Specifikacija i Black-Box testiranje pomoću jezika Spec# i alata SpecExplorer

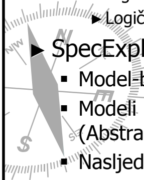
Matko Botinčan
PMF – Matematički odjel

E-mail: mabotinc@math.hr

Sadržaj seminara i slideovi bazirani su na predavanjima Margusa Veanesa (@MSR) koje je održao na ESSCaSS`04

1

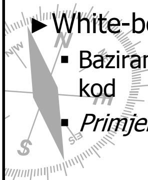
Spec# i SpecExplorer



- Spec#
  - Uključuje kompletni C#
  - Proširuje C# s dodatnim elementima
    - Ugovori (pre/post-uvjeti, invarijante)
    - High-level strukture podataka (tuple, set, sequence, map)
    - Logički kvantifikatori: Forall i Exists
- SpecExplorer
  - Model-based alat za analizu stanja i testiranje
  - Modeli mogu biti napisani u Spec#-u ili AsmL-u (Abstract State Machine Language)
  - Nasljednik AsmL Testera

2

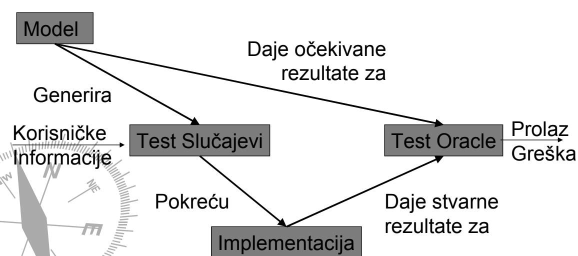
Osnovni tipovi testiranja



- Black-box testiranje – testiranje ponašanja
  - Vezano je uz specifikaciju sustava, bez znanja o implementacijskim detaljima
  - *Primjer:* scenario test
- White-box testiranje – strukturalno testiranje
  - Bazirano na lokalnom uvidu u implementacijski kod
  - *Primjer*: unit test

3

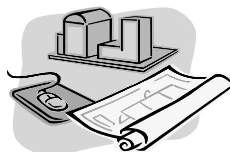
Što je to Model-based testiranje?



5

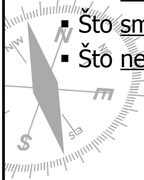
Što je to model? (1)

Model:
- Predstavlja *apstrakciju* sustava iz određene perspektive
- Omogućava *proučavanje, konstrukciju* i *predikciju*
- *Nije* nužno *sveobuhvatan*
- Može biti izražen kao tabela, grafički dijagram, itd. ili u
  → **Spec#**



6

Što je to model? (2)



- Model je program – može se pokrenuti
- Opisuje željeno ponašanje implementacije koje je moguće testirati:
  - Što mora biti učinjeno
  - Što smije biti učinjeno
  - Što ne smije biti učinjeno

7

Modeliranje u Spec#-u

- Da bismo napisali model potrebno je:
  1. Identificirati *pravi nivo apstrakcije*
  2. Definirati varijable stanja
  3. Definirati akcije
- *Stanje* je definirano *preslikavanjem sa skupa varijabli u skup vrijednosti* (struktura prvog reda)
- *Akcije* su definirane metodama (Akcije s proslijeđenim aktualnim parametrima nazivaju se *invokacije*).

Prvi model: *Stopwatch*

- Stanja modela

```
namespace StopWatch;

enum DisplayMode {
  TIME,
  DATE,
  STOPPER
}

DisplayMode SWDisplayMode =
    DisplayMode.TIME;

bool SWStopperRunning = false;

bool SWStopperFrozen = false;

bool SWStopperReset = true;
```

Stopwatch akcije (1)

- Pritisak na gumb `*mode*'

```
[Action]
void Mode() {
  switch (SWDisplayMode) {
  case TIME:
    SWDisplayMode = DisplayMode.DATE;
    break;
  case DATE:
    SWDisplayMode = DisplayMode.STOPPER;
    break;
  case STOPPER:
    SWDisplayMode = DisplayMode.TIME;
    break;
  }
}
```

Stopwatch akcije (2)

- Pritisak na gumb `*start/stop*'

Pre-condition:
Gumb `*Stop*' ima efekta samo u STOPPER modu

```
[Action]
void StartStop()
  requires SWDisplayMode ==
           DisplayMode.STOPPER;
{
  if (!SWStopperRunning && SWStopperReset)
    SWStopperReset = false;

  SWStopperRunning = !SWStopperRunning;
}
```

Stopwatch akcije (3)

- Pritisak na gumb `*reset/lap*'

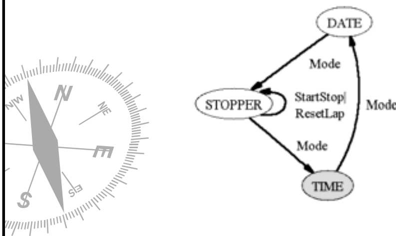
```
[Action]
void ResetLap()
  requires SWDisplayMode ==
           DisplayMode.STOPPER;
{
  if (SWStopperRunning)
    SWStopperFrozen =
      !SWStopperFrozen;
  if (!SWStopperRunning) {
    if (SWStopperFrozen)
      SWStopperFrozen = false;
    else
      SWStopperReset = true;
  }
}
```

Upotreba SpecExplorera (1)

