DDD — Un Debugger Vizual

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

18 mai 1999

Subiect: corectitudinea programelor şi scule pentru depanarea lor

Cunoştinţe necesare: cunoştinţe elementare despre programare şi sisteme de operare

Cuvinte cheie: verificare statică, verificare dinamică, debugger

Cuprins

Societatea contemporană se bazează din ce în ce mai mult pe calculatoare; cu atît mai dramatică este situaţia curentă a software-ului: mulţi specialişti estimează că cea mai importantă criză a tehnologiei informaţionale contemporane este robusteţea redusă a programelor produse.

Situaţia este într-adevăr îngrijorătoare: productivitatea medie a unui programator este de circa 3 linii de cod pe zi (cod comentat, depanat şi verificat); frecvenţa medie a erorilor este de una la o mie de linii.

Productivitatea programatorilor nu s-a schimbat în mod substanţial în ultimii douăzeci de ani, dar s-au schimbat enorm sculele pe care le au la dispoziţie. Limbajele folosite în ziua de azi sunt mult mai expresive, şi ca atare 3 linii de cod pot exprima mult mai mult. Scule sofisticate asistă programatorul în scrierea, verificarea, întreţinerea, portarea şi depanarea programelor.

În acest text voi face reclamă unui produs de excelentă calitate pentru depanarea programelor. Foarte interesant este faptul că produsul în seamă este în întregime free software, şi ca atare este disponibil oricui de pe Internet, pe gratis (am scris un articol întreg despre fenomenul "free software" cu mai mult timp în urmă în PC Report; copia articolului este accesibilă din pagina mea de web.)

Programul în sine este ceea ce se numeşte un "debugger vizual"; "vizual" pentru că permite o interacţiune foarte intuitivă utilizatorului. Calitatea sa rivalizează cu produse de firmă renumite, cum ar fi debugger-ul vizual al lui Microsoft. Pentru cei nerăbdători, programul se numeşte DDD (Data Display Debugger), şi poate fi obţinut, în surse sau executabile, cu documentaţia aferentă, de la Universitatea Tehnică din Braunschweig din Germania: `http://www.cs.tu-bs.de/softech/ddd`. De fapt vom vedea că DDD este doar un înveliş vizual care poate colabora cu alte debuggere care au interfeţe mai simple (linie de comandă).

Trebuie să recunosc că eu însumi sunt un ins destul de conservator, care în general preferă interfaţă seacă în linie de comandă, stil Unix, unei interfeţe vizuale "desktop", gen Microsoft Windows. Dar criteriul care ne face să alegem între cele două nu trebuie să fie unul religios, ci unul pragmatic. Productivitatea pe care o am în depanarea codului este mult crescută cu DDD[1], aşa că programul merită încercat.

1 Corectitudinea programelor

Corectitudinea programelor este o noţiune mult mai complicată decît pare la prima vedere; există o sumedenie de definiţii posibile, unele implicînd un aparat matematic sofisticat. Trebuie însă spus că, deşi există o cantitate enormă de cercetare în ingineria programării (software engineering), în metode de verificare automată a programelor, în generatoare de programe, rezultatele practice sunt cu douăzeci de ani în urma cerinţelor proiectelor moderne.

Fără a mă avînta în detalii (în care dealtfel nu sunt expert) pot totuşi face cîteva observaţii generale, sper interesante pentru cititor. Există două clase mari de metode folosite pentru a garanta/analiza corectitudinea programelor. Prima clasă mare priveşte programele în întregime, şi studiind codul sursă poate garanta proprietăţi pe care programul le va avea oricînd, indiferent de datele de intrare. Aceste soluţii se numesc *statice*.

A doua clasă studiază programele în execuţie cu anumite date de intrare, şi garantează oricînd că anumite lucruri nu se vor întîmpla niciodată, pentru că vor fi prevenite în mod explicit. Aceste

[1] Şi aici putem dezbate: putem zice că atunci cînd nu avem la dispoziţie un debugger, vom fi motivaţi să scriem un cod de calitate mai bună, gîndindu-ne mai mult la structura lui, tocmai pentru că depanarea este dificilă.

metode sunt cele *dinamice*. Debuggerele fac parte din clasa metodelor dinamice, deci vor avea partea leului în acest text. De aceea să acordăm cîteva cuvinte soluţiilor statice.

2 Soluţii statice

Trebuie spus dintru început că cele două clase de soluţii (statice şi dinamice) sunt fundamental diferite, pentru că proprietăţile pe care le pot garanta sunt altele. Există astfel proprietăţi care pot fi garantate numai cu soluţii statice, proprietăţi care pot fi garantate numai cu soluţii dinamice, sau cu ambele (sau, din păcate, cu nici una).

Scula statică cea mai comună este compilatorul. Un compilator traduce un program dintr-un limbaj sursă într-un limbaj destinaţie. Proprietatea pe care trebuie s-o garanteze compilatorul este că ambele programe au acelaşi "înţeles" (semantică).

