ANKARA SCIENCE UNIVERSITY, RESEARCHER
VOL. 1, NO. 1, JULY 2021

Özellik Modellerinin Otomatik Dönüşümü

Ahmet Serkan KARATAŞ [1*]
[1] Department of Software Engineering, Ankara Science University, Ankara, Turkey; ORCID: 0000-0003-2480-8437
* Sorumlu Yazar: ahmet.serkan.karatas@ankarabilim.edu.tr

Received: 21 May 2021; Accepted: 1 July 2021

Öz

Özellik modelleri yazılım ürün hatları mühendisliği alanında değişkenlik yönetimi için kullanılan en popüler araçlardan biridir. Bir özellik modeli, bilhassa büyük bir sistem söz konusuysa, yüzlerce hatta binlerce özellik ve bu özellikler arasında çok sayıda kısıt içerebilmektedir. Bir özellik modelini oluşturmak ciddi bir alan bilgisi ve emek gerektirmekte, ancak tüm olasılıkları baştan öngörebilmek mümkün olmadığı için değişen koşullarla birlikte modellerin de evrimleşmesi kaçınılmaz olmaktadır. Büyük modellerin elle güncellenebilmesi hem çok iş gücü gerektiren hem de hataya açık bir iştir. Bu çalışmada, bahsedilen dönüşümlerin formel olarak tanımlanabilmesi ve otomatik olarak yerine getirilebilmesi için yeni bir dönüşüm dili önerilmektedir. Yeni bir özellik modeli dönüşüm dili olan Feather dilinin temel yapısı, içerdiği komutlar ve Feather'da yazılmış betiklerin otomatik olarak işlenmesini sağlayan yorumlayıcı yazılım sunulmaktadır.

Anahtar Sözcükler: yazılım ürün hatları, değişkenlik yönetimi, özellik modeli dönüşümü.

Automated Transformation of Feature Models

Abstract

Feature models are amongst the most popular tools used for variability management in software product line engineering. A feature model, especially if a large system is under consideration, can include hundreds and even thousands of features and cross-tree constraints. Building a feature model requires significant domain knowledge and substantial effort, however, since it is not possible to foresee every possibility in advance it becomes inevitable for the feature models to evolve. Updating large models manually is an error-prone task and requires a substantial amount of effort. This study discusses a transformation language to enable formal definition and automatic handling of feature model evolutions. Foundations of Feather, a novel transformation language, commands it includes and an interpreter to execute the scripts written in the Feather language are presented in this article.

Keywords: Software product lines, variability management, feature model transformation

1. Giriş

Ürün hatlarının kullanılması fikri otomotiv endüstrisinden elektronik parça üretimine uzanan geniş bir yelpazede yüzyılı aşkın süredir başarıyla uygulanmaktadır. Yazılım ürünlerinin ve yoğun şekilde yazılım içeren sistemlerin gün geçtikçe daha karmaşık hale gelmesiyle birlikte ürün hatlarının yazılım dünyasına uyarlanması daha çok ilgi çeken bir konu olmuştur. Bu konuda yapılan çalışmaların sonucunda, ortak bir nüve eleman kümesinden kuralları belirli bir şekilde geliştirilen ve farklı kesimlerin ihtiyaçlarını karşılamaya yönelik olarak oluşturulan yazılım yoğun sistemler ailesi [8] olarak tanımlanan yazılım ürün hattı (YÜH) kavramı doğmuştur.

YÜH'ler daha kısa zamanda, daha düşük maliyetle, daha yüksek kalitede yazılım ürünü geliştirilmesi amacıyla pek çok farklı projede başarıyla kullanılmıştır [24]. Örneğin, Boeing firması havacılık yazılım aileleri geliştirmek için Bold Stroke YÜH mimarisini kullanmaktadır [26]. Büyük ölçekli geleneksel projelerde YÜH'lerin başarıyla kullanılmış olması araştırmacıları YÜH mimarisinin hızlı değişen kullanıcı gereksinimleri ve oynak çevresel koşullar içeren uygulamalarda da kullanılabilmesi için araştırma yapmaya yönlendirmiştir. Bu araştırmaların meyvesi olarak da değişikliklere daha hızlı uyum sağlayabilen dinamik yazılım ürün hattı (DYÜH) mimarisi elde edilmiştir.

DYÜH mimarisinde değişkenliğin yönetilmesi için kullanılan çeşitli teknikler arasında en popüler olanı özellik modelleridir [7]. Dinamik bir sistem, doğası gereği, tüm parçalarıyla değişime ve dönüşüme açık olmak durumundadır, dolayısıyla sistemin içerdiği özellik modeli de bu şartları sağlamakla yükümlüdür. Örneğin, farklı şirketlerin sağladığı web servislerini ortak bir platformda kullanıcılara sunan bir yazılım ailesi zaman içinde sunduğu servis yelpazesini genişletmek istediğinde, ailedeki değişkenliği yöneten özellik modelinin de yeni servislerin eklenmesi veya var olan servislerin güncellenmesine açık olmalıdır. Bu gereksinim özellik modelinin evrim geçirmesini gerekli kılmaktadır. Ancak model büyük olduğunda dönüşümlerin gerçekleştiriminin elle yapılması hem yavaş olması hem de hataya açık olması nedeniyle istenen bir durum olmaktan çıkmaktadır.

