TOLERANTA SISTEMELOR IN RAPORT CU DEFECTELE SOFTWARE

Dr.Ing.Mat. Ion Bucur

S-a scris mult despre cauzele care conduc la sensibilitatea software-ul la defecte dar şi despre raţiunile care fac ca proiectarea şi scrierea software-ului să fie intrinsec dificilă. Cercetarea a dovedit că există atât dificultăţi esenţiale cât şi accidentale care sunt implicate în producţia software-ului. Dificultăţile esenţiale ţin de natura aplicaţiilor complexe ca şi de mediul complex de operare. Aceste dificultăţi îşi au sorgintea şi în construcţia cu un număr foarte mare de stări, guvernate prin legi de tranzitare complexe.

Introducerea, frecventă, pe durata ciclului de viaţă a software-ului a modificărilor impuse de reglementări ori de situaţii noi. Astfel, sunt adăugate software-ului noi faciltăţi pentru a adapta o aplicaţie software la cerinţele actualizate ale contextului de utilizare.
Platformele hardware şi sistemul de operare se pot modifica ulterior perioadei în care a fost proiectată şi implementată o aplicaţie software iar aceasta trebuie ajustată corespunzător.
Software-ul este, adesea, utilizat să mascheze incompatibilităţile dintre componentele unui sistem de calcul.

Consideraţiile privind costurile nu pot fi ocolite atunci când sunt alternative în implementarea unei aplicaţii complexe. De cele mai multe ori se face apel la software existent, deja disponibil contra cost (COTS *Commercial Off-The-Shelf*) care nu este proiectat întotdeauna pentru aplicaţii de mare fiabilitate.
Utilizarea COTS este mandatată în multe programe guvernamentale ori de afaceri pentru că pot oferi economii semnificative atât în costurile iniţiale de achiziţie cât şi în costurile ulterioare, de întreţinere. Totuşi, specificaţiile software-ului COTS sunt scrise extern, independent, de utilizarea acestui software într-o aplicaţie aparţinând, eventual, unei agenţii guvernamentale.
Acest fapt poate induce rezerve şi temeri deoarece schimbările viitoare ale produsului respectiv nu mai pot fi controlate.
Principalele motive invocate atunci când sunt utilizate produsele COTS vizează:

(1) reducerea globală a timpului de dezvoltare,
(2) costurile de dezvoltare, deoarece componentele pot fi achiziţionate ori licenţiate, în loc să fie dezvoltate pornind de la specificaţii,
(3) costuri mai reduse de întreţinere.

Software-ul conţine, aproape inevitabil, defecte. De aceea se face tot posibilul ca rata de defectare să scadă ori, pentru soluţionarea problemelor erorilor din software, se folosesc tehnici de creştere a toleranţei la defecte.
Teoretic, o metodă foarte eficientă de evitarea erorilor în software este verificarea formală. Dar această metodă nu este aplicabilă modulelor mari, complexe, dintr-o aplicaţie software.

Un pas important, în orice tentativă de tolerare a defectelor, este detectarea acestora. O cale mult utilizată de detectare a defectelor în software este aplicarea testelor de acceptare.
Acestea sunt utilizate în contextul aşa - numitelor *wrappers* (ambalaje) dar şi în blocurile de recuperare.

Aceste teste sunt mecanisme importante în implementarea toleranţei la defecte în software.
Dacă un termometru arată -40^{o} C în miezul unei zile de vară, spre exemplu, termometrul respectiv este presupus că funcţionează eronat. Acesta este un exemplu de *test de acceptare*.

Un test de acceptare este, în esenţă, o verificare a rezonabilităţii. Cele mai multe dintre testele de acceptare, în general, sunt cuprinse într-una din categoriile următoare.

Verificarea duratei de execuţie. Acesta este un alt instrument mult folosit în toleranţa defectelor în software. Dacă există o apreciere cât de cât a duratei rulării codului, se poate programa o întrerupere corespunzătoare intervalului de timp estimat. Atunci când acel timp estimat a fost depăşit se poate presupune că a avut loc o funcţionare necorespunzătoare (o eroare produsă în hardware ori ceva eronat în software care a cauzat depăşirea duratei estimate). Verificările prin durata execuţiei sunt folosite în paralel cu alte teste de acceptare.