Depinzînd de expresivitatea limbajului sursă, compilatoarele pot garanta static anumite proprietăţi ale programelor. De exemplu, în limbajele puternic tipizate (strongly typed), anumite erori sunt pur şi simplu imposibile. În Java, de pildă, nu poţi nicicum aduna un număr cu un caracter; această operaţie nici nu are sens, şi este explicit interzisă prin sistemul de tipuri al limbajului. Din cauza asta, putem fi siguri că programatorul nu va face niciodată o astfel de eroare.

Folosind proprietăţile deduse ale programului, compilatoarele moderne nu numai că transformă un program într-altul, ci efectuează şi o serie întreagă de *optimizări*. Analiza statică pe care compilatoarele o efectuează garantează că optimizările făcute sunt corecte, în sensul că nu schimbă semnificaţia programului (ci doar viteza lui de execuţie sau poate mărimea lui, sau alţi parametri de interes pentru optimizare).

Există proprietăţi care nu se pot garanta static niciodată; astfel de proprietăţi se numesc *nedecidabile*. De exemplu la întrebarea "va avea variabila `X` vreodată valoarea 0?" nu se poate, în general, răspunde în mod static. Fireşte, pentru unele programe, acest lucru poate fi dovedit, dar nu pentru *orice* program care manipulează variabila `X`. Acest lucru poate fi demonstrat matematic. Îmi propun ca într-un articol ulterior, consacrat logicii matematice, să revin asupra acestui fapt. O ramură a informaticii, numită teoria calculabilităţii, se ocupă cu astfel de fapte.

Mi se pare interesant de menţionat şi următorul fapt, care este tot o consecinţă a teoriei calculabilităţii: am văzut că există mai multe programe diferite care fac acelaşi lucru (de exemplu un program şi versiunea lui optimizată.) Ei bine, în general este imposibil de determinat care este *cel mai mic program* care face anumit lucru. Acest enunţ se mai numeşte şi *Teorema Non-Şomajului* pentru cei care scriu compilatoare (*full-employment theorem for compiler writers*). Asta înseamnă practic că seria de optimizări pe care le pot implementa compilatoarele este nesfîrşită! Formal vorbind, dacă cineva implementează un compilator pe care-l decretează perfect, atunci eu pot construi un alt compilator, care pentru cel puţin un program va genera un rezultat optimizat mai bine!

3 Soluţii dinamice

Pe de altă parte, o proprietate ca "`X` este 0" poate fi detectată cu o soluţie dinamică: pur şi simplu este suficient să verificăm înainte de fiecare instrucţiune care se execută dacă nu cumva efectul ei va fi să dea această valoare lui `X`.

În mod dinamic putem garanta o sumedenie de proprietăţi care nu pot fi garantate static. De pildă, pentru limbaje de gen Java, care nu permit accesul înafara marginilor unui vector, putem garanta acest lucru folosind teste dinamice: de fiecare dată cînd accesăm un vector, testăm indicele dacă are o valoare între limitele admise.

Analiza dinamică are şi ea limitările ei. De pildă, analiza dinamică nu va putea *elimina* astfel de teste din program; pe cînd un compilator inteligent va observa că într-un cod ca acesta:

```
int a[10], i;
for (i=3; i < 7; i++) a[i] = 0;
```

variabila `i` va fi tot timpul între 0 şi 9, limitele vectorului, deci va putea elimina testele indicelui din interiorul buclei `for`. Aşa ceva nu putem realiza în mod dinamic.

Este foarte important să înţelegem o altă limitare esenţială a metodelor dinamice: ele depind de datele de intrare. Dacă un program verificat dinamic nu face nici un fel de eroare pentru anumite date, nu avem nici o garanţie că cu alte date nu se va comporta prost. Prin contrast, în mod static putem verifica adesea proprietăţi care sunt valabile pentru orice intrări.

Vom trece în revistă mai jos o serie de alte metode dinamice de garantare a corectitudinii. Deşi depanarea nu garantează proprietăţi, ci doar permite verificarea lor în cursul execuţiei, o vom categorisi ca metodă dinamică. Vreau să subliniez că metodele care urmează sunt *metode practice*, care se bucură de un deosebit succes în activitatea programatorilor, şi să vă îndemn să le folosiţi în proiectele dumneavoastră.

3.1 Trasarea: debugging dumps

O metodă relativ primitivă, dar extrem de eficace, este de a face programul însuşi să indice progresul, inserînd în cod instrucţiuni de tipărire. Metoda este cîteodată singura posibilă: de exemplu cînd depanaţi nucleul unui sistem de operare, acesta nu are o noţiune de intrare/ieşire, şi nici nu poate fi pus sub controlul unui debugger (adesea poate fi *observat* cu un debugger, dar nu oprit şi modificat).

În programele mele eu pun în anumite zone de cod instrucţiuni de genul:

```
if (NIVEL_DEPANARE_OPTIMIZARE > 2)
       fprinf(log, "Am ajuns in acest punct si X=%d\n", X);
```

Apoi inventez o metodă prin care pot controla nivelul de depanare fără a fi nevoie să recompilez, sau chiar să reopresc programul. De pildă, programul poate avea opţiuni în linia de comandă, care modifică nivelul de depanare. În acest fel pot controla nivelul de depanare independent în diferite părţi ale programului; schimbînd nivelul obţin informaţii de detalii diferite.

