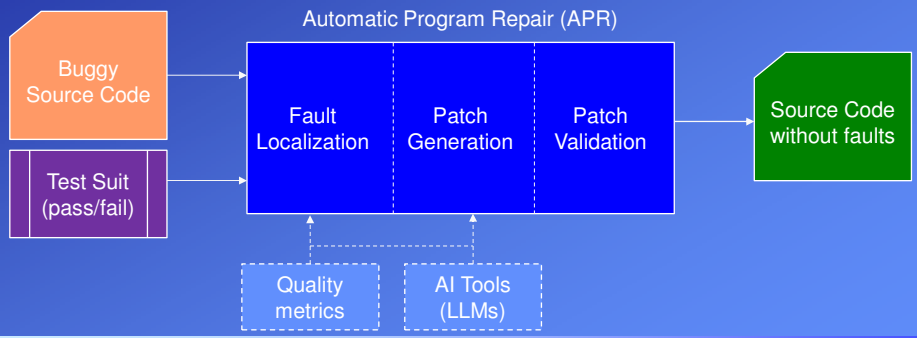
Otomatik Program Tamiri (*Automatic Program Repair -APR*)

- Amaç,
  - programların çalışması sırasında arızalara (*failure*) neden olan
  - kodlama hatlarını (*fault*) belirleyip
  - bu hataları düzeltmek (*patch generation*)
  - ve bu işlemleri doğrudan insan eforuna gerek duymadan (*automatic*) yapmak.



Spektrum temelli hata yeri bulma yöntemi (*Spectrum-Based Fault Localization- SBFL*):

- **Arıza (*failure*):** Sistemin beklenen davranışı göstermemesi, beklenen hizmeti yerine getirmemesi.
- **Kod hatası, kod kusuru (*fault, bug*):** Çalışma zamanı arızaya neden olan kod hataları.
  Hatırlatma: Önceki bölümlerde ele alınan tasarım kusurları (*defect*) doğrudan çalışma zamanında arızalara neden olmuyorlardı.
  O tip kusurlar tasarım kalitesini düşürerek sürdürebilirlik (*maintainability*) sorunlarına neden oluyorlardı.
  Bu bölümde ele alınan kod hataları ise çalışma zamanı arızlara neden olmaktadır.
- **Program spektrumu:** Bir programın belli koşullar altında çalışması sırasında etkin olan birimleri ile ilgili bilgilerin toplamıdır.
  - Programların birimleri farklı çözünürlüklerde ele alınabilir:
    - Program satırı
    - Blok: `if-then-else`, `for`, `while`, C/C++ programlarında `{ }` arası, ...
    - Fonksiyon
    - Sınıf
  - Bir program belli koşullar altında çalıştırıldığında hangi satırın, bloğun veya hangi fonksiyonun kaç defa etkin olduğu (çağırıldığı) o programın spektrumunu oluşturur.
  - Programlardaki kod hataları belirlenirken bu birimler dikkate alınır ve hatalar bunların içinde aranır.
  - Bazı hata bulma yöntemleri, birimlerin kaç defa etkin olduğunu değil sadece etkin olup olmadığını dikkate alır.

Spektrum temelli hata yeri bulma yöntemi (*Spectrum-Based Fault Localization- SBFL*):

Spektrumun oluşturulması:

- Program değişik verilerle çalıştırılarak test edilir.
- Arızaya neden olan (*failed*) ve arıza oluşturmayan (*passed*) testlerde program spektrumu çıkartılır.
- Hangi birimin (satır, blok veya fonksiyon) hangi testlerde kapsandığını gösteren bir spektrum matrisi oluşturulur.

Örnek:

Dört bloktan oluşan bir program parçasına beş test uygulanıyor.

Her test için programın spektrumu oluşturuluyor.

Blok3, Test_1'de kapsanmış.

Blok4, Test_1'de kapsanmamış.

Program, Test_1'den geçmiş.

Program Test_2'de hataya neden olmuş.