Verificarea rezultatului. În anumite cazuri testul de acceptare este cât se poate de natural sugerat chiar de problema în cauză. Adică, natura problemei este de în aşa fel, încât chiar dacă problema este dificil de soluţionat, este mult mai comod de verificat corectitudinea rezultatului deoarece este mult mai puţin probabil ca verificarea, în sine, să fie incorectă.
În acest sens, spre exemplu, soluţionarea unui joc puzzle poate fi de durată, dar verificarea corectitudinii acesteia este trivial de simplă.

Exemple ale unor astfel de probleme în software sunt:

- calculul rădăcinii pătrate (se ridică la pătrat rezultatul şi se verifică obţinerea numărului iniţial),
- factorizarea unor numere mari (se multiplică factorii corespunzători, urmărind obţinerea numerelor respective, dar metoda poate fi de durată),
- soluţionarea ecuaţiilor (se introduc soluţiile în ecuaţiile iniţiale) şi
- ordonarea. De remarcat că în problemele de ordonare este insuficient să se verifice corecta ordonare, pe de-o parte dar trebuie verificat şi faptul că toate numerele supuse ordonării sunt prezente în rezultatul final (şirul ordonat complet de numere).

Adeseori, din raţiuni care ţin de durata testelor, se utilizează verificări probabilistice. Acestea nu garantează detectarea tuturor rezultatelor eronate chiar dacă verificările sunt executate perfect. Un astfel de exemplu de verificare probabilistică, privind corectitudinii înmulţirii matricelor, este descris în cele ce urmează.
Se presupune că sunt înmulţite două matrice A şi B, pătrate de dimensiune n, obţinându-se matricea C.
Verificarea rezultatului, fără repetarea înmulţirii matricelor, utilizează un vector aleatoriu cu n întregi, R, şi efectuează operaţiile $M_1 = A \times (B \times R)$ şi $M_2 = C \times R$. Dacă $M_1 \neq M_2$, atunci a avut loc o eroare.

Dar, dacă $M_1 = M_2$, aceasta nu dovedeşte, totuşi, că rezultatul C este corect.
În acelaşi timp, este *foarte puţin probabil* că vectorul aleatoriu R a fost astfel ales încât $M_1 = M_2$, chiar dacă $A \times B \neq C$. Se poate obţine reducerea, în continuare, a acestei probabilităţi prin alegerea unui alt vector aleatoriu R, tot cu n componente, repetând verificarea. Acest test are complexitatea $O(mn^2)$.

Verificarea ordinului mărimii a rezultatului. Există situaţii când nu sunt la îndemână abordări evidente şi convenabile pentru verificarea corectitudinii rezultatului. Pentru astfel de situaţii sunt folosite majorări ori limite ale rezultatului. În acest sens, se utilizează cunoaşterea naturii aplicaţiei pentru evaluarea unor limite acceptabile ale rezultatului. Un rezultat situat în afara acestor limite justifică declararea rezultatului respectiv ca fiind eronat.
De cele mai multe ori limitele sunt fie valori prestabilite, fie sunt calculabile prin funcţii simple care depind de variabilele în raport cu care se determină rezultatul respectiv.

În cele de-al doilea caz este important ca funcţiile să fie suficient de simplu implementabile astfel încât probabilitatea de rejecţie a testului software însuşi, ca fiind defect, să fie suficient de mică.

Un satelit prevăzut cu sensori de temperatură, spre exemplu, efectuează din spaţiu imagini termice ale suprafeţei pământului. Se pot stabili, evident, limite ale gamei de temperaturi şi se poate considera orice abatere faţă de aceste limite, ca fiind o indicaţie clară a unei malfuncţionări. Chiar mai mult, se pot stabili corelaţii spaţiale, care să repereze diferenţe excesive dintre temperaturile unor arii adiacente şi să se declare eroare dacă diferenţele nu sunt explicabile prin cauze fizice (cum ar fi prezenţa unor vulcani).

Atunci când sunt stabilite limitele pentru acceptarea testelor trebuie să fie consideraţi doi parametri, *sensibilitatea* şi *specificitatea*.

Sensibilitatea este probabilitatea ca testul de acceptare să sesizeze un rezultat eronat. Mai exact, sensibilitatea este probabilitatea condiţionată ca testul să declare o eroare, dat fiind faptul că rezultatul este eronat.

