

Testiranje zasnovano na modelu stanja

Problemi sa tradicionalnim unit testiranjem

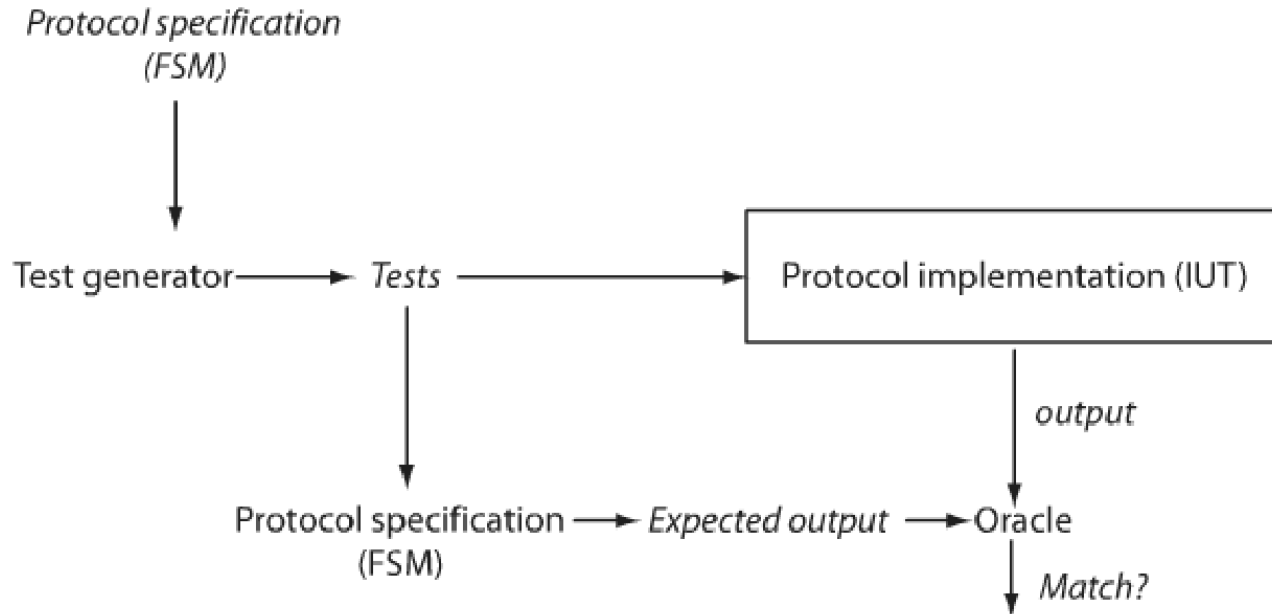
- Ručno pisanje kombinatorike: Šta ako sistem ima 5 stanja i 10 mogućih akcija?
- Eksplozija broja test-slučajeva (Test Case Explosion).
- Ljudski faktor: Testiramo samo "srećne putanje" (Happy Paths) koje su nam očigledne. Programeri razmišljaju sekvencijalno o akcijama, a korisnici klikću nasumično.
- Teško održavanje: Izmena u biznis logici zahteva prepisivanje desetina pojedinačnih @Test metoda.

Testiranje zasnovano na modelima (MBT)

- Šta je MBT? Pristup u kome se ponašanje sistema prvo opisuje apstraktnim modelom (najčešće grafom).
- Koja je uloga alata? Alat (npr. GraphWalker) čita model i automatski generiše putanje (test-slučajeve) kroz sistem.
- Prednost: Umesto 50 linearnih testova, održavamo jedan jedinstven model.
- *Umesto da učimo računar KAKO da testira korak po korak, mi mu dajemo MAPU sistema i kažemo mu: 'Nađi sam put i proverí da li sve radi'.*

Testiranje usaglašenosti (conformance testing)

- Ponašanje (na primer, komunikacioni protokol) se modeluje konačnim automatom (Finite State Machine, FSM)
- Model se implementira nekim softverom (implementation under test, IUT) i želimo testirati da li implementacija odgovara modelu



Model automata sa konačnim brojem stanja (Finite State Machine model)

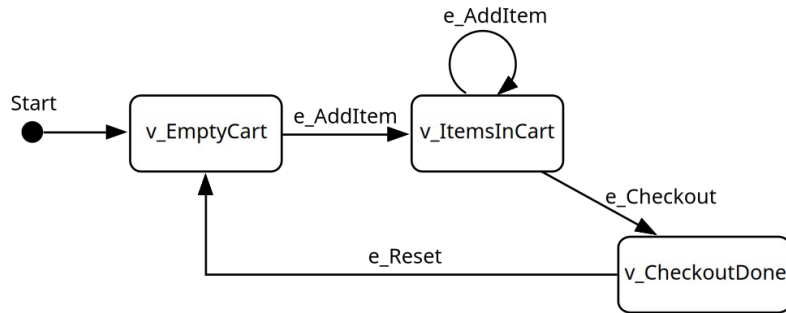
FSM je matematički model izračunavanja definisan kao petorka $(Q, \Sigma, \delta, q_0, F)$

:

- Q – Konačan skup **stanja** (*States / Vertices*)
- Σ – Konačan skup **ulaznih simbola / događaja** (*Inputs / Edges*)
- δ – Funkcija **prelaza** ($\delta: Q \times \Sigma \rightarrow Q$)
- q_0 – **Početno** stanje ($q_0 \in Q$)
- F – Skup **završnih/prihvatljivih** stanja ($F \subseteq Q$)

Ponašanje svakog softvera koji ima promenljivo stanje (ulogovan/izlogovan, prazna/puna korpa,...) može se modelovati konačnim automatom.

