code-Stücke werden beim Parsen der XML-Datei einmal in einen Abstrakten Syntaxbaum (AST) übersetzt und später jedes mal mit den aktuellen Werten für Features und Properties neu evaluiert.

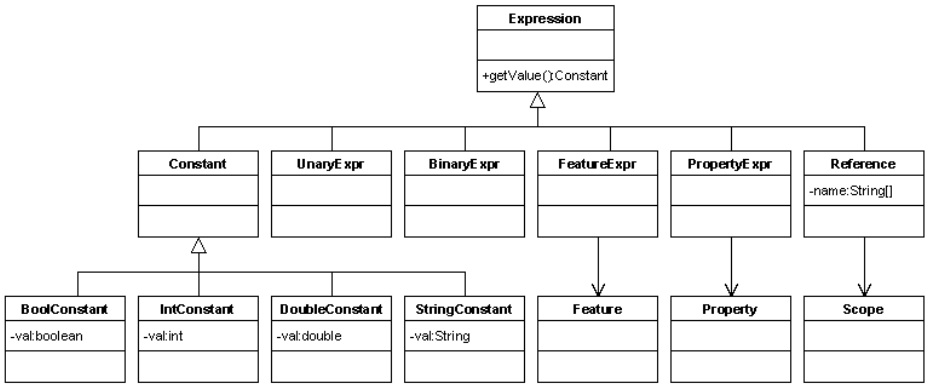


Abbildung 2.5: Klassendiagramm - AST für Expression Constraints

Scanner und Parser wurde mit einer attributierten Grammatik für Coco/R [4] realisiert (siehe Anhang B). Der AST besteht aus Konstanten, Referenzen auf Features und Properties, sowie bestimmten logischen und arithmetischen Verknüpfungsoperatoren. Abbildung 2.5 zeigt die involvierten Klassen als Diagramm.
Jede Expression implementiert die Methode `getValue()`, deren Ergebniss zur Laufzeit immer ein konstanter Wert ist. Features evaluieren zu `true` (selektiert) oder `false` (deselektiert). Properties evaluieren zu ihrem momentanen Wert, Konstanten evaluieren zu sich selbst. Arithmetische oder logische Ausdrücke evaluieren zum Ergebnis der Anwendung des entsprechenden Operators auf die konstanten Werte der Operanden.

2.2 Constraint Propagation

Die Auswahl eines bestimmten Features kann die Auswahl anderer Features implizieren oder verbieten. Selektierte Features können weitere Constraints besitzen, somit kann die Auswahl eines einzelnen Features weitreichende Konsequenzen haben. Dieser Prozess wird als Constraint Propagation bezeichnet, da sich die Constraints eines Features auf den gesamten Graph auswirken können.
Constraint Propagation ist ein Verfahren, das unter anderem zur Lösung des Constraint Satisfaction Problems (CSP) eingesetzt wird [15,16]. Dabei wird nach einer Belegung von Variablen gesucht, die eine bestimmte Menge von zuvor festgelegten Bedingungen erfüllt. Ein möglicher Weg zu einer Lösung sieht wie folgt aus:

1. Initialisiere alle Variablen mit einem beliebigen Wert.
2. Überprüfe ob alle Bedingungen erfüllt sind: Wenn ja, ENDE, ansonsten weiter bei Schritt 3.
3. Wähle eine beliebige Variable und ändere deren Wert.
4. Propagiere die Auswirkungen dieser Änderung auf alle anderen Variablen und ändere gegebenenfalls auch deren Werte.
5. Weiter bei Schritt 2.

Die Erstellung einer gültigen Auswahl an Features für ein bestimmtes Feature-Modell kann als CSP verstanden werden. Das Feature-Modell besteht aus einer Menge von booleschen Variablen (Features), die entweder true (selektiert) oder false (deselektiert) sein können. Constraints beschreiben Bedingungen für eine gültige Belegung dieser Variablen mit konkreten Werten. Im folgenden Beispiel wird ein einfaches Modell durch eine Menge boolscher Ausdrücke beschrieben:

```
1  car
2  car IMPLIES motor
3  motor IMPLIES car
4  motor IMPLIES (unleaded XOR diesel)
5  y IMPLIES motor
6  z IMPLIES motor
```

Das Wurzel-Feature ist *car*, und dieses muss immer selektiert sein (1). Bei *motor* handelt es sich offensichtlich um ein verpflichtendes Feature (2,3), das wiederum aus einer XOR-Gruppe mit 2 Unter-Features besteht. Wird *motor* selektiert, so muss auch *unleaded* oder *diesel* selektiert werden, aber nicht beide (4). Wird *unleaded* oder *diesel* selektiert, so muss auch deren Vater, *motor*, selektiert werden (5,6). Um zu einer Lösung des Problems zu bekommen, werden zunächst alle Variablen initialisiert.

```
car = motor = unleaded = diesel = false
```

Jetzt wird überprüft, ob bereits alle Bedingungen erfüllt sind. Bedingung 1 ist nicht erfüllt, also folgt Schritt 3: Es wird zufällig die Variable *unleaded* gewählt und deren Wert von false auf true gesetzt. Diese Wertänderung propagiert sich aufgrund von Bedingung 5 auch auf *motor*, und aufgrund von Bedingung 3 auch auf root. Der neue Zustand ist wie folgt:

```
car = motor = unleaded = true, diesel = false
```

Jetzt wird erneut überprüft, ob der aktuelle Zustand alle Bedingungen erfüllt und tatsächlich ist dies der Fall.
In diesem kleinen Beispiel hat eine Iteration ausgereicht, um alle Constraints zu erfüllen. In komplexen Modellen sind oft viele Iterationen notwendig, um einen konsistenten Zustand zu erreichen. Um diese Probleme möglichst effizient und voll automatisiert lösen zu können, existieren zahlreiche Algorithmen und Optimierungen, die jedoch für den interaktiven Prozess der Feature-Modellierung nicht notwendig sind. Da die nicht-deterministische Auswahl von Variablen mit anschließender Wertänderung (Schritt 3) durch den Benutzer durchgeführt wird, reicht es aus, nur Constraint Propagation und Konsistenz-Prüfung (Schritt 2 und 4) automatisiert durchzuführen.
Dieser interaktive Vorgang ist im Sequenzdiagramm in Abbildung 2.6 zu sehen. Der Benutzer selektiert ein Feature (Nachricht 5), welches die eigene Wertänderung an alle interessierten Constraints bekannt gibt (Nachrichten 6, 7 und 9) . Der Mandatory Constraint selektiert automatisch, alle Subfeatures (Nachricht 8), propagiert die Wert-Änderung also an andere Features weiter. Der Optional Constraint hat in diesem Fall keine weiteren Konsequenzen, da er auch erfüllt ist, wenn die Subfeatures

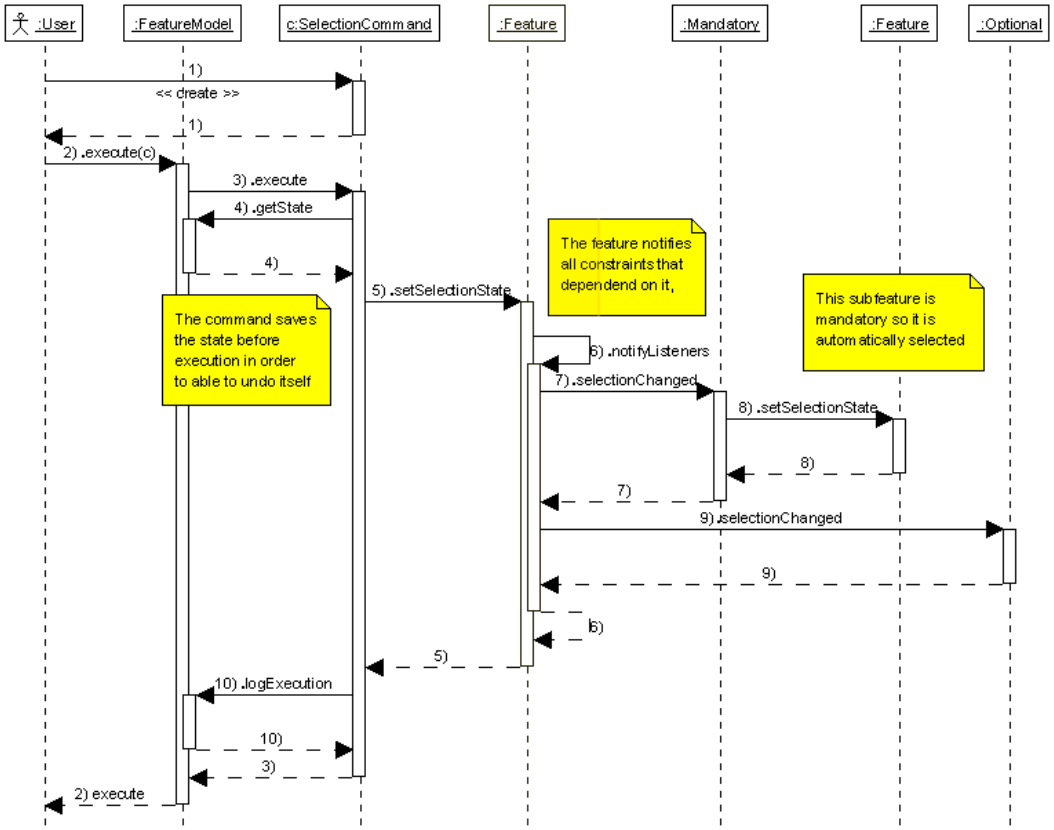


Abbildung 2.6: Sequenzdiagramm - Feature-Selektion

nicht selektiert werden.

In Abbildung 2.7 wird das Propagieren von Constraints in Form von mehreren Momentaufnahmen dargestellt. Features (F) oder Constraints (C), die von der Wertänderung bereits betroffen sind, werden grau dargestellt. Über gestrichelte Kanten werden Constraints an benachbarte Knoten propagiert. Wie im letzten Baum zu sehen ist, hat sich die Wertänderung eines einzelnen Features letztlich auf den gesamten Baum ausgewirkt.

