T.C.
HALİÇ ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ
BİLGİSAYAR MÜHENDİSLİĞİ ANABİLİM DALI
BİLGİSAYAR MÜHENDİSLİĞİ PROGRAMI

MODEL DENETLEME TEMELLİ YAZILIM TESTİ VE ANALİZİ YÖNTEMLERİNDE UYGULANABİLİRLİĞİN ARTTIRILMASI

YÜKSEK LİSANS TEZİ

Hazırlayan
Macit Giray GÖKIRMAK

Danışmanı
Prof.Dr. Muhammet KÖKSAL

İstanbul – 2012

ÖNSÖZ

Bu tez çalısmasının hazırlanmasında; değerli fikirleriyle bana yol gösteren danışmanım Prof. Dr. Muhammet KÖKSAL'a, desteklerini ve yardımlarını esirgemeyen tüm çalısma arkadaslarıma, desteğini ve bana olan güvenini her zaman yanımda hissettiğim eşim'e ve son olarak beni bugünlere getiren aileme tesekkürlerimi sunarım.

İstanbul, 2012 Macit Giray GÖKIRMAK

İÇİNDEKİLER

Sayfa No.

KISALTMALAR .... **III**
TABLO LİSTESİ .... **IV**
ŞEKİL LİSTESİ .... **V**
ÖZET .... **VI**
ABSTRACT .... **VII**
1.GİRİŞ **1**
1.1.**P**roblemin Sunumu **2**
2.YAZILIM TESTİ HAKKINDA GENEL BİLGİLER **4**
2.1.**Y**azılım Testi Nedir? **4**
2.2.**Y**azılım Testlerinde Otomasyon Nedir? **4**
2.3.**Y**azılım Testi Temelleri **5**
2.4.**Y**azılım Testlerinin Geçerliliğinin Değerlendirilmesi **7**
2.4.1.**K**apsama Analizi: **8**
2.4.2.**M**utasyon Analizi: **8**
2.4.3.**F**ormal Test **9**
3.MODEL DENETLEYİCİ TEMELLİ YAZILIM TESTLERİ **11**
3.1.**G**iriş **11**
3.2.**M**odel Denetleyici **12**
3.3.**Z**amansal Mantıklar **14**
3.3.1.**L**TL (Doğrusal Zaman Mantığı) **15**
3.3.2.**C**TL (Hesaplama Ağacı Mantığı) **17**
4.YAZILIM TESTLERİNİN YAYGINLAŞTIRILMASI **18**
4.1.**T**ers Örnek Üretimi **18**
4.1.1.**M**utasyon Operatörleri Kullanımı İle Test Durumu Üretimi **18**
4.2.**D**urum Uzayının Daraltılması **20**
4.2.1.**S**ınırlı Odak **21**
4.2.2.**B**ileşen Etkileşimi **23**
5.VAKA ÇALIŞMASI **26**
5.1.**S**istem ve Model **26**
5.2.**T**est Durumları Üretimi **27**
5.3.**T**estin Yürütülmesi **28**
5.4.**D**eğerlendirme **29**
6.SONUÇLAR **31**
7.KAYNAKLAR **32**
8.ÖZ GEÇMİŞ **34**

KISALTMALAR

ADL **:**Mimari ifadeler dili (Architectural Description Language).
ALLOY **:**Alloy, nesne yönelimli diller için 1990'lı yılların sonunda kabul görmüş komplike yapısal kısıtları ifade edebilmekte kullanılan şartname dilidir.
ATM **:**Otomatik bildirim makinesi, bankamatik adıylada anılan hesap bilgileri kontrolü, para çekme gibi temel bankacılık işlemlerinin yürütülmesi amacıyla üretilmiş bilgisayar sistemi (Automated Teller Machine).
BNF **:**Dil tanımlamada ve o dilin söz dizim kurallarını belirtmekte kullanılır. BNF'in iki farklı açılımı vardır, Backus Naur Form ve Backus Normal Form olarak anılmaktadır.
CLP **:**Kısıt mantığı programlama dili, değişken ve durumlar arası ilişkilerin tanımlarının yapılabilmesini sağlayan programlama yöntemi (Constraint Logic Programming).
CTL **:**Hesaplama ağacı mantığı (Computational Tree Logic).
EEPROM **:**Elektronik olarak silinebilir ve yeniden programlanabilir sadece okunur bellek (Electronically Erasable and Programmable Read Only Memory).
FSM **:**Sonlu durum makinesi (Finite State Machine).
HML **:**M. Hennessy ve R. Milner tarafından ortaya konulan bir zamansal mantık algısı (Hennessy Milner Logic).
LTL **:**Doğrusal zaman mantığı (Linear Temporal Logic).
LTS **:**Etiketli geçiş/dönüşüm sistemleri, sonlu durum makinelerinde durumlar arası dönüşümleri takip amaçlı analiz sistemleridir (Labeled Transition Systems).
PIN **:**Kişisel kimlik numarası (Personal Identification Number).
PTL **:**Önerme zamansal mantığı (Prepositional Temporal Logic).
RAM **:**Rastgele erişimli bellek (Random Access Memory).
ROM **:**Sadece okunabilir bellek (Read Only Memory).
SMV **:**Sembolik model doğrulayıcı, modelin belirli bir dizi kurallara göre kodlamaya aktarılarak doğrulamasını yapabilen uygulama birimi (Symbolic Model Verifier).
SOD **:**Sınırlı odak ile daraltım, zamansal mantık kısıtlarının sonlu durum makinesi üzerine uygulanışı ile durum uzayının daraltılmasını amaçlayan yöntem.
TL **:**Zamansal mantık (Temporal Logic).
UML **:**Birleşik modelleme dili, yazılım sisteminin önemli bileşenlerini tanımlamayı, tasarlamayı ve dokümantasyonunu sağlayan grafiksel bir modelleme dilidir. (Unified Modelling Language).
VDM-SL: Viyana geliştirme metodu şartname dili, nesne yönelimli diller için IBM firması tarafından 1970'li yıllarda geliştirilmiş ve yaygın olarak kabul görmüş bilgisayar sistemleri geliştirme metodu (Vienna Development Method Specification Language).
WAP **:**Kablosuz uyglamalar protokolü (Wireless Application Protocol).
WIM **:**Kablosuz uygulamalar iletişim protokolü kimlik modülü (Wireless application protocol Identity Module).
YKY **:**Sınırlı odak ile daraltım'ın oluşması için gerekli olan yeni zamansal mantık kısıtları için yeni kısıt yazım yöntemi.
Z **:**Z notasyonu, Ernst Zermelo tarafından bulunmuş olan set teorisine dayalı bir formal modelleme dilidir.

TABLO LİSTESİ

Sayfa No.

Tablo 5.1 Test sonuçları istatistiği özeti 30

ŞEKİL LİSTESİ

Sayfa No.

Şekil 2.1 Örnek mutant 9

Şekil 3.1 Örnek sonlu durum makinesi ve karşılığı olan kripke yapısı 12

Şekil 3.2 Örnek bir otomasyonun yürütme ağacı şeması.. 13

Şekil 3.3 Zamansal Mantıklar 15

Şekil 3.4 Örnek ilişki şeması... 16

Şekil 3.5 Genişletilmiş ilişki şeması 17

Şekil 4.1 Mutasyon operatörleri kullanımı ile test durumu üretimi......................... 20

Şekil 4.2 Şartname dönüşümleri 22

Şekil 4.3 Bileşen etkileşimi için test seçim tekniği.. 23

GENEL BİLGİLER

Adı ve Soyadı : Macit Giray GÖKIRMAK
Anabilim Dalı : Bilgisayar Mühendisliği
Programı : Bilgisayar Mühendisliği
Tez Danışmanı : Prof. Dr. Muhammet KÖKSAL

Tez Türü ve Tarihi : Yüksek Lisans – Haziran 2012

MODEL DENETLEME TEMELLİ YAZILIM TEST VE TEST ANALİZİ YÖNTEMLERİNİN UYGULANABİLİRLİĞİNİN ARTTIRILMASI

ÖZET

Projelerde kaliteli yazılım geliştirmeyi sağlamanın tek etkili yolu yazılan gereksinimlerin ve kodların proje dışı kaynaklar tarafından teste tabi tutulmalarıdır. Ancak, test aşaması yazılım projelerinde geliştirmeden sonra gerek duyulan insan kaynağı nedeni ile yüksek maliyetlere katlanılmayı gerektiren bir aşamadır. Testleri daha az kaynak ile yapmak amacıyla otomatikleştirme ihtiyacı doğmaktadır. Bu alanda umut vaat edici gelişmelerden biri olarak model denetleme araçları ile yapılan analiz ve üretilen test durumlarının uygulanması gibi tekniklerden bahsedebiliriz. Model denetleyiciler tarafından üretilen çıktılar doğrudan test durumu olarak kullanılmaya uygundurlar. Hali hazırda yazılım testleri otomasyonu yapılabilinmekte olsa da bu durum tüm projeler üzerinden bakıldığında oldukça ufak bir kesime hitap etmekte, projelerdeki girdi çeşitliliği nedeniyle tam anlamıyla endüstriyel bir seçenek haline gelememektedir. Model denetleyici temelli testlerin uygulanmasındaki en büyük kısıt kullanımda olan araç gerecin çıktılarının mevcut performansının yeterli olmayışıdır. Performans kısıtlarının bu denli fazla olmasının sebebi ise genellikle projelerde çok fazla test durumu tanımı oluşturulmasıdır. Gereksinimler bu araçlara aktarıldığında, genellikle mükerrer test durumları oluştuğu gibi bazı durumlarda ise tamamen gereksiz test durumları oluşturulmasında söz konusu olmaktadır.

Bu tez ile yazılım geliştirme projelerine sağlanacak katkı sayesinde mevcut test durumları oluşturma çalışmalarındaki performans kayıpları ölçülebilir derecede azalarak maliyetlerde büyük düşüşlere imkan sağlanmıştır. Bir diğer husus ise test durumları oluşturma çalışmaları basit ve otomatik olarak yapılabilmekte ancak sonuçları çok verimli olmamaktadır. Aşırı mükerrer test durumlarının varlığı hata toleransını olumsuz etkilemektedir. Bazı teknikler ise büyük kapsamlı test senaryoları içerisinde çok kısa ve yetersiz test durumları oluşturulması gibi sonuçlar doğurmaktadır. Bu ise oldukça verimsiz bir sonuç doğurmaktadır. Bu tezde model denetleyici araçlar ile oluşturulan test senaryolarının daha verimli ve aynı zamanda proje ihtiyaçlarını en iyi şekilde karşılayacak biçimde uygulanışı için farklı tekniklerin birleşimi ele alınacaktır, çalışma sonucu ortaya çıkan maliyet düşüşleri projelerdeki karlılığı daha yüksek oranlara çekecektir.

Anahtar Kelimeler: Yazılım test, senaryo, verimlilik, maliyet, karlılık.

GENERAL KNOWLEDGE

Name and Surname	: Macit Giray GÖKIRMAK
Field	: Computer Engineering
Program	: Computer Engineering
Supervisor	: Prof. Dr. Muhammet KÖKSAL
Degree Awarded and Date	: Master – June 2012

INCREASING APPLICABILITY OF MODEL-CHECKING BASED SOFTWARE TESTING AND TEST ANALYSIS METHODS

ABSTRACT

Software tests being performed by third party resources are the only effective way of providing quality software development projects. However, the test phase of the projects have the highest cost after the development phase. This arises the need to automate, inorder to perform tests with fewer resources. One of the promising developments in this field is model checker based analysis and test case production. Outputs produced by model checkers can directly be used as test cases. Although automation of software testing can be done already but applicable projects through all the projects are quite small percent of the masses. Because of this situation this has not become an industrial option for the masses. The biggest constraint of applicability of model checker based testing is the lack of adequate performance. The reason for this constraint is that tools produce repeated test case definitions oftenly. Software requirements transferred to these tools usually produce too often repeated and useless test case definitions from time to time. Touch of this thesis will provide measurable performance gains and reductions of the project costs with production of optimized test case definitions. On the other hand, test cases can already be produced automatically with other techniques but the results of the work are generally very simple and requirements are not fully covered. The presence of repeated test case definitions extremely affects fault tolerance negatively for the test performed. Some techniques generate large-scale scenarios but very low coverage and very inadequate results for test cases. This result rises a rather inefficiency. In this thesis, results will prove that with the combination of different techniques used to improve model checker based analysis and testing applicability to cover the needs of the projects and increase efficiency. Results from this operation will cause cost reductions and improvement in profitability.