Predstavljanje konačnog automata



Graf stanja

Trenutno stanje:	Akcija: e_AddItem	Akcija: e_CheckOut	Akcija: e_Reset
v_EmptyCart	v_ItemsInCart		
v_ItemsInCart	v_ItemsInCart	v_CheckOutDone	
v_CheckOutDone			v_EmptyCart

Tabela prelaza

Prazni ulazi u tabeli prelaza otkrivaju nedefinisane prelaze

Predstavljanje automata: graf vs tabela

- Svaki FSM se može ravnopravno prikazati na dva načina:
- **1. Graf prelaza (Usmjereni graf):**
 - Čvorovi (krugovi) = Stanja.
 - Usmerene grane (strelice) = Prelazi, označeni simbolom iz alfabeta Σ .
- **2. Tabela prelaza (Matrica):**
 - Redovi predstavljaju trenutna stanja (Q).
 - Kolone predstavljaju ulazne događaje (Σ).
 - Čelije predstavljaju naredno stanje $\delta(q, \sigma)$.

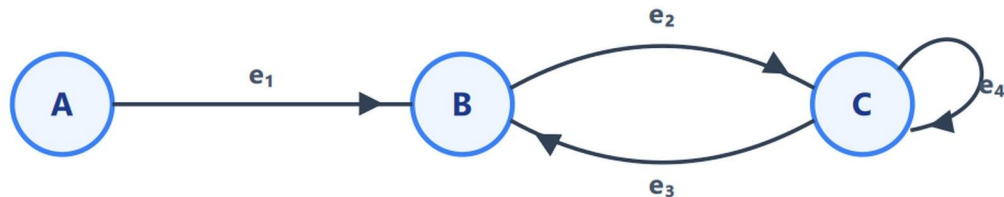
Kriterijumi testiranja modela stanja

- Neka je $G = (V, E)$ usmereni graf koji predstavlja naš model, gde je V skup čvorova (stanja), a E skup grana (prelaza). Neka je P skup putanja izvršenih tokom testiranja.
- **Pokrivenost stanja (Vertex Coverage - VC):**
 - Zahteva da svaki čvor $v \in V$ bude deo bar jedne test putanje $p \in P$.
 - $$VC = \frac{|V_{posjećen}|}{|V|} \times 100\%$$
- **Pokrivenost prelaza (Edge Coverage - EC):**
 - Zahteva da svaka grana $e \in E$ bude deo bar jedne test putanje $p \in P$.
 - $$EC = \frac{|E_{prođen}|}{|E|} \times 100\%$$

Kriterijumi testiranja modela stanja (nastavak)

- **Pokrivenost parova prelaza (Edge-Pair Coverage - EPC):**
 - Zahteva da svaka podputanja dužine 2 (dve uzastopne grane $[e_i, e_j]$) bude izvršena.
 - Služi za testiranje "memorije" ili konteksta (da li ponašanje u stanju zavisi od toga odakle smo u njega došli).
- **Pokrivenost svih putanja (Complete Path Coverage - CPC):**
 - Zahteva da se izvrše sve moguće putanje kroz graf.
 - Ako graf sadrži ciklus (npr. naš prelaz **dodaj** na stanju **Puna**), skup mogućih putanja je **beskonačan** ($|P| = \infty$). Zbog toga je CPC praktično neupotrebljiv kod kompleksnijeg softvera.

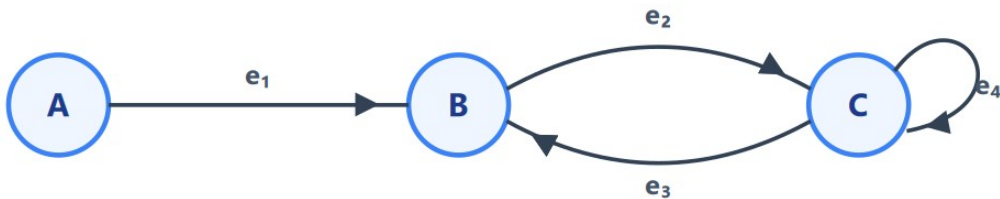
Primer za kriterijume pokrivenosti



- Broj ukupnih stanja $|V| = 3$, broj ukupnih grana $|E|=4$
- Neka je test inženjer ručno kliktao kroz aplikaciju i prešao sledeću sekvencu (test putanju): $A \xrightarrow{e_1} B \xrightarrow{e_2} C$
- Izračunajmo procene pokrivenosti za ovu putanju:
 - **Vertex Coverage:** posećenja su sva tri stanja (A,B,C). $VC = 3/3 = 100\%$
 - **Edge Coverage:** posećene su dve (e_1, e_2) od 4 grane. $EC = 2/4 = 50\%$
- Putanja $A \xrightarrow{e_1} B \xrightarrow{e_2} C \xrightarrow{e_4} C \xrightarrow{e_3} B$ bi pokrila sve grane i postigla $EC = 100\%$.

