UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

DEPARTAMENTUL INFORMATICĂ

SPECIALIZAREA TEHNOLOGIA INFORMAȚIEI

**TESTAREA SISTEMELOR SOFTWARE**

**MODEL BASED TESTING AND ANALYSIS WITH C# AND NMODEL**

PROFESOR COORDONATOR:

Asist.dr. Sorina Preduț STUDENȚI:

Costea Marina

Gembășel Valerică

Voinea Radu

Voinescu Florin

GRUPA 452

**CUPRINS**

**1. NModel și model program**...............................................................................................2

**2. Testarea bazată pe modele**...............................................................................................2

**3. Scenario control**................................................................................................................3

**4. Features**.............................................................................................................................4

**5. Defining features**...............................................................................................................4

**6. Combining features**..........................................................................................................4

**7. Compunerea**......................................................................................................................4

**8. FSMs for scenario control**.............................................................................................11

**9. Alegerea dintre opțiunile pentru controlul scenariilor**...............................................12

**10. Compunerea pentru analiză**........................................................................................14

**11. Bibliografie**....................................................................................................................19

**1. NModel și model program**

*NModel* este un set de instrumente dezvoltat în limbajul C# care poate fi folosit pe diverse platforme pentru testarea aplicațiilor. În cadrul arhitecturii NModel, modelul este numit *program model*. Modelele sunt scrise în C#, iar testele sunt generate automat pe baza acestora, care pot fi utilizate la rândul lor pentru a prezice rezultatele testelor.

Utilizatorul este nevoit să scrie un program model pentru fiecare implementare pe care dorește să o testeze. În NModel sunt două tipuri de programe model:

* *Contract model program*, care reprezintă o specificație completă a sistemului după care este modelat. Acesta generează fiecare "trecere" a programului care include erori cunoscute și răspunsuri adecvate pentru acestea. Contract model programs sunt în general scrise în limbajul C#.
* *Scenario model program* restrânge toate rulările posibile într-o colecție de rulări, care sunt legate într-un fel sau altul, sub forma unui subset cu toate comportamentele permise. Scenario model programs se pot descrie ca teste abstracte deoarece este posibilă specificarea anumitor stări și secvența în care stările trebuie traversate în timp ce părăsesc stările intermediare. Acestea pot fi scrise fie sub forma unor automate cu stări finite (Finite State Machines), fie implementate în C#.

**2. Testarea bazată pe modele**

În contrast cu testarea pe unități, unde testele sunt scrise manual, în testarea bazată pe modele, testele sunt generate automat pe baza modelelui care descrie comportamentul așteptat. Prin urmare, testarea bazată pe modele este recomandată în cazurile în care este necesar un număr mare de teste pentru a obține o acoperire adecvată, și care ar fi imposibilă de atins din scrierea testelor manual.

Testarea bazată pe modele este în special indicată acolo unde o implementare trebuie să permită comportamente dependente de date în continuă desfășurare. Un astfel de exemplu sunt serviciile web, unde comportamentul utilizatorului este imprevizibil. Pentru a testa automat o aplicație web, se înlocuiește utilizatorul cu un instrument de testare. Instrumentul de testare generează date de test care corespund cu comenzile unui utilizator, acestea includ trimiterea de cereri către server și verificarea răspunsului pentru a stabili dacă fiecare test trece sau eșuează.

Testarea se poate realiza în două moduri:

* offline
* online

În testarea offline, testele sunt generate în avans și stocate până în momentul execuției. Acestea pot avea o mărime destul de mare, incluzând multe secvențe similare datorită comportamentului imprevizibil.

În cazul testării online, numită de asemenea și testare *on-the-fly*, testele sunt generate din model în timpul execuției. Testarea online poate fi realizată prin teste aleatoare, sau poate fi ghidată după o strategie specifică pentru a testa comportamente particulare, sau pentru a maximiza aria de acoperire.

