Dokuz Eylül Üniversitesi-Mühendislik Fakültesi
Fen ve Mühendislik Dergisi
Cilt 19 Sayı 55.1. (Özel Sayı) Ocak 2017

Dokuz Eylul University-Faculty of Engineering
Journal of Science and Engineering
Volume 19 Issue 55.1. (Special Issue) January 2017

DOI: 10.21205/deufmd.2017195525

Yazılım Mimarisi Bakış Açılarındaki Tutarlılık Kontrolü İçin Sistematik Bir Yöntem

Gülsüm Ece EKŞİ*[1], Bedir Tekinerdoğan[2]

[1]Bilkent Üniversitesi, Mühendislik Fakültesi, Bilgisayar Mühendisliği Bölümü, 06800, Ankara
[2]Wageningen Üniversitesi, Bilişim Teknolojisi Bölümü, 6700 EW, Wageningen

(Alınış / Received: 23.05.2016, Kabul / Accepted: 12.08.2016,
Online Yayınlanma / Published Online: 09.01.2017)

Anahtar Kelimeler
Yazılım Mimarisi, Mimari Bakış Açılarındaki Uygunluk

Özet: Literatürde, tasarım amaçlarına ulaşmayı sağlayan ve kodla yazılım mimarisi arasındaki uyumsuzlukları bulmaya yarayan bir takım çalışmalar öne sürülmüştür. Mimari bakış açıları ve kod arasındaki uyum nasıl olmalıysa, aynı şekilde her bir bakış açısı da kendi içinde ve diğer bakış açılarıyla uyumlu olmalıdır. Ancak, varolan mimari uygunluğu yöntemleri öncelik olarak kod ile mimari arasındaki uyuma odaklanmış ve bakış açılarının kendi arasındaki uyumsuzluklarını dikkate almamıştır. Bu makalede, yazılım mimarisi bakış açılarının kendi aralarındaki tutarsızlığını ele alan sistematik bir yöntem sunmaktayız. Bu amaç doğrultusunda, metamodelleri tanımlanan mimari bakış açılarını uygulamaya sokan ArchViewChecker adında bir araç geliştirdik ve örnek bir çalışma üzerinde aracımızı değerlendirdik.

A Systematic Approach for the Consistency Checking of Software Architecture Views

Keywords
Software Architecture, Architecture View Consistency

Abstract: *Several approaches have been proposed to detect the inconsistencies between the software architecture and the code to ensure that the original design goals are maintained. Similar to the consistency with the code it is important that an architecture view is consistent within itself and with other related architecture views. Unfortunately, the existing architecture conformance analysis approaches have primarily focused on checking the inconsistencies between the architecture and code, and did not explicitly consider the consistency within and among views. In this paper, we provide a systematic architecture conformance analysis approach that explicitly focuses on conformance analysis within and among architecture views. To this end, we define the meta-models of architecture viewpoints, present the conformance analysis approach, and provide the tool ArchViewChecker with a case study.*

*Sorumlu yazar: gulsumece.eksi@gmail.com

1. Giriş

Yazılım mimarisi, bir sistemin ana yapısını gösterdiği ve her bir paydaşın ihtiyacını dikkate aldığı için yazılım geliştirme sürecinin temel taşlarından biridir [1]. Yazılım geliştirme sürecinde mimariden yararlanabilmek için, hazırlanan mimarinin ve öngörülen tasarım kararlarının kodla uyumlu olması gerekmektedir. Ancak, projelerde koda dair değişen gereksinimler ve/veya uyarlamalar mimari ve kod arasında istenmeyen uyumsuzlukların oluşmasına yol açabilmektedir. Bu mimari ayrışma problemi, mimarinin tanımı ve ortaya çıkan kod arasındaki tutarsızlığı belirtmektedir. Literatürde, projelerin başında tanımlanan tasarım amaçlarına ulaşmayı sağlayan ve kodla yazılım mimarisi arasındaki uyumsuzlukları bulmaya yarayan bir takım çalışmalar öne sürülmüştür. Pratikte yazılım mimarisi, paydaşların ihtiyacını öne sürdüğü mimari bakış açılarıyla belgelenmektedir. Mimari bakış açıları ve kod arasındaki uyum nasıl olmalıysa, aynı şekilde her bir bakış açısı da kendi içinde ve diğer bakış açılarıyla uyumlu olmalıdır. Ancak, var olan mimari uygunluğu yöntemleri öncelik olarak kod ile mimari arasındaki uyuma odaklanmış ve bakış açılarının kendi aralarındaki uyumsuzluklarını dikkate almamıştır. Bu makalede, mimari bakış açılarının kendi aralarındaki uyumsuzluğu ele alan sistematik bir yöntem sunmaktayız. Bu yöntem ile mimari bakış açılarının kendi içinde ve diğer bakış açılarıyla olan uyumsuzlukları sistematik bir şekilde tespit edilebilmektedir. Bu amaç doğrultusunda, meta-modelleri tanımlanan mimari bakış açılarını uygulamaya sokan bir araç geliştirdik. Bakış açıları arasındaki uyumsuzlukları bulan yöntemimizi Görünümler ve Ötesi yaklaşımı [2] ile örneklendirdik. Sunulan yöntemi değerlendirmek için hata enjekte metodunu kullandık.

