DİNAMİK SEMBOLİK İŞLETME YÖNTEMİ İLE MATLAB'DA TEST VERİSİ ÜRETİMİ

TEST DATA GENERATION IN MATLAB BY USING DYNAMIC SYMBOLIC EXECUTION

HALİL İBRAHİM BALCI

DR. ÖĞR. ÜYESİ AYÇA TARHAN

Tez Danışmanı

Hacettepe Üniversitesi

Lisansüstü Eğitim-Öğretim ve Sınav Yönetmeliğinin

Bilgisayar Mühendisliği Anabilim Dalı için Öngördüğü

YÜKSEK LİSANS TEZİ olarak hazırlanmıştır.

2019

HALİL İBRAHİM BALCI'nın hazırladığı **"Dinamik Sembolik İşletme Yöntemi ile MATLAB'da Test Verisi Üretimi"** adlı bu çalışma aşağıdaki jüri tarafından **BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI**'nda **YÜKSEK LİSANS TEZİ** olarak kabul edilmiştir.

Doçent Lale ÖZKAHYA

Başkan .....

Doktor Öğretim Üyesi Ayça TARHAN

Danışman .....

Doktor Öğretim Üyesi Engin DEMİR

Üye .....

Doktor Öğretim Üyesi Barbaros YET

Üye .....

Doktor Öğretim Üyesi Ebru GÖKALP

Üye .....

Bu tez Hacettepe Üniversitesi Fen Bilimleri Enstitüsü tarafından **YÜKSEK LİSANS TEZİ** olarak ..... / ..... /..... onaylanmıştır.

Prof. Dr. Menemşe GÜMÜŞDERELİOĞLU

Fen Bilimleri Enstitüsü Müdürü

Eşim ve aileme...

ETİK

Hacettepe Üniversitesi Fen Bilimleri Enstitüsü, tez yazım kurallarına uygun olarak hazırladığım bu tez çalışmasında,

- tez içindeki bütün bilgi ve belgeleri akademik kurallar çerçevesinde elde ettiğimi,
- görsel, işitsel ve yazılı tüm bilgi ve sonuçları bilimsel ahlak kurallarına uygun olarak sunduğumu,
- başkalarının eserlerinden yararlanılması durumunda ilgili eserlere bilimsel normlara uygun olarak atıfta bulunduğumu,
- atıfta bulunduğum eserlerin tümünü kaynak olarak gösterdiğimi,
- kullanılan verilerde herhangi bir tahrifat yapmadığımı,
- ve bu tezin herhangi bir bölümünü bu üniversite veya başka bir üniversitede başka bir tez çalışması olarak sunmadığımı

beyan ederim.

30 / 09 / 2019

Halil İbrahim BALCI

YAYIMLAMA VE FİKRİ MÜLKİYET HAKLARI BEYANI

Enstitü tarafından onaylanan lisansüstü tezimin/raporumun tamamını veya herhangi bir kısmını, basılı (kağıt) ve elektronik formatta arşivleme ve aşağıda verilen koşullarla kullanıma açma iznini Hacettepe Üniversitesine verdiğimi bildiririm. Bu izinle Üniversiteye verilen kullanım hakları dışındaki tüm fikri mülkiyet haklarım bende kalacak, tezimin tamamının ya da bir bölümünün gelecekteki çalışmalarda (makale, kitap, lisans ve patent vb.) kullanım hakları bana ait olacaktır.

Tezin kendi orijinal çalışmam olduğunu, başkalarının haklarını ihlal etmediğimi ve tezimin tek yetkili sahibi olduğumu beyan ve taahhüt ederim. Tezimde yer alan telif hakkı bulunan ve sahiplerinden yazılı izin alınarak kullanması zorunlu metinlerin yazılı izin alarak kullandığımı ve istenildiğinde suretlerini Üniversiteye teslim etmeyi taahhüt ederim.

Yükseköğretim Kurulu tarafından yayınlanan ***"Lisansüstü Tezlerin Elektronik Ortamda Toplanması, Düzenlenmesi ve Erişime Açılmasına İlişkin Yönerge"*** kapsamında tezim aşağıda belirtilen koşullar haricince YÖK Ulusal Tez Merkezi/H.Ü. Kütüphaneleri Açık Erişim Sisteminde erişime açılır.

☐ Enstitü / Fakülte yönetim kurulu kararı ile tezimin erişime açılması mezuniyet tarihimden itibaren 2 yıl ertelenmiştir.

☐ Enstitü / Fakülte yönetim kurulu gerekçeli kararı ile tezimin erişime açılması mezuniyet tarihimden itibaren .... ay ertelenmiştir.

☐ Tezim ile ilgili gizlilik kararı verilmiştir.

30./09./2019

(İmza)

ÖĞRENCİNİN ADI-SOYADI

Halil İbrahim BALCI

ÖZET

DİNAMİK SEMBOLİK İŞLETME YÖNTEMİ İLE MATLAB'DA TEST VERİSİ ÜRETİMİ

Halil İbrahim BALCI

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Tez Danışmanı: Dr. Öğr. Üyesi Ayça TARHAN

Eş Danışman: Doç. Dr. Aysu BETİN CAN

Eylül 2019, 44 sayfa

Yazılım Test Sürecinde kullanılacak test verilerinin yüksek kapsama (İng. coverage) oranına sahip olması ve uygulanabilir miktarda olması tercih edilir. Dinamik Sembolik İşletme (İng. Dynamic Symbolic Execution) yöntemi bu iki yararı bir arada sağlayabilmektedir. Bu tez kapsamında MATLAB'da yer alan yazılımlar üzerinde Dinamik Sembolik İşletme yöntemi kullanılarak test verisi üretimi yapan bir araç geliştirilmiştir. Bu sayede test verimliliğinin artırılması hedeflenmiştir. Araç ile uygun sayıda test verisi, açık kaynak bir test aracıyla aynı sonuçları verecek şekilde üretilebilmektedir. Ayrıca Dinamik Sembolik İşletme yöntemi sayesinde test verilerinin yüksek kapsama oranına sahip olması sağlanarak test verimliliğinin arttığı gözlemlenmiştir.

Anahtar Kelimeler: Yazılım Test, Test Verisi Üretme, Sembolik İşletme, Test Verimliliği, Yazılım Test Aracı, Dinamik Sembolik İşletme

ABSTRACT

TEST DATA GENERATION IN MATLAB BY USING DYNAMIC SYMBOLIC EXECUTION

Halil İbrahim BALCI

Master of Science, Department of Computer Engineering

Supervisior: Asst. Prof. Dr. Ayça TARHAN

Co- Supervisor: Assoc. Prof. Dr. Aysu BETİN CAN

September 2019, 44 pages

In Software Testing process, it is preferred that test data has a high coverage ratio and is applicable amount. Dynamic Symbolic Execution (DSE) method can provide these two benefits together. Within the scope of this thesis, a tool is developed to generate test data using Dynamic Symbolic Execution method on software in MATLAB. The tool can generate appropriate number of test data as an open source DSE test data generation tool. In addition, the effectiveness of testing will be increased by ensuring the high coverage of the test data through dynamic symbolic execution method.

Keywords: Test Data Generation, Symbolic Execution, Test Efficiency, Software Testing, Software Testing Tool

TEŞEKKÜR

Gerçekleştirmiş olduğum tez çalışmasının, başlangıcından bitimine kadar her aşamasında katkı ve görüşleriyle beni yönlendiren ve destekleyen, bana ayırdıkları kıymetli zaman ve sağladıkları tüm yardımlar için değerli hocalarım Sayın Dr. Öğr. Üyesi Ayça TARHAN'a ve Sayın Doç. Dr. Aysu BETİN CAN'a, bugünlere gelmemde en büyük payın sahibi olan, hayatımın her anında hep yanımda olan, beni her zaman destekleyen, varlıkları ile bana güç veren canım eşim ve aileme,

Sonsuz Teşekkürler.

Halil İbrahim BALCI

Eylül 2019, Ankara

İÇİNDEKİLER

ŞEKİLLER DİZİNİ

ÇİZELGELER DİZİNİ

SİMGELER VE KISALTMALAR

AS	Araştırma Soruları
DSE	Dinamik Sembolik İşletme (İng. Dynamic Symbolic Execution)
PC	Yol Kısıtları (İng. Path Constraints)
SLR	Sistematik Literatür Taraması (İng. Systemetic Literature Review)

1. GİRİŞ

Yazılım test, yazılım kalitesini sağlamada önemli bir faktördür. Ancak, yazılım test süreci maliyetlidir ve yoğun işgücü gerektirir, bazı durumlarda yazılım geliştirme sürecinin %50'si yazılım testine adanmıştır [1]. Verimli yazılım test etkinliği, yazılım kalitesini artırır ve yazılım geliştirme sürecini kısaltır. Yazılım test faaliyetinin etkinliğini belirleyen en önemli faktörlerden biri, uygun test verilerini sağlamaktır. El ile (İng. manual) test verilerinin üretilmesi süreci zaman alıcı, maliyetli olabilmekte ve sonucunda hatalı veriler üretilebilmektedir [2]. Bu durum, test verilerinin otomatik olarak üretilmesini avantajlı kılmaktadır.

Literatürde, otomatik test verisi üretimi için birçok teknik önerilmiştir [3]. Bunlardan bazıları rastgele test etme (İng. random testing), arama tabanlı test etme (İng. search based testing), Sembolik İşletme ve Dinamik Sembolik İşletmedir. Rastgele test etme genellikle düşük kapsamlı test verileri üreterek gereksiz testler oluşturur. Arama tabanlı test etme, test verisi üretimi sırasında yerel en iyileştirme (İng. local-optima) noktalarına takılır. Pratik kullanımda, Sembolik İşletme yöntemi ise çözülemeyen kısıtlarda tıkanmaktadır. Dinamik Sembolik İşletme yöntemi ise her üç metodun bahsedilen sıkıntılarına çözüm sunmuştur. Bu yöntemle üretilen test verileri, her bir test verisi farklı bir kod parçasını test edecek şekilde üretilir. Böylece uygun sayıda test verisi ile yüksek kapsama oranı elde edilerek, uygun test verisi oluşturulur ve test etkililiği artırılmış olur.

Bu tez kapsamında MATLAB üzerindeki yazılımların test verilerini Dinamik Sembolik İşletme yöntemi kullanarak otomatik oluşturabilen bir aracın geliştirilmesi amaçlanmıştır. Böylece elde edilen test verisi ile yazılım test sürecinin daha kısa sürmesi ve daha verimli olması beklenmektedir.

Çalışmaya başlamadan önce konu üzerinde benzer bir araç veya çalışma olup olmadığı araştırılmıştır. Literatürde MATLAB üzerinde Dinamik Sembolik İşletme (DSE) yöntemi ile test verisi üretimi yapan herhangi bir çalışma veya araca rastlanmamıştır. Bu sebeple bu çalışma, alanında bir ilktir. Çalışmanın tasarımına dair fikir edinmek adına, tüm programlama dilleri için Dinamik Sembolik İşletme yöntemini kullanarak test verisi üretimi yapan çalışmalara dair sistematik literatür taraması (İng. systematic literature review) yapılmıştır. Konuya uygun olarak bulunan 261 çalışma içerisinden 39 makale seçilmiştir. Bu 39 makale incelenerek test verisi üretim sürecine dair belirlenen araştırma sorularına cevaplar bulunmuştur. Bu cevaplar ışığında DSE için bir kavramsal model (İng. conceptual model) oluşturulmuş ve araç, bu model göz önüne alınarak geliştirilmeye başlanmıştır.

Tez çalışması toplam 6 bölümden oluşmaktadır. Birinci bölümde konu hakkında genel giriş yapılmıştır, tezin geri kalanı aşağıdaki şekilde düzenlenmiştir:

2. Bölümde, çalışmada kullanılan yöntemlerden, bu yöntemlerin olası sorunlarından ve bu sorunlara dair çözüm önerilerinden bahsedilmiştir.

3. Bölümde, çalışma kapsamında yapılan sistematik literatür taraması hakkında detaylı bilgi verilmiştir. Ayrıca makalelerden elde edilen bilgiler ışığında oluşturulan kavramsal model sunulmuştur.

4. Bölümde, dinamik işletme yönteminin araç için nasıl gerçeklendiğinden ve aracın şu anki kısıtlarından bahsedilmiştir.

5. Bölümde, yöntemin uygulanması ile elde edilen zaman kazancı ve uygulamanın etkililiği anlatılmıştır.

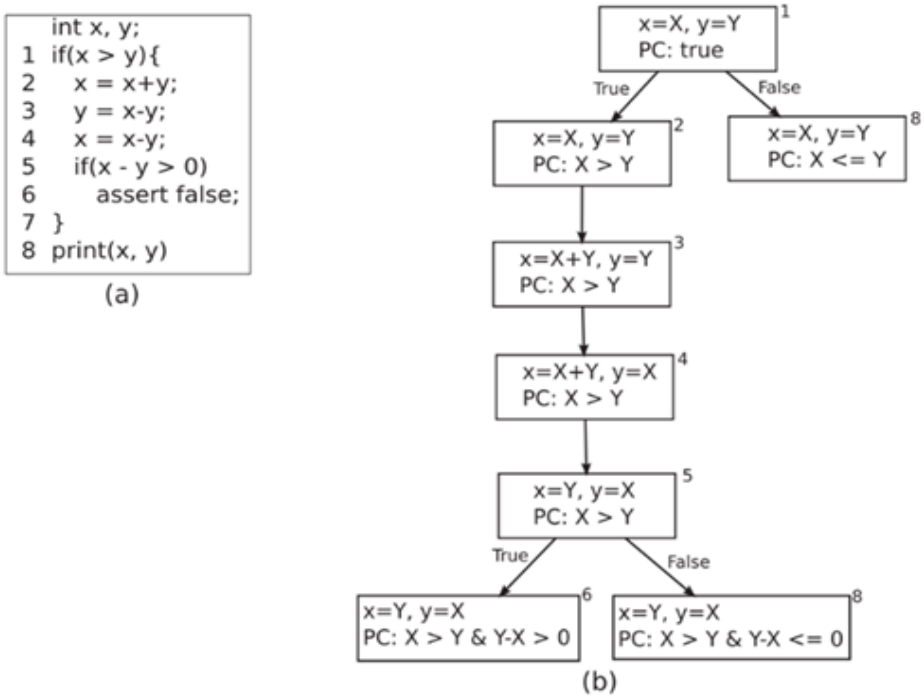
6. Bölümde ise uygulamaların sonuçları ve gelecek çalışmalara ait planlardan bahsedilmiştir.