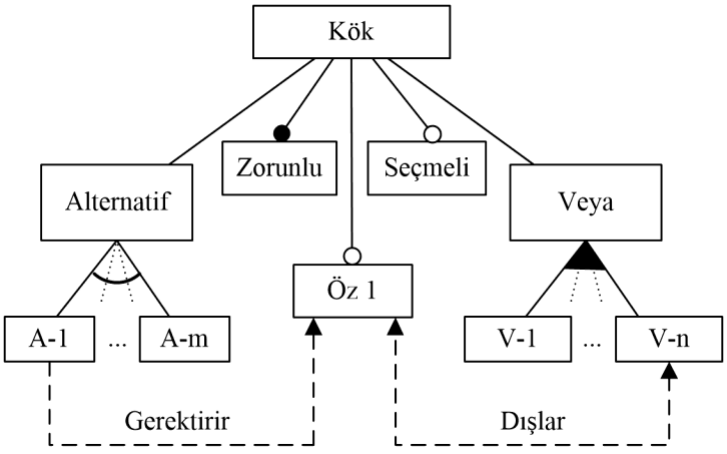
Yeni bir özellik modeli dönüşüm dili olan Feather bu ihtiyaca yanıt vermesi amacıyla tasarlanmıştır. Bir Feather betiği iki temel kısımdan oluşur: dönüştürülecek özellik modelini tanımlayan deklarasyonlar ve dönüşümleri tarif eden komutlar. Yorumlayıcı betiği okur, önce deklarasyonlarla tanımlanan özellik modelini inşa eder, sonra sırasıyla her bir dönüşüm komutunu çalıştırarak tarif edilen işlemleri gerçekleştirir, son olarak da dönüştürülmüş modeli tanımlayan deklarasyonları sonuç olarak kaydeder. Bu sayede dönüşüm otomatik olarak hem hızlı bir şekilde hem de hatasız bir biçimde tamamlanabilmektedir.

Bu makalenin kalanı şu şekilde organize edilmiştir: İkinci bölüm özellik modelleri ve model dönüşümü üzerine kısaca art alan bilgisi sağlar. Üçüncü bölüm kullanılan meta-model ve ifadeleri sunarak çalışmanın çerçevesini çizer. Dördüncü bölüm Feather dilini, beşinci bölüm ise yorumlayıcı yazılımı sunar. Altıncı bölüm ilgili çalışmaları özetler. Yedinci ve son bölüm ise sonuçları sunmaktadır.

2. Özellik Modelleri ve Dönüşüm

Özellik modelleri ilk olarak Feature Oriented Domain Analysis (FODA) çalışmasının bir parçası olarak Kang vd. [13] tarafından ortaya konmuştur. Özellik modellerindeki "özellik" terimi, bir kavramın (sistem, bileşen, sınıf, vb.), kavramla ilgilenen paydaş açısından ayırt edici bir karakteristiği anlamında kullanılmaktadır [28]. Bir özellik modeli ise; hiyerarşik olarak belirlenmiş bir özellik kümesi, bu özellikler arasında tanımlanmış ayrışma ilişkileri, rasgele özellikler arasında tanımlanmış dallar-arası kısıtlar, model üzerinde özellik seçiminde dikkat edilecek hususlar, seçimlerin doğurabileceği kazanç ve kayıplar gibi konulardaki bilgi notlarından oluşan bir model olarak tanımlanmaktadır [13].

Özellik modellerinde bir ebeveyn özellik ile çocukları arasındaki ilişkileri tanımlayan dört çeşit ayrışma ilişkisi bulunur: zorunlu, seçmeli, alternatif veya. Bunların yanı sıra farklı dallarda bulunan rasgele özellikler arasındaki ilişkileri tanımlayan iki çeşit kısıt bulunmaktadır: gerektirir, dışlar. Şekil 1'de bu ilişkileri içeren farazi bir özellik modelini gösteren bir özellik çizeneği verilmiştir.



Şekil 1: Örnek bir özellik modeline ait çizenek.

Özellik modellerinin ilk olarak önerilmesinden bu yana çeşitli geliştirmeler yapılmış, farklı ve genişletilmiş modeller geliştirilmiştir. Genişletilmiş özellik modelleri, özelliklerin içerdiği öznitelikler aracılığıyla ekstra bilgi sağlanmasına olanak verir. Bir öznitelik ait olduğu özelliğin ölçülebilen herhangi bir karakteristiğini tanımlar. Öznitelikler model üzerinde tanımlanan kısıtlarda da yer alabilmektedir. Örneğin, *"Eğer Ö1 özelliğine ait n1 özniteliğinin değeri 50'den büyük ise, o zaman Ö2 özelliği de ürüne dahil edilmelidir"* şeklinde kısıtlar tanımlamak mümkündür.

Bir model dönüşümü temelde bir kaynak modelden otomatik olarak bir hedef model oluşturma işlemidir [15]. İşlem bir dönüşüm kuralı kümesinden oluşan dönüşüm tanımına uygun olarak gerçekleştirilir. Her bir dönüşüm kuralı bir veya birden fazla kaynak model elemanının nasıl dönüştürüleceği ve hedef modelde nasıl görüneceğini tarif eder. Model dönüşümünün detaylı bir taksonomisi için okuyucu Mens ve Van Gorp [20] tarafından yapılmış çalışmaya başvurabilir, izleyen paragrafta sadece bu makaleye ilişkin dönüşüm kavramları hakkında kısaca bilgi verilecektir.

Feather dönüşümleri yatay seviyededir, yani kaynak ve hedef modeller aynı soyutlama seviyesinde olacaktır. Dönüşümler modeli semantik seviyede etkileme gücüne sahiptir. Kaynak ve hedef modeller aynı dilde ifade edilmektedir, dolayısıyla Feather dönüşümleri iç kökenli dönüşümlerdir. Dönüşümlerde sadece tek bir kaynak model bulunur ve dönüşümler doğrudan o model üzerinde gerçekleştirilir.

3. Temeller, Tanımlar ve Çerçeve

3.1 Meta-model

Feather kullanılarak dönüştürülecek özellik modelleri Feather meta-modeline uygun olmak zorundadır. Feather meta-modeli öznitelikler kullanılarak genişletilmiş bir temel özellik modeli biçimindedir. Model nicelik-tabanlı ayrışma ilişkileri [10] veya karmaşık dallar arası kısıtlar [14] içermemelidir. Ayrıca meta-model yapısı ağaç biçimindedir, dolayısıyla özellik hiyerarşisindeki ayrışma ilişkileri döngü içermemelidir.

