S Y S T E M A T I C K Ý P Ř Í S T U P K T E S T O V Á N Í P R O G R A M Ů (A N A L Ý Z A P R O S T Ř E D K Ů)

Ing. Bronislav Lacko - Výpočetní středisko TOS Kuřim

1. Testování programů - status quo

Význam testování programů pro současnou programátorskou praxi byl u nás zdůrazněn na seminářích v Havířově (2,3) i na jiných akcích (1,6). V zahraniční literatuře byla této problematice věnována samostatná monografická díla (15,16), kde publikace 15 byla vydána u nás ve slovenském jazyce. Soustředěnou pozornost na tuto problematiku lze vysvětlit dvěma důvody:

- programátoři spotřebují daleko větší čas (i náklady) na testování programů než na jejich vlastní vytvoření (viz obr.1)
- testování je dnes jedinou prakticky ekonomickou možností, jak vytvořit spolehlivé programy (viz obr.2).

Přitom prognostické předpovědi říkají, že bez ohledu na velký objem výzkumů a investic do prací v oblasti formálního dokazování správnosti programů je nutno vzít v úvahu skromné dosažené výsledky a předpokládat testování programů ještě na 10 let za základní metodu v této oblasti (13).Těmto závěrům nijak neodpovídá skutečnost, kdy tato problematika je zcela opomíjena při výuce programátorů a často zanedbána i v samotné metodologii programování. Význam testování pro současnou programátorskou praxi nesnižuje skutečnost, že testováním lze prokázat přítomnost určitých chyb a nikoliv jejich neexistenci.

Na seminářích DT ČSVTS Ostrava bylo v Havířově mnoho řečeno o progresivních technikách programování, které přinášejí ve svých důsledcích zvýšení spolehlivosti programů a snížení nákladů a času potřebných na ladění programů. Tyto techniky však nevedou k odstranění potřeby testování programů, vytvořených podle jejich principů. Předkládaný příspěvek si blíže všímá prostředků, které se pro testování používají, předkládá jejich možnou klasifikaci, hodnocení a kritiku, přičemž obsahuje i návrhy na zlepšení testovacích prostředků.

2. Terminologická poznámka

Testováním zde budeme rozumět postupy, které vedou k odhalení existujících chyb v programech a které napomáhají k lokalizaci a k detekci chyby (1). Programové prostředky k tomuto účelu používané budeme označovat jako testovací prostředky. Laděním budeme rozumět opakovaný proces, který kromě testování zahrnuje návrh na korekci defektu a opravu programu (1).

3. Klasifikace testovacích prostředků

V tomto odstavci rozdělíme testovací prostředky do jednotlivých skupin podle různých hledisek. Rozdělení z hlediska manipulace s programem

a) Statické testovací prostředky, které nepředpokládají provádění programu. Podrobují analýze zdrojový text programu, ve kterém se snaží nalézt nepravděpodobné konstrukce nebo nesprávné posloupnosti příkazů (např. čtení neotevřeného souboru)ap.
b) Dynamické testovací prostředky, které předpokládají provedení programu na počítači buď přímo nebo interpretačním způsobem.

Rozdělení dynamických testovacích prostředků z hlediska okamžiku podávání zpráv.

a) Sledovací testovací prostředky, které podávají zprávy bezprostředně s průběhem výpočtu.
b) Post mortem prostředky, které poskytují zprávu o výsledných produktech programu po jeho přirozeném nebo mimořádném ukončení.

Rozdělení z hlediska zásahů do programu

a) Testovací prostředky bez nutnosti zásahu do programu. Těmto prostředkům stačí pouze zdrojový text.
b) Testovací prostředky s pasívními doplňky v textu programu. Tyto doplňky se obvykle chovají z hlediska kompilátoru jako komentáře a není nutno je z programu po ukončeném ladění odstraňovat.
c) Testovací prostředky vyžadující modifikaci zdrojového textu tím, že je nutno doplnit zvláštní testovací příkazy, které se po testování odstraní.