Edge-pair coverage kriterijum

- Kriterijum pokrivenosti parova grana zahteva da se prođe kroz svaku sekvencu u grafu od dve uzastopne grane (putanje dužine 2)
- Kriterijum pokrivenosti parova grana može se svesti na kriterijum pokrivenosti pojedinačnih grana transformacijom grafa (model unrolling)
- Radi ilustracije, posmatrajmo sledeći graf



Svi parovi grana

Na ovom grafu imamo ukupno 5 jedinstvenih sekvenci od dve povezane grane koje naša test sekvenca mora da izvrši:

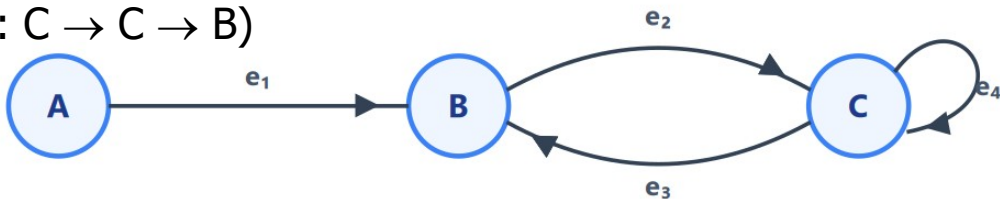
$e_1 \rightarrow e_2$ (Putanja: $A \rightarrow B \rightarrow C$)

$e_2 \rightarrow e_3$ (Putanja: $B \rightarrow C \rightarrow B$)

$e_2 \rightarrow e_4$ (Putanja: $B \rightarrow C \rightarrow C$)

$e_3 \rightarrow e_2$ (Putanja: $C \rightarrow B \rightarrow C$)

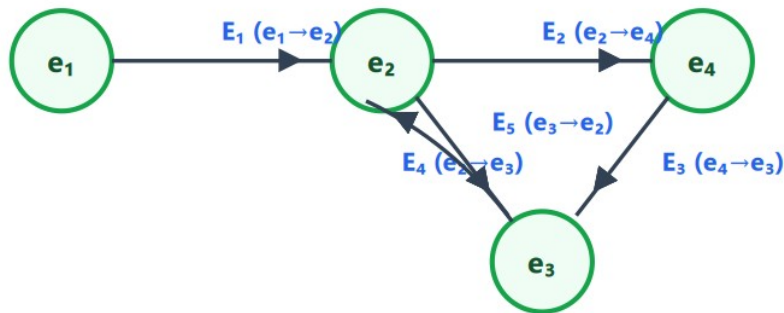
$e_4 \rightarrow e_3$ (Putanja: $C \rightarrow C \rightarrow B$)



Algoritam transformacije

Radimo matematičko "odmotavanje" (unrolling) po sledećim pravilima:

- Konverzija čvorova: svaka grana iz originalnog grafa (e_1, e_2, e_3, e_4) postaje samostalan čvor u novom grafu.
- Konverzija tranzicija: Svaki važeći par uzastopnih grana iz originalnog grafa postaje nova usmerena grana koja povezuje te nove čvorove.



Šta dobijamo ovom transformacijom

- Kada GraphWalker-u zadamo cilj 100% **Edge Coverage na ovom novom (transformisanom) modelu**, on će generisati optimalnu putanju koja mora da pređe preko grana E_1, E_2, E_3, E_4, E_5 . Time smo naterali standardni algoritam da uradi potpunu **Edge-Pair** pokrivenost **originalnog** modela.

Primer testiranja alatom GraphWalker

Posmatramo REST API aplikaciju online korpe za kupovinu (Shopping Cart) sa samo tri endpoint-a koji simuliraju ponašanje korpe:

- **GET /cart** — Vraća trenutni broj artikala.
- **POST /cart/add** — Dodaje artikal (maksimalno npr. 3 artikla).
- **POST /cart/checkout** — Prazni korpu i završava kupovinu (moguće samo ako korpa nije prazna).