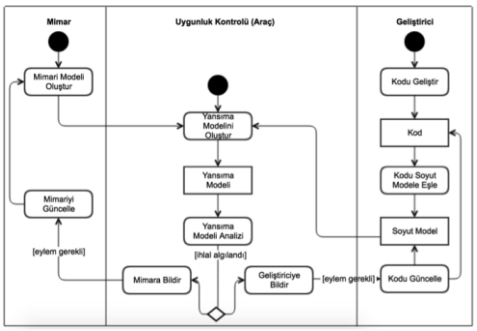
Makalenin geri kalanı şu şekilde düzenlenmiştir. Bölüm 2'de yazılım mimarisi uygunluk analizi hakkında genel bilgiler vermekteyiz. Bölüm 3, yazılım mimarisi bakış açılarının kendi aralarında ve kendi içlerinde olan uygunluk kontrolü yöntemimizi içermektedir. Geliştirdiğimiz ArchViewChecker adındaki aracımızı bölüm 4'te sunmaktayız. Bölüm 5'te örnek çalışmamızı sunmaktayız. Bölüm 6'da yöntemimize benzer çalışmaları anlatmaktayız. Son olarak, bölüm 7 makalemizin sonuç kısmını içermektedir.

2. Genel Bilgiler

Mimarinin tutarlı olması, mimari tasarım elemanlarının kod unsurlarına eşlenebiliriliğini ifade eder. Mimari ve kodun uyuşmama durumundaki ilişki "mimari ihlali" olarak belirtilir. Mimaride bulunan ilişki aynı zamanda kodda da bulunuyorsa buna "çakışan ilişki" denir. Mimarideki bağıntıların kodda bulunmaması durumu ise "yokluk ilişkisi" olarak adlandırılır. Mimari ve kod arasındaki bu yokluk ilişkileri, kodun daha hazır olmamasından kaynaklı olarak projelerin gelişme aşamasının ilk fazlarında ortaya çıkar. İlk başlarda yaşanan bu yokluk ilişkisi daha az bir önem taşırken, projelerin son aşamalarında kod ve mimari arasında bir uyum sağlanmazsa bu ilişki "ayrışma ilişkisi "ne döner. Mimari ihlalleri de işte bu "yokluk" ve "ayrışma" ilişkileri sebebiyle olur.

Mimari uygunluğu kontrolü için Murphy vd. [3] tarafından ortaya atılmış "Yansıma Modellemesi" yöntemi, başarılı tasarım iyileştirme tekniklerinden biridir. Mimari ve kod arasındaki uyum için kullanılan yansıma modeli yönteminin aktivite şeması Şekil 1'de gösterilmiştir. Bu yöntemde, yansıma modeli yazılımcıya sistemin

kaynak yapısını istenilen yüksek düzeyde (çoğunlukla mimarisel olarak) görme imkanı sağlar. Mimari modeli ve kod arasındaki uygunluğu kontrol edebilmek için kodun soyut bir modeli elde edilir. Bu iki model, daha önceden belirlenmiş kod ve uygulama arasındaki eşleştirme kuralları çerçevesinde karşılaştırılır. Karşılaştırmanın sonuçları ise bir yansıma modeli üzerinden kullanıcıya sunulur. Yansıma modeli analiz edilerek mimari, kod ve eşleştirme kuralları değişkenlik gösterebilir. Yansıma modelini kullanan mimari uygunluk analizi yöntemleri genellikle mimariyi modelleme, eşleştirmeleri modelleme, kaynak koddan soyut modeli elde etme, uygunluk analizi denetleyici ve sonuçta oluşan yansıma modeli üretici için araçlar içerir.



Şekil 1. Mimari ve kod uyumu için kullanılan yansıma modelindeki adımların aktivite şeması

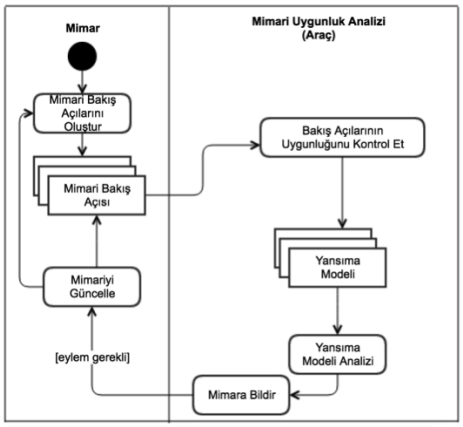
3. Yöntem

Bir önceki bölümde aktardığımız gibi literatürdeki [3-5] gibi çoğu çalışma, mimari ve kod arasındaki uygunsuzlukları bulmaya yöneliktir. Fakat aynı ölçüde mimari tasarımındaki tutarlılık da önemlidir. Genellikle, bakış açılarındaki uygunluk kontrolü manuel olarak yapılmaktadır. Küçük sistemlerde bu yöntem yararlı olsa da sayıca daha çok bakış açısı gerektiren büyük ölçekli sistemlerde makul sonuçlar vermeyebilir. Literatür taramamızı yaparken karşılaştığımız bir diğer konu ise birçok çalışmada geliştirilen uygunluk kontrolü araçları, kullanım senaryosu, sıralama, aktivite ve sınıf şemaları gibi daha çok sistemlerin analiz kısmında oluşturulan diyagramlar içindir. Ancak sistemlerin tasarım kısmında oluşturulan mimari bakış açılarının uygunluk kontrolü için olan araç desteği çok fazla bulunmamakla beraber bütün bakış açılarını içeren bir araç desteği yoktur.

Bu bölümde, otomatikleştirilmiş mimari bakış açıları kontrolü için sunduğumuz sistematik yöntemimizi aktaracağız. Şekil 2'de yöntemimizin üst düzey süreci gösterilmektedir. Mimari bakış açıları kullanıcı tarafından yaratıldıktan sonra bu bakış açılarının kendi aralarında ve kendi içlerinde olan uygunlukları "Bakış Açılarının Uygunluğunu Kontrol Et" adımında ele alınır. Burada önemli olan nokta, sadece birbiriyle alakalı olan bakış açılarının uygunluk kontrolü yapılmaktadır. Uygunluk kontrolü sonuçları kullanıcıya yansıma modeli şeklinde sunulur. "Yansıma Modeli Analizi" adımında bakış açıları araç tarafından analiz edilir ve "Mimara Bildir" adımında kullanıcıya bakış açılarında bulunan hatalar gösterilir.