Rozdělení sledovacích testovacích prostředků podle předmětu sledování.

a) Testovací prostředky pro sledování průběhu výpočtu
b) Testovací prostředky pro sledování hodnot proměnných při výpočtu (snímkování)

ad a) Sledovat průběh výpočtu můžeme buď sledováním adres nebo pomocí symbolických jmen, která označují části programu ve zdrojovém textu (návěští, čísla příkazů, čísla řádků zdrojového textu). Můžeme provádět sledování buď krok za krokem nebo při skocích v programu. Sledovat lze činnost v celém programu nebo v jeho vybraných částech, přičemž ohraničení částí sledovaných v programu může být statické nebo se může dynamicky měnit.

ad b) Sledovat můžeme obsahy paměťových buněk, které jsou zadány prostřednictvím adres nebo které jsou označeny symbolickými názvy podle zdrojového textu. Lze sledovat všechny proměnné programu nebo vybrané proměnné. Logickým požadavkem je vázání tisku hodnoty na určitou podmínku.
- proměnná změnila svoji hodnotu vzhledem ke stavu v posledním snímku
- vyskytl se určitý incident (přerušení, přeplnění apod.)
- určitá proměnná nabyla zadané hodnoty (např. čítač)
- bezpodmínečný tisk hodnoty proměnné

Rozdělení podle režimu testování
a) Testování prováděné v dávkovém zpracování
b) Testování prováděné v interaktivním režimu (on line)

Rozdělení podle přítomnosti testovacích prostředků při výpočtu
a) Rezidentní prostředky, které jsou trvale přítomné v průběhu celého výpočtu
b) Testovací prostředky vyvolávané podle potřeby na základě výskytu předepsaného incidentu.

Rozdělení podle způsobu umístění v programu:
a) Interní, které jsou nedělitelnou částí programu
b) Externí, které jsou umístěny mimo program

Rozdělení podle závislosti na programovacím jazyku
a) Prostředky testování, které jsou součástí programovacího jazyku (kompilátoru)
b) Prostředky testování, které jsou nezávislé na použitém programovacím jazyku

Rozdělení kritérií výběru testů

Pro testovací výpočty je potřeba připravit testovací data, což je soubor vstupních dat, se kterými je prováděn testovací výpočet. Za účelem přípravy testovacích dat je nutno zvolit vhodné kritérium pro jejich výběr a připravit resp. vygenerovat tato data. Podle způsobu postupu lze rozdělit výběrová kritéria na dvě velké skupiny (14):

I. skupina: Kritéria závislá na struktuře programu

a) Testování cest - testovací data mají zajistit provedení všech realizovatelných cest v grafu řízení programu alespoň jednou
b) Testování větví - testovací data mají zajistit provedení všech realizovatelných větví
c) Testování příkazů - testovací data mají zajistit provedení všech příkazů (projít všemi uzly grafu).

II. skupina: Kritéria závislá na problému

a) Testování souborem náhodných hodnot ze zvoleného oboru hodnot
b) Testování reprezentanty stanovených tříd vstupních hodnot (např. číslo kladné, záporné a nula)
c) Testování podle tříd výstupních hodnot, kdy soubor testovacích dat musí navodit postupně pro každou třídu výstupních hodnot jednoho reprezentanta

Protože kritéria I. skupiny vycházejí ze struktury navrženého programu, neumožňují ověřit, zda naprogramovaný algoritmus řeší bezchybně zadaný problém, i když prověří důkladně všechny části programu, což je jejich nesporná přednost. Naopak kritéria II. skupiny ověřují správnost naprogramovaného algoritmu, ale program může po testech vykázat chybu v důsledku průchodu větví, která při testu nebyla prověřena.

Rozdělení režimů výpočtů z hlediska působení testovacích prostředků

a) Testovací režim - kdy je prováděno testování programu prostřednictvím zvolených testovacích prostředků
b) Rutinní výpočetní režim - kdy je prováděn běžný výpočet a testovací prostředky nejsou použity nebo je jejich činnost blokována
c) Režim latentního testování - kdy testovací prostředky zdánlivě nepůsobí, ale jejich činnost se projeví v okamžiku určitého incidentu (při splnění předepsané podmínky).

