YÜKSEK SEVİYE MİMARİ İÇİN DENEYSEL PERFORMANS ANALİZ ÇALIŞMASI

Serhan KARACA [1], Abdülkadir CAN [2], Murat ATUN [3], Burcu YILMAZ [4]

[1,2,3,4] Meteksan Savunma San.Aş, Ankara

[1]e-posta: skaraca@meteksansavunma.com.tr [2] e-posta:acan@meteksansavunma.com.tr

[4]e-posta: byilmaz@meteksansavunma.com.tr [3] e-posta:matun@meteksansavunma.com.tr

Özetçe

HLA [1] dağıtık simülasyon sistemleri için geliştirilmiş genel amaçlı bir mimari olup, temel amacı farklı platformlarda geliştirilen simülasyon sistemleri için birlikte çalışabilirlik altyapısı sunmaktır. HLA performansı; federasyondaki federe sayısına, nesne sayısına ve yapısına, nesnelerin birbiri ile olan etkileşimlerine, kullanılan donanıma ve RTI (Run Time Infrastructure) [2] ürününe, ağ altyapısı gibi farklı kriterlere bağlıdır. Bu bildiri kapsamında yapılan ölçümlerin amacı, HLA (High Level Architecture)/RTI kullanım alternatiflerinin ve farklı federe yapılarının, farklı sayıda federelerin değerlendirilmesi ve en doğru kurulum yaklaşımının belirlenmesidir.

1. Giriş

HLA, birçok simulasyon sisteminin birlikte çalışabilirliğini sağlaması açısından temel alt yapı gereksinimi olmuştur. Bununla birlikte simulasyon sistemleri üzerinde koşturulan unsurların sayısı ve modellerin sadakat seviyeleri ise giderek artmaktadır. Günümüzde farklı ihtiyaçlar için oluşturulmuş sanal, gerçek, yapısal ve C3 sistemleri entegre edilerek [3] eğitim ve tatbikat maksadı ile kullanılabildiği gibi, farklı seviye savaş oyunları (Taktik seviye, operasyonal seviye.) entegre edilerek de kullanılmaktadır [4]. Fakat farklı sistemler entegre edildikçe, kullanıcı sayısı artmakta ve unsur sayısı 100.000 gibi büyük rakamlara çıkmakta, entegrasyon ara katmanı olarak kullanılan RTI'ın performansı sistem performansını önemli ölçüde etkilemektedir. Bu makale kapsamında RTI performansı üzerinde denemeler yapılmış ve RTI performansını etkileyebilecek unsur (nesne) sayısı, nesne özniteliklerinin ve etkileşim parametrelerinin güncellenme sıklığı, federe sayısı gibi faktörler arasındaki ilişkiler bulunmaya çalışılmış, çok kullanıcılı ve yüksek seviyede unsur barındıran simulasyon sistemleri için uygun kurulum ve modelleme alternatifleri test edilmiştir.

2. Test Girdileri

2.1. Denemelerde Kullanılan Nesne Modelleri

Denemeler kapsamında, sistemde silah, sensör ve platform modelleri test edilmiş ve bu modellere ait küçültülmüş bir öznitelik, etkileşim seti kullanılmıştır. Modeller arası etkileşimler Şekil 1'de verilmiştir.

Platform Nesne Modeli

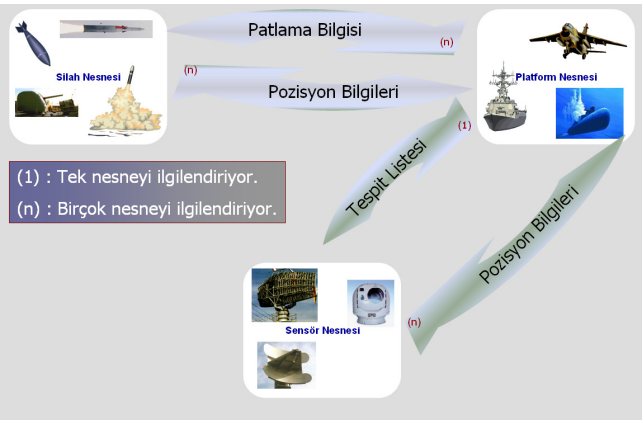
- ***Öznitelikleri*** Tanımlama numarası, adı, bağlı olduğu birlik, durumu, yönü, hızı, pozisyonu.
- ***Davranışları*** Pozisyonunu ve durumunu günceller.
- ***Üye Olduğu Nesneler, Parametreler ve Etkileşimler***
  - Silahlardan gelen patlama etkileşimlerine kayıtlıdır.
  - Sensörlerden gelen tespit listesine kayıtlıdır.
  - Silahların ve sensörlerin pozisyon verilerine kayıtlıdır.
- ***Yayımladığı parametreler*** Pozisyon ve durum değişikliklerini yayımlar.

Sensör Nesne Modeli

- ***Öznitelikleri*** Tanımlama numarası, adı, tipi, üzerinde bulunduğu platformun numarası, durumu, pozisyonu.
- ***Davranışları*** Verilen her zaman adımında tespit listesi günceller.
- ***Üye Olduğu Nesneler, Parametreler ve Etkileşimler***
  - Platformların pozisyon verisine kayıtlıdır.
  - Silahların pozisyon verisine kayıtlıdır.
- ***Yayımdığı Parametreler*** Üzerinde bulunduğu platform için tespit listesi yayımlar.

Silah Nesne Modeli

- ***Öznitelikleri*** Tanımlama numarası, adı, tipi, üzerinde bulunduğu platformun numarası, durumu.
- ***Davranışları*** Verilen her 10 zaman adımında patlama etkile*şimini yayımlar.*
- ***Üye Olduğu Nesneler, Parametreler ve Etkileşimler***
  - Platformların pozisyon değişikliklerine kayıtlıdır.
- ***Yayımladığı Parametreler*** Her 10 zaman adımında patlama etkileşimini ve her zaman adımında kendi pozisyonunu sisteme yayımlar.



Şekil 1 Platform Sensör ve Silah Nenelerinin Etkileşimleri

2.2. Ölçüm Girdileri

Ölçümler için IEEE 1516 onaylı akredite bir RTI ürünü kullanılmıştır. Modelleme sırasında zaman yönetimi olarak zaman adımlı (Time Stepped [5]) zaman yönetimi kullanılmıştır. Ölçümler sırasında RTI aracı tarafından ek olarak sağlanan optimizasyonlar kullanılmamıştır. Ek optimizasyonların kullanılmamasının amacı, ölçümlerde amacın, kullanılan RTI aracının performansının ölçülmesinden çok kurulum ve bahsi geçen diğer alternatiflerin incelenmesidir.