- Otvorite Stopwatch projekt u SpecExploreru i pokrenite ga. Na slici je generirani FSM: (početno stanje s_0 obojano sivo)

Upotreba SpecExplorera (2)

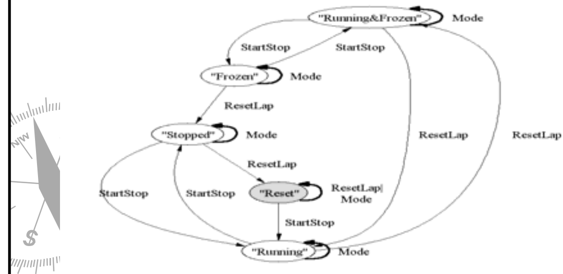
- DisplayMode view
  - Stanja su grupirana prema SWDisplayMode varijabli



14

Upotreba SpecExplorera (3)

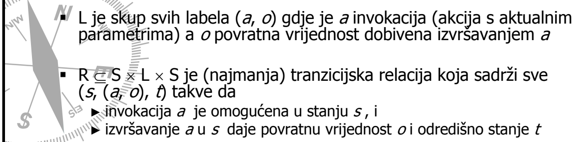
- StopperStatus view
  - Stanja su grupirana prema varijablama: SWStopperRunning, SWStopperFrozen and SWStopperReset



15

Labelirani tranzicijski sistem

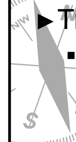
- Program P koji opisuje model inducira (možda beskonačan) labelirani tranzicijski sistem (LTS) (s_0, S, L, R)
  - s_0 je početno stanje zadano početnim vrijednostima svih var. u P
  - S je skup svih dostižljivih stanja
    - dostižljivih iz početnog stanja modela kroz invokacije
  - L je skup svih labela (*a*, *o*) gdje je *a* invokacija (akcija s aktualnim parametrima) a *o* povratna vrijednost dobivena izvršavanjem *a*
  - $R \subseteq S \times L \times S$ je (najmanja) tranzicijska relacija koja sadrži sve (*s*, (*a*, *o*), *t*) takve da
    - invokacija *a* je omogućena u stanju *s* , i
    - izvršavanje *a* u *s* daje povratnu vrijednost *o* i odredišno stanje *t*



16

LTS pogled na *Stopwatch*

- Inicijalno stanje s_0:
  - SWDisplayMode = TIME,
    SWStopperRunning = false;
    SWStopperFrozen = false;
    SWStopperReset = true;
- Tranzicijska relacija R
  - {(s_0, (Mode(), void), s_1), …}
    gdje je u s_1 SWDisplayMode = DATE, a ostale varijable su kao u s_0



17

Generiranje test slučajeva

- Test slučajeve primjenjujemo na konačni LTS (s_0, S, L, R)
  - [baratanju s beskonačnim sistemima vratit ćemo se kasnije]
- Treba odabrati *namjenu* test slučajeva:
  - *Transition coverage* (obilazak svih tranzicija u R)
  - *Shortest path* (dosezanje stanja s zadanim uvjetom minimalnim brojem koraka)
  - *Random walk*
- Pokretanje "Generate Test Suites" u SpecExploreru generira željene test slučajeve
- Generirani test slučajevi mogu postojati kao programi (u C#-u ili VB-u)

18

Obilazak svih tranzicija

- Počevši iz stanja s_0 treba obići sve tranzicije u R
  - Pretpostavimo da je R *deterministička*:
    za sve (s, (a, o), t) i (s', (a', o'), t') $\in$ R,
    (s=s' & a=a') $\Rightarrow$ (o=o' & t=t')
    [kasnije ćemo se vratiti nedeterminističkim sistemima]
  - Pretpostavimo također da je pripadajući graf *jako povezan* (uz eventualno dodavanje "reset" tranzicija (s, (reset, void), s_0) za $s \in S$).
- Da bi se generirali test slučajevi SpecExplorer koristi algoritam *Kineskog poštara* za generiranje tura

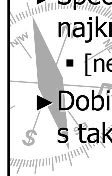
19

Primjer: `Transition coverage’

- Otvorite Stopwatch projekt u SpecExploreru
  - namjena Test Suite #0 je obilazak svih tranzicija (`Transition coverage’)
- Pokrenite ‘Generate Test Suites’
- Generirana je test sekvenca od 27 koraka, koja počinje i završava u s_0

20

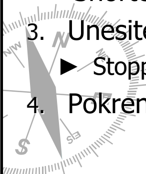
Najkraći put



- Zadan je tranzicijski sistem (s_0, S, L, R)
- Zadano je svojstvo φ na stanjima
  - φ je izraz s Boolean vrijednošću
- SpecExplorer koristi Dijkstrin algoritam najkraćeg puta (u determinističkom slučaju)
  - [nedeterministički slučaj je kompliciraniji]
- Dobiven je najkraći put iz stanja s_0 u stanje s takav da s zadovoljava φ

21

Primjer: `Shortest path’ (1)