Testele sunt generate pentru fiecare caz de utilizare, sau pe scurt, pentru fiecare scenariu.

**3. Scenario control**

Problema limitarii analizei si a testarii a unei arii de interes se numeste controlul scenariului. Controlul scenariului este necesar deoarece atunci cand se scrie un program model care sa fie folosit ca specificatie sau contract, acesta ar trebui sa descrie tot ce implementarea trebuie sa faca, ar putea sa faca si ce nu trebuie sa faca. Ca rezultat, programul model de obicei descrie un numar mare de rulari. Cand se testeaza nu se folosesc toate acele rulari, se folosesc doar cateva, poate doar una, care se leaga de o problema.

Intr-un model putem identifica caracteristici individuale pe care putem realiza analiza sau efectua testarea pentru caracteristici particulare.

**4. Features**

O caracteristica este un cluster de variabile de stare si actiuni care poti fi incluse sau excluse la alegere dintr-un program model. Folosind caracteristicile se definesc diferite configuratii ale programului model intr-un singur fisier sursa sau colectie de fisiere.

**5. Defining features**

Pentru a declara o caracteristica trebuie sa se declare toate variabilele destare si metodele intr-o clasa. Numele caracterisitcii este numele clasei. Clasa se eticheteaza cu atributul [Feature]. Clasele care nu sunt etichetate cu [Feature] sunt sunt incluse. Variabilele de stare care sunt declarate in caracteristica se folosesc doar in caracteristica, daca o variabila se foloseste in mai multe caracterisitci atunci acele variabilese declara in interiorul clasei. Caracteristicile sunt incluse intr-un program model numindu-le in constructorul LibraryModelProgram.

**6. Combining features**

Caracteristicile poti fi combinate pentru a modela caracteristici interactive in implementare sau pentru creea controlul scenariului. Pentru a combina caracteristici se dau aceleasi nume unor metode in caracteristici, toate metodele care au acelasi nume fac parte din aceeasi actiune si sunt executate impreuna.

**7. Compunerea**

Putem scrie modele separate de programe și apoi să le combinăm folosind compunerea.

Compunerea a două sau mai multor programe combină modele separate de programe într-un nou program model numit produs (the product). Produsul format prin compunerea programelor model M1 și M2 poate fi scris M1 x M2 sau M1 \* M2. Compunerea este tehnica de transformare a programului care se efectuează automat utilizând instrumente de analiză și de testare atunci când mai multe programe model sunt introduse pe linia de comandă; instrumentele pot apoi să analizeze sau să testeze produsul.

Se poate realiza compunerea programelor pe care le scriem folosind o librărie de compunere. Compunerea poate fi utilizată pentru a construi modele complexe prin combinarea programelor simple într-unul singur. În lucrarea aceasta vom aborda compunerea pentru controlul scenariilor (scenery control).

Compunerea este definită și implementată într-un mod convenabil astfel încât să optimize analiza și să se poată realiza testarea unor caracteristici sau scenarii particulare.

Caracteristicile unui program (features) sunt clase în același program model (în același namespace), care poate utiliza aceleași variabile de stare și sunt compilate în aceiași sesiune (assembly). În contrast, putem să compunem programe model, dar care se regăsesc în namespaceuri diferite și sunt compilate în sesiuni diferite. Toate instrumentele acceptă orice număr de modele și realizează produsul prin combinarea acestora. Pentru realizarea compunerii nu este necesară utilizarea de cod special, orice două modele pot fi compuse însă acest lucru nu garantează utilitatea produsului.