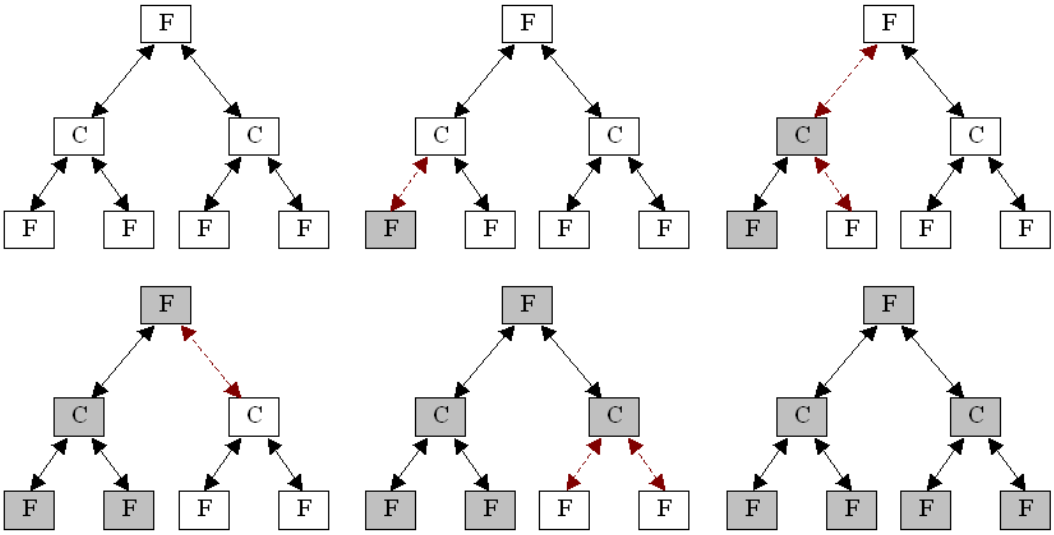


Abbildung 2.7: Propagieren von Constraints auf den gesamten Feature-Baum

Kapitel 3

Werkzeugunterstützung

Nach all der Theorie ist es nun an der Zeit, uns die Umsetzung der vorgestellten Konzepte anzusehen. Die zu entwickelnden Werkzeuge sollten Teil des in der Problemstellung vorgestellten Prototyps, FeatureKing, sein. Im folgenden werde ich kurz das verwendete Applikations-Framework beschreiben und danach die entwickelten Werkzeuge zur Erstellung von Feature-Modellen und anschließender Selektion von Features vorstellen.

3.1 Eclipse

Eclipse [5] [8] ist ein Plattform-unabhängiges Software-Framework, dass die Entwicklung graphischer Desktop-Applikation erleichtern soll. Dies wird einerseits durch umfangreiche Bibliotheken, andererseits durch eine äußerst flexible Architektur erreicht.
Das Flagschiff der auf Eclipse basierten Applikationen ist eindeutig die populäre Java Entwicklungsumgebung (IDE), genannt Java Development Tools (JDT) [6]. Eclipse selbst ist aber keine IDE, sondern eine Plattform für verschiedenste Java-basierter Applikationen.
Die Basis von Eclipse ist die Rich Client Platform (RCP). Sie besteht im wesentlichen aus folgenden Komponenten:

- Core platform (Starten von Eclipse, Laden und Starten von Plugins)
- SWT (Portable graphische Komponenten (widget toolkit)
- JFace (Dateibuffer, Absrakte Sichten auf SWT Komponenten, Text-Editoren,...)
- The Eclipse Workbench (Basisbausteine: views, editors, perspectives, wizards)

SWT, JFace und Workbench Komponenten können dazu verwendet werden, komplexe Applikationen in relativ kurzer Zeit zu erstellen. Aufgrund des riesigen Umfangs

der Bibliotheken ist der Einarbeitungsaufwand zwar relativ hoch, aufgrund von guter Dokumentation und viel Beispielcode können die meisten Komponenten dennoch recht einfach auf die Bedürfnisse bestimmter Applikationen angepasst werden.
Die Core Platform übernimmt typische Funktionalität wie Verwaltung von Bibliotheken, das Laden von Plugins, Sitzungen, Konfigurationen und so weiter. Dadurch kann man sich bei der Entwicklung von Plugins vollständig auf die eigentliche Funktionalität konzentrieren.

3.2 Bisherige Ansätze

Seit Begin der Arbeiten am FeatureKing wurden zwei Ansätze zu Erstellung von Feature-Modellen verwendet. Zunächst wurden Feature-Modelle durch Java Source-Code definiert. Klassen für verschiedene Features, Properties und Komponenten wurden erstellt und anschließend entsprechende Objektgraphen erzeugt. Dieser Ansatz war für erste Testzwecke praktisch, da so gut wie kein Code zu entwickeln war. Für reale Modelle wurde dieser Ansatz bald aufgegeben und ein neuer, benutzerfreundlicher, graphischer Editor erstellt. Dieser wurde mithilfe des populären Graphical Editor Frameworks (GEF) [7], einem Plugin für Eclipse, entwickelt (Abbildung 3.1). Der Editor erlaubt dem Benutzer das Erstellen von Modellen durch Platzieren von Features und Properties auf einer graphischen Zeichenoberfläche und Erstellen von Kanten zwischen Features zu zusammenhängenden Graphen. Obwohl es möglich war, die Darstellungsgröße des Graphen zu variieren, wurde klar, dass größere Modelle in einem graphischen Editor schwer zu handhaben sind.

3.3 Modellierung mit XML

Als erster Schritt war eine persistente Speicherung von Feature-Modellen zu realisieren. Diese sollte sowohl für Mensch, als auch für Maschine, leicht zu lesen, editieren und erstellen sein. Aufgrund der vorwiegend baumartigen Hierarchie, die den meisten Feature-Modellen zu Grund liegt, ist es naheliegend dafür XML [9] zu verwenden. Mittlerweile existieren unzählige Werkzeuge, Editoren und ganze Entwicklungsumgebungen für das Arbeiten mit XML-Technologien. Um das effiziente Erstellen von Modellen zu ermöglichen, sind folgende Features wünschenswert:

- Syntax Highlighting: Farbliche Hervorhebung verschiedener Textteile
- Auto-Indent: Automatisches Einrücken von Text
- Auto-Format: Automatisches Formatieren ganzer XML-Dateien

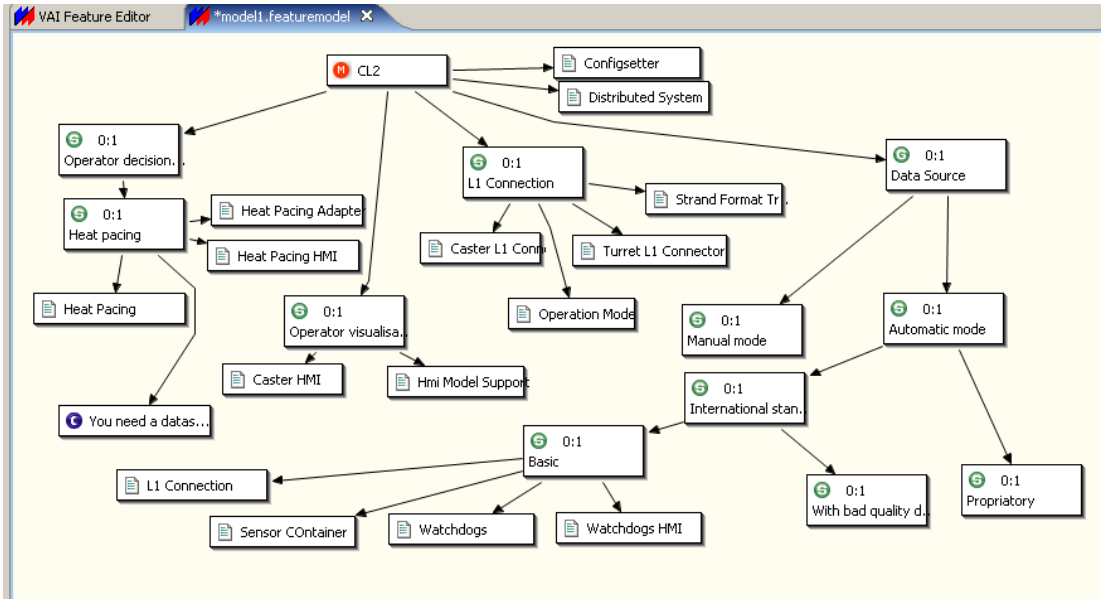


Abbildung 3.1: Graphical Feature Model Editor.

- Outline: Abstrakte Darstellung der XML-Daten als Baum
- Folding: Auf-/Zuklappen einzelner XML-Elemente im Editor
- Validierung gegen Schema und markieren von Fehlern
- Content Assist: Sinnvolle Vorschläge für neue XML-Elemente
- Templates: Benutzer-definierte Schablonen für häufig benötigte Muster

Wie sich herausstellte, unterstützen die meisten Editoren die gewünschte Funktionalität zur Präsentation von XML-Daten. Die Funktionalität zur Erstellung von XML-Daten lässt aber oft zu Wünschen übrig. Das Problem ist, dass konventionelle XML-Editoren für unseren Zweck zu generisch sind. Die einzige Möglichkeit die allgemeinen XML-Editoren zur automatischen Erzeugung von Code zur Verfügung steht, sind DTDs oder XML-Schemas. Aus diesen können zwar korrekte Vorschläge generiert werden, diese weichen aber meistens stark von typischen Verwendungsmustern ab.

3.3.1 XML-Author

Die Verwendung eines bestehenden Editors war somit ausgeschlossen und die gewünschte Funktionalität konnte nur durch ein neues Werkzeug realisiert werden. Da ich das Rad nicht neu erfinden wollte, sah ich mich nach bestehenden Open Source XML-Editoren für Eclipse um. XML-Author [10] ist ein schöner, leicht-gewichtiger XML-Editor, der alle gewünschten Features zur Darstellung von XML unterstützt. Der Editor verwendet Standard-Bibliotheken zum Lesen und Schreiben von XML

Daten sowie viele der von Eclipse bereitgestellten Standard-Mechanismen zur Erweiterung des einfachen Standard-Text-Editors um neue Features.