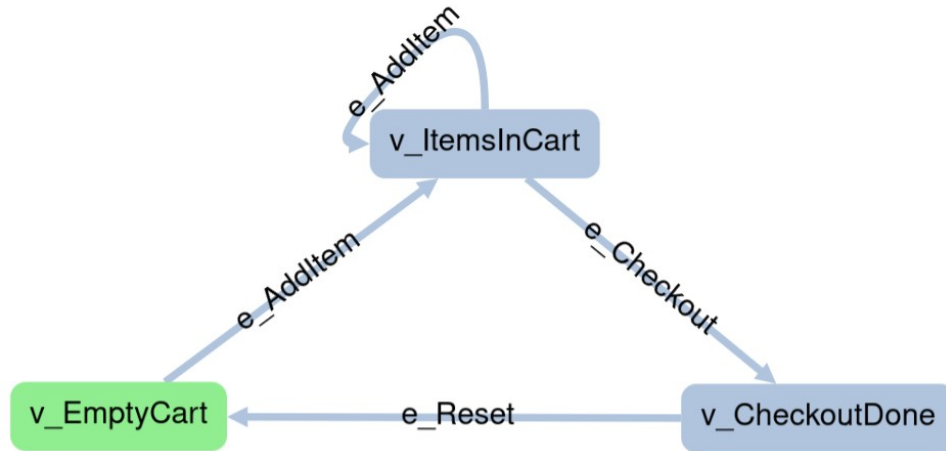
Drugim rečima, ne može se uradi check out ako je korpa prazna, niti se može dodati više od 3 artikla u korpu.

- [Primer implementacije u javalin frejmvorku možete naći u prilogu u folderu: *main/java/com/example/App.java*]. Pokreće se sa:
`mvn exec:java -Dexec.mainClass="com.example.App"`

Proces testiranja alatom GraphWalker

- GraphWalker je open-source MBT alat za Javu.
- Proces rada:
 1. Nacrtamo graf vizuelno (GraphWalker Studio).
 2. Sačuvamo ga u projektu (.json).
 3. Maven plugin generiše Java interfejs na osnovu naziva čvorova i grana.
 4. Implementiramo taj interfejs pišući standardne asertacije i akcije.

Modelovanje aplikacije u GraphWalker studiju



- Konvencija imenovanja:
 - Čvorovi počinju sa **v_** (Vertex)
 - Grane počinju sa **e_** (Edge)
- **Naš model:**
 - **v_EmptyCart** (Početno stanje)
 - **e_AddItem** → vodi u **v_ItemsInCart**
 - **e_Checkout** → vodi u **v_CheckoutDone**
 - **e_Reset** → vraća u **v_EmptyCart**

Generisanje koda iz modela

Nakon pokretanja:

```
mvn graphwalker:generate-test-sources
```

GraphWalker kreira fajl u target/generated-test-sources/.../ShoppingChartModel.java

```
package com.example;  
  
import org.graphwalker.java.annotation.Model;  
import org.graphwalker.java.annotation.Vertex;  
import org.graphwalker.java.annotation.Edge;  
  
@Model(file =  
"com/example/ShoppingChartModel.json")  
public interface ShoppingCartModel {  
  
    @Edge()  
    void e_Reset();  
  
    @Edge()  
    void e_Checkout();  
  
    @Vertex()  
    void v_EmptyCart();  
  
    @Vertex()  
    void v_ItemsInCart();  
  
    @Edge()  
    void e_AddItem();  
  
    @Vertex()  
    void v_CheckoutDone();  
}
```

Mapiranje modela konačnog automata na testiranje

- **Stanje (State / Vertex):** Stabilno stanje aplikacije u kom se čeka akcija korisnika ili sistema (npr. *Korisnik je na stranici za plaćanje*).
- **Prelaz (Transition / Edge):** Stimulus ili akcija koja pomera sistem iz jednog stanja u drugo (npr. *Klik na dugme 'Kupi'*).
- **U MBT kontekstu:**
 - Čvorovi (Vertices) = Mesto gde radimo **Asertacije** (provere stanja).
 - Grane (Edges) = Mesto gde izvršavamo **Akcije** (pozivi API-ja, klikovi).

Implementacija generisanog interfejsa

```
@GraphWalker(value = "random(edge_coverage(100))")
public class ShoppingCartTest extends ExecutionContext
    implements ShoppingCartModel {

    static { baseUrl = "http://localhost:7070/cart"; }

    @org.graphwalker.java.annotation.BeforeExecution
    public void resetApplicationStateBeforeTest() {
        // RestAssured automatski baca izuzetak ako server nije
        // podignut, nema potrebe za try-catch
        given().body("").post("/reset").then().statusCode(200);
    }

    // --- AKCIJE (EDGES) ---
    @Override
    public void e_AddItem() {
        given().body("").post("/add").then().statusCode(200);
    }

    @Override
    public void e_Checkout() {
        given().body("").post("/checkout").then().statusCode(200);
    }

    @Override
    public void e_Reset() {
        given().body("").post("/reset").then().statusCode(200);
    }
}
```

```
// --- STANJA (VERTICES) ---

@Override
public void v_EmptyCart() {
    // Uzmi GET sa korena, proveriti status 200 i da li je
    // u telu odgovora string "0"
    get().then().statusCode(200).body(equalTo("0"));
}

@Override
public void v_ItemsInCart() {
    // Preuzimamo odgovor kao String, čistimo razmake i
    // pretvaramo u int radi provere opsega
    String body =
        get().then().statusCode(200).extract().body().asString().trim();
    int count = Integer.parseInt(body);
    org.junit.jupiter.api.Assertions
        .assertTrue(count > 0 && count <= 3);
}

@Override
public void v_CheckoutDone() {
    get().then().statusCode(200).body(equalTo("0"));
}

@Test
public void runGraphWalkerTest() {
    new org.graphwalker.java.test.TestBuilder()
        .addContext(this)
        .execute();
}
```

Alat nas primorava da napišemo test kod za svaki element koji smo nacrtali.
Ako dodamo novu strelicu na grafu, kod se neće kompajlirati dok je ne pokrijemo u Javi.