2. ÖN BİLGİ

2.1. Sembolik İşletme

Sembolik İşletme [4], bir programın kodunu, otomatik olarak test verileri oluşturmak için analiz eden bir program analiz tekniğidir. Sembolik İşletme yönteminde, program girdileri olarak somut değerler yerine sembolik değerler kullanılır ve program değişkenlerinin değerleri, sembolik değerler ile gösterilir. Analiz sırasında herhangi bir noktada; değişkenlerin sembolik değerleri, o noktaya ulaşmak için gerekli yol kısıtı (İng. path constraint - PC) ve program sayacı (İng. program counter) değerleri tutulur. Program sayacı işlenecek bir sonraki ifadeyi (İng. statement) göstermektedir. Yol kısıtı belirli bir noktaya ulaşmak için gerekli bir formüldür; ilk değer olarak doğru (İng. true) kabul edilir ve her dalda (İng. branch) güncellenir. Her atama (İng. assignment) işleminde değişkenler sembolik değerler ile güncellenir. Analiz sonucunda Şekil 2.1'de görüldüğü gibi kod parçasına ait tüm yolların (İng. path) yer aldığı bir ağaç yapısı elde edilir. Elde edilen ağaç yapısındaki yollara göre maksimum kapsamayı (İng. coverage) sağlayacak girdiler, yol kısıtları çözülerek belirlenir.

Örnek olarak Şekil 2.1(a)'daki kod parçasını ele alabiliriz. Bu kod parçasında eğer x'in ilk değeri y'den büyükse, x ve y değerleri yer değiştirmektedir. Şekil 2.1(b) Sembolik İşletme işlemi sonrasında oluşan ağaç yapısını göstermektedir. Bu ağaç yapısında düğümler programın durumunu (İng. program state) ve kenarlar durumlar arasındaki geçişleri gösterir. Düğümlerin sağ üst köşesinde bulunan numaralar program sayacını gösterir. Program başlatılmadan önce x ve y değerlerine sembolik olarak sırasıyla X ve Y değerleri verilir. Yol kısıtları, 1. ve 5. satırdaki koşullar işletildikten sonra güncellenir. Ayrıca geçilen yol üzerinde x ve y değerlerine ait atamalar yapıldığı zaman, bu değerlerin sembolik değerleri güncellenir.



Şekil 2.1 (a) İki tamsayı (İng. integer) değerin yer değiştirmesini içeren kod parçası, (b) kod parçasının analizine ait Sembolik İşletme ağaç yapısı [1].

Çizelge 2.1'deki tablo, kod parçasındaki üç yolun kısıtlarını ve eğer varsa bu kısıtların çözümünü vermektedir. Örnek olarak programın (1,2,3,4,5,8) yoluna ait yol kısıtı X > Y & Y – X <= 0'dır. X = 2 ve Y = 1 değerleri ile bu kısıt çözülür. Yazılım bu girdilerle işletildiğinde belirtilen yoldan geçildiği görülmektedir. Programın (1,2,3,4,5,6) yoluna ait kısıt ise X > Y & Y – X > 0'dır. Bu kısıta ait bir çözüm yoktur. Bu yol ulaşılmaz (İng. infeasible) bir yoldur [5].

Çizelge 2.1 Farklı yollara karşılık gelen program girdileri ve yol kısıtları [1]

Yol	**Yol Kısıtı (PC)**	**Program Girdisi**
1,8	X<=Y	X=1, Y=1
1,2,3,4,5,8	X>Y & Y-X<=0	X=2, Y=1
1,2,3,4,5,6	X>Y & Y-X>0	Bulunamaz

Sembolik İşletme yöntemiyle test verisi üretimi yapan çalışma örnekleri aşağıda verilmiştir:

- Santelices ve diğerlerinin yaptığı çalışma [8], geliştirilen bir yazılımın iki sürümü arasında programın davranışlarındaki farklılığı ortaya çıkarmak için test verileri üretmektedir.

- Majumdar ve diğerlerinin yaptığı çalışma [9], sağlamlık testi için programın çıktısında önemli farklılıklara neden olabilecek test verilerini ortaya çıkarmaktadır.

- Qi ve diğerlerinin yaptığı çalışma [10] ise, yazılımın başarısız olmasına neden olabilecek girdilere benzer test verilerini üreterek hatanın yerini belirleyebilmektedir.

2.2. Dinamik Sembolik İşletme

Dinamik Sembolik İşletme, diğer adıyla Concolic İşletme (İng. Concolic Execution), Sembolik İşletme yönteminde karşılaşılan karmaşık veya çözülemeyen yol kısıtı problemine çözüm olarak kullanılan bir yöntemdir [6]. Sabit Değerlerle İşletme (İng. Conrete Execution) ve Sembolik İşletme tekniklerinin birleşimi olarak geliştirilen bu teknik, adını da bu iki yöntemden almaktadır (İng. Concolic = CONCrete + symbOLIC).

Concolic İşletme yönteminde program, başlangıçta rastgele belirlenen girdiler ile çalıştırılır ve paralelinde Sembolik İşletme de devam ettirilir. Kodun işletilmesi sırasında geçilen tüm koşullara ait mantıksal ifadeler toplanarak işletme sonunda, ilgili yol için gerekli olan yol kısıtı ve girdi değerleri tespit edilmiş olur.

Diğer yollara girilebilmesi için gerekli girdi değerlerini elde etmek amacıyla, yol kısıtı içerisinde seçilen bir mantıksal ifadenin tersi alınarak süreç tekrarlanır. Böylece işletilen yolların farklılaşması sağlanmış olur. Bu süreç, belirlenen test kriteri sağlanana kadar devam ettirilir. Bu yöntemin Sembolik İşletme yöntemine göre avantajı, çözülemeyecek karmaşıklıktaki koşulların sembolik değerler yerine gerçek değerler kullanılarak çözüme kavuşturulabilir olmasıdır. Örnek olarak Şekil 2.2'deki kod parçasını alabiliriz. Bu kod ifadesini ilk olarak rastgele

x=3, y=5 değerleri ile çalıştırırsak z değeri sabit olarak 63, sembolik olarak ise x^3+3*x^2+9 değerini alır ve program (1,2,3,4,5,8,9) yolunu takip eder. Elde edilen yol kısıtının tersini alırsak $x^3+3*x^2+9=y$ kısıtını elde ederiz ve bu kısıt, kısıt çözücü (İng. constraint solver) tarafından çözülemez. Bu durumda Concolic İşletme yöntemi devreye girer ve koşullardaki sabit değerlerin tutulması sayesinde y=63 ve x=3 değerleri ile program işletilir [7]. Böylece çözülemeyen yol kısıtının olduğu yola girilmiş olur. Bu çözüm yöntemi, Concolic İşletme ve Sembolik İşletme arasındaki en büyük farktır.

```
1  void test_me(int x,int y){
2          z = x*x*x + 3*x*x + 9;
3          if(z != y){
4             printf(y);
5          } else {
6             printf(x);
7              abort();
8          }
9  }
```

Şekil 2.2 Örnek Kod Parçası

Concolic İşletme yöntemi, sembolik değerler yerine gerçek değerler kullanarak karmaşık koşulların çözülememesi problemine bir çözüm olarak günümüzde, Sembolik İşletme yöntemine kıyasla daha tercih edilir bir yöntem haline gelmiştir.

Concolic İşletme yöntemini dezavantajların biri yol farklılaşmasıdır (İng. path divergence) [57]. Yol farklılaşması problemi kütüphane çağrılarında (İng. library calls) ve olağandışı durumlarda (İng. exceptions) olabilmektedir [58]. Bir diğer dezavantaj ise yol patlaması (İng. path explosion) problemidir. Bu problem program içerisindeki yol sayısının programın sayısına bağlı olarak üstel olarak artmasıdır.

3. İLİŞKİLİ ÇALIŞMALAR

Benzer çalışmaları araştırmak için sistematik literatür taraması (İng. Systematic Literature Review – SLR) yapılmış ve belirlenen anahtar kelimeler ile toplam 261 makaleye ulaşılmıştır. Literatürdeki hiçbir çalışmada Concolic İşletme yöntemi kullanarak MATLAB üzerinde test verisi üretimi yapan bir çalışmaya rastlanmamıştır. Bu anlamda bizim çalışmamız bu alanda bir ilktir.

261 makaleden belirlenen kriterlere göre elenerek seçilen 39 makale üzerinden araştırma soruları yanıtlanmıştır. Bu makaleler Concolic İşletme yöntemini kullanarak test verisi üretimi yapan çalışmalardır. Bu çalışmalarda aşağıdaki 5 araştırma sorusu (AS) yanıtlanarak Concolic İşletme yöntemi ile test verisi üretim sürecinin nasıl işlediği daha anlaşılır hale gelmiştir.

- AS-1: Concolic İşletme haricinde test yöntemi kullanılmış mıdır? Kullanıldı ise nedir?

- AS-2: Concolic İşletme yöntemi hangi programlama dili için işletilmiştir?

- AS-3: Concolic İşletme yöntemi kaynak kodun mu makine kodunun mu üzerinde işletilmiştir?

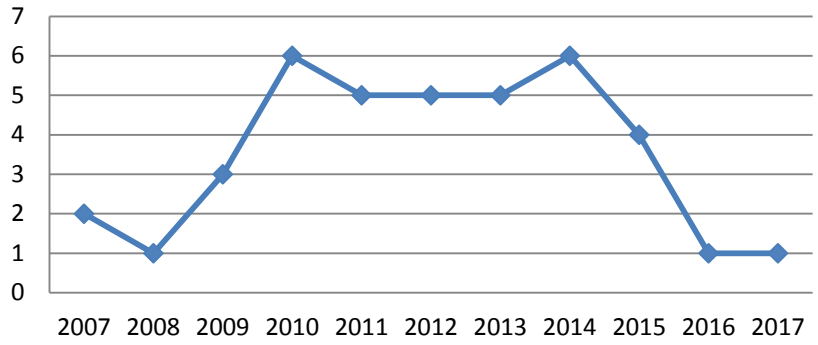
- AS-4: Concolic İşletme yöntemi için bir araç kullanılmış mıdır? Kullanıldıysa nedir?

- AS-5: Concolic İşletme için hangi kısıt çözücü kullanılmıştır?

3.1. Sistematik Literatür Taraması Sonuçları

Bu bölümde 39 makale incelenerek elde edilen araştırma soruları cevapları sunulacaktır.

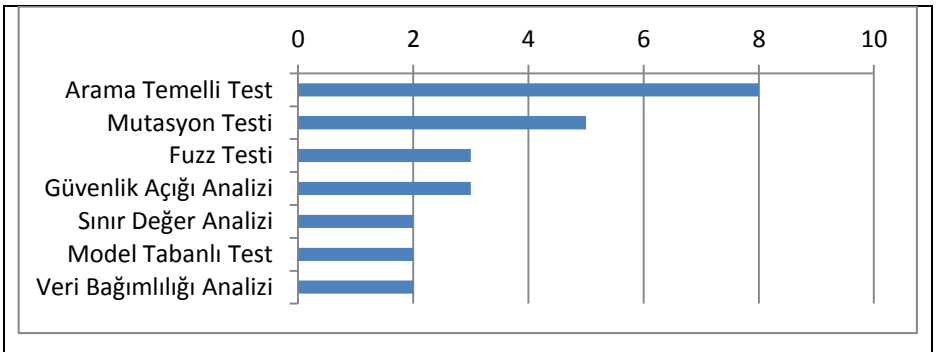
Makalelerin yıllara göre dağılımı Şekil 3.1'de verilmiştir. 2009'dan sonra konunun ilgi uyandırdığı ve 2015'ten sonra bu ilginin kaybedildiği görülmektedir.



Şekil 3.1 Makalelerin yıllara göre dağılımı

3.1.1. Concolic İşletme harici test yöntemi kullanılmış mıdır? Kullanıldı ise nedir?

Bu bölümde, seçilmiş makalelerdeki test verilerini oluşturmak için Dinamik Sembolik İşletme ile birlikte kullanılan test yöntemleri incelenmiştir.