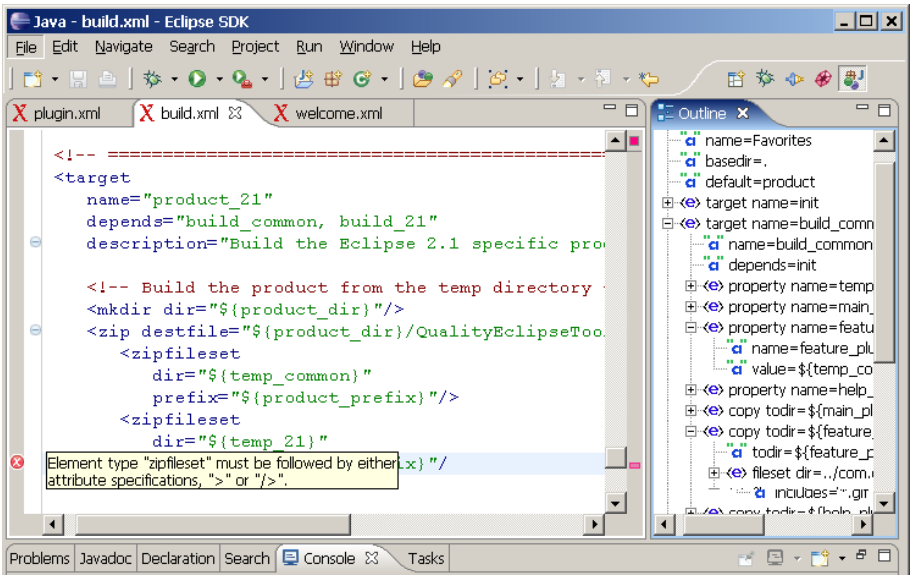


Abbildung 3.2: XML-Author

3.3.2 Vereinfachte Erzeugung von XML

Ähnlich wie die Features zur Präsentation von XML können auch die Features für das Schreiben von XML mit Eclipse Standard-Mechanismen realisiert werden. Wie bereits erwähnt, entsprechen automatisch aus Schemas extrahierte Vorschläge zur Generierung von XML-Elementen oft nicht den Vorstellungen des Benutzers. Die Benutzer sind mit entsprechender Erfahrung viel eher in der Lage, typische Verwendungsmuster zu erkennen und zu definieren. Diese Tatsache macht sich der im folgenden vorgestellte Mechanismus zu Nutzen.

Templates

Templates sind vordefinierte Text- bzw. Code-Schablonen, die dem Benutzer das wiederholte Schreiben ähnlicher Konstrukte erleichtert. Um dies zu verdeutlichen, sehen wir uns kurz ein Beispiel aus der Eclipse Java IDE an:
Ein häufig benötigtes Code-Muster ist das Iterieren über die Elemente einer Liste. Dazu beginnen wir mit der Programmierung einer For-Schleife:

```
1  void foo(List names) {
2    for
```

Anstatt die Schleife nun aber auszuprogrammieren, betätigen wir eine spezielle Tastenkombination um anzudeuten, dass Eclipse dies übernehmen soll. Eclipse

präsentiert in einem kleinen Fenster Vorschläge für typische Code-Muster, und wir wählen das Iterieren über eine Liste aus. Eclipse generiert dann folgendes:

```
1  void foo(List names) {
2    for (Iterator iter = names.iterator(); iter.hasNext();) {
3      String name = (String) iter.next();
4
5    }
```

Der Cursor springt in Zeile 4 und man kann somit sofort den restlichen Code schreiben. Definiert werden Templates in einer eigenen Beschreibungssprache. Im wesentlichen handelt es sich um gewöhnliche Text-Muster, mit Ausnahme von Zeichenfolgen der Form ${name}. Diese Folgen entsprechen Variable die zur Laufzeit entweder automatisch, oder manuell durch den Benutzer durch anderen Text ersetzt werden.
Um zum Beispiel zurückzukehren, sehen wir uns die Definition des verwendeten Templates für das Iterieren über Listen an:

```
1  for (Iterator ${iterator} = ${collection}.iterator(); ${iterator}.hasNext(); ) {
2    ${type} ${element} = (${type}) ${iterator}.next();
3    ${cursor}
4  }
```

Das Template enthält 4 Variablen. `iterator`, `type`, `element` und `cursor`. Die ersten drei können vom Benutzer zur Laufzeit gewählt werden, `cursor` gibt die Cursor-Position nach dem Generieren des Templates an und wird vom Editor automatisch gesetzt.

Context

Einzelne Templates können in strukturierten Dokumenten nur an bestimmten Positionen sinnvoll eingesetzt werden. Im Java Fall sollten Templates, die Statements, wie die For-Schleife, erzeugen, nur dort vorgeschlagen werden, wo laut Grammatik Statements eingefügt werden können. Die Grammatik vieler Dokumente ist aber zu komplex um eine einfache Konfiguration der Templates durch den Benutzer zu ermöglichen. Deshalb ist es meist vorzuziehen das Dokument nur grob in wenige Abschnitte (z.B. Java-Code und Java-Kommentar) einzuteilen und bei der Auswahl der Templates auf die Intelligenz des Benutzers zu vertrauen. Diese groben Gültigkeitsbereiche für verschiedene Templates werden als Kontexte bezeichnet.

3.3.3 Umsetzung für XML-Author

Wir haben gesehen wie Templates den Benutzern von Editoren die Arbeit erleichtern können. Nun wollen wir uns ansehen, wie dieses Konzept mit Hilfe der Eclipse Plattform umgesetzt werden kann.

Plugin.xml

Die grobe Struktur und Funktionalität jedes Plugins für Eclipse wird durch eine XML Datei beschrieben. Dort wird alles beschrieben, was Eclipse, vor der eigentlichen Verwendung, für die Darstellung und Integration der Plugins in die Plattform benötigt. Der Code der Plugins muss somit erst dann geladen werden, wenn diese durch den Benutzer aktiviert werden.
Ein zentrales Konzept von Plugin-Definitionen sind sogenannte „Extension Points“. Plugins können damit definieren wie und wo sie erweiterbar sind. Andere Plugins können diese verwenden um konkrete Erweiterungen zu erstellen. Einer der Extension Points des Standard Text-Editors kann dazu verwendet werden Templates zu definieren. Dies erfolgt deklarativ mittels XML:

```
1  <extension point="org.eclipse.ui.editors.templates">
2    <contextType
3      class="org.eclipse.jface.text.templates.TemplateContextType"
4      name="Root" id="at.jku.ssw.xmlauthor.root" />
5    <contextType
6      class="org.eclipse.jface.text.templates.TemplateContextType"
7      name="FMV2-Feature" id="at.jku.ssw.featuremodel2.feature" />
8    ...
9    <include file="templates/fm2.xml" />
10 </extension>
```

Zunächst wird ein bestimmter Erweiterungspunkt durch seine ID referenziert (Zeile 1), danach werden die gewünschten Erweiterungen definiert. Im vorliegenden Beispiel teilen wir Eclipse zunächst mit, dass 2 neue Kontexte definiert werden sollen (Zeile 2-7). Anschließend geben wir die Datei mit den vordefinierten Templates bekannt. Das folgende Beispiel zeigt die Definition eines Templates für die Verwendung in Eclipse:

```
1  <template
2    id="Solitary"
3    name="Solitary"
4    context="at.jku.ssw.featuremodel2.feature"
5    enabled="true">
6    <![CDATA[
7      <Solitary cardMin="${min}" cardMax="${max}" >
8        <Feature id="${id}" name="${name}"> </Feature>
9      </Solitary>
10   ]]>
11 </template>
```

Jedes Template wird durch ein XML-Element definiert. Für unser Plugin ist vor allem der Kontext, in dem es gültig ist (Zeile 4) und der Template-Code an sich in Zeile 7-9 interessant.

Text-Editor

Text-Editoren sind ein zentraler Bestandteil von Entwicklungsumgebungen. Es ist somit nicht verwunderlich, dass Eclipse bereits einen einfachen Text-Editor zur Verfügung stellt, der nach Belieben erweitert werden kann. Zur Grundausstattung des Standard Text-Editors gehören Navigation, Undo, Suchen, Ersetzen, Tastenkürzel und vieles mehr. Typische Erweiterungen des Editors können durch Subklassen von SourceViewerConfiguration festgelegt werden.

SourceViewerConfiguration

SourceViewerConfiguration stellt die Standardkonfiguration des einfachen Text-Editors dar. Subklassen können diese durch Überschreiben von Methoden ändern. Typische Funktionalität die damit festgelegt werden kann ist z.B. Syntax Highlighting, Verhalten bei Doppelklicks, Automatische Formatierung von Dokumenten und spezielle Assistenten zur automatischen Vervollständigung von Inhalten an der aktuellen Cursor-Position (Content Assistant).

IContentAssistant / IContentAssistProcessor

Templates stellen eine Spezialform von Vorschlägen eines Content-Assistenten dar. Neben Vorschlägen von Templates können Content-Assistenten aber auch andere Vorschläge generieren, z.B. mittels einer Symbol-Tabelle oder aus den Informationen eines XML-Schemas. Den Kern dieses Mechanismus stellt folgende Methode dar:

```
ICompletionProposal[] computeCompletionProposals(ITextViewer viewer, int offset);
```

Die Methode soll für einen bestimmten Text-Editor und eine bestimmte Cursor-Position Vorschläge für Vervollständigungen berechnen. Wir haben gesehen, wie sowohl verschiedene Kontexte, als auch zugehörige Templates in der Datei „plugin.xml“ definiert werden können. Bevor Vorschläge generiert werden können, muss für die aktuelle Position in einer XML-Datei der aktive Kontext bestimmt werden. Ähnlich wie Grammatiken von Programmiersprachen sind Schemas für XML-Dateien oft komplex und schwer zu handhaben. Deshalb habe ich auf den enormen Laufzeitaufwand verzichtet für jede Position XML, Schema, und Template zu parsen, um festzustellen welche Templates syntaktisch möglich sind. Die Baum-Struktur von XML-Dateien erlaubt eine recht natürliche Abbildung von Cursor-Positionen auf bestimmte Kontexte. Für jede Position kann auf einfach Weise das umgebende XML-Element festgestellt werden und dieses kann wiederum einem Kontext zugeordnet werden.

