

## Yazılım Testinde Sonlu Durum Otomatlarının Kullanılması

**Zeynep Altan**

Beykent Üniversitesi, Yazılım Mühendisliği Bölümü, İstanbul  
zeynepaltan@beykent.edu.tr

**Özet:** Hangi büyüklükte olursa olsun bir yazılım geliştirilirken müşterinin isterlerini en yüksek düzeyde karşılayacak ürünü ortaya çıkarmak temel amaçtır. Yazılım testinin, isterlerin belirlenmesinden başlayarak çalışmanın her aşamasında farklı yöntemlerle gerçekleştirilmesi, müşterinin yüksek kalite ölçütlerini sağlayan bir ürüne sahip olması ile sonuçlanacaktır.

Test işleminde soyut makinelerden yararlanıldığında sistemin fonksiyonel gereksinimleri incelenebildiği gibi fonksiyonel olmayan gereksinimlerinin de kontrolü mümkün olur. Sonlu durum makinesi ile yazılım geliştirme araçlarından biri olan Birleştirilmiş Modelleme Dilinin farklı gösterimleri tanımlanabilir. Bu gösterimlerden biri olan UML durum çizeneği ile sistemin dinamik davranışı incelenir. Sistemin herhangi bir bileşenin farklı durumlarının betimlendiği durum çizeneği bileşen-tabanlı yazılım geliştirmeye örnek olarak hızlı bir çözüm sağlar. Koda erişimin sınırlı olduğu durumlarda problemin çözümü bu şekilde tasarlanarak, bileşenin testi için de kara kuru testinden yararlanmak mümkündür.

Durum çizeneğinin sonlu durum makinesine dönüştürülmesi ile test işlemi başlar. Bu şekilde betimlenen fonksiyonel olmayan gereksinimler bir kara-kutu testi olarak sınanır. Farklı test senaryoları araştırılırken çözümlerin izlenebilirliğini kolaylaştırmak için, sonlu durum otomatına karşılık gelen düzgün dilbilgisi dönüşümü yapılabilir; böylece her bir senaryo kendisini oluşturan türetme kuralları ile ifade edilmiş olur. Sonuç olarak; bileşen-tabanlı yazılım geliştirmede UML durum çizenekleri ile elde edilen çözümün testi için sonlu durum otomatlarından yararlanılması, otomatik test geliştirme araçlarının üretimine yardımcı olacaktır.

**Anahtar Sözcükler:** Yazılım Testi, Yazılımın Niteliği, Birleştirilmiş Modelleme Dili(UML), Sonlu Durum Otomatları, Düzgün Dilbilgisi.

**Abstract:** During the development of software by any size whatsoever, the fundamental goal is to generate the product in such a way that the customer requirements will be supplied at the highest level. Software testing must be carried out with various methods at each developing steps beginning from the software requirements phase. Therefore the user will arguably use the product achieving high quality measurements after the delivery.

If test process is realized utilizing the abstract machines, it is possible both to confirm functional and nonfunctional requirements of the product. State chart diagram as a classification of Unified Modeling Language (UML) can also be characterized with finite state automaton at which dynamic behavior of the system is analyzed. The state graph determining the different states of any component on the system concludes the design as a rapid solution and this formation is an example of component-based software development. When it is impossible to access the code, the design may be done with state chart diagram and block box testing can be executed to test that component.

Test process starts by converting the state chart into finite state automaton. Thus nonfunctional requirements will be analyzed as a black box test. For different test scenarios, regular grammar conversion helps to simplify the traceability of the probable paths. Therefore each scenario will be expressed with its own derivation rules.

Consequently, finite state automata as a representation of state chart diagrams used to model the component based software development will help to generate automatic test developing tools

**KeyWords:** Software testing, Software Quality, Unified Modeling Language, Finite State Automata, Regular Grammar

### 1. Giriş

Bir yazılımın ürününün herhangi bir davranışının sistem tarafından kabul edilen bir dizi giriş değerine göre betimlenmesi mümkündür; bunlar sistemin birbirinden farklı eylemleri, koşullar, çıktının verilmiş düzeni veya uygulama modüllerinin ya da yordamların veri akışı olabilir. Test sürecinde uygulanan modelin

farklı işlemler için testin gerçekleştirilmesine katkısı, kolay anlaşılabilir olarak yazılması ile ölçülür. Ortak olarak paylaşılabilen, yeniden kullanılabilen ve test tanımlamaları kesinleşmiş modeller farklı bakış açılara göre pek çok farklı yazılım davranışı gösterebilir. Örneğin kontrol akışı, veri akışı ve programa bağlı çizgeler ürünün kaynak kod yapısına göre nasıl betimlendiğini ifade ederken; karar tabloları

ve durum makineleri kara-kutu testi olarak adlandırılan sistemin davranışındaki dış betimlemeleri test eder.