Účelem výše uvedené klasifikace bylo ukázat mnohotvárnost testovacích prostředků. Z uvedené klasifikace jasně vyplývá, že jeden a tentýž testovací prostředek můžeme klasifikovat podle několika různých hledisek. Ví skutečnost, že programátoři vesmon na vědomí rozdělení testovacích prostředků a stanoví si přednosti a zápory jednotlivých skupin, může mít za následek podstatné zvýšení produktivity jejich práce.

Testovací prostředky jsou často navrženy tak, že v sobě slučují několik různých přístupů k testování z hlediska uvedené klasifikace (např. testovací prostředek provádí sledování průběhu výpočtu a současně vypisuje hodnoty proměnných). Rozdělení testovacích prostředků není samoúčelné. Bere v úvahu CO? se testuje a JAK? se testuje. Pro konkrétní programové vybavení určitého počítačového systému je možno u dostupných prostředků stanovit, kromě rozdělení, vhodnost či nevhodnost pro případné problémové situace při testování.

Některé problémy navazující na testování programů

Uplatnění logiky v programování

Všeobecně se uznává, že existují dva stěžejní momenty při vytváření programů pro počítač:

1. Přechod od zadání problému k formulaci algoritmu řešící zadaný problém
2. Přechod od obecně zformulovaného algoritmu k zápisu programu pro vstup do počítače.

Zapomíná se na třetí stěžejní problém - vysvětlit nesprávnou funkci programu, najít příčinu a odstranit ji.

Zatímco v prvních dvou případech jde o syntetický tvůrčí proces, jedná se ve třetím případě spíše o postupy, opírající se o deduktivní metody. Je zarážející, že většina programátorů se nesnaží osvojit si hlouběji znalosti logiky, aby tyto mohli uplatnit při programování. Na problémy s tímto související ukázal z praktické stránky Novák (4). Z hlediska využití logiky v programování jsou inspirující příspěvky na semináři SOFSEM 79 (8) a COBOL 77 (17). Současní programátoři si většinou osvojují minimální, nezbytně nutné znalosti výrokové logiky. Nevěnují však již pozornost výrokově logickému vyplývání, vztahu vyplývání a procesu deduktivního usuzování, logickým úsudkovým formám a pravidlům správného usuzování.

Zásadně opomíjena je celá partie predikátové logiky. Přitom procedurální interpretace predikátové logiky prvního řádu umožňuje chápat predikátovou logiku jako programovací jazyk, zejména pro řešení řady nenumerických problémů (7). Dále je nutno si uvědomit, že tyto otázky se stávají aktuální dnes v důsledku zavádění databázových systémů. Nástrojem teoreticky promyšlených databázových modelů je vždy určitý formální aparát, který se používá k popisu modelovaných skutečností. Nejběžnějším takovým aparátem je predikátová logika prvého řádu, jejíž fragment - teorie relací prvního řádu - je např. podkladem Coddova relačního modelu (19). Opomíjení praktické aplikace logiky u programátorů je paradoxní vzhledem k obsahu jejich práce. Vždyť Vitold Bélévitch v populární encyklopedii o kybernetice nadepsal kapitolu o řízení počítečů "Programování: umění logiky" (viz: Věk kybernetiky, SNTL Praha 66, Překlad VII. a VIII. dílu Encyklopedie des sciences modernes, nakl. René Kistner, Ženeva).

Nesystematický přístup k testování

Přístup většiny programátorů k problematice testování je výstižně popsán v monografii autorů Brown-Sampson (15):
- program se napíše bez přihlédnutí k testování
- program se při prvním zkušebním výpočtu záhadným způsobem zhroutí
- programátor si narychlo najde v příručce kapitolu o testování a rozhodne se použít hned první prostředek, na který při čtení narazí
- testovací prostředek zařadí do programu
- obratem se mu vrátí z provozu počítače neočekávaný stoh potištěného tabelačního papíru
- výsledná reakce programátora : " K čertu, ty testovací pomůcky jsou nanic! Nikdy víc už je nepoužiju!"
- protože však je nutno v práci pokračovat, předloží svůj problém systémovému programátorovi s odůvodněním, že chyba je pravděpodobně v operačním systému, neboť mu špatně pracuje bezchybně skompilovaný program