Herhangi bir özellik alan bilgisini taşımak için istenen sayıda öznitelik içerebilir. Modellemeyi yapan tarafından tanımlanmış bu tip özniteliklere ek olarak her bir özelliğin dört adet yapısal öznitelik içerdiği varsayılmaktadır. Yapısal özniteliklerin biri özelliği nitelemekte (`_name`; özelliğin adını tutar), diğer üçü ise hiyerarşik ilişkileri tanımlamakta (`_parent`; özelliğin ebeveyni olan özelliğe bir işaretçidir, `_decomp`; özelliğin ebeveyni ile arasında olan ayrışma ilişkisinin tipini belirtir, `_decompID`; aynı ayrışma ilişkisine girmiş olan kardeş özelliklerin belirlenmesini sağlayan bir ayrışma ilişkisi tanımlama numarasıdır) kullanılmaktadır. Model tasarımcısı açısından ilk üç öznitelik güncellenebilirken sonuncusu otomatik olarak atanmış ve salt okunur bir görünüme sahiptir.

3.2 İfadeler

Feather dört genel veri tipi (tam sayı, reel sayı, Boolean ve karakter dizisi) ve özellik modellerindeki ayrışma ilişkilerini tanımlayan bir özel veri tipini barındırır. Dildeki ifadeler bu veri tiplerinden herhangi birine ait sabitlerin yanı sıra <özellik-öznitelik> terimlerini içerebilir. Bir <özellik-öznitelik> terimi *"Ö".öz* gösterimiyle belirtilir ve "*Ö* adlı özelliğe ait *öz* özniteliğinin değeri" olarak anlamlandırılır.

Feather dilinde aritmetik ve mantıksal olmak üzere iki tip ifade bulunmaktadır. Aritmetik ifadeler nümerik değerlere beş aritmetik operatörün (toplama, çıkarma, çarpma, bölme, modulo) uygulanması ile elde edilir. Mantıksal ifadeler, nümerik değerlere uygulandığında Boolean sonuç veren karşılaştırma operatörlerinin (<, <=, >, >=, =, <>) kullanılması veya Boolean değerlere mantıksal operatörlerin uygulanması (`and`, `or`, `not`) ile elde edilir. Operatörlerin öncelik sırası C programlama dilinde kullanılan karşılıklarının öncelik sırası ile aynıdır.

3.3 Özellik Değişkenleri

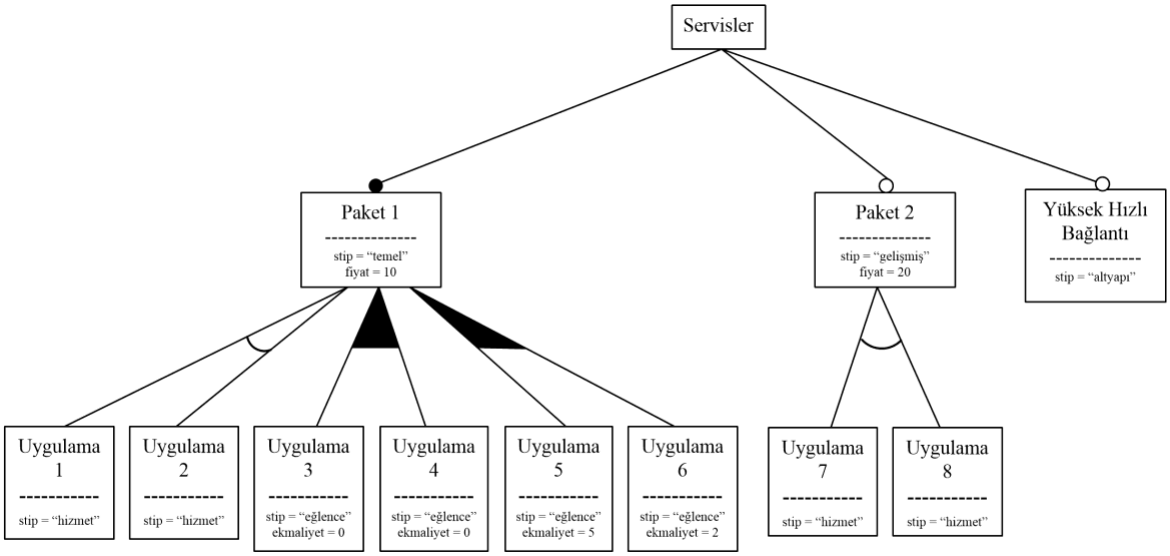
Zaman zaman bir dönüşüm işleminin uygulanacağı özelliğin hangisi olacağı, dönüşüme konu olan özellik modelinin dinamik doğasından dolayı betiğin yazıldığı sırada bilinmiyor olabilir. Veya komutta

belirtilen dönüşümün belli şartları sağlayan tüm özelliklere uygulanması isteniyor ancak bu özelliklerin kesin sayısı ve hangileri olduğu betik yazılırken bilinmiyor olabilir. Örneğin, tasarımcı kök özelliğin çocuklarından belli bir özniteliğe sahip olan ve bu özniteliğin değerinin belli bir değerden küçük olduğu tüm özelliklere bir dönüşüm uygulanmasını belirtmek isteyebilir. Bu gibi durumlar için Feather özellik değişkenlerinin kullanılmasına izin verir.

Bir özellik değişkeni terimi *D.öz* biçiminde kullanılır. Burada *D* özellik değişkenini belirtmekte ve *öz* adlı bir özniteliğe sahip özellikleri nitelemektedir. Bir özellik değişkeninin etki alanı kullanıldığı komutla sınırlıdır ve tüm betiğe yayılmaz. Özellik değişkenleri, dönüşüme konu olacak özellik veya özelliklerin çözümlenmesi işleminin gerçekleştirim anına kadar ertelenmesine olanak sağlayarak hem dile önemli bir esneklik kazandırır, hem de çözümlenme işlemini bilgisayara bırakarak tasarımcının omuzlarından önemli bir iş yükünü alır.

4. Feather Dil Yapısı

Bir Feather betiği temel olarak iki kısımdan oluşur: dönüşüme konu olacak kaynak özellik modelini tanımlayan deklarasyonlar ve dönüşümleri tarif eden komutlar. Bu bölümde deklarasyon ve komut yapıları sunulmaktadır.