Keywords: software tests, model-checkers, efficiency, reduced costs, profitability.

1. GİRİŞ

Test, yazılım süreçlerinin önemli parçalarından biri olarak ele alınmalıdır. Harcanan eforların toplama oranla %50'sinin test ve hata ayıklama çalışmalarına harcandığı bilinmektedir (*J. Myers, 1979; B. Korel, 1990:16/870–879*). Test, yazılımın istenilen sonuçları ortaya doğru biçimde koyabildiğinin teyid edilebilmesi açısından gereklidir. Şartnameye uyum açısından yapılacak kontroller alternatif olarak değerlendirilmelir. Şartnameye uyumu kontrol eden metodlar doğruluğu garanti etse dahi sadece belirli bir kesim projelerde uygulanabilmekte ve otomasyona müsade etmemektedir. Yazılım testleri otomasyona müsait, yapılan hesaplamalar açısından daha az maliyetli ve hataların tespiti açısından daha verimli, ancak tüm yazılımın doğruluğu sağlama noktasında kesin ifadeler sunamayan sadece doğruluğa olan güvenin artmasıyla sonuçlanan çalışmalardır. Hiçbir yazılım %100 hatasız yazılamayacağı gibi hiçbir test sürecinde hataların tümü yakalanamaz. Eğer testler çok kapsamlı şekilde enine boyuna yapılabiliniyor olsa idi kesin doğruluğa yaklaşım limitlerine ulaşılabilinirdi, ancak enine boyuna çok kapsamlı testler uygulamak istenildiğinde ortaya çıkacak test durumları ele avuca sığmayacak duruma gelebilir ve çoğu projede mevcut olan zaman/maliyet kısıtları nedeniyle tatbik edilebilir değildir.

Yazılımda tespit edilen hataların etkileri kimi zaman sadece yazılımın çökmesine neden olacak görsel sorunlar olurken kimi zaman ise hayati tehlikeler oluşturacak noktaya erişebilir. Taşımacılık ya da sağlık sektöründe kullanılacak bir yazılım bu duruma örnek olabilir. Düşününki otomobillerin elektronik beyinlerinde işlemekte olan bir yazılımdaki hata yüksek hızlarda meydana geldiğinde ya da hastanelerde yaygın biçimde kullanılan radyoterapi cihazlarında hedef alınan tümör yerine sağlıklı dokulara radyasyon verilmesine sebep olabilecek bir hata meydana geldiğinde ölümcül sonuçlar doğurduğu görülmüştür (*S. Baase, 2008*).

Yazılım testi önemlidir, ancak pek çok problemin projeye dahil edilmesini ve bu problemlerle başedilmesini gerektirir. Test karmaşık ve oldukça uzun zaman alan bir süreçtir. Genel kanı olarak karmaşık işlerin insan gücü ile yapılması, bu işlerin hiç sonuçlanmamasına ya da sonuçlansa dahi hatalar içermesine neden olmaktadır. Bu durum otomasyon metodları ile aşılabilir, ancak yazılım projelerinde otomasyon metodlarının uygulanması zor ve tüm süreci buna uygun yürütmüş olmayı gerektirir.

Buna ek olarak, otomasyona rağmen enine boyuna kapsamlı bir testin yapılması sonsuz seçenekli test durumlarından oluşabileceğinden genellikle mümkün değildir.

Görüldüğü gibi yazılım testlerine ilişkin problemlerin listesi oldukça kabarık. Testin kendisinin zorluklar içermesinin dışında karşılaşılabilecek problemlerin tamamının tespiti ise neredeyse mümkün değildir. Bu tezde bazı problemlerin üstesinden gelinecek özgün çözüm öneriler ispatları ile birlikte ele alınmıştır. Tezdeki amaç, güvenliğin en büyük öncelik olduğu bir senaryo tabanında gereksinimlere doğrudan bağımlı test durumlarının otomatik bir süreç ile oluşturulması için bir prototip oluşturmaktır. Bu prototip yazılım endüstrisi tarafından benimsendiği takdirde birçok alanda kullanılmak üzere hazırlanan projelerde uygulanabilir ve proje maliyetleri oldukça aşağıya çekebilir.

1.1.Problemin Sunumu

Bu tezde ele alınmakta olan problem bir teknik özellik şartname ve gereksinim belgesi ışığında analiz yapılarak model denetleme metodu ile otomatik test senaryoları ve test durumları oluşturulmasıdır. Bu duruma uygun çeşitli çözümler daha önce düşünülmüş olsa da endüstriyel anlamda kabul görmüş bir çözüm halen bulunmamaktadır. Bu durumun asıl sebebi bu metodun kullanımında özetle aşağıdaki gibi handikaplar bulunmasıdır:

- Gereksinim temelli test: Model denetleyici temelli testlerde halihazırda uygulama yönteminin otomasyona özel tüm durumlara uygunluğu hedef alınmaktadır. Gereksinim temelli test alanında literatürce ihmal edilmiş çeşitli sorunlar bulunmaktadır. Örneğin, izlenebilirlik her satır kodun olduğu gibi her bir test durumunda da bir gereksinimin özelliği ile bağlantılı olması gerektirir. Bu durum test durumları oluşturulmasında ve analiz adımlarında nasıl ele alınabilir?

- Test durumları oluşturulma performansı: Test durumlarının otomatik oluşturulması oldukça karmaşık bir iş olduğu gibi bu konuda performans ise önemli bir sıkıntıdır. Model denetleyici araçların performanslarının kestirimi oldukça zor ve bir okadar da sorunludur. Test durumları oluşturulması nasıl ideal bir hale getirilebilinir?

- Testlerin yürütülme performansı: Otomatik oluşturulmuş test durum setleri genellikle çok kapsamlı ve kötü biçimlendirilmiştir. Test durumları verimli bir şekilde oluşturulmuş olsalar dahi verimli bir şekilde yürütülemedikten sonra uygulanabilir değildir. Test durum setleri nasıl uygulamada ideal bir hale getirilebilinir? Öncelikle verimli test durum setleri oluşturmak için test durum üretimi nasıl ideal hale getirilebilinir?

- Model denetleyici temelli testin kısıtları: Model denetleyici araçlar ile yapılan testler güncel literatürde altta yatan teknik kısıtlar, şartnameler üzerinden varsayım yapar. Örneğin, tüm özellikler deterministik olarak kabul edilir. Deterministik olan bu ortak varsayımlar model denetleyici temelli testlerin uygulanabilirliğinin arttırılabilmesi için ne şekilde gevşetilebilir?

Model denetleyici temelli test ve test analizi yöntemlerinin uygulanabilirliği, performans ve test durumları oluşturulması aşamasında kalitenin arttırılması ve pratikte birbiri ile ilişkili senaryoların kullanımı ile mümkündür.

2. YAZILIM TESTİ HAKKINDA GENEL BİLGİLER

2.1. Yazılım Testi Nedir?

Yararlı geliştirme araçları ve geliştirme süreçleri tanımlanmış yaygınlaşmış olmasına rağmen, yazılım geliştirme, büyük ölçüde manuel bir süreç olmaya devam etmektedir . Bu nedenle, yazılım üretilirken hatalar yapılır. Kullanıcı gereksinimleri, sistem tasarımı ihlal ediliyor olabilir, yanlış yorumlanıyor ya da belki programcı sadece özensiz bir hata yapıyor olabilir. Programlama hatalarının kaynağı boldur.

Yazılım testlerinin amacı geliştirme sürecindeki hataların bir yazılım parçası (hata takip araçları) ile etkileşim süreci içerisinde ifşa edilmesidir (*A. Abran; J.W. Moore, 2004:5/5-1*). Test edilecek yazılımın nasıl test edileceğinin belirlenmesinde yazılımın hangi tip bir yazılım olduğunun çok büyük etkisi vardır. Örneğin, bir grafik kullanıcı arayüzü testi ile gömülü bir sistemin daha farklı bir test yöntemine ihtiyacı vardır. Kapsamı büyük boyutlu bir yazılım genellikle senaryolar için olası yolların çokluğu bakımından içinden çıkılmaz büyük bir test durumları seti sunar. Bu nedenle, yazılım testlerindeki ana zorluk hangi davranışların tam olarak test edileceği için karar vermektir. Bu denemeler sırasında gözlenen davranışların yanlış olup olmadığı konusunda karar vermek zordur.

2.2. Yazılım Testlerinde Otomasyon Nedir?

Verilen bir yazılım parçasının doğru olup olmadığının belirlenmesinde karmaşıklık yüksektir. Geleneksel olarak, yazılım testleri tek tük ve manüel olarak yapılır. Endüstriyel ortamlarda sistematik bir yaklaşım gereklidir; örneğin özellikle güvenlik ile ilgili yazılım geliştirmesi yapıldığında sistematik ve izlenebilir bir yaklaşım esastır. Sektördeki mevcut uygulama, test durumları düzenlemek ve yürütmek için otomatikleştirilmiş araçlar kullanmaktır. Ancak, test durumlarının otomatik oluşturulması yaygın değildir. Otomasyonun gerekliliğinin pek çok nedenleri vardır: test durumları oluşturmak, elle zahmetli ve hata eğilimli oldukça yüksek bir süreçtir.

Otomasyon, geliştirme süreci açısından önemli bir gelişme vaat etmektedir ve sürecin en zaman alıcı kısımlarından birini destekler. Sistematik olarak uygulandığı

için aynı zamanda otomasyon genellikle daha fazla kapsamı olan test durum setleri üretimiyle sonuçlanır.

Yazılım test otomasyonundan söz edilirken çözülmesi gereken birkaç ciddi sorun vardır. Örneğin, bir dizi test vakalarının yürütülmesi kolaylıkla otomatik yapılabilir. Dikkate alınması gereken iki ana konu vardır: Öncelik test veri seçimidir. Bu veri, sistem testi altında bir girdi olarak kullanılır. İkinci sorun, bir test durumu yürütülmesi sürecinde bir hata algılandığında bu bir hata mıdır değil midir karar verebilmektir. Bu test kahini sorunu olarak adlandırılır. İdeal olarak, her iki sorunda da gerçekten faydalı bir uygulama yapılabilmesi için otomatik bir teknikle çözülmesi gerekir.

Yazılım test araştırmaları test durum üretimi otomasyonu için birkaç farklı çözüm önermiştir. Bunların çoğu gerçek uygulamalar üzerinde değerlendirilmiştir ve hatta bazı ticari ürünler şu anda kullanılmaktadır. Ancak, yaygın halde kabul edilebilmesinden önce çözülmesi gereken hala pek çok konu vardır.

2.3. Yazılım Testi Temelleri

Yazılım testi, doğrulama ve onaylama amacıyla yapılır. Onaylama, doğru yazılımın inşa edilmiş olup olmadığının belirlenmesi sürecini ifade etmektedir. Bunun için, gereksinim dokümanları yazılım yapması gerekenlerin ne olduğunu açıklar. Buna karşılık, doğrulama, tasarım açısından doğru olup olmadığının, yazılımın doğru yani kullanıcının ihtiyaçları doğrultusunda inşa edilmiş olup olmadığının belirlenmesi sürecini anlatılmaktadır. Yazılım testlerinin tüm hataların tespitini kapsayacak şekilde yapılması mümkün değildir. Bu nedenle yapılan testlerin temel amacı olabildiğince çok hatayı tespit etmekten öteye gidememektedir.

Bazen tutarsız hata, hata, arıza, böcek veya kusur şekilde ortak terimler kullanılır. Mevcut literatür bu tür terimlerin farklı tanımları ile doludur.

Karışıklığı önlemek için, Bu tezde en tartışmalı terimleri kullanmaktan kaçınmak için aşağıdaki tanımlar kullanılacaktır:

- Arıza, yazılımdan beklenmekte olan fayda ya da sonuçtan sapmaları ifade ederken,
- Hata ise bu tür arızaların oluşumundaki sebep olarak ifade edilecektir (*M. Grindal ve B. Lindström, 2002*)

- Tutarsız hata ise hata ile eş anlamlı olarak kullanılacaktır.

Bir hatanın pek çok farklı yüzü olabilir. Örneğin, John B. Goodenough ve Susan L. Gerhart performans ve mantıksal hataları ayırt etmiştir (*J.B. Goodenough ve S.L. Gerhart, 1975*). Öte yandan, yanlış zamanlama ve yanlış işlevsellik eski birer sorundur.

