Verificarea automată a programelor

Mihai Budiu — mailto:mihaib+@cs.cmu.edu
http://www.cs.cmu.edu/~mihaib/

octombrie 2001

Subiect:

Cunoştinţe necesare:

Cuvinte cheie:

Cuprins

Pe data de 8 august 2000 pe lista de e-mail numită "Linux kernel", care este folosită de către cei care meşteresc la nucleul sistemului de operare Linux pentru a comunica între ei, a apărut un mesaj care descria o serie de potenţiale probleme (bug-uri) din nucleu, trimis de Dawson Engler, un proaspăt profesor la universitatea Stanford. Pe această listă se vehiculează adesea rapoarte de defecţiuni, aşa că unul în plus nu pare ceva deosebit. Mesajul cu pricina însă conţinea nu una, două, sau chiar zece erori, ci peste 160! Şi mai interesant, toate aceste potenţiale erori au fost descoperite automat, folosind un program care studiază codul şi caută inadvertenţe. În final nu toate rapoartele s-au dovedit a fi defecţiuni, dar multe dintre ele erau într-adevăr bug-uri importante.

Acest mesaj de poştă electronică este probabil cea mai "vizibilă" aplicare a unei tehnologii încă relativ imature, dar care face progrese remarcabile, numită *verificare formală*. Verificarea formală este folosită pe scară relativ largă, dar mai ales de către proiectanţii de hardware; multe din circuitele procesoarelor moderne au fost certificate ca fiind corecte folosind verificarea formală. O cercetare febrilă se desfăşoară pentru a adapta metodele dezvoltate pentru hardware şi la analiza programelor; software-ul are însă o natură substanţial diferită de a hardware-ului, şi probabil va fi nevoie de inventarea unor tehnici noi.

În acest articol voi descrie pe scurt metodele folosite de echipa de la Stanford condusă de Dawson Engler pentru a descoperi erori în programe, şi unele din rezultatele lor foarte interesante despre natura erorilor în nucleul Linux.

1 Bug-uri în Linux

Un aforism spune că orice program are cel puţin un bug şi este cu cel puţin o linie prea lung. Aplicînd această regulă în mod repetat putem deduce că orice program poate fi optimizat pînă este redus la o singură linie de cod, care este eronată.

Chiar dacă sistemul de operare Linux este lăudat pentru robusteţea sa deosebită, nu este scutit de imperfecţiuni. Profitînd de disponibilitatea surselor C ale programului, cercetătorii de la Stanford au testat o nouă metodologie pentru a descoperi bug-uri pe 21 de versiuni diferite ale nucleului, începînd de la versiunea 1.0, din ianuarie 1994, pînă la versiunea 2.4.7, din primăvara acestui an. Trebuie menţionat dintru început că sistemul de la Stanford nu descoperă toate bug-urile, ci numai anumite tipuri de bug-uri, relativ simple; voi discuta metodologia folosită ceva mai încolo. Vom considera că bug-urile descoperite de acest sistem sunt reprezentative pentru toate bug-urile în general, cel puţin în anumite aspecte ale distribuţiilor statistice. E de reţiunt deci că rezultatele pe care le prezint în continuare se referă la un subset al tuturor bug-urilor, care poate nu este reprezentativ.

Codul Linux e structurat din mai multe părţi relativ independente; putem factoriza codul însă în două mari clase: nucleul propriu-zis, şi driverele. Driverele sunt mici programe care fac parte din nucleu şi care interfaţează nucleul cu dispozitivele periferice. Datorită popularităţii PC-ului, există literalmente mii de periferice diferite, fiecare avînd nevoie de un driver separat[1]. Ca atare, majoritatea codului Linux se află în drivere. Cantitatea de cod din drivere variază între 50 şi 70 la sută din întregul sistem.

