Beyaz Kutu Testi İçin Çevrimsel Karmaşıklık ve Test Adımlarının Bulunması

Tuçe SARI[1]

[1]ASELSAN A.Ş., Mikrodalga ve Sistem Tekno. Grubu, P.K. 101, Yenimahalle, Ankara

[1]e-posta: tsari@aselsan.com.tr

Özet

Bu makale kapsamında, bir programın veya program parçasının tüm bağımsız yollarından geçmesi için gerekli test adım sayısını veren ve tüm test adımlarının çıkartılmasını sağlamak amacı ile akış çizgesini oluşturan bir yöntemden bahsedilecektir. Bu amaçla yazılım karmaşıklık hesaplamalarında kullanılan çevrimsel (cyclomatic) karmaşıklık yönteminden söz edilecektir. Testi yapan kişilerin verdiği değişik girdiler ile programın tüm bağımsız yollarının kapsanması mümkün olmamakta, çoğunlukla verilen girdiler tüm program parçasının kapsanması için gereken minimum veri girişinden daha fazla olmakta ve bu durum gerek zaman ve gerekse de para kaybına yol açmaktadır.

Abstract

In this paper, a method which is used to measure the number of test cases to complete the linearly-independent paths through a program or a program module and also is used to form a flow graph to give all the test cases will be studied. For this purpose cyclomatic complexity which is used to calculate the software complexity will be mentioned. All the independent paths of a program cannot be covered by the different test cases provided by the people who make tests; mostly the test cases given to the program module is larger than the necessary test cases and this situation makes loss of time as well as money.

1. Giriş

Yazılım yaşam döngüsü başlıca beş aşamaya ayrıştırılabilir. Bu temel beş aşama sırası ile planlama, çözümleme, tasarım, gerçekleştirim ve bakım aşamalarıdır [1].

Bazı süreç modelleri, yazılım yaşam döngüsü içinde genellikle gerçekleştirim yani kodlama aşamasından sonra sınama aşamasını koymakta ve sınamanın gerçekleştirim sonrası yapılan bir işlevmiş gibi gözükmesini sağlamaktadır. Fakat test aşaması yazılım yaşam döngüsü içindeki her aşamada yapılabilir. Örneğin, planlama aşamasında yapılan doküman gözden geçirme toplantıları ile çıkan hataların maliyetleri, yazılım müşteriye teslim edildikten sonra çıkan hataların maliyetlerinden çok daha azdır. Başka bir örnek vermek gerekirse, bakım ve destek aşamasında hatalı bir durumun düzeltilmesi veya yeni bir işlev eklenmesi için yazılımın değiştirilmesi için sistemin tekrar testlerden geçirilmesi ve istenen biçimde ve doğru çalıştığının kontrol edilmesi gerekmektedir. Yukarıda verilen örnekler ışığında yazılım testi, yazılım geliştirmek için izlenen modelden bağımsız olarak yapılmalıdır. İzlenen yazılım geliştirme modelinde ne kadar ilerlersek hataların bulunup düzeltilmesi de o kadar zorlaşıp, maliyetleri artacaktır. Bu artış da geliştirme modeline göre paralel değil üstel bir şekilde olacaktır. Bu duruma bir örnek olarak Tablo 1'de [2]'den alınmış rakamlar ve her adımda harcanan maliyetlerin yüzdeleri verilmiştir.

Gerçekleştirimin ilk safhalarında bulunan hatalar, son safhalarında bulunan hatalardan daha az can sıkıcı olmaktadırlar. Her ne kadar yazılımları kodlayan programcılar, yazılım bittikten sonra testlerin uygulanması yönünde eğimli olsalar da erken safhalarda bulunan hatalar programın ve dolayısıyla projenin verimini arttırmaktadır. Bu doküman kapsamında gerçekleştirim sırasında yapılan testlerden bahsedilecektir.

Tablo 1. Maliyet Yüzdeleri

Geliştirme Süreci		**Geliştirme Sonrası Süreç**	
Gereksinim Analizi	%3	Bakım ve Destek	%67
Gereksinim Özellikleri	%3		
Tasarım	%5		
Kodlama	%7		
Test	%15		

Bir yazılıma uygulanabilecek birçok test yöntemi vardır. Bunlardan birkaçı birim testi, entegrasyon testi, fonksiyonel test, regresyon testi olarak sıralanabilir. Sağlam bir yazılım için yazılımın içinde bulunduğu sistem ile arayüzlerinin doğru çalıştığının kontrolü kadar o yazılımda bulunan her bir fonksiyonun doğru çalışıp çalışmadığı, doğru girdiye karşılık doğru çıktı üretip üretmediği ve fonksiyondaki tüm koşul ve döngü deyimlerinin doğru çalışıp çalışmadığı test edilmelidir. Sağlam bir yazılım için birim testinin yapılması gerekmektedir.

