YAZILIM ÜRÜN HATTINDA YETENEK TABANLI YAZILIM BİLEŞENLERİNİN DOĞRULANMASI

Mert Burkay ÇÖTELİ, Mehmet Emre ATASOY[2]

[1,2]Aselsan A.Ş. SST, 06172, Yenimahalle, Ankara

[1]mbcoteli@aselsan.com.tr, [2]eatasoy@aselsan.com.tr

Anahtar Kelimeler: Yazılım Ürün Hattı, Yetenek ağacı, Doğrulama Yöntemi, Birim Seviyesinde Testler, Yazılım Bileşeni

Özet. Yazılım ürün hattı (YÜH), belirli bir çalışma alanının ihtiyaçlarını karşılamak için, bileşen ve ürün seviyesinde yetenek ağacıyla uyumlu ortak yazılım bileşenleriyle hızlı bir şekilde ürün çıkartmaya dönük bir yazılım geliştirme yöntemidir. Yazılım ürün hattı yaklaşımında yazılım geliştirmeye dönük farklı çalışmalar bulunmakta olup, bileşen seviyesinde doğrulama, geçerli kılma açısından çok fazla çalışma bulunmamaktadır.

Bu çalışma kapsamında, ilk aşamada her bileşen için ürün ağacı kullanılarak olası varyasyon kümeleri oluşturulmuştur. Sonrasında, birim seviyesinde testler bu varyasyon kümeleri yardımıyla tanımlanmış ve her bir varyasyon elemanı için test tanımları çıkartılmıştır. Bu sayede yazılım bileşeninin nihai ürünlerde kullanılacak farklı tipleri oluşturulan testlere göre doğrulanmış ve birim seviyesinde otomatik test tanımlarının koşturulması amaçlanmıştır.

1 GİRİŞ

YÜH yetenek ağacı ve yeniden kullanılabilirlik yöntemleri yardımıyla yazılım maliyetlerini düşürmeye dönük uygulanmakta olan bir yazılım geliştirme yöntemidir. Fakat farklı ürün varyasyonları ile birlikte yazılım test ve doğrulama süreçleri de etkilenmektedir. İki farklı mühendislik alanı YÜH içerisinde bulunmaktadır. Bu mühendislikler Alan ve Uygulama mühendisliğidir [4]. Alan mühendisliği aşamasında YÜH genel yetenek ağaçları çıkartılmakta, bu yetenek ağacı ile bileşen ayrımları yapılmakta ve olası ürün varyasyonları belirlenmektedir. Genel bir yetenek ağacının yanında her bir bileşen için de bir yetenek ağacı oluşturulmaktadır. Uygulama mühendisliğinde ise alan mühendisliğinde elde edilen yetenek ağacı kullanılarak nihai ürün modelleri çıkartılmaktadır. Bu çalışma kapsamında Alan mühendisliğinde tanımlanmış olan yazılım bileşenleri için birim testleri seviyesinde bir doğrulama yöntemi önerilecek ve bu yöntem değerlendirilecektir.

2 YAZILIM ÜRÜN HATTI GELİŞTİRME YÖNTEMİ

2.1 YAZILIM BİLEŞENLERİNİN KODLANMASI VE BİLEŞEN TABANLI DOĞRULANMASI

Yazılım ürün hattında yazılım bileşenlerinin kodlanması normale göre farklıdır. Yazılım ürün hattı yaklaşımı alan analizine dayalı olarak oluşturulan bir alt yapıdır. Bu bağlamda bileşenler de alan analizine göre çeşitlilik gösterir. Yazılım bileşenleri çeşitli varyasyonlardan oluşmaktadır. Bu varyasyonların seçimine göre çalışan yazılım birimlerinin yetenek yönetimi yetenek ağacı kullanılarak yapılır [2].

Yazılım bileşenleri, yetenek ağacından seçilecek varyasyonlara göre farklı şekilde çalışmaktadır. Yazılım yetenek ağacının örnek bir parçası aşağıda verilmiştir.