Bileşen tabanlı yazılım geliştirme[1] hızlı ve düşük maliyetli etkin yazılım geliştirme teknolojilerinden biridir. Yeni ve tekrar kullanılan bileşenlerden oluşan bu teknolojide bileşenlerin birbirleri ile iletişimi arayüzlerle gerçekleştirilir. Bu sistemler yazılım sistemini iyi-tanımlanmış bir mimari ile mevcut bileşenleri birleştirerek oluşturur. Böylece geliştirilen sistemin üretkenliği artar ve maliyeti düşer. Yazılım geliştirme araçlarından biri olan birleştirilmiş modelleme dilinin UML durum çizeneği ile sistemin dinamik davranışı incelenir; burada bir nesnenin farklı durumları ve bu durumlar arasındaki olası geçişler betimlenir. UML modelleme ile hem yazılım sistemini, hem de tüm sistemin yaptıklarını görsel olarak açıklamak mümkündür [2]. Sistemin herhangi bir bileşeninin farklı durumlarını gösteren durum çizeneği bileşen-tabanlı yazılım geliştirmeye örnek olarak hızlı bir çözüm sağlar.

Sistemin bu şekilde testi model tabanlı yazılım testi [3] olarak adlandırılır. Test işlemlerinin sonlu durum makineleri [4] ile gerçekleştirilmesi, model-tabanlı yazılım sistemlerin testinde bir yaklaşım biçimidir. Sonlu durum makineleri ile sisteminin dinamik ve statik davranışlarının hesaplanabilir bir modeli oluşturulur. Bu makineler ile sonlu sayıda belirlenmiş durumların doğru olarak betimlenmesi mümkündür. Bilgisayar donanımı bileşenlerinin tasarımı ve testi sonlu durum modelleri kullanılarak gerçekleştirilir.

Test işlemi sadece giriş tümcelerinin bir dizi sıralanışı ile test verilerinin sağlanması olduğu için, sonlu durum makineleri yazılım testi için de oldukça uygundur. Ayrıca sonlu durum makineleri yazılım yapısından çok semantiğine bağlıdır. Bu yapı üzerinde yapılan test işlemleri, yazılımın davranışının incelendiği giriş-çıkış betimlemeleri üzerinde odaklanmaz.

Durum çizeneğinin sonlu durum makinesine dönüştürülmesi ile test işlemi başlar. Bu şekilde betimlenen fonksiyonel olmayan gereksinimler bir kara-kutu testi olarak sınanır; burada doğru ve yanlış giriş cümleleri ile başlangıç durumundan kabul durumuna her yolun doğru olarak çalışıp çalışmayacağı kontrol edilir. Farklı test senaryoları araştırılırken çözümlerin izlenebilirliğini kolaylaştırmak için, tasarlanmış olan sonlu durum otomatına karşılık gelen düzgün dilbilgisi dönüşümü gerçekleştirilir; böylece her bir senaryo kendisini oluşturan türetme kuralları ile ifade edilir. Çalışmanın 2. Bölümünde otomatlar ve dilbilgisi teorisine ilişkin biçimsel tanımlar verildikten sonra, 3.Bölümde problem tanımlanmakta ve UML çizgeleri ile görsel olarak çözümlenmektedir 4. Bölümde UML durum çizgesinin belirlenimci olmayan sonlu durum

otomatına ve düzgün dilbilgisine dönüşümü yapılır. 5.Bölümde ise bileşen tabanlı yazılım modelinin sonlu durum otomatlarına dayanarak kara kutu testi gerçekleştirilmektedir.