Produsul a două sau mai multe programe conține toate variabilele de stare și toate acțiunile implementate de fiecare program individual. Acțiunile care au același nume în două sau mai multe programe sunt reprezentate ca o singură acțiune în produsul final. Acestea se execută împreună și pot fi executate doar atunci când acțiunile sunt simultan active în programele care participă la compunere. Datorită necesității ca aceste acțiuni să fie active în toate programele, de cele mai multe ori rezultă restricționarea comportamentului și eliminarea unor rulări. De accea compunerea este folositoare la controlul scenariilor. Compunerea restricționează comportamentul la intersecția rulărilor din fiecare program în parte. Pe scurt, rulările pe care le realizează programul compus sunt aceleași pe care le permit și programele individuale.

Cu toate acestea, există o excepție importantă: acțiunile care nu sunt partajate pot intercala (interleave) în orice ordine în produsul final și pot fi executate ori de câte ori sunt active în propriile programe. Dacă se dorește a suprima această intercalare, este necesar să adăugăm aceste acțiuni la celelalte programe, astfel încât nu mai existe acțiuni neparticipate, sau să le dezactivăm în programele individuale. În urma compunerii, programele model își sincronizează pașii pentru acțiunile partajate și își intercalează acțiunile unice, numite și acțiuni care nu se găsesc în vocabular.

Exemplu: Avem modele de program M1 și M2. Vom ilustra stările finite (FSM sau Finite State Machine) pentru fiecare program în parte.

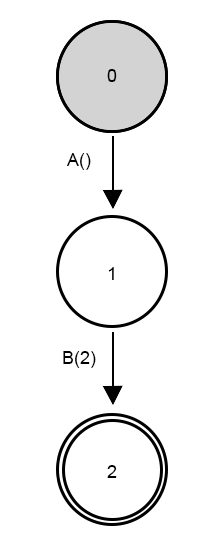


Figura 1. Program model M1

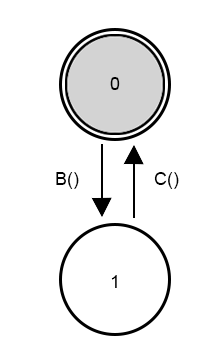


Figura 2. Program model M2

M1 implementează acțiunile A() și B(2) care au tranziții din starea inițială 0, în starea 1, până la starea acceptată 2. M2 implementează acțiunile B() și C() care au tranziții din starea 0, către starea 1, și înapoi; starea 0 reprezentând și starea inițială, dar și starea acceptată. Toate cele trei acțiuni vor fi incluse în produsul M1 x M2.

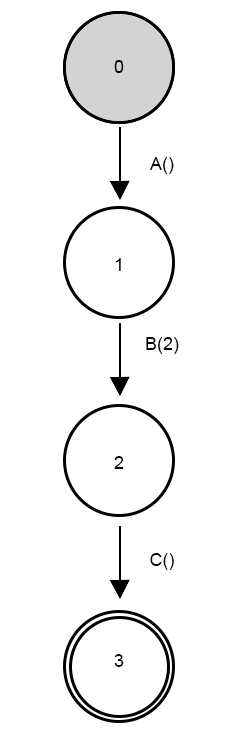


Figura 3. Produsul M1 x M2

Pentru obținerea produsul M1 x M2 a fost utilizată o metodă sistematică pentru generarea produsului a două modele de program. Primul pas reprezintă identificarea vocabularului de acțiuni pentru fiecare program, și identificarea acțiunilor nepartajate din celălalt program. Pentru programul M1 vocabularul de acțiuni este A și B, iar acțiunea nepartajata este C; pentru M2, vocabularul include B și C, iar A este acțiunea nepartajata. În continuare, se formează extensia de bucle (loop extension) a fiecărui program pentru fiecare stare, adaugandu-se o buclă de sine pentru fiecare acțiune nepartajata. Cele două extensii de bucle au acum același vocabular. Chiar mai mult, în extensiile de bucle, acțiunile cu același nume au aceiași aritate prin extinderea acestora cu un parametru înlocuitor indicat de "\_" (underscore). Prin urmare, acțiunea B() din M2 devine B(\_) în extensia de buclă a lui M2, și deci are aceiași aritate ca B(2) în programul M1. Produsul M1 x M2 este generat din aceste extensii de bucle.