Programátor by měl nejprve důkladně analyzovat testovací prostředky, které má k dispozici, což je první předpoklad jejich správného použití. Zvážit, jaká použít testovací data a určit

správné chování, které očekává od programu. Zaručit, aby program poskytl analyzovatelný výstup pro pozdější rozbor. Teprve při splnění těchto podmínek by se mělo přistoupit k testovacímu výpočtu. Analýzou obdrženého výsledku je nutno:

a) určit chyby, jejich příčiny a postup jejich odstranění
b) stanovit hypotézu (hypotézy) odchylného chování programu od předpokládaného chování tam, kde nejsme schopni vyřešit problém způsobem od a).

Teprve pak je možno:

a) Rozhodovat o dalším testování
   - rozhodnout o testovacích datech
   - prověřit, zda plánované testovací prostředky splňují očekávané cíle
b) Pečlivě, s ohledem na možné důsledky, opravit program
c) Provést další testovací chod

Při analýze výsledků testovacího chodu lze s výhodou použít metody založené na tabulce popisované v monografii (15) viz obr.3. Mnoho chyb dělají programátoři v přípravě testovacích dat. Programátoři dělají chybu, když prověřují programy na náhodně vybraných souborech. Testovací data je nutno vybrat promyšleně (viz kritéria pro výběr testovacích dat) a s určitým cílem. Další chybou je, že programátoři zaměňují kvantitu a kvalitu testovacích dat. Provede-li se test s 10 000 větami jednoho typu, je to často totéž jako jeden test (pokud se tímto testem nechce prověřit jiná funkce programu). Naopak programátoři, kteří si chystají správně zvláštní testovací data, doplní do souboru dat ihned na začátku i řadu zvláštních případů, které mají ověřit chování programu za mimořádných situací. To často působí potíže při prvních testech. Nejprve by se měla ověřit základní kostra řízení programu, tj. zpracování nejobvyklejších případů a teprve pak přistoupit k prověřování složitějších případů.

Testovací data by měla být k dispozici v úplném rozsahu po celou dobu zkoušek programu. Někdy se určité testy, které proběhly s dobrým výsledkem už neopakují. Dodatečně se pak zjistí, že v důsledku "zavlečených" chyb při opravách jiných chyb program nesprávně funguje v případech, kde předtím fungoval správně. Často nastane situace, kdy se z odzkoušeného programu odstraní testovací prostředky, přičemž se dosud správný program poškodí.

Pokud se po takovém zásahu program neprověří, setkáme se často při rutinních výpočtech se sklamáním. Z toho dále vyplývá, že bychom měli preferovat takové testovací prostředky, které není nutno opravami programu odstraňovat ze zdrojového textu. Programátoři se často nestarají o účelnou minimalizaci testovacích dat. Testují program na rozsáhlých (z hlediska účelnosti nadbytečně velikých) souborech. Následkem je množství potištěného tabelačního papíru, ve kterém se programátor těžko orientuje a často se ztratí přehlédnutím důležité informace. Výše uvedené chyby programátorů dokazují, že mnoho programátorů nevěnuje otázkám testování náležitou pozornost. Staré české přísloví "Jednou řež, dvakrát měř!" platí při testování bezezbytku. Nesprávný postoj programátorů k testování vyplývá, jak již bylo řečeno, zejména z toho, že ani v metodice programování se nevěnuje těmto otázkám pozornost. Metodické pokyny pro programátory by měly otázky testování bezpodmínečně obsahovat (20).

Prevence a testování

Pravděpodobnost, že program v jazyku COBOL a více než 200 řádky procedurální části nebude poprvé fungovat správně, se blíží témeř jistému ději. Přesto, že tuto skutečnost zná každý programátor a denně si tento fakt potvrzuje, setkáváme se opakovaně se stejným nedostatkem: Programy jsou napoprvé psány bez ohledu na fakt, že budou později testovány. Programátoři se v tomto ohledu podobají lidem, kteří sice dobře vědí, že peníze musí získat poctivou prací, přesto však si pro jistotu měsíc co měsíc koupí jeden los čs. státní loterie, protože: "Co kdyby ...!". Tak i většina programátorů optimisticky věří, že se jim přece jednou podaří sestavit program, který bude pracovat hned poprvé správně. Z toho, co bylo již o problematice testování programů v odborné literatuře napsáno a z praxe, vyplývá zásada: Vzít v úvahu problém testování programu hned na začátku jeho návrhu. Znamená to, položit si vedle otázek, které souvisejí s funkční správností, architekturou, jednoduchostí, spolehlivostí, přenositelností, efektivností atd., i otázku: "Jak budu program testovat!?".