Prevodjenje test koda i pokretanje testova

- Naš kod se prevodi sa:
`mvn test-compile`
- Testovi se pokreću sa:
`mvn graphwalker:test`

Izlaz koji daje GraphWalker primer

[INFO]

Resetujem stanje Javalin aplikacije pre početka testa...
[STANJE] --> v_EmptyCart
[main] INFO
org.graphwalker.core.generator.SingletonRandomGenerator - Seed: 150722542287028

[AKCIJA] --> e_AddItem
[STANJE] --> v_ItemsInCart
[AKCIJA] --> e_Checkout
[STANJE] --> v_CheckoutDone
[AKCIJA] --> e_Reset
[STANJE] --> v_EmptyCart
[AKCIJA] --> e_AddItem
[STANJE] --> v_ItemsInCart
[AKCIJA] --> e_AddItem
[STANJE] --> v_ItemsInCart

[INFO]

[INFO] Result :

[INFO]

[INFO] {
"totalFailedNumberOfModels": 0,
"totalNotExecutedNumberOfModels": 0,
"totalNumberOfUnvisitedVertices": 0,
"verticesNotVisited": [],
"totalNumberOfModels": 1,
"totalCompletedNumberOfModels": 1,
"totalNumberOfVisitedEdges": 4,
"totalIncompleteNumberOfModels": 0,
"edgesNotVisited": [],
"vertexCoverage": 100,
"totalNumberOfEdges": 4,
"totalNumberOfVisitedVertices": 3,
"edgeCoverage": 100,
"totalNumberOfVertices": 3,
"totalNumberOfUnvisitedEdges": 0
}

Strategije pretrage grafa (path generators)

- Kako GraphWalker odlučuje kuda ide kroz graf? Spajanjem generatora i uslova zaustavljanja:

random(edge_coverage(100))

- Bira sledeći prelaz nasumično (kao "pijani šetač") dok ne prođe kroz 100% strelica. *Daje sekvence koje čovek ručno nikad ne bi birao.*

a_star(reached_vertex(v_CheckoutDone))

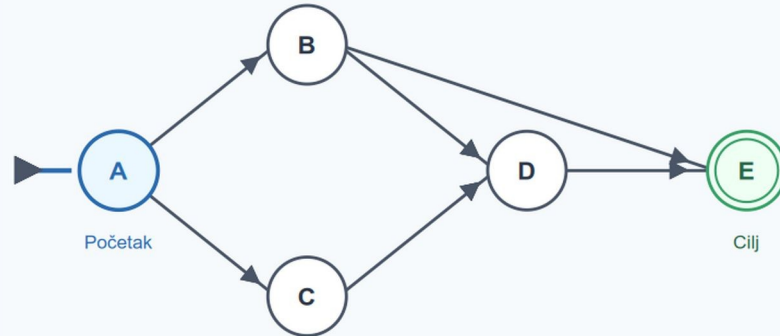
- Primenjuje Djikstra/A* algoritam da najkraćim putem određeno stanje.

A* algoritam graphwalker-a

- a_star u alatu GraphWalker je generator putanje koji služi za pronalaženje **najkraćeg puta** do specifičnog cilja u modelu.
- (A-Star) algoritam radi tako što kombinuje stvarnu cenu puta od početka do trenutnog čvora i procenjenju cenu (heuristiku) od trenutnog čvora do cilja. Njegova ključna formula je:
- $f(n) = g(n) + h(n)$
- $g(n)$: Stvarna cena (broj pređenih grana) od početnog čvora do čvora n .
- $h(n)$: Heuristika – procenjena cena od čvora n do cilja. Pošto GraphWalker "vidi" ceo graf pre izvršavanja, on kao heuristiku koristi stvarnu dužinu najkraćeg puta kroz graf ignorišući prepreke (engl. shortest topological distance), što znači da je heuristika ovde savršeno tačna.
- $f(n)$: Ukupna procenjena cena puta koji prolazi kroz čvor n . Algoritam uvek sledeći otvara čvor koji ima najmanju vrednost $f(n)$.

Ilustracija A* algoritma

- Zamislamo jednostavan, usmereni graf aplikacije. Svaka grana ima cenu 1 (svaki klik/tranzicija je jedan korak). Cilj nam je da stignemo do čvora E .



