View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Sabanci University Research Database

Geribesleme Güdümlü Adaptif Kombinasyonel Test Etme Yaklaşımı

Emine Dumlu ve Cemal Yılmaz
Bilgisayar Mühendisliği Programı
Sabancı Üniversitesi, İstanbul

Mayra B. Cohen
Dept. of Comp. Sci. & Eng.
University of Nebraska-Lincoln
Lincoln, NE USA

Adam Porter
Bilgisayar Mühendisliği Bölümü
Maryland Üniversitesi
College Park, MD USA

Abstract

The configuration space of a software system forms a combinatorial space, whose large size generally makes exhaustive testing infeasible. Combinatorial interaction testing (CIT) approaches systematically sample the configuration space and test only the selected configurations. The basic justification for CIT approaches, such as covering arrays, is that they can cost-effectively exercise all system behaviors caused by the settings of *t* or fewer options. We conjecture however that in practice many such behaviors are not actually tested in the presence of what we call masking effects failures perturbing behavior in ways that prevent some intended behaviors from being tested. In this work we present a feedback driven adaptive combinatorial testing approach aimed at detecting and working around masking effects. At each iteration of this approach, we detect potential masking effects, heuristically isolate their likely causes, and then generate new covering arrays that allow previously masked combinations to be tested in the subsequent iteration. We, furthermore, empirically assess the effectiveness of the proposed approach by using a large widely used open source software system as our subject application.

1. Giriş

Çalışma zamanı veya derleme zamanı tercihlerinin değişikliği vasıtasıyla, yazılım özelleştirme, kullanıcılara yazılımlarının davranışlarını kontrol etme imkanı tanımaktadır. Web sunucuları (Örneğin Apache), veritabanları (Örneğin Mysql) ve uygulama sunucuları (Tomcat), düzinelerce hatta yüzlerce uyarlanabilir seçeneklere sahip olup, çok sayıda yapılandırma alternatifi sunarlar.

Tüm yapılandırma uzayının çekici olmasına karşın, sistemin doğruluğu onaylanırken, bütün yapılandırmaların ayrıntılı testlerinin yapılması genel olarak imkansızdır. Tümleşik Etkileme Testi (CIT) olarak adlandırılan bir çözüm yaklaşımı, sistematik olarak yapılandırma uzayını denemekte ve sadece dikkatlice seçilmiş yapılandırmaları [2,7,9,11,18] test etmektedir.

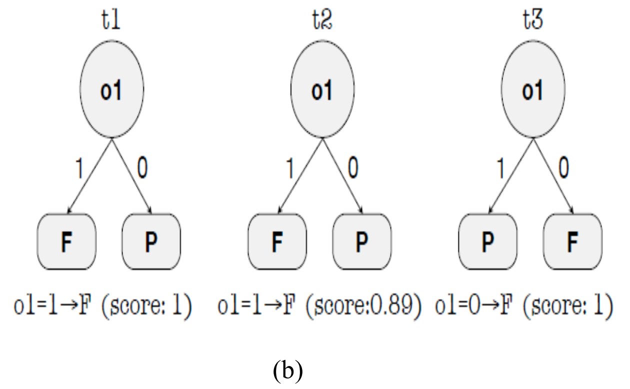
CIT yaklaşımı genel olarak sistemin yapılandırma uzayını tanımlayan bir model üzerinden çalışmaktadır. Tipik olarak, bu model, her biri küçük sayıda seçenek ayarlarını üstlenen yapılandırma seçeneklerinin bir setini içermektedir. Bu model verildiği için, CIT yöntemleri gelecek küçük bir set somut yapılandırmaları, *t* seçeneğinin her kombinasyonu için seçenek ayarlarının her olası kombinasyonu en azından bir kere göründüğü t-yollu bir örtme dizisini hesaplamaktadır [7]. Son olarak, örtme dizisindeki her yapılandırmada test takımı çalıştırılarak sistem test edilmektedir.

Örtme dizisi yaklaşımı genel olarak, yapılandırma seçenekleri, diğer seçenek ayar kombinasyonları tarafından etkili olarak iptal edilen seçenek ayar kombinasyonları arasında bilinmeyen kontrol bağımlılıklarının olmadığını varsaymaktadır. Bilinen kontrol bağımlılıkları kısıtları belirterek veya örtme dizisine ek olarak bir takım ihmal test durumları tanımlanarak çalışılır [7]. Verilen bu varsayımlar, örtme dizileri için temel gerekçe, onların t veya daha az seçeneklerden kaynaklanan tüm sistem davranışlarını maliyeti-etkili olarak yerine getirebilmektir.

Buna rağmen, biz uygulamada bu tür birçok davranışların aslında test edilmediğini hipotez olarak kurmaktayız. Özellikle, test edilecek olan bazı davranışları engellediği ve test süreci boyunca sorumlu olmadıkları birçok yolda, test başarısızlıklarının davranışı alt üst edebileceğine inanmaktayız. Biz bunu "maskeleme etkisi" olarak adlandırıyoruz. Yani, başarısızlıklar, test edilecek olan bu kombinasyonlardan kaynaklanan davranışları engelleyen seçenek ayar kombinasyonlarını etkili olarak maskeleyebilmektedir. Bir maskeleme etkisinin basit bir örneği, programın yürütmesindeki bir program erken olarak kırılan bir hata olacaktır. Sonra kırılma, programın yürütmesinde normal olarak sonradan oluşacak olan bazı davranışlara bağlı yapılandırmayı engellemektedir.

o1	o2	o3	o4	t1	t2	t3
1	1	1	1	F	F	P
1	1	0	0	F	F	P
1	0	1	0	F	F	P
1	0	0	1	F	F	P
0	1	1	0	P	F	F
0	1	0	1	P	P	F
0	0	1	1	P	P	F
0	0	0	0	P	P	F

(a)



(b)

Tablo 1, varsayımlı bir örtme dizisi odaklı test senaryosunda maskeleme etkilerini göstermektedir. Bu senaryoda, 4 yapılandırma seçeneği (o1,o2,o3,o4) ile bir yapılandırma model için yaratılan 3-yollu örtme dizisine sahibiz. Her seçenek 0 veya 1 olmak üzere boole değeri almakta ve iç seçenek kısıtı yoktur. Yapılandırmalar 3 tane test (t1,t2,t3) kullanılarak test edilir. P ve F literalleri, nispeten bir testin başarılı veya başarısız olduğunu göstermektedir. Örneğin test durumlarından t1'i göz önünde bulunduralım: Bu test durumu, o1==1 olduğu her durumda başarısız olmaktadır. Sonuç olarak, o1==1 olduğu 4 tane hatalı durumdaki diğer seçeneklerin (o2,o3,o4) gerçekten test edilmemesi olasıdır. Gerçekten, bu 4 kombinasyon örtme dizisindeki başka hiçbir yerde görünmediği için onların asla test edilmediği mümkündür. Bu durumda, bir çözüm olarak eksilen yapılandırmaların her birinde o1==0 i kurmak ve test durumunu tekrardan çalıştırmak olabilir. Başarısızlığın bir seçenek ayar kombinasyonunun daha fazlasından kaynaklandığı ve çoklu başarısızlıkların olduğu daha karmaşık örnekler için, daha karmaşık bir cevap gerekli olabilir.

Bu çalışmada, maskeleme etkilerinin zararlı sonuçlarını önlemek için geribesleme güdümlü adaptif kombinasyonel test etme yaklaşımını sunmaktayız. Her bir iterasyonda, potansiyel maskeleme etkilerini yakalamakta, onların olası sonuçlarını izole etmekte ve sonrasında izleyen iterasyondaki test etme için maskelenecek olan t-yollu seçenek kombinasyonlarının setini ilave etmekteyiz. Süreç, tüm testler ve her t-yollu seçenek ayar kombinasyonları, en azından testin geçtiği veya olmayan seçenekle ilgili bir sebeple başarısız olan bir yapılandırmada mevcut olana veya kombinasyon başarısız sonuçla işaretlenilmeye kadar süreç tekrarlamaktadır. İki büyük ve popüler kaynak donanım sistemlerini bağlayan deneysel değerlendirmemiz, önerilen yaklaşımın önlenen maskeleme etkilerindeki diğer yaklaşımlardan daha iyi olduğunu önermektedir.

Bu makalenin geri kalan kısmı şu şekilde organize edilmiştir: Bölüm 2, kısaca temel bilgileri vermekte ve ilgili çalışmaları anlatmaktadır. Bölüm 3 hedeflenen yaklaşımı tanımlar.