Goodenough ve Gerhart farklı tipteki mantıksal hataların, uygulamaların şartnamelere göre yapılmayışı, tasarımı tam olarak yansıtmayışları veya tasarımın gereksinimlerini tam olarak karşılamayışının sonucu şeklinde ayrı ayrı guruplandırılmalarını önermektedirler. Hataların, detaylı sınıflandırmaları, Beizer'in tanınmış klasik kitabında belirtilmiştir (*B. Beizer, 1990*). Bu hata türlerinin her biri farklı şekillerde tezahür edebilir.

Bazı hatalar diğerlerine göre daha karmaşıktır, ve bazı hataların etkisi diğer hataların etkilerinden daha şiddetli olabilir. Test genellikle belirli türde hataların bulunmasına odaklanmak durumunda değildir. Çünkü bağlaşım etkisi (*R. Demillo, 1978*) belirtmektedir ki bazı çok kompleks hatalar görece daha basit hatalarla bağlantılı olabilmektedir. Bu sebep ile bir fonksiyonun testi yapılırken bir hata tespit edildiğinde aynı fonksiyonun devam eden adımlarında yürütülmesi planlanan test durumları işletilmemelidir. Sonuç olarak, karmaşık hataları tespit etmek için öncelikle basit hataları test etmek yeterlidir.

Hataların birçok farklı türde olduğu gibi, testlerinde birçok farklı türleri vardır. Birçok özellik yaklaşımları ayırt etmek için kullanılabilir. Örneğin, testlerinin genel amacı büyük ölçüde değişebilir:

- Performans testi yazılımın zamanlama gereksinimleri açısından doğru olup olmadığını belirlemeye çalışır "yeterince hızlı mı?".
- Stres testi, ağır yük altında sistem davranışını değerlendirir.
- Güvenilirlik testleri, bir sistemin daha uzun bir süre boyunca doğru davrandığını belirler.
- Fonksiyonel test sistemin, tasarım ya da şartnameye göre doğru bir yapıda olup olmadığını inceler.

Bu liste, test hedefleri açısından ayrıntılı bir liste değildir, ama çeşitliliği göstermek açısından azda olsa bir anlam taşımaktadır. Test tekniklerini farklılaştırmak için daha farklı bir yaklaşım olarak soyutlama testleri uygulanır:

- Fonksiyon testleri, program kodunun en küçük birimine uygulanır.
- Modül testi bir sistemin farklı bileşenlerinin (Bileşen testi) uygunluk kontrolü amacıyla uygulanır.
- Sistem testleri, adındanda zaten anlaşılacağı gibi komple bir sisteme uygulanır. Genellikle, entegrasyon testi ve arayüz testi gibi ilgili teknikler ile desteklenmektedir.

Test tekniklerinin önemli bir kategorizasyonu, kara kutu ve beyaz kutu testi arasındaki ayrımdır. Kara Kutu Testleri, bir kara kutu olarak test edilen sistemin, sadece girişleri ve beklenen çıktıları dikkate alınarak yapılan testlerdir. Bu tür testler de fonksiyonel testler olarak bilinir. Buna karşılık, beyaz kutu testi veya yapısal testler, gerçek kaynak kodunu dikkate alır. Yapısal test, kodun bazı bölümlerini çalıştırmak üzere oluşturulan test durumlarından meydana gelir. Fonksiyonel testlerin aksine tüm program kodunu test etmek mümkündür. Ancak, fonksiyonel test eksik işlevselliği tespit etmek için bir potansiyele sahiptir. Genellikle, bu tür yaklaşımların bir kombinasyonu, bir sistemin modül yapısı dikkate alınarak fonksiyonel testleri oluştururken kullanılır. Bu tür testler gri kutu testleri şekilde adlandırılır.

Olası sınıflandırmalar listesi daha devam ettirilebilir. Literatürde pek çok farklı test teknikleri sınıflandırması vardır, örnek vermek gerekir ise Baiser'in yazılım tesi üzerine klasikleşmiş kitabı verilebilir (Baizer, 1990). Tam bir kategorizasyon yapmak mümkün değildir çünki, tezde yer alan test teknikleri farklı test methodlarına, ancak altında yatan modele uygun olacak şekilde uygulanabilir.

2.4. Yazılım Testlerinin Geçerliliğinin Değerlendirilmesi

Testlerin asıl hedefi olan hataların tespiti için; en temel sorun ne zaman duracağını bilmektir. Ancak mümkün olan bütün testlerin yürütülmesi ile artık tespit edilememiş hataların kalmadığı söylenebilir. Olası testlerin sayısı genellikle çok fazla, hatta sonsuz olduğundan, bu pratikte mümkün değildir. Bu nedenle, testlerin titizliği için bazı ölçümler gereklidir. Testi yapan bu ölçümleri kullanarak yaptığı testlerin yeterliliğinden emin olabilir. Test durumlarının bir dizi değerlendirme için kullanılan en yaygın iki tekniği, kapsama analizi ve mutasyon analizidir. Bu iki yöntemde deterministik yöntemlerdir, bu üzerinde çalışılacakları ifade etmektedir (örn: kapsama öğeleri, mutantlar, girdi bölümleri), gerekli her öğe için en az bir test

durumu olup olmadığını kontrol etmektir. Aksine, istatistiksel yöntemler test durumlarının seçiminde olasılık dağılımının kullanılmasını gerektirir ve sistem güvenilirliği testi altında tahmini için olanak sağlar. Bu tez sadece deterministik analiz yöntemlerini değerlendirecektir.

2.4.1. Kapsama Analizi:

Kapsama analizi test altındaki sistemin bazı yönlerini kullanarak testlerin hangi yönler üzerinde yoğunlaştırılması gerektiğini ölçümler. Çalışacak sistemin gerçek özellikleri bir kapsama kriterine göre belirlenmiş olmalıdır. Örneğin, bildirimdeki kapsama kriteri, her bir program koduna karşılık olarak, testler sırasında en az bir test durumu yürütülmesini gerektirebilir. Bu durumda uygulanacak testlerde her bir satır çalıştırılıyor ise bu test
kapsanmış sayılır. Kapsam her bir satır kod ya da fonksiyonel kod kümeleri arasından test içerisinde incelenecek olanların seçimi ile oluşan yüzde oranında belirlenir. Eğer yazılan kodun her bir satırı test sırasında işletilmek zorunda ise bu durumda testin kapsamı %100 olarak belirtilmelidir. Bu oran kara kutu test yapıldığında bile, yapısal bir test durumunun başarısını ölçmek için kullanılabilir. Çeşitli kapsama kriterleri yazılım testlerinin ilk başladığı günlerden beri önerilmektedir. Kaynak kodlarına dayalı kapsama kriterleri en yaygın olarak benimsenenlerdir. Böyle bir kriteri ölçümlemek herzaman daha kolay olmuştur ve bu ölçümlemeler için kullanımı kolay yaygınlaşmış araç gereçler mevcuttur. Yukarıda sözü edilen kapsam böyle bir kritere örnektir. Kod bazlı kapsama kriteri, basit bir bildiri şeklinde ya da program girişinden çıkışına kadar olası tüm ihtimalleri kapsayan bir yol haritası şeklinde olabilir. Diğer kod bazlı kapsama kriterleri değişkenlerin yaşam süreci (veri akış kapsamı), döngüler, nesne kodlarını kapsar biçimde olabilir. Bu kapsamı ölçmek için bir resmi model veya şartname kullanmak da mümkündür. Bu kapsama kriterlerinin özellikleri, kullanılan gerçek resmi modele bağlıdır.

2.4.2. Mutasyon Analizi:

Mutant, bir programın sözdizimsel eşdeğeridir(bkz. Şekil 1 Örnek Mutant). Sözdizimsel değişiklik büyük olasılıkla orijinal program aksine farklı bir davranışa

yol açar. Bu mutantlar test durumlarının geçerliliğini değerlendirmek için kullanılır. Bir test, mutant ve orijinal program arasındaki farkı ayırt edebilir ise mutant öldürülür. Mutant analizinin ana fikri gerçekçi hataları temsil eden mutant setleri oluşturmaktır. Mutantlar, orijinal program kodlarına mutasyon operatörleri uygulayarak oluşturulur. Her mutasyon operatörü, farklı bir hata sınıfı ve her mutant operatörü uygulanışı farklı bir mutant sonucunu temsil eder. Mutasyon puanı başarı oranı olarak değerlendirilir. Yüksek mutasyon skoru yüksek bir hata duyarlılık temsilcisidir. Mutasyon testi terimi belirli bir mutasyon skoruna ulaşabilinmesi için test durumlarının sağlanması tekniklerini ifade eder; sonuçta elde edilen test paketi mutasyon gereksinimlerini karşılayacak kadar yeterlidir. Kapsama ölçümü aksine, mutasyon analizi henüz yaygın ticari kullanımda kabul görmüş değildir. Eşdeğer mutantların tespiti, pratik kullanımda mutant testinin önündeki en büyük engeldir. (*A.Jefferson Offut, 1995*) Mutantların eşdeğer olup olmadıklarının kontrolü küçük programlar için dahi oldukça çok efor gerektirir (*P. G. Frankl, S. N. Weiss, C. Hu., 1997*). Örneğin, bir mutant ile orjinal programın maliyeti büyük tartışma konusudur (*T.A. Budd ve D. Angluin, 1982*).

```
if (a && B) {
  c = 1;
  } else {
  c = 0;
  }
```

(a) Orjinal kod parçacığı

```
if (a || B) {
  c = 1;
  } else {
  c = 0;
  }
```

(b) Mutant operatörü ile && yerine || yerleştirilen mutant kod parçacığı

Şekil 2.1 Örnek mutant

2.4.3. Formal Test

Geleneksel olarak, testler plansız ve herhangi bir teorik dayanaklarıma olmadan yapılır. Formal belirtime dayalı testler genellikle uygulamanın şartnameye uygun olup olmadığının resmi olarak "tespitini" hedefler. Bu nedenle, bu tür test uygunluk testleri olarak bilinir. Uygunluk tespiti testlerinde en yaygın uygulama yöntemi protokol testleridir, ancak reaktif sistem testleride oldukça benzerdir. Uygunluk testlerinde yaygın olarak benimsenen çatı modeli J. Tretmans tarafından oluşturulmuştur (*J. Tretmans, 1999*). Bununla bağlantılı olarak daha önce cebirsel temele dayanan bir çatı modeli M.C. Gaudel tarafından ortaya konulmuştur (*M.C.*

Gaudel, 1995). Bir şartname ve uygulama arasındaki ilişkiyi resmi olarak işlemek amacıyla, test hipotezi resmi bir uygulama modeli var olduğu varsayımına dayanır; bu model gerçekte var olmayabilir. Test hipotezi şartname ve uygulama arasında resmi bir ilişki, uygulama ilişkisi formüle eder. Uygulama, test durumları yürütülmesi aracılığı ile incelenmektedir. Bu icra ve gözlemlerin sonucu, formal bir test yürütme prosedürüne modellenmiştir. Çalıştırıldıktan sonra bir karar fonksiyonu gözlemlerin doğruluğunu ve dolayısı ile tüm uygulamaya ait test paketinin başarılı ya da başarısız sonuçlandığını belirler.

Uygunluk testlerinin hedefinde ise testin uygulanması sırasında elde edilen çıktılar incelenerek uygulamanın şartnamesinde belirtilen durumlara uygunluğunun kontrolünün yapılması vardır. Uygulamanın şartnamesine özel olarak bir test paketi olması ve bu test paketinin başarıyla sonuçlandırılması sonucu uygulamanın uygunluğunun onaylanması gerçekten güzel olurdu ve bu "Tam kapsamlı bir test paketi"dir denilebilirdi. Ancak bu durum pratikte mümkün değildir, çünki "Tam kapsamlı bir test paketi" sonsuz sayıdaki test durumlarından oluşmalıdır.

Pratikte daha zayıf ilişkiler formüle edilmektedir. Test paketi, eğer testin herhangi bir adımında hata alınmış ise şartnameye uygun olmadığını kanıtlamaktadır. Bununla beraber geçer sonuç almış fakat halen şartnameyi karşılamayan durumlar olması da mümkündür. Uygun olmayan uygulamaları tespit ancak ayrıntılı bir test paketi ile elde edilebilir; ancak dikkat edilmelidir ki, ayrıntılı bir test paketi kabul uygunluk ilişkisine bağlı olarak sağlıksız test durumları içerebilir. Test, ancak tüm durumlar sorunsuz ve yeterince ayrıntılı ise tamamlanmış sayılır.