În Unix se foloseşte adesea o metodă interesantă pentru a controla nivelul de depanare a unor programe care nu se opresc niciodată din execuţie, cum ar fi demonii care se ocupă de comunicaţia în Internet: nivelul de depanare poate fi controlat trimiţînd anumite semnale acestor procese, cîteodată în conjuncţie cu modificarea unor fişiere de configurare care controlează operaţiile demonilor.

3.2 Aserţiuni

O tehnică de o utilitate greu de supraestimat este cea a folosirii aserţiunilor. Toate limbajele moderne pun la dispoziţie aserţiuni; o aserţiune este o funcţie care termină execuţia programului dacă primeşte un argument nul. (Tocmai am măsurat numărul de aserţiuni în codul scris de mine: am în medie o aserţiune la 40 de linii de cod scrise.)

Aserţiunile sunt folosite pentru a verifica dacă anumiţi *invarianţi* ai programului sunt adevăraţi. De exemplu mărimea unui vector care se schimbă dinamic trebuie să fie pozitivă. Putem avea deci în codul care modifică acest vector ceva de genul:

```
char* resize(char* vector, int marime)
{
   assert(marime > 0);
   ....
}
```

De îndată ce vom viola această condiţie programul se va opri din execuţie.

3.3 Cod care se auto-verifică

Adesea aserţiunile pe care vrem să le verificăm sunt mai complicate; de pildă vrem să vedem dacă toate elementele dintr-un vector se însumează la 1000. Astfel de verificări sunt foarte costisitoare, şi nu ne putem permite să le facem permanent. Dar dacă suspectăm că avem un bug care violează acest invariant, atunci putem proceda astfel (în C cel puţin): creăm o funcţie specială care verifică invariantul, pe care o invocăm apoi dintr-o zonă de cod controlată de nivelul de depanare:

```
if (NIVEL_DEPANARE)
        assert(suma_vector() == 1000);
```

Astfel, folosim viteza calculatorului însuşi pentru a verifica corectitudinea programului, făcînd teste complicate. Cînd programul este “complet” depanat (mai exact cînd credem noi asta), putem scoate aserţiunile şi astfel de fragmente din cod foarte simplu; de pildă în C dacă definim macro-ul `NDEBUG` toate aserţiunile dispar automat din cod.

3.4 Software Fault Isolation

De fapt tehnologiile dinamice descrise mai sus fac parte dintr-o clasă foarte largă, numită *software fault isolation*. Tehnologia aceasta este folosită cu mult succes în mai multe produse, începînd cu nuclee ale sistemelor de operare, care permit utilizatorilor să le insereze în nucleu cod, şi terminînd cu programe comerciale ca Purify şi Insure.

Cele mai sofisticate astfel de scule funcţionează chiar fără a avea la dispoziţie programul sursă; ele fac ceea ce se numeşte *binary instrumentation*: modifică chiar fişierele executabile, inserînd verificări în anumite puncte cheie. De exemplu, foarte popularul program Purify, de la Pure Software, inserează cod special care verifică toate accesele la memorie, şi modifică funcţiile de alocare a memoriei din biblioteca standard. Astfel de software prinde foarte multe erori de acces la memorie, cum ar fi accese în zone de memorie nealocate, re-folosirea memoriei de-alocate, citirea unor zone de memorie neiniţializate, scurgeri de memorie (adică uitarea de a dealoca memoria alocată: "memory leaks"), etc. Anumite clase de erori, ca cele cauzate de pointeri eronaţi, sunt adesea foarte uşor de depanat cu astfel de scule (şi foarte greu altfel).

3.5 Debuggere

În fine, ajungem la subiectul acestui articol. Debuggerele sunt programe sofisticate, care permit executarea altor programe într-un mod controlat; ele permit execuţia, observarea şi modificarea altui program. Debuggerele folosesc o sumedenie de tehnici, înrudite cu software fault isolation (vom vedea cum), şi suport din partea sculelor care genereazā programele, şi a sistemului de operare. Debuggerele se comportă vis-a-vis de programul depanat asemănător cu nişte *interpretoare*, executînd instrucţiunile una cîte una. Pentru că scrierea unui interpretor (interpretoarele de cod-maşină se numesc "emulatoare") este o treabă foarte complicată, debuggerele folosesc anumite servicii puse la dispoziţie de către sistemul de operare.

Sistemul de operare este el însuşi o uriaşă sculă care face software fault isolation: el permite execuţia paralelă a mai multor programe, dar previne interferenţa lor, claustrînd accesele fiecăruia în propria lui zonă de memorie, cu ajutorul memoriei virtuale, şi controlînd interacţiunea dintre procese şi resurse prin apelurile de sistem. (Despre sisteme de operare am scris o mulţime de articole în PC Report în trecut).

4 Funcţionarea unui debugger

Vom discuta aici numai despre debuggerele care permit depanarea unui program în limbajul în care a fost scris (limbajul sursă); debuggerele care depanează numai limbaj maşină sunt substanţial mai simple, şi mai puţin utile.

Figura 1 ilustrează interacţiunile unui debugger; vom detalia pe fiecare în parte în secţiunile următoare.