Şekil -1 Ürün yetenek ağacı

Aslında yazılım bileşeni birbirine alternatif olarak çalışan yeteneklerin hepsini içermektedir. Ancak nihai sistemde bu yetenekler aynı anda çalışmayacaktır. Beraber çalışacak yetenekler birer küme oluşturmakta olup her bir küme ayrı ayrı nihai ürünlerde kullanılacaktır. Bu durum, yazılım bileşeninin bütün yeteneklerinin aynı anda doğrulanmasını gereksiz ve etkisiz kılmaktadır. Yazılım bileşeni zaten bu yetenek kümelerini içermektedir.

Bir yazılım sistemindeki hataları (bug) bulmak birim testler ile mümkün değildir. Çünkü birim testlerin yaptığı iş yazılımın en küçük parçalarını kendi içerisinde test etmektir. Bu küçük parçaların kendi içlerinde çalışıyor olması, yazılımın gerçek kullanıcılar tarafından kullanılmaya başladığı zaman bir bütün olarak çalışacağını göstermez. Bir yazılım sistemi, onu oluşturan parçaların toplamından çok daha fazlasıdır. Dolayısıyla bu bütünü test etmek için farklı yöntemler kullanmak gerekir. Fonksiyonel test ve entegrasyon testi bunlara örnek verilebilir. Ancak birim testler, yazılım birimlerini birim seviyesinde doğrulamak ve yine birim seviyesinde hataları bulmak için kullanılır.

Standart birim test yaklaşımında yazılım biriminin belli bir oranı kapsanacak şekilde birim testler yazılmaktadır. Ancak yazılım ürün hattı yaklaşımında önce de bahsettiğimiz gibi yazılım birimi birden çok varyasyonda bulunmaktadır. Yazılım biriminin bir varyasyonu diğerinin çalışmasını pozitif ya da negatif yönde etkileyebileceği için birim testlerin yetenek modeli düşünülmeden koşturulması anlamlı olmayacaktır. Bu çalışma kapsamında birim testlerin varyasyon kümelerine göre gösterdikleri değişkenliklerden bahsedilecektir.

Yetenek ağacına göre yazılım geliştirme aşamasında, yazılım birim testlerinin de yetenek ağacı baz alarak gerçeklenmesi gerekmektedir. Birim test yazılan her bir fonksiyonun olası girdileri yetenek ağacından elde edilen varyasyonlara göre oluşturulur. Bu varyasyonlar dışında herhangi bir birim test kapsamasına gerek yoktur. Varyasyon kümesinin her bir elemanı için birim test kümeleri oluşturulur. Bu birim test kümeleri sadece ilgili varyasyonu kapsamaktadır. Nihai sistemde kullanılması mümkün her bir varyasyon bu sayede birim seviyesinde doğrulanmış olur.

Bu yöntemdeki amaçlar

1) Nihai ürünlerde beraber çalışacak yeteneklerin birbirlerine yan etkilerini ortaya çıkarmak: Klasik birim test yaklaşımında bu bilgi elde edilmemektedir. Çünkü, bütün yetenekler beraber test edilmektedir ve her bir testin yan etkisi ortaya çıkarmaktadır. (bu yan etkiler: ortak bellek kullanımı, veri tabanı erişimi, işletim sistemi kaynakları kullanımı vb olabilir)

2) Aslında hiçbir zaman oluşmayacak hatalarla uğraşılmasının engellenmesi: Klasik yaklaşımda rastgele birim test koşturulan ve birbirlerine yan etki yaratan iki yetenek nihai üründe beraber çalışmayabilir. Bu iki yeteneğin birbirlerine karşı yan etki yaratması olası ve kabul edilen bir durum olabilir. Nihai üründe hiçbir zaman beraber çalışmayacak ve birbirinin alternatifi olan bu iki yeteneği beraber doğrulamak anlamsızdır.

2.2 YETENEK AĞACININ KULLANILMASI

Ürün ağacı farklı bileşen ağaçlarının birleştirilmesi yardımıyla oluşturulmaktadır. Alan mühendisliğindeki genel yetenek ağacının U_d olduğu düşünülürse uygulama mühendisliğinden çıkacak her bir ürünün yetenek ağaçları ($U_{a1}, U_{a2}, U_{a3} \ldots U_{an}$) , U_d üzerinden türetilebilir. U_d ise alan mühendisliğinde türetilen tüm bileşenlerin ürün ağaçlarının birleşimi ile oluşturulmaktadır. B_{dk} bileşene ait yetenek ağacını göstermektedir.