Die Vorgehensweise ist also wie folgt:

1. Feststellen der aktuellen Position im Dokument
2. Suche nach dem umgebenden XML-Element
3. Feststellen welcher Kontext in diesem Element aktiv ist
4. Feststellen welche Templates für diesen Kontext definiert wurden
5. Präsentation der Vorschläge

Die Punkte 1 und 5 werden von Eclipse erledigt, somit verbleiben die Punkte 2 bis 4. Punkt 2 kann durch Suche in der internen Repräsentation der XML-Datei (DOM) erledigt werden. Interessant ist vor allem Punkt 3. Die Abbildung eines XML-Elementen auf einen Kontext ist natürlich vom jeweiligen Schema abhängig. Die Konfiguration erfolgt über ein eigene XML-Datei, das somit, ähnlich wie die Definition der Kontexte und Templates, ohne Neukompilation auf ein bestimmtes Schema angepasst werden kann. Somit ist es denkbar, XML-Author mit einer geänderten Konfigurations-Datei auch für das Editieren von XML-Daten mit anderen Schemas einzusetzen
In der folgenden Beispielkonfiguration wird ein Template-Context definiert, der innerhalb von bestimmten Constraints vorherrscht.

```
1  <context id="at.jku.ssw.featuremodel2.constraint" name="Constraint">
2    <element>Mandatory</element>
3    <element>Optional</element>
4    <element>XorGroup</element>
5    <element>Requires</element>
6    <element>Excludes</element>
7  </context>
```

Der Context `at.jku.ssw.featuremodel2.constraint` wird von den Elementen `Mandatory`, `Optional`, `XorGroup`, `Requires` und `Excludes` aufgespannt. Immer dann, wenn eines dieser Elemente das umgebende der aktuellen Cursor-Position ist, werden Templates mit diesem Kontext vorgeschlagen. Befinden wir uns beim Editieren also zum Beispiel innerhalb eines Mandatory Constraints, werden alle Templates vorgeschlagen die für den context at.jku.ssw.featuremodel2.constraint definiert wurden.
Der letzte noch verbleibende Schritt, das Suchen nach bestimmten Templates, ist wieder relativ einfach. Die Templates werden von Eclipse verwaltet und können nach Context-ID abfragt werden. Bevor die Templates dem Benutzer präsentiert werden, können noch jene aussortiert werden, deren Name nicht mit den bereits getippten Zeichen des aktuellen Wortes an der Cursor-Position übereinstimmt. Das heißt, nachdem `Ex` getippt wurde, sollten nur mehr Templates mit diesem Präfix (z.B. `Excludes` oder `Expression`) vorgeschlagen werden.

Weitere Erweiterungen

Eine weitere kleine Erweiterung wurde für die Outline-View erstellt, die als Übersicht und zur schnellen Navigation in einem Dokument dienen kann. Die Standard-Outline von XML-Author ist für allgemeines XML konzipiert und unterscheidet daher in der Darstellung nur zwischen Elementen und Attributen (siehe Abbildung 3.2). Für bestimmte XML-Schemas ist es aber wünschenswert unterschiedliche Symbole (Icons) und Texte (Labels) für unterschiedliche XML-Elemente anzuzeigen. Die Konfiguration der Outline View erfolgt wieder über ein XML-Datei.

```
1  <icon file="fm/feature.gif">
2    <element>FeatureModel</element>
3    <element>Feature</element>
4  </icon>
5  <label>
6    <provider>
7      <attribute>id</attribute>
8    </provider>
9    <element>Featuremodel</element>
10   <element>Feature</element>
11 </label>
```

Im vorliegenden Beispiel werden Icon und Label für die XML-Elemente `FeatureModel` und `Feature` festgelegt. Beide Elemente besitzen in der Outline das Icon aus der Datei *feature.gif*. Beide Element besitzen ein verpflichtendes XML-Attribut `id`. Dessen Inhalt wird neben dem Icon angezeigt und erlaubt somit dem Benutzer das schnelle Anspringen bestimmter Features.
Der Screenshot in Abbildung 3.3 zeigt die erstellten Erweiterungen in Aktion. Im Editor sind die Vorschläge der Auto-Completion zu sehen. In der Outline sind die benutzerdefinierten Icons und die aus dem Dokument generierten Labels zu sehen.

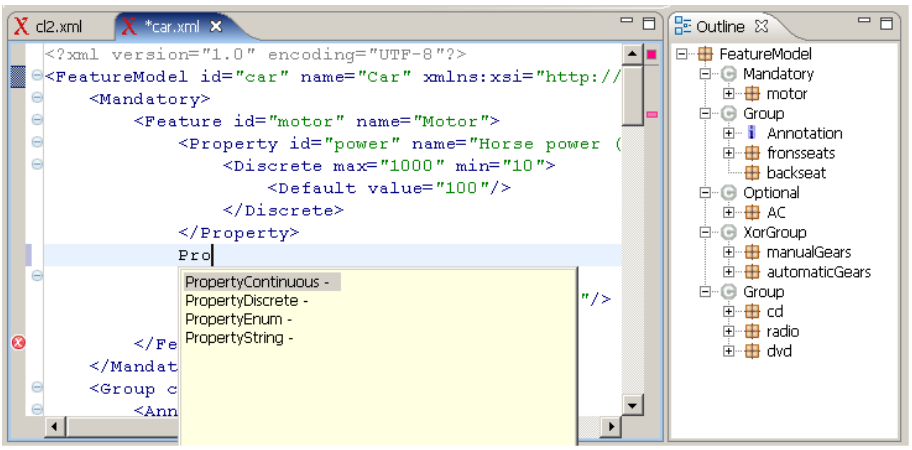


Abbildung 3.3: Erweiterter XML-Author

3.4 FeatureKing

Im letzten Abschnitt haben wir gesehen, wie Feature-Modelle auf einfache Weise erstellt werden können. Ohne weitere Verarbeitung ist der Wert eines in XML vorliegenden Feature.Modells beschränkt. Erst durch automatisierte Weiterverarbeitung durch weitere Werkzeuge, kann Nutzen aus dem einheitlichen, maschinenlesbaren Format gezogen werden. Nach der Erstellung eines Feature-Modells kann damit ein Feature-Auswahlprozess gesteuert werden.
Der einfachste und zugleich mächtigste Auswahlvorgang sieht wie folgt aus.

1. Das System präsentiert alle Features, Beschreibungen, Eigenschaften und Abhängigkeiten.
2. Der Benutzer wählt selbstständig alle gewünschten Features aus.
3. Das System überprüft die Konsistenz der Auswahl.

Dieser Prozess ist zwar mächtig, da der Benutzer zu jedem Zeitpunkt die volle Kontrolle über seine Auswahl behält, aber auch recht mühsam, da er jedes gewünschte oder abhängige Feature selbst wählen und jeden Konflikt selbst auflösen muss. Weitere Probleme dieser Vorgehensweise sind wie folgt:

- Bei komplexen Modellen fällt es dem Benutzer schwer den Überblick zu behalten.
- Bei komplexen Abhängigkeiten fällt es dem Benutzer schwer die Konsistenz des Modelles aufrecht zu halten.
- Mehrere Iterationen der Schritte 2) und 3) bis zum Erreichen einer konsistenten Auswahl sind für den Benutzer sehr mühsam.

Um diese Probleme in den Griff zu bekommen, sollte das Werkzeug für die Feature-Auswahl folgende Eigenschaften aufweisen:

- Übersichtliche und kompakte Darstellung von Modell und bisheriger Auswahl
- Einschränkung der Sichtweise auf bestimmte Teile des Modells
- Folgen von Referenzen auf andere Features
- Hinweise auf noch zu treffende Entscheidungen
- Hinweise auf Konsistenzverletzungen in momentaner Auswahl
- Automatische Konsistenzerhaltung bei hierarchischen Feature-Constraints
- Übersicht über bisherigen Auswahlprozess in Form einer Historie
- Übersicht über bisherige Auswahl in Form von Statistiken

In den folgenden Abschnitten wir das erstellte Werkzeug vorgestellt und gezeigt, wie damit die beschriebenen Anforderungen erfüllt werden konnten.

3.4.1 Allgemeines

Die Featureauswahl wurde in das bereits in Abschnitt 1.1.3 vorgestellte Werkzeug *FeatureKing* integriert. Die ursprünglich in FeatureKing enthaltene Auswahl basierte auf FODA-ähnlichen Feature-Modellen, die mit einem graphischen Editor erstellt wurden. Die in diesem Projekt erstellte Featureauswahl ersetzt die bisherige und basiert komplett auf Constraint-basierten Feature-Modellen.

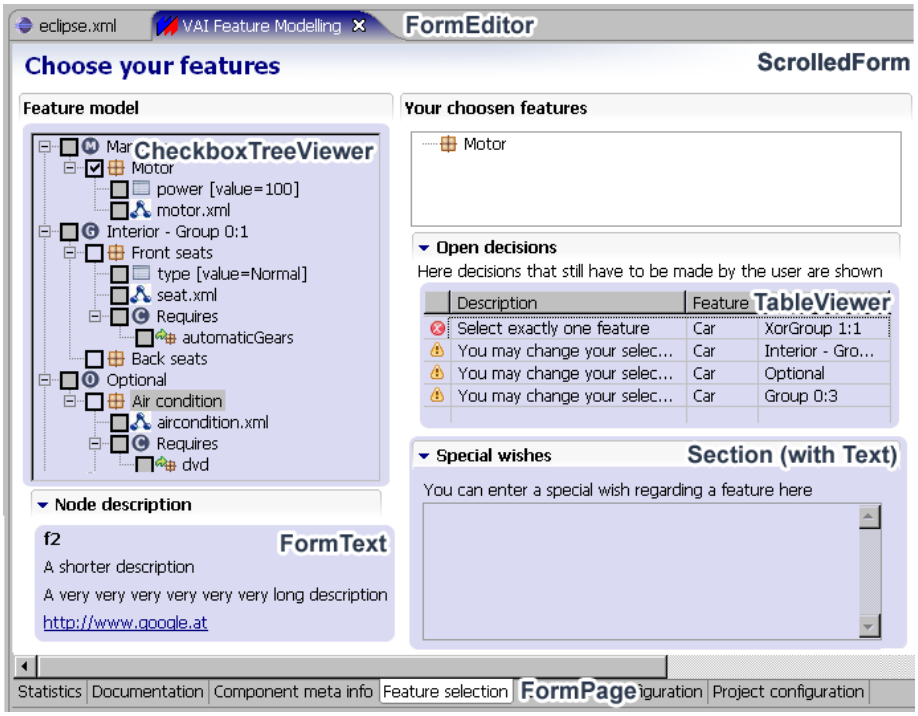


Abbildung 3.4: Verwendete SWT und JFace Darstellungs-Komponenten

3.4.2 Darstellung

Die Darstellung der Feature-Modelle erfolgt über eine hierarchische Baum-Ansicht. Diese eignet sich gut für die Darstellung hierarchischer Daten wie Dateisysteme, Source-Code, XML-Dateien oder Feature-Modelle. Eclipse stellt dafür bereits ein einfach zu konfigurierendes Anzeige-Element (Widget) zur Verfügung.
Der Inhalt eines Tree Widget wird durch ein Modell beschrieben. Das Modell besteht aus Knoten, die wiederum aus mehreren anderen Knoten bestehen können. Für jeden Knoten muss Icon und Text für deren Darstellung festgelegt werden.
Jeder Baum kann auf mehrere Arten dargestellt werden. Eine Möglichkeit ist, das vollständige Modell Knoten für Knoten aufzubauen und dann vom Widget anzeigen zu lassen. Dies hat den Nachteil, dass große Modelle vor der Darstellung immer

komplett neu generiert werden müssen.
Eine zweite Möglichkeit besteht darin, nur jene Knoten des Modells zu generieren, die auch angezeigt werden müssen. Dies kann mit Hilfe von Callbacks und folgenden Schnittstellen erreicht werden.

```
1  public interface ITreeContentProvider {
2    /** Returns the child elements of the given parent element. *
3    public Object[] getChildren(Object parentElement);
4    /** Returns the parent for the given element. */
5    public Object getParent(Object element);
6    /** Returns whether the given element has children. */
7    public boolean hasChildren(Object element);
8  }
```

Mit Hilfe eines `TreeContentProviders` wird das dem Widget zu Grunde liegende Modell implizit beschrieben. Da für jeden Knoten im Modell Subknoten und Vaterknoten bestimmt werden können, kann daraus bei Bedarf auch das komplette Modell rekonstruiert werden.