Mimari bakış açılarındaki uygunluk kontrolü, bakış açılarının kendi içlerinde ve kendi aralarında olmak üzere iki ayrı şekilde incelenmiştir. Çalışmamızda yer alan mimari bakış açıları yönteminin içerdiği bakış açılarının hepsinin birbiriyle ilişkisi bulunmamaktadır. Uygunluk kontrolü yaptığımız bakış açıları, birbiriyle bağlantısı bulunan bakış açılarıdır. Örneğin, Görünümler ve Ötesi yönteminde [2] Ayrışma (Decomposition) ve Kullanım (Uses) bakış açıları arasında bir ilişki vardır. Bu ilişkiye göre kullanım bakış açısında bulunan her bir modül ayrışma bakış açısında da bulunmak zorundadır. Bu iki

bakış açısında uygunluk kontrolü yapılırken modüllerin iki bakış açısında da bulunup bulunmadığına bakılır. Çalışmamızda Görünümler ve Ötesi yönteminde bulunan her bir bakış açısının kendi içinde ve diğer bakış açılarıyla olan ilişkileri incelenmiştir. Bakış açılarının birbirleriyle olan ilişki incelemesi Tablo 1'de gösterilmiştir. Bu çizelgede bulunan çarpı (x) işareti ilgili bakış açısının kendi içinde ve diğer bakış açılarıyla arasında bir ilişkinin olduğunu ve uygunsuzluğunun olabileceğini, eksi (-) işareti de bakış açısının herhangi bir ilişkisinin bulunmadığını dolayısıyla herhangi bir uygunsuzluk içermediğini belirtmektedir. Sonuç olarak bakış açıları arasında uygunluk kontrolü yapılırken sadece aralarında ilişki bulunanlar değerlendirmeye alınmıştır. Bundan sonraki kısımda, yöntemimizi daha detaylı aktaracağız.



Şekil 2. Uygunluk kontrolü sürecini gösteren aktivite şeması

3.1. Mimari bakış açılarını yaratma

Mimari bakış açılarının oluşturulma kısmında JavaScript Object Notation (JSON) [6] adlı veri değişim dilini kullandık. JSON formatını tercih etmemizin sebebi ise makinelerin daha kolay çözümlemesi ve insanların daha rahat okuyup yazabilmesidir. Bu format isim/değer ikililerini desteklediği ve yazılım dillerinden bağımsız olduğu için bakış açılarının rahat bir biçimde yaratılması açısından uygundur. Bakış açıları arasında uygunluk kontrolü yapabilmek için ilk olarak yazılımcının bunları JSON formatı şeklinde oluşturması ve araca yüklemesi gerekmektedir. Yazılımcı JSON formatı şeklindeki bakış açılarını internetteki çevrimiçi JSON editör sitelerinde ya da yerel ortamlarda (Eclipse, IntelliJ IDEA, vb.) yaratabilir.

3.2. Bakış açılarının kendi içinde uygunluk kontrolü

Bu bölümde bakış açılarının kendi içlerindeki uygunluk kontrolünün nasıl yapıldığı anlatılmaktadır. Görünümler ve Ötesi yönteminden seçilmiş bakış açıları üzerinde örneklendirme yapılmıştır.

3.2.1. Ayrışma bakış açısı

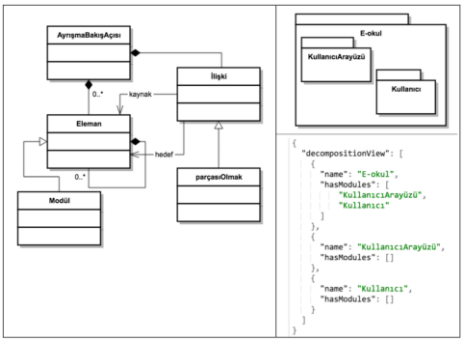
Ayrışma bakış açısı bir sistemin modüllerinin ve altmodüllerinin yapısını, böl ve yönet ilkesi doğrultusunda göstermektedir. Bakış açısının metamodeli, örnek modeli ve ilgili JSON formatı Şekil 3'te gösterilmektedir. Ayrışma bakış açısı modüller ve alt modüllerden oluşmaktadır. Bu modüller arasında da parçası olma ilişkisi vardır. Bir modülün birden fazla alt modülü olabilir.

Ayrışma bakış açısının kendi içinde uygunluk kontrolü yapılabilmesi için oluşturulan kısıtlamalar şu şekildedir:

1. Bakış açısının alana özgü dili uygun formatta olmalıdır.
2. Bir modül bir kez tanımlanabilir.
3. Bütün modüller tanımlanmalıdır.
4. İçteki bir modül dıştaki modüllerine sahip olamaz.
5. Bir modül birden fazla modülün içinde yer alamaz.

Bakış açısında yer alan bu kısıtlamaları kontrol edebilmek için kullandığımız

yöntem ve ilgili algoritmamız Tablo 2'de gösterilmektedir.



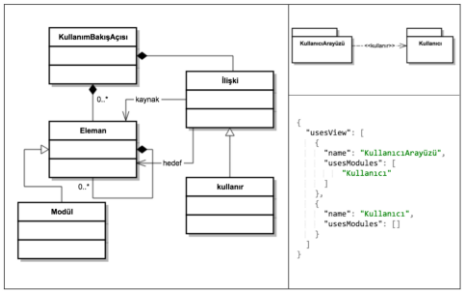
Şekil 3. Ayrışma bakış açısının soyut sözdizimi (solda) ve örnek modele karşılık gelen JSON (sağda)

3.2.2. Kullanım bakış açısı

Kullanım bakış açısı bir sistemdeki modüller ve altmodüller arasındaki ilişkileri gösterir. Bir modülün diğer bir modülü kullanıyor olması kullandığı modülün doğru işlevine bağlı olduğunu gösterir. Ayrışma bakış açısında gösterdiğimiz gibi kullanım bakış açısı için de metamodeli, örnek modeli, ilgili JSON formatını, kendi içinde uygunluk kontrolü yapabilmek için oluşturulan kısıtlamaları ve algoritmaları göstermekteyiz.