Ürün TCP/IP [6] üzerinden veri güncellemesi yapacak şekilde ayarlanmıştır. UDP [7] protokolü hem doğası gereği TCP'ye oranla daha hızlı bir protokoldür, hem de çoklu gönderim (multicast) imkanı sağlamaktadır. Ancak mesaj kayıplarının önemli olması nedeniyle ve çoklu gönderimin kullanımının yerel ağ dışına çıkışlarda problemler oluşturabileceğinden testlerde TCP bağlantısı tercih edilmiştir.

Birinci konfigürasyonda amaç sabit sayıda unsur içeren bir senaryoda, artan federe sayısının etkisinin değerlendirilmesidir. İkinci konfigürasyonda ise amaç farklı yerleşim (deployment) alternatiflerinin değerlendirilmesidir.

Ölçüm yazılımı açıklamaları daha önce verilen silah, platform ve sensör nesneleri için sarıcı (wrapper) [8] bir uygulamadır. Uygulamaya verilen girdiler sayesinde, tek bir federe üzerinde istenen sayıda nesne yaratılıp koşturulabilmektedir. Bu sayede aynı yazılımla, farklı federe türleri koşturulup denenebilmektedir. Bu sayede ölçüm yazılımın test sonuçlarını etkilemesi engellenmiştir.

2.3. Test Donanımı

Testler sırasında kullanılan test donanımı Şekil 2'de verilmiştir.

Şekil 2 Kullanılan Test Donanımı

3. Ölçümler

3.1. Federe Sayısı ile Bağlantılı Performans Ölçümleri

Bu konfigürasyonda; tanımlı 900 platform, 900 sensör ve 900 silah nesnesi kullanılarak, farklı federe sayıları ile federasyon performansının ölçülmesi amaçlanmıştır. 1. Konfigürasyon kapsamında 3 seçenek için ölçüm yapılmıştır. Bu konfigürasyonda, unsur sayısı ve federe yapısı sabit tutularak, sadece artan federe sayısının etkisi irdelenmiştir. Konfigürasyon ölçümleri sunucularda alınmıştır. İlk seçenekte her biri 900 nesneden oluşan platform, silah ve sensör federeleri mevcuttur. Federasyonda toplam 3 federe, 2700 nesne mevcuttur.

İkinci seçenekte her biri 180 nesneden oluşan beser adet platform, silah ve sensör federeleri mevcuttur. Federasyonda toplam 15 federe, 2700 nesne mevcuttur.

Üçüncü seçenekte her biri 90 nesneden oluşan 10'ar adet platform, silah ve sensör federeleri mevcuttur. Federasyonda toplam 30 federe, 2700 nesne mevcuttur.

1.konfigürasyonuna ait 3 ayrı test seçeneğinin, federe adedi, her federede koşan nesne (unsur) adedi ve federenin koştuğu donanım adedi ve tipi aşağıdaki tabloda verilmiştir.

Tablo 1 Birinci Konfigürasyon test seçenekleri

Seçenek	Platform Federesi		Sensör Federesi		Silah Federesi		Donanım		Top. Federe ve Nesne Ad.	
	F.A	*N.A*	*F.A*	*N.A*	*F.A*	*N.A*	*D.A*	*Tip*	*T.F.A*	*T.N.A*
1.	1	900	1	900	1	900	3	S	3	2700
2.	5	180	5	180	5	180	3	S	15	2700
3.	10	90	10	90	10	90	3	S	30	2700

F.A : Federe adeti
N.A : Nesne adeti
D.A : Donanım adeti
T.F.A : Toplam federe adeti
T.N.A : Toplam nesne adeti
Tip : Federenin çalıştığı donanım tipi (Sunucu (S) , istemci (İ))

1 seçenek,2 seçenek ve 3.seçenekte verilen konfigürasyonların koşum test sonuçları aşağıda verilen tabloda gösterilmiştir. Tabloda her federe için çalışma zamanı ve simulasyonun eş zamanlı ilerleyebildiği ortalama zaman (Federasyon Çalışma Zamanı) gösterilmiştir.

Tablo 2 Birinci Konfigürasyon Ölçüm Çıktıları

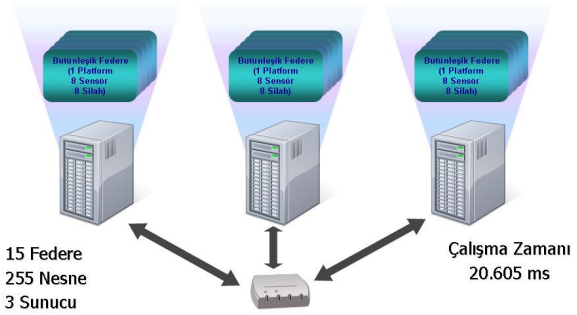
Seçenek	Platform Federesi Çalışma Zamanı (sn)	Sensör Federesi Çalışma Zamanı (sn)	Silah Federesi Çalışma Zamanı (sn)	Federasyon Çalışma Zamanı (sn)
1.Seçenek	0,062649	0,036967	0,011930	0,107000
2.Seçenek	0,032831	0,007007	0,005216	0,603478
3.Seçenek	0,026896	0,007093	0,000676	1,449820

Tablo 2'de verilen ölçüm çıktıları ve elde edilen grafikler incelendiğinde aşağıdaki bulgulara ulaşılmıştır:

- Federe sayısı arttıkça, model çalışma sürelerinin azaldığı ve RTI senkronizasyon faaliyetleri nedeniyle toplam simülasyon zamanının arttığı görülmektedir. Bu artış doğrusal değil, üsseldir (BULGU 1).
- Tablo 1' de silah federesinin çalışma zamanının diğer federelerden düşük olduğu görülmektedir. Bu sebeple, özellikle silah federesi, vaktinin büyük çoğunluğunu zaman senkronizasyonu için diğer federeleri beklemekle geçirmektedir. Bu bekleme, federasyon zamanı ile silah federe çalışma zamanı arası farktan görülebilmektedir (BULGU 2).
- Yine yukarıdaki sonuçlardan hareketle, Platform-Sensör-Silah federeleri ile oluşturulan bir federasyonda, federeleri oluşturan modellerin doğasından gelen farklılıklar nedeniyle, federe çalışma zamanları arasında farklar oluşmuştur. Bu farklar, mevcut yükün (modeller) kaynaklar (işlemciler) arasında uygun dağıtılmadığı veya kaynakların optimum kullanılmadığı sonucunu doğurmaktadır. Federelerin bu şekilde tasarlandığı bir durumda, gerçek bir senaryo işletimi sırasında, angajmanların sıkı olduğu süre boyunca, silah federesi/federeleri diğer federelerin çalışmalarını bekleyecek, angajmanlar serbest bırakıldığı anda ise silah federesinde/federelerinde çok büyük bir yoğunluk oluşacak ve federasyon zamanını silah federesi/federeleri belirleyecektir (BULGU 3).