Şekil 2: Farazi bir özellik modeli.

4.1 Deklarasyonlar

Deklarasyon bölümü her zaman kök özellik deklarasyonu ile başlar, diğer özelliklerin deklarasyonu ile devam eder ve dallar arası kısıtların deklarasyonu ile sona erer. Bir kök özellik deklarasyonu şu yapıdadır:

Kod 1: Kök Özellik Deklarasyonu.

```
1  root "Servisler";
```

Modelde kök özellik dışında farklı özellikler varsa bu özelliklerin deklarasyonları kök özellik deklarasyonundan sonra yapılır. Modeldeki tüm özellikler tam olarak bir kez deklare edilmelidir (hiçbiri atlanmamalı veya mükerrer deklarasyon yapılmamalıdır), ancak deklarasyonların sıralamasında hiyerarşik yapı belirleyici bir faktör değildir (bir çocuk özellik ebeveyn özellikten önce deklare edilebilir). Örneğin, Şekil 2'de verilmiş modeldeki "Uygulama 5" özelliğinin deklarasyonu şu şekilde olacaktır:

Kod 2: Çocuk Özellik Deklarasyonu.

```
1  feature "Uygulama 5"
2        "Paket 1" or to "Uygulama 6"
3        attribute stip "eğlence"
4        attribute ekmaliyet 5;
```

Deklarasyonlar bölümünün son kısmını modeldeki dallar arası kısıtların deklarasyonları oluşturur. Aynı etkiye sahip kısıtlar (örneğin, "X" özelliği "Y" özelliğini dışlar ve "Y" özelliği "X" özelliğini dışlar kısıtları) bulunması halinde kopyalar modelin oluşturulması sırasında elenir. Her bir kısıt, kısıta katılan özellikler ve bu özelliklerin arasındaki kısıt tipi belirtilerek deklare edilir. Örneğin, Şekil 2'de verilen modelde "Uygulama 4" ve "Uygulama 5" özellikleri arasında bir dışlama kısıtı var ise bu kısıt şu şekilde deklare edilecektir:

Kod 3: Dallar-arası Kısıt Deklarasyonu.

```
1  constraint "Uygulama 4" excludes "Uygulama 5";
```

4.2 Komutlar

Feather dilinde beş tanesi özellikleri beş tanesi de dallar arası kısıtları dönüştürme amaçlı toplam on adet komut bulunmaktadır.

Tablo 1: Özellik Dönüştürme Komutları.

Komutun Amacı	**Komut Yapısı**
Yeni bir özellik eklemek	`add feature` *Özelliğin-Adı* `with attributes (` *Öznitelik-Listesi* `)` [where Tümcesi] `;`
Var olan tek bir özelliği güncellemek	`update feature` *Özellik-Tanımlayıcı* `set` *Öznitelik-Güncellemeleri* [where Tümcesi] `;`
Var olan bir özellik kümesini güncellemek	`updateall feature` *Özellik-Değişkeni* `set` *Öznitelik-Güncellemeleri* [where Tümcesi] `;`
Var olan tek bir özelliği silmek	`remove feature` *Özellik-Tanımlayıcı* [where Tümcesi] `;`
Var olan bir küme özelliği silmek	`removeall feature` *Özellik- Değişkeni* [where Tümcesi] `;`

Özellik-Tanımlayıcı bir özellik adı veya özellik değişkeni olabilir. Where tümcesi bir Boolean ifadedir ve isteğe bağlıdır, eğer komut gerektirmiyorsa dahil edilmeyebilir. Komutlar belirsizlik içermemelidir, örneğin, tek bir özelliğin dönüşümünü tarif etmeyi amaçlayan bir komutun uygulanabileceği birden fazla özellik bulunmamalıdır.

Örneğin, tasarımcı "Paket 1" altındaki ek maliyet isteyen tüm özellikleri "Paket 2" özelliğinin altına taşımak istiyor, bu sırada da ek maliyetlere 3 birim zam yapmak istiyorsa şu komutu kullanabilir:

Kod 4: Örnek bir komut.

```
1  updateall feature F
2    set _parent = "Paket 2",
3        _decomp = alternative to G,
4        ekmaliyet = numeric: F.ekmaliyet + 3
5    where F.ekmaliyet > 0
6     and  F._parent = "Paket 1"
7     and  G._parent = "Paket 2"
8     and  G.stip = "hizmet";
```

Dallar arası kısıtların dönüşümünü hedefleyen komutlar şu biçimdedirler:

Tablo 2: Kısıt Dönüştürme Komutları.

Komutun Amacı	**Komut Yapısı**
Yeni bir kısıt eklemek	`add constraint` *Kısıt-Tanımlayıcı* [where Tümcesi] `;`
Var olan tek bir kısıtı güncellemek	`update constraint` *Kısıt -Tanımlayıcı* `set` *Kısıt-Güncellemeleri* [where Tümcesi] `;`
Var olan bir kısıt kümesini güncellemek	`updateall constraint` *Kısıt -Tanımlayıcı* `set` *Kısıt-Güncellemeleri* [where Tümcesi] `;`
Var olan tek bir kısıtı silmek	`remove constraint` *Kısıt -Tanımlayıcı* [where Tümcesi] `;`
Var olan bir küme kısıtı silmek	`removeall constraint` *Kısıt -Tanımlayıcı* [where Tümcesi] `;`

Örneğin, tasarımcı modele yeni bir kısıt olan "Uygulama 2" özelliği "Uygulama 6" özelliğini gerektirir şeklinde bir kısıt eklemek istiyorsa şu komutu kullanabilir:

Kod 5: Örnek bir komut.