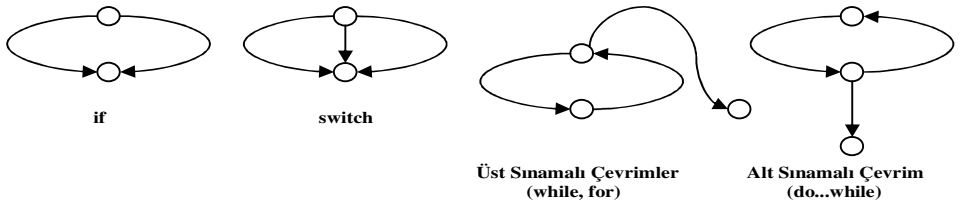
Kodlama ve test aşamalarında Beyaz Kutu (White-Box) veya Kara Kutu (Black-Box) testleri kullanılarak değişik stratejiler izlenebilir. Kara Kutu testleri genelde programı yazan kişiler tarafından değil, testçiler tarafından yapılan ve yazılımın dış arayüzleri üzerine yoğunlaşan testlerdir. Kara Kutu testinde karmaşık ve bu yüzden de hataya açık olan kod parçaları, testçi tarafından bilinmediği için bunların üzerinde gereken yoğunlaşma yapılmayabilir ve hataya açık olan kod parçaları düzgün ve tam bir biçimde test edilmeyebilir. Buna karşılık yazılımı yazan kişilerin yani programcıların kendi yazılımlarının istendiği gibi çalışıp çalışmadığını test etmeleri için uyguladıkları ve kod içeriğini göz önüne alan testlere Beyaz Kutu testi denir. Beyaz Kutu testi ile programcılar tüm bağımsız yolların en az bir kez kullanıldığını, tüm mantıksal deyimlerin doğru ve yanlış durumlarında hata üretmeden çalıştığını, tüm döngülerin limit değerlerinde doğru olarak çalıştığını kontrol edebilirler. Beyaz Kutu testi kapsamında bir modül içindeki tüm yolların, tüm döngülerin, tüm mantıksal ifadelerin kontrolü yapılmalıdır.

Beyaz Kutu testinde kullanılan temel olarak iki adet yöntem vardır. Bu yöntemler, "Ana Yol" (Basis Path) ve "Kontrol Yapısı" (Control Structure) testleri olarak iki grupta incelenebilir. Ana Yol testi ile, yazılımın çalışma sırasında olası tüm yollardan en az bir kere geçip geçmediği test edilir. Mantıksal çalışma akışı ve yordam tasarımını sunmak için akış çizgelerinden yararlanılır. "Yol Kapsama" (Path Coverage) ve "Veri Akış" (Data Flow) test yöntemleri bu grup altında incelenebilir. Kontrol Yapısı testi, yazılımın kod ve yapısı üzerinde yapılan testleri içerir. "İfade Kapsama" (Statement Coverage), "Dal Kapsama" (Branch Coverage), "Durum Testi" (Condition Testing) bu grup altında incelenebilir. Beyaz Kutu testi yaparken ana yol testinin yanında kontrol yapısı testleri de yapılmalıdır. Her iki test grubu yapılmadan ilgili yazılım birimine ilişkin Beyaz Kutu testi bitirilmemelidir.

Beyaz Kutu testinin yapılması, hataların bulunup ayıklanması konusunda diğer testlerden daha fazla fayda sağlar. Bir program kesiminin uygulamada az çalışma beklentisi varsa bu kesim dikkatli

yazılmayabilir. Kullanılma olasılığı az olan kesimler beklenenden fazla çalışıyorsa, dikkatli yazılmadığı için kod parçalarında ve derleyicilerin uyarı veremeyeceği yerlerde rasgele hataların çıkma olasılığı fazladır. Bu hataların Kara Kutu testi ile bulunması çoğunlukla gerçekleşmemektedir. Hatalar daha sonra beklenmedik yerlerde ve durumlarda ortaya çıkmakta ve hem müşteriyi hem de firmayı zor durumda bırakmaktadırlar.