3.2. Yerleşim Yapısı ile Bağlantılı Performans Ölçümleri

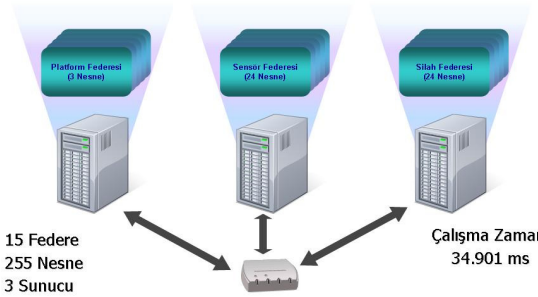
Bu konfigürasyonda; tanımlı 15 platform, 120 sensör ve 120 silah nesnelerinin farklı yerleşim alternatiflerine göre performanslarının denenmesi hedeflenmiştir. Bu ölçümde amaç kullanılabilecek federe sayısı konusunda bir fikir edinmek ve modellerin federelere konuşlandırılma şekillerin avantaj ve dezavantajlarını ortaya koymaktır. Konfigürasyon kapsamında iki seçenek için ölçüm yapılmıştır. Ölçümler sunucular üzerinde ve istemciler üzerinde tekrar edilmiştir. Sunucularda yapılan çalışmalarda iki adet sunucu üzerinde toplam 15 federe çalıştırılmış olup, istemcilerde yapılan çalışmada her istemcide 1 federe koşturulmuştur. Bu konfigürasyonda, unsur ve federe sayısı sabit tutularak, sadece konuşlandırma etkisi irdelenmiştir. İlk seçenekte her biri 17 nesneden (1 Platform, 8 Sensör, 8 Silah) oluşan heterojen federeler mevcuttur. Heterojen (Bütünleşik) ifadesinden kasıt, bir platformun sahip olduğu tüm alt unsurlar (silah ve/veya sensörler) ile aynı federede çalıştırılmasıdır. Federasyonda toplam 15 federe, 255 nesne (unsur) mevcuttur. Bu seçenek için ölçümler hem sunucular üzerinde hem de istemcilerde yapılmıştır. Birinci seçenek Şekil 3' de ve Şekil 4'da verilmektedir.



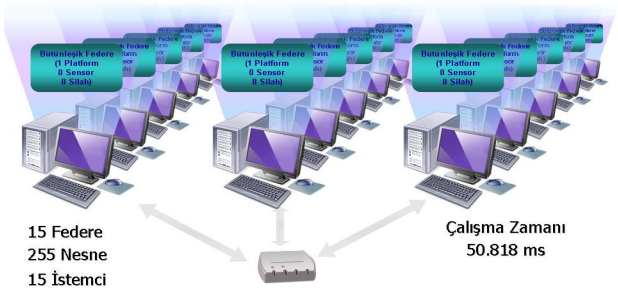
Şekil 3 Sunucularda Konuşlanmış Heterojen Federeler

Şekil 4 İstemcilerde Konuşlanmış Heterojen Federeler

İkinci seçenekte her federenin içinde koşturulan 3 platform nesnesinden oluşan 5 platfrom federesi, her federenin içinde koşturulan 24 nesneden oluşan 5'er adet Sensör federesi ve Silah federeleri mevcuttur. Bu seçenek için ölçümler hem sunucular üzerinde hem de istemcilerde yapılmıştır. Federasyonda toplam 15 federe, 255 nesne mevcuttur. İkinci seçenek Şekil 5 ve Şekil 6' da verilmektedir.



Şekil 5 Sunucularda Konuşlanmış Homojen Federeler



Şekil 6 İstemcilerde Konuşlanmış Homojen Federeler

Tablo 3 İkinci Konfigürasyon Ölçüm Çıktıları

Seçenek	Sunucu Ölçümleri (sn)	İstemci Ölçümler (sn)
Heterojen Federelerle Toplam Çalışma Zamanı	0,020605	0,050818
Platform, Sensor, Silah Federeleriyle Toplam Çalışma Zamanı	0,034901	0,108051

Tablo 3'de verilen ölçümler incelendiğinde aşağıdaki bulgulara ulaşılmıştır:

- Toplam çalışma zamanlarına bakıldığında, federelerin sunucular üzerinde çalıştırıldığı durumda çok daha yüksek performans elde edilmiştir. Sunucu ve istemcilerde heterojen federe toplam zamanları oranlandığında (0.050818 / 0.020605) sonuç yaklaşık 2.4 olarak görünmektedir. Ancak, sunucular üzerinde toplam 8 işlemci üzerinde çalışan 15 federeye karşılık, istemci üzerinde tek işlemci üzerinde tek federe çalışmıştır. Diğer deyişle, sunucular üzerinde işlemci başına çalışan 2 federe ortalama 0.02 sn'de çalışırken, istemciler üzerinde işlemci başına çalışan 1 federe ortalama 0.05 sn'de çalışmaktadır. Bu durumda, gerçek fark, (0.050818 / (0.020605 / 2)) 4.8 kat olmaktadır (BULGU 4).
- Sunucularda elde edilen performanslara bakıldığında, alt unsurların, unsurları ile birlikte aynı federede çalıştığı heterojen federeler ile homojen federeler durumları arasında yaklaşık 1.6 kat fark oluşmuştur. Bu farka, birbiri ile yoğun data alışverişine sahip olan nesnelerin farklı federe içerisinde koşturulduğunda doğan network ek yükünden kaynaklandığı varsayılmıştır(BULGU 5).

4. Sonuçlar

Edinilen bulgular değerlendirildiğinde istemcilerde federe çalıştırılmasının BULGU 1 ve BULGU 4'e istinaden performans açısından sıkıntı yaratabileceği değerlendirilmektedir. Modellerin istemci bilgisayarlarında çalıştırılmasının, ağ problemleri, artan federe sayısı ile orantılı olarak federasyon performansının düşmesi gibi sebeblerle federelerin bu iş için adanmış uygun adetlerdeki sunucularda çalıştırılmasının ihtiyaca göre daha uygun olabileceği sonucuna varılmıştır.

