Arama Tabanlı Yazılım Testi

Ömür ŞAHİN

Erciyes Üniversitesi
Bilgisayar Mühendisliği Bölümü

Sunum Planı

1-Arama Tabanlı Yazılım Testi

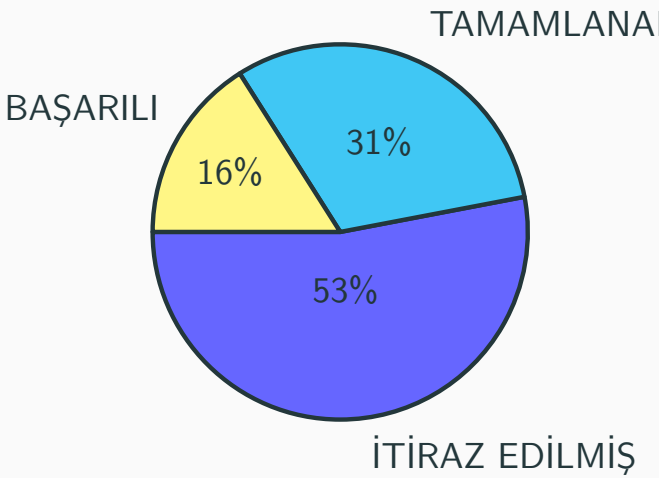
Bilgisayar Yazılımı

- İlk bilinen bilgisayar algoritması Bernoulli sayılarını hesaplamak için Ada Lovelace tarafından 19. yüzyılda yazılmıştır ancak bu çalışma teoride kalmıştır [6].
- İlk modern yazılım teorisi Alan Turing tarafından 1935 yılında yayınlanan bir makale [16] ile ortaya atılmıştır [10].
- İlk hafızada saklanan yazılım 1948 yılında Tom Kilburn tarafından geliştirilmiştir.

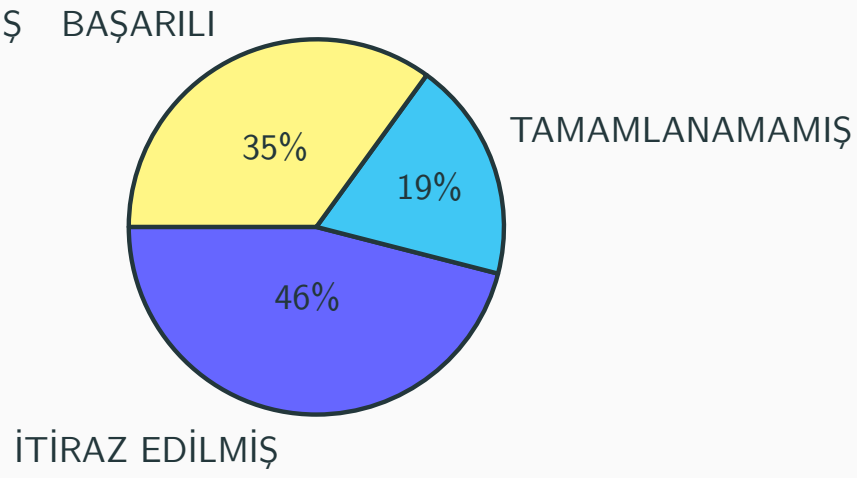
Yazılım Mühendisliği Nedir?

- İlk olarak NATO Bilim Komitesinin 1968 ve 1969 yıllarında yapmış oldukları iki konferansta tartışılmıştır [9].
- 1969 yılındaki konferansta Fritz Bauer yazılım mühendisliğini "**gerçek makineler üzerinde etkin ve güvenilir çalışan ekonomik yazılımlar geliştirilmesi amacıyla mühendislik ilkelerinin kullanılmasıdır**" diyerek tanımlamıştır.
- Yazılım mühendisliği yazılım geliştirmenin her adımıyla ilgilenen bir mühendislik disiplinidir [15].

1-Arama Tabanlı Yazılım Testi



Şekil 1: 1994 Araştırması



Şekil 2: 2006 Araştırması

- Glass'ın yapmış olduğu çalışmada [8], **16** başarısızlıkla sonuçlanan proje incelenmiş ve yalnızca **yönetimsel sorunlar** değil **teknik problemler** sebebiyle de projelerin başarısız olduğunu belirtmiştir.
- Teknik sebeplerin başında ise yazılımların ***yeteri kadar veya doğru şekilde*** test edilememesi gelmektedir.

Yazılım Testi Nedir?

IEEE Tanımı
Yazılım testi, bir yazılım çıktısının (artifact) davranışını dinamik yöntemlerle, sonsuz bir küme içerisindeki belirli test durumlarını seçerek beklenen davranışa uyup uymadığını belirleme işlemidir.

Smith, 1990
Bir sistemin veya bileşenin belirli koşullar altında çalıştırılması, sonuçların gözlenmesi veya kaydedilmesi ve belirli özelliklerin değerlendirilmesi sürecidir.

ANSI
Test, hata bulma amaçlı planlı bir şekilde gerçekleştirilen eylemler dizisi ve doğrulama yöntemidir.

Genel olarak
Giriş belirtimleri verildiğinde çıkış belirtimlerinin üretebilmesinin kontrol edilmesidir.

Yazılım Testi Maliyeti

- Yazılımların kapsamı artmıştır ve çok sayıda modül içerir hale gelmiştir.
- Her bir modül doğru şekilde test edilmezse yazılım maliyeti oldukça artmaktadır.
- Ancak, yazılım testi de oldukça fazla emek gerektirmektedir ve bu testin maliyeti toplam maliyetin neredeyse yarısına eşittir [4].
- Maliyeti düşürmenin en etkili yolu ise yazılım testlerinin otomatik olarak üretilmesidir.

Yazılım Test Üretim Teknikleri

Yazılım testlerinin otomatik olarak üretilmesine yönelik en sık kullanılan teknikler;

- Sembolik çalıştırma (Symbolic Execution)
- Model tabanlı test (Model-based testing)
- Kombinasyonel test (Combinatorial testing)
- Rastgele testin bir türevi olan adaptif rastgele test (Adaptive testing)
- Arama tabanlı test (Search-based testing)

olarak sıralanabilir.