Specificitatea, spre deosebire de sensibilitate, este probabilitatea condiţionată ca atunci când testul de acceptare detectează un rezultat eronat, aceasta este într-adevăr o eroare şi nicidecum un rezultat corect, care se întâmplă să se situeze în afara limitelor testului.

Un parametru strâns legat este probabilitatea unei *false alarme*. Aceasta este probabilitatea condiţionată ca testul să declare eronat un rezultat care, în fapt, este corect. Se poate atinge o creştere a sensibilităţii prin îngustarea intervalului de acceptare. Aceasta ar conduce, în acelaşi timp, din nefericire la micşorarea specificităţii şi creşterea probabilităţii falselor alarme.

Versiunea singulară la toleranţa faţă de defecte

Sunt considerate căi şi metode prin care componente individuale de software pot fi făcute să fie mult mai robuste. Se vor considera, pentru început încapsulările. Acestea sunt interfeţe specifice care îmbunătăţesc interfeţele modulelor software existente.

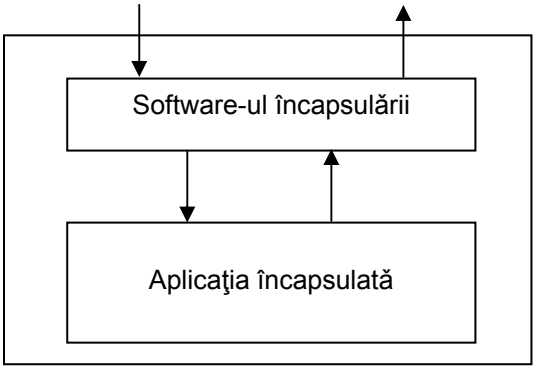
Încapsulările

Acestea sunt structuri software care încapsulează un program atunci când acesta este executat. Se poate efectua încapsularea aproape la orice nivel al software-ului. Exemple curente cuprind software de aplicaţii, *middleware* şi chiar nucleul sistemului de operare. Fluxul intrărilor venind de la mediul extern, lumea din afară, spre entitatea încapsulată sunt interceptate de încapsulare, care decide când să le transmită entităţii încapsulate sau când să semnaleze o excepţie spre sistem.
În mod absolut similar ieşirile entităţii încapsulate sunt de asemenea filtrate prin încapsulare.

Încapsulările au devenit foarte mult uzitate atunci când s-a început utilizarea componentelor software COTS în vederea aplicaţiilor cu înaltă fiabilitate. Componentele COTS sunt scrise pentru aplicaţii de uz general, pentru care erorile sunt supărătoare, în adevăr, dar nu sunt o calamitate.
Înainte ca aceste componente să fie folosite în aplicaţii care impuneau fiabilitate ridicată, acestea trebuiau să fie incluse, cuprinse, într-un anumit mediu software care să le micşoreze rata erorilor.

Acest mediu (încapsularea) trebuia să evite spre aplicaţia încapsulată fluxuri de intrări care sunt fie în afara gamei de valori valide, fie sunt cunoscute deja ca fiind cauzatoare de erori. Similar, încapsularea trece fluxurile de ieşire ale entităţii încapsulate printr-un filtru de acceptare înainte să le trimită spre sistem.

Dacă anumite informaţii, eventual toate ieşirile violează testul de acceptare, acest fapt trebuie transmis sistemului care decide asupra unei acţiuni corespunzătoare.
O încapsulare este caracterizată, specificată, prin entitatea încapsulată dar şi prin sistemul în care se instalează.



Structura generală a unei încapsulări software

Tratarea depăşirii lungimii zonelor tampon de memorie. Limbajul de programare C, tradiţional, nu efectuează întotdeauna verificarea mărimii masivelor, ceea ce poate cauza neajunsuri accidentale ori rău intenţionate. Scrierea unui şir mare de caractere într-o zonă tampon de memorie insuficientă ca mărime, produce o depăşire; deoarece nu se efectuează nici o verificare a lungimii zonelor implicate în transfer, o regiune din memorie, situată în imediata vecinătate a zonei tampon, este suprascrisă. Mai mult, conţinutul zonei suprascrise nu este controlat.