BULGU 2, 3 ve 5'den hareketle, birbiri ile yoğun veri ilişkisi bulunan nesnelerin aynı federe içerisine konuşlandırmanın performans açısından daha uygun olacağı kanatine varılmıştır.

Her federede konuşlu modellerin işletimi paralel olarak yapıldığında bir bilgisayar üzerinde, birden fazla federe çalıştırılmasının herhangi bir performans artışı getirmeyeceği, bu durumda adanmış sunucuların her birinde tek bir federe çalıştırılmasının yeterli olacağı değerlendirilmiştir.

5. Kaynakça

[1] "IEEE 1516, High Level Architecture (HLA)", www.ieee.org, March 2001.

[2] "IEEE 1516, High Level Architecture (HLA)", www.ieee.org, March 2001.

[3]"Modeling Real-Time Distributed Simulation Message Flow in an Open Network", Dennis M. Moen and J. Mark Pullen, 2002.

[4]"Multi-Resolution Modeling in the JTLS-JCATS" Federation, Andy Bowers, David L. Prochnow, 2002

[5] "Parallel and Distributed Simulation",Fujimoto, 1995

[6] http://www.protocols.com,2009

[7] http://www.protocols.com,2009

[8]Design patterns tutorial, URL http://users.csc.calpoly.edu/,1996

Yazılım Klonları ve Klon Belirleme Yöntemleri

Umut Tekin[1] Ural Erdemir[2] Feza Buzluca[3]

[1,2] Ulusal Elektronik ve Kriptoloji Araştırma Enstitüsü, TÜBİTAK-MAM, Kocaeli
[3]Bilgisayar Mühendisliği Bölümü, İstanbul Teknik Üniversitesi, İstanbul

[1]e-posta: umuttekin@uekae.tubitak.gov.tr [2] e-posta: ural@uekae.tubitak.gov.tr
[3] e-posta: buzluca@itu.edu.tr

Özetçe

Yazılımlardaki benzer parçaların bulunması, yazılım kalitesini arttırmaya yönelik çalışmalarda ve telif haklarını ihlal eden, yasadışı kopya yazılımların belirlenmesinde sıklıkla başvurulan bir yöntemdir. Yazılımlardaki benzer parçalar literatürde yazılım klonları olarak adlandırılmaktadır. Çalışma kapsamında yazılım klonu kavramları üzerine geniş bir literatür taraması yapılarak klonların tanımlamaları, sebepleri, çeşitleri ve yazılım kalitesiyle olan ilişkileri açıklanmıştır. Buna paralel olarak yazılımlardaki klonların belirlenmesinde kullanılan teknikler ve prensipler anlatılarak, mevcut ve yeni geliştirilen yazılımların kalitesini arttırmaya yönelik çalışmalarda kullanılabilecek bazı klon belirleme araçları tanıtılmıştır. Tartışma bölümünde ise mevcut çalışmalar değerlendirilerek, eksik yanları ortaya koyulmuş ve ileri ki geliştirmeler hakkında da çeşitli çalışma önerileri sunulmuştur.

1. Giriş

Günümüzdeki yazılımlarda kopya veya benzer kod parçalarının kullanılması oldukça sık karşılaşılan bir olgudur. Açık kaynaklı yazılımların da çoğalmasıyla; yazılım geliştiricileri bir çok problem için hazır yazılmış kodlar bulabilmektedir. Geliştiriciler önceden benzer problemler için yazdıkları veya edindikleri kodları kolaylıkla üzerinde çalıştıkları problemler için kullanabilmektedir.

Yazılımlardaki bu türlü benzer veya aynı kod parçaları yazılım klonları olarak adlandırılmaktadır. Klon için kabul gören en genel tanımlamayı Ira Baxter 2002 yılında şu şekilde yapmıştır: "Klonlar bir benzerlik tanımına göre benzer oldukları belirlenen kod bölmeleridir." Her ne kadar benzerlik tanımı değişik biçimlerde yapılabilse de bu tanım tüm benzerlik tiplerini kapsamakta ve benzerlik tanımının esnek bir biçimde yapılabilmesini sağlamaktadır.

Yazılım klonlarının, yazılım kalitesini kötü yönde etkilediği kesin olarak belirtilemese de genel olarak klonların, yazılımlardaki hataları ve yazılımların bakım maliyetlerini arttıracağı açıktır. Bu nedenle klonlar yazılımda istenmeyen bir olgu olarak karşımıza çıkmaktadır. Yazının ilerleyen bölümlerinde, klonlamanın yazılım üzerindeki kötü etkileri daha geniş kapsamlı olarak anlatılacaktır.

Çalışma kapsamında, yazılımlarda klon kavramı üzerinde geniş bir literatür taraması yapılarak öncelikle klonlamanın nedenleri verilmiştir. Üçüncü bölümde klon tipleri ve özellikleri, dördüncü bölümde ise klonlamanın sonuçları anlatılmıştır. Beşinci, altıncı ve yedinci bölümlerde sırasıyla yazılımlardaki klonların belirlenmesinin yazılım geliştiren şirketlere kazandıracağı faydalar, bu işlemde kullanılan teknikler karşılaştırılmış ve klonları otomatik olarak belirlemek için kullanılabilecek kimi yardımcı araçlardan örnekler verilmiştir. Son olarak mevcut klon belirleme yöntemleri üzerinde tartışılarak konu ile ilgili değerlendirmeler yapılmış, eksikler yeni çalışma alanları belirtilmiştir.

2. Klonlamanın Nedenleri

Yazılımlarda zaman içinde klonların oluşması birçok sebepten kaynaklanabilir. Klonlama nedenlerinin iyi bilinmesi hem klonlamayı önlemede hem de klonların doğru olarak belirlenmesinde yardımcı olacaktır. Kasper ' in [1] büyük yazılımlar üzerindeki yaptığı çalışmalarda şu temel sebepler ortaya çıkmıştır:

Geliştirme ağacının kopyalanması: Kimi zaman bir ürün geliştirilirken, bir noktada yazılım kodu olduğu gibi kopyalanarak aynı kodlar temel alınarak farklı ürünler geliştirilmektedir. Bu durumda kopyalan yazılımların kökenlerinden gelen bir çok kopya veya benzer kod parçası değiştirilmeden veya az miktarda değişiklikle kullanılmaktadır. Bu tür klonlama genelde aynı tipten benzer ürünlerin geliştirilmesi aşamalarında görülmektedir.