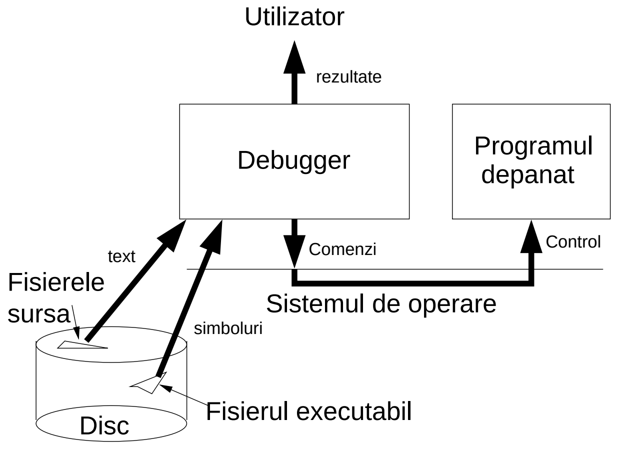

Figura 1: Un debugger interacţionează cu sistemul de operare, pentru a controla programul depanat, cu fişierul executabil, pentru a extrage informaţiile de depanare şi cu fişierele sursă, pentru a conversa cu utilizatorul.

4.1 Tabela de simboluri

Cel mai important suport îl primeşte un debugger de la compilatorul care translatează fişierul din sursă în limbaj maşină. Compilatorul poate fi rugat să depună în fişierul executabil informaţii despre structura programului. Aceste informaţii sunt ceea ce se numeşte "tabela de simboluri". Informaţiile asociază fiecare instrucţiune din codul maşină rezultat cu linia din codul sursă din care provine, indică adresele tuturor variabilelor din programul sursă, tipurile lor, adresele procedurilor şi tipurile lor, etc. Adesea mai mult de 60% din mărimea unui fişier executabil constă doar în informaţii de acest gen. Pe sistemele Unix există un utilitar numit `strip` (dezbracă) care şterge astfel de informaţii; el poate fi folosit pentru a face economie de spaţiu pe disc.

Fără informaţiile de depanare însă, debugger-ul nu poate face corespondenţa între codul obiect şi fişierele sursă.

Există o sumedenie de standarde de reprezentare a informaţiilor de depanare, care sunt menite să facă posibilă depanarea unui program generat de orice compilator cu orice debugger.

4.2 Breakpoint-Uri şi watchpoints

Pentru a fi eficace, un debugger foloseşte şi suportul oferit de hardware. Acest suport îi permite să execute programele de depanat cu viteză maximă, intervenind numai atunci cînd este nevoie, în loc de a le urmări execuţia pas cu pas, emulînd fiecare instrucţiune separat.

Iată cum funcţionează aceste mecanisme: microprocesoarele moderne au nişte regiştri speciali, în care se pot scrie felurite adrese. Procesoarele promit că atunci cînd aceste adrese apar în program, hardware-ul generează o excepţie, care poate fi tratată apoi de software.

Un *breakpoint* (punct de întrerupere) este o valoare care este comparată cu adresa curentă a codului din registrul PC (Program Counter): cu alte cuvinte, cînd programul atinge adresa indicată de un breakpoint, se generează automat o întrerupere.

În mod alternativ, un breakpoint poate fi o instrucţiune care generează ea însăşi o întrerupere atunci cînd este executată, numită chiar "instrucţiune breakpoint". Debugger-ele pot folosi ambele feluri de breapoint-uri.

O variantă de breakpoint-uri sunt cele *condiţionale*: acestea opresc execuţia numai dacă o anumită expresie are o anumită valoare, altfel continuă.

Un *watchpoints* (punct de supraveghere) este o valoare care este comparată cu adresa datelor; dacă un program vrea să citească sau să scrie la o anumită adresă, se generează o întrerupere.

4.3 Interfaţa sistemului de operare pentru depanare

Debugger-ul este un program, iar programul depanat este un altul. Cum de poate debugger-ul să controleze un program independent, care nu a fost scris în acest scop? Sistemele moderne de operare izolează programele unul faţă de altul; atunci cum de poate debugger-ul să se uite la cele mai intime informaţii din spaţiul de adrese al celuilalt program?

Fără ajutorul sistemului de operare nici n-ar putea.

Tot sistemul de operare pune la dispoziţie o interfaţă specială, prin care un program poate observa comportarea altuia. De asemenea, sistemul de operare asigură respectarea unor reguli de securitate: numai utilizatorul care a pornit un proces are dreptul să-l depaneze; altfel debuggerele ar putea fi folosite pentru a extrage informaţii nepermise.

Voi discuta aici interfaţa oferită de sistemele de tip Unix; deşi nu ştiu ce oferă sistemele de tip Windows, este probabil că au mecanisme înrudite.

Există în Unix în mod tradiţional două interfeţe pentru depanare oferite de sistemul de operare. Prima este mai veche, şi mai puţin elegantă. Ea constă dintr-un singur apel de sistem, numit `ptrace`.

4.3.1 `ptrace`

Ptrace vine de la *process trace*: urmăreşte execuţia altui proces. Acest apel de sistem de fapt implementează un mic limbaj, prin care debugger-ul conversează cu nucleul sistemului de operare. Iată cum arată `ptrace` pe Linux:

```
int ptrace(int operatie, int proces, int adresa, int date);
```

Prin execuţia acestui apel, debugger-ul cere nucleului să efectueze operaţia indicată asupra procesului descris, la adresa şi cu valorile pasate.