Yazılım Test Üretim Teknikleri
Ayrıca;

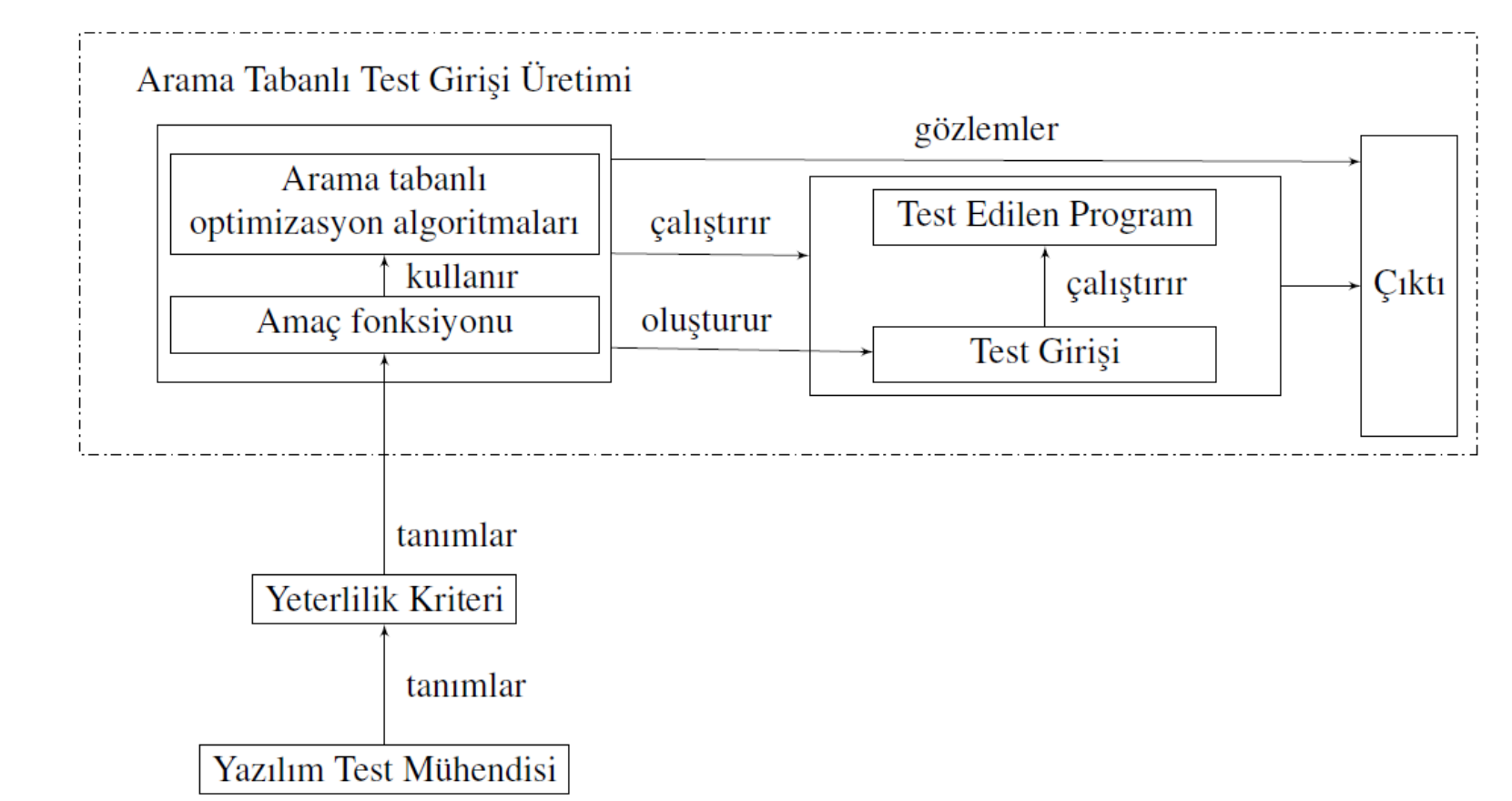
- Mutasyon testi
- Başkalaşım testi
- Belirtim tabanlı test
- Bulanıklaştırma ve veri mutasyonu testi

gibi teknikler bulunmaktadır [1].

Arama Tabanlı Yazılım Testi Nedir?

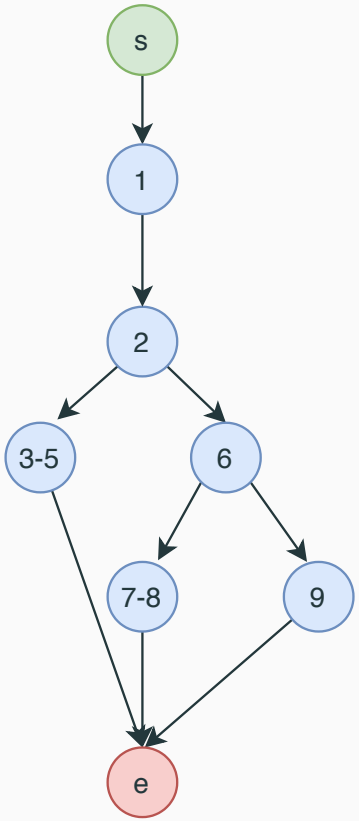
- Kolay kısıtlara sahip küçük kod parçacıkları için rastgele yazılım testi üretimi yöntemi sıkça kullanılmaktadır.
  - Yalnızca kolay problemlerde verimli
  - Zor kısıtlı problemlerde oldukça kötü sonuçlar üretmektedir
- Zorlaşan kısıtlar ve büyük kod parçaları için test üretiminde arama tabanlı test üretimi yöntemi kullanılabilir.
- Bu alanda ilk yayın Webb Miller ve David Spooner tarafından 1976 yılında çıkarılmıştır [13].
- Korel'in ortaya koyduğu çalışmalar [11, 12] ve 1992'de Xanthakis'in genetik algoritma ile bu konuda bir çözüme ulaşmış olması [17] ile arama tabanlı test verisi üretimi popüler hale gelmiştir.