Între versiunile 1.0 şi 2.4.1 codul sistemului Linux a crescut de aproximativ 16 ori, de la 105 mii de linii de cod la 1,6 milioane[2] Cel mai mult au crescut driverele, datorită faptului că din ce în ce mai mulţi fabricanţi de hardware au publicat informaţii despre periferice sau chiar au contribuit cu cod. Alte subsisteme care au crescut sensibil între timp sunt sistemele de fişiere şi codul pentru protocoalele de reţea.

În total, în cele 21 de versiuni de nucleu sistemul de la Stanford a detectat peste 1000 de bug-uri distincte. (Unele dintre bug-uri se manifestă în mai multe versiuni diferite de nucleu.) Numărul de bug-uri este practic direct proporţional cu mărimea codului: nucleele mai mari conţin mai multe bug-uri.

Bug-urile sunt însă departe de a fi uniform distribuite. După cum ne aşteptăm, cele mai multe bug-uri sunt în drivere, pentru că driverele ocupă atît de mult din cod. Putem normaliza numărul de bug-uri în două feluri: împărţim la numărul de linii, pentru a vedea la cîte linii avem un bug, sau putem împărţi la numărul de locuri în care acelaşi tip de bug s-ar fi putut manifesta. De

[1] Desigur, dacă două periferice sunt compatibile între ele, pot folosi acelaşi driver, dar compatibilitatea pare să fie mai curînd excepţia decît regula.

[2] Linux este cel mai portabil sistem de operare, rulînd pe zeci de procesoare diferite. Informaţiile din textul de faţă sunt însă specifice versiunii de Linux pentru procesoarele x86, de pe PC-uri.

exemplu, dacă un bug care constă în lipsa verificării unui cod de eroare după apelul unei funcţii care poate eşua, densitatea de bug-uri este numărul de locuri în care nu verificăm eroarea împărţit la numărul de locuri în care apelăm funcţii care pot returna erori.

Chiar dacă normalizăm numărul de bug-uri, driverele stau însă destul de rău: densitatea de bug-uri în drivere este între 3 şi 7 ori mai mare decît în nucleul propriu-zis. Driverele sunt folosite doar de cei care au perifericele corespunzătoare, spre deosebire de nucleu, care e folosit practic de toată lumea. Ca atare, driverele sunt mai puţin testate, deci este de aşteptăt să aibă mai multe bug-uri.

Înţelepciunea populară spune că cu cît un program e mai "bătrîn", cu atît are mai puţine bug-uri; acest lucru este verificat şi de datele despre Linux: fişierele cele mai vechi au cele mai puţine bug-uri. La nivel de funcţie, funcţiile mai lungi tind să aibă o densitate mai mare de bug-uri, ceea ce confirmă regula că programul trebuie spart în bucăţele mici şi independente, pentru a stăvili complexitatea.

Dacă facem un grafic al distribuţiei erorilor în raport cu fişierele în care apar, graficul este foarte neuniform: cele mai multe fişiere au cam 1 bug, pe cînd cîteva fişiere sunt responsabile pentru majoritatea bug-urilor. Foarte adesea, programatorii au copiat un fragment de program eronat în mai multe locuri, multiplicînd apariţia unui singur bug.

Privind la versiuni succesive ale sistemului de operare, se poate observa în care versiune apare un bug şi în care dispare. Folosind aceste date, se poate calcula durata medie de viaţă a unui bug în cod. În pofida numărului mare de persoane care contribuie la Linux, durata medie de viaţă este relativ ridicată: un bug supravieţuieşte cam un an şi opt luni înainte de a fi corectat.

O ultimă statistică pe care o voi prezenta compară densitatea de bug-uri din Linux cu cea dintr-un alt sistem de operare dezvoltat open-source: OpenBSD. Programatorii sistemului OpenBSD sunt foarte mîndri de faptul că periodic inspectează manual întregul cod, cu scopul de a detecta erori (în engleză această procedură se numeşte *code audit*). Ca atare, ei spun că OpenBSD este cel mai sigur sistem de operare. În mod surprinzător, densitatea de bug-uri în Linux este mai mică, fiind undeva între 1,2 şi 6 ori (cercetătorii au calculat cîte un raport pentru fiecare tip de bug pe care l-au analizat, de aceea prezintă mai multe numere diferite). Acest lucru demonstrează că într-adevăr o comunitate mai mare de programatori este mai eficace în a detecta probleme (Linux este mai popular decît OpenBSD) .