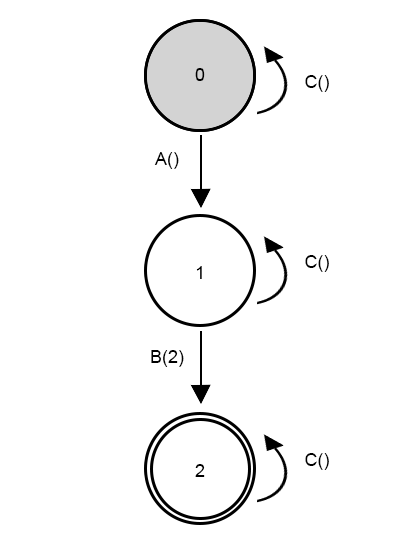


Figura 4. Extensie de bucle pentru M1

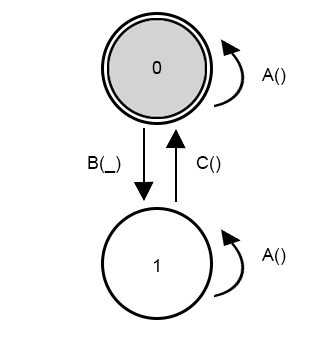


Figura 5. Extensie de bucle pentru M2

Fiecare stare a produsului este o stare pereche formată dintr-o stare a lui M1 și una a lui M2. Starea inițială (0, 0) se formează din stările inițiale ale lui M1 și M2. Fiecare acțiune care este activă într-o pereche de stare trebuie să fie activă în ambele stări corespunzătoare din extensia de buclă a ambelor programe. Acțiunea A() este singura acțiune activă în stările inițiale ale celor două programe; C() și B(\_) sunt active doar în M1, și respectiv, M2. La execuția acțiunii A() se realizează tranziția la starea 1 în M1 și se realizează o buclă de sine înapoi la starea 0 în M2. În produs, A() face tranziția la starea pereche (1, 0). În această stare, B(2) este activ în M1, și B(\_) este activ în M2. Aceste două acțiuni corespund datorită numelor identice, și a parametrului înlocuitor care corespunde cu orice argument. Deci înlocuitorul în B(\_) corespunde argumentului din B(2). Aceste acțiuni realizează tranziții în stările 2 și 1, în M1, și respectiv, M2, rezultând în perechea de stare (2, 1) în produs. Acțiunea care apare în produs este B(2), și nu B(\_). În starea pereche (2, 1), C() este activ în ambele programe, făcând o tranziție finală în starea pereche (2, 0) în produs. În această pereche de stare, nu sunt acțiuni care să fie active în ambele programe, deci explorarea se oprește, produsul fiind complet. Starea pereche (2, 0) este o stare acceptată în produs deoarece ambele stări din pereche sunt stări acceptate în propriile programe.

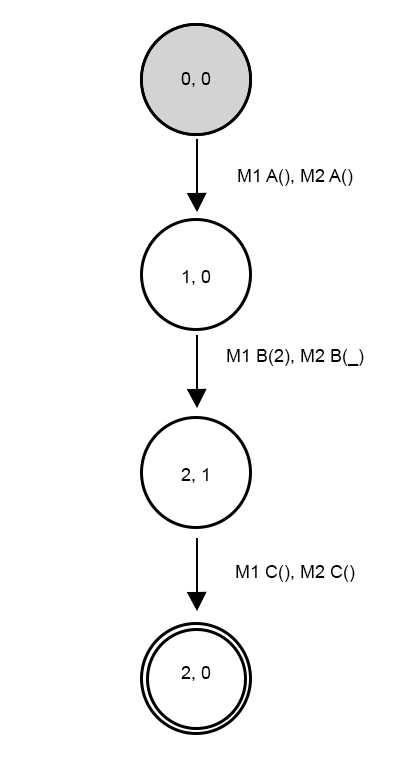


Figura 6. Programul model M1 x M2 care include starile pereche și acțiunile corespondente