Arama Tabanlı Yazılım Testi Mimarisi



Şekil 3: Genel arama tabanlı yazılım test senaryosu/verisi üretim şeması

Düğüm	Komut
s	void denklem_kokleri(double a, double b, double c)
	{
1	double result = b * b - 4.0 * a * c;
2	if (result >0.0)
	{
	double r1 = (-b + Math.pow(result, 0.5)) / (2.0 * a);
3-5	double r2 = (-b - Math.pow(result, 0.5)) / (2.0 * a);
	println("Kökler:" + r1 + " ve " + r2);
	}
	else
	{
6	if (result == 0.0)
	{
7-8	double r1 = -b / (2.0 * a);
	println("Kök: " + r1);
	}
	else
	{
9	println("Reel kök bulunmamaktadır.")
	}
	}
e	return;
	}



Şekil 4: İkinci dereceden denklem kökleri bulan kod parçası ve CFG gösterimi.

2-Nesne Yönelimli Yazılımlar için Test Senaryosu Üretimi

2-Nesne Yönelimli Yazılımlar için Test Senaryosu Üretimi

- Yazılımlar karmaşıklaştıkça ve boyutları arttıkça yapısal programlama yerine daha karmaşık eylemlerin ortaya daha rahat konabileceği nesne yönelimli programlama gibi paradigmalar oldukça popüler hale gelmiştir.
- Nesne yönelimli programlamanın;
  - Kalıtım
  - Kapsülleme
  - Çok-biçimlilik
  - Bilgi Gizleme

  gibi özellikleri sebebiyle test süreçleri de zorlaşmaktadır

2-Nesne Yönelimli Yazılımlar için Test Senaryosu Üretimi

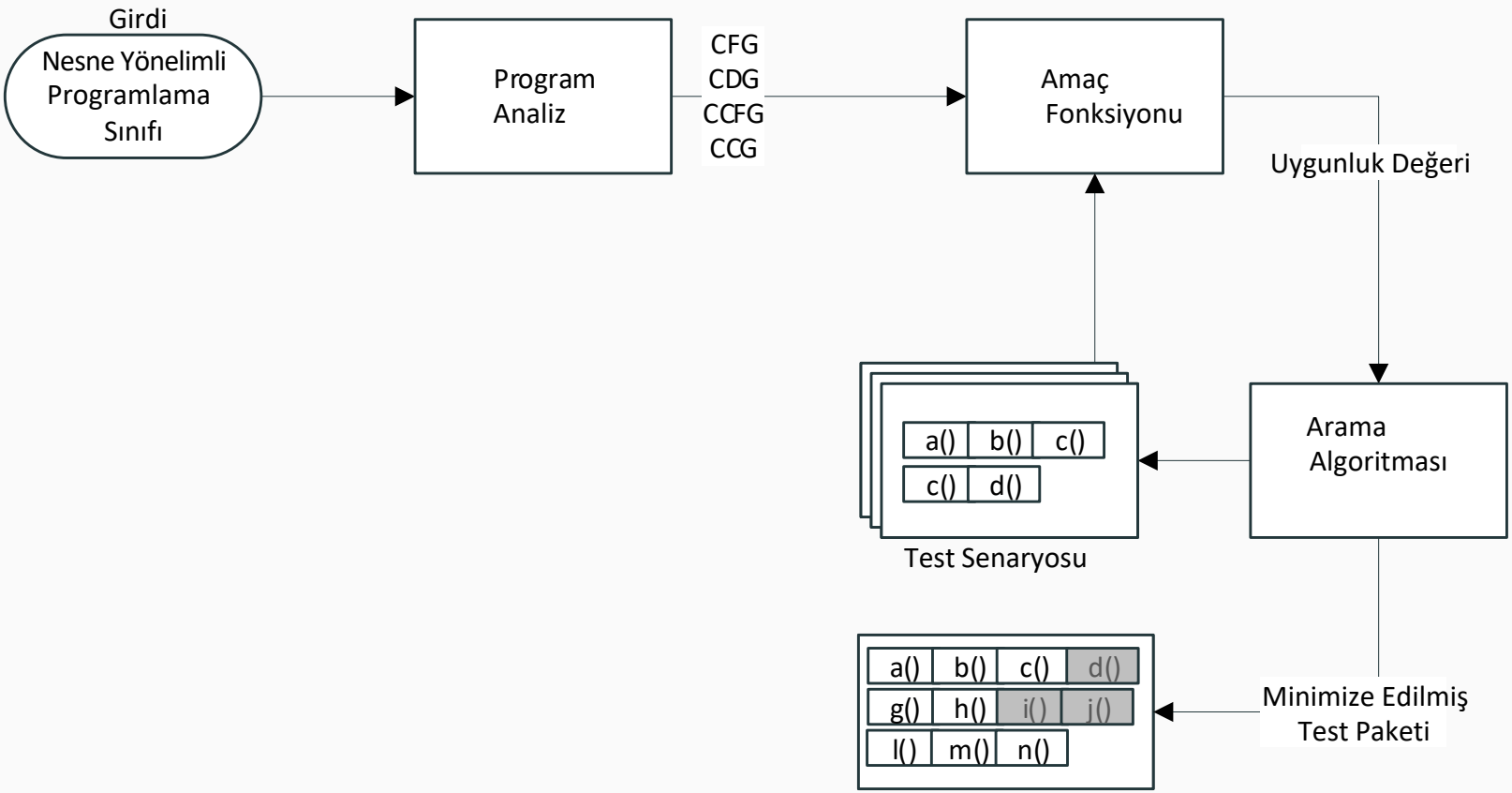
- Bir test senaryosu, komutlardan oluşan metotların birleşiminden meydana gelen birim testlerin bütünüdür.
- Fonksiyon çıktısının doğru ve geçerli olduğunu onaylamak için bir doğrulayıcı tarafından (test mühendisi gibi) belirlenen girdiler fonksiyona verilmektedir.
- Saydam kutu testinde ise bir doğrulayıcı olmaması durumunda en az test ile maksimum kapsamına erişme durumu optimizasyon problemi olarak ele alınabilmektedir.

```
public class Complex{
public Complex log() {

if(isNaN){
return NaN;
}

return createComplex(
FastMath.log(abs()),
FastMath.atan2(imaginary, real)
);
}
public Complex pow(double x){
return this.log().multiply(x)
.exp();
}
...
}
```

```
@Test
public void test1() throws Throwable{

Complex c0 = new Complex(Double.NaN);
Complex c1 = c0.pow(Double.NaN);
assertEquals(Double.NaN,
c1.getArgument(),0.01D);

}
@Test
public void test2() throws Throwable{

Complex c0 = Complex.ZERO;
Complex c1 = c0.pow(c0);
assertFalse(c1.isInfinite());
assertTrue(c1.isNaN());

}
```

Şekil 5: Örnek bir sınıf ve bu sınıfa ait test senaryosu.