2 Testare şi verificare

Care este metodologia prin care aceste bug-uri au fost depistate? Cum au putut cercetătorii să găsească probleme în bucăţi de cod pe care nici măcar nu le pot executa (de exemplu, este limpede că nu pot rula toate driverele, pentru că nu au la-ndemînă toate perifericele posibile)?

Există două metode diferite de a depana un program: metode dinamice şi metode statice.

Testarea este o metodă dinamică: programul este pornit, i se dau nişte date de intrare, şi comportarea lui este observată; eventual se poate folosi un debugger pentru a urmări evoluţia programului în timp şi pentru a-i putea inspecta măruntaiele. Testarea este relativ uşor de înţeles şi folosit, dar are mai multe probleme, mai ales în contextul depanării unui sistem de operare:

- Unele bug-uri se vor manifesta numai pentru anumite date de intrare; ca atare, reuşita unui test nu garantează că programul este corect;

- Am menţionat deja că driverele pentru perifericele care nu sunt instalate pe un sistem nu pot fi executate;
- În general, sistemul de operare nu operează asupra unui simplu se de date, ci manipulează programe, pachete de date care circulă în reţea, întreruperi de la hardware şi alte evenimente care sunt greu de controlat şi reprodus.

Ca atare, grupul de la Stanford foloseşte metode statice. Metodele statice constau în analiza unui program înainte de a fi lansat în execuţie, independent de datele de intrare. Cea mai familiară dintre aceste metode este chiar compilarea: un compilator modern verifică tot felul de proprietăţi ale programului, şi rejectează programele care nu respectă criteriile de corectitudine. Alteori compilatorul poate da avertismente asupra unor construcţii care pot genera probleme la execuţie, cum ar fi de pildă variabile neiniţializate.

Metodele statice sunt extrem de puternice: ele pot raţiona despre comportarea unui program *oricare* ar fi datele sale de intrare. Compilatoarele de astăzi efectuează analize extrem de sofisticate asupra codului, pe care apoi îl optimizează. Multe dintre optimizări pot fi aplicate numai în anumite condiţii; de exemplu, o atribuire poate fi eliminată dacă rezultatul ei nu mai este folosit în restul programului. Pentru a demonstra că atribuirea este inutilă, compilatorul analizează codul şi demonstrează că, orice cărare prin program va urma execuţia, rezultatul atribuirii nu mai este folosit.

În general astfel de demonstraţii nu pot fi făcute: teoria calculabilităţii arată că chestiuni de genul acesta sunt *nedecidabile*. De aceea, compilatoarele sunt conservatoare: aplică optimizări numai cînd sunt absolut sigure că sunt corecte, şi cîteodată nu aplică optimizări corecte pentru că nu pot demonstra legalitatea lor. Cît de greu este de demonstrat o proprietate, depinde de mulţi factori, dar în primul rînd de limbajul de programare care este compilat. La ora actuală se desfăşoară o cercetare extrem de febrilă în domeniul limbajelor de programare, scopul căreia este proiectarea unor limbaje mai restrictive. În astfel de limbaje anumite tipuri de erori devin imposibile; limbajele trebuie să rămînă însă suficient de flexibile pentru a exprima în mod natural calcule arbitrar de complicate.

3 Proiectul "Meta-Compilation" (MC)

Pentru că limbajul C în care este scris nucleul de Linux este un limbaj relativ primitiv, cercetătorii de la Stanford au făcut un compromis: au extins compilatorul de C cu informaţii specifice nucleului Linux. Această metodologie dă şi numele proiectului: meta-compilare.

Un exemplu va clarifica cel mai bine tehnica folosită.