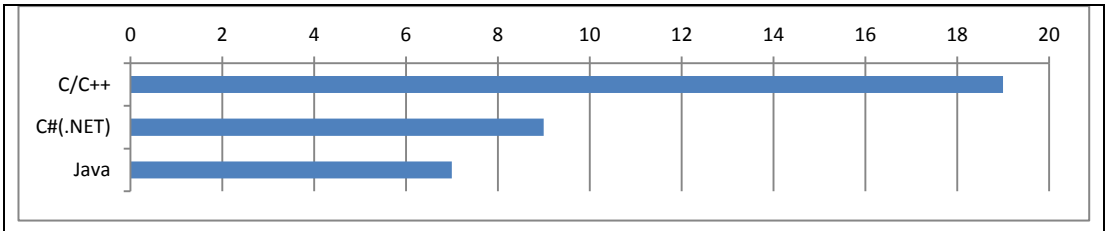
- Arama Tabanlı Test:[22][24][29][31][32][43][46][29]
- Mutasyon Testi: [22][23][24][40][52]
- Fuzz Testi: [13][14][15]
- Güvenlik Açığı Analizi: [13][14][15]
- Sınır Değer Analizi: [29][52]
- Model Tabanlı Test: [33][35]
- Veri Bağımlılığı Analizi: [37][51]

Şekil 3.2 Test yöntemlerinin dağılımı

Şekil 3.2'de birden fazla makalede kullanılan test yöntemleri verilmiştir. Arama tabanlı test (İng. search based testing) 8 makalede DSE ile test verilerinin oluşturulmasında kullanılan en popüler test yöntemi olmuştur. Mutasyon testi (İng. mutation testing) 5 makalede ve Fuzz testi 3 makalede kullanılmıştır. Güvenlik açığı analizi (İng. vulnerability analysis) 3 makalede DSE'ye yardımcı olmuştur. İki makalede, sınır değer analizi (İng. boundary value analysis) ile test verileri üretimi geliştirilmiştir. İki makalede model tabanlı test (İng. model based testing) yöntemleri kullanılmıştır. Veri bağımlılığı analiz (İng. data dependency analysis) metodu 2 makalede DSE'ye yardımcı olmuştur.

3.1.2. Concolic İşletme yöntemi hangi programlama dili için işletilmiştir?

Birden fazla makalede kullanılan programlama dillerinin sıklığı Şekil 3.3'de gösterilmiştir. C / C++, 19 makalede test verisi üretimi için kullanılarak en popüler programlama dili olmuştur. C# 9 makalede kullanılmıştır. Yedi makalede Java programlama dili için test verileri üretilmiştir. Makalelerde kullanılan diğer programlama dilleri WS-CDL [9], SQL [11] ve Pig Latin'dir [26].



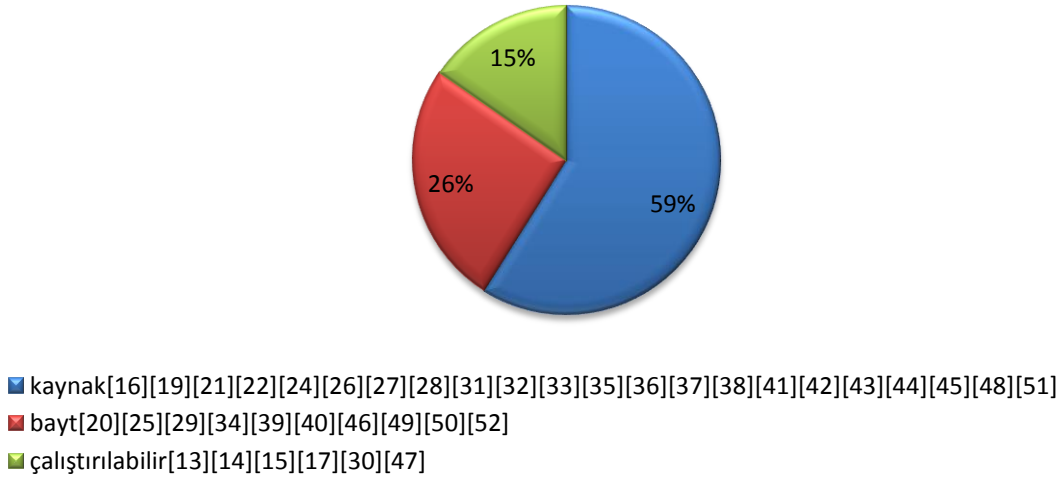
- C/C++:[13][14][15][16][17][22][26][28][30][33][37][38][42][43][44][45][47][48][51]
- C#(.NET): [20][25][29][34][39][40][46][49][52]
- Java: [23][24][27][31][32][35][50]

Şekil 3.3 Programlama dillerinin dağılımı

3.1.3. Concolic İşletme yöntemi kaynak kodun mu makine kodunun mu üzerinde işletilmiştir?

Test verileri kaynak, bayt (İng. byte) veya çalıştırılabilir kod üzerinden DSE yöntemi uygulanarak oluşturulabilir. Şekil 3.4 bunların kullanım yüzdesini göstermektedir. Kaynak kodu 23 makalede, bayt kodu 10 makalede ve

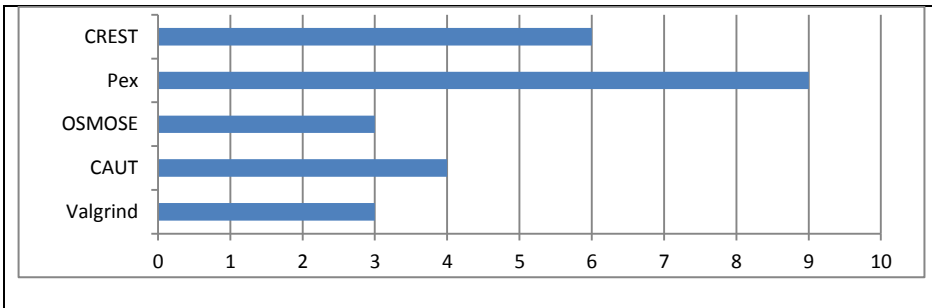
çalıştırılabilir kod 6 makalede, Dinamik Sembolik İşletme yöntemine girdi olarak kullanılmıştır.



Şekil 3.4 Kaynak, bayt ve çalıştırılabilir kod kullanım yüzdeleri

3.1.4. Concolic İşletme yöntemi için bir araç kullanılmış mıdır? Kullanıldıysa nedir?

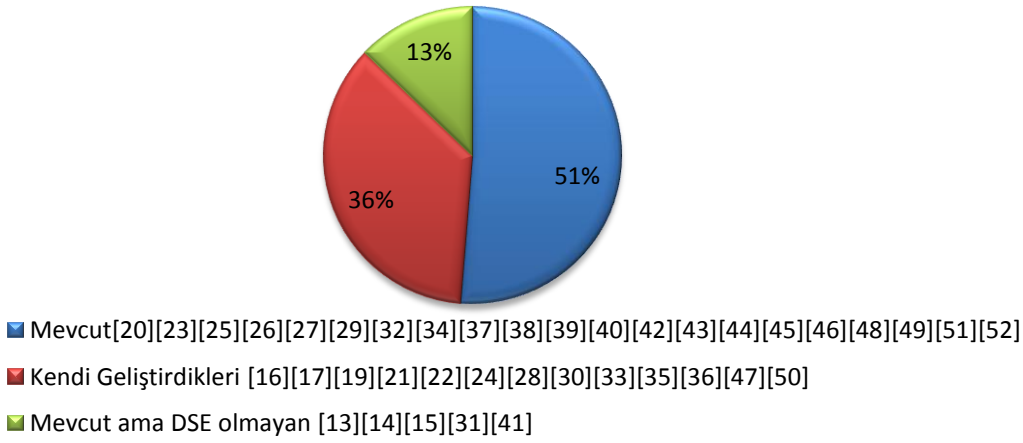
Bu bölümde test verilerinin üretiminde kullanılan araçlar aktarılmaktadır. Şekil 3.5'te birden fazla makalede kullanılan araçların sıklığı verilmiştir. Pex, 9 makalede kullanılarak en çok kullanılan araç olmuştur. Crest, 6 makalede ve CAUT, 4 makalede test verisinin oluşturulmasını sağlamıştır. Valgrind üzerindeki Fuzzgrind uzantısı ve OSMOSE 3'er makalede kullanılmıştır.



- **CREST:** [37][38][42][43][48][51]
- **Pex:** [20][25][29][34][39][40][46][49][52]
- **OSMOSE:** [17][30][47]
- **CAUT:** [16][28][33][44]
- **Valgrind:** [13][14][15]

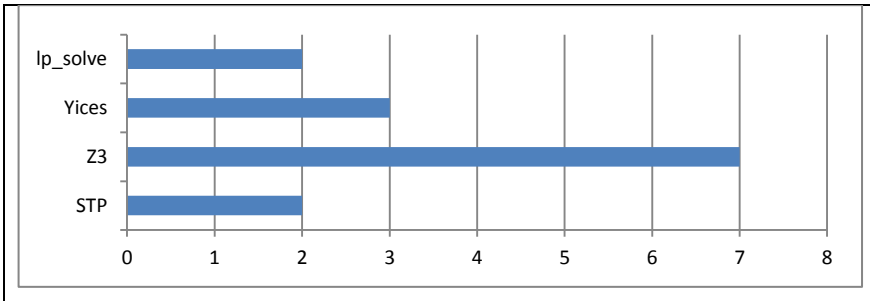
Şekil 3.5 Araç kullanım sayıları

Kullanılan diğer araçlar SHOM [22], JFuzz [23], CUTE [26], JCute [27], EXSYST [31], Pathfinder [32], CATG [35], SEDGE [36], SAFE [41], PathCrawler [45] ve Collider'dır [50].



Şekil 3.6 Araç tiplerine göre kullanım yüzdeleri

Ayrıca, araçlar üç kategoriye ayrılmıştır: Mevcut DSE aracı, kendi geliştirdikleri DSE aracı ve mevcut ama DSE olmayan araç. Araç tiplerinin makale sayısına göre dağılımı Şekil 3.6'da verilmiştir. Mevcut DSE araçları 20 makalede kullanılırken, yazarlar 14 makalede kendi geliştirdikleri DSE araçlarını kullanmışlardır. Beş makalede ise DSE olmayan bir araç değiştirilerek DSE aracı olarak kullanılmıştır.



- **lp_solve:** [26][28]
- **Yices:** [22][38][50]
- **Z3:**[19][21][33][34][36][40][41]
- **STP:** [13][14]

Şekil 3.7 Kısıt çözücülerin dağılımı

3.1.5. Hangi kısıt çözücü kullanılmıştır?

Makalelerde kullanılan kısıt çözücüler Şekil 3.7'de verilmiştir. Şekil, birden fazla makalede kullanılan kısıt çözücüleri içermektedir. Makalelerde kısıt çözücü bilgileri açıkça belirtilmişse uygun bir cevap olarak kabul edilmiştir. Yedi makalede kullanılan Z3 en yaygın kullanılan kısıt çözücü olmuştur. Yices 3 makalede, STP ve lp_solve 2'şer makalede kullanılmıştır. Kullanılan diğer kısıt çözücüler Choco [22], CVC4 [25] ve CORAL'dır [26].

3.2. Sistematik Literatür Taraması Sonuçlarının Değerlendirilmesi

SLR çalışmasına dâhil olan 39 makale yıllara göre analiz edildiğinde, Dinamik Sembolik İşletme yöntemiyle test verisi üretiminin 2009-2015 yılları arasında ilgi uyandırdığı ve 2015'ten sonra bu ilgiyi kaybettiği görülmüştür.

Diğer test yöntemlerine bakıldığında, DSE ile birlikte arama tabanlı test veya mutasyon test yöntemlerinin kullanılmasının ilgi uyandırdığı tespit edilmiştir. Bu teknikler test verisi üretimini geliştirmek için DSE ile birlikte kullanılmıştır. Böylece hibrit yöntemler oluşturulmuştur.

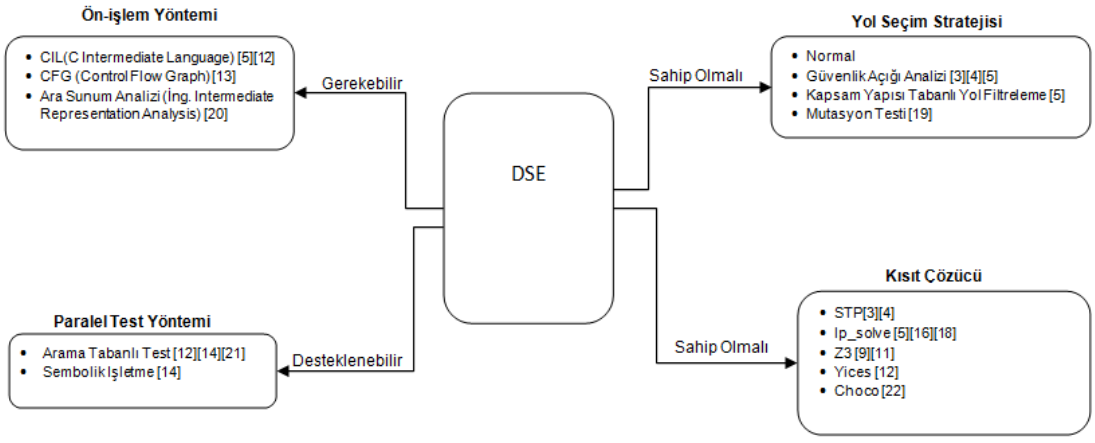
Makalelerin %59'unda kaynak kodu üzerinden DSE yöntemi kullanılmıştır. Bu ilgi, kaynak koduyla DSE sürecinin işletilmesinin ve izlenmesinin daha kolay olmasından kaynaklanmaktadır.