2. ARKA PLAN VE İLGİLİ ÇALIŞMA

Yazılım testi için örnekleme tabanlı örtme dizisi, programlara giriş parametrelerinin [2,5,7,9] kombinasyonunun tam kaplamını garanti altına almanın bir yol olarak hedeflendiği teknik tabanlı bir belirlemedir. Son zamanlardaki daha fazla çalışma örtme dizileri, örtme dizisinin bir "test tarife" olarak tanımladığı ve her yapılandırma test durumlarının tam bir takımıyla test edildiği test etme için seçilmesi gereken [10, 15, 18] yapılandırmaları modellemek için kullanılmıştır. Örtme dizilerinin test etmede kullanıldığı diğer bazı domainler, grafiksel kullanıcı ara yüzeylerini ve test etme tabanlı modelde test etmektedir [4].

Bir örtme dizisi [7], N'nin test edilecek olan birçok yapılandırmayı tanımladığı ve k'nın manipüle edilebilecek olan birçok yapılandırma seçenekleri olduğu N×k boyutunun bir sırasıdır. Yapılandırma seçeneklerinin her biri bazı ayar numaralara (farklı yapılandırma seçenekleri ayar sayıları farklılık gösterdiğinde sıklıkla v veya v_1 olarak gösterilen) sahip olacaktır. Her yapılandırma (veya sıranın dizisi), donanımın test altında olduğu, tüm yapılandırma seçeneği için bir olan geçerli ayarların bir setidir.

N yapılandırmalarının setiyle beraber, en yaygın dayanıklılık gibi t=2'yle beraber test dayanıklılığı genellikle k'dan daha küçük olduğu t'de seçenek ayarlarının her t-yollu kombinasyonu en azından bir kere bulunulacaktır. Deneysel araştırma, t< 6'nın etkileşim hatalarının büyük bir kısmını potansiyel olarak bulabildiğini göstermiştir [11]. t ayarlarının spesifik bir seti aynı yapılandırmada kullanıldığı zaman bir etkileşim hatası sadece manifeste edilebilir. Deneysel sonuçların dahası örtme dizilerinin uygulamada etkili olduğunu göstermektedir [10, 15, 16, 18].

```
# test-specific constraint
o1!=1

# seed
o1 o2 o3 o4
 0  1  1  0
 0  1  0  1
 0  0  1  1
 0  0  0  0
```

(a)

```
# test-specific constraint
o1!=0

# seed
o1 o2 o3 o4
 1  1  1  1
 1  1  0  0
 1  0  1  0
 1  0  0  1
```

(b)

o1	o2	o3	o4	tests
0	1	1	1	$\{t1, t2\}$
0	1	0	0	$\{t1, t2\}$
0	0	1	0	$\{t1, t2\}$
0	0	0	1	$\{t1, t2\}$
1	1	1	0	$\{t3\}$
1	1	0	1	$\{t3\}$
1	0	1	1	$\{t3\}$
1	0	0	0	$\{t3\}$

(c)

Şekil 2: (a) t1 ve t2 için yapılandırma modeli (aynı modeli paylaşmak için oluşmakta olan). (b) t3 için yapılandırma modeli (c) hesaplanan örtme dizisi

Bir örtme dizisinin çekirdeği, seçenek ayarları arasındaki herhangi bağımlılıklarının yanı sıra k, v_is'nin her biri ve t'yi içeren örtme dizisini tanımlayan modeldir. Modellenen yapılandırma seçenekleri hem çalışma zamanı hem de derleme zamanında kontrol edilebilir [18]. Modelde her ikisi de önemli olurken, başarısızlıklar farklı şekillerde test etme sonuçlarını etkileyebilmektedir. Örneğin, bir derleme zamanı seçeneği yapmak için başarısız olursa, tek bir testi çalıştıramayız, oysaki bir çalışma zamanı seçeneği bir test üzerinde başarısız olursa diğerinde başarılı olabilmektedir.

Örtme dizisindeki araştırma için son bir odak nokta, gömülü-seçenek sınırlamaları, spesifik ayarlar arasındaki bağımlılıklarının etkisinin çalışmasına sahip olmasıdır [3, 6, 8]. Örneğin bir yapılandırma seçeneği için ayarın yapısı, başlangıcın bir ayarını almak için diğer bir yapılandırma seçeneğini zorlayabilir. Bu değişiklik kaç tane t-yolunun geçerli kombinasyonu olması ve test etme için gerekli olacak bir takım yapılandırmaları etki edebilmesidir. Sınırlamalar deneylerimizde gördüğümüz gibi maskeleme etkilerinde de sorumlu olabilmektedir. Bu çalışmada kaçınıldığı gibi maskeleme kombinasyonlarını kodlamak için sınırlamaların kavramını kullanmaktayız.

Bu makaledeki çalışma, örtme dizisi test programlarının başarısız/geçersiz sonuçlarına dayanan hataları (gelecek bölümde görülecektir) karakterize etmek için sınıflandırma ağaçlarını kullanmaktadır. Bu çalışma bizim geçmiş çalışmamızın [18] bazılarından birini çizgi boyunca takip etmektedir. Buna ek olarak, yapı örtme dizileri için artımlı bir yöntem, daha düşük güçlü olanlardan daha fazla güçlü örtme dizilerini inşa eden test programları arasındaki sonuçları yeniden kullanmak için bir yol sağlamayı geliştirmişti ve test etme zamanı ve kaynaklarının daha az önsel bir bilgi olduğu zaman kullanılabilmektedir. Bu makaledeki yararlı sınıflandırmamız farklıdır. Maskeleme etkilerini engelleyebilmek için onları tanımlamada kullanmaktayız, örneğin; bu bir son sonuç değil fakat sürecin bir parçasıdır.

Test etmeye odaklı belirlemede, kategori bölüm yöntemi veya Test Belirleme Dili, TSL [14] gibi teknikler boyunca bir sistemin modelleme bölümleri üzerinde kayda değer çalışma mevcuttur. Kategori bölümünde, ilk olarak parametreler ve onların seçimleri tanımlanmaktadır. Buna ulaşır ulaşmaz sınırlamanın birçok türleri eklenilmektedir. Bazı seçimler, bağımlı sınırlamalar haline gelenlere bağlı olabilirken, diğer seçimler yalnız test edilebilir ve hata veya tek olarak işaretlenir ve diğer seçenek ayarlarıyla asla birleştirilmez. Bu durum bizim maskeleme problemimizin bir bölümüyle benzerdir, yine de sınırlamalar el ile keşfedilmektedir.

Test etme yapılandırmaları [15, 18] için örtme dizileri üzerinde kullanılan önceki çalışmada, test altındaki sistem bir yapılandırma modeline sahiptir ve her yapılandırma test durumlarının aynı setinde çalışmaktadır. Bu makaledeki çalışma, onun her test durumu için bir yapılandırma modeli yaratmasından farklı olmaktadır ve her yapılandırmada çalışacak olan test durumlarının farklı potansiyel setlerini programlamaktadır.

3. GERİBESLEME GÜDÜMLÜ ADAPTİF KOMBİNASYONEL TEST ETME

Bu makalede, her test durumunun tüm gerekli seçenek birleşimlerini test etmede uygun bir şansa sahip olmasını sağlamak için deneyen araç ve teknikleri oluşturmakta ve değerlendirmekteyiz. Kısaca, gerçeğe sahip olmadığımız zaman seçenek ayar birleşimlerini test etmiş olduğumuz düşünceye iten maskeleme etkilerini engellemek istiyoruz.

Bunu yapmak için ilk olarak t-yollu etkileşim örtmenin tanımlamamızı değiştiriyoruz, bu değişim aşağıdaki gibidir: verilen bir t değeri için, yapılandırma uzayı kapsamaktadır tüm test durumları için seçenek ayarlarının ve tüm geçerli s ≤ t-yollu birleşimlerini en azından bir yapılandırmada gösterildi, ki bunlarda 1) test durum geçildi, 2) seçenek ayar birleşimi başarısız bir sebep olarak tasarlanılmaktadır, 3) seçenek ayar birleşimi, test durumun seçenek-ilişkili olmayan bir sebeple başarısız olduğu en azından bir yapılandırmada gösterildi.

Bu kriter için gerekçe aşağıdaki gibidir; bir test durumu verilen bir yapılandırmada başarılı bir şekilde çalıştığı zaman, tüm s ≤ t-yollu seçenek ayar birleşimleri yapılandırmanın bu test durumuyla beraber test edildiği durumda mevcuttur. Bazı spesifik seçenek ayar birleşiminin hata yapmak için bir test durumuna yol açtığını belirlediğimiz zaman, sonra bu birleşimi test etmiş olmaktayız, fakat yapılandırmada diğer adı geçen birleşimler test edilmeyebilir. Bir test durum başarısız olduğunda, fakat seçenek-ilişkili bir nedeni belirleyememekteyiz, sonra bir maskeleme etkisi belirlememiş olmaktayız ve yapılandırmada tüm adı geçen seçenek ayar birleşimlerinin bu test durumuyla test edilmiş olduğunu varsaymalıyız.