Yazılım şablonlarının kullanılması: Bir yazılım probleminin çözüm yöntemini esnek olmayan biçimde tarif eden protokoller şablon olarak adlandırılmaktadır. Yazılım geliştirme aşamalarında, şablonlar sıklıkla kullanılmaktadır. Şablonların kullanılması yazılımlar arasında benzerliklerin artmasını sağlamaktadır. Belirli şablonları defalarca kullanan yazılımcıların bir süre sonra benzer problemler için çok benzer veya tamamen aynı çözümler üretme olasılığı oldukça yükselmektedir.

Uyarlama Yapılması: Var olan yazılımın farklı isteklere göre yeniden uyarlanması veya bir kısmının değiştirilmesi eski ve yeni yazılım arasında klon parçalarının oluşmasına sebep olmaktadır.

Basit Kopyalama: Geliştiriciler karşılaştıkları problemlerin bazıları için yazılımın birçok yerinde kendi yazdıkları veya başka yerden aldıkları kod parçalarını direk kopyalama yöntemiyle tekrar tekrar kullanma bilmektedirler. Bu türlü çok kullanılan yazılım kodlarının kütüphaneler haline getirilip ortak bir yerden kullanılması yerine, direk kopyalama yoluyla farklı yerlerde kullanılması yazılımda klonların oluşmasına sebep olmaktadır.

Zaman Kısıtları: Bazı projelerde yeni sorunlara özgü tasarım ve geliştirme yapmak yerine zaman veya kaynak kısıtları sebep gösterilerek, önceden var olan veya bir şekilde bulunan kodların direk ya da uyarlanarak kullanılması

İstem Dışı: Geliştiriciler yazılım geliştirme esnasında bazen kazara veya tamamen tesadüfi olarak aynı veya benzer kod parçalarının oluşmasını sağlayabilirler.

Lisans İhlalleri: Yasadışı bir işlem olmasına karşı, lisanslı kodların bir şekilde elde edilmesi sonucunda, bunların kopyalanarak veya değiştirilerek kişinin veya kurumun kendi menfaatine kullanılması.

3. Klon Tipleri

Yazılımlarda genel olarak benzerlik fiziksel dünyada olduğu gibi iki tipte olabilmektedir. Yazılımlar yapılarına yani kodlarına veya davranışlarına yani çalışmalarına göre benzer olabilmektedir. Literatürde genel olarak yazılımlar için dört tip klon çeşidinden söz edilmektedir. Bunlardan ilk üç tanesi yapısal benzerlik kategorisine girerken dördüncü tip ise davranışsal benzerlik kategorisindedir. Tablo 1'de bu tiplere ilişkin basit birer örnek verilmiştir.

Tip I: Boşluklar ve format dışarıda tutulmak üzere birbirinin aynısı olan kod parçalarıdır. Genellikle basit kopyalama sonucunda oluşmaktadır.

Tip II: Sadece değişken ve fonksiyon isimleri gibi davranışsal olmayan belirteçlerin isimlerinin değiştirilmiş olduğu kod parçalarıdır. Genellikle direk kopyalamanın saklanmak istendiği durumlarda bu tür klonların oluştuğu görülmektedir.

Tip III: Ufak çapta ekleme çıkarma gibi anlamsal değişikliklerin yapıldığı kod parçalarıdır. Genellikle kopyalandıktan sonra uyarlama yapılmasında durumunda oluşmaktadır.

Tip IV: Yapısal olarak farklı olmalarına rağmen anlamsal olarak ayni işi yapan kod parçalarıdır. Yani bir başka deyişle ayni giriş için her zaman aynı çıkışı üreten yazılım parçalarıdır. Bu tür klonlar belirlenmesi en zor olanlarıdır. Faktöriyel hesabının sıralı ve rekürsif versiyonları bu tipin en güzel örneğidir.

Tablo 1: Klon Tipleri

Tip I	Tip II	Tip III	Tip IV

int a,b;
b = 20;
a = b + 1;
	int a,b;
	b = 20;
	a = b + 1;
		int a,b;
		b = 20;
		a = b + 1;
			int r_fonk(**int** n) {
			if (n == 0) {
			return 1;}
			else {
			return n * f(n - 1);
			}
			}

int a,b;
b = 20;
a = b + 1;
	int x,y;
	x = 20;
	y = x + 1;
		int a,b;
		b = 20;
		int c 0;
		c = a + 5;
		a = b + 1;
			int s_fonk(n){
			int c = 2;
			int R = 1;
			for(; c <=n; c++)
			R = R * c;
			return R;
			}

4. Klonlamanın Sonuçları

Klonlama yapmanın yazılım üzerinde birçok sonuçları olmaktadır. Yazılımlarda yeniden kullanılabilirlik istenen bir olgu olmasına rağmen. Klonlama genel olarak istenmeyen tekrarlamalar olarak nitelendirilmektedir. Klonlamanın en önemli kötü etkilerini yazılımın kalitesinde, yeniden kullanabilirliğinde ve bakımında görmekteyiz. Bu etkilerden bazılarını sırlarsak;

Hata tekrarlamaları: Kopyalan kod parçalarındaki hatalar yazılımların bir çok yerinde defalarca tekrarlanmakta ve bakım maliyetlerini arttırmaktadır. Bakım maliyetlerinin artmasının en temel sebebi bir kopyada düzeltilen hataların otomatik olarak diğerlerine aktarılamaması, dolayısıyla da bu hatalarla defalarca farklı yerlerde karşılaşılmasıdır.

Hata olasılığın artması: Genellikle bir kod kopyalandıktan sonra bir takım düzenlemelere tabi tutularak yeni yerinde kullanılmaktadır. Bu düzenlemeler kopyalanan kodun mevcut yapısına uymaya bilir ve bu zorlama yeni hataların oluşmasına sebep olabilmektedir.

Kötü tasarımların tekrarlanması: Kopya kod parçasının mevcut kötü tasarımı kopyalandığı yere taşınmakta tekrar kullanılmasına sebep olmaktadır. Bu durum yeni ve daha iyi tasarımların yapılmasının önünü tıkamaktadır ve yazılımın kalitesini düşürmektedir.

Adaptasyon işlemlerinin zorlaşması: Kopyalanan kodların yeni isteklere göre adapte edilmesi oldukça zahmetli bir iştir. Önce kopya kısmın anlaşılması daha sonra adapte edilmesi gerekir ki bu süre genellikle o kısmın yeniden yazılma süresinden fazladır.

Kullanılan kaynak miktarının artması: Yazılım içinde yapılan kopyalamalar gereksiz yere kaynak kullanımına ve iş tekrarlarına yol açarak kaynak israfına sebep olmaktadır. Bu kopyaların bir fonksiyonda veya makroda birleştirilmesi, kaynak kullanımını azaltacaktır.