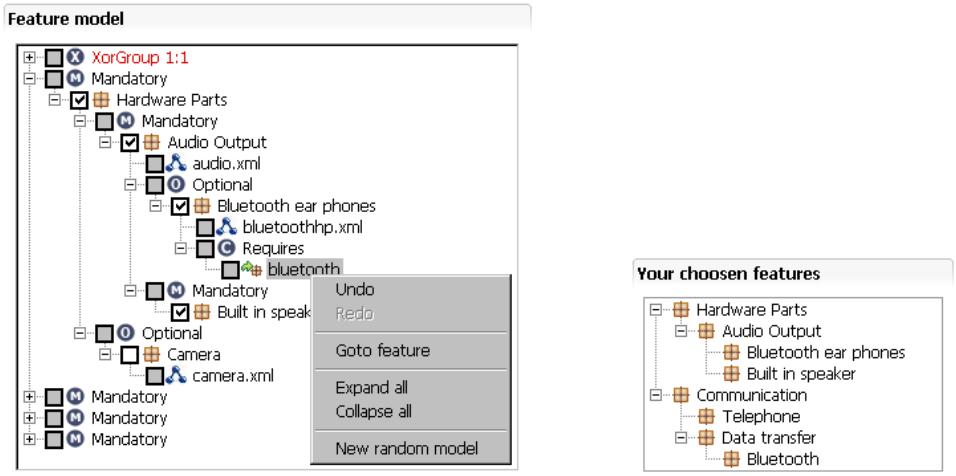
```
1  public interface ILabelProvider {
2    /** Returns the image for the label of the given element. */
3    public Image getImage(Object element);
4    /** Returns the text for the label of the given element. */
5    public String getText(Object element);
6  }
```

Mit Hilfe eines `LabelProviders` werden Icon und Text für die Darstellung eines Knotens im Modell festgelegt. Die Berechnung muss nur für jene Knoten stattfinden, die auch angezeigt werden.
`TreeContentProvider` und `LabelProvider` werden der TreeView bei der Initialisierung mitgeteilt, anschließend ist das Widget voll funktionsfähig. Abbildung 3.6 zeigt zwei Screenshots der Feature-Bäume in Eclipse. Links ist ein Feature-Modell zu sehen, das die interaktive Selektion von Features durch den Benutzer ermöglicht. Rechts ist die Ansicht für momentan ausgewählte Features zu sehen. Zusätzlich zur Baumansicht existiert für momentan ausgewählte Features eine Detailansicht. In dieser werden zusätzliche Informationen zur Dokumentation der Features, sowie die in Abschnitt 3.5.5 beschriebene Historie dargestellt.

3.4.3 Konsistenzprüfung

Modelle für realistische Produktlinien bestehen aus einer Vielzahl von Features und Constraints. Umso komplexer ein Modell, umso schwieriger ist es für den Benutzer die Konsistenz seiner Auswahl manuell zu bewahren. Eine Möglichkeit den Benutzer bei seiner Auswahl zu unterstützen, besteht darin, auf momentan verletzte

Abbildung 3.5: Detailansicht für Features



(a) (b)

Abbildung 3.6: (a) Feature-Selektion, (b) Selektierte Features

Constraints hinzuweisen. Dies Menge verletzer Constraints kann einfach bestimmt werden, indem die Bedingung jedes aktiven Constraint in der aktuellen Auswahl von Features überprüft wird. Aktive Constraints sind all jene, die global für das Feature-Modell gelten oder Teil eines selektierten Features sind. Jeder Constraint kann aufgrund seines eigenen Typs und den abhängigen Features feststellen, ob er für die aktuelle Auswahl erfüllt ist. Ein Group-Constraint kann z.B. auf folgende Weise feststellen, ob er verletzt ist:

```
1  public boolean isConstraintBroken() {
2    int n = numSelectedSubFeatures();
3    Cardinality c = getCardinality();
4    return n < c.getMin() || n > c.getMax();
5  }
```

Die Darstellung verletzter Constraints wird im FeatureKing auf zwei verschiedene Arten realisiert: Im Feature-Baum werden verletzte Constraints rot markiert, umgesetzt durch eine einfache Modifikation des `LabelProviders`. Zusätzliche werden alle verletzen Constraints mit einer kurzen Beschreibung in einer eigenen Tabelle

zusammengefasst. Die Tabelle erlaubt das direkte Anspringen der Constraints im Feature-Baum und erlaubt somit das schnelle Lösen von Konflikten.

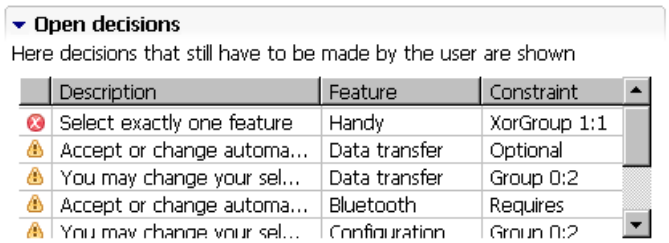


Abbildung 3.7: Verletze Constraints und offene Entscheidungen

Selektionsprozess

Da auch Variationspunkte wie Xor- oder Or-Gruppen durch Constraints beschrieben werden, kann die Tabelle auch den Selektionsprozess leiten. Zu Beginn sind keine Features selektiert und meist einige globale Constraints verletzt. Nun wird eine Entscheidung nach der anderen getroffen um einzelne Constraints zu erfüllen. Viele der Entscheidungen führen, z.B. durch Selektion von Features, zu weiteren Constraint-Verletzungen. Sobald der Prozess konvergiert und alle Constraints erfüllt sind, ist ein fertiges Produkt spezifiziert. Entscheidungen müssen nicht endgültig sein, da gewisse Constraints (z.B. optional oder Xor-Gruppe) auf unterschiedliche Weise erfüllt werden können. Constraints die unter Umständen noch eine alternative Auswahl an Features erlauben, werden in der Tabelle als Hinweis vermerkt.

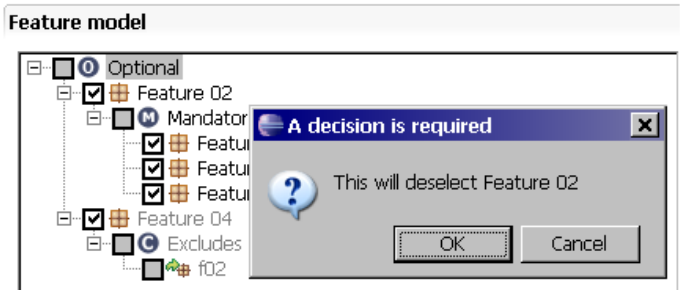
3.4.4 Constraint Propagation

Wie in Abschnitt 2.2 erläutert, kann Constraint Propagation dazu verwendet werden, bestimmte Features automatisch zu selektieren. Im FeatureKing werden nur jene Constraints automatisch propagiert, die keine alternativen Lösungsmethoden besitzen. Diese umfassen:

- Subfeatures: Wird das umgebende Feature deselektiert, müssen auch alle Subfeatures deselektiert werden, wird ein Subfeature selektiert muss auch das umgebende selektiert werden.
- Mandatory Features: Sobald ein Feature selektiert wird, können auch alle verpflichten Subfeatures selektiert werden.
- Required Features: Benötigte Features in anderen Teilen des Baums müssen ebenso selektiert werden.

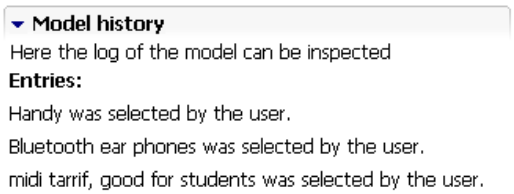
- Excluded Features: Schließen sich zwei Features aus, muss bei der Selektion des einen, das andere deselektiert werden.