Tabelul ilustrează pentru fiecare pereche de stări acțiunile care sunt active în fiecare extensie de bucle, cu acțiunile corespunzătoare și stările acceptate îngroșate.

|  |  |  |
| --- | --- | --- |
| Pereche de stări | Acțiuni active în M1 | Acțiuni active în M2 |
| (0, **0**) | **A()**, C() | **A()**, B(\_) |
| (1, **0**) | **B(2)**, C() | A(), **B(\_)** |
| (**2**, 1) | **C()** | A(), **C()** |
| (**2**, 0) | C() | A(), B(\_) |

Există câteva complicații care nu au apărut în acest exemplu simplu, și anume, în ceea ce privește argumentele acțiunilor. Atunci când acțiunile au același nume dar argumente diferite, altele decât înlocuitorul, acestea nu corespund. De exemplu, dacă M2 ar fi avut acțiunea B(3) în loc de B(\_), nu ar fi corespuns cu B(2) din M1, iar acțiunea B nu ar apărea în produs. Mai mult, este posibilă definirea acțiunilor care prezintă înlocuitori pe orice poziție, deci acțiunea D(\_, 4) într-un program ar corespunde acțiunii D(3, \_) în altul, rezultând perechea de stări D(3, 4) în produs. Câteodată este folositor a vedea proiecția produsului peste unul din programele care participă la compunere.

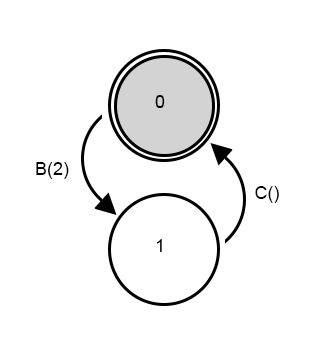


Figura 7. Proiecția produsului M1 x M2 peste M2

Proiecția este generată prin explorarea produsului. De fiecare dată când produsul exclude o acțiune a lui M2, tranziția este adăugată în proiecția peste M2. În exemplul acesta proiecția lui M1 x M2 peste M2 este similară cu programul original M2, cu excepția acțiunii B() a lui M2, care este înlocuită de acțiunea B(2) din produs. Câteodată proiecția are mai puține tranziții față de programul inițial. Acest lucru este importanat atunci când compunerea este utilizată în scopul analizei.

**8. FSMs for scenario control**

Deseori scriem un program de scenarii pentru a compune un program de model contractual pentru controlul scenariilor. Este adesea convenabil să se exprime un program de scenariu ca un FSM. Bibliotecile și uneltele suportă mai multe stiluri pentru programele de codare pentru modele.Toate exemplele de până acum au folosit doar unul dintre aceste stiluri, stilul introdus în capitolul 5 și descrise în Appendix A. În această secțiune introducem încă două stiluri care facilitează codarea FSM-urilor pentru controlul scenariilor.

Acum vom introduce un alt stil, FSM, acesta este scris ca un array de strings, unde fiecare string descrie o tranzitie in formatul "t(current state, action, next state)". In metoda de fabricatie arrayul de strings este dat FSM-ului. Un program model FSM este compilat și

afișat exact în același mod ca și programul de bibliotecă(library model program).

Acum introducem un al treilea stil, fișierul text FSM, care este cel mai compact (Figura 7.13). Acest stil eliberează în totalitate C # și ansamblurile și pune doar FSM într-un fișier text, într-un format parantez. Primul element din interiorul exteriorului paranteze este starea inițială. Cel de-al doilea element este o secvență de stări de acceptare (care este lăsată goală aici, ceea ce indică faptul că nu există stări de acceptare). Nu este necesar pentru a compila un fișier FSM. Pentru a afișa FSM în fișier, invoca mpv cu opțiunea / fsm și denumirea fișierului.