## 2. Temel Kavramlar

Biçimsel diller teorisi bilgisayar bilimlerinde pek çok uygulamanın geliştirilmesinde yararlanılan temel alanlardan biridir. Otomatlar, biçimsel diller teorisi kapsamında bilgisayarların soyut birer modelidir. Eğer otomatın çıktısı olarak yanıtı sadece “evet” veya “hayır” ise, bu makine çıktı içermeyen sonlu durum otomatı olarak adlandırılır. Sonlu durum otomatları sonlu sayıda durum ile sonsuz sayıda giriş dizgisini kontrol edebilir [5]. Herhangi bir sonlu durum otomatı

$$M=(Q, \Sigma, q_0, \delta, A)$$

5-lileri ile tanımlanır. Burada Q mevcut durumların sonlu kümesi,  $\Sigma$  giriş sembollerine ait alfabenin sonlu kümesi,  $q_0 \in Q$  başlangıç durumu, A kabul edilebilir durumların sonlu kümesi,  $\delta$  durumlar arası geçiş fonksiyonudur; bu fonksiyonun tanımına göre hangi sonlu durum otomatının tasarlandığı anlaşılır. Bu çalışmada belirlenimci olmayan bir sonlu durum otomatı (NFA) kullanılacağı için geçiş fonksiyonu

$$\delta: Q \times \Sigma \rightarrow 2^Q$$

şeklinde tanımlanmıştır. Bu tür makinelerde  $\delta$  geçiş fonksiyonunun değerler kümesi Q mevcut durumlar kümesinin bir alt kümesi olur. Kısaca her bir durumda alfabenin sembolleri istenilen sayıda tekrarlanarak, ya da hiç alınmayarak diğer durumlara geçilebilmektedir.

Sonlu durum otomatlarından farklı bir yaklaşımla, herhangi bir dile ait tümceler betimlenmesi için ise bir dizi sonlu kuraldan oluşan dilbilgisi tanımı yapılır. Chomsky Sıradüzeninde<sup>1</sup> tanımlanan tüm dilbilgisi tanımlamaları

$$G=(N, \Sigma, S, P)$$

4-lüleri ile verilir. Burada  $N \cap \Sigma \neq \emptyset$  olmak üzere, N nonterminal sembollerinin sonlu kümesi,  $\Sigma$  terminal yani giriş sembollerinin sonlu kümesi, S başlangıç nonterminali, P ise türetme kurallarıdır. Bu sıralanış içerisinde en alt sınıflandırmada olan düzgün dilbilgisi sonlu durum otomatlarına dönüşebildiği için P türetme kuralları oldukça sınırlıdır.

$A, B \in N$  ve  $x \in \Sigma^*$  olmak üzere,

$$P: A \rightarrow xB \text{ ve } B \rightarrow x$$

koşulları gerçekleşir. Giriş tümcelerinin türetildiği  $\alpha \rightarrow \beta$  biçimindeki türetme kuralları Chomsky tarafından üretici-dönüşümsel dilbilgisi olarak adlandırılmıştır. Chomsky burada sonsuz sayıda giriş tümcesinin sonlu sayıda türetme kuralı ile elde edilebileceğini ifade eder.

<sup>1</sup> Chomsky Sıradüzeni: Tip3 düzgün dilbilgisi sonlu durum otomatlarını, Tip2 bağlamdan bağımsız dilbilgisi son giren ilk çıkar otomatları, Tip1 bağlama duyarlı dilbilgisi lineer sınırlı otomatları, Tip 0 sınırsız dilbilgisi Turing makinelerini oluşturur. Diziliş  $Tip3 \subseteq Tip2 \subseteq Tip1 \subseteq Tip0$  şeklindedir.

$M=(Q,\Sigma,q_0,\delta,A)$ , sonlu durum otomatına ait olan tüm dizgileri tanıyan dil  $L$  olarak simgelendiğinde  $L=L(M)$  şeklinde ifade edilecektir. Bu  $M$  makinesi öyle bir  $G=(N,\Sigma,S,P)$  düzgün dilbilgisine dönüştürülebilir ki,  $G$  dilbilgisi tarafından türetilen tüm dizgiler aynı  $L$  dilini oluşturur ve

$$L(M)=L(G)=L$$

bağıntısı sağlanır.

$M$  makinesinin dilbilgisine dönüşümü

$$G=(\{q_0,q_1,\dots,q_n\},\Sigma,q_0,P)$$

4-lüleri ile tanımlanır.  $q_i, q_j \in Q$  ve  $a \in \Sigma$  olmak üzere  $P$  türetme kuralları