Test Üretim Şeması



Şekil 6: Genel arama tabanlı test senaryosu üretim şeması.

2.1.1-Program Analiz Birimi

- Kontrol akış çizgesi (control flow graph, CFG)
- Kontrol bağımlılık çizgesi (control dependence graph, CDG)
- Sınıf çağrı çizgesi (class call graph, CCG)
- Sınıf kontrol akış çizgesine (class control flow graph, CCFG)

2.1.2-Temsil Biçimi

- Her bir çözüm ($T = \{t_1, t_2, ..., t_n\}$) olarak temsil edilmektedir.
- Bir test senaryosu (t), komutların (s) sıra ile bir araya gelmesinden oluşmaktadır ($l_j \in [1, L]$, $t_j = \langle s_1, s_2, ..., s_{l_j} \rangle$).
- Her bir komutun bir tipi bulunmaktadır.

2.1-Nesne Yönelimli Yazılımlar için Test Senaryosu Üretimi

- **Temel Komutlar (primitive statements)**: *double deger = 1.0* gibi integer, boolean, string değişkenlerini ifade etmektedir.
  Örnek:*Sinif[] n1 = new Sinif[3]*
- **Yapıcı Komutlar (constructor statements)**: Verilen sınıfın yeni bir örneğini oluşturan komutlardır.
  Örnek: *Sinif n1 = new Sinif()*
- **Alan Komutları (field statements)**: Nesnelerin public değişkenlerine erişim sağlamaktadır.
  Örnek: *int boyut = n1.boyut*

- **Metot Komutları (method statements)**: Nesnelerin non-statik veya sınıfların statik metotlarının çağrılmasını sağlayan komut tipidir.
  Örnek: *int m = n1.metot()*
- **Atama Komutları (assignment statements)**: Atama komutları nesnelerin public alanlarının veya dizilerin değer atamasında kullanılan komut türüdür.
  Örnek: *n1[0]=new Sinif()* veya *n1.deger = 3*

2.1.3-Amaç Fonksiyonu

Metot Kapsamı (Method Coverage)

Metot Kapsamı mevcut metotlardan kaçının kapsanabildiğini belirten bir metriktir.

$$f_{MC}(TestPaketi) = |ToplamMetot| - |KapsananMetot| \quad (1)$$

Satır Kapsamı (Line Coverage)

$$f_{LC}(TestPaketi) = v(|NCLs| - |KapsananSatirlar|) + \sum_{b \in B_{CD}} v(d_{\min}(b, Suite)) \quad (2)$$

Dal Kapsamı (Branch Coverage)

Dal kapsamı, test senaryosu olarak temsil edilen hedef t ve kontrol akış (x) şemalarındaki bağımsız kenarların minimum sayısı olan dal uzaklığı ile ($d(t, x)$) hesaplanmaktadır.

$$f_{BC}(Suite) = \sum_{b \in B} v(d(b, Suite)) \tag{3}$$

$$d(b, Suite) = \begin{cases} 0 & \text{Dal kapsanırsa,} \\ v(d_{min}(b, Suite)) & \text{Karşılaştırma belirtimi en az} \\ & \text{iki sefer çalışırsa,} \\ 1 & \text{aksi takdirde.} \end{cases} \tag{4}$$

Çıkış Kapsamı (Output Coverage)
Çıkış kapsamı girişe bağlı olarak elde edilebilecek çıkış değerleri ile hesaplanmaktadır.

$$f_{OC}(Suite) = \sum_{g \in G} v(d_o(g, Suite)) \tag{5}$$

G çıkış hedef setini temsil ederken, $d_o(g, Suite)$ çıktı mesafe fonksiyonudur.

Zayıf Mutasyon (Weak Mutation)

Mutasyon testinde kapsama, kodda küçük değişiklikler ve yapay hatalarla oluşturulan mutant ile orijinal kod arasındaki farklılığa göre hesaplanmaktadır.

$$f_{WM}(Suite) = \sum_{\mu \in M_C} d_w(\mu, Suite) \tag{6}$$

$$d_w(\mu, Suite) = \begin{cases} 1 & \text{mutant } \mu \text{ erişilemediyse,} \\ v(d_{min}(\mu, Suite)) & \text{mutant } \mu \text{ erişildiyse.} \end{cases} \tag{7}$$

$d_w(\mu, Suite)$ mutant ve orijinal kod arasındaki mesafeyi ölçmektedir.

İstisna Kapsamı (Exception Coverage)

İstisna kapsamı her bir sınıftaki tanımlı tekil istisnaların gerçekleşmesi ile hesaplanan bir metriktir.