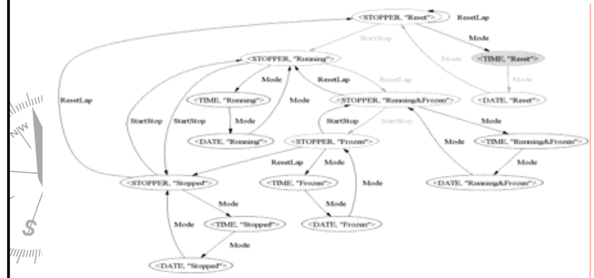
1. Otvorite Stopwatch projekt u SpecExploreru
2. Kliknite na Test Suite #0, promijenite tip u ‘ShortestPath’
3. Unesite izraz za ciljno stanje:
   - StopperStatus == “Frozen”
4. Pokrenite ‘Generate Test Suites’

22

Primjer: `Shortest path’ (2)

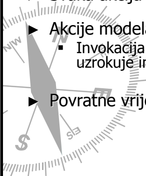
5. Odaberite ‘test segment 0’ u ‘segments’ pod ‘Test Suites’ tabom

- Osvjetljena je test sekvenca (Mode, Mode, StartStop, ResetLap, StartStop):



23

Pokretanje testova



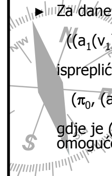
- *Test (segment)* je niz labela

  $((a_1(v_1),o_1), (a_2(v_2),o_2),\ldots, (a_k(v_k),o_k))$

  koje odgovaraju valjanom izvršavanju, tj. gdje je $(s_{i-1},(a_i(v_i),o_i),s_i) \in R$
  - [u nedeterminističkom slučaju – testovi su strategije]
- Svaka akcija a_i vezana je uz njoj korespodentnu implementaciju a'_i
- Akcije modela i implementacije izvršavaju se u paru:
  - Invokacija $a_i(v_i)$ u modelu, gdje je v_i niz aktualnih parametara za akciju a_i, uzrokuje invokaciju $a'_i(v_i)$ u implementaciji
- Povratne vrijednosti akcija moraju međusobno odgovarati

24

Dodatna provjera stanja pomoću sondi (probes)



- *Sonde (probes)* su izrazi vezani uz stanje (specijalni tip akcija)
- Evaluacija sonde p u stanju s producira tranziciju $(s,(p,p^s),s)$ na s gdje je p^s vrijednost od p u s.
- Kao i akcije, sonde su vezane uz metode u implementaciji, te se primjenjuju u svim stanjima kako bi se osigurala dodatna provjera
- Za dane sonde $P = \{p_1,\ldots,p_k\}$ test segment

  $((a_1(v_1),o_1),(a_2(v_2),o_2),\ldots)$

  isprepliće se sa sondama dajući tako prošireni test:

  $(\pi_0, (a_1(v_1),o_1), \pi_1, (a_2(v_2),o_2), \pi_2,\ldots)$

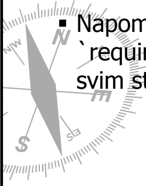
  gdje je $(s_{i-1},(a_i(v_i),o_i),s_i) \in R$, a π_i je niz svih labela sondi (p,p^{s_i}) t.d. $p \in P$ i p je omogućena u stanju s_i

25

Primjer sonde (1)

- Definirajte sondu u Stopwatch primjeru
  - Definirajte metodu:

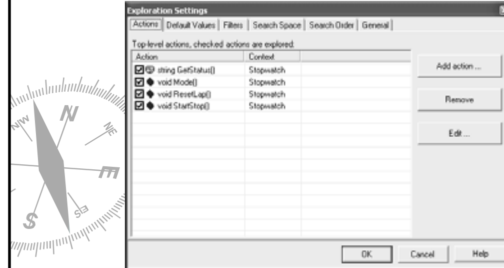
```
[Action(Kind=ActionAttributeKind.Probe)]
string GetStatus() {return StopperStatus;}
```



  - Napomena: uočite da nije naveden niti jedan \`requires' izraz, tako da je sonda omogućena u svim stanjima

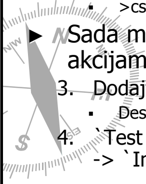
Primjer sonde (2)

- Otvaranjem \`Exploration Settings' u SpecExploreru da su sada definirane četiri akcije, od kojih je jedna sonda:



Zatvaranje kruga: povezivanje s implementacijom

- Stopwatch primjer ne sadrži implementaciju, no možemo generirati dummy implementaciju:
  1. Odaberite 'Generate conformance stubs'
     - odaberite C# (default), i ime datoteke 'StopwatchImpl.cs'
  2. Prevedite generiranu C# datoteku:
     - >csc /target:library StopwatchImpl.cs



- Sada možemo povezati akcije u modelu s akcijama u implementaciji:
  3. Dodajte referencu 'StopwatchImpl.dll' u projekt
     - Desni klik na References...
  4. \`Test Settings' -> \`Action Bindings' -> \`Autofill Scope' -> \`ImplStopwatch'

Zatvaranje kruga: pokretanje test slučajeva

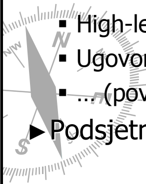
5. Pokrenite test slučajeve:

- Javlja se greška:
  invocation GetStatus(), result "Reset", from S0
  Diagnosis: FAILED: value mismatch
  - observed exception 'NotImplementedException'
  - expected "Reset"

- Uzrok je očigledan – generirani stubovi jedino što rade jest bacanje iznimke

Zašto ne koristiti C# ili VB ili …?