Kullanım bakış açısının kendi içinde uygunluk kontrolü yapılabilmesi için oluşturulan kısıtlamalar şu şekildedir:

1. Bakış açısının alana özgü dili uygun formatta olmalıdır.
2. Bir modül bir kez tanımlanabilir.
3. Bütün modüller tanımlanmalıdır.



Şekil 4. Kullanım bakış açısının soyut sözdizimi (solda) ve örnek modele karşılık gelen JSON (sağda)

3.3. Bakış açılarının kendi aralarındaki uygunluk kontrolü

Bir önceki bölümde, her bir bakış açısının kendi içindeki uygunluk kontrolü yöntemini belirtmiştik. Bunun için her bir bakış açısını ayrı ayrı incelemiş, metamodeli, kısıtlamaları ve ilgili algoritmaları göstermiştik. Bu bölümde bakış açılarının kendi aralarındaki uygunluk kontrolü yöntemini sunmaktayız. Kısıtlamaların kontrolü bir önceki bölümde anlatılana benzer şekilde yapılmaktadır; ancak bu sefer birbiriyle ilişkisi olan bakış açılarının uygunluk kontrolü yapılmaktadır. Tablo 1'de bakış açılarının aralarında mümkün olabilecek bağıntıları göstermiştik. Bu bağıntılara bağlı kalınarak kısıtlamalar ve ilgili algoritmaları tanımlanmıştır. Çalışmamız iki farklı örnek üzerinde gösterilmiştir.

3.3.1. Ayrışma ve kullanım bakış açılarının uygunluk kontrolü

Ayrışma ve kullanım bakış açıları arasındaki uygunluk kontrolü için tanımlanan kısıtlamalar şu şekildedir:

1. Bakış açılarının alana özgü dili uygun formatta olmalıdır.
2. Kullanım bakış açısındaki her bir modül ayrışma bakış açısında tanımlanmış olmalıdır.

Bu kısıtlamaların kontrolünü gerçekleştiren algoritmalar Tablo 4'te gösterilmiştir.

Tablo 1. Bakış açıları arasındaki muhtemel uygunluk ilişkisi

Bakış Açıları	A	K	G	KA	I	VM	BF	IS	BB	SOM	YA	PV	D	Y	IB
Ayrışma (A)	x	x	x	x	x	x	-	-	-	-	x	-	-	-	x
Kullanım (K)	x	x	-	x	-	-	-	-	-	-	-	-	-	-	-
Genelleme (G)	x	-	x	-	-	-	-	-	-	-	-	-	-	-	-
Katmanlı (KA)	x	x	-	x	-	-	-	-	-	-	-	-	-	-	-
İlgi (I)	x	-	-	-	x	-	-	-	-	-	-	-	-	-	-
Veri-Modeli (VM)	x	-	-	-	-	x	-	-	-	-	-	-	-	-	-
Bağlantı ve Filtreleme (BF)	-	-	-	-	-	-	x	-	-	-	-	-	-	-	-
İstemci-Sunucu (IS)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Birebir (BB)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Servis Odaklı Mimari (SOM)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Yayıncı-Abone (YA)	x	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Paylaşılan Veri (PV)	-	-	-	-	-	-	-	-	-	-	-	x	-	-	-
Dağıtım (D)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Yükleme (Y)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
İş Bölümü (IB)	x	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Tablo 2. Ayrışma bakış açısının kısıtlamalarını kontrol eden yöntem

Kısıtlama 1: Ayrışma bakış açısının alana özgü dilinin olduğu dosya okunur ve ayrışma model nesnesi oluşturulur. Eğer alana özgü dil formata uygun değilse ilgili hata mesajı verilir.

Girdi: Ayrışma Bakış Açısının Alana Özgü Dilinin Dosya Yolu
Çıktı: Mesaj
İşlem: Kısıtlama 1

```
1 fileContent ← readFile(viewPath)
2 model ← jsonparser(fileContent)
3 if error occurs
4      Stop and Show error
5 else return model
```

Kısıtlama 2: Ayrışma model nesnesinin bütün modüllerinin listesi alınır ve çift kopyalara bakılır. Eğer listede bir modülün ikinci kopyasına ulaşılırsa ilgili hata mesajı verilir.

Girdi: Ayrışma Modeli
Çıktı: Mesaj
İşlem: Kısıtlama 2

```
1 Create hashmap → H
2 D ← decompositionModel
3 for each module in D.getModuleList()
4      if module !exist in H
5      H.put(module.getName(), module)
6      else Show error
```

Kısıtlama 3: Ayrışma model nesnesinin içindeki bütün modüllere ve tekrar edecek şekilde bu modüllerin sahip olduğu bütün modüllere bakılır. Eğer bir modül sahip olunan listede bulunuyor ancak modelde tanımlanmamışsa ilgili hata mesajı verilir.

Girdi: Ayrışma Modeli
Çıktı: Mesaj
İşlem: Kısıtlama 3

```
 1 D ← decompositionModel
2 moduleList ← D.getModuleList()
3 for each module in moduleList
4      for each submodule in module.getHasModules()
5          if submodule !exist in moduleList
6             Show error
```

Kısıtlama 4: Ayrışma model nesnesinin içindeki bütün modüllere ve tekrar edecek şekilde bu modüllerin sahip olduğu bütün modüllere bakılır. Eğer bir modül kendi altmodüllerinin içinde bulunursa ilgili hata mesajı verilir.