Automatisch selektierte oder deselektierte Features werden im Feature-Baum besonders gekennzeichnet, um den Unterschied zu explizit vom Benutzer ausgewählter Features zu verdeutlichen. Zusätzlich werden sie in der in Abschnitt 3.4.3 beschrieben Tabelle vermerkt.
Da in komplexen Modellen automatische Deselektionen von referenzierten Teilbäumben leicht zu übersehen und unter Umständen vom Benutzer ungewünscht sind, werden Constraints mit Querverweisen (Requires, Excludes) nur nach Absprache mit dem Benutzer automatisch propagiert:



3.4.5 Historie und Undo

Die Feature-Selektion ist ein stark interaktiver Prozess zwischen dem Benutzer und dem dahinterliegenden System, bei dem Fehler und Unsicherheiten nicht zu vermeiden sind. Deshalb wurde eine Historie entwickelt, die Aktionen des Benutzers, sowie deren Konsequenzen, aufzeichnet. Die Historie kann sowohl pro Feature (siehe Abbildung 3.5), als auch zusammengefasst für das gesamte Modell angezeigt werden, wie im folgenden Screenshot:



Um Benutzeraktionen rückgängig zu machen, oder wiederherzustellen, wurde ein einfacher Undo-Mechanismus implementiert. Die Bedienung erfolgt analog zu anderen Eclipse-Applikationen über das *Edit*-Menü, Tastenkürzel oder Context-Menüs. Um bei Standardaktionen wie *Undo* oder *Redo* in Eclipse eigenen Code auszuführen, müssen sogenannte *Actions*, Objekte von Klassen mit einer überschriebenen `run()`-Methode, beim Framework registriert werden. Dies geschieht in plugin.xml und sieht folgendermaßen aus:

```
1  <extension point="org.eclipse.ui.editors">
2    <editor ...
3       class="vai.featuremodel.plugin.editors.FeatureKingEditor"
4       contributorClass="vai.featuremodel.plugin.editors.MyActionContributer">
5    </editor>
6  </extension>
```

In `MyActionContributer` werden dann die eigenen Aktionen folgendermaßen registriert:

```
1  bar.setGlobalActionHandler(ActionFactory.UNDO.getId(), new UndoAction(model));
2  bar.setGlobalActionHandler(ActionFactory.REDO.getId(), new RedoAction(model));
```

Die auszuführenden Aktionen hängen vom gerade aktiven Editor ab. Immer dann, wenn der FeatureKing der aktive Editor im Vordergrund von Eclipse ist, wird beim Auswählen von Undo im Menü oder Eingabe des entsprechende Tastenkürzel die run()-Methode unserer registrierten Aktion ausgeführt.
Die logische Umsetzung basiert auf dem *Command-Pattern*. Benutzeraktionen werden dabei als Objekte von Klassen beschrieben, die das `Command`-Inteface implementieren. Das Interface legt Methoden für das Ausführen, Rückgängigmachen und Wiederherstellen von Aktionen fest. Das Modell verwaltet alle Objekte in einer Liste und führt die entsprechenden Aktionen darauf aus. Da die Selektion eines Features durch Constraint Propagation weitreichende Konsequenzen haben kann, speichert der `SelectionCommand` vor seiner Ausführung den Zustand des Modells. Dieser Zustand kann dann später bei einem Undo wiederhergestellt werden.

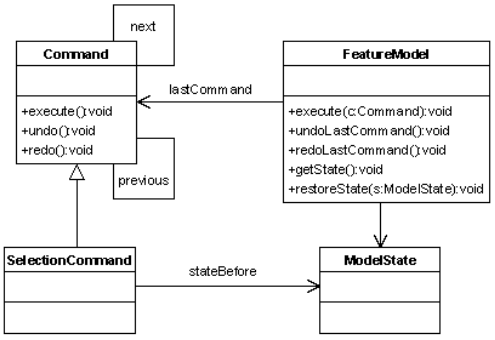


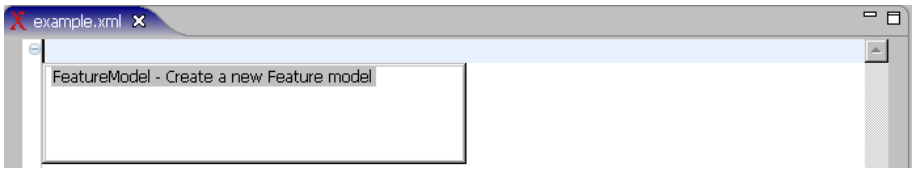
Abbildung 3.8: Klassendiagramm - Historie und Undo

3.5 Beispiel

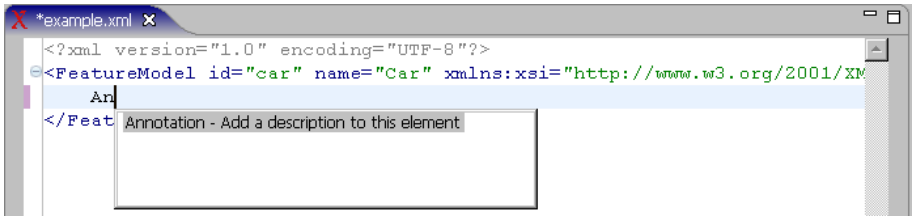
Im folgenden Abschnitt wird an Hand eines größeren Beispiels der Umgang mit den entwickelten Werkzeugen gezeigt. Zunächst wird mit Hilfe von XML-Author ein Feature-Modell erstellt. Anschließend wird für dieses Modell im FeatureKing eine Feature-Selektion durchgeführt.

3.5.1 XML-Author

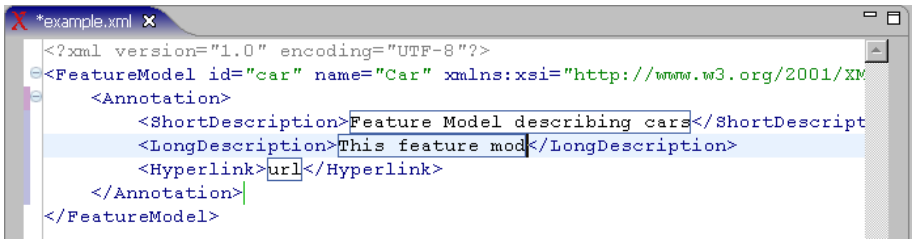
Um mit XML-Author ein Feature-Modell zu erstellen starten wir Eclipse und erstellen eine generische XML-Datei. Nach dem Öffnen der Datei mit XML-Author erscheint ein leeres Dokument, das unser Modell enthalten wird.
Anstatt nun zu versuchen, Syntax und Struktur von XML, sowie das Schema für unsere Feature-Modelle zu rekapitulieren, nehmen wir den Weg des langjährig verwöhnten Eclipse-Benutzers: Strg-Leerzeichen stoßt den automatischen Inhaltsassistenten (Content-Assist) an, und bittet um Vorschläge, wie fortgefahren werden kann:



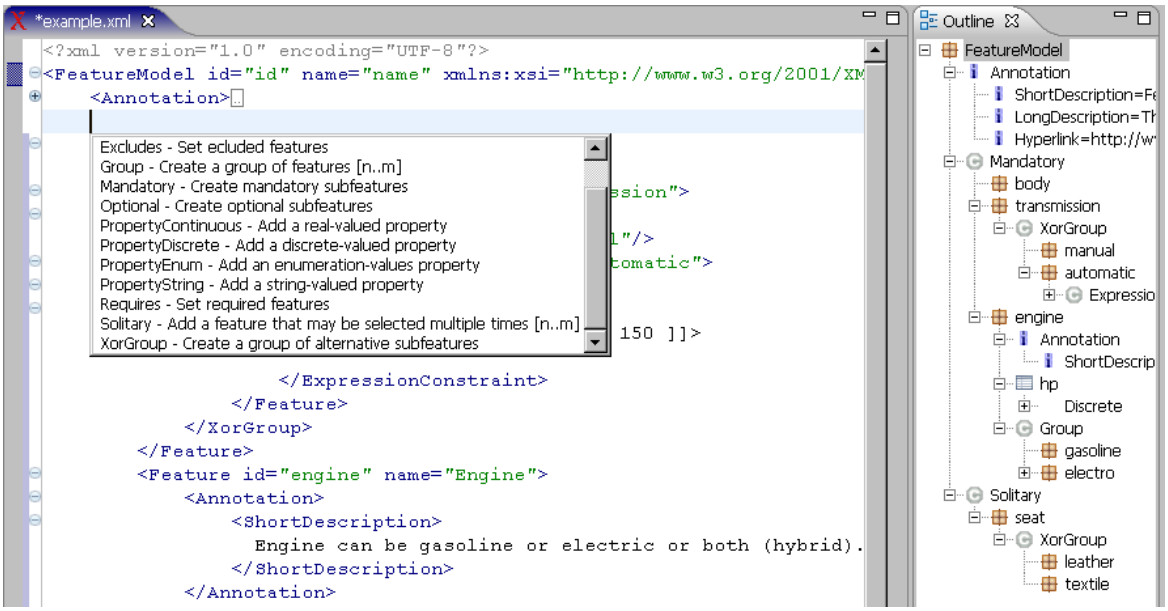
Es wird zwar nur ein einziger Vorschlag präsentiert, aber wie nicht anders erwartet, passt uns dieser sehr gut. Nach der Auswahl des präsentierten Templates wird das Gerüst für ein Feature-Modell in XML erstellt. Dieses besteht aus XML-Header und dem Wurzelelement `FeatureModel`. Das Wurzelelement legt neben der vom Benutzer gewählten ID und dem Namen des Modells auch das verwendete Schema fest.
Als nächsten Schritt wollen wir guten Willen zeigen, und etwas Dokumentation für unser Modell schreiben. Wir erinnern uns düster, dass dies mit `Annotation`-Elementen erfolgen kann.



Der Inhaltsassistenten bestätigt diesen Glauben und präsentiert ein entsprechendes Template.



Nach der Auswahl werden gleich mehrere Elemente eingefügt, mit Tab können wir zwischen den einzelnen vordefinierten Feldern (Variablen) wechseln.
Analog wird nun der Rest des Modells erstellt. Sobald das Dokument eine bestimmte Größe erreicht hat, können Editor Features wie die Outline (rechts) oder das Ausblenden einzelner Elemente im Text (siehe Annotation) zur Beibehaltung der Übersicht beitragen.