Test çerçevesinin önemli bir kısmı test türetimidir. Test durumları üretimi için kullanılacak herhangi bir algoritma yalnızca anlamlı test durumları üretmelidir.

Bu teorik çerçeve, belirli bir modelleme biçimcilik konusunda örneklenmiş olmalıdır. Tretmans, etiketli geçiş sistemlerini kullanmıştır (*J. Tretmans, 1996*). Test durumları seçimi resmi olarak iz ön emirleri veya test ön emirleri gibi belirli uygulama ilişkileri tarafından yönlendirilir.

3. MODEL DENETLEYİCİ TEMELLİ YAZILIM TESTLERİ

3.1. Giriş

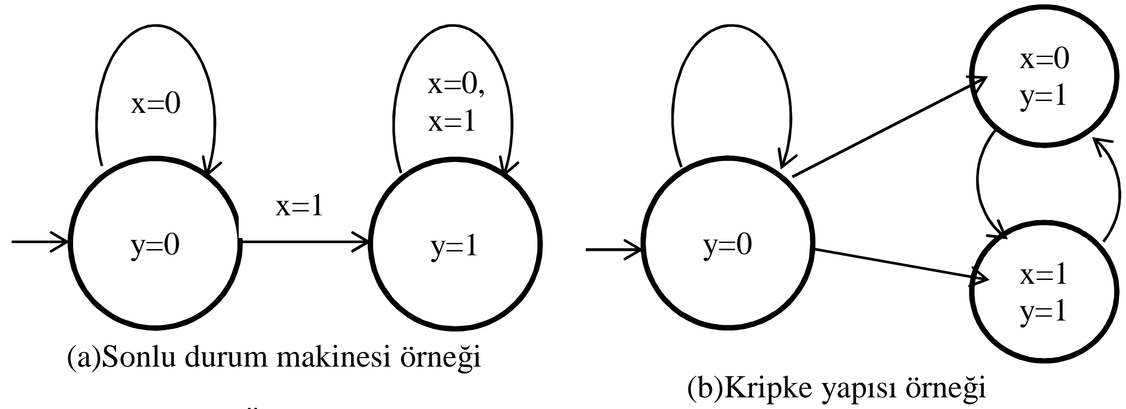
Yazılım testi alanındaki araştırma otomatik test durumları üretimi için birkaç farklı yaklaşımın belirlenmesi ile sonuçlanmıştır. Genel olgu, tüm otomasyon kullanımı ile test durumları üretimi yaklaşımlarında temel ihtiyacın bir model veya şartnameye ihtiyacı oluşudur. Bu sebeple, bu tür teknikler model bazlı test teknikleri olarak bilinir. Bir model olmadan da test verisi elde etmek mümkündür. Buradaki asıl sıkıntı modelin tam olarak ne olduğunun ifade edilmesindedir. Örneğin, giriş değişkenleri ve bu değişkenlerin etki alanı şartnamesi modelin kendisi olarak kabul edilebilir ve test durumları denklik bölümleri ve sınır değerlere istinaden oluşturulabilir. Tam bir otomasyon oluşturabilmek için birde test kahinine ihtiyaç duyulmaktadır, bu test kahini karşılaşılan sorunun hata olup olmadığının tespitini yapacaktır. Eğer test durumları sadece denklik bölümlerine göre oluşturulmuş olsalardı beklenen değerler otomatik olarak bilinemezdi, bu durum genellikle test verisi ile kontrol edilir. Test edilmekte olan sistemin formal ve fonksiyonel açıklamalarına ilişkin bir model tanımı yapılmıştır. Sonuç olarak, formal model test kahini olarak kullanılabilecek bir yapıdır. Test kahininin işlevi oldukça önemlidir. Öyle ki, kullanmış olan test durumu terimi bir çift test verisi ve bu işlemin beklenen sonucunu ifade etmektedir. Bununla beraber bazı senaryolarda akla uygun bir test kahinine ihtiyaç duyulmayabilir. Örneğin, bazı testlerde test durumları doğrudan uygulamanın kodundan oluşturulabilir ve hedef platform üzerinde uygulanarak uygulamanın kendi kendisinin hedef platformda çalıştırılabilir kod üretmesi durumunun kontrol edilmesi şeklinde olabilir.

Otomasyon ile test durumları oluşturulmasının temel karakteristiği model biçimciliktir. Çok benimsenmiş ve kapsamlı araştırmalar yapılmış bir diğer alan da sonlu durum makineleridir (*Finite State Machines* – FSM). Örneğin protokol testleri genellikle bu şekilde bir yaklaşım ile uygulanır. Model bazlı yazılım dilleri olan VDM-SL, Z veya Alloy'da test durumları oluşturmada UML alt kümeleri kadar verimli kullanılırlar. Etiketli geçiş sistemleri (*Labeled Transition Systems* – LTS) son dönemde oldukça popüler bir halde kullanılmaktadır. Bir başka yaklaşım olan model denetleyiciler ise test otomasyonu alanında dikkat çekici bir noktaya ulaşmıştır.

Model denetleyici formal denetim yapabilen bir araçtan ibarettir. Ücretsiz olarak bulunabilen farklı çeşitlerde çok sayıda model denetleyici mevcuttur, bu sebeple bu yaklaşımın kullanımı rahatlıkla denenebilir. Model denetleyici temelli yazılım testi, test durumları oluşturmada bir çok farklı şekilde uygulanabilen esnek yaklaşım özelliği nedeni ile alanında maliyetleri düşürecek önemli bir seçenek haline gelmiştir.

3.2. Model Denetleyici

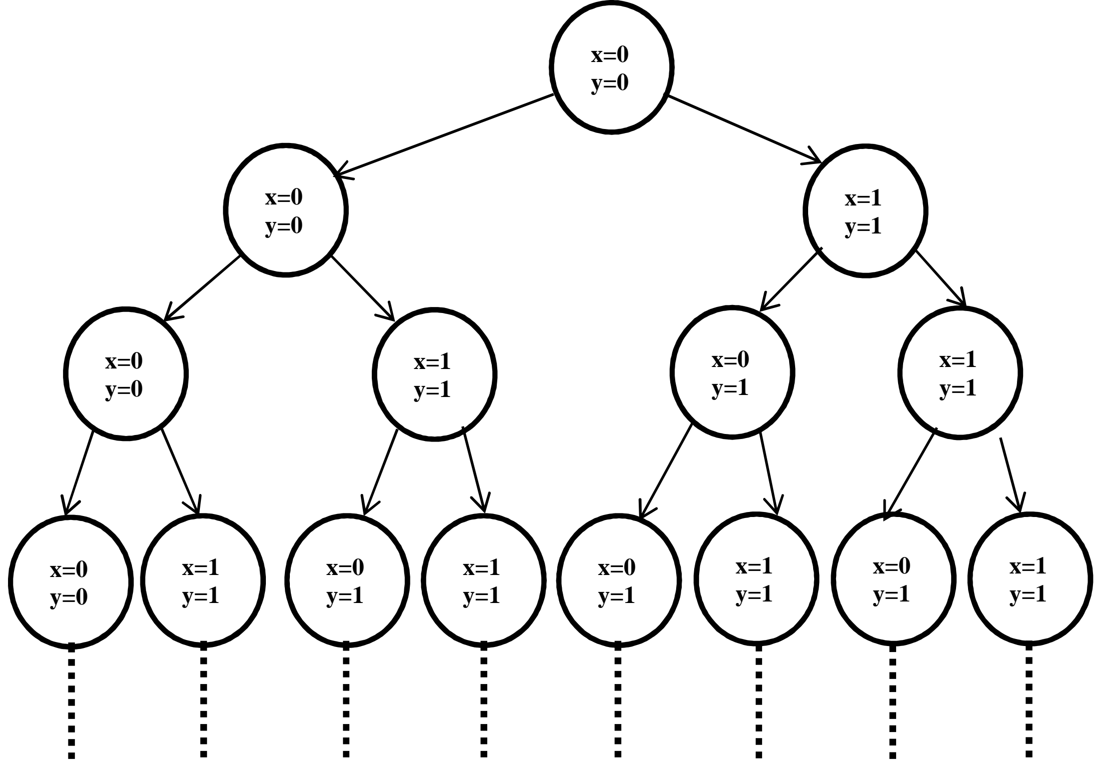
Temel olarak model denetleyici formal denetim yapabilen bir araçtır (*B.A. Myers, 2001*). Operasyonel şartnameye uyan bir girdi alır ve bir zamansal mantık (*Temporal Logic* – TL) çerçevesinde özelliklerin ihlal edilip edilmediğinin belirlenmesi çerçevesinde modelin tüm durum uzayını analiz eder. Durum uzayının tamamının dolaşılması sonucu özelliklerin ihlal edilmediğini tespit edebildiğinde ise başarılı sonuçlandığına dair ispat edinilmiş olur. Model denetleyicilerin en basit özelliği aynı anda hem ihlalleri hem de başarılı sonuçlanan durumları birarada sunabilmeleridir. Tipik bir model denetleyici bir özelliğin ihlal edildiğini tespit ettiğinde bu durumu ortaya koyan bir karşı örnek üretir. Analist kişi bu karşı örneği kullanarak tasarımda bulunan sorunu çözebilir. Test aşamasında bu karşı örnekler doğrudan test durumu olarak kullanılabilirler. Model denetleme süreci üç ana başlıktan meydana gelir. Modelleme, doğrulanacak özelliklerin tespit edilmesi ve doğrulama.



Şekil 3.1 Örnek sonlu durum makinesi ve karşılığı olan kripke yapısı

Model denetleyicilerin kullandığı yapı Kripke çatısı modelidir. Saul Kripke tarafından tanımlanan bu yapı bir sonlu durum makinesidir. Durumların etiketleri vardır. Etiketler sistemde tanımlı doğru/yanlış ifadelerininden o durum içinde doğru değeri alanlarını gösterir. Bunlar atomik önermelerdir.

Örnek olarak, bir girdi (x) alan ve x değerinin daha önce hiç doğru/geçerli (1) olup olmadığı sonucunu y olarak bize sunan otomasyonu düşünelim. Şekil 2'de görülen (a) sonlu durum makinesi örneği x değerinin geçerli (1) olma koşulu sağlandığında y = 1 durumuna geçtiği ve bu durumda sonlanarak artık geriye dönüşün herhangi bir girdi tarafından sağlanamadığını göstermektedir. Şekil'de (b) kripke yapısı örneğinde ise aynı durumun otomasyonunu görmekteyiz.



Şekil 3.2 Örnek bir otomasyonun yürütme ağacı şeması

Kripke yapısında durum, x girdisinin aldığı değerler ile oluşarak y çıktısının alacağı sonucu sunmaktadır. Tüm olası yürütmeleri Şekil 3'de görülen kripke yapısı ile açıklamak mümkündür. Şekilde görülen tüm yollar kripke yapısı ile oluşmaktadır.

Pratikte sonsuz yollar kullanılamayacağından, model denetleyiciler "izler/yollar" olarak ifade edilen sonlu dizileri kullanırlar. Bununla beraber ihtiyaç halinde son durumun bir döngü halinde sürekli tekrar edildiği sonlu diziler gerekli hallerde kullanılabilir. Modellemedeki en büyük sıkıntı durumlar uzayının çok sayıda durum içermesi ve ele alınamayacak boyutlara ulaşmasıdır. Büyük boyutlardaki durumlar uzayının küçültülebilmesi için gerçek sistem üzerinde soyutlama teknikleri kullanılmak sureti ile model oluşturulur. Oluşturulan modelde dikkat edilmesi gereken nokta doğrulamanın sonuçlarını tehlikeye atmamaktır.