Girdi: Ayrışma Modeli
Çıktı: Mesaj
İşlem: Kısıtlama 4

```
1 D ← decompositionModel
2 moduleList ← D.getModuleList()
3 for each module in moduleList
4       allSubModuleList ←D.recursiveGetAllSubmodules(module)
5       if module exists in allSubModuleList
6             Show error
```

Kısıtlama 5: Boş bir hash tablosu oluşturulur ve ayrışma model nesnesinin içindeki her bir modulün ve o modülün sahip olduğu bütün modüllere bakılır. Eğer sahip olunan modül hash tablosunda yok ise tabloya eklenir. Diğer durumda ise ilgili hata mesajı verilir.

Girdi: Ayrışma Modeli
Çıktı: Mesaj
İşlem: Kısıtlama 5

```
1 D ← decompositionModel
2 moduleList ← D.getModuleList()
3 Create hashmap → H
4 for each module in moduleList
5      for each hasModule in module.getHasModules()
6             if hasModule !exist in H
7                    H.put(hasModule,module)
8             else Show error
```

Tablo 3. Kullanım bakış açısının kısıtlamalarını kontrol eden yöntem

Girdi: Kullanım Bakış Açısının Alana Özgü Dilinin Dosya Yolu
Çıktı: Mesaj
İşlem: Kısıtlama 1

```
1 fileContent ← readFile(viewPath)
2 model ← jsonparser(fileContent)
3 if error occurs
4   Stop and Show error
5 else return model
```

Girdi: Kullanım Modeli
Çıktı: Mesaj
İşlem: Kısıtlama 2

```
1 Create hashmap → H
2 U ← usesModel
3 for each module in U.getModuleList()
4   if module !exist in H
5         H.put(module.getName(), module)
6    else Show error
```

Girdi: Kullanım Modeli
Çıktı: Mesaj
İşlem: Kısıtlama 3

```
1 U ← usesModel
2 moduleList ← U.getModuleList()
3 for each module in moduleList
4    for each usedModule in module.getUsesModules()
5       if usedModule !exist in moduleList
6          Show error
```

3.3.2. Ayrışma ve katmanlı bakış açılarının uygunluk kontrolü

Ayrışma ve katmanlı bakış açıları arasındaki uygunluk kontrolü için tanımlanan kısıtlamalar şu şekildedir:

1. Bakış açılarının alana özgü dili uygun formatta olmalıdır.
2. Katmanlı bakış açısındaki her bir modül ayrışma bakış açısında tanımlanmış olmalıdır.
3. Ayrışma bakış açısındaki her bir modül ve bu modülün altmodülleri, katmanlı bakış açısında aynı katmanda yer almalıdır.

Bu kısıtlamaların kontrolünü gerçekleştiren algoritmalar Tablo 5'te gösterilmiştir.

Tablo 4. Ayrışma ve kullanım bakış açılarının kısıtlamalarını kontrol eden yöntem

Girdi: Ayrışma ve Kullanım Bakış Açılarının Alana Özgü Dillerinin Dosya Yolu
Çıktı: Message
İşlem: Kısıtlama 1

```
1 fileContent ← readFile(viewPath)
2 model ← jsonparser(fileContent)
3 if error occurs
4       Stop and Show error
5 else
6       return model
```

Girdi: Ayrışma ve Kullanım Modelleri
Çıktı: Mesaj
İşlem: Kısıtlama 2

```
1 U ← usesModel.getModuleList()
2 D ← decompositionModel.getModuleList()
3 for each module in U
4       if module !exist in D
5             Show error
```

Tablo 5. Ayrışma ve katmanlı bakış açılarının kısıtlamalarını kontrol eden yöntem

Girdi: Ayrışma ve Katmanlı Modelleri
Çıktı: Mesaj
İşlem: *Kısıtlama* 2

```
D ← decompositionModel.getModuleList()
for each layer in layeredModel
      for each module in layer.getModuleList()
            if module !exist in D
                  Show error
      return model
```

Girdi: Ayrışma ve Katmanlı Modelleri
Çıktı: Mesaj
İşlem: *Kısıtlama* 3

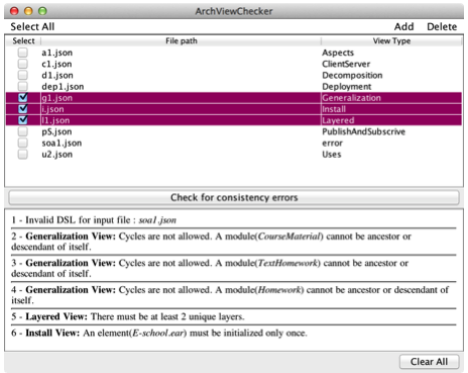
```
Create allModulesInAllLayersList
L ← layeredModel.getLayerList()
for each layer in L
      Create allInnerModuleList
      for each module in layer. getHasModules()
      allInnerModuleList.add(module)
H←decompositionModel.getAllInnerModuleList(module)
            allInnerModuleList.add(H)
      for each module in allInnerModuleList
            if module !exist in allModulesInAllLayersList
                  allModulesInAllLayersList.add(module)
            else Show error
```

4. Araç

ArchViewChecker aracı Java programlama diliyle IntelliJ IDEA [7] ortamında geliştirilmiştir. Aracı bilgisayarınızda kullanabilmek için JDK 5 ya da daha üst versiyonları sisteminizde yüklü olması gerekmektedir. Araç [8] adresinden çalıştırılabilir .jar dosyası indirilerek kullanılabilir. Aracın anlık durum görüntüsü Şekil 5'te gösterilmektedir.