Lisans ihlalleri: Bilerek veya bazı durumlarda istenmeyerek de olsa, lisanslı kodların kullanılması hem geliştiricileri hem de kurumları zor durumların içine sokabilmektedir. Lisans ihlallerinin basit ama sık rastlanan örneği öğrenci ödev ve projelerindeki kopyalardır. Bu kopyaların bulunması yazılım mühendisliği eğitim disiplinleri için önemlidir.

5. Klonların Belirlenmesinin Faydaları

Var olan veya geliştirme aşamasındaki yazılımların incelenerek klonların belirlenmesinin yazılımın kalitesini ve bakılabilirliğini arttırmak adına bir çok faydası bulunmaktadır. Bunları kısaca özetleyecek olursak:

Kütüphaneleştirme: Yazılımdaki kopya kısımlar belirlenerek, bu kısımlar hatasız çalışan kütüphaneler haline getirilebilir. Bu kütüphaneler hem tekrar kullanıla bilirliği arttıracak hem de bakım maliyetlerini azaltacaktır. Dolayısıyla aynı hatanın yazılımın farklı yerlerinde görülme olasılığı azalacaktır.

Bakım maliyetlerinin azaltılması: Benzer veya aynı kod parçalarının yerlerinin bulunması ve mantısal olarak bağlanması yazılım klonları üzerinde yapılacak olası değişikliklerin diğer kopyalara da yansıtılmasını otomatikleştirecek ve kolaylaştıracaktır. Böylece bir kod parçasında yapılacak düzeltmeler otomatik olarak kopyalarına da yansıtabilecektir.

Kaynak kullanımın azaltılması: Kopyalanmış kısımlar birleştirilmesi hem kaynak kodun hem de çalışabilir yazılımın.bellek ve işlemci gibi kaynakların kullanımını daha verimli hale getirecektir, büyük projelerde derleme zamanı da

ciddi anlamda azalacaktır. Bu özellikle gerçek zamanlı ve gömülü sistemler oldukça önemlidir.

Hataların bulunması: Hatalı olduğu bilinen veya belirlenen kod parçalarının benzerlerinin bulunması yazılımdaki olası hataların önceden bulunarak temizlenmesini sağlayacaktır.

Yasa dışı kopyalamaların bulunması: Mevcut yazılım içinde bir kişi veya kuruma ait lisanlı yazılım parçalarının kopyalarının tespit edilerek, yasa dışı kullanımlarının önüne geçilebilir. Bu özellik hem bu tür kodları yanlışlıkla kullanılmasını önleyecek hem de lisans sahiplerinin kodlarının nerelerde kullanıldığı otomatik olarak çıkartmalarını sağlayacaktır. Böylece haksız rekabetler de önlenebilir. Bazı üniversitelerin bilgisayar mühendisliği bölümlerinde ödev kontrollerinde klon belirleme yöntemleri kopya kodların bulunması amacıyla kullanılmaktadır.

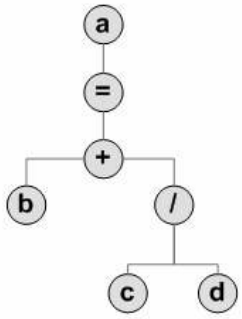
6. Klon Belirleme Teknikleri

Literatürde bu güne kadar klonların otomatik olarak belirlenmesi için birçok program ve algoritma geliştirildiği görülmektedir. Bu programlar kullandıkları tekniklere göre sınıflandırılmaktadır.

Yazı tabanlı teknikler [2]: Bu teknikte yazılım kodlarındaki cümleler ve cümle grupları yazım tabanlı olarak karşılaştırılmakta ve benzerlikleri ortaya çıkartılmaktadır. Genel olarak Tip 1 klonların bulunmasında etkin olarak kullanılabilen bir teknik olarak öne çıkmaktadır.

Simge tabanlı teknikler [3]: Bu teknikte yazılım kodları derleyicilerin yaptığı gibi tek tip simgelere dönüştürerek karşılaştırılmaktadır. Simge tabanlı teknik, Tip 1 ve Tip 2 çeşidinden klonların bulunmasında etkin olarak kullanılabilmektedir. Yazı tabanlı tekniğe göre daha etkin olan bu yöntem isim değişikliği gibi uyarlamalardan etkilenmemektedir.

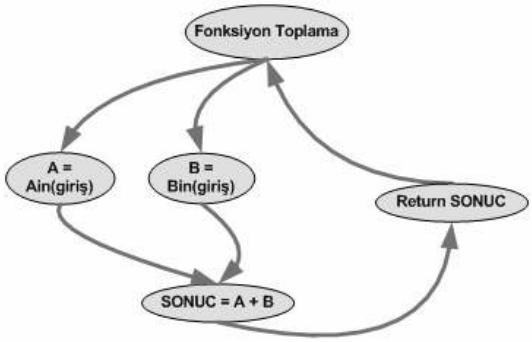
Sentaks ağacı tabanlı teknikler [4]: Bu teknik kapsamında yazılımların soyut sentaks ağaçları (SSA) çıkartılarak karşılaştırılmaktadır. SSA kodun yazımından bağımsız bir veri yapısı olduğu için yazım değişikliklerinden etkilenmemektedir. Tip 1, Tip 2 ve Tip 3 klonların bulunmasında bu yöntem kullanılabilmektedir. Şekil 1'de örnek bir kod cümlesi için oluşturulan SSA ağacı verilmiştir.



Şekil 1: "a = b+c/d" için örnek SSA.

Bağımlılık Grafı tabanlı teknikler [4]: Bağımlılık grafları [6] programda bulunan veriler ve akış arasındaki bağımlılıkları ifade eden graflardır. SSA ağaç graflarına göre daha sade olan bu grafların karşılaştırılmaları da algoritmik olarak daha hızlıdır. Özellikle Tip 3 cinsinden klonların belirlenmesinde kullanılan etkin bir tekniktir. Şekil 2'de örnek bir kod için bağımlılık grafı verilmiştir.

```
int toplama (int Ain, int Bin){
    int A = Ain;
    int B = Bin;
    int SONUC = A+B;
    return SONUC;
    }
```



Şekil 2: Örnek bağımlılık grafı.

Metrik tabanlı teknikler [7]: Metrik tabanlı tekniklerde fonksiyon çağırma sayısı, kod satır sayısı vb.. yazılım metrikleri bulunarak karşılaştırılmaktadır. Yazılım klonları metriklerle de belirlenebilir, metriklerin yazılım özellikleri ve kalite ile olan ilişkisi [8] 'de yapılan çalışmada ortaya koyulmuştur. Metrik tabanlı teknikte karşılaştırma diğer yöntemlere göre daha kolay ve çok hızlı olsa da yanlış sonuçların elde edilme olasılığı oldukça yüksektir. Genel olarak elde edilen ilk sonuçların değerlendirilmesi için dışarıdan insan müdahalesine ihtiyaç duyulmaktadır.