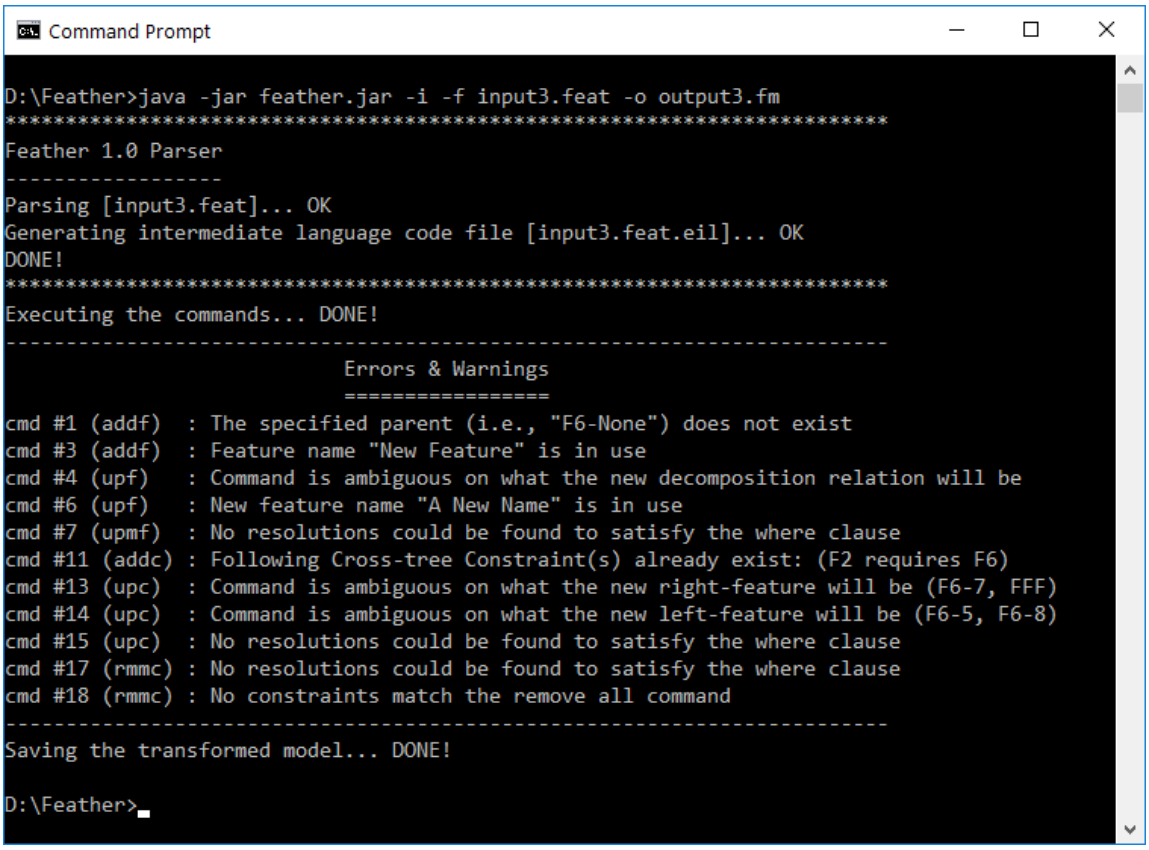
```
1  add constraint
2    "Uygulama 2" requires "Uygulama 6";
```

5. Yorumlayıcı

Feather yorumlayıcısı bir Feather betiğinin geçerliğini denetlemek ve çalıştırmak görevlerini yerine getirir. Yorumlayıcı Java programlama dilinde geliştirilmiş ve çalıştırılabilir bir jar dosyası halinde paketlenmiştir. Yorumlayıcı BSD lisansı ile kaynak kodları açık olacak şekilde kullanıma sunulmuştur [29]. Yorumlayıcı oluşturulan temel bileşenler izleyen paragraflarda sunulmaktadır.

Yorumlayıcı argümanlarını komut satırından alır ve mesajlarını yine komut penceresinde verir. Girdi-Çıktıdan sorumlu bileşen girdi betiği dosyasının okunmasından ve dönüştürülmüş modelin bir dosyaya kaydedilmesinden sorumludur. Betik hem deklarasyonları hem de komutları içeren tek bir dosyadan oluşabileceği gibi, deklarasyonlar ve komutlar için ayrı dosyalardan oluşuyor da olabilir. Bileşen girdi

dosyasındaki deklarasyonların ve komutların geçerliğini kontrol etmek için bir ayrıştırıcı bileşenden yararlanır.



Şekil 3: Yorumlayıcının çalışması ve verdiği mesajlara bir örnek.

Ayrıştırıcı bileşen girdideki deklarasyonların ve komutların geçerliğini Feather gramerine uygun olup olmadıklarını denetleyerek kontrol eder. Bileşenin iskeleti ANTLR ayrıştırıcı üretici [21] aracının ürettiği Java kodunun üstüne inşa edilmiştir. Feather tasarlanırken özellik modelinin oluşturulmasından sorumlu alan uzmanlarının yazılım mühendisliği metotlarına aşina olmayabileceği göz önünde tutulmuş, bu nedenle okunurluk ve anlaşılırlığın mümkün olduğunca yüksek bir seviyede olmasına özen gösterilmiştir. Ayrıştırıcı insanlar için okunurluğu yüksek olan deklarasyon ve komutları alır, otomatik işlemeye daha uygun olan bir dönüşüm ara diline çevirir. Bu amaçla Feather ara dili için tasarlanmış bir özellik grameri kullanılır.

Dönüşüm ara diline çevrilmiş deklarasyon ve komutlar yorumlayıcı motoru tarafından kaynak modelin oluşturulması ve dönüşüm komutlarının yerine getirilmesinde kullanılır. Her bir komutun tarif ettiği dönüşümler motor tarafından anlamlandırılır ve yerine getirilir. Ayrıca komutların yan etkileri de (örneğin, modelden silinen bir özelliğe ait tüm alt özelliklerin bulunması ve modelden silinmesi, ayrıca bu özellikleri içeren tüm dallar arası kısıtların tespit edilip modelden çıkarılması) yorumlayıcı motoru tarafından tespit edilip model yapısına yansıtılır. Eğer komutlar özellik değişkenleri içeriyorsa yorumlayıcı motoru değişkenlerin çözümleneceği özelliklerin bulunması için kısıt çözücü bileşenden yararlanır.