Řadu testovacích prostředků je nutno zabudovat do programu hned při jeho psaní a ne dodatečně. To značně usnadní celou etapu testování. Zejména tento přístup zkrátí průběžnou dobu testo-

vání. Ušetří se totiž jednak řada opravných chodů, kdy se testovací prostředky do programů dodatečně doplňují, jednak bývá hned první výstup analyzovatelný z hlediska lokalizace a detekce chyb. Dalším problémem bývá situace, která nastane v okamžiku, kdy se již prověřený program předaný do rutinního provozu s odstraněnými testovacími prostředky havaruje a je nutno ho dodatečně z různých důvodů (dodatečně objevená chyba, dodatečná modifikace programu apod.)testovat. Zde se opakuje často situace popsaná v předchozím odstavci o testování nového programového produktu. Ideálním řešením těchto prpblémů by bylo použití testovacích prostředků bez nutnosti zásahu do programu trvale přítomné v průběhu celého výpočtu. Bohužel toto zdánlivě ideální řešení není bez problémů. Testovací prostředky bez nutnosti zásahu do programu obvykle neřeší celou řadu zvláštních požadavků (výběrové sledování průběhu a výběrové sledování hodnot proměnných, doplnění zvláštních akcí podle pokynů programátora při výskytu incidentu apod.). Rezidentní prostředky pro testování odčerpávají značnou část paměti a někdy i strojový čas. Zdá se, že nejlepším přístupem je kompromisní řešení režimem latentního testování. Činnost testovacích prostředků se řídí nejen prostřednictvím systémových přepínačů (např. v jazyku COBOL: ON SIZE ERROR, USE BEFORE STANDARD ERROR, ON), ale i vnějšími přepínači, zadávanými při vyvolání programu (např. UPSI, OPTION PARAM, PARAM v EXEC příkazu apod.).

Operační systémy z hlediska testování

Všeobecně lze říci, že problematika testování programů byla vzata v úvahu při návrhu koncepce operačních systémů až při vytváření programového vybavení pro počítače 4. generace. Řada inovačních návrhů byla v tomto směru realizována např. u operačních systémů VME/K a VME/B firmy ICL pro počítače 4. generace řady 2900. U počítačů 3. generace jsou obvykle k dispozici pouze ty ladící prostředky, které si vytvořili programátoři realizující operační systém pro své vlastní potřeby při testování systému. Tím lze také vysvětlit fakt, že testovací prostředky počítačů 3. generace vyhovují spíše systémovým programátorům než aplikačním programátorům. Řada konkrétních požadavků na operační systémy z hlediska testování bude uvedena v následujícím odstavci.

Větší pozornost problematice testování by měla být věnována

i při návrhu programovacích jazyků. Ze známých jazyků jen jazyk PL/I. řeší celou řadu problémů, které souvisí s testováním (např. dokonce i simulaci havarijních stavů technického vybavení). Připravované počítačové systémy budou muset závazně počítat s realizací interaktivního ladění. Tento přístup vyžaduje zvláštní technické prostředky, ale i speciální programové vybavení. Zahraniční zkušenosti ukázaly, že prostřednictvím interaktivního ladění lze podstatně zkrátit dobu vývoje programů a zvýšit produktivitu programátorské práce. Bezprostřední dohled programátora nad průběhem testovacího výpočtu a jeho reakce a zásahy do průběhu testování, výrazně dovolují zvýšit kvalitu a produktivitu testování. (Poznamenejme na druhé straně, že interaktivní ladění musí čelit jiným úskalím. Zahraniční prameny uvádějí případ, kdy programátor víc než 20x opakoval stejný test s předpokladem, že chybný výpočet byl způsoben náhodnou technickou poruchou!). Požadavek interaktivního ladění je tak aktuální, že už počítače JSEP 2 by měly být dodávány výhradně s touto možností. Stejně tak počítače řady SMEP 2.