Bu bildiri kapsamında Beyaz Kutu test yöntemlerinden olan Ana Yol testi üzerinde durulacaktır. Ana Yol testi, yazılımın çalışması esnasında geçebileceği bağımsız yolları ortaya çıkarır. Bu bağımsız yolların saptanması için önce programın çizgesel bir şekilde ifade edilmesi gerekmektedir. Bu çizgesel ifade akış çizgesi kullanılarak yapılır. Akış çizgesinde her program parçasının bir başlangıç ve bir bitiş noktası vardır. Başlangıç ve bitiş noktaları düğümler ile ifade edilir. Bunların arasında her koşul veya döngü deyimi için ayrıca bir düğüm kullanılır. Düğümler arasında ise düğümleri birbirlerine bağlayan kenarlar yer alır. Koşullara karşı gelen düğümler dallanma oluştururlar; bu ayrışmalar, ilgili koşul akışlarının sonunda yeniden birleşimi sağlayacak ek düğümlerle sonlandırılmalıdır. Şekil 1'de C ve C++ dili için koşul ve döngü deyimlerinin akış çizgesi örneği gösterilmiştir:



Şekil 1. Koşul ve Döngü Deyimleri için Akış Çizgesi

Dal kapsama işlemi sırasında program içindeki her deyimin çalıştırılıp, kapsanması gerekmektedir. Her dalın en az bir kere kapsanacağı yollar bulunmalıdır ki testçi testlerini minimum zamanda ve en verimli şekilde gerçekleştirebilsin. Bu minimum test adımlarının bulunmasında çevrimsel karmaşıklık yöntemi yardımcı olur. Bu makalede sunulan yöntem yardımı ile bir fonksiyon veya metodun akış çizgesi çıkartılmakta ve çevrimsel karmaşıklık hesabı yapılmaktadır. Daha sonra çıkartılan bu akış çizgesi yardımı ile test adımları bulunmaktadır.

Bu bildiri kapsamında, Bölüm 1'de yazılım testi genel olarak anlatılmıştır. Bölüm 2'de çevrimsel karmaşıklığın ne olduğu ve neden gerektiği açıklanmaktadır. Bölüm 3'de geliştirilen yöntem ile oluşturulan araç hakkında bilgi verilmektedir. Bölüm 4'te ise test adımlarının bu araç yardımı ile nasıl elde edildiğinden bahsedilmekte ve çevrimsel karmaşıklık hesaplamak için kullanılan akış çizgeleri yardımı ile fonksiyonları veya metotları test etmek için gereken en az sayıda test adımlarının bulunduğu örnekler verilmektedir. Son olarak Bölüm 5'te kısa bir değerlendirme yapılmaktadır. Bildirinin genelinde C ve C++ dillerinde örnekler verilmiştir.

2. Çevrimsel Karmaşıklık

Çevrimsel karmaşıklık, bir programın veya program parçasının karmaşıklığını ölçmek için bir ölçüt sunar. Bu ölçüte göre bir programda kullanılan koşul ve döngü deyimleri bir yazılım programının karmaşıklığını etkileyen yegâne etkendir. Çevrimsel karmaşıklık, programdaki satır sayısı gibi ölçütlerden etkilenmez. Çevrimsel karmaşıklığın uygulanması için öncelikle programın çizge biçimine dönüştürülmesi gerekmektedir. Daha sonra oluşturulan çizgeden karmaşıklık hesaplanır.

Çevrimsel karmaşıklık iki şekilde hesaplanabilir [3]:

$$V(G) = K - D + 2 \quad (1)$$

$$V(G) = Kapalı\ Alan\ Sayısı + 1 \quad (2)$$

Denklem 1'deki D çizgedeki düğüm sayısını, K ise çizgedeki kenar sayısını verir. Denklem 2'deki kapalı alan ise düğüm ve kenarlarla kapatılmış alan anlamına gelmektedir.

Kuvvetli bağlı grafiklerde (strongly connected graph) bulunan bağımsız yolları hesaplamaya yarayan bu formülün nasıl türetildiğinin bilgisi [4]'te detaylı olarak anlatılmıştır.

