Miroslav Vykydal

Závod výpočetní techniky ČSAD Brno

TESTOVÁNÍ PROGRAMŮ

1. Motivace

Již od počátku programování si každý programátor nutně klad otázku,zda jeho program plní právě ty funkce,které měl na mysli při jeho vytváření.Zpočátku byly nároky malé,stačila vůbec nějaká zdařilá realizace programu.Při detailnějším pohledu se však ukázaly menší či větší závady ve vyprodukovaných výsledcích,bylo nutné hledat příčiny těchto chyb,dát záruku,že se již nebudou opakovat. Se zvyšující se složitostí programů se zvyšovala i složitost ověřování jejich správnosti.Pojem "správný program" se ukázal jako velmi relativní. Je "správný" program,který sice produkuje správné výstupy,ale nedokáže si poradit s nestandartními daty? Zpočátku se testování programů spojovalo mylně s testováním hardwerové spolehlivosti počítače. S příchodem programovacích jazyků vyšší úrovně, rozvojem programovacích metod a vytvářením složitých programových komplexů se zjišťování,zda program pracuje správně, stalo velmi aktuálním programem.Ukázalo se,že dodatečné úpravy a opravy spotřebovávají neúměrné množství programátorské kapacity,brání ve vytváření nových spolehlivých programů. Poznatky získané při ladění programů,ale i aplikace výsledků některých matematických disciplin utvořily metodu ověřování správnosti programu - testování programů.

Tento příspěvek nechce být vyčerpávající studií o testo-

vání, chce přispět ke zvýšení zájmu širší programátorské veřejnosti o tuto aktuální problematiku.Zasvěcenci jistě odpustí některé terminologické nepřesnosti zaviněné drobnými nejasnostmi v užívaných označeních, milovníci formálního stylu zápisu prominou nedostatečnou formalizovanost, praktici snad vydrží některé formálně pojaté pasáže. Pokusím se naznačit základní problémy testování,uvést možnosti řešení následujících otázek.

Co je to testování?
Kdo testuje?
Kdy testovat?
Co testujeme?
Proč testovat?
Jak testujeme?

2. Co je to testování?

Pod pojmem testování programů budeme rozumět metodu ověřování správnosti programu jeho praktickou aplikací na počítači na množině vstupních dat vybrané dle předem zvolené strategie. Nedílnou součástí testování je analýza získaných výstupních dat, lokalizace zdroje chyb a registrace již prověřených částí programů.

Program P nazveme korektním vzhledem k funkci F právě když Dom F$\subset$Dom P a pro všechna v $\in$ Dom F platí F/v/ = P/v/,kde F je funkce,jež má být realizována programem P,Dom P množina vstupních dat programu P, na nichž P skončí/bez havarie/, Dom F definiční obor funkce F.

O formální důkaz korektnosti programu se pokouší tzv. dokazování programů /5/. Podat důkaz korektnosti programu je však algoritmicky neřešitelný problém /13/. Proto si teoretikové testování už ani nekladou otázku - jak testovat /my se jí budeme zabývat později/,nýbrž hledání řešení problému - jak dlouho testovat, do jaké úrovně testování jít. Je samozřejmé,že nutnou podmínkou pro zjištění korektnosti programu je realizace všech příkazů programu. Kdybychom některé příkazy opomněli realizovat,

pak o jejich správnosti nevíme nic.

Pod pojmem realizace určité množiny příkazů v programu budeme rozumět proces, kdy po konečně mnoha příkazech je provedena daná množina příkazů. Pokud je tato množina taková, že příkazy následují bezprostředně za sebou a první/poslední/ příkaz je příkaz START /STOP/, pak hovoříme o realizovatelné cestě v daném programu. Množinu vstupních dat, která umožňuje realizaci určité cesty, budeme nazývat - množina vstupních dat realizujících danou cestu.

S jistou dávkou nepřesnosti budeme dále předpokládat, že každý program lze prezentovat jeho příslušným vývojovým diagramem. Pak můžeme užít grafové terminologie a dvojici bezprostředně následujících příkazů, z nichž pouze první a poslední může být součástí více než dvou hran, ostatní tvoří hrany pouze se svým bezprostředním předchůdcem a následníkem, označíme jako větev programu.