Kısıt çözücü bileşenin görevi bir komutta kullanılan özellik değişkenlerinin, where tümcesinde belirtilmiş ifadeyi tatmin edecek şekilde gerçek özelliklere çözümlenmesini sağlamaktır. Bileşen bu hedefe ulaşmak için önce komutu bir sonlu-alanlar üzerinde tanımlı kısıt problemine çevirir, SICStus Prolog jasper kütüphanesini [27] kullanarak probleme ait çözümleri bulur, bulunan çözümleri kullanarak değişken – özellik çözümlemelerini gerçekleştirir. Önceki komutlar kaynak modeli dönüştürmüş

olacağından bu çözümlemelerin betiğin çalıştırılması sırasında yapılması gereği mevcuttur, bu nedenle bileşen her şeyi betiğin çalışması sırasında dinamik olarak gerçekleştirir.

6. İlgili Çalışmalar

Capilla vd. [6] özellik modellerindeki yapısal değişikliklerin otomatik olarak gerçekleştirimini önemli bir problem olarak tanımlamıştır. Yazarlar bu problemi öne sürdükleri Runtime Variability (RunVar) modeline dahil ettikleri bir çözümle aşmaya çalışmıştır. Önerdikleri çözüm bir süper-tip kullanarak özellik ekleme, silme ve özelliği başka bir dala taşıma işlemlerini gerçekleştirmek şeklindedir. Süper-tipler kullanıcı tarafından tanımlanmış karakter dizgeleridir ve özellikleri kategorize etmekte kullanılırlar. RunVar dönüşüm işlemlerini sadece modeldeki yaprak özelliklere uygulayabilmektedir, zira yazarlar model ağacındaki ara özelliklere uygulanacak dönüşümlerin çok daha karmaşık bir doğaya sahip olacağını belirtmişlerdir. Feather ise dönüşüm işlemlerini yaprak özelliklerin yanı sıra ara özelliklere de başarıyla uygulama yeteneğine sahiptir. Ayrıca RunVar dönüşüm komutunda belirtilen tariflerin ilgili özelliklerin süper-tipleri arasındaki uyumluluk denetlemeleriyle, ki temelde bu uyumluluk denetimi iki karakter dizgesinin eşitliğini kontrol etmekten ibarettir, sınırlandırmıştır. Oysa Feather dönüşüm tarifinde herhangi bir özelliğin herhangi bir özniteliğini içeren formüllerin kullanılmasına olanak sağlamaktadır. RunVar bir komutla tek bir özelliğe ilişkin dönüşümlere olanak sağlarken Feather tek komutla bir veya birden fazla özelliğin dönüştürülebilmesini desteklemektedir. RunVar dönüşüm konusu olarak sadece özellikleri göz önünde bulundururken, Feather dallar arası kısıtların da dönüştürülebilmesine olanak sağlamaktadır.

Pleuss vd. [23] YÜH'lerin dönüşümü için model-tabanlı bir yaklaşım benimsemiş ve evrim adımlarının tarifi için EvoFM adını verdikleri özel bir özellik modeli tasarlamıştır. Yazarlar bu amaçla EvoOperators adını verdikleri bir dizi operatör tanımlamış ve yapısal değişiklikleri bu operatörler aracılığıyla tarif etmişlerdir. İlgili çalışmada model dönüşümlerinin gerçekleştirimi için Epsilon Transformation Language (ETL) [16] kullanılmıştır. Feather EvoOperators kullanılarak ifade edilebilecek tüm dönüşümleri ve daha fazlasını tarif edebilecek ve gerçekleştirecek güce sahiptir. Ayrıca, Feather özel olarak özellik modellerinin dönüşümü hedefine yönelik olarak tasarlanmış olduğu için özellik modellerine aşina bir uzmanın yeni ve genel bir dönüşüm dilini öğrenme zahmetine katlanmadan kolayca kullanabileceği bir yapıya sahiptir.

Bürdek vd. [5] ürün-hattı evriminin incelenebilmesi için kompleks özellik modeli farklarını değerlendiren bir yöntem önermiştir. Yazarlar özellik modelinin önceki ve sonraki sürümlerini karşılaştırıp farkları düzenleme operasyonları ile ifade etmektedir. Bu amaçla düzenleme operatörlerinin listelenmesi ve sınıflandırılmasına yönelik olarak bir katalog sunulmuştur. Yazarlar, bu operatörlerin gerçekleştirimi için cebirsel çizge dönüşümü tabanlı bir dönüşüm dili olan Henshin model dönüştürme dilini [2] kullanmıştır. Feather, anılan çalışmada listelenmiş tüm düzenleme operatörlerinin ve bu operatörleri içeren karmaşık ifadelerin tarif edilmesine olanak sağlamaktadır.

Literatürde özellik modeli evrimini konu alan çeşitli çalışmalar bulunmaktadır. Seidl vd. [25] özellik modeli evrimleştiğinde özelliklerin de tutarlı bir şekilde evrimleşmesini sağlayan kavramsal bir temel sunmaktadır. Baresi ve Quinton [3] özellik modeli evrimleştikçe modelden üretilmiş olan konfigürasyonun, konfigürasyon uzayındaki geçerli bir başka konfigürasyona bağlanmasını sağlayan bir mimari inşa etmiştir. Borba vd. [4] YÜH'lerin güvenli bir şekilde evrimleşmesini güvence altına almak için bir dizi dönüşüm şablonu tanımlamaktadır. Peng vd. [22] evrim geçmişini çözümlemek ve gelecekteki olası evrim adımlarını öngörebilmek için *'bir zorunlu özelliğin seçmeli özelliğe dönüşmesi'* gibi bir dizi dönüşüm şablonu içeren değer-bazlı bir evrim çözümleme metodu öne sürmüştür. Lotufo vd. [19] Linux çekirdek özellik modelinin evrimini analiz etmiş ve modeldeki yapısal değişiklikleri kategorize etmiştir. Thüm vd. [30] özellik modeli düzenlemeleri üzerinde çıkarsamalar yapılabilmesi için bir algoritma tasarlamış ve değişiklikleri sınıflandırma yoluna gitmiştir. Feather, anılan çalışmalardaki tüm dönüşüm örnek ve senaryolarını tarif etme ve gerçekleştirme yeteneğine sahiptir.