Pentru a preveni accesele concurente la unele structuri de date importante, nucleele folosesc *regiuni critice*. Intrarea unui program într-o regiune critică este permisă numai după achiziţionarea unei încuietori (lock). Un singur program poate obţine o încuietoare, care-i permite accesul în regiunea critică; la terminarea regiunii critice, programul eliberează încuietoarea. Dacă un program accesează încuietoarea cînd un altul este în regiunea critică, este blocat pînă cînd posesorul încuietorii iese din regiune şi eliberează încuietoarea.

De exemplu, funcţiile `lock_kernel()` şi `unlock_kernel()` încuie şi respectiv descuie anumite porţiuni din nucleu. Aceste operaţii sunt implementate în două funcţii obişnuite. Dar felul în care aceste funcţii sunt folosite este foarte specific: de exemplu, de fiecare dată cînd se foloseşte `lock`, trebuie să urmeze eventual `unlock` (altfel nimeni nu mai poate intra în regiunea critică, şi calculatorul probabil se va bloca). De asemenea, nimeni nu poate face de două ori la rînd `lock`, fără a face între timp şi un `unlock`: a doua oară programul s-ar bloca din cauză că se află deja în regiunea critică.

Metacompilarea exploatează acest gen de informaţie, care nu ţine de limbajul de programare, ci de *aplicaţia* care este programată. Proiectul MC le permite programatorilor să scrie extensii ale compilatorului care încorporează astfel de informaţii.

MC foloseşte compilatorul de C numit gcc (Gnu C Compiler), care este compilatorul folosit în mod tradiţional pentru a compila nucleul Linux; acest compilator este disponibil, ca şi Linux, cu codul său sursă. Cercetătorii de la Stanford au definit un limbaj foarte simplu, pe care l-au numit Metal. Compilatorul extins cu Metal poartă numele de xgcc (eXtended gcc). Iată un exemplu de program complet scris în Metal, care verifică folosirea corectă a instrucţiunilor de încuiere:

```
{ #include "linux-includes.h" }

// descriem un automat finit: state machine
sm verifica_lock {
   // Pattern-uri de cautat in cod
   pat incuie = { lock_kernel(); }
   pat descuie = { unlock_kernel(); }

   // stari; prima stare e starea initiala
   descuiat: incuie ==> incuiat
           | descuie ==> { err("descuiere dubla"); }
           ;

   incuiat: descuie ==> descuiat
          | incuie ==> { err("incuiere dubla"); }
            // acest pattern indica sfirsitul functiei
          | $end_of_path$ ==> { err("functia se termina incuiata"); }
          ;
}
```

Programele Metal sunt compilate şi transformate în programe C şi apoi executabile, care sunt apoi legate dinamic de compilatorul gcc. Cînd xgcc este executat pentru a compila nucleul, extensiile scrise în Metal sunt executate una cîte una. Fiecare extensie analizează fiecare funcţie compilată şi o verifică.

Programele Metal descriu *automate finite*. Programul de mai sus descrie automatul din figura 1.

Acest automat finit este executat în mod abstract de către compilator pe program, pe fiecare procedură în mod separat. În mod conceptual, compilatorul execută fiecare cărare posibilă din procedură şi face tranziţiile indicate de automat cînd întîlneşte pattern-urile indicate. Dacă vreuna din cărări cauzează intrarea în starea de eroare, xgcc generează un mesaj de eroare.

Observaţi că acest automat ignoră cele mai multe operaţiuni din program, şi este interesat numai de apelurile funcţiilor `lock_kernel()` şi `unlock_kernel()`.

Să ilustrăm funcţionarea automatului pe o procedură fictivă, al cărei schelet este următorul:

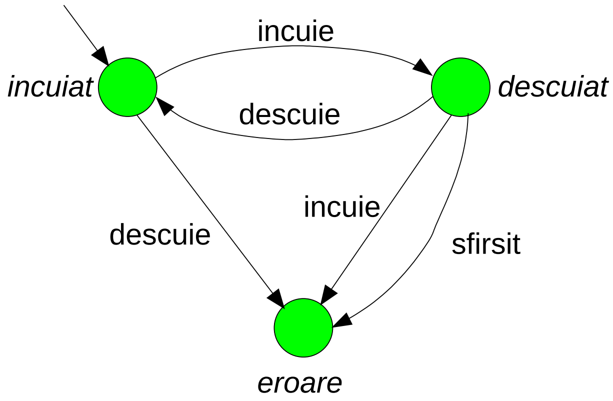

Figura 1: *Automatul finit care testează folosirea corectă a instrucţiunilor de încuiere şi descuiere. Două încuieri succesive, două descuieri succesive, sau terminarea procedurii în starea "încuiat" cauzează o eroare.*

```
f() {
   lock_kernel();
   ...
   p = malloc(10);
   if (!p)
       return OUT_OF_MEMORY;
   ...
   unlock_kernel();
   return OK;
}
```

Această procedură obţine încuietoarea după care încearcă să aloce 10 octeţi. Dacă alocarea eşuează, returnează un cod de eroare. Altfel procedura face tot felul de calcule, descuie nucleul, şi returnează un cod OK.

Execuţia poate urma două cărări prin această procedură, ilustrate în figura 2: amîndouă intră pe la început, dar una iese în caz de eroare din procedură, pe cînd a doua descuie lacătul înainte de a ieşi. Cu litere cursive în figura 2 am indicat starea curentă a automatului finit cînd parcurge cărările respective. Observaţi că pe cărarea roşie automatul ajunge la sfîrşitul procedurii şi se află în starea "încuiat"; ultima regulă din programul Metal va semnaliza în acest caz o eroare.

În realitate xgcc nu parcurge toate cărările posibile, ci calculează simultan toate stările în care se poate afla automatul în fiecare punct din program folosind o metodă numită *interpretare abstractă*. Sper să pot consacra un articol special interpretării abstracte şi altor tehnici de verificare formală, aşa că nu voi mai discuta aici despre ea.

Toate analizele descrise în Metal sunt strict intraprocedurale, adică analizează fiecare funcţie separat[3].

Aceasta este şi una dintre limitările cele mai mari ale acestei scheme. Vom ilustra falibilitatea

[3]Autorii au introdus şi un mecanism rudimentar prin care se pot efectua unele analize interprocedurale, facînd două treceri asupra programului, în prima trecere colectînd informaţii despre toate procedurile vizitate.

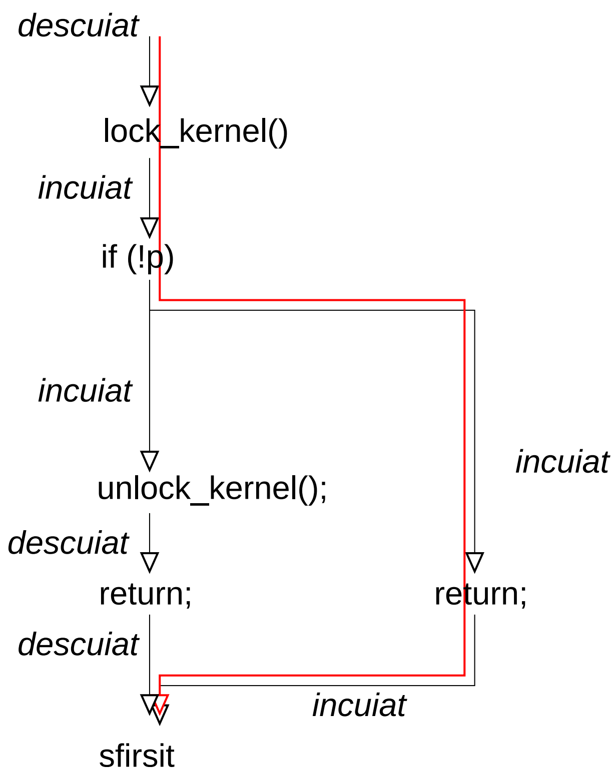

Figura 2: *Traiectoriile posibile ale programului printr-o procedură. Cărarea indicată cu roşu va fi semnalată ca o eroare de programul Metal, pentru că execuţia procedurii se termină într-o stare încuiată.*

ei printr-un alt exemplu. Să presupunem că funcţia `f()` de mai sus este apelată în următorul fragment de cod:

```
...
eroare = f();
if (eroare == OUT_OF_MEMORY)
     unlock_kernel();
...
```

Atunci de fapt codul funcţiei `f` nu era eronat, pentru că pînă la urmă ambele cărări duc la o descuiere.