Hibrit yöntemler: Tüm bunların yanı sıra yukarıda anlatılan tekniklerden birden fazlasının birlikte kullanılarak klonların belirlenmesi mümkün olabilmektedir. Bu sayede hem başarım hem de klon çeşitleri bakımından kapsam arttırıla bilmektedir. Örneğin SSA ve simge tabanlı tekniklerin birleştirerek çok daha etkin bir belirleme yöntemi oluşturulabildiği görülmüştür [9].

Klon belirleme tekniklerinde başarım, kapsam ve algoritmik hız gibi özellikler oldukça önemlidir. Başarım tekniğin doğru sonuçları bulabilme yeteneğidir. Yazı ve simge tabanlı tekniklerde başarım oldukça yüksekken bu tekniklerin kapsamları dardır, çünkü bu teknikler sadece Tip 1 ve/veya Tip 2 çeşidinden klonların bulunmasında kullanılabilmektedir. Bağımlılık grafı ve SSA tabanlı tekniklerin ise kapsamı oldukça geniş ancak algoritmik olarak çalışma hızları düşüktür. Metrik tabanlı yöntemlerin ise yanlış sonuçlar döndürme olasılığı diğerlerine göre çok yüksek olmaktadır.

Tablo 2'de de gösterildiği gibi yapılan bir karşılaştırma çalışmasında, bu teknikleri kullanan programların hız ve bellek kullanımına göre performansları "--" den "++" ya kadar göreceli olarak derecelendirilmiştir.

Tablo 2: Tekniklerin Karşılaştırılması [10]

	Yazı Tabanlı	Simge Tabanlı	SSA Tabanlı	Bağımlılık Tabanlı	Metrik Tabanlı
Hız	-	++	-	- -	++
Bellek	-	+	-	+	++

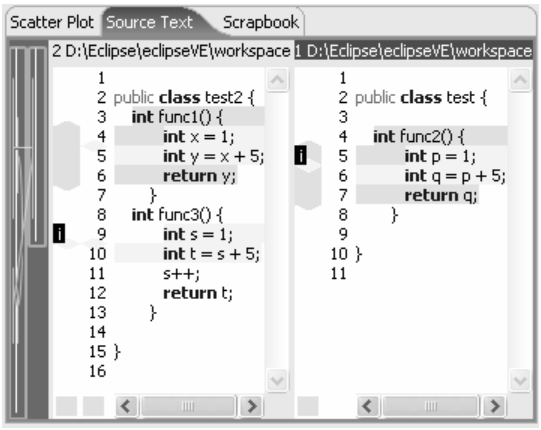
7. Klon Belirleme Araçları

Yazılımlarda bulunan klonların otomatik olarak belirlenebilmesi için birçok yazılım aracı geliştirilmiştir. Bu yazılım araçların bazıları ticari uygulamalarken bazıları ise açık kaynak kodlu ücretsiz programlardır. Yazılımların geliştirilmesi esnasında bu yardımcı araçlardan faydalanılması klonların önlenmesi veya en azından belirlenmesi için kullanılabilirler. Bu bölümde bu yazılım araçlarından en çok kullanılanları kısaca tanıtılmıştır.

DUP [11]: Akademik lisansa sahip olan program yazı tabanlı teknikleri kullanarak yazılım klonlarını belirlemektedir. Özellikle tip 1 klonları hızlıca bulabilen araç Brenda S. Baker tarafından geliştirilmiştir [12].

JPLAG [13]: Simge tabanlı tekniği kullanan akademik lisanslı programdır. Tip 1 ve tip 2 klonların belirlenmesi amacıyla kullanılabilir.

CCFinder [14] [15]: Simge ve yazı tabanlı teknikleri kullanan CCFinder, Toshihiro Kamiya tarafından geliştirilmiş bir başka akademik lisansa sahip programdır aynı zamanda ticari olarak da pazarlanmaktadır . Şekil 3'de CCFinder programının bir ekran görüntüsü verilmiştir.



Şekil 3: CCFinder ekran görüntüsü.

CloneDr [4]: SSA ve simge tabanlı yöntemleri birleştiren CloneDr ticari bir uygulamadır. Ira D. Baxter tarafından geliştirilmiştir.

Duplix [16]: Jens Krinke tarafından geliştirilen ve bağımlılık ağacı karşılaştırma tekniğini kullanan bir programdır. Tip 3 klonların bulunmasında kullanılabilir.

Microsoft Phoenix [9]: Microsoft tarafından geliştirilen ticari bir uygulama olup SSA ve simge tabanlı teknikleri birleştirerek kullanmaktadır.

CloneTracker [17]: Eclipse eklentisi olarak geliştirilmiş olan açık kaynak kodlu program yazı tabanlı karşılaştırma tekniğini kullanmaktadır.

8. Tartışma

Tartışma bölümünde klon belirleme konusu üzerine yapılan çalışmalar incelendiğinde görülen eksikler ve gelecekte yapılabilecek çalışmalar hakkında bazı fikirler sunulmuştur.

Yazılım klonlarının bulunması için günümüze kadar kısıtlı sayıda yöntem ve araç geliştirilmiştir. Bu yöntemlerin birçoğunun klon çeşitlerine göre kapsamları geniş değildir ve genellikle tip 1 ve tip 2 klonların belirlenmesinde kullanılmaktadır. Bu yöntemlerden iki veya daha fazlasının birleştirilerek kapsamlarının arttırılması sağlanmakta ancak bu da yazılım üzerinde yapılması gereken tarama sayısını arttırarak gereksiz iş tekrarlarına sebep olmaktadır. Bu konuda klon çeşidi bakımından kapsamı daha geniş yeni yöntemlerin araştırılması gerekmektedir.

Özellikle tip 3 klonların belirlenmesi için geliştirilen graf tabanlı algoritmaların karmaşıklığı polinom zamanlı değildir ve graf boyuna göre üstel olarak artmaktadır. Bu sorun tip 3 klonların belirlenmesini zorlaştırmakta hatta büyük yazılımlar için bazen bu işlemi imkansız hale getirmektedir. Bu algoritmaları hızlandırmak için geliştirilmiş kimi sezgisel yöntemlerin uygulandığı görülse de bu alan klon belirleme konusunda yeni araştırmaların ve sezgisel yaklaşımların önünü açmaktadır. .