Dönüşüm dilleri belli bir meta-model yapısına uyan modelleri dönüştürmek için tasarlanırlar. Bu meta-modeller arasında en popüler olanları OMG'nin Meta-Object Facility'sini (MOF) temel alanlardır.

Örneğin, ATL [12], Tefkat [18], UML-RSDS [17] ve VIATRA [9] gibi diller MOF tabanlı meta-modellere sahiptirler. Feather, genişletilmiş özellik modeli biçiminde bir meta-model yapısına sahiptir.

Dönüşüm dilleri dönüşümlerin tarif edilebilmesi için farklı stilleri içermektedirler. Tefkat ve UML-RSDS gibi diller bildirimsel bir stil benimsemiştir ve dönüşümün nasıl yapılacağından ziyade dönüşümün hangi elemanlara hangi şartlarla uygulanacağını odak noktasına alır. Zorunlu stili benimsemiş SiTra [1] ve Kermeta [11] gibi diller dönüşümün adım adım nasıl yapılacağının tarif edilmesine olanak sağlarlar. ATL ve VIATRA gibi diller melez bir yaklaşımı benimsemiştir. Feather bildirimsel bir stile sahiptir.

7. Sonuç

Bu makale yeni bir özellik modeli dönüşüm dili olan Feather'ın nüvesini sunmaktadır. Feather, dinamik ürün hatlarının evrimi sırasında özellik modellerinin otomatik dönüşümüne katkı sağlamak amacıyla tasarlanmıştır. Her ne kadar bir dönüşüm dili olsa da, genel amaçlı bir dönüşüm dili olmamasından dolayı, Feather mevcut dönüşüm dillerine bir alternatif olarak tasarlanmamıştır, bu nedenle Feather'ın sunduğu katkı temel olarak model dönüşümü dünyasından ziyade yazılım ürün hatları alanınadır.

Özellik modelleri üzerinde dönüşümleri elle gerçekleştirmek, bilhassa modeller büyük olduğunda, çok fazla emek gerektiren ve hataya açık bir iştir, bu nedenle otomatik dönüşüm desteği almak zaruri hale gelmiştir. Feather yorumlayıcısı tarif edilen dönüşümlerin verimli bir şekilde gerçekleştirilmesi için otomatik destek sağlar. Yorumlayıcı geliştirilirken verimliliğin arttırılması için ANTLR ve SICStus Prolog gibi olgunlaşmış ve kendi alanlarında etkisini ispatlamış araçlardan yararlanılmıştır. Bu sayede yorumlayıcının klasik yazılım ürün hatlarının yanı sıra dinamik ürün hatlarında da kullanılabilmesi hedeflenmiştir.

Feather'da yer verilen özellik değişkenleri istenen özelliklerin çeşitli nitelikler kullanılarak tarif edilmesine olanak sağlamaktadır. Bu durum Feather'in ifade gücünü ve esnekliğini önemli ölçüde arttırmaktadır. Tasarımcılar gerçek özellikleri tarif etmek için özellik değişkenleri kullanıp değişkenlerin mevcut özelliklere çözümlenmesini yorumlayıcıya bırakabilirler. Bu strateji iş yükünü tasarımcının omuzlarından alıp yorumlayıcıya kaydırdığı gibi dilin esnekliğini de arttırmaktadır. Özellik değişkenleri özellik modelinin yapısının tam olarak bilinmediği durumlarda bile dönüşüm işlemlerini tarif etmeyi sağlayarak uygulama alanlarını da önemli ölçüde arttırmaktadır.

Model dönüşümü doğası itibariyle zorlu bir işlemdir ve genel bir dönüşüm dilini öğrenmek ciddi bir zaman ve emek gerektirebilir. Bu nedenle öğrenilmesi ve kullanılması kolay bir dönüşüm diline sahip olmak tercih edilir bir durumdur. Bu etkenler Feather'ın tasarlamasında etkin bir şekilde rol oynamıştır. Feather SQL'den ilham alınmış bildirimsel bir yapıya sahiptir ve özellik modeli tasarımcılarının aşina olduğu elemanlar içermektedir. SQL, bilgisayar programcısı olmayan kişiler de dahil olmak üzere, çok geniş bir kullanıcı tabanına sahiptir. Alan uzmanları da benzer bir kullanıcı tabanına mensup olabileceğinden, etkinliği on yıllar boyunca kanıtlanmış olan SQL benzerliğinin Feather kullanımını kolaylaştırılacağı öngörülmektedir.

Feather'ı geliştirmek için çeşitli zenginleştirmeler öne sürülebilir. Tam sayı, reel sayı, Boolean ve karakter dizgesinden oluşan öznitelik tipleri, koleksiyonları (kümeler, listeler, vb.) içerecek şekilde genişletilebilir. Meta-model, nicelik-bazlı özellik modellerini de tarif edebilecek şekilde genişletilebilir. Dil SQL benzeri yordamları (min, max, sum, vb.) içerecek şekilde zenginleştirilebilir. Yorumlayıcıya dönüşümü otomatik olarak analiz edecek ve istenmeyen dönüşüm adımlarının (örneğin modelde tutarsızlığa yol açacak olan bir dönüşüm komutunun) atlanmasını sağlayacak yetenekler eklenebilir.

Teşekkür

Bu çalışma TÜBİTAK ARDEB 1001- Bilimsel ve Teknolojik Araştırma Projelerini Destekleme Programı çerçevesinde, 215E188 kodlu proje kapsamında desteklenmiştir.

Referanslar

[1] D. H. Akehurst, B. Bordbar, M. J. Evans, W. G. J. Howells, and K. D. McDonald-Maier, "SiTra: simple transformations in Java", in *Proc. MoDELS 2006*, Genova, Italy, 2006, pp. 351–364.

[2] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, "Henshin: advanced concepts and tools for in-place EMF model transformations", in *Proc. MoDELS 2010*, Oslo, Norway, 2010, pp. 121–135.