Jestliže realizujeme všechny příkazy, pak ještě nemusí být program korektní. Pokud na obr. 1 zajistíme realizaci cesty /0,1,2,3,4,5,6/, chybné předání řízení 2 - 4/čárkově/ nebude odhaleno. Zvýšením úrovně testování, např. požadavkem na realizaci všech hran, již tuto chybu odhalíme. Je nutné vždy zvážit, jakou úroveň zvolit za mezní. Neboť pokud neodhalíme žádnou chybu, pak program prohlásíme za správný vzhledem k dané úrovni testování. Stále však musíme mít na paměti, že program může obsahovat chyby, které však nebyly odhaleny neboť nebyla zvolena dostatečně vysoká úroveň testování. Proto nesmíme považovat za chybu metody, když i přes všechnu snahu, čas a vynaložené prostředky otestovaný program zhavaruje. Jako klasický příklad může sloužit projekt Apollo, kde cena za testování tvořila 80% celkových nákladů na softwarové vybavení/23/, a přesto došlo u Apolla 8 a Apolla 11 k selhání tohoto softwaru /22/.

Co je tedy cílem testování? Cílem není dokázat korektnost programu, nýbrž právě naopak dokázat jeho nekorektnost, odhalit jeho chyby. Objevení tohoto faktu mělo pro programování a vývoj

důkazů korektnosti programů téměř takový význam jako objevení neeuklidovské geometrie pro matematiku /9, str.335/.

Při tom si je nutné uvědomit,že při zjištění chyby,byly vlastně objeveny jen následky chyby v sémantice programu /16, str. 189 - 199/. Programátor je potom zaměstnán hledáním příčiny chyby a opravou chyby /6/. Má-li být testování užitečné, musí poskytovat i informace umožňující lokalizovat chybu. Testování se někdy zaměňuje s tzv. debuggingem, což je proces lokalizace a odstranění chyby,která byla zjištěna testováním.

I když testování neumožní zachytit všechny situace,je to patrně jediný způsob,který se dá v praxi použít /6/. Testování programu si však nelze v žádném případě představovat jako náhodné bloudění programem bez jakéhokoliv sysému a metody. /23, str. 166 - 192 / a jen výjimečně nadaní umělci se dožijí úspěchu při testování programu.

3. Kdo testuje?

Zdálo by se,že nejpovolanější osobou je sám autor programu. Tato volba má svoje výhody,ale i nevýhody.Nezapomínejte, že cílem testování je odhalit chybu v programu.To je však proti jistě normální snaze každého programátora produkovat správné programy. Proto se nelze ani příliš divit,když význam chyby se snaží programátor zlehčovat a po provedené opravě považuje následný výskyt chyby za osobní prohru a životní zklamání.

Aby byl programátor zbaven těchto nepříjemností,je doporučováno/23/ vytvořit zvláštní testovací skupinu.Cílem pracovníků v této skupině je odhalit chybu v programu,počet odhalených chyb je pak měřítkem úspěšnosti jejich práce. Praktické zavedené této skupiny by však narazilo na známé problémy. Program by musel být zdokumentován natolik, aby byly ale poň přesně popsány vstupy a výstupy.Jen tak by bylo možné rozlišit správné vstupy od chybných a zjistit chování programu na nesprávných vstupech. Z popisu výstupů z programu musí být jasné, co se má považovat za chybný výsledek. V průběhu testování by nemělo docházet ke změnám v požadavcích zadavatele,které

by změnily správnost či nesprávnost vstupů a výstupů. A nakonec by muselo být výpočetní středisko natolik personálně silné, aby si mohlo dovolit vyčlenit do testovácí skupiny alespoň jednoho zkušeného programátora.

Všem znalcům poměrů v našich výpočetních střediscích je jasné, že ustavení testovací skupiny je více či méně vzdálená budoucnost. A to ještě ani netuší, že tato skupina by musela být vybavena příslušným testovacím softwarem, tj. testovacím systémem, do kterého by vstupoval program a výstupem by byly výsledky realizace tohoto programu na vytvořených testovacích datech a komentářem minimálně o tom, které části programu byly realizovány a v jakém pořadí.

Všem znalcům poměrů v našich výpočetních střediscích bych však doporučil, aby se před kategorickým odmítnutím a zavržením testovací skupiny pokusili vyčíslit, kolik času se věnuje na opravy /nikoliv údržbu/ programů. Mnohdy se chyba objeví až po upozornění uživatele, obtížně se stanovuje za jakých okolností nastala, autor programu již nemusí být zaměstnán ve středisku a proto je nutno celou logiku programu nastudovat znovu, program může být přebrán z jiného střediska atd.