Test_1'in blok spektrumu

	Blok1	Blok2	Blok3	Blok4	Arıza?
Test_1	X	X	X	-	Pass
Test_2	X	X	-	X	Fail
Test_3	X	X	X	-	Pass
Test_4	X	X	-	-	Pass
Test_5	X	X	-	X	Fail

Aranan: Arızaya neden olan kodlama hatası en yüksek olasılıkla hangi bloktadır?

(Spectrum-Based Fault Localization- SBFL) (devamı):

Spektrum matisi:

- Spektrum matrisinin her satırında bir testle ilgili spektrum bilgisi ikili sayı olarak (0/1) yer alır. Sütunlarda ise incelenen birimler (satır, blok, fonksiyon) bulunur.
- Matrisin x_{ij} elemanı i. testte j. birimin kapsanıp kapsanmadığı bilgisi yer alır.
  - $x_{ij} = 0$ ise i. testte j. birim kapsanmamıştır.
  - $x_{ij} = 1$ ise i. testte j. birim kapsanmıştır.
- Ayrıca her bir test için sonuçta arıza oluşup oluşmadığını gösteren bir sütun vektör oluşturulur.
  - $e_i = 0$ ise i. testte arıza oluşmamıştır (*passed*).
  - $e_i = 1$ ise i. testte arıza oluşmuştur (*failed*).
- Spektum matrisi ve arıza vektörü:

  M adet test, N adet birim (örneğin blok):

N adet birim

M adet test

$$\begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1N} \\ x_{21} & x_{22} & \cdots & x_{2N} \\ \vdots & \vdots & \cdots & \vdots \\ x_{M1} & x_{M2} & \cdots & x_{MN} \end{pmatrix} \quad \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_M \end{pmatrix}$$

Örnek:

$x_{12} = 0$ ise 1. testte 2. birim kapsanmamıştır.

$e_1 = 1$ 1. testte arıza tespit edilmiştir.

(*Spectrum-Based Fault Localization- SBFL*) (devamı):

Benzerlik katsayıları (*similarity coefficients*):

- Arızaya neden olan spektrumlarda yer alıp arıza oluşturmayan spektrumlarda bulunmayan birimlerin (satır, blok veya fomksiyon) hata içerme olasılıkları daha yüksektir.
- Birimlerin hata (*fault*) içerme olasılıklarını belirlemek için j. birimin kapsanma vektörü (x_{1j}, …, x_{Mj}) ile arıza vektörü (e_1, …, e_M) arasında benzerlik (s_j) hesaplanır.
- İkili sayılardan oluşan vektörler arasında benzerlik hesaplayan yöntemler aşağıdaki sayaçları kullanırlar:

Sayaç:	x_{ij}	e_i	
$a_{00}(j)$	0	0	j. birim i. testte kapsanmamıştır ve i. testte arıza oluşmamıştır.
$a_{01}(j)$	0	1	j. birim i. testte kapsanmamıştır ve i. testte arıza oluşmuştur.
$a_{10}(j)$	1	0	j. birim i. testte kapsanmıştır ve i. testte arıza oluşmamıştır.
$a_{11}(j)$	1	1	j. birim i. testte kapsanmıştır ve i. testte arıza oluşmuştur.

- **Jaccard benzerlik katsayısı:** Veri kümelemede kullanılan bir yöntemdir.

$$s_j = \frac{a_{11}(j)}{a_{11}(j) + a_{10}(j) + a_{01}(j)}$$

- **Ochiai benzerlik katsayısı:** Moleküler biyolojide genetik benzerlik için kullanılmaktadır. SBFL konusunda başarılı olduğu gösterilmiştir.

$$s_j = \frac{a_{11}}{\sqrt{(a_{11} + a_{01}) \times (a_{11} + a_{10})}} = \frac{a_{11}}{\sqrt{\text{Toplam\_arıza\_sayısı} \times (a_{11} + a_{10})}}$$

R. Abreu, P. Zoeteweij, and A. Van Gemund, "An Evaluation of Similarity Coefficients for Software Fault Localization," PRDC'06, 2006.