Sınırlamanın bu tanımını kullanarak, bir sistem, onun yapılandırma uzay modeli ve girdi olarak test durumlarının bir kümesini alan otomatik bir süreç sonrasında uygulamaktayız ve bizim yeni kriterimiz altında tam örtmeye ulaşmayı tekrarlayarak teşebbüs etmekteyiz.

3.1.Sürece Bakış

Yüksek bir seviyede bizim geridönüşüm güdümlü adaptif kombinasyonel test etme yöntemimiz aşağıdaki gibi çalışmaktadır:

1. Kapsama kriterini karşılayan yapılandırmaların örtme kümesini üret.
2. Örtme kümesindeki her yapılandırma üzerinde her test durumunu yürüt.
3. Olası maskeleme etkilerini tanımlamak için test durum sonuçlarını analiz et.
4. Önceden maskelenen etkileşimleri test etmek için yeni örtme kümesi tekrardan üret.
5. Eğer bütün kapsama(coverage) başarılamazsa, 2. adıma dön.

Şimdi süreçteki her adım için genel bir özet ve mantıklı bir açıklama sunmaktayız.

Adım 1: Örtme Kümesi Üretme

Sürecimizin her iterasyonunda, mevcut iterasyonda kapsanılcak olan, her test için bir tane, t-way birleştirme kümesi hesaplamaktayız. Test etme maliyetini azaltmak için, mümkün olduğunca birkaç somut yapılandırmaları ve test çalışmaları olarak kullanan bu birleşimleri kapsamayı istiyoruz.

Sonuç olarak, bu amaçla geleneksel bir t-way örtme dizisi hesaplamaya karar verdik. Bu yaklaşımla gelen bir problem şudur ki her test durumunun kendi testine özgü kısıtlara sahip olabilmesidir. Örneğin her test durumu, bazı yapılandırmalarda çalışacak diğerlerinde çalışmayacak şekilde tasarlanabilir. Bu nedenle, tüm test durumları karşısında tek bir örtme dizisi inşa etmek, daha geniş örtme dizilerine neden olabilir. Örneğin, MySQL(geniş ölçüde kullanılan açık kaynak veritabanı yönetim sistemi) ile yapılan önceki çalışmada, yaklaşık 1000 test durumundan 250'si sistem belli şekilde yapılandırılmadığında çalışmaz. Özünde, bu durumlar için, yapılandırma uzayının geniş bir oranı bulunmamaktadır.

Çoklu test durumları için birleştirilmiş örtme dizisi üretimini ele alan herhangi bir varolan araştırma veya araç bulamadık. Ayrıca biz örtme dizisi hesaplamasında teste özgü kısıtları tutmak için dar bir yaklaşım geliştirdik. Yaklaşımımız, ilkel bir hesaplama olarak örtme dizisi yapımı kullanmaktadır. Bu durumda ACTS olarak iyi bilinen bir araç kullandık. Şunu da söylemek gerekir ki diğer araçlar da çalışırdı.

ACTS, girdi olarak bir yapılandırma modeli alır. Model

yapılandırma seçenekleri, onların ayarlarını, global olarak uygulanan gömülü-seçenek kısıtlarını ve bir *seed* içermektedir. Bir dizi verildiğinde(örneğin t gücünde), ACTS, seed etrafında geleneksel bir örtme dizisi üretir. Kavramsal olarak, ACTS henüz kapsamış olarak seed'de yer alan tüm geçerli t-way birleşimleri işletir ve birleşimlerin geri kalanını kapsamak için yeni yapılandırmalar üretir.

Buna göre, her test için ayrı bir yapılandırma altmodeli ekleriz. Bu altmodel, global olarak uygulanması gereken devralan tüm gömülü-seçenek kısıtlarına ek olarak , teste özgü kısıtlar içermektedir. Dahası, test tarafından zaten kaplanıldığı düşünülen seçenek ayarlarının tüm t-way birleşimleri modelde bir *seed* olarak anlatılmaktadır. Bir örnek olarak, şekil 2a-b bizim çalışan örneğimizdeki testler için yaratılan yapılandırma modellerini göstermektedir. Biz sadece test kısıtlarını ve *seed*'leri göstermekteyiz.(temel model şekil1'den çıkarılabilir.)

Her test için, onun yapılandırma modelini ACTS'a verdik. Çıktı henüz örtüsüz yapılandırmaları içeren bir örtme dizisidir. Daha sonra testler için üretilen örtme dizilerini birleştiririz ve bir sonraki iterasyonda yapılacak olan test çalışmalarının sayısının yanı sıra test edilecek olan yeni yapılandırmaların sayısını azaltmak için bir *greedy reduction* algoritması uygulamaktayız.(Bölüm 3.2)

Bu sürecin çıktısı her yapılandırmada yürütülecek olan test durumlarının bir listesi ile birlikte son yapılandırma kümesidir. Şekil 2-c , şekil 2a-b'de verilen yapılandırma modeli için hesaplanan 3-way test-farkında örtme dizisini göstermektedir.

ADIM 2: TEST DURUMLARINI YÜRÜTME

Her iterasyonda, kapsayan kümede ve onların geçti/başarısız sonuç kayıdında testleri yürütürüz. Test sonuçları daha sonra yapısal Tablo 1'de verilen birine benzer bir veri tablosu düzenlenir. Bu çalışmaya dahil ettiğimiz bir isteğe bağlı adım ise, bir kurulum testi denilen özel bir test olarak, kurulum sistemi sürecini denememizdir. Bu test, tabii ki, derlenmiş, düzgün-yapılandırılmış sistem mevcut olmadıkça, sürece diğer geleneksel çalışma zamanı testlerinden önce çalışması gerekir. Kurulum testi için, geçici bir sonuç, herhangi bir kurulum hatası olmadan sistem kurulumu anlamına gelir. Başarısızlık bazı kurulum hatasının oluştuğu anlamına gelir. Herhangi diğer düzenli test olduğu gibi, biz kurulum testleri için tam bir kapsam başarmak istiyoruz.

Adım 3: Test Durumu Analizi Sonuçları: Daha sonra test sonuçlarını başarısızlıklara neden olan ve böylece potansiyel maskeleme efektleri yaratan kombinasyonları ayarlayan seçeneği belirlemek için analiz edeiz. Tam anlamıyla otomatik bir şekilde bunu yapamadığımız için, yerine otomatik olarak olası başarısızlık nedenlerini tespit eden sınıflandırma ağaçları adı verilen, bir öğrenme makinesi yaklaşımı kullanırız. Sınıflandırma ağaçları ölçülebilir özelliklerin bir kümesi açısından sınıf üyelerini (örneğin bir test durumunu geçme veya geçememe) tahmin eden bir model kurmak için özyinelemeli bir bölümleme yaklaşım kullanır.

Örneğin, Şekil 1a'dan sınıflama ağaç algoritmasına kadar test sonuçları verileri verilmiştir, Şekil 1b'den gösterilenler gibi üç sınıflama modelleri, her bir test durumu için yaratacaktır. Hiç yaprağı olmayan düğümler seçenekleri temsil eder, kenarlar seçenek ayarlarını ve yaprak düğümler beklenen test sonuçlarını gösterir. Basit sınıflama modeli t1 için elde edilmiştir. Örneğin, test durumu t1 o1 == 1 olan yapılandırma üzerinde çalıştığı zaman, test durumu başarısız beklenebilir. Aksi takdirde, test durumunun geçmesi beklenebilir. Bu basit sınıflandırma modelleri, ancak test durumu başarısızlığını gösteren tek bir yaprak düğümüne sahiptir, fakat genelde daha fazla bu tür yaprak düğümleri olamazdı. Bu durumlarda, bir hata olduğunu gösteren her yaprak düğümünü inceleyerek, etkileşimleri uyaran tüm olası başarısızlıkları elde edebiliriz. Her bir yaprak düğüm için, ağaç kökünden yaprağa kadar ve yolu üzerinde bulunan seçenek ayarlarının birlikte karşılık gelen bir mantıksal kural olan çıktıyı tanımlarız. Bu kural test durumu başarısız sonuçlandığı zamanki birleşimlerin seçenek ayarının bir dizisini gösterir. Sınıflandırma ağaçları üretirken, *overfitting* verileri önlemek için çeşitli adımlar attık. Bu, bizim sınıflandırılmış modellerin rasgele hata veya başarısızlık nedenleri olarak gürültü denemesi olmadığından emin olmak için çalışıyoruz. Kullandığımız bir standart teknik olan "n-katmanlı çapraz doğrulama" kullanarak sınıflandırma modelleri oluşturmaktır. Bu yaklaşım aslında, birçok farklı girdi verisinin farklı altkümelerinden çoklu modeller oluşturur ve aşırı birkaç bireysel veri puan tarafından etkilenmemiş aday modelleri tanımlamak için sonuçları kullanır.