Makalelerin %51'i mevcut DSE araçlarını kullanmıştır. Dokuz makalede kullanılan PEX, araştırmalarda en yaygın kullanılan araçtır ve C # (.NET)'deki tüm DSE makaleleri PEX ile yapılmıştır. Bu durum, PEX'in .NET ortamı için baskın bir araç olduğunu göstermiştir. C / C++ dillerinde çalışılan 18 makalenin 9'unda hazır araçlar, diğer 9'unda ise yazarların kendi geliştirdikleri araçlar kullanılmıştır. Java'da da benzer bir durum geçerlidir. Java ortamında yapılan 7 makalenin 3'ünde hazır araçlar kullanılmış, diğer 4 makalede ise test verisi üretim araçları geliştirilmiştir. Daha önce de belirtildiği üzere MATLAB dili için test verisi üretimi yapan herhangi bir çalışmaya rastlanmamıştır.

Z3 kısıt çözücü 39 makalenin 7'sinde kullanılmıştır. Sadece 5 makale kendi kısıt çözücülerini geliştirmişlerdir. İki makalede ise farklı kısıt türlerine çözümler bulmak için farklı türdeki çözücüler kullanılmıştır.

3.3. Kavramsal Model

İncelenen makalelerdeki araştırma sorularına verilen cevaplar göz önüne alınarak temel kavramlar toplanmış ve Dinamik Sembolik İşletme yöntemi ile test verilerinin üretilmesi için kavramsal bir model oluşturulmuştur. Şekil 3.8'de gösterilen modele göre DSE, yol seçim stratejisine ve kısıt çözücüye sahip olmalıdır. Kodun yorumlanması gerekirse bir ön işleme yöntemi seçilebilir. Diğer isteğe bağlı bir yöntem ise farklı test kapsama oranlarını iyileştirmek için kullanılabilecek paralel test yöntemidir. Geliştirmekte olduğumuz araç bu kavramsal modelden yararlanarak ortaya çıkmıştır. Geliştirme detayları 4. Bölümde verilmiştir.

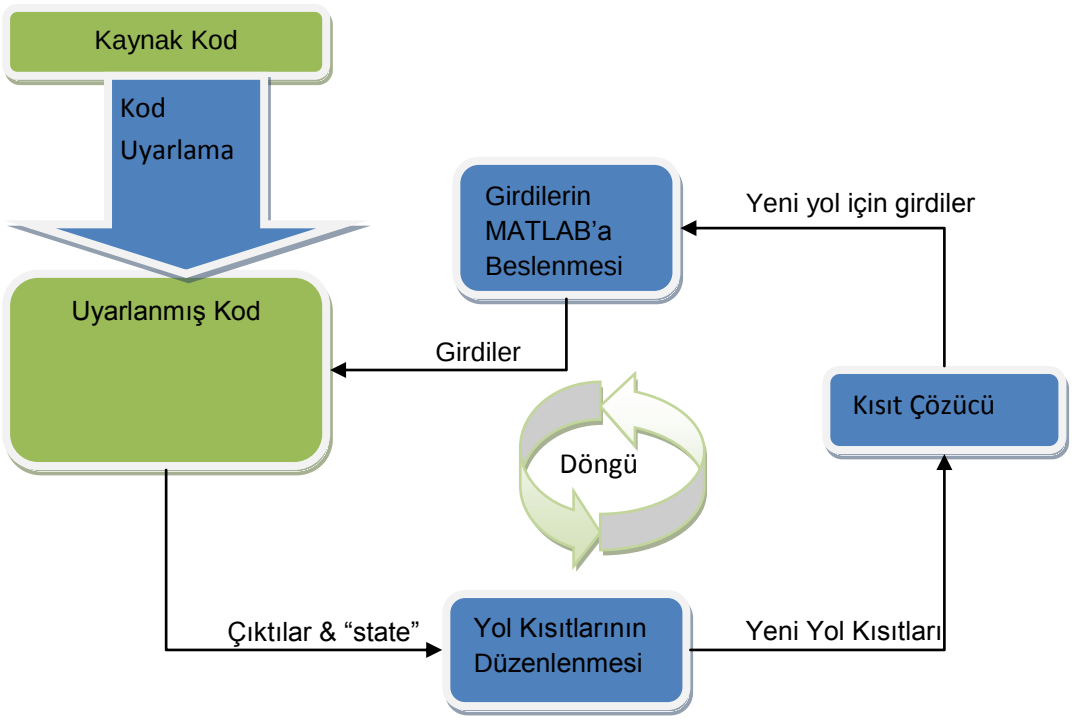


Şekil 3.8 DSE için Kavramsal Model

4. YÖNTEM

Bu çalışmada MATLAB kaynak kodları üzerinde Concolic İşletme yöntemi ile test verisi üreten bir aracın ortaya çıkarılması amaçlanmıştır. Araca DSEMA (ing. Dynamic Symbolic Execution for MATLAB) adı verilmiştir. DSEMA'nın iç işleyişi Şekil 4.1'de sunulmuştur. İşleyişten kısaca bahsetmek gerekirse, MATLAB kaynak kodu ilk olarak kod uyarlama (İng. instrumentation) ile düzenlenir. Daha sonra belirlenen girdilerle (ilk girdiler rastgele belirlenir) Java'da yazılan bir program içinden MATLAB çağrılarak, MATLAB'da daha önceden uyarlanmış kaynak kod çalıştırılır. Uyarlanan kaynak kodun çıktısı olarak çıktılar ve içerisinde yol kısıtları bulunan "state" veri yapısı Java'ya gönderilir. Java'da, yol kısıtlarından daha önce düzenlenmemiş bir yol kısıtı seçilerek tersi (İng. not) alınır. Bu işlemin amacı bir sonraki işletmede farklı bir yola girebilmektir. Ters çevrilme işleminden sonra yeni yol kısıtları elde edilir. Bu kısıtlar kısıt çözücüye gönderilerek yeni yol için girdiler belirlenir ve bu girdiler kullanılarak tekrar uyarlanmış MATLAB kodu çağrılır. Bu döngüye dal kapsama oranı önceden belirlenen orana ulaşılana kadar devam edilecektir. DSEMA'da test verisi üretim süreci bu şekildedir ve süreç kendiliğinden işlemektedir. Şekil 4.1'de yeşil kutular MATLAB ortamındaki, mavi kutular ise Java ortamındaki kodları ifade etmektedir. DSEMA'nın hedeflenen ortamdaki diğer araçlarla veya kütüphanelerle bütünleşik olarak çalıştırılmasını kolaylaştırmak için araç, Java dilinde hazırlanmıştır.

Araç tasarımı sırasında Şekil 3.8'deki kavramsal modelden yararlanılmıştır. Ön işleme için MATLAB'da herhangi bir dil bulunmadığından uyarlama işlemi ve ilgili fonksiyonlar, bu çalışmaya özel olarak geliştirilmiştir. Yol seçim stratejisi Dinamik Sembolik İşletme yöntemi için dal kapsamayı artıracak şekilde en derin ilk önce (İng. depth-first) olarak belirlenmiştir.



Şekil 4.1 DSEMA'nın İç İşleyiş Süreci

4.1. Yöntemin Gerçeklenmesi

Araç iki bölüme ayrılabilir. Bunların birincisi MATLAB üzerinde kaynak kodun uyarlanması ve uyarlamaya ait fonksiyonlar, ikinci bölüm ise uyarlanmış kodun JAVA üzerinden çağrılması, elde edilen kısıtlar kümesinin değiştirilip çözülerek yeni girdiler elde edilmesi ve bu yeni girdilerle uyarlanmış MATLAB kodunun çağrılmasıdır. İkinci bölüm bir döngü olarak işlemektedir.

4.1.1. Uyarlama

Yöntemin uygulanmasındaki ilk adım kaynak kodun uyarlanması (İng. Instrumentation) işlemidir. Bu işlem DSEMA'nın iç yapısındaki uyarlama modülü sayesinde otomatik olarak yapılmaktadır. Bu modül MATLAB kaynak kodunu metin olarak çözümleyerek (İng. parsing) her bir atamayı ve koşulu tanımlayabilmektedir ve ilgili atama veya koşulun sonrasına, DSEMA'nın MATLAB fonksiyonlarını çağırabilecek fonksiyon çağrıları (İng. function call)

yerleştirmektedir. Örnek bir kaynak kod ve bu kodun uyarlama işlemi yapıldıktan sonraki halleri EK 1'de verilmiştir. Uyarlama yapıldıktan sonra kaynak kod Java'ya "state" veri yapısını (İng. struct) döndürür. Uyarlanmış kaynak kod dinamik olarak işletilirken "state" yapısı her bir atama (İng. assignment) ve koşulun (İng. condition) ardına uyarlama sırasında eklenen fonksiyon çağrıları sayesinde, sürekli güncellenerek o işletim sırasında geçilen yola ait kısıtlara ve değişkenlerin aldığı sembolik ve sabit değerlere ilişkin bilgiyi tutar.

Uyarlama sırasında 11 farklı fonksiyon tipi eklenebilir. Bu fonksiyonların hepsinin amacı "state" yapısına bir önceki satırdaki bilgiyi aktarmaktır. DSEMA'nın çözümleyebileceği yapı tipleri ve ifadeler listesi Tablo 4-1'de verilmiştir. Uyarlama sonucunda eklenen fonksiyonlar alt başlıklarda açıklanmıştır.

4.1.1.1. "gncasgnc"

Bir değişkene sabit bir değer atandığında kullanılır. "state" yapısındaki sabit değerlerin tutulduğu haritada (İng. map) ilgili sabit değer güncellenir veya değer yer almıyorsa haritaya eklenir.

4.1.1.2. "gncasgns"

Bir değişkene sembolik bir değer atandığında kullanılır. Programın girdi değerleri sembolik değerlerdir. Program içerisinde eğer bir değere sembolik bir değer kullanılarak atama yapılırsa atanan değer de sembolik değer olur.

Bu fonksiyon, "state" yapısı içerisindeki ilgili sembolik değerler haritasındaki sembolik değeri, sembolik olarak günceller. Ayrıca, sabit değerler haritasında ilgili değeri, sabit değer ile günceller.

Örnek olarak girdileri "a" ve "b" olan bir programın, "a"=3 ve "b"=4 değeri ile çağrıldığını düşünelim. Program içerisinde "d=a+3" ifadesi yer aldığında "a" sembolik bir ifade olduğundan artık "d" de sembolik bir ifade olmuştur. Sembolik değerler haritasına "d" eklenir, değer olarak "a+3" atanır. Ayrıca sabit değerler haritasına da "d" eklenir, değer olarak "6" atanır (3+3).

Çizelge 4.1 DSEMA tarafından uyarlanabilen ifadelerin listesi

Tip	**Uyarlamadan Önce**	**Uyarlamadan Sonra**
"if" koşulu	if (x < a)	if (x < a) [state] = gncif (state, 'x < a', x, a, *Id1*);
"elseif" koşulu	elseif (x < b)	elseif (x < b) cndns = {'x < b', 'x < a'}; ids = {*Id2, Id1*}; cncvls = {x, b, x, a}; [state] = gncelseif (state, cndns, cncvls, ids, "*number of conditions*");
"else" koşulu	else	else cndns = {'x < a', 'x < b'}; ids = {*Id1, Id2*}; cncvls = {x, a, x, b}; [state] = gncelse (state, cndns, cncvls, ids, "*number of conditions*");
"end" sonlandırma ifadesi	end	end cndns = {'x < a', 'x < b'}; ids = {*Id1, Id2*}; cncvls = {x, a, x, b}; [state] = gncnotif (state, cndns, cncvls, ids, "*number of conditions*", "*level of depth*");
"for" koşulu	for i=1: c	for i=1: c [state] = gncfor (state, 'c >= 1', c, 1, "*level of depth*", "*id*");
"while" koşulu	while (y < d)	while (y < d) [state] = gncwhile (state, 'y < d', y, d, "*level of depth*", "*id*");
Sabit Değer Ataması (tamsayı, vektor, matris)	K = 3;	K = 3; [state] = gncasgnc (state, K, 'K');
	M1=[1 2; 3 4];	M1 = [1 2; 3 4]; [state] = gncasgnc (state, M1, 'M1');
Sembolik Atama (tamsayı, vektor, matris)	a = c;	a = c; [state] = gncasgns (state, a, 'a', 'c');
	M2 = [a b; c d];	M2 = [a b; c d]; [state] = gncasgns (state, M2, 'M2', '[a b ; c d]');
Temel Matematik İşlemleri (tamsayı)	C = x + y;	C = x + y; [state] = gncasgns (state, C, 'C', 'x + y');
Temel Matematik İşlemleri (vektor, matris)	M3 = M1 + M2;	M3 = M1 + M2; [state] = gncasgnsmatsum (state, M3, 'M3', M1, M2, 'M1', 'M2');
	M3 = M1 - M2;	M3 = M1 - M2; [state] = gncasgnsmatsub (state, M3, 'M3', M1, M2, 'M1', 'M2');
	M3 = M1 * M2;	M3 = M1 * M2; [state] = gncasgnsmatmul (state, M3, 'M3', M1, M2, 'M1', 'M2');

4.1.1.3. “gncif”

“if” koşulunun ardından kullanılır. Koşulun içerisindeki karşılaştırma işlemini “state” yapısı içerisindeki yol kısıtlarına ekler.

4.1.1.4. “gncelseif”

“elseif” koşulunun ardından kullanılır. Koşulun içerisindeki karşılaştırma işlemini “state” yapısı içerisindeki yol kısıtlarına ekler. Ayrıca bu “elseif”den önceki “if” koşuluna ait ifadenin de tersini (İng. not) alarak yol kısıtlarına ekler.

4.1.1.5. “gncelse”

“else” koşulunun ardından kullanılır. Koşulun içerisinde herhangi bir ifade bulunmadığından “else” ifadesinin öncesindeki “if” ve varsa “elseif” koşullarının içerisindeki ifadelerinin tersini alarak teker teker yol kısıtlarına ekler.