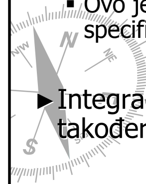
- Spec# i AsmL sadrže konstrukte koji služe kao potpora modeliranju:
  - Literate programiranje
  - Nedeterminizam
  - High-level strukture podataka
  - Ugovori
  - … (povrh svega) alat SpecExplorer



- Podsjetnik: Spec# je "nadskup" od C#

Literate programiranje

- Modeli su pisani s ciljem kako bi bili čitani i direktno razumjevani
  - Model mogu pregledavati i developeri i project manageri
  - Ovo je dobar način za otkrivanje grešaka u specifikaciji

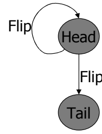


- Integracija s MS Wordom: Word dokument također je i izvorni kod programa

Nedeterminizam

- Nedeterminizam u nekom smislu označava mogućnost apstrakcije od detalja implementacije
- Primjer: poruka na mreži može biti izgubljena, mogu se javiti ili "timeout" ili "ack";
  Ponašanje implementacije može ovisiti o rasporedu dretvi, mrežnim karakteristikama, …

```
enum Coin {Head,Tail};
Coin coin = Coin.Head;
void Flip(){
  choose (x in enumof(Coin)){
    coin = x;
  }
}
```



High-level strukture podataka

- Apstraktni tipovi po vrijednosti (strukture)
  - jednakost po vrijednostima, ne po referenci (kao kod objekata)
  - Deklaracije mogu biti rekurzivne i generičke u tipu
- Set, map i sequence
  - Alternativa poljima, vezanim listama, hash tabelama
  - Nema potrebe za indeksima, pokazivačima, petljama
- Razumljiviji izrazi, kvantifikatori
  - Snaga, kompaktnost, ekspresivnost

Ugovori

- Pre-condition
  - `requires` φ
  - uvjeti za omogućavanje akcija
- Post-condition
  - `assures` φ
- Invarijante
  - `invariant` φ
  - zahtjev koji uvijek mora biti istinit

Strukture

- Spec# upotrebljava value semantiku za sve osim instance klasa i C# polja
- Fleksibilnije od C# "struct"-a
  - Nema zahtjeva za reprezentacijom fiksne duljine
  - Dopuštene su rekurzivne strukture (npr. stablo)
- Napomena za Spec# modele:
  koristiti klase samo kod dijeljenih identiteta;
  inače koristiti strukture

```
structure Tree{
  int root;
  Tree left;
  Tree right;
}
```


Spec# kolekcije

- Set
- Sequence
- Map
- Bag (multiskup)
- Napomena: sve Spec# kolekcije imaju value semantiku
  - Ne mogu biti dijeljene, nego samo "kopirane"
  - (Efikasna implementacija)
  - Jednakost je "strukturalna"

Set

- Konstruktori su oblika Set{…}
  - Set{"a", "a", "b", "c"} ima 3 elementa
  - Set{s in AllStudents, Grade(s) == 6; s}
- Operacije (za set s)
  - in (∈)
  - + (unija), * (presjek), - (skupovna razlika)
- Deklaracija
  - Set<string> s = Set{"a", "b"};

Sequence

- Uređena kolekcija
- Konstruktori su oblika Seq{…}

```
Seq{1, 2 ,3}
Seq{x in MyElements, x > 12; x}
```

- Operacije (za sequence s)
  - in (∈)
  - + (konkatenacija)
  - s.Head, s.Tail, s.Length, s.Values
- Deklaracija

```
Seq<int> s = Seq{3, 2, 1};
```


Map

- Map asocira ključeve i vrijednosti, poput tabela i rječnika
- Konstruktor je oblika Map{…}
  - Zapisi mogu biti designirani s a :> b gdje je a ključ, a b vrijednost
  - Map{"book" :> Noun, "jump" :> Verb}
- Deklaracija
  - Map<string, PartOfSpeech> m;

Map operacije

- Provjera ključa u map-u, upotreba ključa za dohvat vrijednosti

```
let myMap = Map{1 :> "a", 2 :> "b"};
if (2 in myMap)
  return myMap[2];
```

- Ubacivanje para ključ/vrijednost

```
myMap[3] = "c";
```

- Unija
  - "+"

Ograničeni kvantifikatori

- Opći oblik:
  Q { generator+ , constraint* ; predicate }
  - Q je 'Exists' ili 'Forall'
- Primjer
  - svi učitelji svih studenata su zaposlenici:

```
Forall{s in AllStudents,
       t in Teachers(s); IsEmployee(t)}
```


Tuple

- Tuple predstavlja zapis koji sadrži dvije ili više vrijednosti unaprijed određenih tipova
  - Različito od sequence — broj elemenata uvijek fiksiran; tipovi mogu biti različiti
  - Analogno zapisima u bazi podataka
- Konstruktori su oblika

```
<"a", 2>
```

- Deklaracija

```
<string, int> myTuple = <"abc", 23>;
```

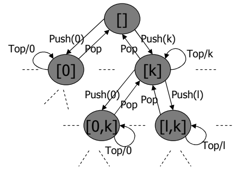

Eksplozija prostora stanja

- Tipično, modeli imaju kompleksna stanja
  - Objekti
  - Set, map, sequence, itd.
  - → Općenito, tranzicijski sistem je beskonačan
- Akcije mogu uzimati parametre
  - Stringovi, integeri, objekti, …
  - Često nekoliko parametara, s ogromnim ili beskonačnim brojem kombinacija