Çıkan *V(G)* değeri dal kapsama testinde kapsanması gereken yollar için uygulanması gereken minimum test adım sayısını verir. Çevrimsel karmaşıklık, program karmaşıklığının yanı sıra bir yazılımda tüm doğrusal bağımsız yollar için bir üst sınır verir ve maksimum sayıda doğrusal bağımsız yolları göstermenin pratik bir yolunu tanımlar. Böylece test için gereken minimum test adımı sayısını göstererek yazılımları test etmenin zorluğunun ölçütünün de bir göstergesi olur.

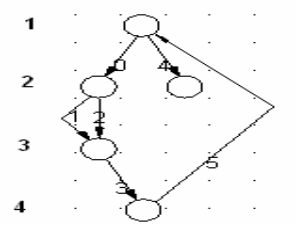
Yapılan araştırmalar çevrimsel karmaşıklık ile yazılımın hata sıklığı arasında bir ilgileşim olduğunu göstermiştir. Düşük çevrimsel karmaşıklık programın anlaşılabilir, değiştirilmesinin kolay ve test edilebilirliğinin fazla olduğunun göstergesidir.

Tablo 2'de [5]'ten alınan bazı sınır değerler ile çevrimsel karmaşıklık arasındaki ilişki verilmiştir.

Tablo 2. Çevrimsel karmaşıklık sınır değerleri

Çevrimsel karmaşıklık	Anlamı
1 – 10	Risk taşımayan, basit bir program
11 – 20	Daha karmaşık, riskli bir program
21 – 50	Karmaşık ve çok riskli bir program
> 50	Bakımı yapılamayacak kadar risk taşıyan bir program

Şekil **2**'de C dilinde yazılmış sıralı arama yapan bir fonksiyonun akış çizgesi verilmiştir. Bu fonksiyonun çevrimsel karmaşıklığı Denklem 1 kullanılarak hesaplandığında, 5 adet düğüm ve 6 adet kenarı olduğu için *6–5+2=3* olarak bulunur. Bu da Tablo 2'ye göre risk taşımayan, basit bir program anlamına gelmektedir ki akış çizgesine bakıldığında da bu fonksiyonun kaç tane koşul ve döngü deyimi içerdiği görülmektedir. En tepede 1 sırasındaki düğüm bir üst sınamalı döngü deyiminin düğümüdür. 2 numaralı sırada, 0 numaralı kenarın girdiği düğüm bir koşul deyiminin düğümüdür. Bu düğümün iki koşulu olduğu 1 ve 2 numaralı kenarlardan anlaşılmaktadır. Bu gözlemlerden yola çıkarak bu akış çizgesinin bir adet üst sınamalı döngü deyimi ve bir adet koşul deyiminden oluştuğu anlaşılmaktadır.

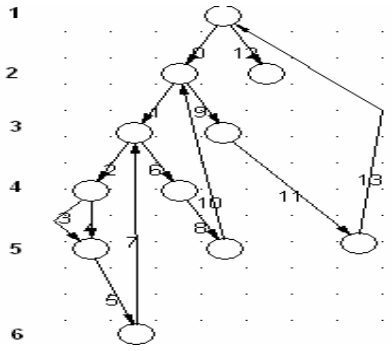


Şekil 2. Sıralı Arama Yapan Fonksiyonun Akış Çizgesi

Şekil **3**'de ise yine C dilinde yazılmış kabuk sıralama yapan bir fonksiyonun akış çizgesi verilmiştir. Bu fonksiyonun çevrimsel karmaşıklığı Denklem 1 kullanılarak hesaplandığında, 11 adet düğüm ve 14 adet kenarı olduğu için *14–11+2=5* olarak bulunur. Akış çizgesine bakıldığında 1, 2 ve 3. sıraların sol tarafındaki düğümlerin birer üst sınamalı içice döngü olduğu görülür ve en içteki döngüde yani 4 numaralı satırda, 2 numaralı kenarın girdiği düğümde bir koşul döngüsü olduğu görülür.