In practica scriem rar un program de model FSM scriem doar un fisier text FSM.

Instrumentele nu depind de forma pe care o are sursa programelor model. Este accesat fiecare model intr-un mod uniform printr-o intasnta a clasei ModelProgram class, aceasta din urma oferind intodeauna anumite proprietati si metode, cum ar fi InitialState, GetActions, GetTargetState.

Acum că înțelegem compoziția și știm cum să scriem programe de scenariucu ușurință, suntem gata să folosim compoziția pentru controlul scenariului.The unwanted runs pot fi eliminate prin compunerea contractului model cu un program model de scenarii care codifică run-urile pe care le dorim sa le vedem.

Dacă există o anumită secvență de acțiuni pe care dorim să o vedem în produs, punem acea secvență în programul de scenarii model. Dacă sunt alte actiuni pe care le-am dori să le intercalam în produs, nu le mentional pe acestea în programul modelului de scenarii. Dacă există acțiuni pe care dorim să le eliminam de la produs, vom include acele acțiuni din vocabularul programului de scenarii model, dar nu vom scrie nici o tranziție care să le folosească. Acest lucru are ca efect dezactivarea acestora acțiuni în toate starile.

Exista un scenariu tipic pe care deseori dorim sa-l folosim pentru testare: mai intai folosim o secventa de actiuni de configurare iar mai apoi folosim mai multe actiuni de testare intercalate iar in final o secventa de actiuni de curatare.

**9. Alegerea dintre opțiunile pentru controlul scenariilor**

Există trei moduri pentru a controla scenariile unui program model: validarea condițiilor, caracteristicile sau compunerea.

Scenariul de control ar trebui să fie separat cu caracteristicile sau compunerea, astfel încât același program model de contract (contract model program) poate analiza sau testa diferite scenarii, nu doar pe cele particulare.

Caracteristicile (features) pot accesa programul modelului de stare, astfel se utilizeaza caracteristici pentru dependența de stare a generării parametrilor de stare sau, în cazul în care condițiile de acordare a condițiilor, trebuie să se limiteze la starea programului de contract în alte moduri.

Deoarece programele compuse nu pot imparti starea, compunerea se utilizeaza atunci cand se poate descrie scenariul dorit in termeni de secvente de actiuni, fara a se referi la starea programului model al contractului, in mod special atunci cand scenariul poate fi exprimat ca FSM (Finite State Machine).

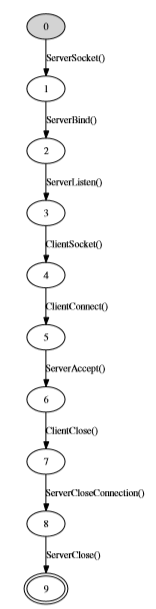


Figura 7.14. Client / server: scenariul de testare.

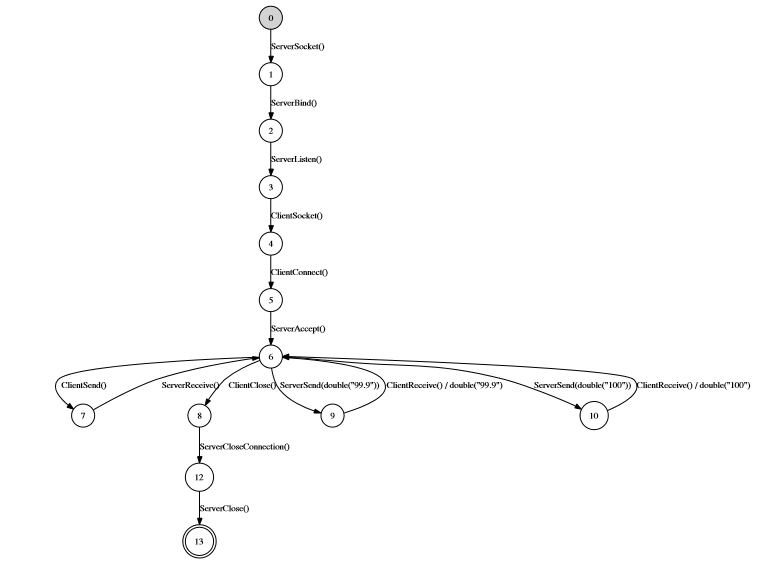


Figura 7.15. Client / server: scenariu de testare compus cu programul modelului contractului