Se consideră, spre exemplu, funcţia *strcpy*() din limbajul C. Aceasta funcţie copiază şiruri de caractere dintr-o locaţie de memorie în alta. Dacă se execută apelul *strcpy*(*str*1, *str*2), unde *str*1 desemnează o zonă tampon de memorie cu lungimea 5 iar *str*2 este adresa unui şir cu lungimea 25, atunci depăşirea zonei de memorie va suprascrie o regiune din memorie în afara zonei de tampon *str*1.
Astfel de depăşiri au fost exploatate în vederea producerii unor agresiuni.

O încapsulare poate verifica şi asigura că astfel de depăşiri de suprascriere nu sunt posibile, spre exemplu, prin verificarea capacităţii zonei de memorie tampon în raport cu şirul desemnat.
Violarea acestei reguli face să se anuleze apelul funcţiei *strcpy*() iar încapsularea returnează o eroare sau ridică o excepţie spre sistem.

Utilizarea unor aplicaţii software având erori cunoscute. Sunt cunoscute situaţii în care o aplicaţie achiziţionată se dovedeşte să aibă defecte. Stabilirea funcţionării defectuoase poate avea loc în urma unor verificări specifice sau ca urmare a unor rapoarte provenite din utilizări

anterioare şi care au dovedit existenţa unor anumite malfuncţionări, corespunzătoare unor anumite seturi de date *S*, etc.
Se poate ca versiunea nouă, corectată şi adusă la zi, urmând să fie lansată de producător să nu fie încă disponibilă, cel puţin un timp.
În consecinţă, trebuie utilizată cea curentă, aşa cum este aceasta. Pentru evitarea defectelor cunoscute această aplicaţie trebuie controlată. Instituirea acestui control se realizează prin încapsulare.

Încapsularea va intercepta intrările de date ale aplicaţiei şi va stabili dacă acestea au intersecţie nevidă cu mulţimea *S* de seturi de date, cunoscute ca fiind cauzatoare a malfuncţionării. În cazul în care datele de intrare ale aplicaţiei sunt din mulţimea *S,* acestea se pot redirecţiona spre o aplicaţie special constituită, capabilă să trateze situaţia ori, în cel mai defavorizat caz, să poată emite o excepţie spre sistem.

Utilizarea unei aplicaţii de încapsulare care să verifice corectitudinea rezultatelor produse. O astfel de aplicaţie de încapsulare cuprinde un test de acceptare prin care se poate filtra orice rezultat produs de aplicaţia verificată. Dacă rezultatul produs de aplicaţia verificată trece de testul respectiv, atunci acel rezultat este transmis mai departe. Dar, dacă rezultatul nu satisface testul de acceptare atunci se generează o excepţie iar sistemul are de soluţionat un caz suspect.
O alternativă la generarea unei excepţii, este implementarea în aplicaţia de încapsulare a unui modul specific de tratare a situaţiei generate în urma ne-satisfacerii testului de acceptare. Acest modul specific poate stabili rezultatul corect şi îl poate substitui celui produs în aplicaţia verificată.

Capacitatea încapsulării cu succes a unei aplicaţii existente depinde de anumiţi factori:

(a) Calitatea testelor de acceptare. Acest factor este intens dependent de aplicaţia respectivă şi are un impact imediat asupra abilităţii aplicaţiei încapsulatoare să blocheze ieşirile eronate produse de aplicaţia încapsulată.
(b) Disponibilitatea informaţiilor necesare, proprii aplicaţiei încapsulate. Adesea componenta încapsulată este tratată ca fiind o *cutie neagră* şi tot ce se poate observa în legătură cu comportamentul acesteia sunt valorile produse la ieşire, ca răspuns la anumite valori ale intrărilor. În astfel de situaţii încapsularea va fi cumva limitată.
(c) Măsura în care aplicaţia încapsulată a fost deja testată. Testarea extensivă a acestei aplicaţii permite identificarea regiunilor din spaţiul valorilor de intrare pentru care aplicaţia nu funcţionează corect şi implicit diminuarea probabilităţii propagării în sistem a valorilor eronate.

Se consideră, spre exemplu, verificarea corectitudinii unui modul planificator de sarcini (*scheduler*). Pentru aceasta se consideră o încapsulare a planificatorului de sarcini dintr-un sistem în timp real, tolerant la defecte.
Spre deosebire de sistemele de operare de uz general, planificatoarele sistemelor în timp real nu utilizează o planificare de tip *R-R* (*round-robin*).