Prostor stanja modela stoga

► Stack model

```
var Seq<int> content = Seq{};
type MyInt = int;
[Action]
public void Push(MyInt x) {
  content = Seq{x} + content;
}
[Action]
public void Pop() {
  requires !content.IsEmpty;
  content = content.Tail;
}
[Action]
public int Top() {
  requires !content.IsEmpty;
  return content.Head;
}
```



Kontrola eksplozije prostora stanja

- ► Cilj
  - Kreiranje prostora stanja prikladne veličine koji zadovoljava zadani cilj testiranja
- ► Dvije osnovne zadaće
  - Restringiranje domene parametara akcija samo na interesantne vrijednosti
    - ► Ovo se u literaturi obično naziva kombinatorno testiranje
  - Restringiranje prostora stanja samo na interesantna stanja
    - ► Obično direktnim pretraživanjem i selektivnim izborom akcija
- ► Napomena:
  Ove dvije zadaće nisu nužno nezavisne!

Definiranje parametara akcija

- ► Parametri akcija definirani su preko izraza za domenu parametara akcija
  - Dana je akcija a s formalnim parametrom x koji je tipa T
  - Za x treba biti dan izraz E koji je tipa Collection<T>
    - ► U stanju s E se evaluira kako bi se dobile vrijednosti parametara $v_1,\ldots,v_k$
    - ► Ovime se generiraju tranzicije $(s,(a(v_i),o_i),t)$ takve da je $a(v_i)$ omogućena u s.
- ► Za akcije s više argumenata, vrijednosti se kombiniraju
  - u *Kartezijevom produktu*, ili
  - u *parovima*
- ► Napomena: izrazi koji definiraju domenu mogu biti *ovisni o stanju*

Parametri za 'Push'

- ► U projektu stack, \`Push' za argument x ima domenu Seq{1,2,3}.
- ► Provjera:
  - Otvorite 'Exploration Settings'
  - Selektirajte metodu \`Push'
  - Click na \`Edit'

Restringiranje prostora stanja

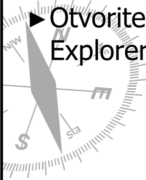
- ► SpecExplorer koristi slijedeće tehnike:
  - Filteri stanja (state filters)
  - Grupiranje stanja (state groupings)
  - Omogućavajući uvjeti na akcijama (enabling conditions)
- ► Obično je potrebna kombinacija nekih (ili svih) tehnika kako bi se željeni cilj ostvario

Filteri stanja

- ► Brute-force način za rezanje tranzicija
  - Za dani filter stanja φ,
    tranzicija (s, l, t) dodaje se u generirani skup tranzicija akko t zadovoljava φ,
    - ► Uočimo da u pojedinom koraku stanje s već zadovoljava φ

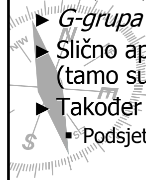
Filteri stanja: primjer

- ► Dodajmo slijedeći filter u model stoga
  - Dopušteni su stogovi s najviše 2 elementa: content.Size < 3
- ► Otvorite stack projekt u SpecExploreru: Explorer Settings -> Filters



Grupiranje stanja

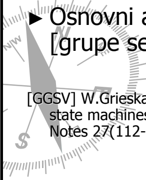
- ► *Grupiranje* G je niz izraza $g_1,\ldots,g_k$ koji mogu ovisiti o stanju
- ► Dva stanja s i t nalaze se u istoj grupi G (kažemo da su *G-ekvivalentna*) ako
  - $g_i^s = g_i^t$ za $1 \leq i \leq k$
- ► *G-grupa* je skup svih G-ekvivalentnih stanja
- ► Slično apstrakciji predikata u model-checkingu (tamo su grupiranja Booleovska svojstva)
- ► Također se koristi u view-ovima
  - Podsjetnik: DisplayMode view u Stopwatch primjeru



Upotreba grupiranja pri generiranju FSM-a

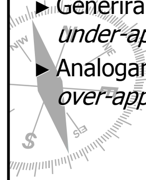
- ► Svakom grupiranju G asociran je cjelobrojni izraz B (grouping bound)
- ► Prilikom generiranja FSM-a, tranzicija (s, l, t) je izostavljena iz FSM-a ako za *sva grupiranja* G, $B^t \leq$ trenutnog broja stanja u G-grupi
- ► Osnovni algoritam opisan je u [GGSV]. [grupe se nazivaju *hiperstanjima* u [GGSV]]

[GGSV] W.Grieskamp, Y.Gurevich, W.Schulte, and M.Veanes. Generating finite state machines from abstract state machines. *ISSTA'02*, Software Engineering Notes 27(112--122), 2002.



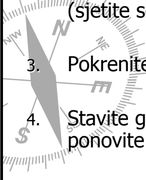
Neke napomene oko grupiranja

- ► Grupiranje G s konačnom domenom inducira konačnu particiju $R_{/G}$ tranzicijskog sistema R:
  - $R_{/G} = \{(G^s, l, G^t) : (s, l, t) \in R\}$
- ► Problem generiranja $R_{/G}$ je općenito neodlučiv
- ► Generiranje FSM-a tipično daje *under-approximation* od $R_{/G}$
- ► Analogan problem u model-checkingu tipično daje *over-approximation* od $R_{/G}$



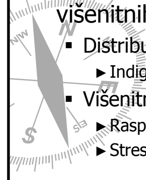
Grupiranje: Primjer

1. Otvorite stack projekt u SpecExploreru: ‘Exploration Settings’ -> ‘Search Space’ -> ‘Representative examples’ -> ‘Size of Content’ -> `Edit’ Group by: ‘content.Size’
2. Što bi trebao biti rezultat nakon generiranja FSM-a? (sjetite se da je također aktivan i filter)
3. Pokrenite FSM Generation radi provjere.
4. Stavite grupiranje obzirom na ‘content.Size == 1’ i ponovite eksperiment.