$$f_{EC}(Suite) = \frac{1}{1 + N_E} \tag{8}$$

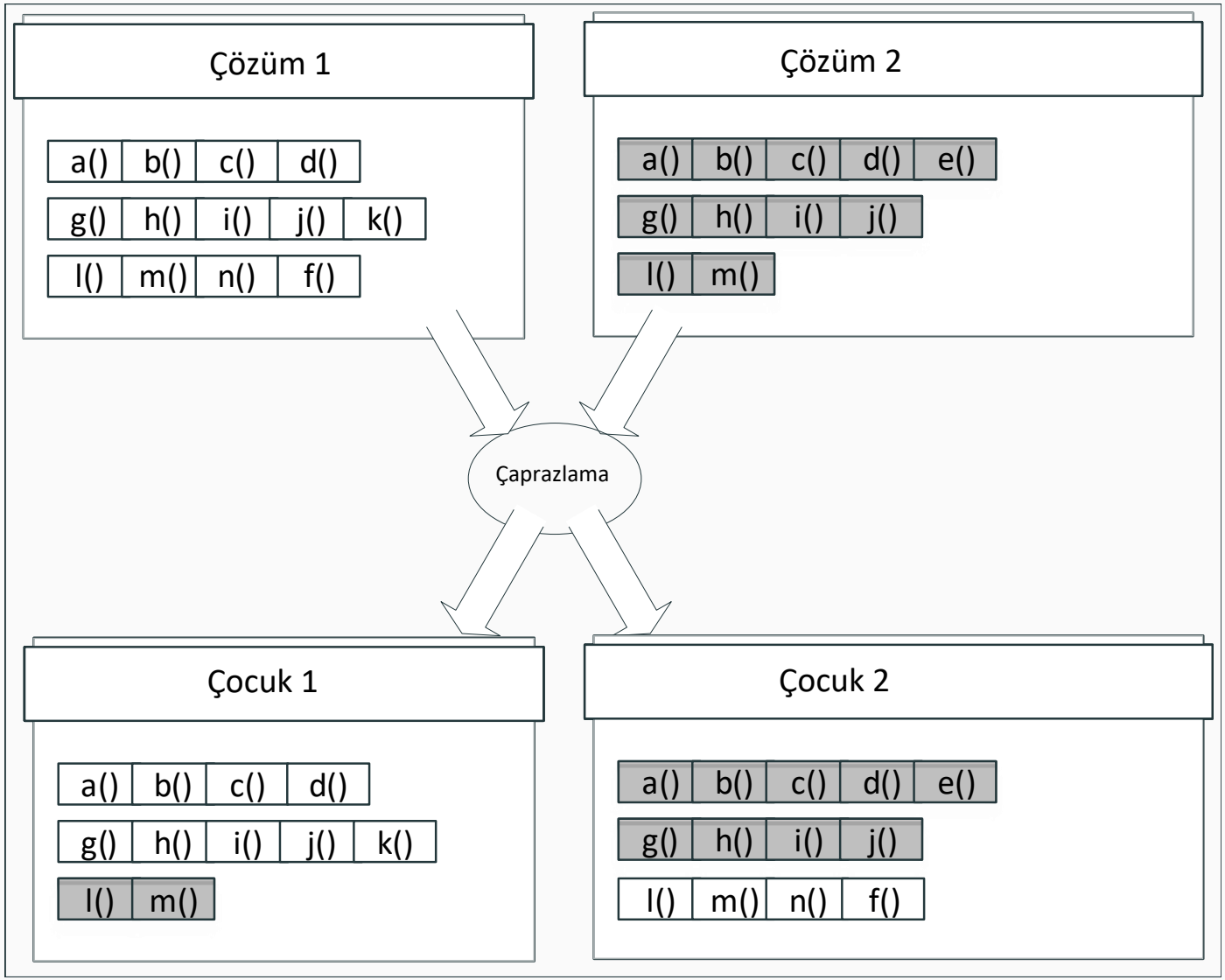
Çok Kriterli Amaç Fonksiyonu

$$f_{combined} = \sum_{i=1}^{n} w_i f_i \tag{9}$$

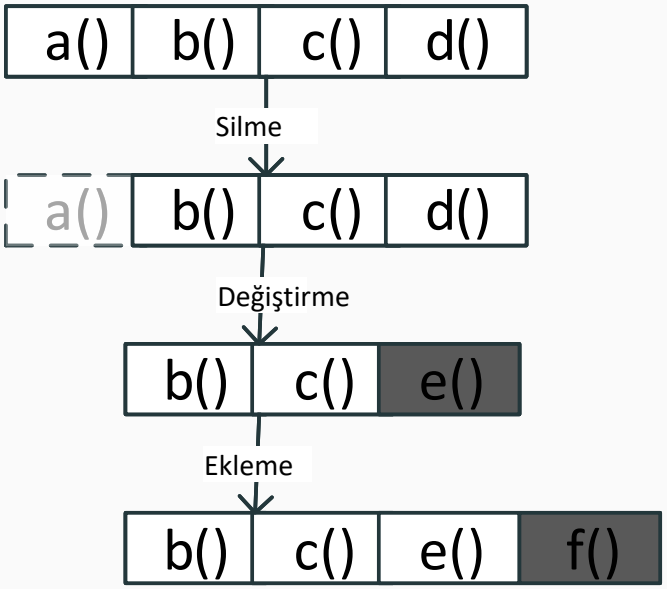
Çaprazlama

- Tek noktalı çaprazlama (Single Point, SP)
- Sabit tek noktalı çaprazlama (Single Point Fixed, SPF)
- Bağıl tek noktalı çaprazlama (Single Point Relative, SPR)

Çaprazlama


Çözüm 1
a() b() c() d()
g() h() i() j() k()
l() m() n() f()
Çözüm 2
a() b() c() d() e()
g() h() i() j()
l() m()
Çaprazlama
Çocuk 1
a() b() c() d()
g() h() i() j() k()
l() m()
Çocuk 2
a() b() c() d() e()
g() h() i() j()
l() m() n() f()

Mutasyon



Şekil 8: Mutasyon operatörünün çalışmasına örnek.

Arşiv

- Test senaryosu üretiminde bir hedefi kapsayan test senaryosu üretilse bile, diğer kapsama hedeflerine erişebilmek için arama devam etmektedir.
- Bu da sistem kaynaklarının hali hazırda kapsanmış hedefler için harcanmasına sebep olabilmektedir.
- Kapsanmış olan hedeflere ait test senaryoları arşive atılarak kapsanmamış hedeflerin keşfine odaklanılmıştır.
- Her bir iterasyonun sonunda amaç fonksiyon değerleri kapsanmış hedefleri yok sayarak güncellenir.