Un algoritm de planificare pentru planificatoarele de sarcini în timp real este de tip *EDF* (*EDF* fiind abrevierea denumirii *Earliest Deadline First*) în care, aşa cum rezultă din denumire, sistemul execută prioritar sarcina cu cel mai strâns, apropiat, termen de execuţie dintre toate sarcinile care sunt gata să fie lansate în execuţie.

Un astfel de algoritm trebuie să ia în consideraţie anumite restricţii privind întreruperea execuţiei unei sarcini înainte de terminarea execuţiei acesteia, deoarece anumite sarcini nu se pot întrerupe pe durata unor anumite porţiuni, dinainte cunoscute, ale execuţiei acestora.

Un astfel de planificator poate fi supus unei încapsulări printr-o aplicaţie care verifică modul corect de execuţie al planificatorului:

- planificatorul alege, dintr-o mulţime existentă, mereu sarcina gata de execuţie caracterizată prin termenul de execuţie cel apropiat.
- apariţia, la un moment dat, a unei noi sarcini executabile având termen mai scurt decât sarcina executată curent trebuie să conducă la lansarea acesteia în execuţie doar dacă execuţia sarcinii curente poate fi întreruptă.

Este evident că pentru soluţionarea verificării funcţionării corecte a planificatorului EDF, aplicaţia încapsulatoare trebuie să aibă acces la anumite informaţii esenţiale:

- mulţimea sarcinilor gata de execuţie,
- termenele de execuţie ale acestora şi
- informaţia privind posibilitatea întreruperii înainte de termen a sarcinii executate curent.

Obţinerea acestor informaţii impune, în fapt, accesul printr-o interfaţă corespunzătoare a respectivului planificator al sarcinilor în timp real, privitor la starea sarcinilor gata de lansat în execuţie ca şi asupra termenelor lor de execuţie, pe de-o parte, dar şi asupra momentului când sarcina aflată curent în execuţie este interuptibilă înainte de termen.

Reîntinerirea software-ului

Sunt multe situaţii în care un utilizator se confruntă cu situaţii în care calculatorul se blochează. Reacţia firească este adesea reîncărcarea sistemului de operare. Această operaţie este cea mai simplă cale de reîntinerire a software-ului.
Pe măsură ce se execută un proces de calcul acesta ocupă memorie şi acaparează fişiere fără să elibereze corespunzător aceste resurse. În paralel datele unui astfel de proces tind să fie corupte şi să se acumuleze erori necorectate. Un astfel de proces tinde să consume, în egală măsură, fire de execuţie şi semafoare fără să le elibereze. Dacă un asemenea proces continuă să ruleze indefinit, va deveni defect şi va stopa execuţia. Pentru evitarea unei astfel de situaţii se poate proceda să se stopeze, proactiv, procesul. Astfel, starea internă a acestui proces este re-iniţializată şi se poate demara reluarea respectivului proces. Aceasta este ceea ce se numeşte o reîntinerire a software-ului.

Nivelul reîntineririi

Se poate aplica un procedeu de reîntinerire fie la nivelul aplicaţiei, fie la nivelul procesorului respectiv.
Reîntinerirea practicată la nivelul unei aplicaţii constă din suspendarea individuală a respectivei aplicaţii, ştergerea stării acesteia, reiniţializarea structurilor de date ale acesteia ş.a.m.d., după care se relansează în execuţie respectiva aplicaţie.
Reîntinerirea aplicată la nivelul procesorului constă din re-încărcarea sistemului de operare şi afectează toate aplicaţiile care rulau pe respectivul procesor. Dacă situaţia se petrece la nivelul unui cluster de procesoare este de dorit ca să se mărginească reîntinerirea astfel încât o atare reîntinerire să afecteze doar o mica parte a procesoarelor la un moment dat.