4.1.1.6. “gncnotif”

Koşulların bittiğini belirten “end” ifadesinden sonra kullanılır. Fonksiyon içerisinde, ilk olarak bir önceki satırdaki “end” ifadesi ile biten koşullara girilip girilmediği kontrol edilir. Bu kontrol “state” yapısı içerisindeki bir sayı değeri ile yapılır. Eğer bu sayı beklenen değerde ise yol kısıtlarına değer eklenmeden çıkılır. Eğer sayı beklenen değerde değilse koşullar içerisindeki tüm ifadelerin tersi alınarak birer birer yol kısıtlarına eklenir.

4.1.1.7. “gncfor”

“for” koşulunun ardından kullanılır. Koşulun içerisindeki değerin döngünün ilk değerine eşit veya büyük olduğu bilgisini “state” yapısı içerisindeki yol kısıtlarına ekler.

4.1.1.8. “gncwhile”

“while” koşulunun ardından kullanılır. Koşulun içerisindeki ifadeyi “state” yapısı içerisindeki yol kısıtlarına ekler.

4.1.1.9. “gncasgnsmatsum”

Matris (İng. matrix) toplama işlemlerinden sonra kullanılır. MATLAB’da matris toplama ile tamsayı (İng. integer) toplama arasında ifade olarak herhangi bir fark bulunmamaktadır. Bu nedenle, matris toplama işlemini ayırt edebilmek adına uyarlama modülü her bir değişkenin matris olup olmadığını takip etmektedir. Bu fonksiyon oluşan toplam matrisini eğer içerisinde sembolik değer var ise sembolik matris olarak sembolik değerler haritasına ekler.

Matris işlemlerinde her bir eleman ayrı ayrı ele alınmıştır. Bu elemanların sembolik veya sabit değer olma durumları incelenip sonuç matrisinin elemanlarının durumuna karar verilmiştir.

4.1.1.10. “gncasgnsmatsub”

Matris çıkarma işlemlerinden sonra kullanılır. Toplama ile benzer şekilde çıkarmanın da ayırt edilmesi gerekir. Bu fonksiyon oluşan sonuç matrisini eğer içerisinde sembolik değer var ise sembolik matris olarak sembolik değerler haritasına ekler.

4.1.1.11. “gncasgnsmatmul”

Matris çarpma işlemlerinden sonra kullanılır. Toplama ile benzer şekilde çarpmanın da ayırt edilmesi gerekir. Bu fonksiyon oluşan sonuç matrisini eğer içerisinde sembolik değer var ise sembolik matris olarak sembolik değerler haritasına ekler.

Bu 11 fonksiyonun haricinde, ilklendirme (İng. initiation) işlemleri için “gncall” fonksiyonu sadece bir kere programın başında kullanılır. Bu fonksiyon “state” veri yapısını oluşturur ve bu yapı içerisindeki ilgili haritalara Java’dan gelen girdilerin sembolik ve sabit değerlerini aktarır. Ayrıca çeşitli veri yapılarının ilklendirmesi de burada yapılır.

Koşul ifadelerinin sonrasında yer alan tüm ifadeler koşulların sağ ve sol sabit değerlerini (ing. concrete) “state” yapısı içerisine aktarır. Böylece koşulların çözümü sırasında çözülemeyecek bir durum (polinom denklemler, alt kütüphane çağrıları vb.) ile karşılaşıldığında sabit değerler kullanılabilecektir. Ayrıca koşul ifadelerin sonrasında yer alan fonksiyonların hepsinde koşullara verilen bir numara yazılır. Bu numara “state” yapısına aktarılır. Böylece tüm koşulların bir kimlik numarası olur ve bu numara ile takip edilir.

4.2. İşletilme

Bu bölümde Java üzerinden uyarlamış MATLAB kodunun nasıl çağrıldığı ve yol kısıtlarının nasıl toplanıp işlenerek yeni girdilerin oluşturulduğu aktarılacaktır. DSEMA Java üzerinden MatlabEngine kütüphanesi sayesinde, uyarlanmış MATLAB kodunu çağrılabilmekte ve çıktı olarak Java’ya dönen “state” yapısı içerisindeki yol kısıtları, yol kısıtlarının kimlik numaraları, kısıtlarının sağ ve sol taraflarındaki sabit değerler, değişkenlere ait sembolik ve sabit değerler incelenebilmektedir. Kısıtların kimlik numaralarından daha önce geçilip geçilmediği ve geçildiyse olumlu olarak mı olumsuz olarak mı geçildiği bir harita (İng. map) üzerinden belirlenir. Belirlenen duruma göre ve en derin- en önce (İng. depth-first) yöntemi ile daha önce olumsuzu alınmamış bir koşul seçilerek olumsuzu alınır ve yeni oluşan kısıt kümesi kısıt çözücüye gönderilir. Bu işlem sırasında birden fazla alt koşul içeren (AND ve OR koşulları) koşullar tek tek ele alınmalı ve sonuçlar birleştirilmelidir. DSEMA’da kısıt çözücü olarak Choco Solver [53] kullanılmıştır. Bazı durumlarda kısıt çözücü kısıt kümesine ait bir çözüm bulamaz. Bu durumda daha önce olumsuzu alınmamış diğer kısıtların olumsuzu alınır. Yine çözüm bulunamazsa kısıtların ikili kombinasyonları alınarak çözüm bulunana kadar bu işleme devam edilir. Eğer polinom bir denklem veya alt kütüphanelere ait bir çağrı nedeniyle çözüm bulunamıyorsa bu

durumda “state” içerisinden kısıtların sağ ve sol sabit değerleri değerlendirilerek yeni girdiler belirlenir. Belirlenen yeni girdilerle uyarlanmış MATLAB kodu tekrar çağrılır. Bu işlem belirlenen dal kapsama (İng. branch coverage) oranına ulaşılana kadar devam edilir.

4.3. Yöntemin Kısıtları

Araç şu anda MATLAB’da matris, vektör ve tamsayı atamalarını, indekslemelerini ve bu veri yapıları üzerine yapılan basit matematiksel işlemleri tanımlayabilmektedir. DSEMA, koşul çözücü olarak Choco Solver’ı kullanmaktadır. Bu nedenle Choco Solver’ın çözemediği string yapılarını doğal olarak DSEMA da çözümleyememektedir. Ayrıca “if”, “elseif”, “else”, “for” ve “while” koşullarını çözebilmektedir. DSEMA’nın MATLAB bölümünde tanımlı tüm atama veya koşullar birbirinden bağımsız ayrı fonksiyonlarda çözülmektedir. Bu nedenle eklenecek bir işlev diğer fonksiyonlardan bağımsız olarak kolayca eklenebilir. MATLAB’a özel fonksiyonların nasıl çözüleceğine, fonksiyonların ne kadar sık kullanıldığı ve programın içerisindeki yolları ne kadar etkilediği göz önüne alınarak karar verilecektir. Çözümleme işlemi için iki yol takip edilebilir. İlk yol Concolic İşletme sayesinde özel fonksiyonun içerisine girilmeden, bir önceki işletilmede tutulan sabit değerin atanmasıdır. Bu sayede özel fonksiyona girilmeden bu fonksiyonun olduğu koşula ait iki yola da girilebilmektedir. Bu işlev çözülemeyecek kadar karmaşık yol kısıtlarında ve alt kütüphanelere ait çağrılarda da uygulanacaktır. İkinci yol ise fonksiyonu çözümleyecek özel bir fonksiyonun yazılmasıdır. Daha önce belirtildiği gibi DSEMA’nın esnek yapısı sayesinde özel fonksiyonlar kolayca araca eklenebilmektedir.

5. YÖNTEMİN SINANMASI

Bu bölümde DSEMA'nın doğruluğunu ve etkililiğini ortaya koymak adına yapılan vaka çalışması (ing. case study) sunulmaktadır. Bu vaka çalışmasında Yin [54] tarafından önerilen gömülü ve çoklu vaka tasarımı (İng. embedded and multiple case design) bir rehber olarak alınmıştır.

5.1. Araştırma Yöntemi

5.1.1. Araştırma Soruları

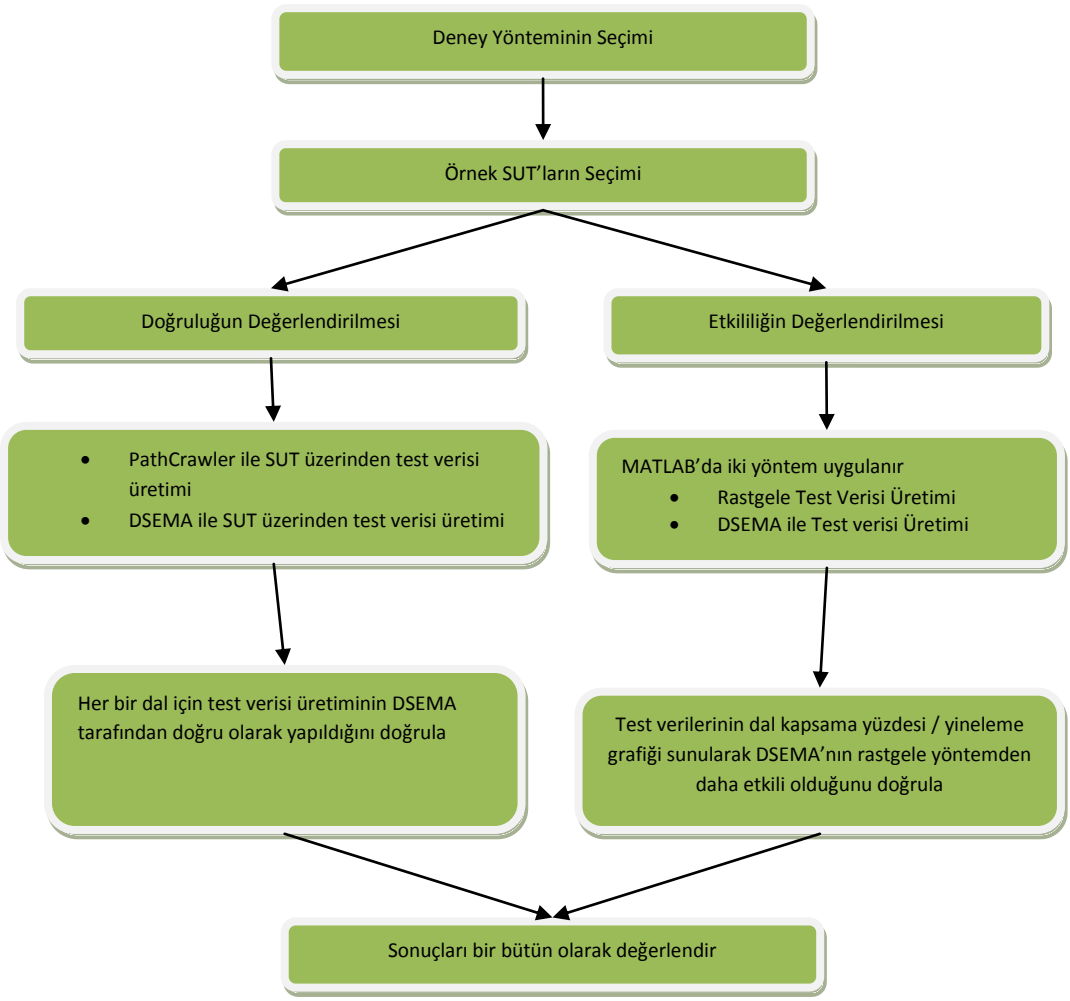
DSEMA'nın doğruluğunu ve etkililiğini ortaya koyabilmek adına aşağıdaki araştırma soruları (AS) tanımlanmıştır:

- AS1: DSEMA kullanılarak yapılan test verisi üretim süreci ne ölçüde doğru sonuç vermektedir?
- AS2: DSEMA kullanılarak yapılan test verisi üretim süreci ne ölçüde etkilidir?

5.1.2. Araştırma Süreci

Araştırma süreci Şekil 5.1.'deki akış diyagramı ile verilmiştir. Araştırma sürecinin başında, araştırma sorularını yanıtlamak için deneysel yöntem seçilmiştir. Daha sonra, bu yöntemlerin hangi örnek programlar ile gerçekleneceği belirlenmiştir. Test araçlarının doğrulanması için klasik bir program olarak kullanılan üçgen sınıflandırma programı [55], en büyük - en küçük tespiti ile üçgen sınıflandırma programı (Şekil 5.2) ve basit matris programı (Şekil 5.6) Test Edilen Yazılım (İng. Software Under Test – SUT) olarak seçilmiştir. AS1'e cevap verebilmek için, test verileri SUT'lardan PathCrawler [56] ve DSEMA araçlarıyla üretilmiş ve üretilen test verileri

karşılaştırılmıştır. PathCrawler DSE yöntemi ile test verisi üretimi yapan bir araçtır. Aracın sonuç raporlarında her bir dalın pozitif ve negatif yönü için üretilen test verisi ve test verilerin dal kapsama oranı sunulmaktadır. Bu durum PathCrawler'in ürettiği test verilerin DSEMA'nın ürettiği test verileri ile karşılaştırılmasını kolaylaştırmaktadır. SUT'ların C ve MATLAB versiyonları bu işlem için kullanılmıştır. AS2'ye yanıt vermek için, test verileri rastgele ve DSEMA aracıyla SUT'lar üzerinden üretilmiş ve sonuçlar karşılaştırılmıştır.