3-RESTful Web Servisler için Test Senaryosu Üretimi

3-RESTful Web Servisler için Test Senaryosu Üretimi

- Mikroservisler küçük, birlikte çalışan otonom servislerden oluşmaktadır.
- Mikroservis mimarisi ise belli bir bağlam doğrutusunda, gerçekleştirebileceği eylemlere göre düzenlenmiş, otomatikleşmiş ve geliştirilebilir mikroservislerden oluşan bir mühendislik yaklaşımıdır.
- Mikroservis mimarisi, bir uygulamayı yüksek düzeyde bakım yapılabilir servisler topluluğuna dönüştürür ve monolitik mimarinin karmaşıklığını azaltır.
- Ancak mikroservis mimarisinin aşağıdaki özellikleri yazılım testini zorlaştırmaktadır.
  - Otomatik Hizmet Keşfi (automated service discovery)
  - Aşırı Geç Bağlanma (ultra-late binding)
  - Kullanım başına maliyet

3-RESTful Web Servisler için Test Senaryosu Üretimi

- Pek çok mikroservis REST veya SOAP gibi yaklaşımlar ile tasarlanmaktadır.
- HTTP kullanarak çeşitli veri formatını desteklemesi sebebiyle genelde **REST**, SOAP'tan daha fazla tercih edilmektedir.
- REST yapısı [7] ilk olarak 2000 yılında Roy Fielding tarafından tanımlanmıştır.
- REST, HTTP gibi tercih edilen bir protokole dayalı bir dizi mimari kılavuz görevi görmektedir.

3-RESTful Web Servisler için Test Senaryosu Üretimi

- RESTful web servislerin otomatik olarak test edilmesi yazılım bakım maliyetlerini düşürmekte ve manuel test etme sonucunda ortaya çıkan insan faktörüne bağlı hata yapma ihtimalini azaltmaktadır.
- RESTful web servisler genelde HTTP protokolüne dayalı çalışmaktadır.
- Bir HTTP mesajı, kaynak yolu, metot/fiil, başlık ve gövdeden oluşmaktadır. Bir HTTP talebi gönderildiği zaman HTTP cevabı sunucu tarafından geri gönderilmektedir.
- HTTP cevabı bir başlık, gövde ve üç haneli durum kodundan meydana gelmektedir.

3.1-RESTful Web Servisler için Test Senaryosu Üretimi

- Temsil Biçimi
- Amaç Fonksiyonu
- Yeni Çözüm Üretimi
- Arama Operatörü

3.1.1-Temsil Biçimi

- RESTful web servislerin testinde son kullanıcıya sunulan son çözüm bir veya daha fazla test senaryosundan oluşan bir test paketidir.
- Test senaryosu üretilirken temel amaç, her bir hedef için ayrıca tutulan test senaryosunun iyileştirilmesidir.
- Her bir test senaryosu bir veya daha fazla HTTP talebinden meydana gelmektedir.
- Test senaryosu üretilirken;
  - HTTP fiili → POST, GET, PATCH, DELETE vs.
  - Başlık
  - Yol parametreleri →/user/1
  - Sorgu parametreleri → /get.php?user=1
  - Gövde yükü

rastgele olarak oluşturulmaktadır.

3.1-RESTful Web Servisler için Test Senaryosu Üretimi

- Bir HTTP talebi oluşturmadan önce ilgili API hakkında bilgi sahibi olunması gerekmektedir.
- RESTful web servisler için oluşturulmuş olan Swagger[1] isimli araç bu bilgiyi sağlayabilmektedir.
- Swagger aracı API hakkındaki bilgiyi JSON formatında kullanıcıya sunmaktadır.

[1] https://swagger.io

3.1-RESTful Web Servisler için Test Senaryosu Üretimi

```
"/products/{productName}/configurations/{configuration
/features/{featureName}" :
{"post" : {
        "parameters" : [
        {"name" : "productName",
            "in" : "path",
            "required" : true,
            "type" : "string"
        },
        {"name" : "configurationName",
            "in" : "path",
            "required" : true,
            "type" : "string"
        },
        {"name" : "featureName",
            "in" : "path",
            "required" : true,
            "type" : "string"
        }]
}}
```

3.1-RESTful Web Servisler için Test Senaryosu Üretimi

- Swagger tanımını indirip ayrıştırdıktan sonra, her bir API uç noktası için taslak kromozomlar oluşturulmaktadır.
  - API'ın IP adresi, gerekli HTTP başlıkları gibi sabit genler
  - Değiştirilebilir genler
- Değiştirilebilir genler basit genlerin birleşiminden oluşan karmaşık veri tiplerinden meydana gelmektedir.

Değiştirilebilir Genler

- Array
- Base64String
- CycleObject
- Date
- Time
- DateTime
- Disruptive
- Double
- Enum
- Float
- Integer
- Long
- Map
- Optional
- String

3.1.2-Amaç Fonksiyonu

- Her bir test senaryosu bir veya birden fazla hedefi kapsayabilmektedir.
- Her bir t hedefi için ayrı tutulan popülasyonunun bireylerinin kalitesi yalnızca t hedefi için toplanan sezgisel değerlere göre belirlenmektedir.
- Her bir hedefe ait $[0, 1]$ aralığında sezgisel amaç fonksiyonu değeri bulunmaktadır.
- Test senaryolarının;
  - Test edilen sistemin komutlarının kapsamı
  - Bytecode seviyesinde dalların kapsamı
  - HTTP durum kodları

  olmak üzere üç farklı kapsanması gereken hedefi bulunmaktadır.

Komut Kapsamı

- Yüksek kapsamlı test senaryoları üretebilmek için ilgili test senaryosunun elde ettiği kapsama miktarının ölçülebilmesi gerekmektedir.
- Aksi taktirde hangi test senaryosunun daha yüksek kapsama sahip olduğu bilinemez.
- Dal kapsamı gibi saydam kutu metriklerine erişebilmek için kaynak koda erişim gerekmektedir.
- Dolayısıyla bir SUT başlatıldığında prob eklenmekte ve bu kapsama metrikleri elde edebilmektedir.

Bytecode Seviyesinde Dal Kapsamı

- Bir testin ne kadar kapsama ulaşabildiğini elde etmek tek başına yeterli değildir.
- Genellikle bir kodun kapsanamama sebebi if blokları içerisinde bulunmasıdır.
- Rastgele girdilerin bu karmaşık if bloklarını çözebilmesinin olasılığı oldukça düşüktür.
- Bu problemin en etkin çözüm yöntemlerinden biri, test verisinin çözüme ne kadar yaklaştığını gösteren sezgisel değerin atanmasıdır.

```
1
2                if(x==500){
3                    ...
4                } else if (x > 1000) {
5                    ...
6                } else {
7                    ...
8                }
```

Örnek

Üç farklı dal (if, else if ve else) bulunmaktadır. Bu dallardan if bloğuna ait $x == 500$ kısıtı düşünüldüğünde 600 sayısı 5 sayısından daha yakın bir çözümdür.