$\delta(q_i,a)=q_j \notin A$  için

$$P: q_i \rightarrow aq_j,$$

$\delta(q_i,a)=q_j \in A$  için

$$P: q_i \rightarrow aq_j, q_i \rightarrow a|\lambda$$

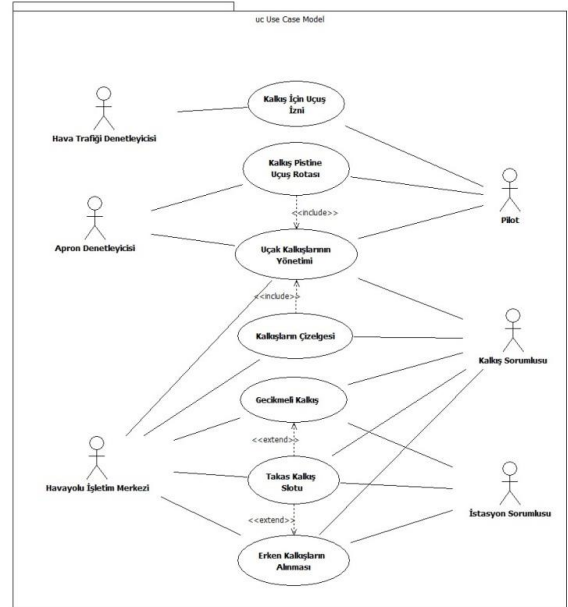
şeklinde  $M$  makinesinden türetilen olacaktır.

### 3. Problemin Tanımı ve Durum Çizgilerinin Çıkarılması

Test işlemini gerçekleştirmek üzere havaalanı uçak kalkış trafiğinin yönetimi problemi incelenecektir. Bu problemin çözümünde uçakların havaalanı kalkış slot<sup>2</sup> rezervasyonlarının gerçekleştirilmesi ve uçak kalkışları için kuyruk sistemi tasarlanmaktadır. Operasyonlardan sorumlu yönetici havaalanı civarındaki uçakların yönetimi ve buralardaki ayrıntılı operasyonlardan sorumludur. Uçak kalkışlarının yönetimi probleminin çözümüne ilişkin *use case*<sup>3</sup> çizgesi Şekil 1'de verilmiştir. Uçak kalkışlarının yönetimi için *havayolu çizelgesi* giriş olarak ön-koşuldur. *use case* çizgesi *Havayolu İşletim Merkezi*, *Apron Denetleyicisi*, *Pilot*, *İstasyon Sorumlusu*, *Kalkış Sorumlusu* aktörleri ile iletişim kurar; amaç uçağı deniz seviyesi üzerine<sup>4</sup> çıkartmaktır. Bu amaca aşağıdaki 7 adımda ulaşılabılır.

- 1.Havaalanı İşletim Merkezi ön çizelgeyi hazırlar ve kalkış slotlarını ister.
- 2.Kalkış sorumlusu kalkış slotlarını belirler.
- 3.İstasyon sorumlusu uçuşlara kapıları atama işlemini gerçekleştirir.
- 4.Kalkış sorumlusu beklenen geri itme zamanını hesaplar.
- 5.Apron denetleyicisi beklenen geri itme zamanına göre pilot ile iletişime geçer.
- 6.Pilot apron kontrolü yönergesini uygulayarak kalkış pistine doğru ilerler.
- 7.Pilot kalkış için hava trafik denetleyicisi ile iletişime geçer.

Uçak kalkışlarının yönetimi *use case* çizgesinin amacı olarak tanımlanan 7 eylemin gerçekleştirilmesi ile *uçak kalkış için hazır* son-koşulu belirlenmiş olur. Uçuşların ertelenmesi ya da planlanandan daha önce gerçekleşmesi olasılığı vardır. Bu durumlarda ön-koşullar bunlar olarak değişecektir. Böylece uçakları deniz seviyesine çıkartma amacı ile düzenlenecek *use case* çizgeleri ya ertelenmiş güvenli bir çıkış slotunun düzenlenmesi ya da planlanandan daha erken güvenli bir çıkış slotunun düzenlenmesi olacaktır. *Tablo 1*'de hem uçağın kalkışının gecikmesi, hem de planlanandan erken gerçekleşmesi ile ilgili eylemler tanımlanmaktadır. *Şekil 1* ve *Tablo 1*'de eklenen senaryolar dışında sisteme erişimin nasıl sağlanacağı ve sistemin altyapısının işleyişi de *use case* çizgeleri ile ayrı ayrı tanımlanmalıdır.



Şekil 1: Uçak Kalkış Yönetimi Use Case Çizgesi

Bir meta-model olan UML<sup>5</sup>[6] ile geliştirilen sistemin davranışsal özelliklerinin *use case* çizgelerine ilave olarak *activity* çizgeleri ve *sequence* çizgelerinin de tasarlanması, davranışsal yapının ayrıntılı olarak çıkarılmasında önemli rol oynar. Gecikmeli ve erken çıkış slotları için güvenli kalkışların atanması süreci, istasyon sorumluları ile kalkış sorumlusu arasındaki iletişimle gerçekleşir; buradaki eylemlerin işleyişi operasyonel *aktivite* çizgeleri ya da *sequence* çizgeleri olarak görselleştirilebilir.