(*Spectrum-Based Fault Localization- SBFL*) (devamı):

Hatalı birimin belirlenmesi:

Örnek: Yansı 11.3'teki testler ve bloklar

	B1	B2	B3	B4	
T1	1	1	1	0	0
T2	1	1	0	1	1
T3	1	1	1	0	0
T4	1	1	0	0	0
T5	1	1	0	1	1

Ochiai benzerlik katsayıları:

- $s_{B1} = \frac{2}{\sqrt{(2+0)(2+3)}} = 0.63$
- $s_{B2} = \frac{2}{\sqrt{(2+0)(2+3)}} = 0.63$
- $s_{B3} = \frac{0}{\sqrt{(0+2)(0+2)}} = 0$
- $s_{B4} = \frac{2}{\sqrt{(2+0)(2+0)}} = 1$ En yüksek şüphe değerine sahip olan blok

SBFL Yöntemlerinin Başarımı:

- Kolay uygulanabilir yapısı sayesinde SBFL yaygın olarak kullanılmakla beraber bazı durumlarda hataların yerini bulmakta başarısız olmaktadır.
- Sadece testlerin sonuçlarına (*passed*/*failed*) ve kapsadıkları bloklara bakmak yeterli olmaz. Başarılı (*passed*) ve başarısız (*failed*) testler her zaman hatanın kaynağını gösterecek şekilde oluşmazlar.
- Hatalı birimleri belirlemek için ek bilgiler de kullanan ve makine öğrenmesinden yararlanan yöntemler de geliştirilmiştir.

Öğrenme Temelli Hatalı Birim Belirleme Yöntemleri:

- Hatalı birimleri belirlemede başarımı artırmak için uygulanan yöntemler:
  - Ek bilgiler kullanımı: Test sonuçları ve kapsamaya ek olarak başka veriler de kullanan yöntemler geliştirilmiştir.

    Ek bilgiler: kod metrikleri, evrimsel (değişim) metrikleri, geçmiş hata raporları vb.
  - Kodun ayrıntılı gösterilimi: Hata aranan kodların AST'den (*Abstract Syntax Tree*) elde edilen bilgilerle ayrıntılı olarak ele alınması.

    Bu sayede kodun yapısı, ifadelerin anlamları da hata yeri bulmada kullanılabilmektedir.
  - Ayrıntılı bilgilerin makine öğrenmesi (derin öğrenme) yöntemleri ile işlenmesi.

Örnekler:

- Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. "**DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization.**" 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, July 15-19, 2019, ACM 169-180.
  https://doi.org/10.1145/3293882.3330574
- Y. Lou et al., "**Boosting coverage-based fault localization via graph-based representation learning**," the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ACM, Aug. 2021, pp. 664-676.
  doi: 10.1145/3468264.3468580

Boosting Coverage-Based Fault Localization via Graph-Based Representation Learning

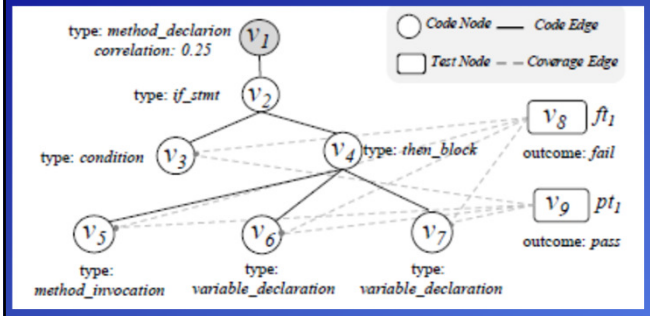
GRACE: Graph-based representation learning to fully utilize coverage information

1. Kod birimleri (deyimler veya kod blokları), testler ve aralarındaki ilişkiler tümleşik bir graf şeklinde temsil ediliyor.
   - Kodun yapısını temsil etmek için AST (*Abstract Syntax Tree*) kullanılıyor.
   - Gafın düğümleri: Kod birimleri ve testler.
   - Grafın kenarları: Kod birimleri arasındaki bağlantılar, kod birimleri ile testler arasındaki bağlantılar
   - Amaçlar,
     - Testler ile kapsanan kod arasındaki ilişkiyi daha iyi temsil etmek
     - Farklı kod deyimlerinin hatalar üzerindeki etkisini de dikkate almak
2. Oluşturulan graf Gated Graph Neural Network (GGNN) modeliyle incelenerek şüpheli metodlar belirleniyor.
3. Yöntemin başarımı Defects4J üzerinde ölçülüyor.