Son olarak, etkileşim uyaran her olası başarısızlık için, , test verilerini başarısızlık tahmin etmede kural başarısı gösteren, F1 önlemi olarak adlandırılan bir puan atarız. F1 ölçeği iki standart ölçümleri birleştirerek hesaplanır. Tahmin (P), hatırlama (R):

$$recall = \frac{\#\, of\, correctly\, predicted\, failures\, by\, R}{total\, \#\, of\, failures}$$

$$precision = \frac{\#\, of\, correctly\, predicted\, failures\, by\, R}{total\, \#\, of\, predicted\, failures\, by\, R}$$

$$F1 - measure = \frac{2PR}{P + R}$$

F1-ölçü aralıkları 0 ile 1 arasındadır. Değer ne kadar yüksekse, daha iyi kural tahmin başarısızlıklarında o kadar iyidir. Tablo 1 bizim çalışan örneğimizde elde edilen kurallar için F1-önlemlerini gösterir. Bu makalede, ilgili kural puanı limit(cutoff) olarak adlandırılan önceden belirlenen değerden büyükse, bir etkileşimin başarısızlık nedeni olduğu düşünülür. Önemli bir başarısızlık nedeni tarafından açıklanmayan herhangi başarısızlıkların hiç seçenek ilişkili olmadığı düşünülür.

Bir örnek olarak, tablo 1'deki test t2'yi düşünelim. Bu test durumu için ilk dört başarısızlık o1==1 olduğunda meydana gelmiştir. 0.89'un F1-ölçüm limitinden

yukarda olduğunu varsayarak, bu başarısızlıkların seçenek ilişkili olduğu düşünülür. (Örneğin o1==1 olması olası bir başarısızlığa neden olan nedendir.) Ancak, beşinci başarısızlık, hiç seçenek ilişkili olmadığı düşünüldüğü için önemli bir kurala dayandırılamayabilir.

Adım 4: Örtme kümelerini tekrar etme

Geçmiş adımdan gelen çıktı bize tüm test durumu için birçok bilgi parçası vermektedir. İlk olarak, test durumu geçtiği için biz kapsayan her etkileşimi bilmekteyiz. Test durumunun geçildiği en azından bir yapılandırmada adı geçen etkileşimler vardır. İkinci olarak, test durumu başarısız olsa dahi kapsayan her etkileşimi bilmekteyiz. Test durumunun başarısız olduğu ve başarısızlığın seçenek-ilişkili olmaması ve başarısızlığa neden olmanın olası olduğu bu etkileşimlerin olduğu en azından bir yapılandırmada bunlar adı geçen etkileşimleri kapsamaktadır. Üçüncü olarak, potansiyel olarak maskelenen her etkileşimi bilmekteyiz ve bu yüzden geriye kalanlar kapsanmaz.

Bu bilgiyi kullanarak, amacımız mevut kapsanmayan etkileşimleri kapsayan yeni yapılandırmaları içeren güncellenmiş bir örtme kümesini üretmektir. Bunu yapmadan önce birçok hazırlık aşamaları almaktayız. İlk olarak her test durumu için yeni tüm kapsayan etkileşimleri tanımlamaktayız ve onları test-spesifik yapılandırma modellerine çekirdek olarak ekliyoruz. Sonra, her yeni tanımlanan başarısızlığa neden olan etkileşimin tamamlayıcısını almaktayız ve onları her test-spesifik yapılandırma modeline sınırlamalar olarak ekliyoruz. Buna ek olarak, çalışma zamanı testleri için test-spesifik modeller, kurulum test başarısızlığından kaynaklanan etkileşimlere neden olan herhangi başarısızlığı dahil etmektedir. Son iki adımın her ikisi de sonraki test iterasyonlarında, bilinen başarısız sebepleri sakınan sürece yardım etmektir.

Daha sonra her teste özgü yapılandırma modelinin yetersiz kısıt içerip içermediğini belirlemek için onları incelemekteyiz. Böyle durumlar, bir test durumu çoklu yollarda başarısız olduğunda ortaya çıkabilir. Örneğin, farzedelim ki bir tane test durumu belirli bir ikili seçenek doğru olduğunda bir şekilde başarısız olmaktadır ve aynı seçenek yanlış olduğunda farklı olarak başarısız olmaktadır. Bu durumda sürecimiz hiçbir yapılandırmanın karşılayamadığı çelişkili kısıtlar üretir. Her iterasyonda sadece hiç çelişmeyen bir küme kısıt tutarak döngülü bir şekilde çatışmaları karşılama girişiminde bulunmaktayız. İlerleyen döngülerde tutulacak olan geriye kalan kısıtları ertelemekteyiz.

İleriki buluşlar olarak, buluşsal bir kural ile önceliklendirme stratejisini de geliştirdik ve denedik. Ön çalışmada, daha uzun kuralların daha kısa kurallardan yanlış etiketlendirmeleri vermesinin daha çok olası olduğunu analizlerden gözlemledik. Sonuç olarak, bu buluşsalı kullanarak mevcut iterasyonda üretilen tüm kurallara bakarız ve en kısa kuralın uzunluğunu hesaplarız. Daha sonra bu uzunluğun tüm kurallarını alırız ve uzunluğu en kısa kuralla aynı olduğu kurallarla devam ederiz. Kuralların geri kalanı ilerleyen döngülere ertelenir.

Son olarak, tüm etkileşimleri kapsayan bütün test durumlarını kaldırırız. Eğer herbir ve bütün test için tüm kapsam başarılı olmuşsa, süreç biter. Aksi halde, yeni kapsayıcı küme üretiriz ve 2nci basamağa geri döneriz.

Algorithm 1 – **adaptiveCA**

Input t: Covering array strength
Input $cfgMdl$: Config. model

1: $fMatrix \leftarrow empty$
2: $knownCauses \leftarrow empty$
3: $currentCA \leftarrow computeCA(t, cfgModel)$
4: **repeat**
5: $\quad executeCA(currentCA, fMatrix)$
6: $\quad currentCA \leftarrow prepareNextRound(t, cfgMdl, fMatrix, knownCauses)$
7: **until** $currentCA$ is $empty$

Algorithm 2 – **prepareNextRound**

Input t: Covering array strength
Input $cfgMdl$: Config. model
Input $fMatrix$: Fault matrix
Input $knownCauses$: Likely failure inducing option combinations

1: $nextCA \leftarrow empty$
2: **for** each test τ **do**
3: $\quad knownCauses_\tau \leftarrow identifyCauses(fMatrix_\tau, knownCauses_\tau)$
4: $\quad cfgMdl_\tau \leftarrow updateCfgMdl(fMatrix_\tau, knownCauses_\tau)$
5: $\quad CA_\tau \leftarrow computeCA(t, cfgMdl_\tau)$
6: $\quad CA'_\tau \leftarrow reduceCA(CA_\tau)$
7: $\quad nextCA \leftarrow nextCA \cup CA'_\tau$
8: **end for**
9: **return** $nextCA$

3.2.Algoritma

Verilen önceki tartışma, bizim temel algoritmamız takip etmeye kolay olmalıdır. Algoritma 1, temel adaptiveCA yordamını tanımlar. AdaptiveCA, her iterasyonda birkez prepateNextRound'a bir çağrım yaparak, test yapılacak hiçbir test durumu işletilmeyene kadar döner.