**10. Compunerea pentru analiză**

Pentru analiza poate fi de folos simpla vizualizare a graficului FSM, dar este mai util să investigați anumite probleme. Explorarea, de exemplu, poate oferi răspunsuri la mai multe tipuri de întrebări specifice. De exemplu, ele dezvăluie dacă explorarea a generat de fapt FSM adevărat. Uneori, insa, este mai ușor să se exprime cerințe în termeni de acțiuni decât de stari. De exemplu, în sistemul reactiv, putem exprima cerința de siguranță în acest fel: acțiunea de calibrare nu trebuie să urmeze sosirea unui mesaj care să conțină o temperatură in afara parametrilor (out-of-range). Iar cerința de viață este: acțiunea de calibrare trebuie să urmeze sosirea unei comenzi. O proprietate exprimată în acest fel, prin descrierea ordonării acțiunilor, este o proprietate temporală.

Scenariul programului model (Scenario Model Programs) poate exprima proprietățile temporale, iar compunerea o poate verifica.

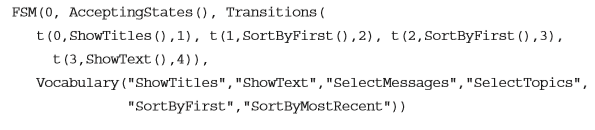
Se compune contractul programului model cu programul de scenariu care exprimă proprietatea, se examinează proiecția producției în scenariu,iar in cazul în care proiecția este completă (contine tranzitii din cadrul programului de modele pentru modelul de bază), atunci contractul programului model poate executa scenariul respectiv și se va putea găsi rulajele scenariului în produs. În cazul în care proiecția este incompletă (dacă se înregistrează semnale tranzitorii), programul de contract nu poate executa acest scenariu și nu se vai găsi nicio rulare a scenariului din produs.

Se poate observa cum această utilizare a compunerii diferă de controlul scenariului, unde scriem un scenariu pe care dorim să îl execute programul de contract, pe care îl presupunem că îl poate executa. Pentru analiză, se scrie un scenariu pe care programul de model al contractului nu îl poate executa; asta este ceea ce vrem să aflăm. Să ne uităm la câteva exemple. În primul rând, luăm în considerare un exemplu în care modelul de scenariu descrie exact modul de execuție. Analizează dacă scenariul programului model poate executa acea execuție; furnizează aceleași informații ca și simularea sau explorarea interactivă , dar poate fi mai ușor de configurat și repetat.

Pentru a face acest lucru ușor de înțeles, se foloseste exemplul simplului de cititor de știri.Vom ce se întâmplă atunci când programul modelului contract nu poate executa scenariul. Alegem urmatorul script:

https://lh3.googleusercontent.com/ugzlSDxJlpPXxvPdj-Axdq2CsXm3mQxpwE4k39PLWw92DdJn2W48urrDDe_NyFmWm9QSxBsgpFf82nKztrru_tr5U4jkk_0xLHIuLszxa1TkUNTA-m9XeA9YvnJVY-vqQvRJb5Sm

Codificăm această rulare ca scenariu într-un fișier text cu FSM cu următorul conținut:



Acest fisier text FSM conține o intrare la sfârșit care defineste vocabularul mașinii de scenarii, numele tuturor acțiunilor sale. Dacă această intrare este omisă, vocabularul este înțeles ca fiind acțiunile care apar în tranziții.Trebuie sa se listeze vocabularul deoarece include acțiuni care nu apar în niciunul dintre tranzițiile acestui scenariu. Atunci când folosim compunerea pentru a verifica dacă programul de model al contractului poate executa o anumită rulare, trebuie să includem întregul vocabular al programului modelului contract în programul model de scenarii. În produs, orice acțiuni ale programului model de contract care nu apar în vocabularul scenariului ar putea interfera oriunde între acțiunile scenariului. Definirea vocabularului scenariu, așa cum facem aici, împiedică aceste interferari nedorite să confunde analiza.

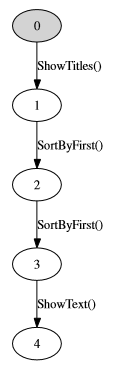


Figura 7.16 Scenariu de știri FSM pentru analiza prin compoziție

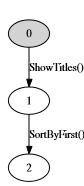
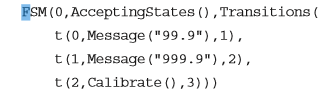


Figura 7.17. Cititor de știri FSM, proiecție incompletă după compoziție.

O încălcare a siguranței apare dacă programul efectuează o calibrare după ce recepționează un mesaj care conține o temperatură de depășire a temperaturii. În cazul în care calibrarea este inacordibilă, programul trebuie să primească mai întâi o temperatură în raza de acțiune. Încălcarea siguranței poate fi descrisă printr-un scenariu care conține doar cele trei acțiuni:



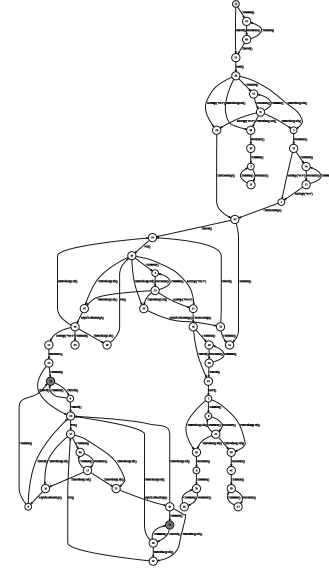


Figura 7.19.Sistem reactiv: proiectie dupa compunere pentru analiza de siguranta.

Structurarea programului model cu caracterisiti si compunere.

Este important să înțelegem că aici nu e nevoie să identificăm stările nesigure pentru a efectua această analiză, ci a trebuie doar să scriem scenariul. Mai mult, nu trebuia să descriem complet căile în scenariul nostru, identificați ce este comun cu rutele nesigure: câteva acțiuni și ordonarea acestora.

Se poate observa ca o parte din contractul FSM apare aici în FSM proiectat. Lobul stâng al contractului FSM nu apare în proiecție, deoarece pe toate căile spre lobul respectiv, raza de raportare a mesajelor de la distanță nu depășește 99,9 (vezi Figura 6.12). Prin urmare, acele căi nu se limitează la prețuri și sunt excluse de la produs. Într-o astfel de lobă apar astfel de stări, astfel încât acestea nu sunt accesibile de scenariul pe care l-am specificat. Cele două analize produc rezultate similare, dar nu identice.

**11. BIBLIOGRAFIE**

[1] Model-Based Software Testing and Analysis with C#, Jonathan Jacky, Margus Veanes, Colin Campbell, Wolfram Schulte

[2] a scenario driven approach to model based... (PDF Download Available). Available from: https://www.researchgate.net/publication/228938532\_a\_scenario\_driven\_approach\_to\_model\_based\_testing

[3] Model-Based Scenario Testing and Model Checking with Applications in the Railway Domain - Helge L¨oding Bremen im September 2014

[4] Model-Based Testing of Web Applications using NModel, Juhan Ernits, Rivo Roo, Jonathan Jacky and Margus Veanes: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/extended\_version.pdf