Gdje se u testiranju pojavljuje nedeterminizam?

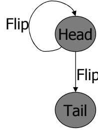
- ► Model je apstraktniji od implementacije
  - Nedeterminizam modela
    - ► unutrašnji *(internal)*
    - ► plitki *(shallow)*
- ► Rastuća potreba za testiranjem distribuiranih i višenitnih (multithreaded) aplikacija
  - Distribuirano testiranje
    - ► Indigo
  - Višenitne aplikacije
    - ► Raspoređivanje niti (thread scheduling)
    - ► Stress-testing



Unutrašnji nedeterminizam

- *Unutrašnji nedeterminizam* se javlja kada su za danu labelu (invokacija i rezultat) moguća dva ili više ciljnih stanja
- Uzmimo za primjer slijedeći program:

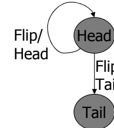
```
enum Coin {Head,Tail};
Coin coin = Coin.Head;
void Flip(){
  choose (x in enumof(Coin)){
    coin = x;
  }
}
```



Plitki nedeterminizam

- *Plitki nedeterminizam* javlja se ukoliko je sustav nedeterministički, ali povratna vrijednost akcije a u stanju s jedinstveno određuje ciljno stanje, tj. za dane dvije tranzicije (s,l,t) i (s',l',t') vrijedi: ako s=s' i l=l' tada t = t'.
- Slijedeći model ima samo plitki nedeterminizam:

```
enum Coin {Head,Tail};
Coin coin = Coin.Head;
Coin Flip(){
  choose (x in enumof(Coin)){
    coin = x;
    return x;
  }
}
```



Napomene vezane uz SpecExplorer

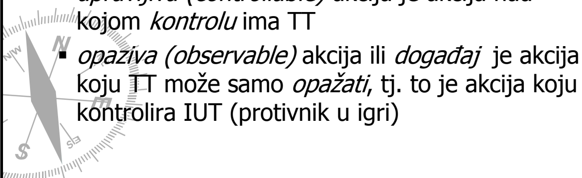
- U trenutnom izdanju SpecExplorera nedeterminizam modela nije podržan
- *Plitki* nedeterminizam bit će podržan u narednim izdanjima
- SpecExplorer podržava *vanjski* nedeterminizam preko *opazivih* akcija (*događaja*)

Testiranje nedeterminističkih sustava

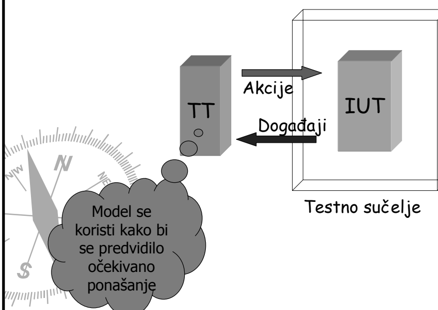
- Cilj je testirati nedeterminističke sustave "kao takve"
- Testiranje se može vidjeti kao *igra* između dva igrača:
  - Alata za testiranje (Testing tool – TT)
  - Implementacije koja se testira (Implementation under test – IUT)
- Test slučajevi postaju *strategije*
  - Strategija govori koje poteze mora napraviti TT u stanju u kojem ima omogućene poteze
- Test verifikacija uspoređuje povratne vrijednosti

Vanjski nedeterminizam preko opazivih akcija

- Skup svih akcija u modelu je podijeljen u dva disjunktna skupa: *opazive* akcije i *upravljive* akcije. Intuitivno:
  - *upravljiva (controllable)* akcija je akcija nad kojom *kontrolu* ima TT
  - *opaziva (observable)* akcija ili *događaj* je akcija koju TT može samo *opažati*, tj. to je akcija koju kontrolira IUT (protivnik u igri)



Proces testiranja



Primjer vanjskog nedetermizma

- Stanje modela:

```
enum Coin {Head,Tail};
Coin coin = Coin.Head;
bool flipping = false;
```

- Upravljiva akcija: kaže IUT da počne bacati novčić, ukoliko bacanje nije otpočelo

```
[Action]
void Flip()
  requires !flipping;
{flipping = true;}
```

- Opaziva akcija: (odgovor IUT-a)
  - Glava (Head)
  - Pismo (Tail)

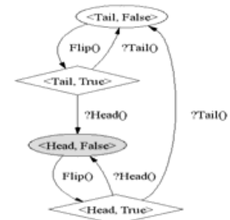
```
[Action(Kind=Observable)]
void Head()
  requires flipping;
{ coin = Coin.Head;
  flipping = false;}

void Tail()…
```


Točke izbora

- *Točka izbora* je stanje u kojem je moguć neki događaj
  - Točke izbora također se nazivaju *nestabilnim* stanjima
- U SpecExploreru:
  - Točke izbora su predstavljene rombovima
  - Događaji su označeni s '?'