Şekil 5.1 Araştırma Süreci

5.2. Vaka Çalışması Sonuçları

5.2.1. AS 1 "DSEMA kullanılarak yapılan test verisi üretim süreci ne ölçüde doğru sonuç vermektedir?"

Bu bölümde, iki örnek program SUT olarak seçilmiştir. Bu SUT'lar üzerinden DSEMA ve PathCrawler araçları tarafından test verisi üretimi yapılmış ve sonuçlar karşılaştırılmıştır.

```
1  function [trityp]=ucgenori(a,b,c)
2   trityp = 0;
3    if (a < 0.0 || b < 0.0 || c < 0.0)
4      trityp=3;
5      return;
6    end
7    if (a + b <= c || b + c <= a || c + a <= b)
8        trityp=3;
9        return;
10   end
11   if (a == b)
12       trityp = trityp + 1;
13   end
14   if (a == c)
15       trityp = trityp + 1;
16   end
17   if (b == c)
18       trityp = trityp + 1;
19   end
20   if (trityp >= 2)
21       trityp = 2;
22   end
23   return;
24  end
```

Şekil 5.2 Üçgen Sınıflandırma Programı

Üçgen sınıflandırma, test verisi üretiminde referans olarak kullanılan örnek bir programdır [55]. DSEMA ve PathCrawler, test verileri üretimi için üçgen sınıflandırma programı ile çalıştırılmıştır. Doğal olarak DSEMA, üçgen sınıflandırma programının MATLAB versiyonunu, PathCrawler ise C versiyonunu işlemiştir.

Çizelge 5.1 Üçgen Sınıflandırma Programı için Üretilen Test Verileri

Dallanma Noktalarının Satır Numarası	Olumlu/ Olumsuz	DSEMA (a,b,c)	PathCrawler (a,b,c)
3	+	(-10,-10,1)	(-8,-9,7)
	-	(1,3,2)	(7,10,7)
7	+	(1,3,2)	(2,2,8)
	-	(1,1,1)	(7,10,7)
11	+	(1,1,1)	(7,7,10)
	-	(2,4,3)	(7,10,7)
14	+	(1,1,1)	(7,10,7)
	-	(2,2,1)	(7,7,10)
17	+	(1,1,1)	(6,6,6)
	-	(2,2,1)	(7,10,7)
20	+	(1,1,1)	(6,6,6)
	-	(2,2,1)	(7,10,7)

Üçgen sınıflandırma programının MATLAB versiyonu Şekil 5.2'de verilmiştir. Programda 6 dallanma noktası (İng. branch point) vardır. Bu noktalar 3, 7, 11, 14, 17 ve 20 numaralı satırlardadır. PathCrawler ve DSEMA tarafından her bir noktanın pozitif ve negatif dallarını girmek için oluşturulan test verileri Tablo 5-1'de verilmiştir. Üç girdi (a, b, c) [-10,10] aralığında tamsayı olarak seçilmiştir.

```
function [max,min,trityp]=ucgen2ori(a,b,c)
if(a >= b && a >= c)
    max=a;
elseif(b >= a && b >= c)
    max=b;
else
    max=c;
end
if(a <= b && a <= c)
    min=a;
elseif(b <= a && b <= c)
    min=b;
else
    min=c;
end
trityp = 0;
if (a <= 0 || b <= 0 || c <= 0)
    trityp=3;
    return;
end
if (a + b <= c || b + c <= a || c + a <= b)
    trityp=3;
    return;
end
if (a == b)
    trityp = trityp + 1;
end
if (a == c)
    trityp = trityp + 1;
end
if (b == c)
    trityp = trityp + 1;
end
if (trityp >= 2)
    trityp = 2;
end
return;
end
```

Şekil 5.3 En Büyük ve En Küçük Değer Tespiti ile Üçgen Sınıflandırma Programı

İkinci SUT en büyük - en küçük değer tespiti ile üçgen sınıflandırma programıdır. Bu program, üç girdiden oluşacak üçgenin sınıflandırılmasının yanı sıra, bu değerlerin en büyüğü ve en küçüğünü bulabilmektedir. İlk örneğe benzer şekilde DSEMA, programın MATLAB versiyonunda ve PathCrawler programın C versiyonu üzerinde test verileri oluşturmuştur. Girdi değerleri [-10,10] aralığında tamsayı olarak seçilmiştir. Sonuçlar Tablo 5-2'de verilmiştir. DSEMA'nın PathCrawler gibi 10 dallanma noktasında negatif ve pozitif dallar için test verileri oluşturduğu görülmektedir. Böylece, her iki program da en

büyük - en küçük değer tespiti ile üçgen sınıflandırma programı için %100 dal kapsama (İng. branch coverage) oranına ulaşabilmektedir.

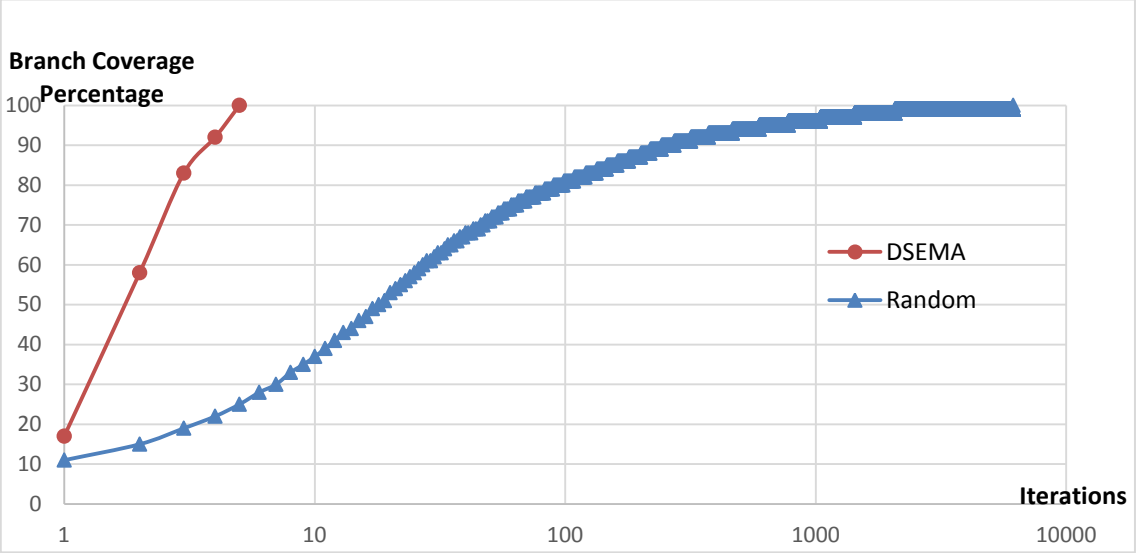
Çizelge 5.2 En Büyük-En Küçük Değer Tespiti ile Üçgen Sınıflandırma Programı için Üretilen Test Verileri

Dallanma Noktalarının Satır Numarası	Olumlu/ Olumsuz	DSEMA (a,b,c)	PathCrawler (a,b,c)
2	+	(1,1,-10)	(4,-10,-2)
	-	(1,3,2)	(-10,5,-2)
4	+	(1,3,2)	(-10,5,-2)
	-	(1,1,2)	(-8,2,7)
9	+	(1,3,2)	(-10,5,-2)
	-	(-9,-8,-10)	(4,-10,-2)
11	+	(1,-10,1)	(4,-10,-2)
	-	(-9,-8,-10)	(4,9,-2)
17	+	(-10,1,-10)	(-10,5,-2)
	-	(1,3,2)	(7,10,7)
21	+	(1,3,2)	(2,2,8)
	-	(2,3,2)	(7,10,7)
25	+	(1,1,1)	(7,7,2)
	-	(2,3,2)	(2,3,2)
28	+	(2,3,2)	(2,3,2)
	-	(2,4,3)	(7,7,10)
31	+	(1,1,1)	(3,3,3)
	-	(2,3,2)	(2,3,2)
34	+	(1,1,1)	(3,3,3)
	-	(2,2,1)	(7,7,10)

5.2.2. AR 2 “DSEMA kullanılarak yapılan test verisi üretim süreci ne ölçüde etkilidir?”

Bu bölümde, DSEMA'nın etkililiğini göstermek amacıyla üç farklı SUT için, DSEMA ve rastgele yöntem tarafından oluşturulan test verileri karşılaştırılmıştır.

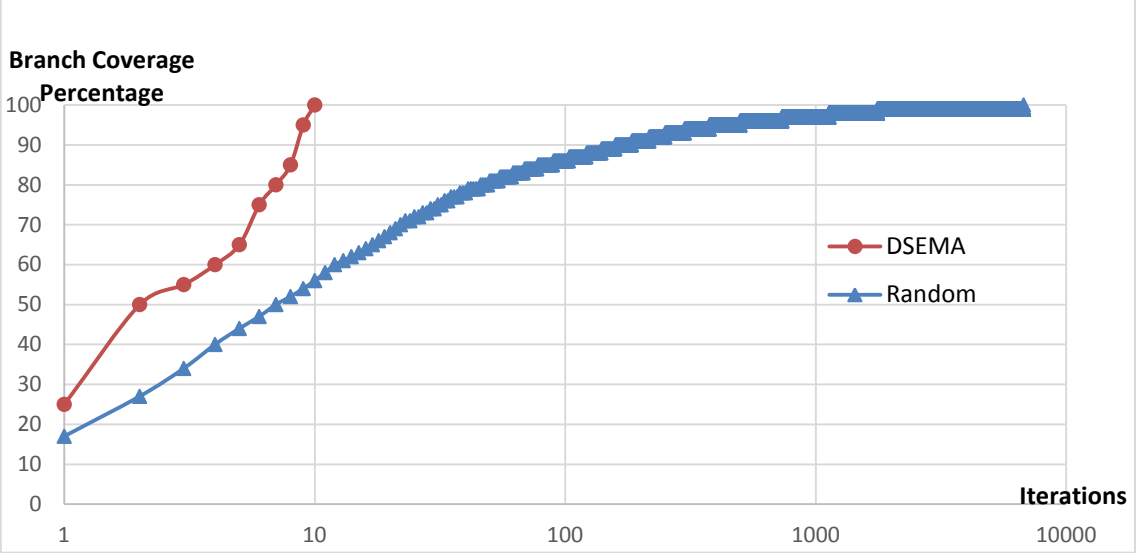
İlk SUT, Şekil 5.2'de verilen üçgen sınıflandırma programıdır. Üçgen sınıflandırma yazılımının girdileri rastgele yöntemle ve DSEMA ile üretilmiştir. Girdiler, [-10, 10] aralığında tamsayı olarak seçilmiştir. Rastgele üretimle test verileri 1000 kez üretilir ve bu denemelerin dal kapsama yüzdesi (İng. branch coverage percentage) değerlerinin ortalaması alınır. DSEMA ile rastgele yöntemle üretilmiş test verilerinine ait dal kapsama yüzdesi / yineleme grafiği Şekil 5.4'de verilmiştir. DSEMA, 5 yinelemede %100 dal kapsamayı sağlayacak test verileri üretmiştir. Rastgele üretilen test verileri, 6141 yinelemede %100 dal kapsamaya ulaşabilmiştir.



Şekil 5.4 Üçgen Sınıflandırma Programı için Dal Kapsama Yüzdesi / Yineleme Grafiği

İkinci SUT olarak Şekil 5.3’de verilen en büyük - en küçük değer tespiti ile üçgen sınıflandırma programı kullanılmıştır. İlk örneğe benzer şekilde bu örnek için de DSEMA ve rastgele yöntemle test verisi üretimi yapılmıştır. Üç girdi, [-10,10] arasından tamsayı değeri olarak seçilmiştir. Üretilen test verilerinin dal

kapsama yüzdesi / yineleme grafiği Şekil 5.5'de verilmiştir. DSEMA 10 yinelemede %100 dal kapsama oranına ulaşmıştır. Rastgele yöntem ise 6759 yinelemede bu yüzdeye ulaşabilmiştir.