Modul de aplicare al reîntineririi

Reîntinerirea software-ului se poate determina în raport cu timpul sau se poate stabili prin predicţie. Reîntinerirea software-ului în raport cu timpul constă în acţiuni de reîntinerire la intervale constante de timp.
Determinarea perioadei optime de inter-reîntinerire trebuie să echilibreze beneficiile în raport cu costurile operaţiei de reîntinerire.
Se poate constitui un model matematic simplu al modului de aplicare periodică a reîntineririi software-ului.
Se vor utiliza câteva notaţii:

$N(t)$ – numărul mediu de erori dintr-un interval de timp t, fără efectuarea vreunei reîntineriri.
C_e – costul fiecărei erori,
C_r – costul fiecărei operaţii de reîntinerire,
P – perioada de timp scursă între două reîntineriri succesive.

Prin sumarea costurilor datorită erorilor şi aferente operaţiilor de reîntinerire se determină expresia costului reîntineririi aferente unei perioade de timp P, notat prin $C_{re}(P)$:

$$C_{re}(P) = N(P) \cdot C_e + C_r$$

Prin raportarea acestui cost la timpul de aplicare a unei reîntineriri, se obţine costul (în raport cu o unitate de timp) unei operaţii de reîntinerire.
Acest cost este notat prin $C_{rata}(P)$ şi are expresia:

$$C_{rata}(P) = \frac{C_{re}(P)}{P} = \frac{N(P)C_e + C_r}{P} \qquad (1)$$

Este util de considerat diferite cazuri asupra modului cum evoluează numărul mediu de erori între două operaţii de reîntinerire.
Întâi se poate considera că viteza de defectare a software-ului λ este constantă pe toată durata de execuţie a software-ului.
Această ipoteză conduce la o expresie a numărului mediu de erori, acumulate după un interval P de timp, de forma:

$$N(P) = \lambda P.$$

Substituind această expresie pentru $N(P)$ în (1) se obţine:

$$C_{rata}(P) = \lambda C_e + C_r / P\,.$$

Se poate uşor remarca faptul că se poate minimiza această expresie, dedusă, pentru $C_{rata}(P)$ dacă P tinde spre infinit. Concluzia se poate formula în termenii următori:

- Atunci când viteza de defectare a software-ului este constantă în timp, este rentabil să se evite reîntinerirea software-ului.
- Reîntinerirea software-ului este profitabilă atunci când viteza de apariţie a erorilor creşte pe măsură ce este executat software-ul respectiv.

Următoarea ipoteză asupra vitezei de defectare a software-lui consideră că numărul mediu de defecte care apar într-un interval de timp P este de forma:

$$N(P) = \lambda P^2.$$

Din expresia (1) cu $N(P) = \lambda P^2$ rezultă că:

$$C_{rata}(P) = \lambda P C_e + C_r / P$$

Minimizarea expresiei astfel obţinute se va determina pentru:

$$\frac{dC_{rata}(P)}{dP} = 0,$$

Cu condiţia:

$$\frac{d^2 C_{rata}(P)}{dP^2} > 0.$$

Prin soluţionarea condiţiilor anterior menţionate se obţine o durată optimă, notată prin P^*, de forma:

$$P^* = \sqrt{\frac{C_r}{\lambda C_e}}.$$

În cel de-al treilea caz se consideră că $N(P) = \lambda P^n$, $n > 1$.
Acest caz este o generalizarea a cazului precedent. Astfel, în acest caz se obţine, pentru $C_{rata}(P)$, expresia:

$$C_{rata}(P) = \lambda P^{n-1} C_e + C_r / P .$$

Printr-un calcul similar celui aplicat în cazul anterior, se stabileşte că perioada optimă de reîntinerire are expresia:

$$P^* = \left(\frac{C_r}{(n-1)\lambda C_e}\right)^{1/n}$$

Pentru alegerea corespunzătoare a valorii perioadei P este necesară cunoaşterea valorilor parametrilor C_r/C_e şi $N(t)$.
Aceşti parametrii se pot obţine experimental prin rularea unor simulări ori alternativ, sistemul poate fi proiectat adaptiv, având anumite valori iniţiale implicite, alese pentru început.
În timp, pe măsură ce se culeg date şi se determină statistici, care reflectă natura căderilor software-ului, se poate ajusta corespunzător perioada de reîntinerire.

Reîntinerirea bazată pe predicţii cuprinde monitorizarea caracteristicilor sistemului (cum ar fi volumul de memorie alocat, numărul de fişiere etc.) şi predicţiile privitor la condiţiile în care au loc căderile sistemului. Dacă un proces, spre exemplu, este intens consumator de memorie, cu o anumită viteză, atunci sistemul va putea estima destul de precis când va avea loc depăşirea volumului de memorie disponibilă. Reîntinerirea software-ului, în acest caz, va avea loc chiar înaintea momentului predeterminat al epuizării memoriei disponibile.