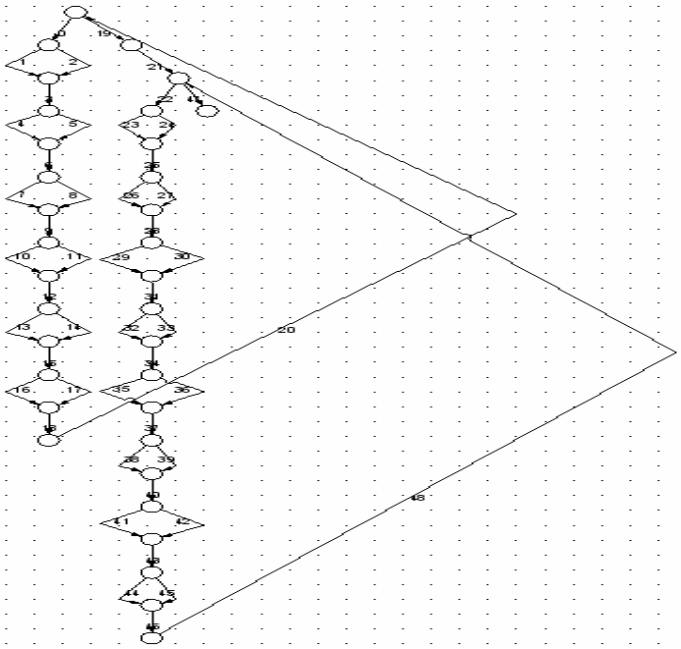
Şekil **2**'deki akış çizgesine de bakarsak, bu örneğin

Şekil **2**'deki örnekten daha karmaşık gözüktüğü görülür. Bu farkı, iki örneğin çevrimsel karmaşıklık değerlerinden de anlayabiliriz. İkinci örnek, ilkinden daha büyük bir karmaşıklık değerine sahip olsa da iki örnek de Tablo 2'ye göre riski az olan program parçalarından oluşmaktadır.



Şekil 3. Kabuk Sıralama Algoritmasının Akış Çizgesi

Şekil 4'de ise [6]'dan alınan ve C++ dilinde yazılmış bir takvim uygulamasına ait bir metodun akış çizgesi bulunmaktadır. Diğer örneklerden daha karışık bir görünümü olan bu akış çizgesinin, çevrimsel karmaşıklık değerini ise yine Denklem 1 kullanılarak, akış çizgesindeki düğüm ve kenar sayılarını hesaplayarak bulabiliriz. 49 adet kenar ve 34 adet düğümü olan bu akış çizgesinin karmaşıklığı 17 olarak bulunur. Bu da yukarıdaki ilk örnekten yaklaşık 6 kat, ikinci örnekten ise yaklaşık 3,5 kat fazla karmaşıklığa sahip olduğunu gösterir. Akış çizgesinden de karşılaştırıldığında bu fark rahatlıkla gözlemlenir.



Şekil 4. C++'da Yazılmış Bir Takvim Uygulamasının Bir Metodunun Akış Çizgesi

Yukarıda verilen örneklerden anlaşılacağı gibi programın karmaşıklığının artması akış çizgesinin de karmaşık bir hal almasına neden olmaktadır. Çıkan bu karmaşıklık değeri tasarım ve kodlama aşamasında uygun bir sınırda tutulduğu takdirde, yani bağımsız yollara bir limit konulduğu takdirde testlerin yapılabilirliği diğer aşamalarda da korunacaktır [7]. Bu şekilde hem risk yönetimi sağlanmış hem de program satır sayısı değil karmaşıklık sayısı temel alınmış olunacaktır.
Şekil 4'teki karışık akış çizgesinden de anlaşılacağı gibi ilgili program parçasının test adımlarını çıkartmak oldukça zahmetli ve zaman alıcı olduğu kadar programcının yaptığı işten sıkılmasına ve zamanının çoğunu test yazmak için harcamasına yol açacak bir hal almasına neden olacaktır.

Tablo 2'de sınır değer olarak 50 verilmiştir fakat her yazılım için ilgili yazılım ekibi içinde alınan karara göre istenilen ve karar verilen bir değer, sınır değer olarak kabul edebilir. Ayrıca limit değerden daha büyük karmaşıklık değerinin çıkması çoğu zaman iyi bir kodlama yapılmadığının da belirtisini verebilmesi açısından önem taşımaktadır.

Çevrimsel karmaşıklık sadece test aşamasında kullanılan bir yöntem değildir. [5]'te diğer kullanılabilirlik alanları verilmiştir. Bunlardan biri risk analizinde kullanımıdır. Kod geliştirme aşamasında iken, çevrimsel karmaşıklık değeri ile çevrimsel karmaşıklığın doğasında olan risk değerleri önceden belirlenebilir. Bir diğer kullanım ise bakım ve destek aşamasındadır. Kodun karmaşıklığı bakımdan önce ve sonra değişebilmektedir. Bakımdan önce ve sonra karmaşıklık değerine bakarak değişimin riski minimuma indirilebilir. Bir diğer kullanımı ise test planı aşamasındadır. Çevrimsel karmaşıklık, test için gerekli olan minimum sayıyı vermektedir. Karmaşık modüllerin daha ufak parçalara bölünmesi sağlanarak test edilebilirlik arttırılabilir.