Ilustracija A*: Proračun Heuristike $h(n)$ do cilja E

Pre nego što krene, algoritam izračunava koliko je svaki čvor udaljen od cilja E vazdušnom (topološkom) linijom:

- $h(E) = 0$ (Već smo na cilju)
- $h(B) = 1$ (Iz B se stiže u E u 1 koraku: $B \rightarrow E$)
- $h(D) = 1$ (Iz D se stiže u E u 1 koraku: $D \rightarrow E$)
- $h(C) = 2$ (Iz C se stiže preko D: $C \rightarrow D \rightarrow E = 2$ koraka)
- $h(A) = 2$ (Najkraće je preko B: $A \rightarrow B \rightarrow E = 2$ koraka)

Ilustracija A*: korak po korak

Održavamo dve liste:

- **Open List** (Čvorovi koji su otkriveni, ali još nisu analizirani).
- **Closed List** (Čvorovi koji su potpuno analizirani).

Početno stanje:

- Krećemo iz čvora A.
- Za čvor A: $g(A) = 0$ (nismo napravili ni korak), $h(A) = 2$. Ukupno $f(A) = 0 + 2 = 2$.
- **Open List:** [A (f=2)]
- **Closed List:** []

KORAK 1: Analiza čvora A

Uzimamo čvor sa najmanjim f iz Open liste (to je **A**). Prebacujemo ga u Closed listu i gledamo njegove komšije (**B** i **C**).

- **Za komšiju B:**

- $g(B) = g(A) + 1 = 0 + 1 = 1$
- $h(B) = 1$
- $f(B) = 1 + 1 = 2$

- **Za komšiju C:**

- $g(C) = g(A) + 1 = 0 + 1 = 1$
- $h(C) = 2$
- $f(C) = 1 + 2 = 3$

- **Open List:** [B ($f=2$), C ($f=3$)]

- **Closed List:** [A]

KORAK 2: Analiza čvora B

Iz Open liste biramo čvor sa najmanjim f . To je **B** (jer je $2 < 3$). Širimo se iz čvora B na njegove komšije (**D** i **E**).

- **Za komšiju D:**

- $g(D) = g(B) + 1 = 1 + 1 = 2$
- $h(D) = 1$
- $f(D) = 2 + 1 = 3$

- **Za komšiju E (Naš Cilj!):**

- $g(E) = g(B) + 1 = 1 + 1 = 2$
- $h(E) = 0$
- $f(E) = 2 + 0 = 2$

Sada ažuriramo liste. Čvor B ide u Closed.

- **Open List:** [E ($f=2$), C ($f=3$), D ($f=3$)]

- **Closed List:** [A, B]

Korak 3: Završetak

Iz Open liste biramo čvor sa najmanjim f . To je čvor **E** ($f = 2$).

Pošto je čvor koji smo upravo izvukli iz Open liste zapravo naš **ciljni čvor**, algoritam se ovde **zaustavlja**.

Rekonstrukcijom unazad (kako smo stigli do E), dobijamo idealnu i najkraću test-putanju:

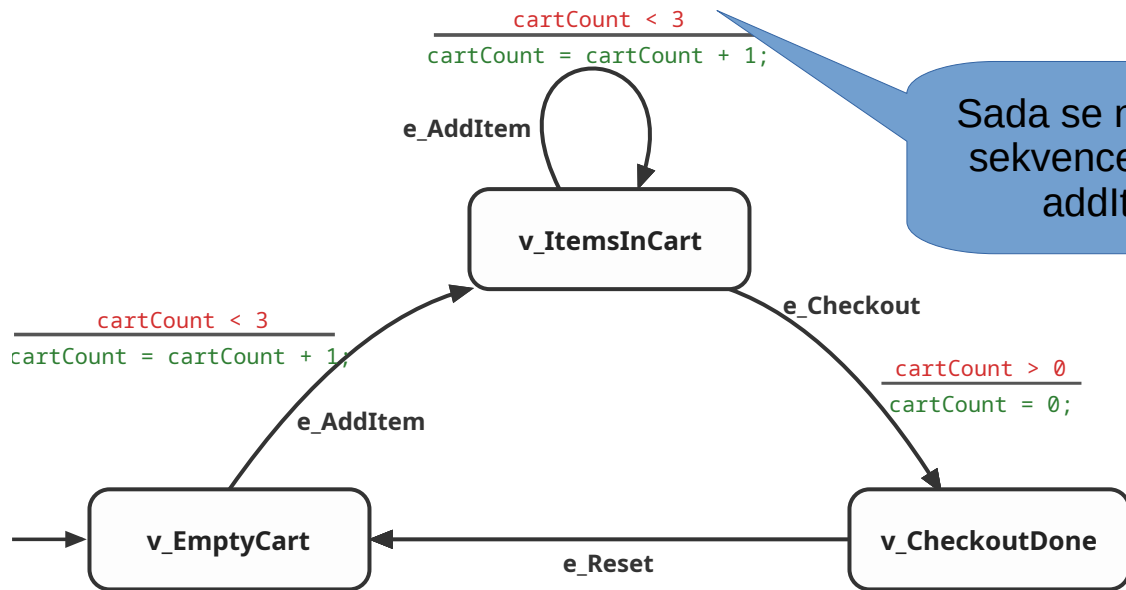
$$\mathbf{A} \rightarrow \mathbf{B} \rightarrow \mathbf{E}$$

Zašto je ovo bolje od nasumičnog lutanja?