[3] L. Baresi and C. Quinton, "Dynamically evolving the structural variability of dynamic software product lines", in *Proc. SEAMS '15*, Florence, Italy, 2015, pp. 57–63.

[4] P. Borba, L. Teixeira, and R. Gheyi, "A theory of software product line refinement", *Theor Comput Sci*, vol. 455, pp. 2–30, Oct. 2012, 10.1016/j.tcs.2012.01.031.

[5] J. Bürdek, T. Kehrer, M. Lochau, D. Reuling, U. Kelter, and A. Schürr, "Reasoning about product-line evolution using complex feature model differences", *Autom Softw Eng*, vol. 23, no. 4, pp. 687–733, Dec. 2016, 10.1007/s10515-015-0185-3.

[6] R. Capilla, J. Bosch, P. Trinidad, A, Ruiz-Cortés, and M. Hinchey, "An overview of dynamic software product line architectures and techniques: observations from research and industry", *J Syst Softw*, vol. 91, pp. 3–23, May 2014, 10.1016/j.jss.2013.12.038.

[7] L. Chen and M. A. Babar, "A systematic review of evaluation of variability management approaches in software product lines", *Inform Software Tech*, vol. 53, no. 4, pp. 344–362, Apr. 2011, 10.1016/j.infsof.2010.12.006.

[8] P. Clements and L. Northrop, *Software product lines: practices and patterns*, Boston, MA, USA: Addison-Wesley, 2001.

[9] G. Csertan, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varro, "VIATRA - visual automated transformations for formal verification and validation of UML models", in *Proc. ASE 2002*, Edinburgh, Scotland, UK, 2002, pp. 267–270.

[10] K. Czarnecki, S. Helsen, and U. Eisenecker, "Formalizing cardinality-based feature models and their specialization", *Softw Process Improve Pract*, vol. 10, no. 1, pp. 7–29, Mar. 2005, 10.1002/spip.213.

[11] J. M. Jézéquel, O. Barais, and F. Fleurey, "Model driven language engineering with Kermeta", in *Proc. GTTSE 2009*, Braga, Portugal, 2009, pp. 201–221.

[12] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool", *Sci Comput Program*, vol. 72, no. 1–2, pp. 31–39, Jun. 2008, 10.1016/j.scico.2007.08.002.

[13] K. Kang, S. Cohen, J. Hess, W. Novak, ve S. Peterson, *"Feature-Oriented Domain Analyses (FODA) Feasibility Study"*, Technical Report CMU/SEI-90-TR-21, Software Eng. Inst., Carnegie Mellon Univ., Pittsburgh, 1990.

[14] A. S. Karataş, H. Oğuztüzün, and Ali Doğru, "From extended feature models to constraint logic programming", *Sci Comput Program*, vol. 78, no. 12, pp. 2295–2312, Dec. 2013, 10.1016/j.scico.2012.06.004.

[15] A. Kleppe, J. Warmer, and W. Bast, *MDA explained, the model-driven architecture: practice and promise*, Boston, MA, USA: Addison-Wesley, 2003.

[16] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "The Epsilon transformation language", in *Proc. ICMT 2008*, Zürich, Switzerland, 2008, pp. 46–60.

[17] K. Lano and S. Kolahdouz-Rahimi, "Specification and verification of model transformations using UML-RSDS", in *Proc. IFM 2010*, Nancy, France, 2010, pp. 199–214.

[18] M. Lawley and J. Steel, "Practical declarative model transformation with Tefkat", in *Proc. MoDELS 2005*, Montego Bay, Jamaica, 2005, pp. 139–150.

[19] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wąsowski, "Evolution of the Linux kernel variability model", in *Proc. SPLC 2010*, Jeju Island, South Korea, 2010, pp. 136–150.

[20] T. Mens and P. Van Gorp, "A taxonomy of model transformation", *Electron Notes Theor Comput Sci*, vol. 152 pp. 125–142, Mar. 2006, 10.1016/j.entcs.2005.10.021.

[21] T. J. Parr and R. W. Quong, "ANTLR: A predicated-LL(k) parser generator", *Software Pract Exper*, vol. 25, no. 7, pp. 789–810, Jul. 1995, 10.1002/spe.4380250705.

[22] X. Peng, Y. Yu, and W. Zhao, "Analyzing evolution of variability in a software product line: From contexts and requirements to features", *Inform Software Tech*, vol. 53, no. 7, pp. 707–721, Jul. 2011, 10.1016/j.infsof.2011.01.001.

[23] A. Pleuss, G. Botterweck, D. Dhungana, A. Polzer, and S. Kowalewski, "Model-driven support for product line evolution on feature level", *J Syst Softw*, vol. 85, no. 10, pp. 2261–2274, Oct. 2012, 10.1016/j.jss.2011.08.008.

[24] K. Pohl, G. Böckle, and F. Linden, *Software product line engineering: foundations, principles, and techniques*, Berlin-Heidelberg, Germany: Springer-Verlag, 2005.

[25] C. Seidl, F. Heidenreich, and U. Aßmann, "Co-evolution of models and feature mapping in software product lines", in *Proc. SPLC '12*, Salvador, Brazil, 2012, pp. 76–85.

[26] D. C. Sharp, "Component based product line development of avionics software", in *Proc. SPLC-1*, Denver, CO, USA, 2000, pp. 353–369.

[27] SICStus prolog, https://sicstus.sics.se/, *son erişim Mayıs 2021*.

[28] M. Simos et al. *"Software Technology for Adaptable Reliable Systems (STARS) Organization Domain Modeling (ODM) Guidebook Version 2.0"*, STARS-VC-A025/001/00, Manassas, VA, Lockheed Martin Tactical Defense Systems, 1996.

[29] Supplementary material, https://github.com/askaratas/Feather, *son erişim Mayıs 2021*.

[30] T. Thüm, D. Batory, and C. Kastner, “Reasoning about edits to feature models”, in *Proc. ICSE '09*, Vancouver, Canada, 2009, pp. 254–264.