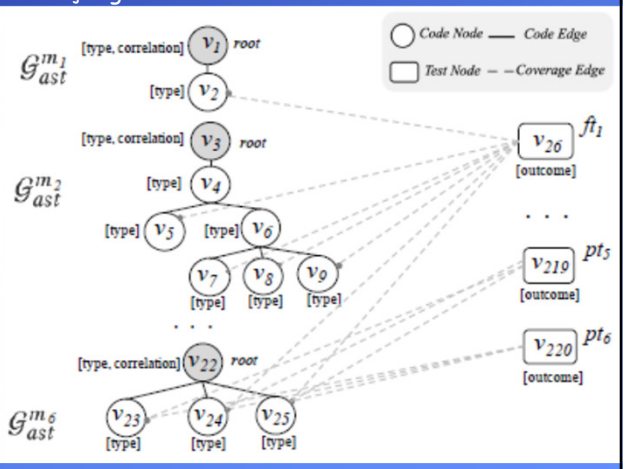
   R. Just, D. Jalali, and M. D. Ernst, "Defects4J: a database of existing faults to enable controlled testing studies for Java programs," the 2014 International Symposium on Software Testing and Analysis, ACM, July 2014, pp. 437-440.
   doi: 10.1145/2610384.2628055
   https://github.com/rjust/defects4j

GRACE (devamı):

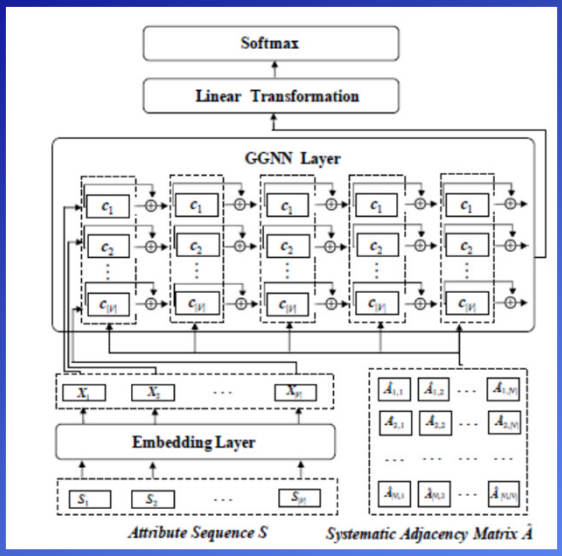
Metod m2 için oluştrulan graf



Lang-47 hatasının yerini belirlemek için oluşturulan tümleşik graf



GRACE (devamı):



Yöntem:	Top-1	Top-3	Top-5
Grace	195	263	298
DeepFLco	166	248	279
FLUCCS	160	249	275
Ochiai	80	165	196
CNNFL	55	117	143

AutoStructor: An LLM-assisted fault localization and automated program repair framework

H. Özaytürk, F. Buzluca, "AutoStructor: A Generative AI-Based Framework for Automated Program Repair with Deep Learning-Guided Fault Localization", UBMK 2025.
https://doi.org/10.1109/UBMK67458.2025.11206755

- İncelenen metotlarla ilgili kalite metrikleri de toplanarak grafa katıldı.
- Yapay sinir ağına dikkat (*attention*) mekanizması eklendi.
- Hatalı sonuç veren testlerin açıklamaları LLM ile elde edildi; bu bilgiler hata düzeltme aşamasında kullanıldı.
- Başarım ölçümü: Defects4J Lang Project
  Toplam 61 hatalı metot var.

Method	Top-1	Top-3	Top-5	Top-10
Grace	45	56	57	61
+ Multi-Modal Attention	45	56	59	61
+ Quality Metrics	46	55	59	61
AutoStructor	50	57	60	61