Primetite da čvor **C** i čvor **D** (preko B) uopšte nisu morali biti detaljno otvarani i analizirani. Heuristika je tačno "povukla" algoritam na gornju granu i GraphWalker je pronašao put u minimalnom broju koraka bez trošenja resursa.

Prošireni model stanja (Extended Finite State Model)

- Graphwalker podržava prošireni model, gde pored grafa možemo imati globalne promenljive. Tada se grane grafa mogu proširiti guardovima i akcijama (prelaz se vrši samo ako je guard ispunjen i tada preduzima akcija). I stanja po potrebi mogu imati akcije (ažuriranje vrednosti promenljivih).



Testiranje robusnosti (negativni testovi) za primer Shopping Charta

- Kao što smo rekli, tabela prelaza (koja je u stvari Dekartov proizvod *Stanja x Prelazi*) otkriva nam tranzicije koje nisu definisane grafom:

Stanje (*S*) \ Događaj

(<i>E</i>)	e_AddItem	e_Checkout	e_Reset
v_EmptyCart	Validno (v_ItemsInCart)	✗ (N1)	✗ (N2)
v_ItemsInCart	Validno (v_CheckoutDone)	Validno (v_CheckoutDone)	✗ (N3)
v_CheckoutDone	✗ (N4)	✗ (N5)	Validno (v_EmptyCart)

Analiza i rešavanje nelegalnih tranzicija

- N1 (**v_EmptyCart, e_Checkout**): Korisnik želi da plati, a korpa je prazna. Sistem treba da ignorise akciju (dugme je sivo) ili da baci poruku "Korpa je prazna". Rešenje: Ostanak u v_EmptyCart.
- N2 (**v_EmptyCart, e_Reset**): Reset prazne korpe nema efekta. Rešenje: Ostanak u v_EmptyCart.
- N3 (**v_ItemsInCart, e_Reset**): Korisnik želi da isprazni korpu pre kupovine. Rešenje: Povratak u v_EmptyCart.
- N4 (**v_CheckoutDone, e_AddItem**): Kupovina je gotova, korisnik pokušava da doda artikl pre nego što resetuje sistem. Sistem to ne dozvoljava. Rešenje: Ostanak u v_CheckoutDone (ili odlazak u stanje greške).
- N5 (**v_CheckoutDone, e_Checkout**): Korisnik ponovo klikće na plaćanje iako je već platio. Rešenje: Ostanak u v_CheckoutDone.

Ako bismo crtali svaku od ovih strelica pojedinačno, model bi postao nepregledan sa ukupno 9 strelica za ovako mali automat.