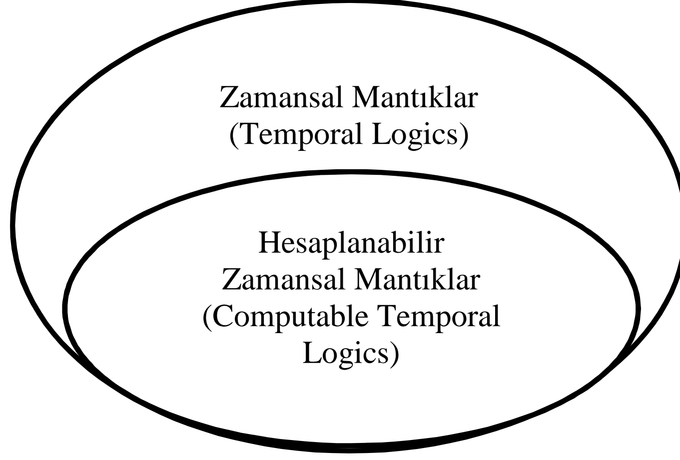
Model denetlemenin son aşaması verilen modelin verili özellikleri sağlayıp sağlamadığını denetlemektir. Bu aşama araçlar tarafından otomatik olarak yapılmaktadır. Modelin özellikleri sağlamadığı durumda, araçlar aynı hatanın meydana geldiği bir yürütme raporu üretirler. Bu sayede hata olup olmadığının öğrenilmesi ile kalmayıp hatayı tekrar nasıl oluşturabileceğimizi de tespit etmiş oluruz.

3.3. Zamansal Mantıklar

Zaman, doğrusal ya da dallanan şekilde ifade edilebilir. Örnek olarak "oturana kadar belim ağrıyacak" önermesi verilebilir. Zamansal mantık Amir Pnueli ile birlikte bilgisayar alanında programların belirtilmesi ve doğrulanması için sıkça kullanılmaya başlanmıştır (*H. Gümüşkaya, M.V. Nural, 2008*). En yaygın kullanılan zamansal mantık, doğrusal zaman mantığı (*Linear Temporal Logic* – LTL) ve dallanan zaman mantığı veya hesaplanabilir zaman mantıklarından biri olan hesaplama ağacı mantığı (*Computation Tree Logic* - CTL) (*Clarke ve Emerson, 1982*)'dır. CTL tüm bu mantıkların çatı kümesi olarak ifade edilmiştir (*Emerson ve Halpern, 1982*). En yaygın kullanımı olan model denetleyiciler ya LTL ya da CTL bazende her ikisini de destekler.

Model denetleycilerde kullanılan diğer zamansal mantık yapılarına Hennessy-Milner Mantığı HML (*M. Hennessy ve R. Milner, 1985*), Modal μ-calculus (*D. Kozen, 1983*) ve CTL'nin farklı türleri olan zamanlı, adil ve dinamik CTL örnek olarak gösterilebilir.

Zamansal mantıklar, belirli bir zaman akışı sırasında bazı önerme kısıtları ile işlemler tanımlayan matematiksel sistemlerdir. Bu durumda zamansal mantığın en önemli ve ana işlevi, durumlar, olaylar ve bunların zamansal gösterimleridir. Burada temel amaç, zamansal mantıkların bu değerlerin modellenmesini doğru olarak gerçekleştirmesi ve doğru ilişkilerin bu değerler arasında kurulabilmesidir. Değinilmesi gereken diğer bir nokta da zamansal mantıkların tümünün hesaplanabilir (*computable*) olmayışıdır. Zamansal mantıkları iki ana grupta ele almak gerekir ki, bu gruplar hesaplanabilir ve hesaplanabilir olmayan(*a-computable*) zamansal mantıklardır.



Şekil 3.3 Zamansal Mantıklar

Hesaplanabilir zamansal mantıklar, zamansal mantıkların kapsadığı bir alt kümedir. Hesaplanabilir olmayan zamansal mantıklar bu kümelerin farkını meydana getirir. Aradaki fark, yapısal açıdan zamansal mantığın üzerinde kurulmuş olan klasik mantıktadır. Mantığın bilgisayar bilimlerinde kullanılabilmesi işlenebilmesi için ilk şart, matematiksel bir yapıya sahip ya da doğrudan hesaplanabilir olmasıdır. Dolayısı ile kullanılabilir olmadığından hesaplanabilir olmayan mantıklar kapsam dışında tutulmuştur. Sadece temel olarak kullanılan doğrusal zaman mantığı ve hesaplanabilir mantıkların çatı kümesini oluşturan hesaplama ağacı mantığı ile çalışılmıştır.

3.3.1. LTL (Doğrusal Zaman Mantığı)

Şu anki zaman doğrusal zaman mantığı çerçevesinde geçmişten gelen ve geleceğe devam eden çizgisel düzlemde bir noktayı temsil eder. LTL sayesinde aynı anda birden fazla olayın gerçeleşmesi durumunu ifade eden alternatif olaylar arasındaki ilişkiyi doğrusal olarak modelleyerek ifade etmek mümkün kılınmıştır. Doğrusal olarak modellediği önermeler ve üzerine kurulu olduğu mantık itibarı ile literatüre önerme zamansal zantığı (*Prepositional Temporal Logic* - PTL) olarak da geçmiştir.

Tüm zamansal mantıklara dayanak olan bu yaklaşımda ifadelerde kullanılan bazı operatörler aşağıdaki gibi açıklanabilir:

- Gp Globally (tüm zamanlar), p önermesini tüm zamanlar için doğru kabul etmek anlamına gelmektedir.
- Fp Future (gelecekteki), p önermesini ileride herhangi bir zamanda doğru kabul etmek anlamına gelmektedir.
- pUq Until (e kadar), p önermesini, q'nun doğru/geçerli olduğu ana kadar geçerli kabul etmek anlamındadır. Bu durumda p olayı herzaman q'dan önce gerçekleşmiştir.
- Xp Next (sonraki), p önermesini aynı zamanda bir sonraki önerme olarak kabul etmek anlamına gelmektedir.

Yukarıdaki durum operatörlerine örnek olarak aşağıdaki örneği ele alabiliriz:

"Ahmet ders çalışıyorken Aslı spor yapıyordu. Ahmet dersini bitirip oyun oynamaya gitti."

p: "Ahmet ders çalışır"

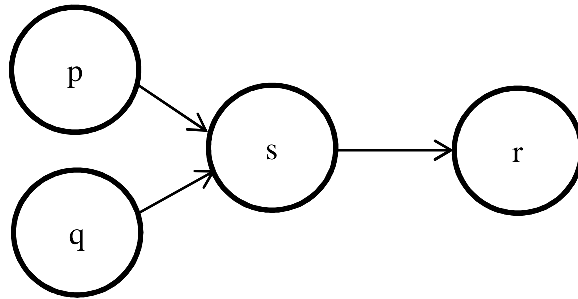
q: "Aslı spor yapar"

s: "Ahmetin dersi biter"

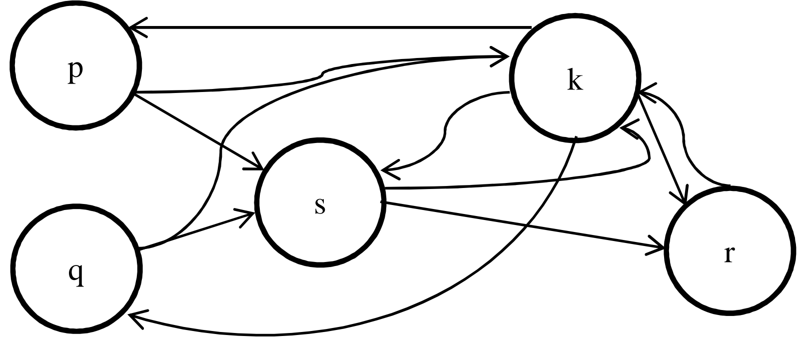
r: "Ahmet oyun oynamaya gider"

Önermelerin LTL semantiği (yazım biçimselliği) ile gösterimi "$p\ U\ s \wedge q\ U\ s \wedge X\ r$" olarak ifade edilir..

Örnekte belirtildiği gibi zamansal düzlemde bir nokta referans olarak belirlenmiş ve buna göre olayların akışı ele alınmıştır. Ek olarak, bir olay üzerinden aynı anda farklı olayların birleşmesi durumu gerçekleştiğinde, olaylara ilişkin yollar çizilerek bu sıralama aşağıdaki gibi ifade edilebilir.



Şekil 3.4 Örnek ilişki şeması



Şekil 3.5 Genişletilmiş ilişki şeması

Bu örnekte ilk yol p→s→r olarak ifade edilirken, diğer bir yol q→s→r olarak ifade edilebilmektedir. Her iki yolda bir LTL modeliyle temsil edilmektedir. Doğrusal zaman mantığı ile modelleme yaparken birbiri ile bağlantısız olan bilgiler gösterilemez. Örnek vermek gerekirse "k" olarak isimlendirdiğimiz "Ali otobüse biniyor" şeklindeki ilgisiz bir durum eklenseydi bu durum zamansal doğrusal mantık ile modellenemezdi. Bu durum, yeni eklenen önermenin mevcut durumlara ait önermelere bağlanabileceği bir yolun mevcut olmayışı nedeniyledir. Yeni olaya ait önermenin mevcut olaylardan önce veya sonra gelmesi muhtemeldir.

3.3.2. CTL (Hesaplama Ağacı Mantığı)

Hesaplanabilir zaman mantıklarından biri olan ve model denetleyicilerde sıklıkla tercih edilen CTL'de zaman, olaylar ile birlikte çatallanarak büyüyen bir ağaç yapısı şeklinde modellenir (*Edmund M. Clarke, E. Allen. Emerson ve A.P. Sistla, 1986*). Hesaplama ağacı mantığında, doğrusal zaman mantığından farklı şekilde A (tüm zamanlar için) veya E (herhangi bir zamanda) şeklinde ekler bulunur. Hesaplama ağacı mantığında EF (gelecekteki zamanda), AF (gelecekte tüm zamanlarda), EP (geçmişte), AP (geçmişteki tüm zamanlarda), EG (zamansız), AG (tüm zamanlar için) şeklindedir. Doğrusal zaman mantığı için verdiğimiz örneği, hesaplama ağacı mantığı için ilerletecek olursak aşağıdaki (a) gibi, ek olarak işin içine "k" önermesi eklendiğinde ise aşağıdaki (b) şekilde ifade edilebilir.

(a). $p\ AU\ s \wedge q\ EU\ s \wedge EX\ r$

(b). $p\ AU\ s \wedge q\ EU\ s \wedge EX\ r \wedge EG\ k$

4. YAZILIM TESTLERİNİN YAYGINLAŞTIRILMASI

Normalde model denetleyici, bir sistemdeki durum ihlallerini sonlu durum ifadeleri ile analiz etmek amacıyla kullanılır. Eğer model denetleyici ulaşılabilir tüm durumları analiz eder ve ihlal ile karşılaşmaz ise özellik kabul edilir. Ancak, model denetleyici erişebildiği durumlar içerisinde özelliğin ihlal edildiği bir durum tespit eder ise geçerli durumdan ihlal durumuna ulaşılana kadar olan durumları içeren bir ters örnek ile rapor oluşturur. Bu teknik sayesinde model denetleyici bir test kahini olarak kullanılmış ve beklenen sonuçların ters örneklerinin oluşturulması ile test durumları üretimi için kullanılmıştır.

4.1. Ters Örnek Üretimi

Ters örnek üretimi için en uygun yöntem mutasyon analizidir (*A. Gargantini ve C. Heitmeyer, 1999*). Bu bölümde mutasyon analizi ile benzer bir yöntemin uygulanışı ele alınmıştır.

4.1.1. Mutasyon Operatörleri Kullanımı İle Test Durumu Üretimi

Strateji, model denetleyici ile test durumları üretimi için mutasyon analizi uygulanmasıdır. Mutasyon analizi, programın yapısında sözdizimsel ufak değişimler konusunda oldukça hassas bir “beyaz kutu” test metodudur. Eğer program kodu yerine şartname üzerinden test durumları üretiminden söz ediyor olunur ise, bu durum da “kara kutu” testi olarak ifade edilir. Buradaki mantıksal temel, test durumlarının programdaki hafif değişimleri layıkı ile ayırt edebilmesidir.

Bu metod ile kullanılan yaklaşımda amaç, sistem şartnamesi ile başlayarak bu şartnameyi sonlu modelleme ile model denetlemeye uygun bir şartname haline getirmektir. Mutasyon operatörleri, durum makinesine (*state machine*) ya da mutant şartnamesini sağlayan kısıtlara uygulanmaktadır. Model denetleyici mutantları birer birer işlemektedir.

Model denetleyici bir tutarsızlık ya da ihlal tespit ettiğinde bir ters örnek üretir. Ters örnekler kümesi ancak mükerrer örnekleri eleyerek ve bir başka örneğe “ön ek” olan örnekleri göz ardı ederek küçültülebilir. Ters örnekler hem girdileri hem de beklenen çıktıları içeriyor olduğundan doğrudan test durumları olarak kullanılabilirler.