Bu yordam 3 tane temel veri yapısı tanımlar ve kullanır: *fMatrix, knownCauses* ve *cfgMdl. fMatrix,* test edilen tüm yapılandırmaların izini, onların üzerinde yürütülen test durumlarını ve elde edilen test sonuçlarını saklayan aksaklık matrisidir. k*nownCauses,* etkileşimlere neden olan tüm olası bilinen başarısızlığın izini saklar. c*fgModel* ise seçenekleri,

İsim	Ayarlar	Tip
asm	{NULL,enable-assembler}	ct
linfile	{NULL,enable-local-infile}	ct
tsc	{NULL,disable-thread-safe-client}	ct
bt	{NULL,with-big-tables}	ct
ec	{NULL,with-extra-charsets=complex, with-extra-charsets=all}	ct
innodb	{with-innodb,without-innodb}	ct
libedit	{with-libedit,without-libedit}	ct
ndbcluster	{NULL,with-ndbcluster}	ct
pic	{NULL, with-pic}	ct
readline	{with-readline,without-readline}	ct
ssl	{NULL,with-yassl}	ct
zdir	{NULL,with-zlib-dir=bundled}	ct
ase	{NULL,with-archive-storage-engine}	ct
bse	{NULL,with-blackhole-storage-engine}	ct
fse	{NULL,with-federated-storage-engine}	ct
trans.-iso.	{NULL,uncommitted, serializable, committed, repeatable}	rt
innodb_flush	{NULL, 0, 1, 2}	rt
sql_mode	{strict_all_tables, traditional, ansi}	rt
lp	{NULL,large-pages}	rt
eng-pdown	{on, off}	rt
rbt	{NULL,big-tables}	rt
binlog-format	{row, statement, mixed}	rt
lb	{skip-log-bin,log-bin}	rt

Tablo 1: Çalışmada kullanılan MySQL'in yapılandırma modeli. ct değerleme zamanına, rt ise çalışma zamanına karşılık gelmektedir

onların ayarlarını, onlar arasındaki kısıtları ve bir seed'i saklayan bir küme yapılandırma modelidir. Bu değişkenler bir test T'ye indislendiğinde, herkes kendi veri yapılarında teste özgü bilgiye ait olmaktadır. İndisli değişkenler üzerinde yapılan tüm değişiliklerin orijinal değişkenleri yansıttığı varsayılır.

Temel adaptiveCA yordamının girdi olarak ilk bir yapılandırma modeli *cfgNdl* ve bir *t* gücünü aldığını görebiliriz. İlk olarak *fMatrix*'i ve *knownCauses* veri yapılarını ilklendirir. Satır 3'te ilk bir t-way örtme dizisi yaratır ve test etme döngüsüne girer. Satır 5'de tüm seçilen yapılandırma/test durumu ikilileri test edilir ve sonuçlar döndürülür. Bu test sonuçları daha sonra onların analiz edildiği ve bir sonraki iterasyonda test edilecek olan örtme dizisinin hesaplandığı prepareNextRound'a aktarılır.(satır 6) Yeni hesaplanan örtme dizisi boş olduğunda (bu da bizim kapsama kriterimiz içinde tam bir kapsamanın elde edildiğini gösterir), döngü sonlarıır.(satır 7)

Algoritma 2 aşağıdaki gibi çalışan prepareNextRound'u göstermektedir. Her test için, ilk olarak bir sınıflandırma ağacı algoritması kullanarak etkileşimlere neden olan olası başarıszlığı belirler. Sınıflandırma modelini hesaplamak için, hiç seçenek-ilişkili nedeni olmayan testin ya başarılı ya da başarısız olduğu şuana kadar test edilen tüm yapılandırmaları içeren uygun veri alanları yaratırız. Eğitim verisi sınıflandırma algoritmasına verilir. Olası başarısızlıklara neden olan seçenekler sonuçlanan sınıflandırma modelinden çıkarılır ve daha sonra puanlandırılır.(Bölüm 3.1) Olası başarısızlığa neden olan birleşimler arasından, verilen limit değerden daha büyük bir puana sahip olan sadece bir tanesi seçilir. Kalan önemsenmez. Önceden de bahsedildiği gibi, overfitting'i düşürmek ve çelişen kısıtları tutma stratejileri de bu adımda gerçekleştirilir.(Bölüm 3.1)

Ardından, teste özel yapılandırma modelini güncelleştiririz. (satır 4) Bunı yapmak için, yeni olarak tanımlanan teste özel kısıtlarla test durumu için kısıt listesini ilk olarak doldururuz. Şunu not etmek gerekir ki, bir testin yapılandırma modelinde yer alan kısıt listesi monotonik olarak ilerlemektedir. Yani bir kısıt bir yapılandırma modelinde içerildiğinde, sürecin yaşam süresi boyunca orada kalır. Bu, süreci önceden belirlenen başarısız alt alanları incelemeden önler. Daha sonra zaten örtmüş olduğu düşünülen t-way birleşimleri gösteren testin yapılandırma modeli için bir seed hesapladık. Bu *seed* şuana kadar test edilen tüm yapılandırmaları içerir.

Daha sonra testin yapılandırma modeli ACTS aracına verilir.(satır 5) Çıktı, test durumunun bir sonraki iterasyonda yürütmesi gerektiği bir küme yapılandırmadır. Seed edilen yapılandırmalar, kapsam açısından gereksiz yapılandırmalar içerebildiği için, bu adımda(satır 6) gereksiz yapılandırmaları belirlemek ve elemek için *greedy* algoritması kullanmaktayız.

Son olarak her test için hesaplanan örtme dizilerini birleştiririz(satır 7) ve daha sonra onu geri döndürürüz.(satır 9) Sonuç, bir sonraki iterasyonda test edilecek olan yapılandırma-test ikilisidir.

Tablo 1: Çalışmada kullanılan MySQL'in yapılandırma modeli. ct derleme zamanını, rt ise çalışma zamanına karşılık gelmektedir.

4. Deneyler

Yaklaşımımızı değerlendirmek için, bir küme deneysel çalışmalar yönettik. Bu çalışmalar için söz konusu

Iterasyon	Test edilen cfgs	Test çalışmaları	Örtünen satırlar	Örtünen dallar	Eşsiz errs	Eşsiz test-errs	T'li maskeleme	Test etme zamanı	Analiz zamanı	Toplam zaman
1	20	5896	161556	82316	96	1052	525149	266	1	267
2	31	12529	223074	108606	117	1380	278228	818	6	824
3	34	14740	223164	108679	117	1381	242368	1005	10	1015
4	71	17588	223455	108890	118	1388	186919	1604	26	1630
5	121	18599	223621	108967	118	1388	180064	2003	44	2047
6	159	19395	223656	108993	118	1388	179174	2382	74	2456
7	167	19585	223663	108995	118	1388	179166	2455	115	2570
8	178	19753	223669	109004	118	1388	178798	2634	147	2781
9	181	19765	223669	109004	118	1388	178794	2656	247	2903
10	182	19769	223669	109004	118	1388	178794	2665	300	2965

Tablo 2: t=2 ile önerilen yaklaşım

sistemler Mysql (v5.1) ve gcc(v.4.5.2)dir. Mysql bir veritabanı yönetim sistemidir. Gcc ise *GNU Compiler Collection* versiyon 4.5.2'nin resmi sürümüdür. Her iki sistem de herkesce erişilebilir.

4.1. Deneysel Ayarlar

Bu deneylerde, covering-dizi'yi oluşturmak için ACTS aracını kullandık. Güven faktörünü 0.25'e ve her birindeki minimum izin verilebilir nesne sayısını 2'ye ayarlayan sınıflandırma ağaçlarımızı oluşturmak için Weka'nın J48 sınıflandırma algoritmasını kullandık. Sınıflandırma modelleri beş katmanlı çapraz doğrulama ve budama ile eğitilip değerlendirilmiştir. Olası başarısızlığa neden olan etkileşimleri belirlemek için kullanılan limit değer 0.8'e ayarlanmıştır. Daha uzun kurallar üzerinde daha kısa kuralları destekleyen önceliklendirme buluşunu kullandık ve döngüsel bir şekilde çakışan kısıtları çözdük. (Bölüm 3)

Bütün deneyler CentOs 5.2 işletim sistemi üzerinde çalışan 2gb Ram'li Intel Xeon çift çekirdekli işlemci makine üzerinde gerçekleştirildi. Kendi çalışma bölümlerinde belli ölçevleri ve konuları tanımlamaktayız.

4.2. Çalışma 1:MySQL Deneyleri

MySQL, açık kaynak kodlu, çoklu iş parçacıklı, sql veritabanı yönetim sistemidir. 12 yıl önce ilk olarak sürümü yayınlanmıştır. Çeşitli bileşenleri iki milyondan daha fazla kod satırı içermektedir. On milyondan daha fazla kez indirilmiştir. Yirmi platform üzerinde kullanım için uygundur. MySQL çok sayıda test durumuna (Hem kurulum testlerini hem de genel testleri içermektedir.) ve aktif bir şekilde sistemi güncelleyen ve test eden geniş bir geliştirici topluluğuna sahiptir.