Iată exemple de operaţii: citeşte un cuvînt (de date, de cod, de stivă), scrie un cuvînt, trimite un semnal, opreşte procesul, reporneşte procesul, etc.

Un debugger pune un breakpoint în celălalt proces astfel: citeşte o instrucţiune din proces, o memorează local, şi o înlocuieşte cu o "instrucţiune breakpoint". Apoi nucleul este rugat să continue execuţia procesului, iar cînd breakpoint-ul este executat, o întrerupere transferă din nou control debugger-ului, care pune la loc instrucţiunea originală şi decide apoi ce să facă cu procesul.

Un breakpoint condiţional poate fi implementat ca un breakpoint obişnuit, doar că debugger-ul va verifica la fiecare oprire condiţia (citind din spaţiul celuilalt proces valorile necesare), şi va reporni programul doar dacă condiţia este falsă.

Interfaţa aceasta este oarecum incomodă, din mai multe motive.

- În primul rînd, ideea de a avea un mic limbaj de comunicare între nucleu şi debugger prin argumentul "operaţie" nu este în spiritul Unix, care are interfeţe cu semantică foarte simplă şi clară;

- În al doilea rînd, interfaţa este foarte restrictivă, pentru că permite accesul numai la nivel de cuvînt; mai mult, pentru fiecare cuvînt debugger-ul trebuie să execute un apel de sistem. Ori un apel de sistem este foarte costisitor, în termeni de timp de execuţie; din cauza asta un breakpoint condiţional va încetini substanţial execuţia programului depanat.

Din cauza aceasta, în 1984, cercetătorii de la AT&T (unde a fost dezvoltat Unix-ul original) au propus o interfaţă alternativă, extrem de elegantă, bazată pe un sistem de fişiere virtual.

4.3.2 `/proc`

Sistemul de fişiere `proc` oferă acces la toate procesele în execuţie în acelaşi fel în care oferă acces la fişiere. Am mai vorbit în alte articole despre eleganţa interfeţei Unix pentru accesul la fişiere, şi despre sistemul de fişiere `proc`; aici nu voi face decît să recapitulez faptele esenţiale.

În paranteză voi nota că tot cercetătorii de la AT&T au împins ideea accesului la obiecte prin interfaţa de fişiere şi mai departe în proiectul lor de sistem distribuit numit Plan9, care însă din păcate nu a devenit niciodată un produs comercial.

Dacă aveţi un sistem Linux la-ndemînă puteţi experimenta `proc` în mod direct, şi veţi înţelege mai precis ce vreau să povestesc. Mergeţi în directorul `/proc` şi uitaţi-vă. Fiecare proces în execuţie are aici un fişier care îi corespunde; procesul cu identificatorul 100 va avea un director cu numele `100`.

În acest director se află o mulţime de "fişiere virtuale", care descriu informaţiile nucleului despre acest proces. Aceste fişiere nu există pe disc; atunci cînd cineva încearcă să citească din aceste fişiere, nucleul de fapt extrage datele din memorie, din anumite structuri de date, şi le returnează cititorului.

În acest director există şi un fişier virtual care reprezintă întreg spaţiul de memorie al acestui proces; citind octetul 5 din acest fişier se citeşte de fapt octetul de la adresa virtuală 5 din spaţiul de adrese al acestui proces.

Utilizînd acest fişier, debugger-ul are imediat o vedere de ansamblu asupra întregului proces, putînd citi şi scrie zone întregi cu un singur apel de sistem de citire, respectiv scriere.

4.4 Core dumps

Sistemele de operare pun adesea la dispoziţie încă o interesantă funcţiune: fişierele-imagine (*core*). Utilizatorul poate inhiba crearea acestora cu apelul de sistem `setrlimit`.

Atunci cînd un proces execută o operaţie ilegală, nucleul îl omoară şi simultan crează un fişier numit `core`, care este foarte asemănător cu fişierul din `proc` care descrie imaginea de memorie a procesului; acest fişier conţine imaginea procesului aşa cum arăta el în momentul răposării. Debugger-ul poate folosi acest fişier pentru a studia cauzele morţii: poate inspecta variabilele, stiva, etc. în momentul morţii, dar nu poate executa programul.

4.5 Bibliotecile; bibliotecile dinamice

Dacă bibliotecile cu care programul este legat vin cu informaţii de depanare, debugger-ul poate fi folosit pentru a le depana şi pe ele. Debugger-ul captează apelurile prin care procesele îşi leagă bibliotecile dinamice în momentul execuţiei, şi le citeşte simbolurile de îndată ce sunt încărcate.