- Bir istisna meydana gelmesi durumunda satır kapsamı için pek de iyi bir seçenek olmamaktadır.
- Bu durumun amaç fonksiyonunda ele alınabilmesi için bir sezgisel değer bulunmaktadır ve ilk başta $h = 0.5$ olarak atanmaktadır.
- Eğer bir test senaryosu istisna ortaya çıkarmazsa $h = 1$ olmaktadır.
- Eğer ilgili komuta hiç erişilemediyse $h = 0$ olmaktadır.

```
1  if(topla){
       //h=1
2      res = x+y
3  }else if
       (carp){
       //h=1
4      res = x*y//
           h=0.5
5  }
6  else if (bol){
7      res = x/y
8  }
9  return res
       //h=0
```

A Testi

```
1  if(topla){
       //h=1
2      res = x+y
3  }else if
       (carp){
       //h=1
4      res = x*y
5  }
6  else if (bol){
       //h=1
7      res =
           x/y//h=0.5
8  }
9  return res
       //h=0
```

B Testi

HTTP Durum Kodları

- Döndürülen HTTP durum kodlarının amaç fonksiyon değerinde ara değerler bulunmamaktadır.
- Bu durum kodları ya kapsanır ($h = 1$) veya kapsanmaz ($h = 0$) olarak atanmaktadır.

Amaç Fonksiyonun Sunulması

Aşağıdaki gibi bir sorgu gerçekleştiğinde;

```
POST /products/s07atT07/constraints/excludes
```

```
"data": {
    "targets": [
    {
        "id": 0,
        "descriptiveId":
            "Line_at_org.javiermf.features.CORSFilter_
        "value": 1,
        "actionIndex": 0
    }
    {
        "id": 1,
        "descriptiveId":
            "Line_at_org.javiermf.features.CORSFilter_
        "value": 1,
        "actionIndex": 0
    }
```

3.1.3-Yeni Çözüm Üretimi

- Bir test senaryosu rastgele örneklendiğinde içerisinde 1 ile n aralığında değişen HTTP çağrısı bulunmaktadır.
- Rastgele dizelerin geçerli bir HTTP çağrısı oluşturması oldukça düşük bir olasılık olduğundan HTTP çağrılarının örneklenmesi için şablonlar kullanılmaktadır.

```
POST /products
POST /products
GET /products/100
```

Şekil 9: HTTP çağrı örneği

3.1-RESTful Web Servisler için Test Senaryosu Üretimi

- Yeni test senaryosu rastgele örneklenirken belirli bir P olasılığı ile bu testin yapısının önceden tanımlandığı bir şablon kullanılmaktadır.
- Bu şablonlar;
  - GET
  - POST
  - PUT
  - PATCH
  - DELETE
  - Ara Kaynaklar
  - Zincirleme Konumlar

  şablonlarından oluşmaktadır.

GET Şablonu - I

```
GET /products
```

- Bu uç nokta bir veya birden fazla kaynağı temsil edebilmektedir.
- Bir GET isteğini düzgün test edebilmek için birkaç adet girdiye ihtiyaç duyulmaktadır.
- Bu nedenle rastgele k adet POST çağrısı eklenir ve bu şablonda, aşağıda gösterildiği gibi k sayıda (örnekte 3 adet) POST ve en sonda da bir adet GET bulunur.

```
POST /products
POST /products
POST /products
GET /products
```

GET Şablonu - II

Bir diğer durum da aşağıdaki istek gibidir.

```
GET /products/{id}
```

Bu durumda önce kaynak oluşturulur daha sonra bu kaynak GET ile çekilmektedir.

```
POST /products
GET /products/{id}
```

POST ile oluşturulan kaynağın GET ile nasıl ilişkilendirildiği ise Zincirleme Konumlar kısmında açıklanmaktadır.

POST Şablonu

- POST, kaynak oluşturmak için kullanıldığı için özel olarak ele almaya gerek yoktur. Yalnızca tek bir POST çağrısı yeterlidir.

PUT Şablonu

- PUT bir kaynağı tamamen değiştiren idempotent (tek sefer etki gösteren) bir fiildir.
- Bir kaynak yoksa PUT metodu ile oluşturulabilir veya 4xx hatası dönebilir.
- Tek bir PUT ile bir kaynak oluşturulması veya POST ile önce bir kaynak oluşturma ardından PUT ile bu kaynak üzerinde değişiklik yapmak olmak üzere iki farklı durum ele alınmaktadır.

DELETE Şablonu

- Bir DELETE fiilinin doğru test edilebilmesi için kaynağın var olması gerekmektedir. Bu nedenle önce POST/PUT ile kaynak oluşturulur ve DELETE ile silinmektedir.

PATCH Şablonu

- PATCH bir kaynak üzerinde kısmi güncelleme yapmaktadır. Bu nedenle test etmek için önce bu kaynağı yaratmak gerekmektedir.
- PATCH, PUT yönteminden farklı çalışmaktadır. Eğer kaynak eksikse PUT bu kaynağı oluşturabilmekteyken PATCH oluşturamamaktadır. Bu nedenle önce POST/PUT ve ardından PATCH çağırılması gerekmektedir.

```
POST /products
PATCH /products/{id}
PATCH /products/{id}
```

Ara Kaynaklar Şablonu

- Bir kaynak bir başka kaynağa bağlı olduğunda kullanılmaktadır (*GET /products/id/price* gibi).

GET /products/{pid}/subitems/{sid} isteği için:

```
POST /products
POST /products/{pid}/subitems
GET /products/{pid}/subitems/{sid}
```

test senaryosu oluşturulmalıdır.