4.2.1. Kurulum Çalışması

MySQL için 15 tane derleme-zamanı, 8 tane çalışma-zamanı olmak üzere 23 tane seçenek içeren başlangıç bir konfigurasyon model yarattık. Setting'lerin sayısı konfigurasyon seçenekleri arasında farklılaşmaktadır. 18 tane seçenek 2-seviyeli kuruluma, 3 seçenek 3-seviyeli, 1 seçenek 4 ve diğer seçenek 5-seviyeli kurulama sahiptir.(Tablo 1) Konfigurasyon model başlangıçta hiçbir kısıtlamaya sahip değildir.

Bizim test takımımız için, MySQL kaynak dağıtımıyla gelen 738 tane test durumu kullandık. Her test kendi oracle'ına sahiptir. Oracle'lar her test çalışmasını 3 sınıf halinde kategore etmektedir: Başarılı, başarısız, skip? Başarılı test durumları başarıyı emit eder. Başarısız test durumları başarısızı emit eder ve ek olarak bir hata mesajı içerir. Bir test durumu verilen bir konfigurasyon üzerinde çalışamadığını belirlediğinde “skipped” mesajını döndürmektedir. Örneğin bir sayıdaki test durumu sadece MySQL bir iç-bellek kümeli depolama motoru olan NDB destek kümesi ile yapılandırılırsa çalışacağı şeklinde tasarlanmıştır. Eğer mevcut konfigurasyon NDB'yi desteklemezse, bu test durumları “skipped” sonucunu döndürüp hemen çıkacaktır. Sınıflandırma modelleri bu test durumları için inşa edilir ve ayrıca ikili sınıflandırmadan ziyade üçlü sınıflandırmayı kapsamaktadır.

4.2.2.Değerlendirme Çerçevesi

Kesin olarak önerilen yaklaşımı değerlendirmek için, test durumları (örneğin maskelenenler) ile hiçbir şekilde test etme şansına sahip olmayan geçerli etkileşimlerin sayısını bilmemiz gerekir. Daha sonra bu

konuda önerilen yaklaşımın nekadar iyileştiğini analiz etmemiz gerekir. Ancak bu, bizim tüm hataya neden olan etkileşimleri el ile belirlememizi ve deneylerde test edilen yüzlerce yapılandırma üzerinde meydana gelen on binlerce hata için maskeleme etkilerini ölçmemizi gerektirdi. Bu olası olmadığı için, kesin bir nedene sahip olduğumuz ve probleme neden olan maskeleme etkilerinin ne olduğunu bildiğimiz sadece bu başarısızlıklar üzerinde yaklaşımı değerlendirmeyi tercih ettik. Bu yüzden *build* hatalar ve test *skip*'leri tarafından neden olan maskeleme etkileri üzerinde yaklaşımı değerlendirmekteyiz.

T-maskeleme olarak adlandırılan bir ölçev tanımladık. Bu ölçev, build hataları veya test skip'leri yüzünden test durumları ile hiç test edilmemiş eşsiz t-way birleşim-test ikilisinin sayısını sayar. Eğer bir yapılandırma kurulum (build) hatası veriyorsa, Hiçbir test durumunun yapılandırmadaki geçerli t-way etkileşimlerini test etmediğini biliyoruz. Benzer şekilde, eğer bir test durumu bir yapılandırmayı atlıyorsa(skip), yapılandırmadaki t-way birleşimlerinin hiçbiri test edilmeyecektir. T-maskeleme değeri nekadar az olursa yaklaşım makeleme etkilerini silmede okadar iyi olur. Kurulum (build) başarısızlıkları gerçek başarısızlık iken, olacak test atlamalarını(skip) göz öününde bulundurmayız. Geliştiriciler tam olarak her testin *skip* edebilme nedenini bir dereceye kadar bilirler. Örtme dizisi yapılandırmasına test kısıtlarını tanıtarak konuyu ele alabilirler. Bu yüzden t-maskeleme yaklaşımımızın pratik değerini büyütmektedir. Bununla beraber, böyle test atlamalarını hesaba katmamanın etkisi, bizim tekniğimizin göstermeyi hedeflediği problemi olan etkileşimlerin test etmemesi ve onun için deneyin yaklaşımımızın kullanışlı bir testi olmasıdır.

t-maskelemeye ek olarak, gözlemlenen eşsiz hatalar ve eşsiz test-hata ikilisinin yanısıra kapsanan kaynak satır ve dalların sayısını da sayar. Eşsiz hatalar ve test-hata ikilisini belirlemek için, test durumları başarısız olduğunda yayımlanan hata kodlarını analiz ediyoruz. Bu ölçevler, test etme sürecimizde maskelemenin etkisini ölçmemizin başka bir yolunu vermektedir.

4.2.3. Veri ve Analiz

İlk olarak t=2 ile süreci başlattık. Tablo 2 sonuçları göstermektedir. Bu tabloda sutun 1-3 iterasyon sayısını, test edilen konfigurasyon sayısını ve yapılan test çalışmalarının sayısını göstermektedir. Bir sonraki üç sütun, eşsiz hataların, eşsiz test-hata ikilisinin sayısını ve t-maskeleme'nin değerini göstermektedir. Son üç sütun ise test etme zamanı, analiz zamanı ve toplam zamanı göstermektedir. Tüm zaman ölçümleri dakika cinsinden verilmiştir. Üstelik, bir iterasyon için bildirilen tüm sayılar, mevcut olana kadarki(o da dahil) tüm iterasyonların üzerinden elde edilen ölçümleri yansıtır.

T	Cfgs	Test çalışmaları	Örtünen satırlar	Örtünen dallar	Errs	Test-err ikilisi	Toplam zaman
2	20	5896	161556	82316	96	1052	267
3	72	18425	216612	106506	119	1377	1775
4	248	67804	218170	107187	122	1428	4681

Tablo 3: Geleneksel t-way örtme dizisi tabanlı test etme

Bu deneyde, tam kapsama(full coverage) ulaşmak 10 iterasyonda gerçekleşti. İlk iki iterasyon, iki kurulum hatası gösterdi. Test atlamaları (skip) kalan iterasyonlarda gösterildi. İlk döngüden sonra(örneğin geleneksel 2-way test etmeyi yaptıktan sonra) kurulum hataları (builf failure) veya test atlamaları (test skip) yüzünden, tüm geçerli 2-way birleşim-test ikililerinin %55'inin gerçekten maskelendiğini gözlemledik. Bu hatalar arasında, süreç, kurulumda başarısız olan, 20 yapılandırmadan 11'sine neden olan ssl=with-yassl seçenek ayarını belirledi. Daha sonra birleşimin mantıksak eksikliği(örneğin ssl!=with-yassl) otomatik olarak yapılandırma modeline eklendi ve ssl=NULL ile 11 tane yeni yapılandırma kümesi hesaplandı. İkinci iterasyonda bu yapılandırmaları test etme büyük bir ölçüde %47 ile maskelenen ikililerin sayısını azalttı.

İkinci iterasyondan sonra, libedit=with-libedit and readline=with-readline arasındaki bağımlılığa neden olan bir başarısızlık doğru bir şekilde tanımlandı. İlginç bir gözlem şudur ki, bu hata ilk döngüde gözlemlenmesine rağmen daha sonra yaratılan sınıflandırma modelleri herhangi bir istatistiksel deseni ortaya çıkaramamıştır. Nedeni, bu bağımlılığa sahip olan ilk döngüdeki yapılandırmalardan biri hariç tümünün ssl hatası yüzünden zaten başarısız olmasıydı. Yani ilk başarısızlık, ikinci başarısızlığı maskeledi. Ancak mevcut iterasyondaki ilk başarısızlığı önleme, daha fazla yapılandırmadaki ikinci başarısızlığı gözlemlemeyi olası yaparak, bizim doğru bir şekilde ikinci başarısızlığın nedenini belirlememize yardım etti. Toplamda 3 yapılandırma, hata yüzünden kurulumda başarısızdı. Üç yeni yapılandırma kümesinin (örneğin libedit!=with-libedit readline!=with-readline ile) bir sonraki iterasyonda test edilmesi planlanmıştır. Üçüncü iterasyonda onları test etmek, bir önceki iterasyonla kıyaslandığında %13 ile maskelenen ikililerin sayısı azalmıştır.

Üçüncü döngüde sonuncu döngüye, artık kurulum hatası olmadığı için, süreç test atlamalarını(test skips) gösterdi. Toplamda, bilinen kısıtlarla 298 testin 225'i (%76'sı) için gerçek test kısıtları doğru bir şekilde

tanımlanmıştır. Bu, ek bir %26'lık maskelenen ikililerin sayısını azaltmıştır.