Astfel, analizele descrise în Metal sunt *incomplete*, şi pot genera alarme false, numite *falsuri pozitive*. (Un fals negativ este o eroare reală care nu este semnalată). xgcc poate fi folosit cu succes dacă cei care programează folosesc o disciplină şi un stil de programare consistent. De exemplu, în general programatorii vor considera codul funcţie `f` de mai sus ca fiind defectuos, pentru că este asimetric: uneori descuiem în `f()`, alteori în apelantul lui `f`.

În proiecte atît de mari ca Linux, disciplina în programare este absolut esenţială. Contributorilor le este bine-cunoscută atitudinea cvasi-dictatorială a lui Linus Torvalds în această privinţă: el va rejecta programe perfect funcţionale dacă nu este mulţumit cu stilul de programare. Cu siguranţă că această disciplină de fier contribuie în mod substanţial la calitatea ridicată a nucleului Linux.

Cele aproximativ 1000 de bug-uri menţionate mai sus au fost detectate folosind 8 programe Metal, toate în jurul a 100 de linii de cod fiecare. Alte extensii scrise pentru xgcc au avut rezultate mixte: deşi generau erori, multe dintre ele erau falsuri pozitive. Din păcate, fiecare mesaj trebuie să fie inspectat manual pentru a decide dacă este o eroare adevărată sau nu. Ca atare, extensiile Metal sunt utile mai ales dacă generează relativ puţine alarme false, altfel efortul necesar pentru verificarea mesajelor devine repede prohibitiv.

4 Inferenţa automată a regulilor de testat

În mod interesant, nici unul dintre programatorii de la Stanford din proiectul MC nu este expert în nucleul Linux. Ca atare, pentru a valida mesajele xgcc, ei trebuia sa scruteze codul pentru a înţelege ce se întîmplă de fapt. Din această activitate le-a venit o idee foarte interesantă, care este subiectul unuia din cele mai recent articole pe care le-au scris, şi care vor fi prezentate în luna octombrie la conferinţa "Symposium on Operating System Priciples".

xgcc este atît de bun pe cît sunt extensiile scrise. Dacă ştii ce fel de bug să cauţi, îl poţi găsi (de exemplu, trebuie să ştii că `lock/unlock` se folosesc împreună în felul indicat). Cînd cercetătorii de la Stanford validau erorile, căutau exemple similare în cod, pentru a vedea cum trebuie făcute de fapt lucrurile.

De aici s-a născut ideea lor cea mai interesantă: avem la dispoziţie aproape două milioane de linii de cod. De ce să nu folosim acest cod pentru a extrage automat regulile de programare? De exemplu, dacă în cod funcţiile `lock` şi `unlock` apar mereu împreună, şi în ordinea asta, înseamnă că asta e o regulă care trebuie respectată.

Bine, dar dacă în program sunt bug-uri? Atunci regula va fi încălcată. Pentru a infera reguli, vom căuta în program construcţii care apar foarte frecvent, chiar dacă nu apar întotdeauna. Dacă în 999 de cazuri din 1000 `lock` este urmat de `unlock`, dar în 1 la mie nu, considerăm că am dat peste o regulă importantă.

A fost făcută o listă de tipuri de reguli care trebuie căutate în cod (de exemplu, `<A>` este mereu urmat de `<B>`, unde `<A>` şi `<B>` trebuie descoperite), şi apoi codul a fost analizat exhaustiv pentru a obţine posibile reguli (de exemplu, `<A>` este `lock()` şi `<B>` este `unlock()`). Fiecare regulă a fost apoi analizată statistic, şi regulile au fost apoi sortate în raport cu deviaţia de la distribuţia uniformă. Apoi regulile cele mai promiţătoare au fost transformate în programe Metal.