4.6 Funcţionarea unui debugger

Iată aici recapitulată funcţionarea unui debugger, pas cu pas:

1. Un debugger poate lansa procesul de depanat ca pe un proces-copil, sau se poate *ataşa* (attach) unui proces care deja se execută;

2. După aceea debugger-ul extrage informaţiile de depanare din tabela de simboluri a fişierului executabil;

3. Apoi debugger-ul încearcă să localizeze fişierele sursă din care programul a fost compilat; utilizatorul poate de obicei indica debugger-ului în ce directoare să caute aceste fişiere;

4. Apoi debugger-ul intră într-o buclă, aşteptînd comenzi de la utilizator.

5. După fiecare comandă debugger-ul acţionează asupra imaginii executabile în conformitate cu comanda;

6. Dacă utilizatorul vrea să vadă o variabilă, de pildă, debugger-ul va căuta în tabela de simboluri adresa variabilei, apoi va citi zona de memorie unde variabila se află. Pentru variabilele locale, care pot avea mai multe instanţe, debugger-ul le va căuta pe stivă, iar nu în segmentul de date; pentru asta debugger-ul trebuie să ştie cum îşi aranjează datele pe stivă fiecare procedură;

7. Cînd utilizatorul vrea să execute programul pentru un număr de paşi, debugger-ul instalează un breakpoint, după care roagă nucleul să pornească programul, el dormitînd apoi în aşteptarea semnalului de oprire prin care nucleul indică faptul ca breakpoint-ul a fost atins;

8. Debugger-ul poate de asemenea să roage nucleul să trimită semnale pentru a întrerupe un proces în execuţie;

9. Utilizatorul poate cere modificarea unor valori, pe care debugger-ul le înscrie în zonele de memorie corespunzătoare;

10. Debugger-ul poate permite utilizatorilor să întrerupă secvenţa normală de execuţie; de exemplu utilizatorii pot termina imediat procedura curentă, pot cere apelul unor anumite funcţii, sau pot cere ca execuţia să se continue de la un anumit punct, şi nu de la instrucţiunea următoare.

5 gdb: un debugger cu linie de comandă

În această secţiune voi discuta pe scurt despre unul dintre cele mai răspîndite debuggere, cel al proiectului GNU (despre proiectul GNU puteţi citi în articolul meu anterior despre free software, pe care l-am mai menţionat). GDB a fost scris iniţial de Richard Stallman, celebrul iniţiator al Fundaţiei pentru Free Software (FSF), care a scris şi editorul de texte Emacs (despre care am avut un alt articol în PC Report) şi compilatorul de C numit GCC (Gnu C Compiler).

La data apariţiei sale, gdb avea tot felul de trăsături inovatoare. Voi menţiona pe scurt unele dintre cele mai exotice; gdb poate face toate lucrurile pe care le-am descris mai sus, cînd am vorbit în general despre debuggere.

Gdb este un debugger în linie de comandă, dar există mai multe interfeţe vizuale pentru el. O interfaţă excelentă permite folosirea lui gdb din editorul de texte Emacs; alte interfeţe sunt xxgdb, gdbtk, şi DDD, care este pretextul acestui articol.

- Gdb poate permite utilizatorului să evalueze expresii arbitrar de complicate; de la aflarea valorii unei variabile, la expresii care conţin operatori complicaţi, sau chiar apeluri de funcţii;

- Gdb poate fi folosit atît pentru inspectat cît şi pentru modificat valori. În ambele cazuri comanda este `print`. De exemplu putem spune:

```
(gdb) print a
$1 = 5
(gdb) print a=3
$2 = 3
(gdb) print sin(a)
```

```
$3 = .14112000805986722210
(gdb) print $1
$4 = 5
```

Semnele $ de mai sus pot fi folosite pentru a re-evalua expresii complicate. $1 ţine locul expresiei `a` mai sus.

- Comenzile gdb pot fi abreviate, daca nu sunt ambigue; de pildă `print` se poate scrie simplu `p`. De asemenea, tastînd ENTER drept comandă va repeta execuţia comenzii precedente, ceea ce este foarte economic.

- Gdb poate permite tipărirea vectorilor, şi tipăreşte şirurile de caractere într-un mod inteligent. De pildă, dacă în program aveţi un vector alocat cu

  ```
  int *array = (int *) malloc (lungime * sizeof (int));
  ```

  puteţi tipări array-ul cu:

  ```
  (gdp) p *array@lungime
  ```

- Puteţi folosi gdb şi pentru a chema propriile dumneavoastră funcţii. Eu implementez în codul meu funcţii care tipăresc structurile de date complicate. Cind folosesc gdb pot inspecta aceste structuri chemînd funcţiile din linia de comandă:

  ```
  (gdb) print tipareste_arbore(t)
  1
  |--2
  |--3
     |--4
     |  |--5
     |--6
  $1 = void
  (gdb)
  ```

  Rezultatul evaluării funcţiei `tipareste_arbore` este nul (`void`), dar efectul executării ei se vede pe ecran.

- Gdb oferă o multitudine de comenzi pentru controlul stivei:

  backtrace: arată procedurile în curs de execuţie pe stivă, cu argumentele lor;

  down/up: mută punctul de interes în procedura apelată, respectiv apelantă (dar nu şi execuţia); se pot apoi inspecta variabilele locale ale acestor proceduri;

  return: termină imediat procedura curentă, returnîndu-se la apelant;

  frame: face o anumită procedură de pe stivă punctul de interes.