Bizim neden tim test kısıtlarını belirleyemedğimizi anlamak için, el ile bir analiz yürüttük. Sebebin, bazı test atlamalarının deterministik olmadığını belirledik. Burada, bir nedenden ötürü atlanan bazı test durumları yapılandırmayla ilişkili değildir. Başlangıç çok uzun bir süre aldığı için, gerçek nedenin zaman aşımı olduğuna inanıyoruz.

Bu kesintili atlamaların çoğunda sınıflandırma modellerinin desenleri belirlediğini fakat daha düşük bir güvenle ile yaptığını gözlemledik. Önerilen yaklaşımdaki bu konuyu gösteren bir yaklaşım, kesim parametresi için daha düşük bir değer kullanmaktır. Örneğin 0.8'in yerine deneylerde kesim parametresi olarak 0.6'yı kullanmamız, gerçek test kısıtlarının %94'ü dördüncü iterasyonun sonunda doğru bir şekilde tespit edilmiş olurdu.

Sürecin sonunda, maskelenmekte olan 2-way birleşim-test ikililerin sayısı ilk iterasyonla kıyaslandığında %66'lık bir düşüş göstermiştir. Üstelik, maskeleme etkilerini önlemek, kaynak satır kapsamasını(source line coverage) %39'luk, dallanma kapsamasını(branch coverage) %32'lik, eşsiz hataların sayısını %23'lük ve eşsiz test-hata ikililerinin sayısını %32'lik geliştirdi.

Bu gelişmeler, yapılandırmalar ve test yürütmelerinin sayısı artırılmış pahasına elde edilmiştir. Bu durum beklenmektedir. Çünkü ilk döngünün sonunda, örneğin biri kurulum hatalarından kaynaklanan maskeleme etkikerini kaldırmak isterse, tek şans yeni yapılandırmalar seçmek ve tüm test durumlarını onlarda çalıştırmaktır. Önemli bir gözlem olarak şunu söyleyebiliriz ki, iyileştirmelerin çoğu sürecin erken aşamalarında elde edildi. Örneğin dördüncü iterasyonun sonunda iyileştirmeler, sonunda elde edileninkinden %4 daha az içindeydi. Bu iterasyondan sonra durduğumuzda , son döngü ile kıyasladığımızda, test edilen yapılandırmaların sayısı, yapılan test yürütmelerinin sayısı ve toplam süre sırasıyla %60, %11 ve %45 düşürülmüştür. Test etme kaynakları korkulduğunda ve önceliklendirme gerekli olduğunda bu durum önemlidir.

Diğer bir gözlem ise, dördüncü iterasyondan sonra, her sonraki iterasyonda (sonuncusu hariç), süreç yeni maskeleme etkilerini silmesine rağmen, yapısal kod kapsama ölçümleri ve eşsiz hatalar ve test-hata ikilileri sayısının dengelendiği görülmektedir. Biz bunu test edilmekte olan yazılıma, onun test takımına ve çalışma için seçilen yapılandırma modeline bağlamaktayız. Testlere yeni birleşimleri çalıştırma şansı verilmiştir. Fakat bu gözlemlenen ölçevleri yansıtmadı.

Daha Yüksek Güçlü Geleneksel Dizilerle Kıyaslama

Bir sonrakinde, t=2 ile yaklaşımımızı kullanmayı, geleneksel 3-way ve 4-way örtme dizisi tabanlı test etmeyi kullanarak kıyasladık. Daha yüksek güçlü örtme dizisi kullanmak sonuçlarımızı kıyaslamada en iyi adaydır. Örneğin, seçenek ayarlamalarının 1-way veya 2-way birleşimlerinden kaynaklanan maskeleme etkikerini önlemek için 3-way örtme dizisi kullanırsak, her 1-way ve 2-way birleşimi, farklı 3-way birleşimlerinde çoğu kez görülecek ve maskelemeyi önleyebilecek. Tablo 3 geleneksel test etme yaklaşımları üzerindeki performansı özetlemektedir.

Yaklaşım	2'li maskelenen	Düşüş
Önerilen yaklaşım	178794	n/a
Geleneksel 2-way	525149	%66
Geleneksel 3-way	240489	%26
Geleneksel 4-way	133977	%-34

Tablo 5: t=2 ile önerilen yaklaşımı geleneksel tway test etme ile kıyaslama. İleriki analizler için dökümana bakınız.

Yaklaşım	3'lü maskelenen	Düşüş
Önerilen yaklaşım	2797589	n/a
Geleneksel 3-way	6670639	%58
Geleneksel 4-way	2970152	%6

Tablo 6: t=3 ile önerilen yaklaşımı geleneksel t-way test etme ile kıyaslama

Nekadar güçlü örtme dizilerinin maskeleme etkilerini silmede bize yardım edeceğini bilmek istedik. Tablo 5, geleneksel örtme dizisi tabanlı test etme yaklaşımlarında ve önerilen yaklaşımdaki 2-way birleşim-test ikilisinin sayısını göstermektedir. Daha yüksek güçlü örtme dizisi kullanmak, maskeleme etkilerinin istenmeyen sonuçlarını azaltsa da, kesin bir şekilde tamamen problemi çözemez.

Tablo5'te ki son sütun, geleneksel yaklaşımlarla maskelenen ikililere bizim yaklaşımımızca elde edilen maskelenen ikililerin oranını 1 eksi gösterir. Örneğin, yaklaşımımız, geleneksel 3-way yaklaşımdan %26

iterasyon	test edilen CFGS	Test çalışmaları	Kapsanan satırlar	Kapsanan dallar	Eşsiz Errs	Eşsiz Test-errs	3'lü maskelem	Test etme zamanı	Analiz zamanı	Toplam zaman
1	72	18425	216612	106506	119	1377	6670639	1774	1	1775
2	108	44957	223500	127900	125	1438	3365540	3909	6	3915
3	230	51626	223719	128001	125	1440	2915475	6183	128	6311
4	315	52171	223725	128010	125	1440	2797589	6869	152	7017

Tablo 4: t=3 ile önerilen yaklaşım

daha az maskelenen ikililere sahiptir. Geleneksel 4-way yaklaşımla kıyaslandığında bizimkisi %34 daha fazla maskelenen ikililere sahipti. Aynı zamanda, hâlbuki yaklaşımımız daha fazla satır ve dallanmayı kapsadı ve yaklaşık %35 daha az bir zaman aldı.

Daha fazla araştırmak için, yaklaşımımızı t=3'lük daha yüksek bir değer ile çalıştırmayı denedik. Fakat hala 2-way birleşim-test ikililerini gözlemlemeye devam etmekteydik. Biz her iki yaklaşımın aslında aynı gerçekleştirdiğini gözlemledik. (Bizim yaklaşımımız için maskelenen ikililer 133.636, geleneksel 4-way yaklaşımı için 133.977)

Bu sonuçlar, bizim yaklaşımımız tarafından sağlanan çeşitli maliyet/yarar ödünleşimlerini ilerideki çalışmalara önemli olacağını ileri sürmektedir. Eğer yaklaşımımızı daha fazla iterasyonla çalıştırmaya izin verirsek, daha fazla zaman alabilir, fakat maskeleme etkisini azaltabilir. Bunun yerine, daha yüksek güçlü örtme dizisi kullanmayla başlamak, maskelemede daha küçük bir düşüş sağlayabilir, fakat daha kısa bir süre içerisinde çalışabilir.

T=3 ile Yaklaşımı Değerlendirme

Son olarak, t'nin daha büyük bir değeri ile yaklaşımımızı değerlendirmek için, t=3 ile çalışmayı tekrar gerçekleştirdik ve 3-way birleşim-test ikililerini izledik. Tablo 4 sonuçlarımızı göstermektedir. 15.144.193 geçerli 3-way birleşim-test ikililerinin %44'ünün geleneksel 3-way test etmede maskelendiği ortaya çıktı.(örneğin sürecin ilk turu)

Yaklaşımımız 4 iterasyon aldı. Bu, t=2 olduğu 10 tane iterasyondan daha azdır. Bunun sebebi, bizim her iterasyonda çalıştırdığımız daha geniş veri kümelerinin daha hızlı bir şekilde kısıtları keşfetmemize izin vermesiydi. Her iterasyonda, sürecin maskelenen 3-way birleşim-test ikililerinin sayısını azalttığını gözlemledik. Kurulum hatalarına neden olan seçenek birleşimlerinin tümü ve test atlamalarına neden olan birleşimlerin %61'i doğru bir şekilde belirlenmiştir. Limit değeri olarak 0.6 kullandığımızda, kısıtların %96'sı doğru bir şekilde tanımlanmıştır.