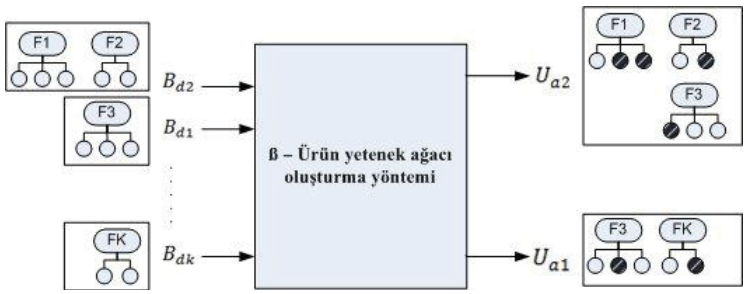
$$U_d = B_{d1} \cup B_{d2} \cup B_{d3} \cup \ldots \cup B_{dk} \tag{1}$$

Uygulama mühendisliğinde geliştirilecek bir ürün için ilk olarak alan mühendisliğindeki bileşenlerin ürün ağacı kullanılarak ürüne özgü yetenek ağaçları tanımlanmaktadır. Bu aşamada, B_{dn} üzerinden varyasyonlar oluşturulmakta ve her ürünün yetenek ağacı çıkartılmaktadır. Bileşen üzerinde varyasyon oluşturma işlemi **ß** olarak tanımlanırsa uygulama mühendisliğindeki nihai ürün ağacının oluşturulma yöntemi matematiksel olarak aşağıdaki gibi ifade edilebilir.

$$ß_n(U_d) = ß_n(B_{d1} \cup B_{d2} \cup B_{d3} \cup \ldots \cup B_{dk}) = U_{a2} \tag{2}$$

$$= ß_n(B_{d1}) \cup ß_n(B_{d2}) \cup ß_n(B_{d3}) \cup \ldots \cup ß_n(B_{dk}) = U_{an} \tag{3}$$

YÜH 'da ürüne dönük yetenek ağacı oluşturma yöntemi Şekil – 2 'de görülmektedir.



Şekil -2 Ürün yetenek ağacı oluşturma yöntemi

Her bir bileşen için N_k farklı varyasyon olacağını düşünürsek alan mühendisliğindeki yetenek ağacı kullanılarak çıkartılabilecek olası ürün sayısı aşağıdaki şekilde ifade edilebilir [1].

$$N_{maks} = \prod_1^k N_k \tag{4}$$

Bu ifade ile olası ürünler için kompleksite 'nin O($N_k{}^k$) olduğu ve çok farklı ürün çeşidinin yazılım ürün hattı yardımıyla çıkartılabileceği görülmektedir. Alan mühendisliği çerçevesinde birim testleri tanımlamak; tüm varyasyonları göz önünde bulundurmak anlamına gelmelidir. Tek bir denek varyasyon seti seçerek test girdilerini oluşturmak çok gerçekçi doğrulama sonuçları sağlamayacaktır.

3 YETENEK AĞACI TABANLI DOĞRULAMA YÖNTEMİ

Farklı ürünler aynı yazılım bileşenlerine farklı varyasyonların bağlanmasıyla oluşturulabilmektedir. Bu sebeple, yazılım bileşenine ait metod ve sınıflar farklı ürünlerde farklı şekilde sonuç veriyor olabilirler. Yazılım mimarisine bağlı olarak metodlar varyasyon değişikliği sebebiyle değişikliğe uğramış veya metodun önkoşulları ile son koşulları sınıf değişmezlerine bağlı olarak değiştirilmiş olabilir. Bu aşamada birim testlerin uygulanacağı metodlar iki tipte incelenebilir [3].