4. Kdy testovat?

Jak z předchozího výkladu vyplývá, je testování náročné na čas i na pracovní kapacitu. Proto je nutné, aby nutnost prověřit správnost programu uznávali, alespoň mlčky, i uživatelé. Ti však většinou měří úspěšnost programátora dle absolutní délky tvorby programu, kde počátek tvoří sdělení požadavků od uživatele a konec je uživatelem spatřován v momentě, kdy spatří první výsledky povětšinou ve formě sestavy. Tento zvyk traduje se patrně z dob, kdy rozradostněný programátor hnán překvapením, že vůbec něco ze stroje vyšlo, běžel k nedočkavému uživateli. Ten okouzlen správným uspořádáním údajů na stránce začal věnovat pozornost její hodnotě až po prvním rutinním zpracování. Pak mu nezbylo než programátora upozornit, nejlépe osobně, na nedostatky v sestavě a trpělivě čekat na opravu.

Proto je nutné, aby programátor předkládal uživateli i ukázky testů,ze kterých je patrná správnost výsledků jeho práce.

Je někdy až zarážející,že např. nikdo by si nedovolil přesvědčovat soustružníka,aby bez důkladného proměření všech rozměrů součástky zahájil její sériovou výrobu,ale kontrolu práce programátora považuje za zdržování a plýtvání časem.

Pokud čas vynaložený na testování bude považován za promarněný,nebude s touto časovou kapacitou počítáno při plánování práce programátora,do té doby můžeme o testování jen se závistí,zájmem či dokonce obdivem číst v zahraniční literatuře.

5. Co testovat?

Testuje se samozřejmě program bez sysntaktických chyb. Jako zvlášť vhodné pro námi navrhovanou metodiku jsou nerekurzivní programy psané ve vyšším programovacím jazyku/Cobol,Fortran,PL/I/ nejlépe s dodržováním zásad strukturovaného programování /28/.

Mimo programu jako celku je možné testovat i jednotlivé programové moduly.To samozřejmě umožňuje lepší lokalizaci chyby. Pokud by však bylo nutné simulovat správnou činnost dalších dosud neotestovaných modulů, zvyšují se náklady na testování. Obdobná situace je i při testování celého systému programů na sobě závislých. Pokud tedy následují v pořadí jeden za druhým bez ohledu na výsledky jejich činností, je testování poměrně snadné. Pokud je však po provedení jednoho programu vybráno na základě výsledků jeho činnosti pokračování z více alternativ,je nutno prověřit/i při otestování jednotlivých programů/ i tuto logiku předávání řízení v rámci systému programů.

O nutnosti minimální dokumentace programu jsme se už zmínili.Ať už bude testovat program kdokoliv, musí mít možnost zjistit, zda výsledky práce jeho programu jsou správné. Zajištění správných výsledků pro srovnání je nezbytné. Pokud se tedy jako testovací data vezmou reálné údaje,může výpočet správných výsledků působit potíže a prodlužovat čas nutný k testování.

Pokud tedy není možné záskat správné výsledky snadno,např. z dřívějšího ručního zpracování,je vhodné volit vstupy jednoduché i za cenu jejich vytvoření.

O tom, že by programátor měl mít zdokumentovaný algoritmus natolik,aby dokázal najít příčinu případné chyby,se nebudeme rozepisovat neboť to přísluší metodikům a teoretikům vlastního programování.

6.Jak testovat?

Laskavý čtenář je jistě notně netrpělivý,očekává patrně název československého testovacího systému, jeho stručný popis a podmínky při jeho získávání, Tento čtenář od tohoto momentu přestane být patrně laskavý,neboť o žádném našem testovacím systému nic nevím a názvy TDEM,PET,HXVP,DAVE,JAVS, EFFIGI jsou pouze lákavým obalem na málo známém zahraničním zboží.

Chtěl bych zde předložit k posouzení návrh metodiky testování.Čím více diskusí vyvolá,tím lépe.

Systém pro testování je možno vytvářet dvěma způsoby.

Přesně definovat třídu programů,pro které bude testovací systém určen a ostatní programy ponechat bez povšimnutí.Takto můžeme algoritmicky zvládnout generování testovacích dat.Třída programů je však značně omezena / 1/ /2/ /3/ /4/ /29/.Zhruba řečeno programy nemohou obsahovat aritmetické operace s proměnnými.Jsou povoleny jen pro proměnné,které řídí opakování cyklu. I přes značné úsilí nepodařilo se zatím tuto třídu programu rozšířit /2/. Co je tedy vůbec možné? Je možné číst a zapisovat proměnné /READ,WRITE/,porovnávat dle hodnoty /A menší B, A menší konstanta,atd./,přesunovat hodnotu proměnné do jiné proměnné/MOVE A TO B/ a přesunovat konstantu do proměnné /MOVE konstanta TO A/. Odměnou za toto omezení jsou data,která prověří program tak,že všechny větve programu schopné realizace jsou realizovány/mluvíme o tzv.úplnosti vzhledem k větvím/, jsou označeny nerealizovatelné cesty programu /tj. neexistují vstupní data,která by je realizovala/ a identifikovány části programu, kde dojde k zacyklení a data,která toto zacyklení

způsobí.