Mutasyon operatörlerini iki kategoride ele alabiliriz:

- Durum makinesi değişiklikleri: Ters örnekler durum makinelerinden türetildiği için, asıl testten farklı bir yaklaşımla uygulanmalıdır. Buradaki kasıt, böyle bir test için verilen giriş değerleri doğru olduğunda ters örnek ile farklı sonuçlar kayıt altına alınıyor olmalıdır. Bunlar negatif/başarısız testler ya da uygulanışı sırasında başarısız olması beklenen testler olarak ifade edilir.
- Zamansal mantık kısıt değişiklikleri: Bu kategoride, ters örnekler "asıl" sonlu durum makinesi üzerinden elde edildiği için doğru uygulamada beklenen durum geçişlerini vererek sonuca ulaşmalıdır. Deterministik bir şartname olduğunu varsaymak zorundayız, aksi takdirde mutasyon ile elde ettiğimiz test durumları kullanışsız olacaktır. Bu durumda uygulamada testlerin pozitif/başarılı sonuçlanması beklenmelidir.

Dört tip mutasyon operatörü kullanılmaktadır:

1. Durum makinesine yeni bir koşul eklemek
2. Birden fazla koşul bulunduğunda koşul silmek
3. Bir özelliğin alt ifadesini bir başka geçerli alt ifade ile değiştirmek
4. Bir özelliğin alt ifadesini geçersiz bir alt ifade ile değiştirmek

Örnek olarak şartnamede verilmiş bir geçişi kullanalım:

Sonraki(v):= case c_1: e_1; c_2: e_2; ... esac;

Mutasyon operatörlerinin ilki c_i koşulunu değiştirir ve bu koşula bir e değeri ekler.

İkinci tip mutasyon operatörü eğer birden fazla koşul içeriyor ise c_i koşullarından birtanesini siler.

Verilen CTL özelliği:

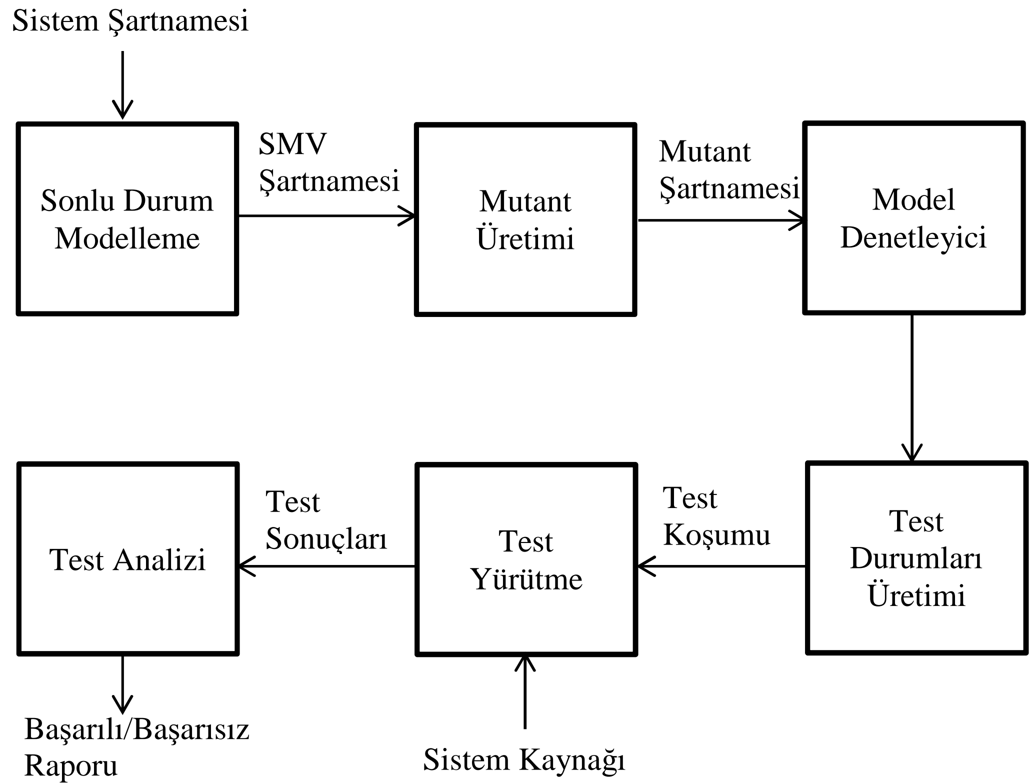
ÖZELLİK AG $(x=mode_i \rightarrow ...)\ \&\ (c_i\ \&\ x=mode_i)$ ...

Operatörlerinden üçüncüsü $mode_i$ ifadesini geçerli başka bir ifade olan x ile değiştirir.

Dördüncü operatör $x=mode_i$ ifadesini geçersiz bir ifade ile değiştirir. Örn; $!(x=mode_i)$.

Operatörlerden ihlal durumu tespiti için değiştirilenlerden sadece zamansal mantık niteliklerini sağlayan ters örnekler ile ilgilenilir. Örnek vermek gerekir ise, durum makinesinin hiçbir zaman doğru/geçerli olmayacak bir duruma güdülenmesi sonucunu doğuran bir ters örneği oluşturmak pek gerçekçi ve kullanılabilir değildir. Bu nedenle hiçbir zaman doğru/geçerli olmayacak durumlar baştan elenir. Üretilen test kodu uygulama kaynağı ile yürütülür ve aynı zamanda kapsamı kayıt altına alır.

Test kodunun kayıt altına almış olduğu sonuçlar işlenerek testlerin tam kapsamlı olup olmadığını ortaya koyan bir rapor haline gelir. Bu metod ile uygulamanın negatif/başarısız ya da pozitif/başarılı sonuçlanması gereken durumlarda doğru biçimde sonuçlandığı kontrol edilir. Aşağıda bu tekniği ele alan bir akış şeması yer almaktadır.



Şekil 4.1 Mutasyon operatörleri kullanımı ile test durumu üretimi

4.2. Durum Uzayının Daraltılması

Model denetlemede en büyük sorun durumlar uzayının ele alınması mümkün olmayacak şekilde geniş bir kapsama ulaşmasıdır. Bu bölümde test bakış açısı ile bu durumun üstesinden gelmek üzere birkaç teknik ele alınmıştır.

4.2.1. Sınırlı Odak

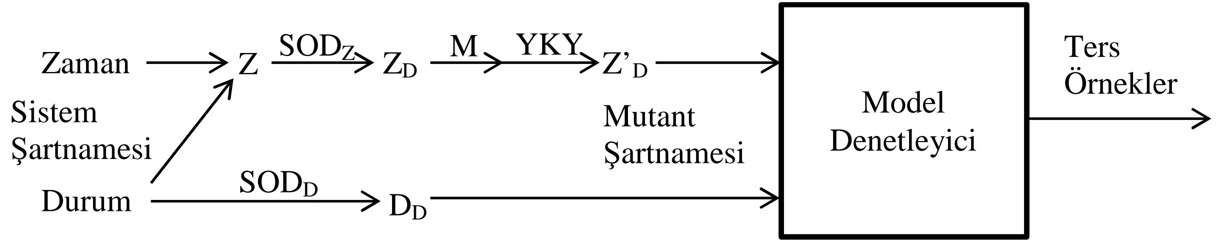
Mutasyon test üretimi yönteminin kurucuları bu yöntemi ifade etmek için göreceli olarak küçük çaplı bir örnek kullanırlar. Bu durumun ortaya çıkarmış olduğu en büyük sıkıntı, endüstriyel alanda kullanılacak boyutlarda bir model için model denetleyici tarafından ele alınamayacak kadar büyük bir durum uzayının var olmasıdır. Bu kısıt, model denetleyicilerin yazılım endüstrisi tarafından kabul görmeyişinin ana sebebidir. Sınırlı odak bu kısıtı belirleyebilmek için anahtar rol üstlenmektedir

Sınırlı odak, otomatik test durumları üretimi açısından ortaya çıkan durum uzayının daraltılmasını amaçlamaktadır (*P. Ammann ve P.E. Black , 1999*). Bu nedenle geleneksel model denetlemede ana amaç özelliklerin analizi ve doğrulanması iken daraltılan durum uzayı ile durum geçişlerinin bazıları gözardı edilerek tüm durumların bir özeti çıkarılmış olmaktadır. Daraltılmış model tam kesinlikte olmasada tüm analiz ve doğrulamanın yapılması noktasında oldukça başarılıdır. Otomatik test durumları üretimi denildiğinde, sınırlı odak tekniğinde uygulamanın doğru davrandığını ortaya koyan detaylar asla atlanmamalıdır. Test durumları tek tek ele alınarak özenle daraltılmış durum uzayından meyda gelmelidir.

Tipik soyutlama yöntemi, büyük ya da birbirinden bağlantısız alanlarda yer alan değişkenleri olası değerlerden oluşan daha basit bir altkümede toplamaktır. Test durumları üretimi açısından gerçek hayatta kullanılabilecek değerlerin olası tüm alt ve üst sınırlarını kapsıyor olması yeterlidir.

Basit bir örnekle durumu açıklamak gerekirse, bir banka müşterisinin hesabından 1TL para çekmesi durumunu modellediğimizi düşünelim. İlgilenmemiz gereken ana husus hesabın 1TL para çekme işlemine müsait bir bakiyede olup olmadığıdır. Sıradan bir analizde ana odak hesapta hiç para olmadığı durumda gelen talepleri gözardı etmek üzerine olacaktır. Bu durumda 2TL ve üzeri yapılan tüm taleplerin gözardı edilmesine denk gelen "diğer" durumuna işaret edecektir. Ancak model denetleyici hesap bakiyesinin 2TL üzerinde olduğu farklı durumlarda gerçekleşecek göz ardı edilmesi gereken tüm operasyonel adımları da bilmelidir. Hesap bakiyesinin 2TL veya daha büyük olduğu durumlarda ortaya çıkan muğlaklığı ortadan kaldırmanın yolu bu "diğer" duruma işaret edecek tüm operasyonel adımlarda geçersiz hale gelecek "mükemmellik" durum değişkeni tanımlanması ile

olur. Bu sayede model denetleyici geçersiz değerine sahip durumları gözardı edebilir ve bunların dışında kalan tüm problemleri ortaya koyabilir.



Şekil 4.2 Şartname dönüşümleri

Sınırlı odak bir sistem şartnamesini durum makinesi ve zamansal mantık kısıtlarından oluşan bir çift olarak görür (*D ve Z*). Mutasyon operatörleri ile test durumları üretilebilmesi için SOD'ın (*Sınırlı Odak ile Daraltım*), model denetleyici tarafından analiz edilebilecek biçimde olması zorunludur. Test durumları üretiminde durum makinesi D, daha sonraki mutasyon analizi için gereken açıklamaları zamansal mantık kısıtları biçiminde yansıtır. Mevcut zamansal mantık kısıtları (Şekil 8, Z), durum makinesini oluşturan kısıtları yansıtacak şekilde eklenebilir.

Bazı sınırlı sayıdaki durumlar, ilk durum etrafında odaklanmıştır. Bu durumlar daraltılmış şartnamedeki durumları işaret etmektedir. Diğer tüm durumlar tek bir "diğer" durumuna işaret etmektedir. Her bir geçişin kaynağı ve hedefi bu şekilde birbirini işaret etmektedir. SOD_Z zamansal mantık kısıtlarını, SOD_D durum makinesini işaret etmektedir. Bu iki fonksiyon, "mükemmellik" için yeni kısıt yazımı YKY (Yeni kısıt yazımı) sınırlı odak ile daraltım, SOD'u meydana getirmektedir.

SOD_D ilk durumu "geçerli" olan ayrı bir durum makinesi ekler. Ne zaman ki daraltılmış durum makinesi "diğer" durumunda sonlanır, bu eklenen durum makinesi "geçersiz" durumunu alır ve bundan sonra hep "geçersiz" kalmaya devam eder. Bu adımda daraltılmış durum D_D meydana gelir. SOD_Z daraltılmış zamansal mantık kısıtlarını Z_D meydana getirir ancak SOD_D kadar katı değildir. D_D ve Z_D birlikte mutasyon analizi test durumları üretimi konusunda tüm durum şartlarını karşılamaktadır.