Software-ul care implementează reîntinerirea bazată pe predicţii trebuie să poată să aibă acces la suficient de multe informaţii de stare ale sistemului pentru determinarea unor predicţii fiabile. Atunci când aplicaţia care gestionează reîntinerirea software-ului face parte din sistemul de operare, este satisfăcut accesul la informaţiile vitale pentru stabilirea unor predicţii satisfăcătore.
Dar dacă această aplicaţie rulează peste sistemul de operare, fără să aibă privilegii speciale, atunci aceasta va fi constrânsă să utilizeze orice interfaţă este oferită de sistemul de operare pentru ca să colecteze informaţii privitor la starea sistemului.

Sistemul de operare Linux, spre exemplu, oferă anumite facilităţi de acces la informaţiile de sistem prin intermediul unor utilităţi cum ar fi:
vmstat – oferă informaţii despre utilizarea procesorului, memoriei şi activităţii în pagini, întreruperi şi operaţii de I/E.
iostat – produce utilizarea procentuală a UC, la nivelurile de utilizator şi sistem, precum şi un raport asupra utilizării fiecărui dispozitiv de I/E.
netstat – arată conexiunile în reţea, tabelele de rutare şi tabelul tuturor interfeţelor de reţea.
nfsstat – oferă statistici despre clientul NFS şi activităţile NFS.

Odată ce s-au colectat informaţiile corespunzătoare stării sistemului, se pot identifica tendinţele şi se pot face predicţii privitor la momentul în care aceste tendinţe vor produce erorile de sistem. Dacă se cunoaşte modul în care a fost cerută şi alocată memorie, unui anumit proces, de-a lungul timpului, se poate determina o aproximare polinomială (eventual utilizând metoda celor mai mici pătrate) a alocărilor de memorie dintr-o anumită fereastră, interval, de timp scurs recent.
Cea mai simplă abordare, în acest sens, este potrivirea peste punctele de alocare a unei linii drepte, echivalentă unui polinom de gradul întâi, de forma: $f(t) = mt + c$.
Aproximări mai complexe pot utiliza un polinom de grad neprecizat, de gradul n, spre exemplu. Fereastra de timp se presupune că include ultimele k momente de timp:

$$t_1 < t_2 < \ldots t_k,$$

unde t_k este cel mai recent moment.
Se presupun cunoscute valorilor corespunzătoare celor k momente de timp:

$$\mu(t_1), \mu(t_2), \ldots \mu(t_k).$$

Unde $\mu(t_i)$, $1 \le i \le k$, reprezintă volumul de memorie alocată la momentul t_i.

Se caută să se determine coeficienţii polinomului:

$$f(t) = m_n t^n + m_{n-1} t^{n-1} + \ldots m_1 t + m_0,$$

astfel încât să se minimizeze expresia:

$$\sum_{i=1}^{k} [\mu(t_i) - f(t_i)]^2 .$$

Acest polinom poate fi utilizat pentru extrapolarea funcţiei de alocare a memoriei şi să se poată prognoza momentul când procesul va epuiza memoria disponibilă.
În aproximarea prin cele mai mici pătrate fiecare punct observat $\mu(t_i)$ are aceeaşi pondere în determinarea funcţiei de aproximare. O variantă a acestei metode este metoda *ponderată* a

celor mai mici pătrate, în care se caută să se minimizeze suma ponderată a pătratelor. Pentru problema aproximării cererilor de memorie se vor alege anumite ponderi:

$$w_1, w_2, \ldots, w_k,$$

urmând să se determine coeficienţii funcţiei $f(t)$ astfel încât să se minimizeze expresia:

$$\sum_{i=1}^{k} w_i[\mu(t_i) - f(t_i)]^2 .$$

Introducerea ponderilor face posibilă introducerea unei anumite discriminări între anumite puncte din domeniul de definiţie al funcţiei f.
Astfel, dacă $w_1 < w_2 < \ldots < w_k$, atunci datele mai recente vor influenţa procesul de potrivire mai mult decât cele anterioare.