FSM generiran iz prethodnog primjera:



Timeout

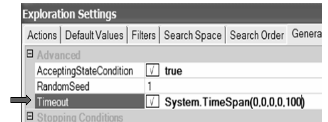
- *Timeout* d je izraz (ovisan o stanju) koji opisuje vremenski raspon
  - Aproksimira *mirovanje* – odsutnost događaja ili opazivih akcija
- Značenje timeout tranzicije (s, d^s, s'): timeout se "javlja" u stanju s ukoliko nije bio opažen niti jedan događaj unutar vremena d^s, nakon čega sistem prelazi u stanje s' (s "nakon" timeouta)

Timeout: primjer

- Dodajte slijedeću timeout akciju u primjer s bacanjem novčića:

```
[Action]
void Cancel(){
  requires flipping;
  flipping = false;
}
```

- Unos timeouta:
  - `Cancel' je omogućen kada je IUT u stanju bacanja novčića i izvršava se ukoliko nije opažena nikakva reakcija od IUT unutar 100ms

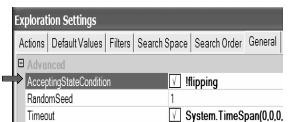


Prihvaćajuća stanja

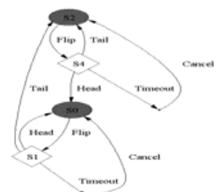
- Prihvaćajuća stanja specificiraju se preko Booleovskog izraza φ_{acc}
  - s je *prihvaćajuće stanje* ako s zadovoljava φ_{acc}
  - *mrtvo stanje* je stanje iz kojeg nije moguće dohvatiti prihvaćajuće stanje
- Svi test slučajevi moraju završiti u prihv. stanjima
- U SpecExploreru:
  - Po defaultu sva stanja su prihvaćajuća
  - Vizualizacija prihvaćajućih stanja može biti uključena/isključena putem `view properties'

Prihvaćajuća stanja: primjer

1. Unesite uvjet za prihvaćajuće stanje u primjer s bacanjem novčića:



2. Generirajte FSM:



Testiranje kao igra

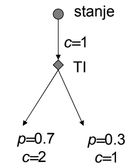
- TT poziva *upravljive akcije* prema predefiniranoj *testnoj strategiji*
- IUT odgovara putem različitih povratnih vrijednosti ili pozivanjem *opazivih akcija*
- Testne strategije se generiraju kako bi se ostvarile različite *namjene testiranja*

68

Test graf

A *test graf* G je usmjeren graf t.d.

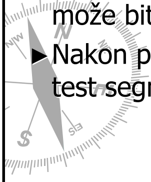
- postoje dva tipa vrhova u *G*:
  - stanja
  - točke izbora (TI)
- za svaki brid e definirana je vjerojatnost p(e) t.d. za svaku TI *u* $\sum\{p(e): u$ je poč. vrh od $e\}=1$
- postoji nenegativna funkcija cijene *c* definirana na bridovima



69

Test slučajevi kao strategije

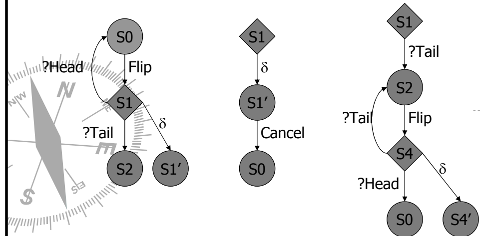
- Test slučaj je skup *test segmenata*.
- Svaki segment završava s *točkom izbora* reflektirajući na taj način različite poteze koje može odigrati IUT (od kojih jedan može biti i timeout).
- Nakon poteza IUT, strategija govori s kojim test segmentom treba nastaviti TT.



70

Test strategija: primjer

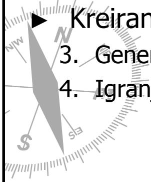
- Generirajte test slučajeve za primjer s bacanjem novčića. Primjer test segmenata u generiranom test slučaju gdje je δ=Timeout(100ms):



71

Pogled na kompletnu sliku

- Kreiranje FSM-a
  1. Modeliranje: Definiranje (beskonačnog) tranzicijskog sistema
  2. Eksploracija: Svođenje na konačan test graf G
- Kreiranje i pokretanje testova
  3. Generiranje strategije za G
  4. Igranje igre prema zadanoj strategiji



72

Algoritmi za generiranje test strategija

1. Eliminacija mrtvih stanja
2. Prekrivanje svih bridova u test grafu (edge coverage)
3. Dohvat određenih ciljnih stanja u test grafu (reachability game)
   - Optimalna strategija minimalne cijene
   - Pobjednička strategija minimalne cijene

Algoritmi za 2 i 3 detaljno su opisani u [NVSTG].

[NVSTG] L.Nachmanson, M.Veanes, W.Schulte, N.Tillmann, W.Grieskamp, Optimal Strategies for Testing Nondeterministic Systems, *ISSTA'04*, *Software Engineering Notes,* 2004.

73

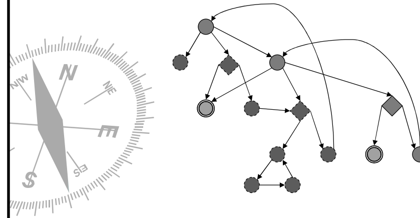
Eliminacija mrtvih stanja

- Pretpostavljajući *poštenost* IUT-a (tj. u točkama izbora ne postoji izbor koji neće biti odabran), *mrtvo stanje* je stanje u kojem TT nema pobjedničke strategije čak i s neograničenim brojem koraka.
- Tipično ograničenje na test slučajeve jest da završavaju u prihvaćajućim stanjima. Iz tog razloga mrtva stanja trebaju biti eliminirana iz generiranog FSM-a.
- Algoritam koji koristi SpecExplorer može se naći u [Alfaro99].