Ters örnek üretimi için, zamansal mantık kısıtlarına çeşitli mutasyon operatörleri "M" tekrarlayan şekilde uygulanır. Daha sonra, geçersiz ters örnekleri engellemek amacıyla kısıtlar yeniden yazılarak sürekli geçerliliği sağlayacak hale getirilir. Bu YKY ile, Z'_D mutantı ortaya çıkar. D_D ve Z'_D birlikte model

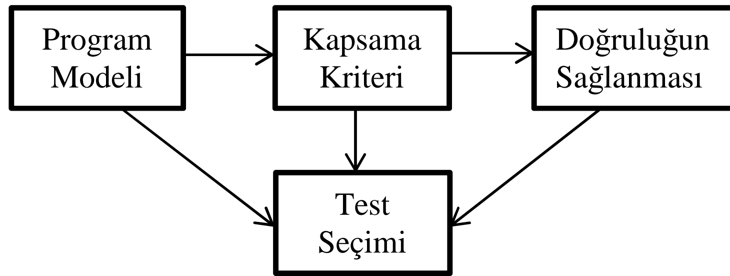
denetleyiciye girdi oluşturduğunda model denetleyici ters örnekleri ortaya koyar. Test durumları üretiminin sağlıklı olması, daraltılmış şartnameden (D_D , Z'_D) üretilen tüm ters örneklerin orjinal durum makinesi şartnamesi D içerisinde geçerli bir dayanağının bulunmasına bağlıdır.

4.2.2. Bileşen Etkileşimi

Bileşen etkileşimi metodu bileşen etkileşimlerinin formal modelini ve formal olarak ifade edilen test etkileşim kriterlerini kullanır (*W. Lui ve P. Dasiewicz, 2000*). Formal etkileşim modelleri obje temelli analiz ve tasarımda geliştirilen durum makinesi modellerinin yeniden kullanılması ile tasarlanır. Bu teknik birleşik modelleme dilinin (*Unified Modeling Language, UML*) obje durumu (*ObjectState*) adlı bir uzantısını kullanmaktadır. Gereken etkileşimli test durumları model denetleyici tarafından otomatik olarak üretilir.

Obje durumu üç adet alt dilden meydana gelmektedir. Bileşenler ve bağlantıların ifade edildiği mimari ifadeler dili (*Architectural Description Language, ADL*), tüm bileşenlerin davranışlarını ifade eden sonlu durum makinesi ifadelerini içeren davranışsal dil, ve bileşen verilerinde geçişlerin etkilerini modelleyen ifadelerden oluşan veri manipülasyon dili.

Obje durum modelinin formal temelini etiketli geçiş sistemleri oluşturmaktadır. Etiketli geçiş sistemleri, bileşenler, birbiri ardına gelen etkileşimler ve durum bazlı etkileşimler arası arayüz oluşturur. Teorik olarak olgunlaşmış kaideleri, verimli algoritmaları ve çok sayıda araç gereci mevcuttur.



Şekil 4.3 Bileşen etkileşimi için test seçim tekniği

Formal modeli tanımlamadan önce formal kapsam kriteri tanımlanabilir. Hangi kapsam kriterinin seçileceğine dair bir şart bulunmamaktadır. Bileşen

etkileşimi, test seçim kriteri olarak olay akışını kullanır. Bu akış kullanıcı tarafından kullanılan "bağlı etkileşim" çiftleri tarafından meydana gelmektedir.

Teknik, test durumları üretimi için iki algoritmadan meydana gelmektedir. İlk olarak, test durumlarının parçalar halinde artarak üretilebilmesi avantajından faydalanan artarak ilerleyen test durumları üretimidir. Tam bir test durumu şeklinde genişletilmek üzere kısmi test durumları ile girdi/çıktılardan oluşan altkümeden faydalanılmaktadır. Girdi ve çıktılardan oluşan altküme düşünüldüğünde bunların sistem içerisinde hali hazırda gizli durumda mevcut olabilecekleri göz önünde bulundurulmalıdır. Bu durumda tespit edilerek ortaya çıkarılmaları için gereken çaba tüm test durumunun meydana getirilmesi için harcanacak çabadan daha azdır. Algoritma obje durumu içerisindeki dış kapılardan birtanesini seçer ve diğer dış kapılardan gelen mesajları gizler. Bileşenler aza indirgenmiş ve sonuca giden yol ortaya konulmuş olur.

Sonuca giden yolda ortaya çıkan girdi ve çıktılar ile yeni etiketli geçiş sistemi oluşmuştur. Bu yeni etiketli geçiş sistemi test durumunun kendisini meydana getirmektedir.

Yeni etkietli geçiş sistemi, seçilen girdi ve çıktılar sonucu ortaya çıkan davranışa uygun olacak şekilde bütünleşmiştir. Bu durum, model denetleyici için sistemin muhtemel sayıda davranışı sonucu oluşan durumların azalmasını sağlar. Algoritma dış kapıların tümünü kapsayana kadar tekrar eder. Son aramanın sonucu sistemin tüm girdi ve çıktılarını takip eden ve test durumunun kendisini oluşturan bir yol meydana getirmiştir.

İkinci algoritma, test durumu oluşturmak için gereken tüm bilgiyi kaybetmemek şartıyla bazı luzümsuz bilgileri dikkate almaz. Test durumları üretiminde çevresel etkileşimler tamamıyla korunur ancak bileşenler arası etkileşim etkileşimin sistem üzerine olan etkisi hariç olmak şartıyla gözardı edilir. Eğer iki etkileşim sonucunda bileşen aynı durum değerini alıyor ise bu iki etkileşim birleştirilebilir. Ancak, hangi durumların eşit değerde sonuçlanacağı sonraki bileşen ile birleştirilene kadar anlaşılamayacağından algoritma tekrarlamalı olarak çalışmalıdır. Algoritma iyimser bir bakış ile önce birleştirmeyi yapar ancak diğer bileşenin sonucunu elde ettiğinde hata yaptığını anlayarak geri döner ve bir önceki bileşeni tekrar ele alır. Algoritma hiç hata kalmayana dek tekrarlamalı olarak devam eder. Sonuç olarak tekrarlamalı bir şekilde üzerinden geçerek ortaya çıkacak durum

uzayının olabildiğince küçülmesini sağlamaktadır. Bu, uygulamada en büyük sorunlardan biri olan durum uzayı patlamasının önüne geçebilmektedir.

5. VAKA ÇALIŞMASI

5.1. Sistem ve Model

Örnek çalışmada ele alınan konu, kablosuz iletişim sistemi GSM hücresel telefon standartlarında uygulama protokolü kimlik modülü olan WIM (*Wireless application protocol Identity Module , WIM*) aracılığı ile kontrol edilen iletim seviyesi güvenliğinin sağlanması, dijital imza ve genele açık anahtar kriptografisidir.

WIM bir akıllı kart uygulaması olarak hazırlanmaktadır. Akıllı kartlar kendilerine ait rastlantısal erişimli belleğe (*Random Access Memory , RAM*), elektronik olarak silinebilir ve yeniden programlanabilir salt okunur belleğe (*Electrically Erasable Programmable Read Only Memory , EEPROM*), salt okunur belleğe (*Read Only Memory , ROM*), bir dahili mikroişlemci ve bankamatik (*Automated Teller Machine , ATM*), cep telefonu gibi cihazlarla iletişim kurabilmelerini sağlayan seri arayüze sahip birer tek-çip bilgisayardır.

Akıllı kartların programlama modeli birer komut yorumlayıcıdan ibarettir: Kartın mikroişlemcisi giriş kanalı aracılığı ile iletilen komutlarını okur (örn; cep telefonu), komutu ve komuta ait parametreleri alır ve yürütür (bu durum kart üzerinde saklı verilerde değişiklik yapabilir) ve çıkış kanalı üzerinden bir yanıt iletir. Başarılı yürütme işlemi bir önceki komutlarla bağlantılı olabilir (örn; kullanıcı doğrulanmalı, anahtarlar ayarlanmalı ve kartın dosya sistemi okunabilir hale gelmeli).

Sistemin yapısı bir ağ biçiminde birbirlerine doğrudan iletişim kanalları ile bağlı bileşenlerden oluşur.Komutlar sisteme girer, çeşitli fonksiyonel bloklar halinde sınıflandırılır: dosya sistemi, kart sahibi doğrulaması(çeşitli PIN kontrol komutları ele alındığı alan), güvenlik bölgesi (anahtarlar, sertifikalar ve diğer parametrelerin saklandığı alan), güvenlik işlemleri (kriptopgrafik işlemlerin yürütüldüğü alan). Bu fonksiyonel bloklardan alınan yanıtlar toplanır, bazı durumlarda tampon belleğe alınır ve WIM'in çıkış kanalından iletilir.

Bazı kart bileşenleri, başta dosya sistemi ve kart sahibi doğrulama verileri diğer kart uygulamalarında tekrar kullanılabilir yapıdadır.

5.2. Test Durumları Üretimi

Modelden test durumları üretimi için test durumları şartnamesine ihtiyaç duyulmaktadır. Bu şartname sonucu doğrudan belirleyen bir öneme sahiptir, belirlenen uygulama için neyin test edileceğini ve "pozitif test durumları" gibi bilgileri doğrudan yansıtır. Buna ek olarak, testin kendisinin bir maliyeti vardır ve bu maliyeti azaltmanın yolu test edilmesi gereken özelliklere konsantre olunmasından geçmektedir.

Fonksiyonel, yapısal ve olasılıklı rastlantısal şeklinde sınıflandırdığımız test durum şartları ile vaka çalışmasında ilerlenmiştir. Tüm sınıfların durum uzayları hesaplama ağacı mantığı kullanarak daraltılmıştır. Fonksiyonel şartlar genellikle belirli çıktıların beklentilerini içermektedir. Kısmi girdi/çıktı dizileri olarak ifade edilebilen senaryolar bahsedilen fonksiyonel şartlara örnek teşkil etmektedir. Benzer düzen, kapsamdaki tüm durumları içerecek şekilde, geçişler veya geçiş dizilerini kapsayacak şekilde yapısal test şartlarında da mevcuttur. Olasılıklı rastlantısal test durum şartlarında ise, verilen olasılıklara uygun girdileri üreten bir sürücü bileşene gereksinim duyulmaktadır.

Test durumları şartları modeli temsil edecek biçimde kısıtlamalı programlama mantığı diline (Constraint Logic Programming – CLP) dönüştürülür. Bu dönüştürme işlemi ile üç ana amaca hizmet edilmektedir: İz takip prosedürünün sınırları belirlenir, durum uzayının araştırılması yönlendirilir ve durum uzayının sınırları belirlenir.

Fonskiyonel test durum şartları;

WAP standartları, WIM uygulamalarının bazıları için çeşitli senaryolar sunmaktadır. Bunlardan birtanesi dijital imzanın hesaplanması adımlarını içermektedir. Bu senaryo 4 adımdan meydana gelmektedir. Doğru PIN numarasının girilmesi (1), doğru güvenlik ortamının seçimi (2), özel anahtarın ayarlanması (3), ve hesaplamaların yapılması (4). Test durumları şartları belirleyici olmayan durum makinesi, permutasyonları kodlayan sürücü bileşenden oluşmaktadır. Ek olarak, her durum için şartnameye göre etkisi olmayan komutlar olabileceği gibi (örn; bir dosyanın seçimi ya da rastgele rakam üretimi gibi), bazı komutların kartın dönüş yaptığı yanıta göre erkisi olabileceği bilinmektedir (örn; seçilen güvenlik ortamının değişimi gibi). Her iki türdeki komutlar sürücü otomasyonu açısından kolaylıkla

kodlanmaktadır. Bu durum dijital imzanın başarılı hesaplanmasının dışında ayrıca reddedilen komutlarında işletilebilmesini sağlamaktadır.

Yapısal test durum şartları;

Sistemin fonksiyonel ayrışımı sebebi ile her bileşen (fonksiyonel blok) için birbirine daha az bağımlı test durumları üretimi mümkündür. Örneğin, kart sahipliği doğrulamasının yapıldığı bileşenin test edilmesi için dosya sistemi ile ilgili olan komutlar durum uzayının daraltılması amacıyla kapsam dışı bırakılmıştır. Ek olarak, PIN durumunun "doğrulanmamış" değerine geçişine sebep olunduğunda artık belirli komutlar dışındaki tüm durum uzayının sona erdirilmesi sağlanmıştır.