Abordarea problemei predicţiei prin metoda de aproximare a celor mai mici pătrate este vulnerabilă la impactul câtorva puncte cu valori disparate (valori mult în afara celorlalte, mult mai mari ori mult mai mici) care pot distorsiona potrivirea căutată.
Pentru astfel de situaţii sunt utilizate tehnici care să favorizeze o potrivire mai robustă, prin reducerea impactului unor astfel de puncte cu valori disparate.

Abordările combinate sunt adeseori utilizate. Astfel, reîntinerirea software-ului poate avea loc:

- fie la momentul planificat P,
- fie corespunzător momentului pentru care s-a făcut predicţia producerii următoarei erori.

Este ales acel moment care îl precede pe celălalt.

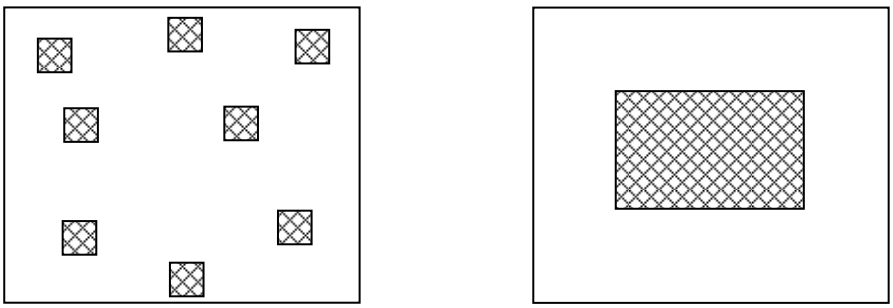


Figura 1. Regiuni de eşec.
(a) regiuni de eşec mici, dispersate. (b) regiunea de eşec mare, compactă.

Diversitatea datelor

Spaţiul intrărilor unui program este spaţiul ce cuprinde toate intrările posibile. Acest spaţiu poate fi divizat în regiuni de *eşec* şi regiuni de *non-eşec*. Un program eşuează dacă şi numai dacă intrările sale aparţin unor regiuni de eşec.
Regiunile de eşec pot fi determinate în orice formă geometrică şi mărime. Spaţiul intrărilor poate avea, tipic, un număr mare de dimensiuni, dar poate fi vizualizat doar într-un nerealist

de simplu spaţiu bidimensional. Figura 1 prezintă două reprezentări arbitrare ale unor spaţii de eşec. Regiunile de eşec, reprezentate în figura 1 prin suprafeţele haşurate, ocupă, în ambele cazuri, aproximativ aceeaşi arie din spaţiul intrărilor. Dar, regiunea de eşec în figura 1 (a) constă dintr-un număr de opt zone insulare relativ mici, în timp ce regiunea de eşec din figura 1 (b) apare reprezentată printr-o arie compactă şi relativ întinsă.
În ambele cazuri, figura 1 (a) sau (b), dacă intrările sistemului sunt din zona de eşec acesta va eşua. Diferenţa esenţială dintre figura 1 (a) şi figura 1 (b) constă în faptul că o mică perturbaţie a valorilor intrărilor din figura 1 (a) poate face ca valorile liniilor de intrare să părăsească regiunea de eşec şi să treacă în regiunea de non-eşec.

Defectări de felul celor reprezentate prin figura 1 (a) sugerează o posibilă abordare a toleranţei defectelor. Dacă se consideră că intrările sistemului sunt dintr-una din zonele de eşec, atunci o mică perturbaţie a intrărilor sistemului este posibil să scoată intrările acestuia din aria de eşec şi să le treacă în zona non-eşec. Această abordare generică se numeşte, în literatură, *diversitatea datelor*. Modul în care se implementează aceasta depinde de mecanismele de detecţie a erorilor.
Dacă se rulează doar o singură copie a software-ului la un moment dat şi se utilizează un test de acceptare pentru detectarea erorilor, atunci se poate recalcula acelaşi rezultat cu intrări perturbate şi verifica rezultatul astfel determinat.
Dacă se utilizează un sistem masiv redundant, se pot aplica seturi de intrări uşor diferite diferitelor versiuni ale programului şi urmărit rezultatul la circuitul de votare. Perturbări ale datelor de intrare se pot face atât implicit cât şi explicit.