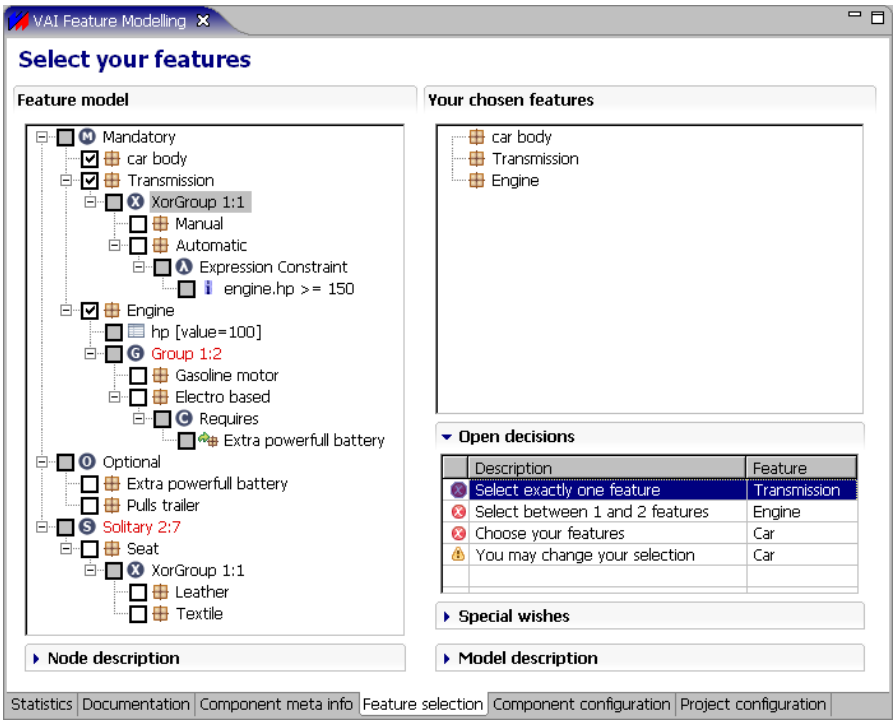


Ein mit vielen guten Templates konfigurierter XML-Author ermöglicht es Feature-Modelle schnell zu erstellen, auch wenn das genau Schema und die möglichen Elemente für bestimmte Positionen im Dokument vom Benutzer noch nicht verinnerlicht wurden.
Die vollständige XML-Datei für das Feature-Modell in diesem Beispiel ist in Anhang A zu finden.

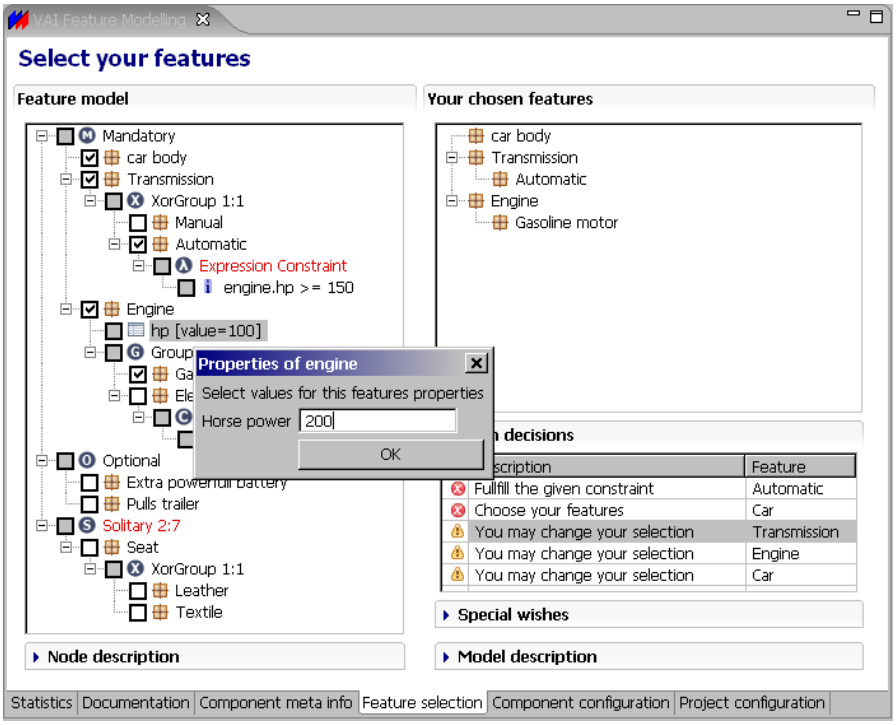
3.5.2 FeatureKing

Mit dem eben erstellen Feature-Modell kann nun eine Feature-Selektion durchgeführt werden. Dazu wird der FeatureKing gestartet und die XML-Datei mit

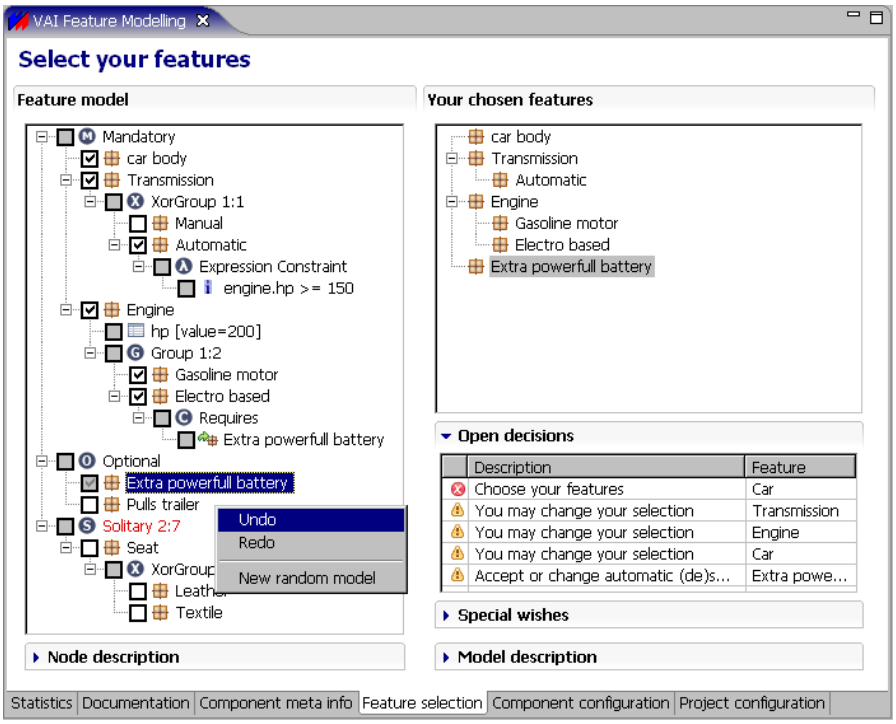
dem Modell geladen. Nach dem Übersetzen der XML-Daten in eine interne Repräsentation als Objektgraph kann der Feature-Baum gezeichnet werden. Der folgende Screenshot zeigt den FeatureKing unmittelbar nach dem Laden des Modells:



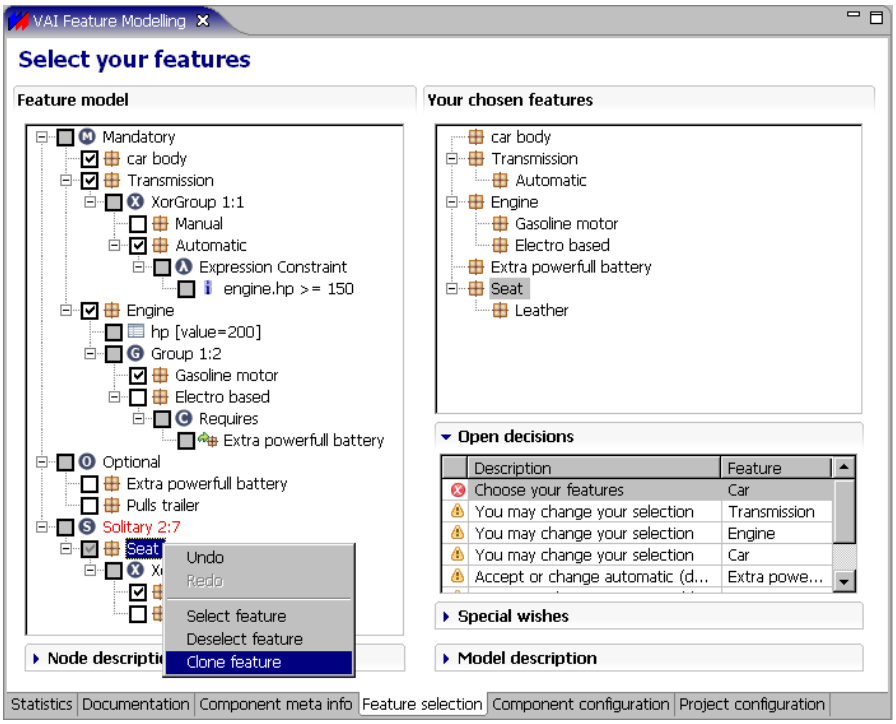
Die verpflichtenden Subfeatures *car body*, *Transmission* und *Engine* wurde bereits automatisch selektiert und verletzte Constraints werden im Baum farblich markiert, sowie in der *Open Decisions* Tabelle aufgelistet.



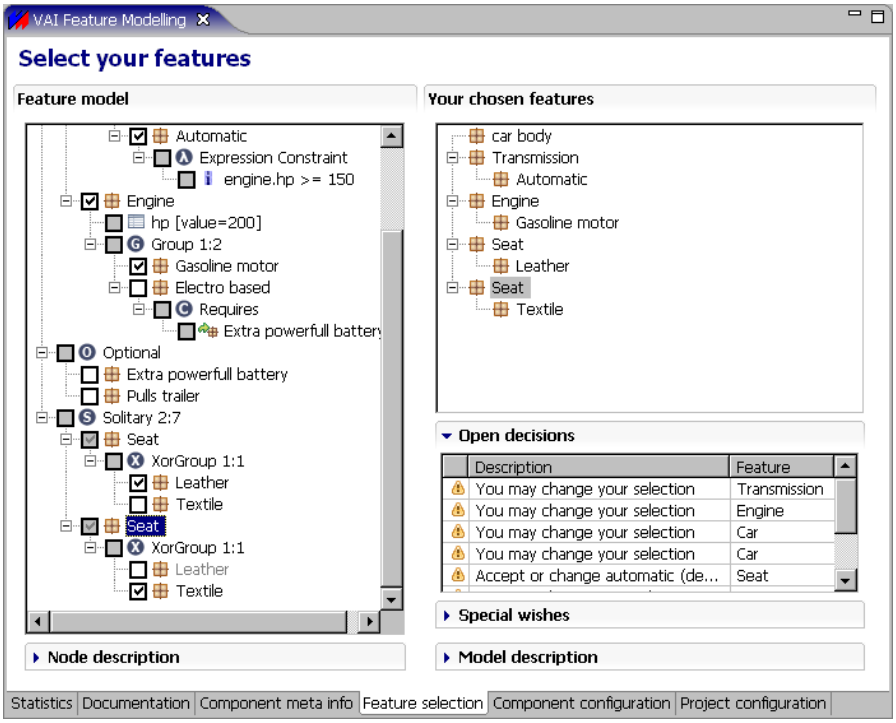
Wir entscheiden uns für einen Benzinmotor (*Gasoline motor*) und eine automatische Schaltung (*Automatic*) und können damit 2 zuvor verletzte Constraints (*XorGroup*, *Group*) erfüllen.
Durch die Auswahl der automatischen Schaltung wird nun aber der *Expression constraint* verletzt. Dieser setzt für die Automatik mindestens 150 Pferdestärken voraus. Die Leistung in Pferdestärken ist ein Property (*hp*) des Features *Engine*, das standardmäßig den Wert 100 annimmt. Um den Constraint zu erfüllen, und später mehr Spaß an unserem neuen Auto zu haben, rüsten wir auf 200 PS auf.
Da wir gerne auf dem neuesten Stand der Technik sein wollen, wählen wir zusätzlich den Elektormotor und stellen fest, dass dadurch auch automatisch die besonders starke Batterie selektiert wurde.