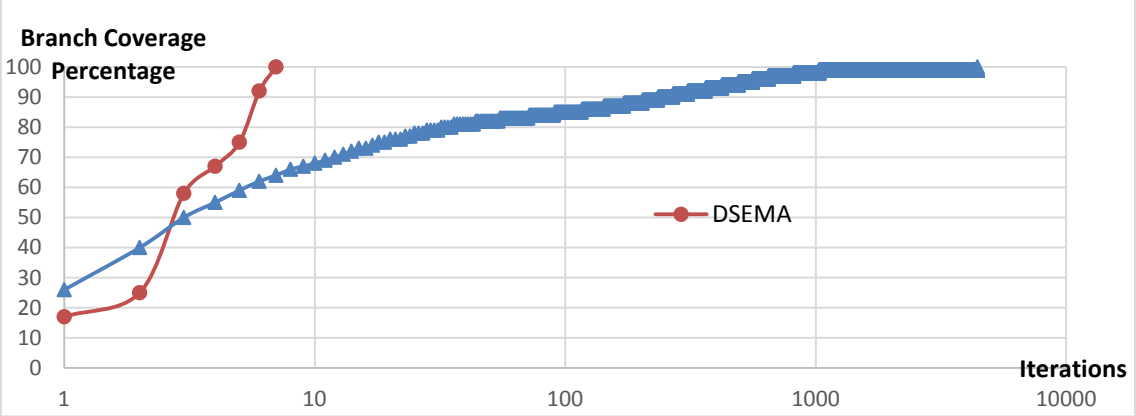


Şekil 5.5 En Büyük- En Küçük Değer Tespiti ile Üçgen Sınıflandırma Programının Dal Kapsama Yüzdesi / Yineleme Grafiği

Üçüncü SUT programı Şekil 5.6'da verilmiştir. Görülebileceği üzere bu program matris yapıları ve iç içe (İng. nested) koşullar içermektedir. DSEMA ve rastgele yöntemin ürettiği test verileri [-10,10] arasında tamsayı değerler alabilmektedir. Şekil 5.7'de üretilen test verilerinin dal kapsama yüzdesi / yineleme grafiği gösterilmiştir. Rastgele yöntemle test verisi üretimi 1000 defa yapılarak, 1000 denemedeki dal kapsama yüzdelerinin ortalaması alınmıştır. DSEMA 7 yinelemede %100 dal kapsama oranına ulaşırken rastgele yöntem, bu orana 4424 yinelemede ulaşabilmiştir.

```
function [count]=Matrix_target(a,b,c,d)
count=0;
allMat=[a b;c d];
ortomat1=[a b;b a];
ortomat2=[c d;d c];
totalallort1=allMat+ortomat1;
totalort12=ortomat1+ortomat2;
totalallort2=allMat+ortomat2;
suballort2=allMat-ortomat2;
if(suballort2(1,1)>ortomat1(1,2))
    if(totalallort1(2,1)<= allMat(2,2))
        if(suballort2(1,2)>a)
            count=count+1;
            if(totalort12(1,1)==ortamat2(2,1)
                count=count+3;
            end
        else
            count=count-2;
        end
    elseif(totalallort1(1,1)>ortomat2(2,2))
        count=count+2;
    else
        count=count-1;
    end
else
    count=count-1;
end
for i=0:totalallort2(1,2)
    count=count+1;
end
end
```

Şekil 5.6 Örnek Matris Programı



Şekil 5.7 Örnek Matris Programının Dal Kapsama Yüzdesi / Yineleme Grafiği

5.3. Sonuçların Değerlendirilmesi

5.3.1. AS 1 “DSEMA kullanılarak yapılan test verisi üretim süreci ne ölçüde doğru sonuç vermektedir?”

MATLAB programlama dilinde referans alınabilecek başka bir DSE test verisi üretim aracı bulunmamaktadır. Bu nedenle DSEMA ile elde edilen sonuçlar, C programlama dilinde DSE yöntemiyle test verisi üretimi yapan PathCrawler aracı ile elde edilen sonuçlarla karşılaştırılmıştır. Tablo 5-1 ve Tablo 5-2'den görülebileceği gibi, DSEMA tüm dal noktalarında pozitif ve negatif koşullar sağlamak için PathCrawler'a benzer dallara girebilecek test verileri üretmiştir. Bu durum, her iki aracın da örnek programlar için %100 dal kapsama sağladığı anlamına gelmektedir.

5.3.2. AS 2 “DSEMA kullanılarak yapılan test verisi üretim süreci ne ölçüde etkilidir?”

Aracın etkililiğini göstermek için, test verileri rastgele ve DSEMA ile üretilmiştir. Üretilen test verileri dal kapsama yüzdesi / yineleme olarak grafiklerle gösterilmiştir. DSEMA, 5 test verisi ile %100 dal kapsama oranına ulaşırken rastgele test verisi üretimi, ilk SUT için %100 dal kapsama oranını 6141 test verisi ile sağlayabilmiştir. Bu durum, DSEMA ile test verilerinin üretilmesinin, rastgele yöntemden %99.92 daha etkili olduğunu göstermektedir. Ayrıca ikinci numune için DSEMA, yineleme bakımından %99,85 daha etkili olmuştur. Üçüncü SUT etkililik oranı ise %99,84'tür. Bu üç örnek ile elde edilen değerlerden görülebileceği gibi dal kapsama yüzdesi / yineleme bakımından DSEMA, rastgele test verisi üretme yöntemine kıyasla en az %99,84 daha etkili olmuştur.

5.4. Geçerlilik Tehditleri

Bu çalışmada kullanılan dal kapsama oranının hesaplanması, yapısal geçerlilik (İng. construct validity) için bir tehdit olarak kabul edilebilir. Dal kapsama, test kapsamını değerlendirmek için genel olarak kullanılan bir program bileşenidir. Ek olarak, ölçüm doğruluğu PathCrawler ile kontrol edilmiştir.

Bu çalışmada, iç geçerliği (ing. internal validity) sağlamak için kullanılan tüm SUT'ların kaynak kodları paylaşılmıştır. PathCrawler aracı çevrimiçi olarak herkese açıktır. Rastgele test verisi oluşturma yönteminin detayları ve tüm sonuçlar paylaşılmıştır. Araştırma Sorularına cevap verilirken daha güçlü kanıtlar sağlamak için birden fazla SUT üzerinde deneyler yapılmıştır. Ayrıca rastgele yöntemle çalışılırken daha doğru ölçümler alabilmek için, deneyler 1000 kez tekrarlanıp sonuçların ortalaması alınmıştır.

Dış geçerlilik açısından, bu çalışmanın sonuçlarını genelleştirmek için daha büyük ve daha karmaşık örnekler üzerinde çalışılmalıdır. Bununla birlikte, bu çalışmada sunulan vaka çalışmaları, DSEMA'nın etkililiğini göstermek için bir temel oluşturmuştur.

Güvenilirliğin sağlanması için çalışmanın tüm detayları açıkça belirtilmiştir. Tekrarlanabilirlik için tehdit teşkil edebilecek bir faktör, DSEMA'nın kamuya açık olmamasıdır, çünkü ticari projeler yürüten bir sistem mühendisliği şirketine özel bir çözüm olarak geliştirilmiştir. Bununla birlikte bu tezde önerilen tasarımın ve gerçekleştirme detaylarının, bundan sonraki çalışmalarda başka araştırmacılara benzer çözümler üretmede ilham vermesi beklenebilir.

6. SONUÇ

Yazılımlar büyüdükçe yazılım test sürecinin önemi artmaktadır. Testlerin hızlı ve verimli bir şekilde yapılması, yazılımın hedef piyasaya sürülme süresinin kısaltılmasını ve proje maliyetlerinin azaltılmasını sağlar. Hızlı ve verimli bir test süreci için önemli etmenlerden biri yüksek kapsama oranına sahip ve makul sayıda test verisi ile testlerin yapılmasıdır. Bu özelliklere sahip test verisi üretimi için çeşitli test verisi üretim metotları araştırılmış ve DSE yönteminin bu özelliklere sahip test verisini üretebileceği görülmüştür.

Bu çalışma kapsamında DSE yöntemi ile test verisi üretimi yapan literatürdeki çalışmaları incelemek amaçlı olarak bir sistematik literatür taraması yapılmıştır. Sistematik literatür taramasında 39 makale üzerinden 5 araştırma sorusu cevaplanarak DSE ile test verisi üretimi konusunda literatürdeki eğilimler belirlenmiştir. Bu cevaplar üzerinden yola çıkılarak bir kavramsal model oluşturulmuştur. Bu çalışmada kavramsal model temel alınarak MATLAB kaynak kodu üzerinde DSE yöntemini kullanarak otomatik olarak test verisi üretimi yapan DSEMA aracı tasarlanmış ve gerçekleştirilmiştir.

DSEMA aracı temel olarak uyarlama ve işletilme olmak üzere iki ana bölümden oluşur. Uyarlama (İng. instrumentation) bölümünde SUT'un MATLAB kaynak kodu içerisine dinamik işletilme sırasında hangi yollardan geçildiğinin anlaşılması için fonksiyonlar eklenir. İşletilme bölümünde ise uyarlanmış kod koşturularak geçilen koşullara ait koşul seti çıkarılır. Koşul setinden bir koşul seçilerek tersi alınır ve yeni koşul seti çözülür. Böylece yeni girdilerle uyarlanmış kod tekrar koşturulur ve her seferinde farklı bir dal üzerinden geçilir. Bu döngü belirlenen bir test kriteri sağlanana kadar devam eder.

Otomatik olarak yapılan test verisi üretimi sayesinde yazılım test sürecinin süresinin azalması ve etkililiğinin artması beklenmektedir. Yapılan vaka

çalışması dâhilindeki deneylerle aracın doğru çalıştığı açık kaynak bir DSE aracı olan PathCrawler ile karşılaştırılarak güvence edilmiştir. Ayrıca rastgele test verisi üretimi ve DSEMA ile yapılan test verisi üretimi karşılaştırıldığında, DSEMA'nın %99,84 oranında daha az sayıda test verisi ile aynı kapsama oranını sağladığı örnek SUT'lar üzerinde kanıtlanmıştır.

Gelecek çalışmalarda DSEMA'ya yeni özelliklerin eklenmesi planlanmaktadır. Bu iyileştirmelerden biri, farklı test kriterlerine yönelik test verisi üretiminin yapılmasıdır. DSEMA'nın kullanılabilirliğini ve görselleştirilmesini artırmak amaçlı bir kullanıcı arayüzü eklenmesi ve bu arayüz üzerinde geçilen yolların ağaç yapısı ile gösterilmesi de iyileştirilecek hususlar arasındadır. Ayrıca DSEMA'nın esnek yapısı sayesinde araçta henüz tanımlanmamış ifadeler ihtiyaç duyulduğunda kolayca eklenebilecek, bu da DSEMA'nın zenginleşmesine yardımcı olacaktır.

KAYNAKLAR

[1] B. Beizer, Software Testing Techniques, 2nd ed.: Thomason Computer Press, 1990.

[2] Le Thi My Hanh, Nguyen Thanh, and Khuat Thanh Tung Binh. "Survey on Mutation-based Test Data Generation." International Journal of Electrical and Computer Engineering (IJECE) 5.5 (2015): 1164-1173

[3] Anand, Saswat, et al. "An orchestrated survey of methodologies for automated software test case generation." Journal of Systems and Software 86.8 (2013): 1978-2001

[4] King, J. C. (1975, April). A new approach to program testing. In ACM Sigplan Notices (Vol. 10, No. 6, pp. 228-233). ACM.

[5] Khurshid, S., Păsăreanu, C. S., & Visser, W. (2003, April). Generalized symbolic execution for model checking and testing. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems (pp. 553-568). Springer, Berlin, Heidelberg.

[6] Godefroid, P., Klarlund, N., & Sen, K. (2005, June). DART: directed automated random testing. In ACM Sigplan Notices(Vol. 40, No. 6, pp. 213-223). ACM.

[7] Sen, K., Marinov, D., & Agha, G. (2005, September). CUTE: a concolic unit testing engine for C. In ACM SIGSOFT Software Engineering Notes (Vol. 30, No. 5, pp. 263-272). ACM.

[8] Santelices, R., Chittimalli, P. K., Apiwattanapong, T., Orso, A., & Harrold, M. J. (2008, September). Test-suite augmentation for evolving software. In 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (pp. 218-227). IEEE.

[9] Majumdar, R., & Saha, I. (2009, December). Symbolic robustness analysis. In 2009 30th IEEE Real-Time Systems Symposium (pp. 355-363). IEEE.

[10] Qi, D., Roychoudhury, A., Liang, Z., & Vaswani, K. (2001). Darwin: An Approach for Debugging Evolving Programs. ACM Transactions on

Software Engineering and Methodology, 2(3), 1-0.

[11] Araki, L. Y., & Peres, L. M. (2018). A Systematic Review of Concolic Testing with Aplication of Test Criteria. In ICEIS (2) (pp. 121-132).

[12] Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W& Li, J. J. (2013). An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software, 86(8), 1978-2001.

[13] Mouzarani, M., Sadeghiyan, B., & Zolfaghari, M. (2017, November). Detecting injection vulnerabilities in executable codes with concolic execution. In 2017 8th IEEE International Conference on Software Engineering and Service Science (ICSESS) (pp. 50

[14] Mouzarani, M., Sadeghiyan, B., & Zolfaghari, M. (2016). Smart fuzzing method for detecting stack

[15] Mouzarani, M., Sadeghiyan, B., & Zolfaghari, M. (2015, November). A smart fuzzing method for detecting heap-based buffer overflow in executable codes. In 2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing (PRDC) (pp. 42-49). IEEE.

[16] Su, T., Pu, G., Fang, B., He, J., Yan, J., Jiang, S., & Zhao, J. (2014, June). Automated coverage-driven test data generation using dynamic symbolic execution. In 2014 Eighth International Conference on Software Security and Reliability (SERE) (pp. 98-107). IEEE.

[17] Bardin, S., Baufreton, P., Cornuet, N., Herrmann, P., & Labbé, S. (2013, July). Binary-level testing of embedded programs. In 2013 13th International Conference on Quality Software (pp. 11-20). IEEE.

[18] Sen, K. (2007, November). Concolic testing. In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (pp. 571-572). ACM.

[19] Zhou, L., Ping, J., Xiao, H., Wang, Z., Pu, G., & Ding, Z. (2010, November). Automatically testing web services choreography with assertions. In International Conference on Formal Engineering Methods (pp. 138-154). Springer, Berlin, Heidelberg.

[20] Christakis, M., Müller, P., & Wüstholz, V. (2014, September). Synthesizing parameterized unit tests to detect object invariant violations. In International Conference on Software Engineering and Formal Methods (pp. 65-80). Springer, Cham.

[21] Marcozzi, M., Vanhoof, W., & Hainaut, J. L. (2014, May). Towards testing of full-scale SQL applications using relational symbolic execution. In Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis (pp. 12-17). ACM.

[22] Harman, M., Jia, Y., & Langdon, W. B. (2011, September). Strong higher order mutation-based test data generation. In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering (pp. 212-222). ACM.

[23] Papadakis, M., & Malevris, N. (2010, November). Automatic mutation test case generation via dynamic symbolic execution. In 2010 IEEE 21st International Symposium on Software Reliability Engineering (pp. 121-130). IEEE.