Druhý způsob se skládá ze dvou kroků/29/. Nejprve provedeme výběr cest a stanovení odpovídajícího popisu testovacích dat,potom generování dat na základě jejich popisu.Na programy nebudou kladena žádná omezení. Proto první krok je možné provést vždy,druhý nemusí být vždy algoritmicky řešitelný.Často však generování dat provede programátor sám,bez pomocí testovacího systému.

Výběr cest provedeme nejprve tak,aby bylo zjištěno testování úplné vzhledem k příkazům.Využijeme výsledků z prací/21/ /25/. Nejprve ve stručnosti definujeme pojem - báze vývojového diagramu.

Je-li B podmnožina množiny všech příkazů vývojového diagramu a C množina cest tohoto vývojového diagramu taková,že k libovolnému příkazu z B existuje cesta z C obsahující tento příkaz, pak B nazýváme báze vývojového diagramu,právě když B je minimální množinou,pro kterou platí,že sjednocení všech cest z C je rovno množině všech příkazů programu.

Tedy pokud zjistíme realizaci všech příkazů báze,potom máme zaručenou i realizaci všech příkazů programu.

Možnosti určení báze jsou popsány v /21/ /25/, nebudeme se jimi podrobně zabývat. Naším úkolem je sestrojit množinu cest C. Cesty nebudou mít libovolnou délku/rovno počtu příkazů,které ji tvoří minus jedna/,nýbrž pro snadnější orientaci ve výsledcích testování budeme konstruovat cestu o minimální délce. Ze stejných důvodů budeme i minimalizovat počet těchto cest /29 /30/.

Vývojový diagram rozdělíme na tzv. vrstvy. I-tou vrstvu vzhledem k počátku tvoří příkazy,jejichž minimální vzdálenost od počátku je rovna i. Nultou vrstvu vzhledem k počátku tvoří pouze příkaz START. Vrstvu i + 1 sestrojíme takto: pokud některý bezprostřední následník prvku i-té vrstvy neleží v žádné z vrstev 0,1,---,i,je prvkem vrstvy i + 1. Takto projdeme bezprostřední následky všech prvků i-té vrstvy.

Pokud změníme orientaci hran vývojového diagramu v opačnou dostaneme tzv. opačný vývojový diagram.Pak můžeme použít

na tomto opačném vývojovém diagramu výše popsané metody konstrukce systému vrstev a obdržíme vrstvy vzhledem ke konci, neboť prvním příkazem v opačném vývojovém diagramu je příkaz STOP původního vývojového diagramu.

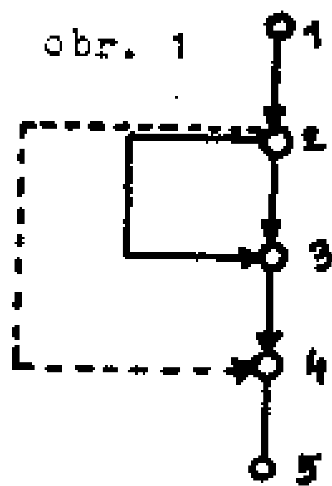
Vrstvy vzhledem k počátku budeme zakreslovat od nulté směrem dolů, vrstvy vzhledem ke konci od nulté směrem vzhůru. Dva příkazy z různých vrstev spojíme orientovanou hranou pokud v příslušném vývojovém diagramu byl jeden bezprostředním následníkem druhého. Pak můžeme pomocí systému vrstev vzhledem k počátku zkonstruovat minimální cestu z příkazu START do libovolného příkazu a pomocí systému vrstev vzhledem ke konci můžeme zkonstruovat minimální cestu z libovolného příkazu do příkazu STOP. Této vlastnosti využijeme při konstrukci množiny cest C obsahující všechny prvky báze B.