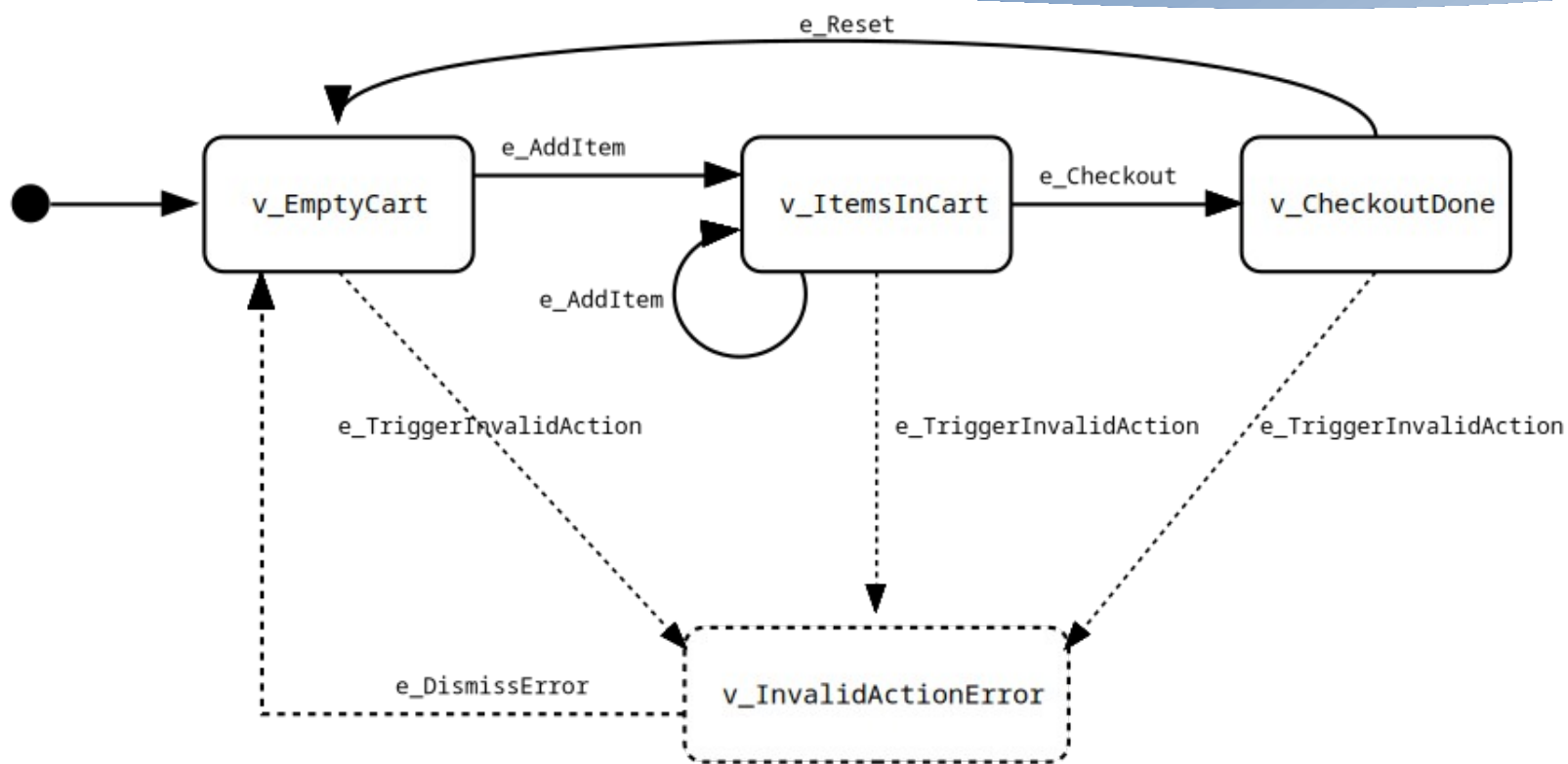
Rešenje sa “catch all” tranzicijom

Da bismo izbegli špageti-graf, uvodimo univerzalnu tranziciju za greške, ali pošto radimo u modelu bez guard-ova, moramo modifikovati same tranzicije u GraphWalkeru tako da Java kod preuzme odluku šta je u kom trenutku nelegalno.

U GraphWalker Studiju model bismo nacrtali ovako:

1. Zadržavamo četiri originalna ("srećna") prelaza.
2. Kreiramo jedno novo stanje greške: **v_InvalidActionError**.
3. Iz **svakog** od tri osnovna stanja povlačimo po jednu tranziciju koja se zove **e_TriggerInvalidAction** i sve one vode u **v_InvalidActionError**.
4. Iz stanja **v_InvalidActionError** povlačimo tranziciju **e_DismissError** koja vraća sistem nazad u tekuće stanje (ili ga resetuje u početno stanje).

Izgled modifikovanog grafa



```
// Pomoćna promenljiva za praćenje trenutnog stanja u modelu  
private String currentModelVertex = "v_EmptyCart";
```

```
@Override  
public void v_EmptyCart() { currentModelVertex = "v_EmptyCart"; }  
@Override  
public void v_ItemsInCart() { currentModelVertex = "v_ItemsInCart"; }  
@Override  
public void v_CheckoutDone() { currentModelVertex = "v_CheckoutDone"; }
```

```
// Pamtim samo HTTP status kod nelegalne akcije  
private int lastResponseStatusCode;
```

```
@Override  
public void e_TriggerInvalidAction() {  
    logStep();  
  
    if (currentModelVertex.equals("v_EmptyCart")) {  
        // N1/N2: Checkout prazne korpe -> Očekujemo 400  
        lastResponseStatusCode = given().body("").post("/checkout").then().extract().statusCode();  
    } else if (currentModelVertex.equals("v_ItemsInCart")) {  
        // N4: POST na nepostojeću rutu -> Javalin vraća 404  
        lastResponseStatusCode = given().body("").post("/nema").then().extract().statusCode();  
    } else if (currentModelVertex.equals("v_CheckoutDone")) {  
        // N6: Ponovni checkout -> Očekujemo 400  
        lastResponseStatusCode = given().body("").post("/checkout").then().extract().statusCode();  
    }  
}
```

```
@Override  
public void v_InvalidActionError() {  
    logStep();  
    // Konceptualna provera: Da li je sistem uspešno odbio nelegalnu akciju?  
    // Ako smo gađali lošu rutu očekujemo 404, za sve ostale biznis greške očekujemo 400.  
    int expectedStatus = currentModelVertex.equals("v_ItemsInCart") ? 404 : 400;  
  
    org.junit.jupiter.api.Assertions.assertEquals(expectedStatus, lastResponseStatusCode);  
}
```

Implementacija test koda

Prednost pristupa sa catch all nelegalnom tranzicijom

- Broj tranzicija na grafu raste linearno (N stanja = N dodatnih strelica za greške), a ne kvadratno kao kod čistog Dekartovog proizvoda. Graf ostaje čist i čitljiv, a sva kompleksnost "loših podataka" i "loših klikova" je prebačena u Java kod unutar **e_TriggerInvalidAction** metode.

Kada konačni automati nisu dobar izbor za testiranje?

- Kada aplikacija nema izražena stanja (npr. jednostavan servis za računanje matematičkih formula).
- Kada su podaci ključni, a ne koraci (Data-Driven testiranje je tu bolje).
- Kada se interfejs i biznis pravila menjaju na svakih sat vremena (konstantno menjanje grafa stvara preveliki režijski trošak).