3. Test Adımlarının Bulunması

Test adımlarını bulmak için geliştirilen yöntem kapsamında, verilen program veya program parçasının çevrimsel karmaşıklık değeri hesaplanır, akış çizgesi bulunur ve bu çizge yardımı ile test adımlarının bulunması sağlanır. Geliştirilen yöntem ile oluşturulan araç daha önceden derlenen ve hatası olmayan bir projeyi alır ve tersine mühendislik yaparak ilgili projenin içindeki sınıfları ve bunların içindeki metotları bulur. İstenirse bu aracın ilgili projeyi derleyip, hataları bulması yani tıpkı bir derleyici gibi çalışması da sağlanabilir. Tersine mühendislik yapmak için yazılabilecek ufak bir ayrıştırıcı (parser) veya hali hazırda bulunan daha kapsamlı bir ayrıştırıcı kullanılabilir. Daha sonra geliştirilen araç ile bu sınıflar ve sınıfların içindeki metotlar kullanıcıya sunulur. İlgili metotların istendiği takdirde, çevrimsel karmaşıklık değerleri hesaplanır. Hesaplama işlemi sırasında ayrıştırıcı ile Şekil 1'de verilen koşul ve döngü deyimlerinin ayıklanması sağlanır. Şekil 1'deki her koşul ve döngü deyimi ilgili program parçası içinde bulunduğu zaman, daha sonra kullanıcıya bir grafik üzerinde sunulacak olan akış çizgesine düğümler ve kenarlar olarak eklenir. Bu şekilde oluşturulan akış çizgesi, kullanıcının test adımlarını nasıl yapacağını görebilmesi için bir grafik üzerine çizdirilir. Ayrıca tüm bağımsız testleri yapabilmek için kaç adet testin yapılması gerektiği yani hesaplanan çevrimsel karmaşıklık değeri de kullanıcıya sunulur. Çevrimsel karmaşıklık değeri, oluşturulan akış çizgesindeki düğüm ve kenar sayılarına Denklem 1'i uygulayarak elde edilir. Oluşturulan grafik yardımı ile kullanıcı ilgilendiği program parçasının tüm test adımlarını rahatça çıkartmakta ve test oluşturma konusunda hız kazanmaktadır. Ayrıca programcı, bu grafiksel gösterim yardımı ile karmaşık gözüken metotları küçültme yoluna giderek daha sağlam ve bakım yapılabilir bir yazılım üretebilmektedir.

Akış çizgesi ile test adımları kolayca çıkartılır. Akış çizgesinde, her düğüm ilgili koşul veya döngü deyimini gösterir ve düğümlerin hangi deyime ait olduğu kullanıcıya grafik yardımı ile gösterilir. Bu sayede kullanıcı test adımı yazarken programın kaynak kodu ile ilgilenmez, sadece ilgili deyimlerdeki değişken değerleri ile ilgilenir. Test adımları bu şekilde sıra ile yazılır. Her bir yol için grafikteki akış çizgesinde bir sayı verilmiştir ve bu sayılar ile test adımlarının hangi yollardan geçeceği yazılır. Kullanıcının karışık program parçalarına kolay ve hızlı bir şekilde test adımlarını yazabilmesi için bu numaralı kenarlar kullanılır. Çıkartılan bu test adımları daha sonra programcının veya yazılım firmasının kullandığı test aracı yardımı ile oluşturulur ve test aracının sunduğu olanak ile otomatik test adımları yazılmış olunur. Bu sayede dal kapsama testlerinde tüm dalların kapsanmış olduğundan emin olunur.