Dördüncü iterasyon ile yaklaşımımız, 1. iterayona ilişkin %58'lik 3-way birleşim-test ikililerinin sayısını azaltmıştır. Kaynak satır kapsama(source line coverage) ve dallanma kapsama(branch coverage) sırasıyla sadece %3 ve %2'lik gelişme gösterdi. Dahası, %5 daha fazla eşsiz hataları ve %7 daha fazla eşsiz test-hata ikililerini tanımladık. 4-way geleneksel test etme ile kıyaslandığında, önerilen yaklaşım %6'lık maskelenen 3-way birleşim-test ikililerinin sayısını düşürdü.(Tablo 6)

Özetle, hedef uygulama, onun test takımı ve seçilen yapılandırma modeli için, yaklaşımımızın tam olarak maskeleme etkilerini tanımlayabildiğini ve tüm kapsamı(coverage) iyileştiren yeni yapılandırmalar üretebildiğini gözlemledik. Özellikle, yaklaşımımız ilerledikçe, herzaman önemli ölçüde kapsamı(coverage) arttırdı. Bilhassa, çoğu mevcut yaklaşımların yaptığı gibi ilk iterasyondan sonra dursaydık, etkileşimlerin önemli miktarı test edilmeden kalıcaktı.

Aynı zamanda, yaklaşımımız maliyete maruz kalır ve alternatif yaklaşımlara karşı kıyaslanmalıdır. Örneğin yeterince daha yüksek güçlü örtme dizisi kullanma bir dereceye kadar maskelemeyi azaltır fakat daha fazla maliyeti düzeltir.

5.Sonuç

Örtme dizileri gibi birleşimsel etkileşim test etme yaklaşımları için temel gerekçe, t veya daha az seçeneklerin ayarlarından kaynaklanan tüm sistem davranışlarını çalıştırabilmesidir. Gerçekte, çoğu böyle davranışların, başarısızlıklardan kaynaklanan maskeleme etkileri yüzünden test edilemediğini tahmin etmekteyiz.

Maskeleme etkilerini göstermek için, geribesleme güdümlü birleşimsel test etme yaklaşımı geliştirdik. Örtme dizilerini istatistiksel olarak hesaplama yerine,

bizim etkileşimli kapsam(coverage) hedeflerimizi daha iyi karşılamak için maskeleme etkilerini tanımlamaya ve onları örtme dizisi yapılandırma sürecinden silmeye yönelik döngülü bir şekilde hesaplamaktayız. Sürecin her iterasyonunda, test durumlarını yürütürüz, sonuçları analiz ederiz, olası seçenek-ilişkili nedenleri tanımlayarak potansiyel maskeleme etkilerini belirleriz ve daha sonra önceden maskelenen etkileşimleri kapsarken, olası başarısızlık nedenlerinden kaçınan yeni örtme dizileri üretiriz. Süreç, testin hiç seçenek ilişkili bir neden ile başarılı veya başarısız olmadığı veya birleşimin başarısızlık nedeni olarak işaretlendiği en azından bir yapılandırmada her t-way seçenek ayarlaması birleşimi mevcut olana kadar bütün testler için dönmektedir.

Daha sonra, 2 deneysel çalışmayı yöneterek bu yeni süreci değerlendirdik. Bu çalışmalar temel uygulamalar olarak MySQL ve GCC olmak üzere 2 tane geniş açık kaynaklı yazılım sistemi kullandık. Önerilen yaklaşımın herzaman geleneksel t-way örtme dizisi ile kıyaslandığında maskelenen t-way birleşim-test ikilisinin sayısını önemli bir şekilde düşürdüğünü gözlemledik. t=2 olduğunda ikililerin sayısı %66'ya düşmüştür ve t=3 olduğunda ise %58 olmuştur. Her iki durumda da, tüm geçerli niyet edilen birleşim-test ikililerinden sadece %19'u maskeli kalmıştı.

Dahası, verilen bir t için, önerilen yaklaşım genelde daha yüksek güçlü geleneksel örtü dizisiyle kıyaslandığında maskelenen t-way birleşim-test ikililerinin sayısını azaltmada daha iyi olduğunu gözlemledik. t=2 ile önerilen yaklaşım, geleneksel 3-way test etmeyi kullanma ile kıyaslandığında, %26 ile 2-way birleşim-test ikililerinin sayısını azalttı. t=3 olduğunda yaklaşım daha geniş 4-way test etme ile kıyaslandığında sadece %6 ile maskelenen 3-way birleşim-test ikililerini azalttı. İleriki çalışmalarda gerekli olacak çeşitli maliyet/kar ödünleşimlerini görürüz.

Araştırmanın bu satırının umut verici olduğunu düşünmekteyiz. Bizim sürdürme niyetinde olduğumuz açık bir konu, daha pratik ayarlarda maskeleme etkilerinin yaygınlığını ölçmektir. Ayrıca alternatif makine öğrenmesi yaklaşımlarını ve yapılandırma ilişkili olası başarısızlık nedenlerini tanımlamak için eniyilemeyi inceleyeceğiz. Diğer bir ilginç hedef ise yapılandırma seçenekleri arasında otomatik bir şekilde kontrol bağımlılıklarını tanımlayan bir yaklaşım üzerinde çalışmaktır. Bu bizim hatalı çalışmalara ek olarak başarılı test çalışmalarını kullanmamızı gerektirecektir.

6.Onaylar

Bu araştırma 7. Avrupa Birliği Çerçeve Programı içerisinde Marie Curie International Reintegration Grant (FP7-PEOPLE-IRG-2008), TÜBİTAK (109E182) ve US NSF (CCF-0811284) tarafından desteklenmiştir.

7.Referanslar

[1] Advanced Combinatorial Testing System (ACTS), 2010.
[2] R.Brownline, J.Prowse and M.S. Phadke Robust testing of AT&T PMX/StarMAIL using OATS. AT&T Technical Journal, 71(3):41-7, 1992.
[3] R. C. Bryce and C. J. Colbourn. Prioritized interaction testing for pair-wise coverage with seeding and constraints. Journal of Information and Software Technology, 48(10):960-970, 2006.
[4] R. C. Bryce, A. Rajan, and M. P. Heimdahl. Interaction testing in model-based development: E ffect on model-coverage. In Asia Pacic Software Engineering Conference, pages 259-268, 2006.
[5] K. Burr and W. Young. Combinatorial test techniques: Table-based automation, test generation and code coverage. In Proc. of the Int'l Conf. on SW Testing Analysis & Review, 1998.
[6] A. Calvagna and A. Gargantini. A logic-based approach to combinatorial testing with constraints. In Tests and Proofs, Lecture Notes in Computer Science, 4966, pages 66-83, 2008.
[7] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: an approach to testing based on combinatorial design. IEEE Transactions on Software Engineering, 23(7):437-44, 1997.
[8] M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing interaction test suites for highly-congurable systems in the presence of constraints: A greedy approach. IEEE Transactions on Software Engineering, 34(5):633-650, 2008.
[9] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In Proc. of the Int'l Conf. on SW Eng., (ICSE), pages 285-294, 1999.
[10] S. Fouch e, M. B. Cohen, and A. Porter. Incremental covering array failure characterization in large configuration spaces. In ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis, pages 177-188, New York, NY, USA, 2009. ACM.
[11] D. Kuhn, D. R. Wallace, and A. M. Gallo. Sw fault interactions and implications for software testing. IEEE Trans. on SW Eng., 30(6):418-421, 2004.
[12] T. M. Mitchell. Machine Learning. McGraw Hill, 1997.
[13] MySQL, 2006. http://www.mysql.com.
[14] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. Communications of the ACM, 31:678-686, 1988.
[15] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: An empirical study of sampling and prioritization. In International Symposium on Software Testing and Analysis, pages 75-85, July 2008.
[16] X. Qu, M. B. Cohen, and K. M. Woolf. Combinatorial interaction regression testing: A study of test case generation nd prioritization. In International Conference on Software Maintenance (ICSM), pages 255-264, October 2007.
[17] I. H. Witten and E. Frank. Data Mining: Practical

Machine Learning Tools and Techniques with Java Implementations. Morgan Kaufmann, 1999.
[18] C. Yilmaz, M. B. Cohen, and A. Porter. Covering arrays for ecient fault characterization in complex configuration spaces. IEEE Transactions on Software Engineering, 31(1):20-34, Jan 2006.
[19] X. Yuan, M. Cohen, and A. M. Memon. Covering array sampling of input event sequences for automated GUI testing. In International Conference on Automated Software Engineering, pages 405-408, 2007.