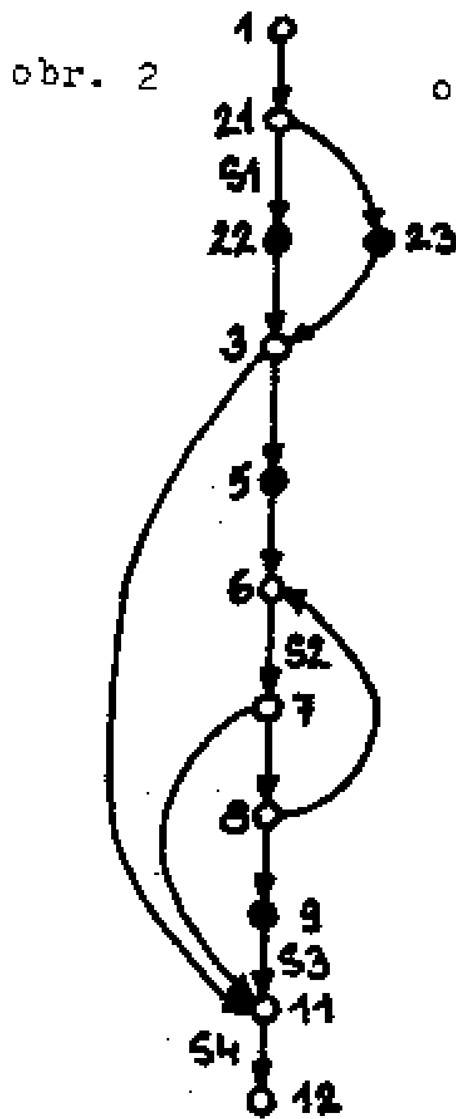
Vybereme libovolný prvek b báze B. Pak v systému vrstev vzhledem k počátku určíme minimální cestu z b do STOP. Na takto vytvořené cestě mohou ležet mimo b ještě další prvky báze B. Množinu těchto prvků báze ležících na vytvořené cestě odečteme od B,ze zbytku zvolíme další prvek báze a celý postup opakujeme dokud zbytek není prázdná množina.

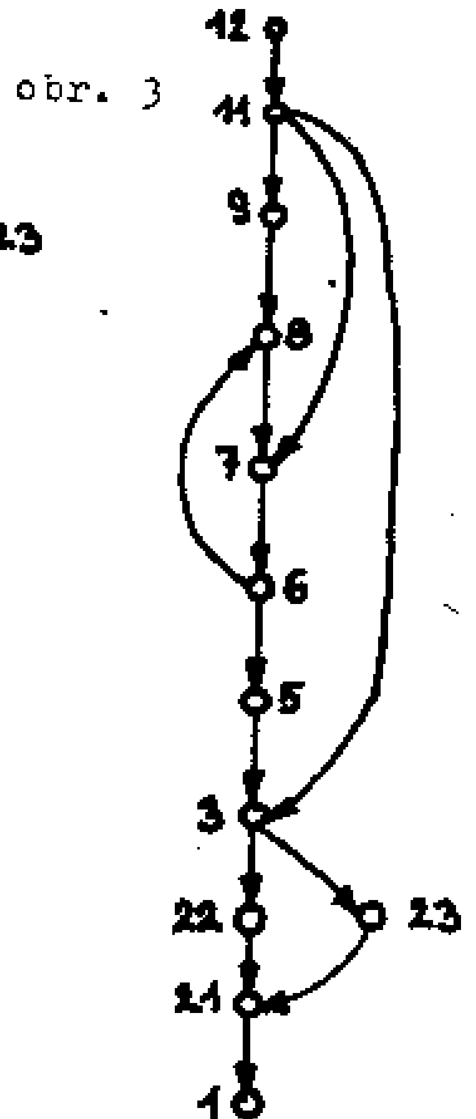
Může nastat případ,že k danému prvku báze existuje více než jedna cesta o minimální délce. Pak je možné provést výběr cest tak,aby jejich počet byl minimální /30/ a zůstal splněn základní požadavek na množinu cest C,tj.,aby každý prvek báze B ležel alespoň na jedné cestě z C.

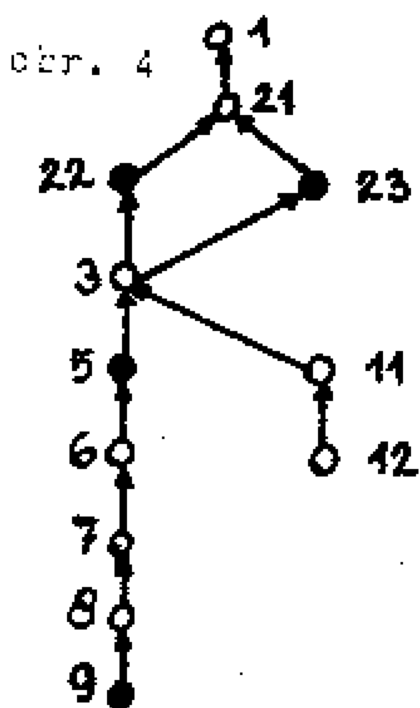
Příklad:

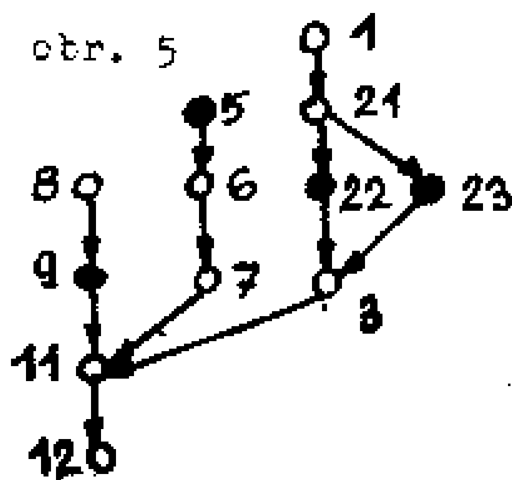
Na části cobolovského programu si ukážeme postup při sestrojení množiny cest C. Na obr.2 je vývojový diagram příslušný k tomuto modulu,na obr. 3 je pak zakreslen opačný vývojový diagram. Na obr.2 jsme označili čítače S 1,---,S4,které jsme zařadili dodatečně do programu. Na počátku při programu je hodnota všech čítačů rovna nule,při každém průchodu čítačem se jeho hodnota zvětší o jedničku. Z hodnoty čítačů je pak možné usuzovat na postup předávání řízení v programu,určit nerealizované části programu,příkaz,který byl prováděn v okamžiku


obr. 1
1
2
3
4
5


obr. 2
1
21
S1
22
23
3
5
6
S2
7
8
9
S3
11
S4
12


obr. 3
12
11
9
8
7
6
5
3
22
23
21
1


obr. 4
1
21
22
23
3
5
11
6
12
7
8
9


obr. 5
1
5
21
8
6
22
23
9
7
3
11
12

abnormálního ukončení,apod. Plně jsou vyznačeny prvky báze B. Na obr.4 je systém vrstev vzhledem k počátku,na obr.5 je systém vrstev vzhledem ke konci.

Ukažme si alespoň ilustrativně část konstrukce systému vrstev vzhledem k počátku. Nultou vrstvu tvoří pouze příkaz 1. Jediným bezprostředním následníkem je 21,to je tedy první vrstva. Druhou vrstvu tvoří bezprostřední následníci 22 a 23, třetí pouze 3,čtvrtou 5 a 11,pátou jejich bezprostřední následníci 6 a 12,šestou pouze 7 /12 nemá následníka/,atd.Bezprostředním následníkem 8 z vrstvy sedmé je i příkaz 6, ale ten je již prvkem páté vrstvy,proto jej do osmé vrstvy nezařadíme. Vrstvy vzhledem ke konci konstruujeme podobně na opačném vývojovém diagramu.

Část cobolovského programu:

```
1        ACCEPT PARAMETR
2        PERFORM KONTROLA - PARAMETRU THRU EX1
3        IF CHYBA - PARAMETRU = 'ANO'
4           GO TO KONEC.
5        OPEN INPUT VSTUPNI - SOUBOR
      CTENI.
6        READ VSTUPNI - SOUBOR
7           AT END GO TO KONEC.
8        IF VSTUP = PARAMETR
9           DISPLAY' NALEZENO '
         ELSE
            GO TO CTENI.
      KONEC.
11       CLOSE VSTUPNI-SOUBOR
12       STOP RUN.
      KONTROLA - PARAMETRU.
21       IF PARAMETR NOT NUMERIC
22          MOVE 'ANO' TO CHYBA - PARAMETRU
            DISPLAY 'CHYBA PARAMETRU '
         ELSE
23          MOVE 'NE' TO CHYBA - PARAMETRU.
      EX1.  EXIT.
```

Množinu cest C můžeme sestrojit např. takto: k prvku báze B 22 najdeme minimální cestu z počátku do 22 /obr.4/, je to cesta /1, 21,22/ a minimální cestu z 22 do konce /obr.5/ je to /22,3,11,12/. Cesta o minimální délce obsahující příkaz 22 je cesta C1 = /1. 21, 22, 3, 11, 12/. Podobně k prvku báze B 9 sestrojíme cestu C2 = /1, 2, 23,3, 5, 6, 7, 8, 9, 11, 12/. Cesty C1 a C2 obsahují všechny prvky báze B = /23, 22, 5, 9/, a proto další cesty již nebudeme sestrojovat.