Olasılıklı rastlantısal test durum şartları;

Görülmüştür ki rastgele test, eğer hata olasılıklarındaki artış belirli bir veri bölümü ile ilişkilendirilemiyor ise bölümlemeli testler kadar etkilidir. Bu, belirli bir uzunluktan (örn; birkaç yüz komut) oluşmakta olan test dizilerinden rastgele üretilebilen test durumlarını oluşturulmasına olanak sağlamıştır. Üretilen test dizilerinin sayısını azaltabilmek için ihtiyacımız olan üretilen iki farklı dizinin bir ölçüde birbirlerinden farklı olduklarını görmek yeterlidir.

5.3. Testin Yürütülmesi

Test durumları üretimi başlıklı bir önceki bölümde belirtilen şekilde üretilen test dizileri WIM modeli ile akıllı kart üzerinde uygulamanın hayata geçirilebilmesini sağlamıştır. Her bir test dizisinde yer alan her bir komut için kartın geri dönüş yapmış olduğu cevap model ile kıyaslanmıştır.

En basit şekliyle, beklenenden farklı bir cevap karttan alınıyorsa test başarısız kabul edilmektedir. Bu noktada testin başarısız oluşuna sebep cevap bir hata sonucu ya da uygulamanın hayata geçirilmesi sırasındaki bir yanlışlıktan kaynaklanmış ise tespit edilmelidir.

Ancak, bu karşılaştırmalı kontrol tüm kart komutları üzerinde yapılamaz. WIM modeli aslen WIM uygulamasının sadeleştirilmiş halidir, kartta kullanılmış olan kriptografi algoritmalarını doğrudan yansıtmamaktadır. Bu sorunu aşmak için, kriptografik cevaplar model tarafında ifade edilmez, test sırasında incelenir. Diğer komutlar, örneğin rastgele rakam üretici fonksiyonunu çağıran komut gibi, hiç test edilemez. Bu tür komutlar için sadece işlevini doğru şekilde yerine getirip getirmediğine yönelik deneysel gözlem yapılır.

Testin kendisinin koşulacağı çatı Phyton dili ile kodlanmıştır. Bu dil özellikle C dilinde akıllı kart iletişimi için ihtiyaç duyulan alt yapı ve kütüphanelere sahip olduğundan tercih edilmiştir.

Kart'a özel veriler, PIN kodları, sertifikalar modelde kullanılan diğer sembolik veri isimleri basit konfigürasyon dosyalarında hazır tutulmaktadır.

Verilen tüm test dizileri için test çatı uygulaması dizileri ayrıştırarak komutları tek tek akıllı kart terminali üzerinden akıllı karta iletir ve kartın verdiği yanıtın beklenen yanıt olup olmadığını kontrol eder. Eğer uygunsuzluk ile karşılaşılırsa mevcut dizi terk edilerek bir sonraki dizi ile teste devam eder. Ek olarak bu çatı uygulaması, kapsamı ve hataları takip edebilmek için detaylı bir kayıt defteri oluşturmaktadır.

5.4. Değerlendirme

Bölüm 5.2'de belirtilmiş olan fonksiyonel, yapısal ve olasılıklı rastlantısal her bir test şartnamesi için test dizi setleri üretilmiştir.

Test yürütme süresini kısaltabilmek için sadece % 2-3'lük kesimi oluşturan rastgele test dizileri seçilerek örnek vaka çalışmasında kullanılmıştır. Tablo 1'de kullanılan dizilerden elde edilen özet veriler yer almaktadır. Kolonlar sırasıyla test şartlarını, test edilen dizi sayısını, dizilerin ortalama komut sayılarını, model ve kart cevapları karşılaştırıldığında görülen uyumsuzlukları ve test dizilerinde yürütülmüş olan modelde yer alan komut ve kart yanıtı oranını belirten kapsama yüzdesini ifade etmektedir. Farklı test setlerinde yer alan yedeklik sağlayıcı komut sayısı vb. eklenmemiş olduğundan, özet verilerde yer alan kapsam yüzdesi sadece özet tabloda yer alan verilerden oluşmamaktadır. Vaka çalışmasında yer alan komut kapsama oranı % 93 olarak kalmıştır. Modelin detaylı incelenmesi ile geri kalan % 7'lik kısma tekabül eden komut ve kart cevaplarının alınabileceği noktalar ulaşılamayan durumlar olarak kayıt altına alınmıştır. Yazılım testlerinde çıkış kriteri seçimi projenin maliyet, risk ve canlı ortama çıkış tarihlerine göre belirlendiğinden elde edilen kapsama oranı olan % 93 proje yönetimi tarafından uygun bulunmuş ve test başarılı kabul edilmiştir.

Tablo 5.1 Test sonuçları istatistiği özeti

Şartlar	**Dizi sayısı**	**Komut sayısı**	**Uyumsuzluk**	**Kapsama oranı**
Senaryolar, uyumsuzluklar	41	7	14	%60
Dijital imza permutasyonları	322	10	0	%15
Güvenlik ortamı	32	38	20	%40
Kart sahipliği doğrulaması	275	10	10	%49
Model kapsamı	312	5	8	%42
Gereksinim kapsamı	528	7	32	%63
...	**...**	**...**	**...**	**...**
Test Projesi Özeti	**1506**	**8**	**84**	**%93**

Tablo 5.1'de görüleceği gibi, testlerin uygulanması ile model ve kart cevapları arasında birçok uyumsuzluk gözlemlenmiştir. Bu durum beklenen bir durumdur, vaka çalışmasındaki amaç model bazlı testlerde belirli metodların kullanımının ortaya çıkaracağı olumlu yönlerin ifade edilmesini sağlamak ve kavramın kanıtını sunabilmektir.

Ortaya çıkan sonuçlar göstermiştir ki, daha önce insan gücü ile tamamen manuel yöntemler kullanılarak ve insan hatasına açık olarak üretilen test durumları kapsamı ile model bazlı üretilen test durumlarının kapsamı kabul edilebilir sınırlar dahilinde olduğunu göstermiştir.

6. SONUÇLAR

Yazılım geliştirmenin analiz ve şartname safhalarında kullanılan model denetleyici temelli metodlar projelerin test fazlarında oluşan maliyetlerin düşürülmesinde fırsat olarak görülebilir. Ele alınmakta olan birkaç formal metod yazılım testlerinin omurgasını oluşturmaktadır. Teknikler, model denetleyici kullanarak şartnameye uygunluğu doğrulamış ve sisteme uygun olmayan ters örnekleri sağlayarak modele uygun olmayan durumları açığa çıkarmıştır.

Ek olarak, geleneksel endüstriyel kullanımda ortaya çıkan ve model denetleyici araçların hafıza sorunları oluşması sebebiyle çökerek tüm durum uzayını analiz edemeden sonlanmalarına sebep olan durum uzayı patlaması sorununun önüne nasıl geçileceği ele alınmıştır. Bu durum kısmi düzenli ya da diğer durum uzayı azaltım metodlarında dahi meydana gelmektedir. Model denetleme genellikle hataların tespiti ve ters örnek üretiminde ve sonrasında doğrulama adımlarında oldukça verimlidir.

Test durumları üretiminde model denetleyicilerin kullanımındaki potansiyeli görülmüştür. Ancak, halen yazılım test sektöründe tüm alanlarda kullanımı ve bu teknolojinin uygulanabilmesi için yapılması gereken çalışmalar olduğu kaçınılmazdır. Uygulama ve kullanım kolaylığı katılmasının yanı sıra maliyet ve karmaşıklık seviyesinin azaltılması için yapılması gerekenler halen oldukça fazladır. Sektörün kısıtlı bir alanında dahi uygulanmasında görmüş olduğumuz bu faydaların maliyetleri azalmadaki katkısının ise yadsınamaz olduğu tespit edilmiştir.

7. KAYNAKLAR

Sara Baase (2008) *A Gift of Fire*. ABD: Pearson Prentice Hall.

Bogdan Korel (1990) *Automated Software Test Data Generation*. ABD: IEEE Trans. Softw.

Glenford J. Myers (1979) *The Art of Software Testing*. ABD: John Wiley & Sons, Inc.

Alain Abran ve James W. Moore (2004) *SWEBOK Guide to the Software Engineering Body of Knowledge* ABD: IEEE Computer Society

Mats Grindal ve Brigitta Lindström (2002) *Challenges in Testing Real-Time Systems.* İSKOÇYA: EuroSTAR

John B. Goodenough ve Susan L. Gerhart (1975) *Toward a theory of test data selection*. ABD: Uluslararası Güvenilir Yazılım Konferansı

Boris Beizer (1990) *Software testing techniques* ABD:Van Nostrand Reinhold Co.

Richard A. DeMillo, Richard J. Lipton, ve Frederick G. Sayward (1978) *Hints on Test Data Selection: Help for the Practicing Programmer*. ABD: Computer Dergisi

A.Jefferson Offut (1995) *A Practical System for Mutation Testing: Help for the Common Programmer* ABD: ISSE Department George Mason University

P. G. Frankl, S. N. Weiss, ve C. Hu. (1997) *All-uses versus mutation testing: An experimental comparison of effectiveness.* ABD: Journal of Systems and Software, Sayı 38:235–253

Timothy A. Budd and Dana Angluin (1982) *Two notions of correctness and their relation to testing*. ABD: Acta Informatica Sayı 18

Jan Tretmans (1999) *Testing Concurrent Systems: A Formal Approach.* ABD:ConCUR 1999

Marie-Claude Gaudel (1995) *Testing Can Be Formal, Too.* ABD: TAPSOFT

Jan Tretmans (1996) *Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation* ABD:Computer Networks and ISDN Systems

B.A. Myers (2001) *Using Hand-Held Devices and PCs Together* ABD:Communications of the ACM Cilt 44, Sayı 11, s. 34 - 41.

Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, ve Marco Roveri (1999) *NUSMV: A New Symbolic Model Verifier.* İNGİLTERE: 11. Uluslarası Bilgisayar Destekli doğrulama Konferansı , ISBN 3-540-66202-2 Sayfa 495

H. Gümüşkaya ve M.V. Nural (2008) *Service-Oriented Context-Awareness and Context-Aware Services* ABD: Advances in Computer and Information Sciences and Engineering

Edmund M. Clarke ve E. Allen Emerson (1982) *Design and synthesis of synchronization skeletons using branching-time temporal logic.* İNGİLTERE: Programların mantığı atölye çalışmaları Springer-Verlag.

E. Allen Emerson ve Joseph Y. Halpern (1982) *Decision procedures and expressiveness in the temporal logic of branching time.* ABD: 14. Bilgisayar teorisi ACM sempozyumu ACM Press.

Matthew Hennessy ve Robin Milner (1985) A*lgebraic laws for nondeterminism and concurrency*. ABD: ACM

Dexter Kozen (1983) *Results on the propositional mu-calculus*. ABD: Teorik Bilgisayar Bilimleri sayı 27

Edmund M. Clarke, E. Allen. Emerson ve A.P. Sistla (1986) *Automatic verification of finite-state concurrent systems using temporal logic specifications.* ABD: ACM Trans. Program. Lang. Syst. 8, 2, 244–263.

A. Gargantini ve C. Heitmeyer (1999) *Using Model Checking to Generate Test from Requirements Specifications* Fransa:7. Avrupa Yazılım Mühendisliği Konferansı

P. Ammann ve P.E. Black (1999) *Abstracting Formal Specifications to Generate Software Tests via Model Checking* ABD:IEEE, sayı 2, sayfa 10.A.6.1-10

W. Lui ve P. Dasiewicz (2000) *Component Interaction Testing Using Model-Checking* ABD: CSMR

8. ÖZ GEÇMİŞ

Kişisel Bilgiler

Soyadı, adı	: GÖKIRMAK, Macit Giray
Uyruğu	: T.C.
Doğum tarihi ve yeri	: 22.02.1982, İzmit
Medeni hali	: Evli
Telefon	: 0 543 313 3813
e-mail	: giray@gokirmak.gen.tr

Eğitim

Derece	**Eğitim Birimi**	**Mezuniyet tarihi**
Lisans	Haliç Üniversitesi Bilgisayar Mühendisliği Bölümü	2004
Lise	Yeni Ufuklar Amerikan Koleji	1999

İş Deneyimi

Yıl	**Yer**	**Görev**
2004-2005	Teknodizayn Sist.Tic.Ltd.Şti.	Sistem Yöneticisi
2005-2007	Escort Bilgisayar Sist.Tic.Ltd.Şti	Sistem Yöneticisi
2007-2008	T.A.C. Bilgisayar ve Danış.Tic.Ltd.Şti.	Bilgi Güven. Uzm.
2008-2012	İş Bankası Softtech A.Ş.	Yaz. Test Sorum.

Yabancı Dil

İngilizce