Aşağıdaki adımlar aracın çalışma mekanizmasını göstermektedir:

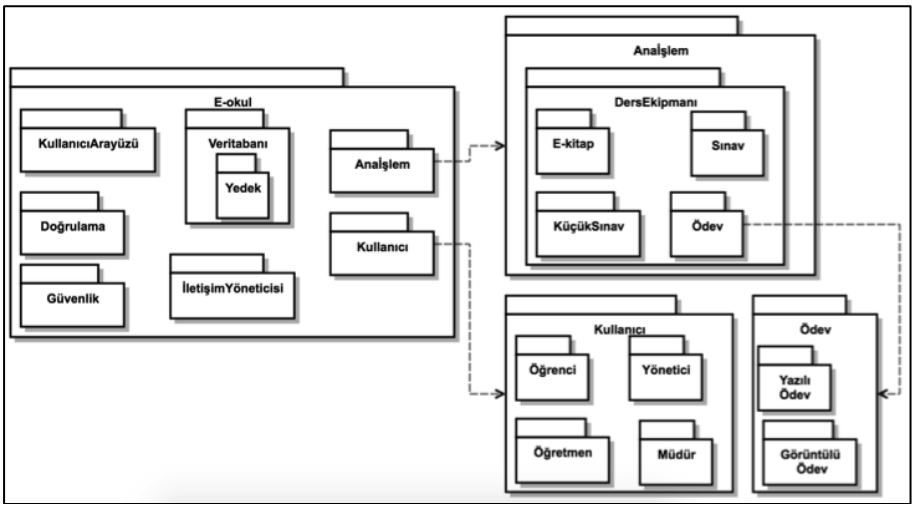
1. Bakış Açılarını Alma
   Kullanıcı bakış açılarını json formatında yaratır ve araca yükler.
2. Uygunluk Kontrolü Yapma ve Hataları Gösterme
   Bakış açıları yüklendikten sonra, araç bakış açılarının kendi aralarında ve kendi içlerinde olan tutarsızlıkları belirler. Eğer yüklenilen bakış açısının alana özgü dili hatalıysa, araç tutarlılık kontrolü yapmaz ve kullanıcıya ilgili hatayı gösterir. Eğer bakış açısı düzgün formatta yüklenmişse tutarlılık kontrolü yapmaya devam eder ve bakış açılarının kendi içlerinde ve birbirleriyle olan hatalarını kullanıcıya gösterir.



Şekil 5. ArchViewChecker aracının anlık durum görüntüsü

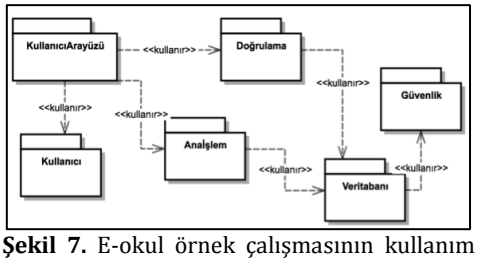
5. Örnek Çalışma

Bu kısımda örnek çalışmamız olan E-okul adını verdiğimiz, Görünümler ve Ötesi yaklaşımı kullanarak hazırladığımız mimari tasarımını sunuyoruz. E-okul sistemi öğrenciler, öğretmenler ve okul yönetimi arasında iletişim kurmayı sağlayan bir uygulamadır. Bu uygulama sayesinde



Şekil 6. E-okul örnek çalışmasının ayrışma bakış açısı

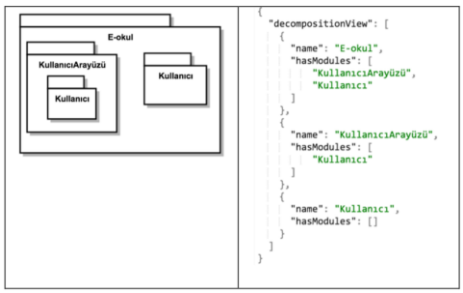
öğrenciler internet ortamındaki ders ekipmanlarına çevrimiçi bir şekilde ulaşabileceklerdir. Aynı zamanda sınav ve ara sınavlarının tarihlerinden, ödev ve dönem ödevlerinin teslim tarihlerinden e-posta vasıtasıyla haberdar olabileceklerdir. Öğretmenler e-posta yoluyla öğrencilere ders materyallerini paylaşabilecekler, okul yöneticileri ve öğretmenler e-posta sayesinde iletişim kurabileceklerdir. Okul yöneticileri ve öğretmenler öğrenci etkinliklerini, okul yöneticileri de öğretmenlerin etkinliklerini uygulama sayesinde takip edebilecektir. Kullanıcılar bu sistemi cep telefonlarında, bilgisayarları ya da tabletlerinde kullanabileceklerdir. Şekil 6 ve 7'de örnek çalışmamızın Ayrışma ve Kullanım bakış açıları gösterilmektedir.



Şekil 7. E-okul örnek çalışmasının kullanım bakış açısı

Ayrışma bakış açısındaki E-okul modülü temel modüldür ve içerisinde KullanıcıArayüzü, Kullanıcı, Veritabanı, Doğrulama, Güvenlik, AnaİşlemIletişimYöneticisi gibi modülleri içermektedir (Şekil 6). Bu modüller de kendi içlerinde ayrışarak alt modüller içermektedirler. Kullanım bakış açısında da bu modüllerden bazılarının birbirlerini kullanma ilişkileri gösterilmiştir (Şekil 7). Şekil 6 ve 7'deki bakış açıları E-okul sistemi için tasarlanmış doğru bakış açılarıdır. Aracımızın uyumsuzlukları nasıl bulduğunu göstermek amacıyla hatalı bakış açıları oluşturup aracımıza yükleyeceğiz.

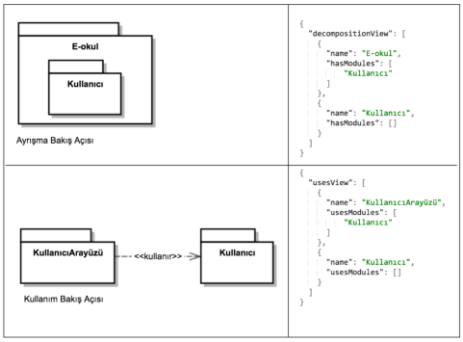
Şekil 8 ve 9'da hatalı bakış açıları gösterilmektedir. Şekil 8'de "Kullanıcı" modülü hem "E-okul" hem de "KullanıcıArayüzü" modülü içerisinde yer almaktadır. Bu durum ayrışma bakış açısının kısıtlamalarından (Kısıtlama 5) "Bir modül birden fazla modülün içinde yer alamaz." ifadesiyle ters düşmektedir.