Geliştirilen bu araç yardımı ile test adımlarının kolay bir şekilde görselleştirilip, çıkartılması sağlanmış olmaktadır. Piyasada kolayca bulunan ve kullanılan test araçları Beyaz Kutu testi uygulama olanaklarını sağlamaktadır. Fakat bu araçların çoğu yol kapsama testi için test adımlarının kullanıcı tarafından tanımlanmasını beklemektedir. Piyasadaki bu araçlar yol testi için o an program hangi parametreler ile çalıştırılmışsa, çalıştırılan parametreler doğrultusunda hangi yolun izlendiğini ve hangi yolların kapsandığını vermektedir. Bu tür araçlar çıktı olarak ise, verilen girdilerle ilgili programda kapsanan yolların yüzdesini vermektedirler. Testi yapan kişilerin yol kapsama testi için verdiği değişik girdiler ile programın tüm bağımsız yollarının kapsanması mümkün olmamakta, çoğunlukla verilen girdiler ile tüm program parçasının kapsanması gereken minimum veri girişinden daha fazla olmakta ve bu durum gerek zaman ve gerekse de para kaybına yol açmaktadır. Kullanılan hata bulmaya yarayan araç nesne yönelimli test aracı ise, genellikle sınıfın arayüzlerinin doğru çalışıp çalışmadığı ile ilgilenmekte, yazılmış olan metotların detaylı birim testleri ile ilgilenmemektedir.

4. Akış Çizgesinde Test Adımlarının Gösterilmesi

Şekil 5'te C dili ile yazılmış, sıralı arama yapan bir fonksiyonun kaynak kodu,

Şekil **2**'de ise bu fonksiyonun akış çizgesi yer almaktadır.

```
int SequentialSearch(searchArray,i, j, key)
{
        while (i <= j)
        {
                if (searchArray[i] == key) //found
                {
                        return i;
                }
                i++;
        }
        return -1; //not found
}
```

Şekil 5. Sıralı Arama Yapan Fonksiyon

Koddan veya akış çizgesinden anlaşılabileceği gibi bir üst sınamalı döngünün içinde bir koşul deyimi yer almaktadır. Akış çizgesinde en tepede 1 sırasındaki düğüm bir üst sınamalı döngü deyimi olan `while` döngüsüne aittir. 2 numaralı sırada, 0 numaralı kenarın girdiği düğüm bir koşul deyimi düğümü olan `if` koşuluna aittir. Akış çizgesinden görüldüğü gibi 5 adet düğüm, 6 adet kenar vardır. Daha önceden de hesaplandığı gibi Denklem 1 uygulandığında 6–5+2=3 bulunur. Aşağıda bulunabilecek 3 adet test adımı,

Şekil **2**'deki akış çizgesi üzerine yazılmış olan kenar numaraları kullanılarak çıkartılmıştır. Bu test adımlarından daha farklı yollardan gidilerek bulunabilen adımlar da mevcut olabilir. Önemli olan tüm bağımsız yollardan en az bir kez geçmiş olmaktır.

- 0 – 4
- 0 – 1 – 3 – 5 – 4
- 0 – 2 – 3 – 5 – 4

Şekil 6'da C dili ile yazılmış, kabuk sıralama yapan bir fonksiyonun kaynak kodu,
Şekil **3**'de ise bu fonksiyonun akış çizgesi yer almaktadır.

```
void ShellSort(sortArray, int N)
{
        int i, j, increment;
        ElementType arrayElement;

        for(increment = N / 2; increment > 0; increment /= 2 )
        {
                for( i = increment; i < N; i++ )
                {
                        arrayElement = sortArray[i];
                        for( j = i ; j >= increment; j -= increment )
                                if(arrayElement < sortArray[j - increment])
                                        sortArray[j] = sortArray[j - increment];
                                else
                                        break;
                        sortArray[j] = arrayElement;
                }
        }
}
```

Şekil 6. Kabuk Sıralama Algoritması

Koddan veya akış çizgesinden anlaşılabileceği gibi iç içe geçmiş üç adet üst sınamalı döngünün içinde bir koşul deyimi yer almaktadır. Akış çizgesinden görüldüğü gibi 1, 2 ve 3. sıraların sol tarafındaki düğümler birer üst sınamalı iç içe döngü olan `for` döngüsüne ve en içteki döngü yani 4 numaralı satırda, 2 numaralı kenarın girdiği düğüm ise bir koşul deyimi olan `if` deyimine aittir. Akış çizgesinden görüldüğü gibi 11 adet düğüm, 14 adet kenar vardır. Daha önceden de hesaplandığı gibi Denklem 1 uygulandığında 14–11+2=5 bulunur. Aşağıda bulunabilecek 5 adet test adımı,
Şekil **3**'teki akış çizgesi üzerine yazılmış olan kenar numaraları kullanılarak çıkartılmıştır.