Sjednocením příkazů tvořících cesty C1 a C2 obdržíme množinu všech příkazů našeho programu. Jestliže realizujeme cesty C1 a C2, pak realizujeme všechny příkazy programu /úplnost vzhledem k příkazům/. A k jakým výsledkům na této úrovni dospějeme, to ukazuje následující přehled:

	C1	C2
Parametr	SPACES	111...1
Vstup	libovolný	111...1
Display	chyba parametru	nalezeno
S1	1	0
S2	0	1
S3	0	1
S4	0	1

V případě C1 dojde k chybě neboť příkaz CLOSE nepředchází příslušný příkaz OPEN.

Dalším úkolem je k vybrané množině cest programu nalézt data,která by tu množinu realizovala a opakovanou realizací programu prověřit jeho korektnost. Data nutná pro realizaci určité cesty nalezneme pomocí tzv.popisu dat,což je množina podmínek,které musí data splňovat, aby realizovala danou cestu. Zatímco popis dat sestrojíme vždy, je problém generování dat na základě jejich popisu obecně algoritmicky neřešitelný. I přesto však může programátor v mnoha případech sestrojit hledané data heuristickým postupem.

Popis dat je možno provést několika způsoby.Můžeme postupovat od prvního příkazu cesty k poslednímu a postupně slučovat podmínky.Např./zápis v pseudojazyku blízkém k jazyku Cobol/

```
1           START
2           READ VSTUP INTO PROMENNA
3           MOVE KLIC TO S - KLIC
        ZPRACOVANI.
4           IF S-KLIC = KLIC
5              PERFORM SOUCET-HODNOT
            ELSE
6              PERFORM ZMENA-KLICE
7              MOVE KLIC TO S-KLIC.
8           READ VSTUP INTO PROMENNA
9              AT END GO TO KONEC.
10          GO TO ZPRACOVAT.
        KONEC.
11          STOP
```

Sestrojíme popis dat k cestě /1, 2, 3, 4, 5, 8, 10, 4, 6, 7, 8, 9, 11/. Pomocí Vi označíme prvky souboru VSTUP, Ki hodnoty proměnné KLIC z prvku Vi.

Příkaz	PROMENNA	KLIC	S-KLIC	Popis dat
1	-	-	-	-
2	V1	K1	-	-
3	V1	K1	K1	-
4,5	V1	K1	K1	K1=K1
8,10	V2	K2	K1	K1=K1
4,6	V2	K2	K1	K1=K1 AND K1≠K2
7	V2	K2	K2	K1=K1 AND K1≠K2
8,9,11	V2	K2	K2	K1=K1 AND K1≠K2 AND VSTUP=/V1,V2/

Popis cesty: K1≠K2 AND VSTUP=/V1,V2/

Jak je patrné z příkladu, je nutné evidovat hodnoty všech proměnných,aby bylo možné zapsat správný tvar podmínky do popisu dat. I kdybychom předem určili,které proměnné vystupují ve všech testech, museli bychom evidovat sice jen hodnoty těchto proměnných,ale proces utváření popisu dat by vyžadoval dvě fáze.To by zvýšilo náklady na celé testování.

Tuto nevýhodu odstraňuje metoda zpětného postupu /15/, kdy postupujeme od posledního příkazu k prvnímu.Výhodou je, že nemusíme registrovat hodnoty proměnných. Každý příkaz měnící hodnotu proměnné vystupující v podmínce,přímo realizujeme v této podmínce.

Příkaz	Popis dat
11,9,8	END VSTUP
7	END VSTUP
6,4	END VSTUP AND S-KLIC≠KLIC
10,8	VSTUP=/V1/ AND S-KLIC≠K1
5,4	VSTUP=/V1/ AND S-KLIC≠K1 AND S-KLIC=KLIC
3	VSTUP=/V1/ AND KLIC≠K1 AND KLIC=KLIC
2,1	VSTUP=/V2,V1/ AND K2≠K1 AND K2=K2

Protože postupujeme opačným směrem,musíme výsledný popis upravit změněnou pořadí na tvar VSTUP=/V1,V2/AND K1≠K2.

Shrneme-li navrhovanou testovací strategii, je nutné

- stanovit vývojový diagram k danému programu
- stanovit bázi tohoto vývojového diagramu
- stanovit systém vratev vzhledem k počátku a konci
- provést výběr množiny minimálních cest
- určit popis dat ke každé vybrané cestě
- generovat testovací data na základě jejich popisu
- realizovat program na testovacích datech
- analyzovat výsledky programu

7. Proč testovat?

Jen systematický přístup k prověřování správnosti programu může přinést alespoň částečné úspěchy v podobě spolehlivých programů.Umožní věnovat čas tvůrčí práci a neztrácet ho často nákladnými a zbytečnými opravami chyb,které neodhalil tvůrce díla nýbrž až praxe sama.