Algoritmalara bağlı geliştirilmiş birçok yardımcı aracın programlama diline bağımlı olduğu görülmektedir. Bu kısıt araçların her türlü yazılım üzerinde denenmesine imkan vermediği gibi yazılımlar arası benzerliklerin de bulunmasını zorlaştırmaktadır. Programlama dilinden bağımsız klon belirleme araçlarının ve yöntemlerinin geliştirilmesine ihtiyaç duyulmaktadır.

Geliştirilen klon belirleme tekniklerini yazılımdaki benzer parçaları bulurken, büyük oranda da benzer olmayan parçaları, klon olarak nitelendirebilmektedir. Bu durum otomatik çalışmayı zorlaştırmakta ve bazı aşamalarda uzman insanların müdahalesi gerektirmektedir. Yanlış sonuç döndürme olasılığı az olan yöntemlerin geliştirilmesi insan müdahalesine gerek olmadan çalışacak otomatik belirleme yöntemlerinin önünü açacaktır.

Yazılımlarda tip 4 klonların belirlenmesi zor bir konu olarak nitelendirilmektedir ve üzerine pek çalışma yapılmadığı görülmektedir. Ancak bu tür klonların belirlenmesi özellikle gömülü sistemler için hem kaynak kullanımı azaltacak hem de yazılımları sadeleştirecektir. Dinamik yazılım metrikleri gibi yazılımın çalışma zamanını ifade eden özelliklerden faydalanılarak bu tür klonların belirlenmesi için yeni yöntemler geliştirilebilir.

Özellikle endüstriyel yazılım projelerinde klonların etkileri ve gelişimleri gibi konuların pek incelenmediği görülmektedir. Bu tür konuların incelenmesi sonucunda elde edilecek bilgiler klonların yazılımlar üzerindeki kötü etkilerini anlamamıza yardımcı olacağı gibi aynı zamanda klon belirleme tekniklerinin de başarımını arttıracaktır.

9. Sonuç

Günümüzde yazılım bakım ve üretim maliyetleri hızla artmaktadır. Yazılımlardaki klonların belirlenmesi ve klonlamanın önlenmesi, yazılım maliyetlerini azaltmada ve yazılım kalitesini arttırmada büyük rol oynamaktadır. Ayrıca lisansız kullanımların tespit edilmesiyle, yazılımda haksız rekabetin önüne geçilebilir. Klonlar belirlenmesi zor ancak belirlendikten sonra düzeltilmesi oldukça kolay kod kusurlarıdır.

Bu çalışma kapsamında yazılım klonları konusunda temel kavramlar üzerine geniş çerçeveli bir araştırma yapılarak, klon tanımlamaları yapılış, klon çeşitleri ve nedenleri belirtilerek, klonların belirlenmesinin faydaları anlatılmıştır. Literatürde bulunan mevcut klon belirleme teknikleri ve yardımcı araçları şu anki durumları da değerlendirilerek anlatılmıştır. Klonlama konusunda var olan eksikler ve gelecekte varılması gereken noktalar tartışılmıştır.

Çalışma, aynı zamanda yazılımda benzerliklerin bulunması kavramı ile ilgilen endüstri için yön verici ve bilgilendirici olacak şekilde hazırlanmıştır. Yazılım klonlarının doğru anlaşılması ve daha iyi otomatik belirleme yöntemlerinin geliştirilmesi için hem akademiye, hem de endüstriye büyük görevler düşmektedir.

9. Kaynakça

[1] Kapser, C., Godfrey, M.W.: ”clones considered harmful” considered harmful. In: Working Conference on Reverse Engineering. (2006)

[2] Brenda S. Baker. A Program for Identifying Duplicated Code. In Proceedings of Computing Science and Statistics: 24th Symposium on the Interface, Vol. 24:4957, Mart 1992.

[3] Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. Transactions on Software Engineering, Vol. 28(7): 654-670, Haziran 2002

[4] Ira Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant Anna. Clone Detection Using Abstract Syntax Trees. In Proceedings of the 14th International Conference on Software Maintenance (ICSM'98), pp. 368-377, Bethesda, Maryland, Kasım 1998

[5] Raghavan Komondoor and Susan Horwitz. Using Slicing to Identify Duplication in Source Code. In Proceedings of the 8th International Symposium on Static Analysis (SAS'01), Vol. LNCS 2126, pp. 40-56, Paris, France, Haziran 2001

[6] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst., 9(3):319349, 1987

[7] Jean Mayrand, Claude Leblanc, Ettore Merlo. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In Proceedings of 104 the 12th International Conference on Software Maintenance (ICSM'96), pp. 244-253, Monterey, CA, USA, Kasım 1996.

[8] U. Erdemir, U.Tekin, F.Buzluca, "Nesneye Dayalı Yazılım Metrikleri ve Yazılım Kalitesi" Yazılım Kalitesi ve Yazılım Geliştirme Araçları Sempozyumu (YKGS08), İstanbul, 2008

[9] Robert Tairas, Je Gray. Phoenix-Based Clone Detection Using Suffix Trees. In Proceedings of the 44th annual Southeast regional conference (ACM-SE'06), pp. 679-684, Melbourne, Florida, USA, Mart 2006

[10] Bellon, S., Koschke, R.: Comparison and evaluation of clone detection tools. IEEE Computer Society Transactions on Software Engineering , 2006

[11] Brenda S. Baker. A Program for Identifying Duplicated Code. In Proceedings of Computing Science and Statistics: 24th Symposium on the Interface, Vol. 24:4957, Mart 1992.

[12] Baker, B.S.: On finding duplication and near-duplication in large software systems. InWills, L., Newcomb, P., Chikofsky, E., eds.: SecondWorking Conference on Reverse Engineering, Los Alamitos, California, IEEE Computer Society Press (1995) 86–95

[13] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding plagiarisms among a set of programs with JPlag. In Journal of Universal Computer Science, 8(11):10161038, Kasım 2002

[14] Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. Transactions on Software Engineering, Vol. 28(7): 654-670, Haziran 2002

[15] Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: A Multi-Linguistic Tokenbased Code Clone Detection System for Large Scale Source Code. IEEE Computer Society Transactions on Software Engineering 28(7) (2002) 654–670

[16] Jens Krinke. Identifying Similar Code with Program Dependence Graphs. In Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01), pp. 301-309, Stuttgart, Germany, October 2001

[17] Ekwa Duala-Ekoko and Martin P. Robillard: CloneTracker: Tool Support for Code Clone Management. In Proceedings of the 30th ACM/IEEE International Conference on Software Engineering, May 2008, Formal Research Demonstration