- Gdb poate face ca breakpoint-urile să evalueze de fiecare dată o expresie şi să continue execuţia dacă expresia e falsă. Sau poate ignora fiecare breakpoint pentru un număr de repetiţii. Breakpoint-Uri se instalează cu comanda `break`, se şterg cu comanda `delete`, şi pot avea ataşate condiţii cu comanda `cond`. Watchpoints se pot instala cu comanda `watch`;
- Gdb poate fi folosit pentru a depana programe care au mai multe thread-uri concurente în execuţie.
- Gdb permite definirea de variabile noi, numite cu $ în faţă. Să zicem că avem un vector de structuri, şi vrem să vedem din fiecare cîmpul `cheie`; putem face asta în felul următor:

  ```
  (gdb) set $i = 0
  (gdb) p vector[$i++].cheie
  (gdb) ENTER
  (gdb) ENTER
  ...
  ```

  La fiecare ENTER se va re-executa instrucţiunea anterioară, care va incrementa variabila `$i`, plimbîndu-se prin vector;
- Cu comanda `display` gdb poate fi rugat să evalueze şi tipărească o expresie de fiecare dată cînd se programul urmărit se opreşte din execuţie;
- Gdb poate tipări informaţii despre tipurile obiectelor, şi chiar şi tipuri:

  ```
  (gdb) whatis v
  type = struct complex
  (gdb) ptype v
  type = struct complex {
      double real;
      double imag;
  }
  ```

- Gdb poate fi folosit pentru a depana programe la distanţă, pe alte calculatoare; gdb poate fi conectat prin reţea la programul depanat. În felul asta se pot depana nucleele unor sisteme de operare, sau programe pe sisteme de operare care nu suportă debuggere. Programele vor trebui atunci să fie legate cu nişte proceduri speciale, care transformă programul depanat într-un server, debugger-ul fiind un client pe o maşină la distanţă. Pentru a depana la distanţă pe sisteme care suportă debuggere, programul `gdbserver` funcţionează simultan ca un debugger şi ca un server (şi atunci nu mai trebuie legat codul special în programul depanat). Gdbserver rulează pe aceeaşi maşină cu programul de depanat, iar gdb se conectează la server prin reţea, permiţînd depanarea la distanţă.
- Gdb oferă un limbaj simplu în care poate fi programat, şi poate executa "scripturi" (mici programe). Oferă construcţiile `if`, `while`, `define` (care defineşte macrouri).

Figura 2:

6 DDD – o interfaţa vizuală

În fine, voi spune cîteva cuvinte despre DDD. DDD nu este un debugger, ci doar o interfaţă pentru un alt debugger. DDD beneficiază de toată puterea debugger-ului pe care îl foloseşte, pentru că una din ferestrele sale oferă de fapt chiar interfaţa în linie de comandă clasică cu debugger-ul.

DDD face practic inutilă învăţarea majorităţii comenzilor gdb, pentru că oferă interfeţe foarte intuitive. Figura 3 arată principalele ferestre ale lui DDD.

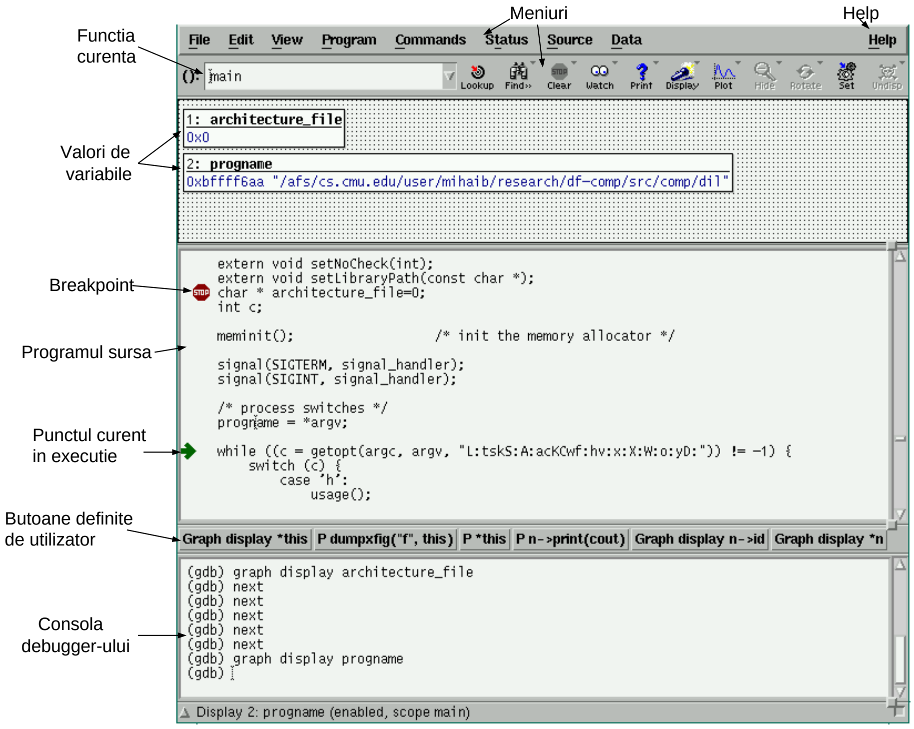

Figura 3: Cele trei ferestre ale lui DDD: **sus** pentru inspecţia valorilor, **centru** pentru surse, **jos** pentru consola depanatorului.