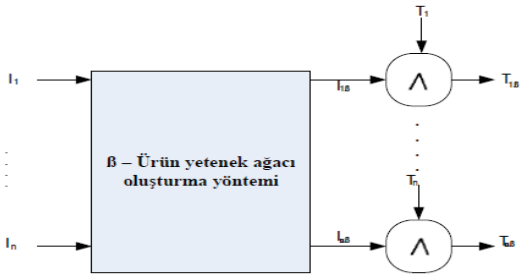
a) İçeriği sabit olan metodlar
b) İçeriği değişen metodlar (Override)

Literatürde metodların testlerinin tam kapsaması düşünülerek farklı doğrulama yöntemleri önerilmektedir. Bu yöntemlerden Concolic test ile yazılım metodu beyaz kutu olarak kabul edilip olası tüm test girdileri çalıştırma yolu ağacı yardımıyla tanımlanabilmektedir [5]. Test girdileri bazı araçlar yardımıyla otomatik olarak da çıkartılabilmektedir. Bu çalışmada YÜH 'da Concolic test yöntemi kullanılmış olup farklı yöntemlerle de test girdileri oluşturulabilir. YÜH üzerinde varyasyon bağlamasından sonra metodların çalışma şekillerinin değişeceği ve iki tip metodun oluşacağı öngörülmektedir.

YÜH 'te Concolic test metodunu kullanarak test girdilerini oluşturduktan sonra bu test girdileri üzerinden olası varyasyonlar için güncellemeler yapmak gerekmektedir. Concolic test sonrası oluşturulacak veri kümeleri (5) 'te görülmektedir.

$$T_1(girdi, \text{sonuç}), T_2(girdi, \text{sonuç}) \ldots \quad (5)$$

Sınıf değişmezleri testin öncesinde, sırasında ve sonrasında değişmemesi gereken değerlerdir. Bu sebeple hem test girdisinde hem de sonucunda bu değişmezlerin yer alması gerekmektedir. YÜH varyasyon bağlaması sonrası oluşturulacak (a) tipindeki metodlarda sınıf değişmezlerinin incelenmesi gerekmektedir. Bu değişmezler test girdi ve sonuçları direk etkilemektedir. Şekil -3 'te sınıf değişkenlerinin yetenek ağacı bağlantılı değişimi gösterilmektedir.



Şekil -3 (a) tipi metod test seti oluşturma yöntemi

(a) Tipindeki metodlarda Concolic test sonrası oluşturulmuş test setleri 6. denklemdeki gibi güncellenebilir. Yani metodun tam doğrulanması için her olası ürün konfigürasyonu kullanarak test setlerini oluşturmak gerekmektedir.

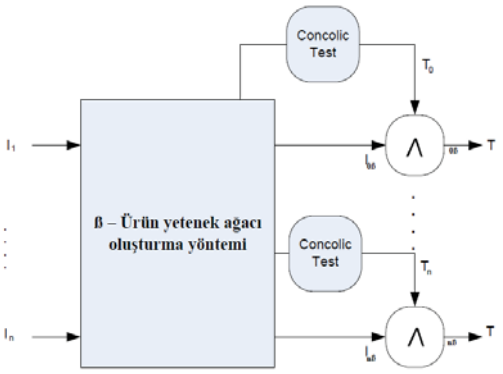
$$T_1(girdi \wedge I_s, \text{sonuç} \wedge I_s), T_2(girdi \wedge I_s, \text{sonuç} \wedge I_s) \ldots \quad (6)$$

Aynı fonksiyon farklı varyant değerleri için farklı değerler verebilir. Fonksiyon işlevi aynı olmasına sınıf değişmezleri farklılıklarından dolayı aynı fonksiyonun farklı test girdileriyle test edilmesi gerekmektedir. Örneğin özellik seti olarak silah tipi seçilirse A, B veya C tipindeki silahlar için sınıf değişmezleri atış menziline göre değişkenlik gösterir. Aşağıdaki fonksiyon bu sınıf değişmezine göre işlem yapmakta olup atış yapılıp yapılamayacağına dönük çıktı vermektedir.

```
//When Silah Tipi = A || B || C
    public Boolean CheckItCanShoot(Location targetloc, Location
ownloc)
```

Bu fonksiyon için tasarlanacak olan birim testlerinin farklı varyant tipleri için değerlendirilmesi gerekmektedir. Varsayılan bir varyant tipi için birim test girdileri tanımlanacak, sonrasında bu birim test girdileri varyant değişimlerine göre değişkenlik gösteren sınıf değişmezleri ile güncellenecektır. Test girdilerinin her bir varyant için baştan çıkartılmasına gerek yoktur.

(b) tipindeki metodlarda ise sınıf değişmezlerinin ve concolic test yönteminin değişen her varyasyon için incelenmesi gerekmektedir. Şekil - 4 'te sınıf değişkenlerinin yetenek ağacı bağlantılı değişimi gösterilmektedir.