[24] Papadakis, M., Malevris, N., & Kallia, M. (2010, May). Towards automating the generation of mutation tests. In Proceedings of the 5th Workshop on Automation of Software Test (pp. 111-118). ACM.

[25] Zhang, L., Xie, T., Zhang, L., Tillmann, N., De Halleux, J., & Mei, H. (2010, September). Test generation via dynamic symbolic execution for mutation testing. In 2010 IEEE International Conference on Software Maintenance (pp. 1-10). IEEE.

[26] Majumdar, R., & Xu, R. G. (2007, November). Directed test generation using symbolic grammars. In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (pp. 134-143). ACM.

[27] Emmi, M., Majumdar, R., & Sen, K. (2007, July). Dynamic test input generation for database applications. In Proceedings of the 2007 international symposium on Software testing and analysis (pp. 151-162). ACM.

[28] Wang, Z., Yu, X., Sun, T., Pu, G., Ding, Z., & Hu, J. (2009, July). Test data generation for derived types in c program. In 2009 Third IEEE International Symposium on Theoretical Aspects of Software Engineering (pp. 155-162). IEEE.

[29] Jamrozik, K., Fraser, G., Tillmann, N., & De Halleux, J. (2012, September). Augmented dynamic symbolic execution. In 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (pp. 254-257). IEEE.

[30] Bardin, S., & Herrmann, P. (2008, April). Structural testing of executables. In 2008 1st International Conference on Software Testing, Verification, and Validation (pp. 22-31). IEEE.

[31] Salvesen, K., Galeotti, J. P., Gross, F., Fraser, G., & Zeller, A. (2015, May). Using dynamic symbolic execution to generate inputs in search-based GUI testing. In Proceedings of the Eighth International Workshop on Search-Based Software Testing (pp. 32-35). IEEE Press.

[32] Malburg, J., & Fraser, G. (2011, November). Combining search-based and constraint-based testing. In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (pp. 436-439). IEEE Computer Society.

[33] Su, T., Fu, Z., Pu, G., He, J., & Su, Z. (2015, May). Combining symbolic execution and model checking for data flow testing. In Proceedings of the 37th International Conference on Software Engineering-Volume 1 (pp. 654-665). IEEE Press.

[34] Vogels, F., Jacobs, B., & Piessens, F. (2009, January). A machine checked soundness proof for an intermediate verification language. In International Conference on Current Trends in Theory and Practice of Computer Science (pp. 570-581). Springer, Berlin, Heidelberg.

[35] Tanno, H., Zhang, X., Hoshino, T., & Sen, K. (2015, May). TesMa and CATG: automated test generation tools for models of enterprise applications. In Proceedings of the 37th International Conference on Software Engineering-Volume 2 (pp. 717-720). IEEE Press.

[36] Li, K., Reichenbach, C., Smaragdakis, Y., Diao, Y., & Csallner, C. (2013, November). SEDGE: Symbolic example data generation for dataflow programs. In Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (pp. 235-245). IEEE Press.

[37] Do, T., Fong, A. C. M., & Pears, R. (2012, June). Dynamic symbolic execution guided by data dependency analysis for high structural coverage. In International Conference on Evaluation of Novel Approaches to Software Engineering (pp. 3-15). Springer, Berlin, Heidelberg.

[38] Baluda, M., Braione, P., Denaro, G., & Pezzè, M. (2011). Enhancing structural software coverage by incrementally computing branch executability. Software quality journal, 19(4), 725-751.

[39] Pan, K., Wu, X., & Xie, T. (2014). Guided test generation for database applications via synthesized database interactions. ACM Transactions on

Software Engineering and Methodology (TOSEM), 23(2), 12.

[40] Sarkar, T., Basu, S., & Wong, J. (2013, April). Synconsmutate: Concolic testing of database applications via synthetic data guided by sql mutants. In 2013 10th International Conference on Information Technology: New Generations (pp. 337-342). IEEE.

[41] Bae, S. (2014, April). Concolic testing with static analysis for JavaScript applications. In Proceedings of the companion publication of the 13th international conference on Modularity (pp. 7-8). ACM.

[42] Chung, I. S. (2012). Generating Test Data for Programs with Flag Variables using Goal-oriented Concolic Testing. The Journal of The Institute of Internet, Broadcasting and Communication, 12(1), 123-132.

[43] Xu, Z., & Rothermel, G. (2009, December). Directed test suite augmentation. In 2009 16th Asia-Pacific Software Engineering Conference (pp. 406-413). IEEE.

[44] Yu, X., Sun, S., Pu, G., Jiang, S., & Wang, Z. (2011). A parallel approach to concolic testing with low-cost synchronization. Electronic Notes in Theoretical Computer Science, 274, 83-96.

[45] Bardin, S., Kosmatov, N., & Cheynier, F. (2014, March). Efficient leveraging of symbolic execution to advanced coverage criteria. In 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation (pp. 173-182). IEEE.

[46] Prelgauskas, J., & Bareisa, E. (2012). Generating Unit Tests for Floating Point Embedded Software using Compositional Dynamic Symbolic Execution. Elektronika ir Elektrotechnika, 122(6), 19-22.

[47] Bardin, S., & Herrmann, P. (2009, April). Pruning the search space in path-based test generation. In 2009 International Conference on Software Testing Verification and Validation (pp. 240-249). IEEE.

[48] Xu, Z., Kim, Y., Kim, M., Rothermel, G., & Cohen, M. B. (2010, November). Directed test suite augmentation: techniques and tradeoffs. In Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering (pp. 257-266). ACM.

[49] Lakhotia, K., Tillmann, N., Harman, M., & De Halleux, J. (2010, November). Flopsy-search-based floating point constraint solving for symbolic execution. In IFIP International Conference on Testing Software and Systems (pp. 142-157). Springer, Berlin, Heidelberg.

[50] Jensen, C. S., Prasad, M. R., & Møller, A. (2013, July). Automated testing with targeted event sequence generation. In Proceedings of the 2013 International Symposium on Software Testing and Analysis (pp. 67-77). ACM.

[51] Pears, R. L., Fong, A., & Do, T. (2012). Precise guidance to dynamic test generation. DBLP.

[52] Qu, X., & Robinson, B. (2011, September). A case study of concolic testing tools and their limitations. In 2011 International Symposium on Empirical Software Engineering and Measurement (pp. 117-126). IEEE.

[53] Jussien, Narendra, Guillaume Rochart, and Xavier Lorca. "Choco: an open source java constraint programming library." 2008.

[54] Yin, Robert K. Case study research and applications: Design and methods. Sage publications, 2017.

[55] Myers, Glenford J., Corey Sandler, and Tom Badgett. The art of software testing. John Wiley & Sons, 2011.

[56] Williams, Nicky, et al. "Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis." European Dependable Computing Conference. Springer, Berlin, Heidelberg, 2005.

[57] Chen, Ting, et al. "An empirical investigation into path divergences for concolic execution using CREST." Security and Communication Networks 8.18 (2015): 3667-3681.

[58] Baldoni, Roberto, et al. "A survey of symbolic execution techniques." ACM Computing Surveys (CSUR) 51.3 (2018): 50.

EKLER

EK 1 – Örnek Kod

- **Orijinal kod**

```
function []=denemeoriginal(a,b)
c=12;
d=0;
a=a+c;
if (a>5)
    d=2*a+3;
    disp('ilki');
elseif (b<8)
    disp('iki');
end
if (a<13)
    disp('üç')
    if(d>4)
        a=b;
    end
end
while(a<2)
    a=a+10;
end
for i=1:b
    disp(a);
end
end
```

- **Uyarlanmış kod**

```
function [state]=deneme(a,b)
%% İlklendirme işlemleri
dummyvar ={a,b};
dummyvarn = {'a','b'};
dummy=java_array('java.lang.String',100); %#ok<NASGU>
[state]=gncall(2,dummyvar,dummyvarn);
%%
%%State her assignmentin ve decisiondan sonra güncelleniyor
c=12;
[state]=gncasgnc(state,c,'c'); %% constant bir değerin güncellenmesi
d=0;
[state]=gncasgnc(state,d,'d');
a=a+c;
[state]=gncasgns(state,a,'a','a + c');%% sembolik değerin
güncellenmesi
if (a>5)
    [state]=gncif(state,'a > 5',a,5,4);%% if e girildiği durumda pc
'nin state'e kaydedilmesi
    %disp('ilki');
    d=2*a+3;
    [state]=gncasgns(state,d,'d','2 * a + 3');%% yeni bir sembolik
değer ekleme
elseif (b<8)
    cndns={'b < 8','a > 5'};ids={7,4};cncvls={b,8,a,5};
[state]=gncelseif(state,cndnc,cncvls,ids,2); %% elseifden sonra pc
'nin eklenmesi
```

```
    %disp('iki');
end
cndns ={'a > 5','b < 8'}; ids={4,7};cncvls={a,5,b,8};
[state]=gncnotif(state,cndns,cncvls,ids,2,1);%% if veya elseif'e
girilmediyse girilecek fonksiyon, conditionların tersini pc'ye ekliyor
if (a<13)
    [state]=gncif(state,'a < 13',a,13,10);
    %disp('üç');
    if(d > 4)
        [state]=gncif(state,'d > 4',d,4,12);
        a=b;
        [state]=gncasgns(state,a,'a','b');
    end
    cndns={'d > 4'};ids={12};cncvls={d,4};[state]=gncnotif(state,cndns,
    cncvls,ids,1,2);
end
dummy={'a < 13'};ids={10}; cncvls={a,13};[state]=gncnotif(state,cndns,
cncvls,ids,1,1);

while(a<2)
    [state]=gncwhile(state,'a < 2',a,2,1,16);%% while e girildiği
durumda pc 'nin state'e kaydedilmesi
    a=a+10;
    [state]=gncasgns(state,a,'a','a + 10');
end
cndns={'a < 2'};ids={16};cncvls={a,2};
[state]=gncnotif(state,cndns,cncvls,ids,1,1);
for i=1:b
    [state]=gncfor(state,'b >= 1',b,1,1,19);%% for e girildiği durumda
pc 'nin state'e kaydedilmesi
    %disp('1');
end
dummy={'b >= 1'}; ids={19};cncvls={b,1};
[state]=gncnotif(state,cndns,cncvls,ids,1,1);
end
```

EK 2 - Tezden Türetilmiş Bildiriler

Bu tez çalışması kapsamında yapılmış olan sistematik literature taraması, “SLR on Test Data Generation by Dynamic Symbolic Execution” adı altında UYMS 2019’a kabul edilerek sunulmuştur.

HACETTEPE ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
YÜKSEK LİSANS/~~DOKTORA~~ TEZ ÇALIŞMASI ORJİNALLİK RAPORU

HACETTEPE ÜNİVERSİTESİ
FEN BİLİMLER ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI BAŞKANLIĞI'NA

Tarih: 30/09/2019

Tez Başlığı / Konusu: Dinamik Sembolik İşletme Yöntemi ile MATLAB'da Test Verisi Üretimi

Yukarıda başlığı/konusu gösterilen tez çalışmamın a) Kapak sayfası, b) Giriş, c) Ana bölümler d) Sonuç kısımlarından oluşan toplam 64 sayfalık kısmına ilişkin, 30/09/2019 tarihinde ~~şahsım~~/tez danışmanım tarafından *Turnitin* adlı intihal tespit programından aşağıda belirtilen filtrelemeler uygulanarak alınmış olan orijinallik raporuna göre, tezimin benzerlik oranı % 2 'tür.

Uygulanan filtrelemeler:
1- Kaynakça hariç
2- Alıntılar hariç
3- 5 kelimeden daha az örtüşme içeren metin kısımları hariç

Hacettepe Üniversitesi Fen Bilimleri Enstitüsü Tez Çalışması Orjinallik Raporu Alınması ve Kullanılması Uygulama Esasları'nı inceledim ve bu Uygulama Esasları'nda belirtilen azami benzerlik oranlarına göre tez çalışmamın herhangi bir intihal içermediğini; aksinin tespit edileceği muhtemel durumda doğabilecek her türlü hukuki sorumluluğu kabul ettiğimi ve yukarıda vermiş olduğum bilgilerin doğru olduğunu beyan ederim.

Gereğini saygılarımla arz ederim.

30/09/2019

Adı Soyadı:	Halil İbrahim BALCI
Öğrenci No:	N15224995
Anabilim Dalı:	Bilgisayar Mühendisliği A. B. D.
Programı:	Bilgisayar Mühendisliği
Statüsü:	☒ Y.Lisans ☐ Doktora ☐ Bütünleşik Dr.

DANIŞMAN ONAYI

UYGUNDUR.

Dr. Öğr. Üyesi Ayça TARHAN

ÖZGEÇMİŞ

Adı Soyadı	: Halil İbrahim BALCI
Doğum yeri	: ANKARA
Doğum tarihi	: 27.02.1989
Medeni hali	: Evli
Yazışma adresi	: Konya Yolu 8. Km, Oğulbey Mah. 3051. Sok. No:3, 06830 Gölbaşı, Ankara, Türkiye
Telefon	: 05052008193
Elektronik posta adresi	: hibalci@hotmail.com
Yabancı dili	: İngilizce

EĞİTİM DURUMU

Lisans	: ODTÜ, Elektrik-Elektronik Mühendisliği

İŞ TECRÜBESİ

2013 Şubat – 2013 Eylül	: Polaran Ltd. Sistem Mühendisi
2013 Eylül – Halen	: ASELSAN, REHİS, Yazılım Test Mühendisi