Iată ce poate face DDD mai bine decît debuggerele:

- Afişează simultan consola de interacţiune cu debugger-ul, o vedere asupra fişierului sursă, şi o imagine instantanee a structurilor de date care se doresc vizualizate;
- Drag-and-drop breakpoint-uri: poţi să creezi, muţi şi modifici breakpoint-uri cu uşurinţă;

- Controlul sesiunilor: DDD ţine minte o grămadă de lucruri pe care le faci, le poate salva, şi reîncărca altă dată. De exemplu DDD poate pune automat la loc toate breakpoint-urile. Cînd debugger-ul "crapă" (se mai întîmplă şi asta) DDD îl reporneşte automat;

- DDD se poate interfaţa cu debuggere multiple, nu numai cu gdb. Ştie de multe alte debuggere comerciale, ca dbx, sau debuggere specializate;

- Ca atare DDD suportă o mulţime de limbaje, nu numai C şi C++: folosind debuggere specializate, DDD poate fi folosit pentru a depana programe Perl, Python, Modula-2, Java;

- Cea mai impresionantă trăsătură a lui DDD este modul în care poate afişa structuri de date complexe şi relaţiile dintre ele. Figura 4 arată cum DDD înfăţişează structuri. Un simplu dublu-click pe un cîmp al unei structuri va afişa apoi cîmpul respectiv;

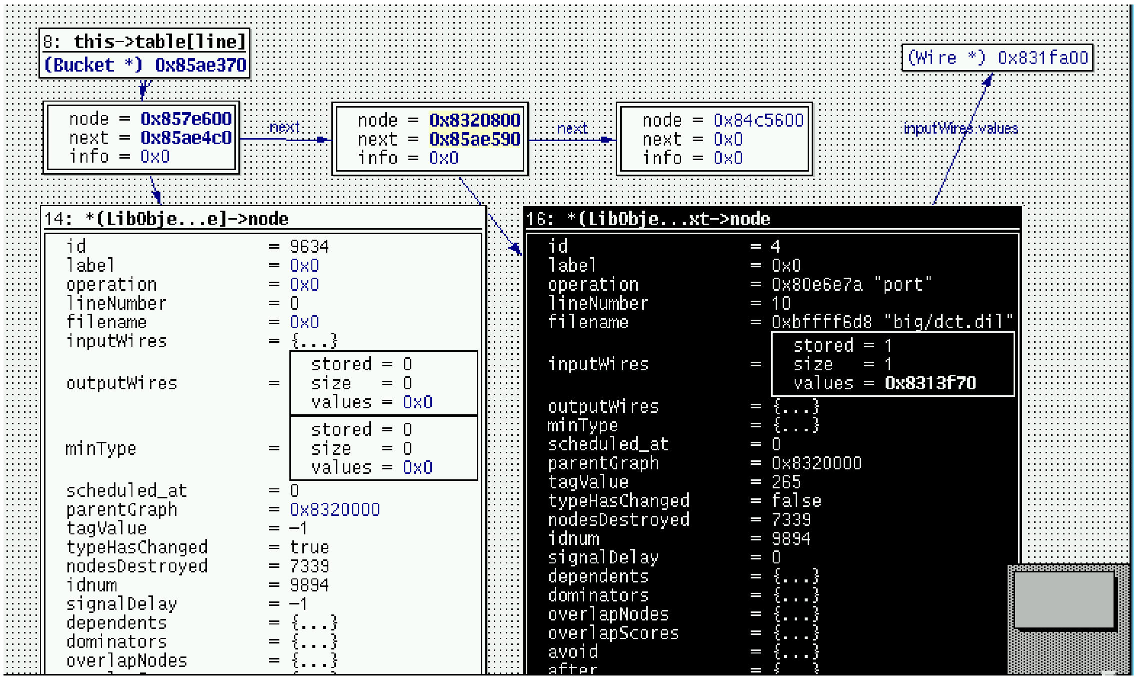

Figura 4: Un display care afişează relaţii complicate între variabile de tip structură.

- Pe lîngă afişarea datelor, DDD poate folosi programul de desenat grafice `gnuplot` pentru a face grafice ale unor array-uri enorme. Asta este foarte util pentru cei care scriu cod pentru calcule cu matrici mari;

- Utilizatorul îşi poate defini propriile butoane, care cînd sunt apăsate trimit comenzi complexe debugger-ului;

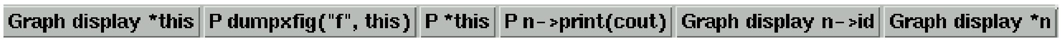

Figura 5: Butoane definite de utilizator, cu funcţiuni complexe.

- Este suficient să duci mouse-ul pe o variabilă din fereastra cu sursa programului pentru a vedea valoarea ei;

- Debugger-ul memorează valorile afişate atunci cînd programul se opreşte, şi permite mersul înainte şi înapoi pentru a vizualiza valori din trecut, cu butoanele Undo/Redo.

Figura 6: Panoul de comandă central al lui DDD, cu funcţiunile cele mai importanta.

Una peste alta, DDD este o sculă extrem de ergonomică, şi care se poate dovedi de un ajutor de nepreţuit pentru cei care au de depanat programe complexe.

Vă doresc programe cît mai corecte!