Şekil -4 (b) tipi metod test seti oluşturma yöntemi

(b) Tipindeki metodlarda ise Concolic test sonrası oluşturulmuş test setleri 7. denklemdeki gibi güncellenebilir. Bu koşulda varyasyonun bağlanması sonrası her metod için concolic test metodunu uygulayıp, sonrasında sınıf değişkenlerini kullanarak bu test setlerini güncellemek gerekmektedir.

$$T_{1\mathrm{k}}(girdi \wedge I_s, \text{sonuç} \wedge I_s), T_{2\mathrm{k}}(girdi \wedge I_s, \text{sonuç} \wedge I_s) \ldots \quad (7)$$

Bu tipteki metodlar için test girdi seti oluşturma yöntemi aşağıdaki örnekle açıklanabilir. YÜH hattında özellik seti olarak açı tipi ve buna karşılık gelen varyant değerleri olarak da Derece, Radyan ve Grad seçilebildiğini düşünelim. (i) numaralı fonksiyon varyant tipi Derece olarak seçildiğinde yapılacak olan işlemleri göstermekte olup, (ii) numaralı fonksiyon varyant değeri olarak Radyan seçildiğinde çağırılacak olan fonksiyondur. YÜH 'dan çıkacak ürünlerde sadece Derece cinsinden bir varyant kullanımı seçilebilir veya Radyan ile birlikte Derece kullanımı seçilebilir. Varyant seçimleri bu fonksiyonları direk etkileyeceğinden dolayı fonksiyonların isimleri aynı kalmasına rağmen fonksiyonların iki duruma göre farklı yazılması gerekmektedir.

```
    (i)    //when Default Degree
         public float CalculateAngleBetweenPos(float targetangle1,
float targetangle2)

    (ii)   //when Radian
         public override float CalculateAngleBetweenPos(float
targetangle1, float targetangle2)
```

Yetenek ağacı yardımıyla ürün oluşturulurken direk etkilenebilecek olan bu tipteki fonksiyonlar için test girdilerinin baştan oluşturulması gerekmektedir. Birim testlerin yetenekler ürüne eklendikten sonra yapılması daha uygun olacaktır. Örneğin yukarıdaki örnek için 3 farklı test veri setinin kullanılması gerekmektedir.

4 SONUÇ VE DEĞERLENDİRME

Bu çalışma kapsamında YÜH kullanılarak geliştirilen yazılımların birim testlerinin farklı ürün varyasyonları kapsamında çıkartılmasına dönük bir yöntem önerilmiştir. Bu yöntem ile farklı yeteneklerin varyasyon seçimlerine bağlı olarak birim test girdileri güncellenmiştir. Tüm varyasyonların birlikte çalışamayacağı düşünüldüğünde nihai ürün içerisinde yer alabilecek varyasyonların seçilip bu varyasyonlara göre test girdilerinin şekillendirilmesi önemlidir. Test etkisinin incelenmesi gelecekte yapılacak bir çalışma olarak değerlendirilebilir. YÜH 'da geleneksel yöntemlerle birim testlerinin gerçekleştirilmesi ve aynı yazılımda bu önerilen yöntemle birim testlerinin gerçekleştirilmesi sonrasında test etkisindeki değişim incelenebilir. Test girdilerinin yetenek bağlanması sonrasında bir yazılım yardımıyla otomatik olarak çıkartılması amaçlanmakta olup gelecek bir çalışma olarak hedeflenmektedir.

5 REFERANSLAR

1. Çöteli M. B.; "Testing effectiveness and effort in Software Product Lines", M.Sc.Thesis, METU, Ankara, 2013
2. Atasoy E.; "Hierarchical Variability Management in Software Product Lines", M.Sc.Thesis, METU, Ankara, 2013
3. Bruns D.; Klebanov V.; Schaefer I.; "Verification of Software Product Lines with Delta-oriented Slicing", 2007
4. Kolb, R.; Muthig, D.; "Making Testing Product Lines More Efficient by Improving the Testability of Product Line Architectures", in Workshop on Role of Software Architecture for Testing and Analysis, 2007, pp 22-27. ACM.
5. Sen, K.; Agha, G.; "CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools", CAV'06, 2006.