Ingeniozitatea cercetătorilor însă nu s-a oprit aici. Dacă suntem nefamiliari cu nucleul, tot nu ştim dacă regulile acestea trebuie sau nu să fie respectate. Analizînd regulile, cercetătorii au observat că folosirea unei anumite construcţii în cod implică faptul că programatorul are o anumită *credinţă* despre cum trebuie să arate codul. De exemplu, dacă un programator compară un pointer cu zero, înseamnă că programatorul se aşteaptă că acel pointer să poată avea valoarea zero.

Cercetătorii au definit apoi nişte reguli simple care arată cum se propagă credinţele prin program şi care dintre credinţe sunt contradictorii. Voi ilustra din nou acest lucru cu un exemplu simplu:

```
char* p = f();
b = *p;     /* credinta: p nu poate fi zero */
...
if (p == 0) /* credinta: p poate fi zero */
   ...
```

În acest fragment de program, pointerul `p` este dereferenţiat (se citeşte valoarea la care punctează). Asta înseamnă că programatorul nu se aşteaptă ca în acest punct valoarea pointerului să fie zero. Dar, ceva mai departe în program, pointerul este comparat cu zero. Programatorul nu ar fi făcut această comparaţie dacă nu s-ar fi aşteptat să reuşească citeodată. Aceste două credinţe (că `p` nu poate fi zero, respectiv că `p` poate fi zero) sunt contradictorii. Una dintre ele trebuie să fie o eroare!

Folosind această metodă au fost construite noi teste care au descoperit alte cîteva sute de bug-uri în Linux.

5 Concluzii

În acest articol am prezentat proiectul numit Meta-compilare, care augumentează compilatorul cu informaţii specifice programului compilat. Compilatorul foloseşte aceste informaţii pentru a detecta posibile erori în program. Tipul proprietăţilor care se pot verifica folosind această schemă este destul de restrîns: fiecare procedură din programul de analizat este redusă la un automat finit a cărui traiectorie este urmărită pentru a vedea dacă trece prin stări de eroare. Am văzut de asemenea o schemă ingenioasă prin care codul este analizat pentru a extrage reguli de stil de programare în mod automat; cazuri în care regulile de stil sunt încălcate sunt potenţiale erori de programare. Aceste exemple constituie o aplicare a tehnicilor de verificare formală pentru analiza automată a programelor.

6 Alte surse de informaţie

- Articolul meu despre depanare din PC Report din iunie 1999 conţine consideraţii despre metodele statice şi dinamice: `http://www.cs.cmu.edu/~mihaib/articles/articles.html#ddd`.
- Am vorbit despre teoria calculabilităţii în articolul meu din PC Report din decembrie 1999: `http://www.cs.cmu.edu/~mihaib/articles/articles.html#complex`.
- Am descris pe scurt automatele finite în articolul meu din PC Report din aprilie 2000: `http://www.cs.cmu.edu/~mihaib/articles/articles.html#regex`.
- Pagina de web a proiectului MC de la Stanford: `http://hands.stanford.edu/linux`.
- Pagina de web a lui Dawson Engler, liderul proiectului MC: `http://www.stanford.edu/~engler`. Aici puteţi găsi toate articolele publicate de acest grup despre Metacompilation pe care se bazează prezentarea din acest articol.
- O pagină cu legături spre publicaţii care sintetizează starea curentă a metodelor de verificare formală: `http://www.cerc.utexas.edu/~jay/fv_surveys`

- O biblioteca virtuală cu informaţii despre verificare formală: `http://www.afm.sbu.ac.uk`
- O excelentă introducere în metodologia verificării formale pentru hardware: C.-J. H. Seger, "An Introduction to Formal Hardware Verification", Technical Report 92-13, UBC, Department of Computer Science, Vancouver, B.C., Canada, June 1992, `http://www.cerc.utexas.edu/~jay/fv_surveys/Seger.ps`