- 0 – 12
- 0 – 9 – 11 – 13 – 12
- 0 – 1 – 6 – 8 – 10 – 9 – 11 – 13 – 12
- 0 – 1 – 2 – 3 – 5 – 7 – 6 – 8 – 10 – 9 – 11 – 13 – 12
- 0 – 1 – 2 – 4 – 5 – 7 – 6 – 8 – 10 – 9 – 11 – 13 – 12

Akış çizgelerinde dikkat edilmesi gereken bir durum iç içe olan ve olmayan döngüler için aynı karmaşıklık değeri bulunabilmesidir. Fakat iç içe döngülerin anlaşılmasının daha zor olduğu unutulmamalıdır. İçinde birçok ufak karşılaştırmanın olduğu veya yazılımın durumlarının tutulduğu yapılarda karmaşıklık değeri fazla çıkabilir fakat bu kodun karmaşık veya test edilebilirliğinin zor olduğu anlamına gelmez. Oluşturulan akış çizgesi ile bu durumlar kolayca tespit edilebilir. Test adımlarının yazılmasında geliştirilen aracın kullanılabilmesi ile hem test adım sayısı öğrenilmekte hem de test adımlarının nasıl oluşturulacağı bir akış çizgesi ile hızlıca çıkartılmaktadır. Test adımı çıkartılması istenen programların karmaşıklığı ne kadar artarsa, sonuçta ortaya çıkacak olan akış çizgesi karmaşık gözükecektir. Bu da ister istemez, programcıya ilgili kod parçasını sadeleştirmeye yönlendirecektir. Büyük ve karışık bir kod parçasına test yazmaktansa, bunu ufak parçalara bölüp, her birine ufak testler yazmak daha az zahmetli olacaktır.

4. Sonuç

Çevrimsel karmaşıklık bir metrik aracı olarak kullanılabildiği gibi akış çizgesi yardımı ile test adımlarının çıkartılmasında da kullanılan etkili bir yöntemdir. Değişik uygulamalar üzerinde yapılan araştırmalar sonucunda çevrimsel karmaşıklığın hızlı, etkili ve uygulanabilir bir yöntem olduğu görülmüştür. Bu kapsamda birim testlerinin yapılmasını sağlayan bir araç kullanılarak çevrimsel karmaşıklıkta çizilen akış çizgesinden faydalanılıp, ilgili metot veya fonksiyonların tüm test adımlarının kolayca çıkartılması sağlanmıştır. Tablo 1'de verilen değerleri temel alarak programların içindeki metot veya fonksiyonların tümünün çevrimsel karmaşıklıklarının çıkartılması ve gerekli görülenlerin daha küçük çevrimsel karmaşıklık değerine sahip olana kadar parçalanması sonucunda ilgili program parçasının daha okunabilir, dolayısı ile anlaşılabilir, ilerde olabilecek değişikliklere açık ve hatalara karşı sağlam olduğu gözlenmiştir.

Çevrimsel karmaşıklık, bakım ve destek aşamasının yanı sıra program parçalarının kalite metriklerinin çıkartılması için yardımcı bir yöntem olarak kullanılabilir. Yazılım kodlamanın ilk aşamasından itibaren yapılabilecek basit bir yöntemdir. Test aşamasında kullanılacak olan minimum test adımlarını çıkartmakta faydalı bir yöntemdir. Bu yöntem bir ana yol kapsama testi

olduğu için bir programın Beyaz Kutu testinin tamamlanması için Kontrol Yapısı testinin de uygulanması gerektiği unutulmamalıdır.

Kaynakça

[1]. Arifoğlu A., Doğru A., "Yazılım Mühendisliği: Yöntemler, Metodolojiler, CASE Ortamları, Günün Teknolojisi", SAS Bilişim Yayınları, Ankara, 2001

[2]. Martin J., McClure C. L., "Software Maintenance: The Problems and Its Solutions", Prentice-Hall, Inc, Englewood Cliffs, NJ., 1983.

[3]. Watson A. H., McCabe T. J., "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric", NIST Special Publication 500-235, 1996

[4]. Thomas J. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, vol.Se-2, No.4, 1976

[5]. http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html

[6]. http://www.cprogramming.com/source/calendar.cpp?action=Jump&LID=39

[7]. Thomas J. McCabe, "Structured Testing: A Testing Methodology Using the McCabe Complexity Metric", NBS Special Publication, s. 82-119, 1982