Bu görsel çözümlerden sonra, test işleminde kullanılacak *durum (state chart) çizgeleri* ile davranışsal özellikler açık olarak belirlenmiş olacaktır. *Şekil 2*'de problemin *use case* çizgesine ve ayrıntılı olarak tanımlanan diğer senaryolara göre tasarlanmış kuyruk yönetimi bileşen durum çizgesi görülmektedir.

<sup>2</sup> Slot: Düzenli hava trafiği akışı için uçakların kalkış zamanları ve rotalarının belirlenmesi

<sup>3</sup> Use Case: Kullanım durumu olarak Türkçeleştirilse de, UML çizgelerindeki işlevini tam olarak açıklamaz.

<sup>4</sup> Above sea level

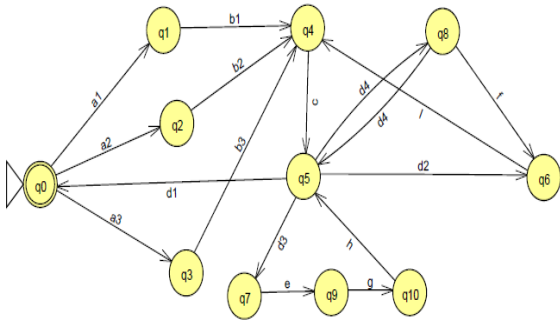
<sup>5</sup> UML ile geliştirilen sistemin yapısal özellikleri ve davranışsal özellikleri görsel olarak tanımlanır. Sistemin yapısal özelliklerine örnek olarak sınıf çizgeleri, nesne çizgeleri, bileşen çizgeleri, paket çizgeleri verilebilir.



işlemini tamamlayıp kuyruğu boşaltmaktadır. Problemin stratejik önemi nedeni ile test sürecinin fazla sayıda senaryo ile gerçekleşmesi mümkün olmayacaktır; ama en önemlisi akışların tümü sıra ile en doğru şekilde sınanacaktır. Tablo 2’den de görüldüğü gibi,  $q_5$  durumuna kadar farklı senaryoların sınanması imkânsızdır. Çünkü hem belirlemeci olmayan sonlu durum otomatından, hem de türetme kurallarından  $q_4$  durumunun başlangıç durumundan dallanan farklı eylemlerin birleştikleri durum olduğu görülmektedir.  $b_1, b_2$  ve  $b_3$  girişleri bu birleşimi gerçekleştirdikleri için aslında  $\lambda$ -geçiş oluşturur. Benzer şekilde  $q_4$  durumundan  $q_5$  durumu gerçekleştiren  $c$  giriş sembolü de bir  $\lambda$ -geçiştir. Çünkü  $q_5$  durumundan  $q_4$  durumuna tekrar geçişin mümkün olabilirdiği iki farklı yol vardır. Bu koşulların test edilebilmesi için  $q_4$  ve  $q_5$  olarak iki farklı durumun tasarlanması gerekmiştir. Sistem  $q_5$  durumunda iken birbirinden farklı dört test yolu izlemek mümkündür. Bu test yollarından sadece Senaryo1 ve Senaryo2 ‘de sistem tekrar  $q_4$  durumuna geri döner. Benzer şekilde tüm durumların ve geçişlerin UML durum çizgesine göre tanımlanması ile belirlemeci olmayan sonlu durum otomatının tasarımı tamamlanır.

Bu durum denklikleri kümesi ile uçak kalkışları için kuyruk yönetiminin tasarlandığı sonlu durum makinesi Şekil 3’de verilmektedir.

$\{a_1, a_2, a_3, b_1, b_2, b_3, c, d_1, d_2, d_3, d_4, e, f, g, h, l\}$  olaylarına göre değişen bir durumdan diğerine geçişlerle uçak kalkış kontrolünü gerçekleştiren sonlu durum makinesi  $\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}\}$  durumlarından oluşur. Belirlemeci olmayan bu sonlu durum otomatının düzgün dilbilgisine dönüşümü Tablo 2’deki dilbilgisi kuralları ile gerçekleşir.