Şekil 8. E-okul örnek çalışmasının hatalı ayrışma bakış açısı, soyut modeli (solda) ve JSON örneği (sağda)

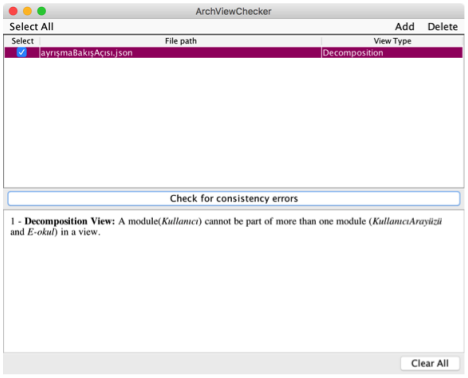
Şekil 9'da kullanım bakış açısında yer alan "KullanıcıArayüzü" modülü ayrışma bakış açısında bulunmamaktadır. Bu durum, iki bakış açısının kısıtlamalarından (Kısıtlama 2) "Kullanım bakış açısındaki her bir modül ayrışma bakış açısında tanımlanmış olmalıdır." ifadesiyle ters düşmektedir.

Şekil 8 ve 9'da aktardığımız bakış açılarındaki hataları ArchViewChecker adlı aracımız sayesinde bulmaktayız. İlgili hataları bulan aracın görüntüleri Şekil 10 ve 11'de verilmiştir. Aracımız, yüklenen bakış açıları için önce kendi içlerinde sonra da (eğer yüklenmişse) diğer bakış açılarıyla olan uygunluk kontrolünü yapmaktadır.

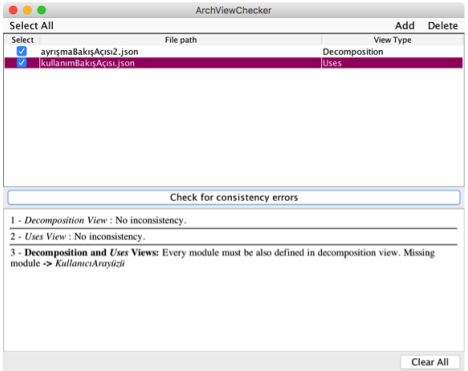


Şekil 9. E-okul örnek çalışmasının hatalı ayrışma ve kullanım bakış açıları, soyut modelleri (solda) ve JSON örnekleri (sağda)

Örnek çalışmamızda yer alan her bir bakış açısı bağlı bulunduğu bakış açısı grubunun kurallarına uygun olarak tanımlanmıştır [9]. Temelde bütün bakış açıları aynı sistemi göstermektedir ve önemli olarak hepsi kendi içinde ve birbiri arasında tutarlı olarak tasarlanmıştır. Mimaride olan tutarsızlıklar, bakış açılarının ilk tanımlandığı zaman ortaya çıkabilir. Ayrıca, gelişmekte olan gereksinimler neticesinde bu bakış açılarının değişmesi gerekebilir. Mimari ayrışma problemi literatürde fazlaca ele alınmıştır [3]. Ancak, buradaki sorun daha çok mimarinin koddan ayrışmasıyla ilgilidir. Aynı ölçüde mimari tasarımındaki tutarlılık da önemlidir. Genellikle, bakış açılarındaki uygunluk kontrolü manuel olarak yapılmaktadır. Küçük sistemlerde bu yöntem yararlı olsa da sayıca daha çok bakış açısı gerektiren büyük ölçekli sistemlerde makul sonuçlar vermeyebilir.



Şekil 10. Şekil 8'de verilen ayrışma bakış açısındaki hataların bulunması



Şekil 11. Şekil 9'da verilen ayrışma ve kullanım bakış açılarındaki hataların bulunması

6. Benzer Çalışmalar

Bizim çalışmamıza benzer olarak Michel ve Galal-Edeen de çalışmalarında [5] birden çok mimari bakış açısını ve aralarındaki uygunsuzlukları incelemiş ve bu uygunsuzlukları bulmaya yarayan bir sistem oluşturmuşlardır. Yazarlar Mimari Tanımı'nda (Architecture Description) kullanılacak olan bakış açıları arasındaki uygunsuzlukları belirlemişler ve bu uygunsuzlukları tespit etmeye çözüm bulmuşlardır. Uygunluk kontrolü yapabilmek için öncelikle mimari tanımında kullanılan bakış açılarını alıp ilgili metamodellerini ve Extensible Markup Language Metadata Interchange (XMI) dökümanlarını oluşturmuşlardır. Uygunluk kontrolü kurallarını bu XMI dökümanlarına uygulayıp sonucunda yazılımcıya hataları gösteren bir rapor sunmaktadırlar. Bu rapor sayesinde yazılımcı bakış açılarını değiştirip düzenleme yapabilmektedir. Bizim çalışmamızdan farklı olarak bu çalışmada uygunsuzluk tanımlarının yapılması ve uygunluk kontrolü otomatikleştirilmemiştir.