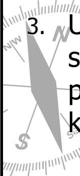
[Alfaro99] L. de Alfaro, Computing minimum and maximum reachability times in probabilistic systems. CONCUR'99, vol 1664 LNCS, p. 66-81, 1999.

Mrtva stanja: primjer

- U slijedećem grafu prihvaćajuća stanja su dvostruko zaokružena (zelena) a mrtva stanja su iscrtkana (crvena).

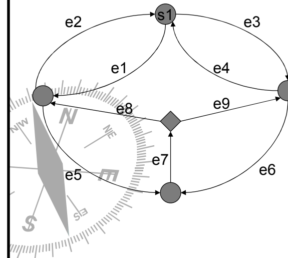


Strategija za prekrivanje bridova



1. Generiranje ture na bridovima u grafu (npr. tura kineskog poštara)
2. Podjela ture u segmente koji počinju i završavaju u točkama izbora
3. U točki izbora IUT odabire brid *e*, a TT slučajno odabire bilo koji segment *s* s početkom u *e*, te slijedi osabrani *s* do krajnje točke (*s* završava s točkom izbora)

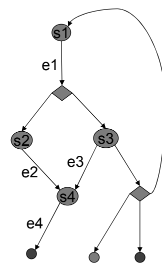
Prekrivanje bridova: Primjer



- Tura: [e8,e2,e1,e5,e7, e9,e4,e3,e6,e7]
- Segmenti: [e8,e2,e1,e5,e7] [e9,e4,e3,e6,e7]

Reachability games: Pobjednička strategija

- Generiranje strategije S kojom se stiže u neko od ciljnih stanja (crvena):
  S(s1)=e1
  S(s2)=e2
  S(s3)=e3
  S(s4)=e4
- Algoritam koji se koristi je proširenje Dijkstrinog alg. za najkraće puteve ([NVSTG])

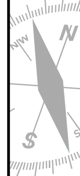


Bounded reachability games

Zadano je:
G=(*V*,*E*), ciljna stanja *P*, max. broj koraka u igri *n*

Trenutno stanje:
Vrh *v*, broj odigranih koraka *k*
Inicijalno: *v=s*, *k=0*

Igrači igraju poteze prema slijedećim pravilima:



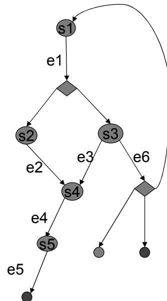
```
if (v is a choice point){
   if (k ≥ n) {TT  loses}
   else { IUT chooses an edge (v,u); v=u; k=k+1;}
}
else if (v in P) TT wins and the game stops;
else if (k ≥ n) TT loses and the game stops;
else {
  TT chooses an edge (v,u) or loses if there is no choice;
  v=u; k=k+1;
}
```

Bounded Reachability games: Računanje optimalne strategije

Zadano je:
G=(V,E), P, n max. broj koraka u igri

Rezultat je strategija:
S: Vx{0..n} → E∪{null}

Iteracije algoritma:
- 0 : S(_,0)=null
- 1: S(s5,1)=e5
- 2: S(s4,2)=e4
  - S(s3,2)=e6
- 3: S(s2,3)=e2
  - S(s3,3)=e3 //poboljšanje
- 4: S(s1,4)=e1
- 5: nema izmjena



Modeliranje višenitnih aplikacija

- Pod višenitnom aplikacijom podrazumijevamo API koje može biti izvršavan od strane više istodobnih dretvi.
- Tipično se koriste razni tipovi objekata za zaključavanje kako bi se garantirao atomarni pristup dijeljenim memorijskim lokacijama, te na taj način zaštitili dijeljeni podaci i resursi od korupcije.
- U high-level modelu obično je nužno *apstraktirati se* od implementacijskih specifičnosti pojedinih mehanizama zaključavanja.

Primjer: Bag (1)

- Otvorite bag model u SpecExploreru i upoznajte se s modelom
- Otvorite bag implementaciju u Visual Studiu
- Proučite postavke u `Exploration Settings'
- Generirajte FSM
  - Inicijalno samo s jednom dretvom (NrOfThreads = 1)
- Genirajte test slučajeve
- Pokrenite test slučajeve

Primjer: Bag (2)

- Bag implementacija sadrži namjernu locking grešku koja ne može biti otkrivena koristeći samo jednu dretvu
  - Promijenite broj dretvi u modelu na 2 (NrOfThreads = 2)
  - Generirajte i pokrenite test slučajeve kako biste otkrili grešku
  - Pokušajte ukloniti grešku u implementaciji i rekompilirajte
  - Ponovo pokrenite eksperiment

Trenutno aktualni projekti @ MSR vezani uz Spec#

- Boogie
  - Statička verifikacija Spec# ugovora
- Integracija s Zing-om
  - Zing je framework za model-checking razvijen @ MSR
- Daljnji razvoj događaja pratite na web stranici FSE @ MSR:
  - http://research.microsoft.com/fse/