**Şekil 3:** Uçak Kalkış Kuyruklarının Yönetimine ait Sonlu Durum Makinesi

*Senaryo 1:* Sistem  $q_5$  durumunda  $q_5 \rightarrow d_2 q_6$  kuralını türeterek  $q_6 \rightarrow l q_4$  türetme kuralı ile tekrar  $q_4$  durumuna geçer ve  $\lambda$ - geçiş ile  $q_5$  durumuna döner.

*Senaryo 2:* Sistem  $q_5$  durumunda iken karşılıklı olarak  $q_5 \rightarrow d_4 q_8$  ve  $q_8 \rightarrow d_4 q_5$  kuralları türetilir. Eğer geri dönüşte  $q_5$  durumunda kalınıyorsa tüm senaryoların her birinin tekrarlanması mümkündür. Aksi halde  $q_8 \rightarrow$

$f q_6$  ve  $q_6 \rightarrow l q_4$  iteratif türetmeleri ile  $q_4$  durumuna dönülür ve tekrar  $q_5$  durumuna geçilir.

*Senaryo 3:* Sistem  $q_5$  durumunda  $q_5 \rightarrow d_3 q_7$  kuralı ile yeni bir iterasyon başlatır ve sıra ile  $q_7 \rightarrow e q_9$ ,  $q_9 \rightarrow g q_{10}$  ve  $q_{10} \rightarrow h q_5$  türetmeleri ile tekrar  $q_5$  durumuna döner.

*Senaryo 4:* Sistem  $q_5$  durumunda  $q_5 \rightarrow d_1 q_0$  kuralı ile başlangıç durumuna geri döner.

**Tablo 2:** Uçak Kalkış Kuyruklarının Yönetimine ait Düzgün Dilbilgisi kuralları

$q_0 \rightarrow a_1 q_1 \mid a_2 q_2 \mid a_3 q_3$
$q_1 \rightarrow b_1 q_4$
$q_2 \rightarrow b_2 q_4$
$q_3 \rightarrow b_3 q_4$
$q_4 \rightarrow c q_5$
$q_5 \rightarrow d_1 q_0 \mid d_2 q_6 \mid d_3 q_7 \mid d_4 q_8$
$q_6 \rightarrow l q_4$
$q_7 \rightarrow e q_9$
$q_8 \rightarrow d_4 q_5 \mid f q_6$
$q_9 \rightarrow g q_{10}$
$q_{10} \rightarrow h q_5$

Senaryo1, Senaryo2 ve Senaryo3’ün tamamlandıktan sonra, Senaryo 4’i gerçekleştirmesi arzu edilen durumdur; fakat farklı senaryoların tekrarı da olasıdır. Ayrıca  $q_5$  durumunda doğrudan Senaryo4 gerçekleşebilir. Farklı senaryolar daha etkin ve doğru bir test süreci için önemlidir

## 6. Sonuçlar

UML durum çizgeleri genellikle haberleşme protokolleri ya da kullanıcı arayüz sistemleri gibi olay sürücülü sistemlerin davranışlarının betimler. Yazılım testinde temel gereksinim, testi gerçekleştirilen sistemin özelliklerine göre testin uygunluğunun sağlanmasıdır. Model tabanlı ürünlerin test yaklaşımlarında UML durum çizgelerinden ve bunların daha kolay izlenebildiği sonlu durum otomatlarından yararlanılır. Böylece model içerisinde farklı iş akışları ya da yolların sınanması ile testin uygunluğuna güven sağlanacaktır.

## **Kaynakça**

- [1] Gross, H.G. Component-Based Software Testing with UML, Springer, 2005.
- [2] Blaha M., Rumbaugh J.R., Object-Oriented Modeling and Design with UML2. Pearson (2010).
- [3] Dalal S.R., Jain A., Karunanithi N., Leaton J.M., Lott C.M., “Model-Based Testing of a Highly Programmable System”, in Proceedings of ISSRE’98, pp. 174–178.
- [4] Chow T., “Test design modeled by finite-state machines,” IEEE Trans. Software Eng., vol. 4, no. 3, pp. 178–187, 1978.
- [5] Altan Z., Formal Diller ve Soyut Makineler, İstanbul Üniversitesi Yayın No: 4303, 2001”
- [6] Martin F., UML Distilled, Third Edition, A Brief Guide to the Standard Object Modeling Language, 2003.
- [7] Hung D., Anh B. “Model Checking Component Based Systems with Blackbox Testing” United Nations University International Institute for Software Technology UNU-IIST Report No. 317, 2004.