Děkuji svým učitelům Doc.RNDrJ.Hořejšovi,CSc a RNDr R. Ochranové za veškerou jejich pomoc.

Literature

/1/ Barzdin J.M.,Bičevskij J.J.,Kalninš A.A.,Postrojenije polnoj sistěmy primerov dlja prověrki korektnosti program, Učennyje zapiski Latvijskogo gos.univ.,210,Riga 1974, 152-187

/2/ Barzdin J.M.,Bičevskij J.J.,Kalninš A.A.,Razrešimyje i něrazrešimyje slučai problemy postrojenija polnoj sistěmy priměrov,Učennyje zapiski Latvijskogo gos.univ.,210, Riga 1974,188-205

/3/ Barzdin J.M.,Kalninš A.A.,Postrojenije polnych sistěm priměrov dlja program rabotajuščich s prjamym metodom dostupa,Učennyje zapiski Latvijskogo gos.univ.,233, Riga 1975,123-154

/4/ Barzdin J.M.,Bičevskij J.J.,Kalninš A.A.,Construction of complete sample system for correctness testing,Lecture Notes in Computer Science 32,Mathematical Foundations of Computer Science 1975,Mariánské Lázně,September 1975,1-12

/5/ Brim L.,Metody dokazování programů,diplomová práce, PřF UJEP Brno,1976

/6/ Brown P.J.,Programming and documenting software projects, Comp.Surveys,6,4,December 1974

/7/ Clarke L.,A System to Generate Test Data and Symbolically Execute Programs,Techn.Rep.Univ.of Colorado,Boulder,February 1975

/8/ Goodenough J.B.,Gerhart S.L.,Toward A Theory of Test Data Selection,Softech,Inc.,Waltham,Mass.

/9/ Gruska J.,Privara I.,Dokazovanie správnosti programov, sborník Sofsem 1976

/10/ Hetzel E.C.,/editor/,Program Test Methods,Prentice-Hall 72

/11/ Hořejš J.,Ladění programů /esej o prevenci,diagnoze a terapii/, sborník Sofsem 1976

/12/ Howden W.,Methodology for the Generation of Program Test Data,IEEE Trans.on Comp.,5, May 1975,554-559

/13/ Howden W.,Reliability of the Path Analysis Testing Strategy, Techn.Rep.Univ.of California,La Jolla,November 1975

/14/ Howden W.,Automatic Case Analysis of Programs,Univ.of California,February 13,1975
/15/ Huang J.C.,An Approach to Program Testing.Comp.Surveys 7, 3/1975/,113-128
/16/ Infotech State of the Art Report,Infotech 20 /sborník/
/17/ King J.C.,A New Approach to Program Testing,Proc.1975 Int.Conf.on Reliable Software,IEEE,Los Angeles 1975
/18/ King J.C.,Symbolic Execution and Program Testing,CACM 19, 7,July 1976,385-394
/19/ Knuth D.E. Stevenson F.R.,Optimal Measurement Points for Program Frequency Counts,BIT 13/1973/,313-322
/20/ Miller E.F.,Melton R.A.,Automated Generation of Testcase Datasets,Program Validation Project General Research Corporation,Santa Barbora,California
/21/ Moravec A., Užití dominance při analýze vývojových diagramů,Fakultní kolo SVK,PřF UJEP Brno,1977
/22/ Ramamoorthy C.V.,Ho S.F.,Testing Large Software with Automated Software Evaluation Systems,Proc.1975 Int.Conf.on Reliable Foftware,IEEE,Los Angeles 1975 382-394
/23/ Tassel van D.,Program Style,Design,Efficiency,Debugging and Testing,Prentice-Hall 1974
/24/ Yourdon E.,Techniques of Program Structure and Design, Prentice-Hall,Englewood Clifs,New Jersey,1975
/25/ Moravec A.,Využití dominance v analýze struktury programu Informačné systémy 2 - 1978
/26/ Gerbrich J., Analýza struktury programu,diplomová práce, PřF UJEP Brno,1978
/27/ Miller E.F.,Program Testing Technology in the 1980s, Proc.of the Conf.on Computing in the 1980´s,1978
/28/ Miller E.F.,Program Testing: Art meets Theory,Computer 77
/29/ Vykydal M.,Testování programů, diplomová práce,PřF UJEP Brno,1977
/30/ Vykydal M., Path Analysis Testing Strategy,Scripta PřF UJEP Brno,1980 /v tisku/