Bileşen ve Bağlayıcı bakış açılarındaki (Component and Connector views) bakış açılarıyla Mimari Tasarımı Kararları (Architectural Design Decision) arasındaki uyumsuzlukları tespit edebilmek için Lytra ve Zdun bir araç geliştirmiştir [10]. ADvISE [11] adlı araçları tekrar kullanılabilen mimari tasarımı kararlarını modellemek için, VbMF adlı araçları da model güdümlü mimari bakış açılarını modellemek için kullanmışlardır. Mimari Bilgisi (Architectural Knowledge) diliyle mimari kararlarını tasarıma dönüştürmüşlerdir. Bileşen ve bağlayıcı bakış açılarının örnekleri mimari tasarımı kararları kullanılarak oluşturulmuştur. Kısıtlama onaylayıcı yardımıyla mimari tasarımı kararları ve bakış açıları arasındaki uyumsuzluklar kontrol edilmiştir. Bakış açılarındaki hataların giderilmesi otomatik bir şekilde yapılırken, mimari tasarımı kararlarının düzeltilmesi yazılımcı tarafından yapılmaktadır. Bizim yöntemimizden farklı olarak [10] çalışmasında mimari tasarımı kararları kullanılarak uygunluk kontrolü yapılmaktadır. Biz ise mimari bakış açıları modellerinin bizzat kendilerine yöntemimizi uyguladık.

Tekinerdoğan ve diğer yazarlar tarafından ortaya konulan çalışmada [12] İlgi İzlenebilme Metamodeli (Concern Traceability Metamodel) adlı bir metamodel yaratılıp mimari bakış açılarının durumları izlenmiştir. Bu konuda ilgiye yönelik yazılım geliştirme alanındaki yöntemler de kullanılmıştır [13-14]. Sunulan metamodel Extensible Markup Language (XML) Document Type Definition (DTD) kullanılarak kodlanmış ve XML ile mimari bakış açıları modellenmiştir. DTD ve Xquery kullanılarak mimari bakış açılarının kendi içlerinde ve aralarındaki izlenebilirlik bağlantıları oluşturulmuş, sonuçlar da bir XML dosyasında kullanıcıya sunulmuştur. Mimari bakış açılarından farklı olarak, Tekinerdoğan ve diğer yazarlar tarafından yürütülen çalışmada [15] uygunluk analizi ürün hattı mühendisliği alanına uygulanmıştır. Bizim yöntemimize benzer şekilde uygulama mimarisiyle ürün hattı mimarisinin uygunluğu Yansıma Modeli yöntemi kullanılarak ele alınmıştır.

7. Sonuç

Kodla yazılım mimarisi arasındaki uyumsuzlukları bulmaya yarayan çeşitli mimari uyum analizi yöntemleri bulunmaktatır. Bu makalede, yazılım mimarisi bakış açılarının kendi içlerinde ve birbirleriyle olan uyumsuzluklarını bulmaya odaklandık. Bu amaç doğrultusunda, uyumsuzlukları bulmaya yarayan sistematik bir yöntem sunduk ve ArchViewChecker isimli bir araç geliştirdik. Yöntemimizi ve geliştirdiğimiz aracı değerlendirmek için örnek bir çalışma üzerinde çalıştık.

Gelecek bir çalışma olarak, aracımızın kapsamını genişleterek endüstriyel çalışmalarda kullanacağız.

Kaynakça

[1] Tekinerdoğan, B. 2014. Software Architecture in Volume I. Computer Science Handbook, 2nd Edition, , CRC Press-Taylor and Francis Group, 3816s.

[2] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., Stafford, J. 2010. Documenting Software Architectures: Views and Beyond. 2nd edition. Addison-Wesley, 592s.

[3] Murphy, G., Notkin, D., Sullivan, K. 2001. Software reflexion models: Bridging the gap between design and implementation, *IEEE Transactions on Software Engineering*, Cilt. 14, No. 4, s. 364-380.

[4] Adersberger, J., Philippsen, M. 2011. ReflexML: UML-Based Architecture-to-Code Traceability and Consistency Checking. 5th European Conference on Software Architecture (ECSA 2011), 344-359.

[5] Michel, M.M., Galal-Edeen, G.H. 2009. Detecting inconsistencies between software architecture views. International Conference on Computer Engineering and Systems, 429-434.

[6] Introducing JSON. http://json.org/ (Erişim Tarihi: 01.12.2015).

[7] IntelliJ IDEA, The Most Intelligent Java. https://www.jetbrains.com/idea/ (Erişim Tarihi: 01.12.2015).

[8] ArchViewChecker.jar. https://goo.gl/m01yi4 (Erişim Tarihi: 01.12.2015).

[9] Ekşi, G.E. 2015. Model Güdümlü Yazılım Mimarisi Bakış Açılarında Uygunluk Kontrolü. Bilkent Üniversitesi, Mühendislik ve Fen Bilimleri Enstitüsü, Yüksek Lisans Tezi, ss 38-53, Ankara.

[10] Lytra, I., Zdun, U. 2014. Inconsistency Management between Architectural Decisions and Designs Using Constraints and Model Fixes. Australian Software Engineering Conference, 230-249.

[11] Architectural Design Decision Support Framework (ADvISE). https://swa.univie.ac.at/Architectural_Design_Decision_Support_Framework_(ADvISE) (Erişim Tarihi: 01.12.2015).

[12] Tekinerdoğan, B., Hofmann, C., Akşit, M. 2007. Modeling Traceability of Concerns for Synchronizing Architectural Views, *Journal of Object Technology*, Cilt. 6, No. 7, Özel Konu: İlgi Odaklı Modelleme, s. 7-25.

[13] Bakker, J., Tekinerdogan, B., Aksit, M. 2005. Characterization of Early Aspect Approaches, in: Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design Workshop. The International Conference on Aspect-Oriented Software Development, 14 Mart, Chicago.

[14] Chitchyan, R., Rashid, A., Sawyer, P., Bakker, J., Alarcon, M.P., Garcia, A., Tekinerdogan, B., Clarke, S., Jackson, A. 2005. Early Aspects at ICSE 2007: Workshop on Aspect-Oriented Requirements Engineering and Architecture Design. International Conference on Software Engineering (ICSE Companion), 127-128.

[15] Tekinerdoğan, B., Çilden, E., Erdoğan, O.O., Akdağ, O. 2014. Architecture Conformance Analysis Approach within the Context of Multiple Product Line Engineering. Australasian Software Engineering Conference (ASWEC), 7-11 Nisan, 25-28.