Zincirleme Konumlar Şablonu

- Ara kaynaklar oluşturulduğunda, değişken değerlerin aynı olmasının sağlanması gerekmektedir.
- id'ler el ile oluşturulduğunda mutasyona uğramasına izin verilmez.
- Bu kaynaklar POST ile oluşturulduğunda
  - Location Header bilgisi olarak elde edilebilir. **Örnek:** *POST /products/100* oluşturulduğunda *location: /products/100* belirtilir.
  - Bazı uç noktalarda bu bilgiler HTTP yanıtının yükü olarak gönderilmektedir. Eğer bu yük JSON veya XML gibi formatlar ile yapılandırıldıysa, uç noktadaki yol parametrelerinden biri ile ilişkili (id gibi) olup olmadığı kontrol edilmekte ve sezgisel olarak POST cevabından çıkarılmaya çalışılmaktadır.

```
1  GET /products/GEWJ
2  POST /products/Liv/features/Z
3  GET /products
4  DELETE /products/ED/constraints/4
5  GET /products/M/features
```

Şekil 10: Rastgele örnekleme ile oluşturulan test senaryosu.

```
1  POST /products
2  POST /products/WLiv4O2z/features
3  GET /products/WLiv4O2z/features/U
```

Şekil 11: Akıllı örnekleme ile oluşturulan test senaryosu.

3.1.4-Arama Operatörü

- Yeni birey üretimi ve var olan çözümün değiştirilmesi üzere iki arama operatörü bulunmaktadır.
- Var olan bir testin değiştirilmesi için mutasyon operatörü kullanılmaktadır.
  - Boole değeri tersine çevrilir.
  - Tam sayılar $\pm 2^i$ delta ile değiştirilir. Buradaki "i" rastgele olarak 0 ile arama sırasında azalan bir maksimum değer arasından seçilmektedir.
  - Float/double değerler için aynı tür $\pm 2^i$ delta uygulanır ancak gauss değeri ile çarpılmaktadır.
  - Dizelerde, bir karakter rastgele değiştirilebilmekte, son karakter bırakılabilmekte veya dizinin sonuna yeni oluşturulan bir karakter eklenebilmektedir.

```
1  POST
       /products/9Xs3n1XecXX1C2/constraints/excludes
2  GET /products/YJ
3  POST /products/WLiv4O2z/features/U
4  GET /products
5  DELETE /products/2D/constraints/28069080
6  GET /products/SNlpoIbZ2Li
7  GET /products/X/features
```

Şekil 12: Yedi HTTP isteğinden oluşan örnek test senaryosu.

```
1  POST
       /products/9Xs3n1XecXX1C2/constraints/excludes
2  GET /products/YJ
3  POST /products/WLiv4O2z/features/U
4  GET /products
5  DELETE /products/2D/constraints/28069080
6  GET /products/SNlpoIbZB%5Ci
7  GET /products/X/features
```

4-Referanslar

4-Referanslar

[1] Saswat Anand et al. “An Orchestrated Survey on Automated Software Test Case Generation”. In: ***Journal of Systems and Software*** 86.8 (2013), pp. 1978–2001. ISSN: 0164-1212. DOI: 10.1016/j.jss.2013.02.061. URL: http://cs.stanford.edu/people/saswat/research/ASTJSS.pdf{\%}5Cnhttp://dl.acm.org/citation.cfm?id=2503991.

[2] A. Arcuri. “EvoMaster: Evolutionary Multi-context Automated System Test Generation”. In: ***2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)***. 2018, pp. 394–397. DOI: 10.1109/ICST.2018.00046.

4-Referanslar

[3] A. Arcuri. “RESTful API Automated Test Case Generation”. In: ***2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)***. 2017, pp. 9–20. DOI: 10.1109/QRS.2017.11.

[4] Boris Beizer. ***Software Testing Techniques (2nd Ed.)*** New York, NY, USA: Van Nostrand Reinhold Co., 1990. ISBN: 0-442-20672-0.

[5] Kalyanmoy Deb et al. “A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimization: NSGA-II”. In: ***Parallel Problem Solving from Nature PPSN VI***. Springer Berlin Heidelberg, 2000, pp. 849–858. DOI: 10.1007/3-540-45356-3_83. URL: https://doi.org/10.1007/3-540-45356-3_83.

4-Referanslar

[6] Claire L Evans. ***Broad band: the untold story of the women who made the Internet***. Portfolio, 2020.

[7] Roy T Fielding and Richard N Taylor. "Architectural styles and the design of network-based software architectures". PhD thesis. University of California, 2000.

[8] Robert L. Glass. ***Software Runaways: Monumental Software Disasters***. USA: Prentice-Hall, Inc., 1998. ISBN: 013673443X.

[9] Ali Gürbüz. ***Yazılım Test Mühendisliği***. Papatya Yayıncılık, 2010. ISBN: 978-605-4220-09-0.

[10] Mike Hally. ***Electronic brains: stories from the dawn of the computer age***. Granta books, 2005.

4-Referanslar

[11] B Korel. “Automated Software Test Data Generation”. In: ***IEEE Transactions on Software Engineering*** 16.8 (1990), pp. 870–879. DOI: 10.1109/32.57624.

[12] B. Korel, H. Wedde, and R. Ferguson. “Dynamic method of test data generation for distributed software”. In: ***Information and Software Technology*** 34.8 (1992), pp. 523–531. ISSN: 09505849. DOI: 10.1016/0950-5849(92)90146-G.

[13] W. Miller and D. L. Spooner. “Automatic Generation of Floating-Point Test Data”. In: ***IEEE Transactions on Software Engineering*** SE-2.3 (1976), pp. 223–226. DOI: 10.1109/TSE.1976.233818.

4-Referanslar

[14] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. “A large scale empirical comparison of state-of-the-art search-based test case generators”. In: ***Information and Software Technology*** 104 (2018), pp. 236–256. ISSN: 09505849.

[15] Ian Sommerville. ***Software Engineering (9th Edition)***. Pearson, 2010. ISBN: 0137035152.

[16] Alan Mathison Turing. “On computable numbers, with an application to the Entscheidungsproblem”. In: ***Proceedings of the London mathematical society*** 2.1 (1937), pp. 230–265.

4-Referanslar

[17] S Xanthakis et al. ***Application of Genetic Algorithms to Software Testing***. Generic. 1992.