Hochleistungs-Batterien erscheinen uns zu anfällig für Defekte und zu teuer in der Wartung, deshalb überdenken wir unsere Auswahl und entscheiden uns letztlich auf den Elektromotor verzichten zu können. Dazu machen wir unsere letzte Auswahl wieder rückgängig.
Als letzten Schritt wollen wir unser Auto noch mit 2 Sitzen ausstatten. Da mehrfach selektierbare Features beliebig große Kardinalitäten definieren können, ist es nicht sinnvoll vorab mehrere selektierbare Instanzen anzuzeigen.



Um einen zweiten Sitz zu selektieren, erzeugen wir einfach eine Kopie des ersten Sitzes im Baum. Die Kopie kann unabhängig konfiguriert werden, in unserem Fall lässt unser Budget leider nur eine Lederaustattung zu.



Damit ist auch schon eine Endkonfiguration und damit das Ende dieses Beispiels erreicht.

Kapitel 4

Zusammenfassung

Im Rahmen dieses Projektes wurden zwei Werkzeuge für Constraint-basierte Feature-Modellierung entwickelt. Die Erweiterung des Open Source XML-Editors XML-Author um Templates und eine konfigurierbare Outline-View machen diesen zu einem nützlichen Werkzeug für das Erstellen und Bearbeiten von XML-Daten mit bestimmten, vorgegebenen Schemas. Benutzer können den Editor mit geringem Aufwand inkrementell um eigene Templates erweitern und somit immer wiederkehrende Kombinationen von Elementen und Attributen mit geringem manuellen Aufwand erstellen. Die Übersicht der Darstellung von XML als Text ist trotz Ein- und Ausklappen von Elementen manchmal nicht so anschaulich wie eine graphische Repräsentation als Baum. Als eine zukünftige, wenn auch sehr ehrgeizige Erweiterung von XML-Author ist somit eine zusätzliche graphische Darstellung, ähnlich jener für XML-Schemas in anderen Editoren (Abbildung 2.2) denkbar.
Der FeatureKing ist ein ehrgeiziges Projekt, Feature-Modellierung für die automatische Konfiguration von Produktlinien einzusetzen. Der FeatureKing wurde im Rahmen dieser Arbeit mit einer neuen Feature-Selektion erweitert. Die Auswahl basiert nun auf Constraint-basierter Feature-Modellierung und unterstützt Constraint Propagation. Durch die Unterstützung eines inkrementellen Auswahlprozesses und der Implementierung eines einfachen Undo-Mechanismus, konnte die Benutzbarkeit des Werkzeugs weiter gesteigert werden. Die entwickelte Feature-Selektion ist nur ein kleiner Bestandteil des FeatureKings, dennoch sind dafür noch viele zusätzliche Erweiterungen denkbar. Eine benutzerfreundlichere Oberfläche und eventuell „intelligente“ Software-Assistenten zur Unterstützung des Auswahlprozesses könnten den Wert des Werkzeugs weiter erhöhen. Da dieses Gebiet derzeit noch relativ unerforscht ist, könnten Usability-Experten und Experimente mit Testpersonen wahrscheinlich am ehesten Aufschluss auf erfolgsversprechende Ansätze geben.

Literaturverzeichnis

[1] K. Czarnecki and M. Antkiewicz. *FeaturePlugin: Feature modeling plug-in for Eclipse.* http://www.swen.uwaterloo.ca/~kczarnec/etx04.pdf.

[2] Krzysztof Czarnecki, Chang Hwan Peter Kim. *Cardinality-Based Feature Modeling and Constraints: A Progress Report* http://softwarefactories.com/workshops/OOPSLA-2005/Papers/Czarnecki.pdf.

[3] The Spring Framework http://www.springframework.org/

[4] The Compiler Generator Coco/R http://www.ssw.uni-linz.ac.at/Coco/

[5] The Eclipse Projkect http://www.eclipse.org

[6] Java Development Tools for Eclipse http://www.eclipse.org/jdt

[7] Graphical Editor Framework for Eclipse http://www.eclipse.org/gef

[8] The Eclipse Project in Wikipedia http://en.wikipedia.org/wiki/Eclipse_(computing)

[9] Extensible Markup Language (XML) http://en.wikipedia.org/wiki/XML

[10] XML Author http://www.svcdelivery.com/xmlauthor/

[11] Krzysztof Czarnecki. *Generative Programming. Methods, Tools and Applications.* Addison Wesley, 2000

[12] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, A. Spencer Peterson: *Feature-Oriented Design Analysis (FODA) Feasibility Study.* TechnicalReport, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[13] K. Czarnecki, S.Helsen, U.Eisenecker: *Staged configuration using feature models.* Software product lines: Third International Conference SPLC 2004.

[14] W3C: XML Schema. `www.w3c.org`

[15] Alan K. Mackworth: *The Logic of constraint satisfaction.* University of Britisch Columbie. Constraint-Based Reasoning, MIT Press 1994.

[16] S. Russell, P. Norvig: Artificial Intelligence: A Modern Approach (chapter 5), Prentice Hall Series in Artificial Intelligence. (1995).

[17] P. Clements, L. Northrop: Software Product Lines: Practice and Patterns. Addison-Wesley 2002.

[18] D. Dhungana, R. Rabiser P. Grünbacher H. Prähofer, C. Federspiel, K. Lehner: Architectural Knowledge in Product Line Engineering: An Industrial Case Study Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA'06), Cavtat/Dubrovnik (Croatia), September 2006.

[19] G. Böckle, P.Knauber, K.Pohl, K.Schmid: *Software-Produktlinien: Methoden Einführung und Praxis*, dpunkt, 2004.

Anhang A - Beispiel für ein Feature-Modell in XML

```
<?xml version="1.0" encoding="UTF-8"?>
<FeatureModel id="id" name="name" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="fmodel.xsd">
  <Annotation>
    <ShortDescription>Feature Model describing cars</ShortDescription>
    <LongDescription>
      This simplyfied feature  model describes cars in an abstrat way.
    </LongDescription>
    <Hyperlink>http://www.czarnecky.com</Hyperlink>
  </Annotation>
  <Mandatory>
    <Feature id="body" name="car body"/>
    <Feature id="transmission" name="Transmission">
      <XorGroup>
        <Feature id="manual" name="Manual"/>
        <Feature id="automatic" name="Automatic">
          <ExpressionConstraint>
              <Code><![CDATA[ engine.hp >= 150 ]]></Code>
          </ExpressionConstraint>
        </Feature>
      </XorGroup>
    </Feature>
    <Feature id="engine" name="Engine">
      <Annotation>
        <ShortDescription>
          Engine can be gasoline or electric or both (hybrid).
        </ShortDescription>
      </Annotation>
      <Property id="hp" name="Horse power">
        <Discrete min="40" max="300"><Default value="100"/></Discrete>
      </Property>
      <Group cardMin="1" cardMax="2">
        <Feature id="gasoline" name="Gasoline motor"/>
        <Feature id="electro" name="Electro based" >
          <Requires><FeatureRef ref="battery" /></Requires>
        </Feature>
      </Group>
    </Feature>
  </Mandatory>
  <Optional>
    <Feature id="battery" name="Extra powerfull battery" />
    <Feature id="trailer" name="Pulls trailer"/>
  </Optional>
  <Solitary cardMin="2" cardMax="7">
    <Feature id="seat" name="Seat">
      <XorGroup>
        <Feature id="leather" name="Leather"/>
        <Feature id="textile" name="Textile"/>
      </XorGroup>
    </Feature>
  </Solitary>
</FeatureModel>
```

Anhang B - Grammatik für Expression-Conctraints

```
COMPILER Constraint
  public static Manager manager;
CHARACTERS
  letter    = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".
  digit     = "0123456789".
  cr        = '\r'.
  lf        = '\n'.
  newLine   = cr + lf.
  tab       = '\t'.
  stringChar  = ANY - '"' - '\\' - newLine.
TOKENS
  ident     = letter { letter | digit }.
  intCon    = digit {digit}.
  realCon   = ...
  stringCon = ...
COMMENTS FROM "/*" TO "*/"
IGNORE cr + lf + tab

PRODUCTIONS
Constraint =
  { VarDecl }
  Expression<out Expr expr>.
VarDecl =
  "VAR" ident "=" Expression<out Expr expr> ";".
Expression<out Expr expr> =
  AndExpr<out Expr lhs> { "||" AndExpr<out Expr rhs> }.
AndExpr<out Expr expr> =
  EqlExpr<out Expr lhs> { "&&" EqlExpr<out Expr rhs> }.
EqlExpr<out Expr expr> =
  RelExpr<out Expr lhs> { EqlOp<out BinaryOperator op> RelExpr<out Expr rhs> }.
RelExpr<out Expr expr> =
  AddExpr<out Expr lhs> { RelOp<out BinaryOperator op> AddExpr<out Expr rhs> }.
AddExpr<out Expr expr> =
  ["-"] MulExpr<out Expr left> { Addop<out BinaryOperator op> MulExpr<out Expr rhs> }.
MulExpr<out Expr expr> =
  Factor<out Expr left> { Mulop<out BinaryOperator op> Factor<out Expr rhs> }.
Factor<out Expr fac> =
    VarRef<out Expr var>
  | Literal<out Expr lit>
  | "(" Expression<out Expr exp> ")".
VarRef<out Expr var> =
    ident { "." ident }.
EqlOp<out BinaryOperator op> =
    "==" | "!=".
RelOp<out BinaryOperator op> =
    "<" | ">" | ">=" | "<=".
Addop<out BinaryOperator op> =
    "+" | "-".
Mulop<out BinaryOperator op> =
    "*" | "/" | "%".
Literal<out Expr lit> =
    intCon | realCon | stringCon | "true" | "false".

END Constraint.
```