Testovat jen z hlediska funkčních chyb?

Účelem testování programových produktů by měla být komplexní prověrka všech vlastností, které požadovala specifikace. Na tento fakt se často zapomíná zejména z důvodů časové tísně, která charakterizuje většinu implementací. Potíže s odstraňováním množství funkčních chyb obvykle prodlužují dobu ladění, což nepříznivě ovlivňuje (a posunuje!) termín ukončení. Při skluzu termínů se snadno "opomene" testování, které by odhalilo, jak efektivně je využívána operační paměť nebo strojový čas počítače, kapacita sekundárních pamětí apod. Ke zvýšení efektivnosti programového systému nebo programu není většinou zapotřebí zvláštních prostředků software. Stačí otestovat kompilátor a provést kontroly zdrojového textu (10). Každý systém má malý počet kritických míst (1% až 10% v průměru), která mají rozhodující vliv na čerpání strojového času a spotřebu operační paměti. Výkonnost celého systému záleží ve větší míře od výkonnosti těchto částí. Jsou to obvykle následující úseky programů:

- místa, která jsou procházena s největší frekvencí (i když mohou být výpočtově jednoduchá)
- místa, která požadují extrémní počítačové zdroje, i když jsou

aktivována zřídka
- místa, kde se manipuluje s dlouhými řetězci bytů a často se opakují
- místa, kde se provádějí poměrně často složité výpočty.

Kontrolní test programů není potřeba provádět strojově. Programátor může vyhodnotit text programů podle předem doporučených hledisek.

Dělba práce při testování

Hodně problémů v souvislosti s testováním programů přináší neujasněná dělba práce při testování. V našich výpočetních střediscích přežívá názor, že testování programů je výhradně jen záležitostí programátorů. Systémově pojetí koncepce testování však musí vycházet ze zásady, že testování programů se musí stát nedílnou součástí pracovní náplně všech profesí, které se podílejí na tvorbě programového produktu. Již celostátní pokyny pro budování ASŘP zdůrazňují význam kontrolního příkladu pro testování programů v prováděcích projektech. Příkladem by mohla být organizace testování v zahraničí (9), kde se zodpovědnost za testování rozkládá na řadu profesí:
- vedoucí projektu je osobně zodpovědný za kvalitu aplikačního software a jeho oddělení. Neznamená to, že by osobně prováděl testy. Vyvíjí ale testovací standarty a vyžaduje, aby testy proběhly se všemi náležitostmi. Dbá, aby testy byly řádně dokumentovány.
- systémový analytik koordinuje postup testování. Sestavuje zadání testů podle předpisů. Sleduje programátory a uživatele v jejich požadavcích. Provádí závěrečné vyhodnocení testů.
- programátor zodpovídá za otestování modulů, programů a programových systémů a opravu nalezených chyb.
- operátor provádí vlastní testy a současně provádí ověření programového produktu z hlediska obsluhy počítače.
- referent odborného útvaru vypracovává s analytikem detailní plán testování. Připravuje podklady pro testovací data s přesně a předpokládané výsledky pro porovnání s vypočtenými hodnotami.
- vedoucí odborného útvaru je zodpovědný za konečné přijetí nebo zamítnutí hotového programového produktu.

Při komplexním přístupu k testování je testovací postup vy-

víjen souběžně s analýzou a programováním. V ideálním případě je komplexní test připraven okamžitě po sestavení posledního programu, přičemž jsou již k dispozici výsledky dílčích testů. Myšlenka průběžného testování není nová. Byla již důkladně popsána a vyzkoušena jako součást strukturovaného programování (22). Zahraniční prameny (18) často zdůrazňují, že závěrečné komplexní testy by měla provádět skupina specialistů. Nikoliv autoři programů.

Návrhy na zdokonalení některých testovacích prostředků

1. Generátory testovacích dat

Většina současných generátorů testovacích dat je navržena tak, aby dovolila nějakým úsporným způsobem předepsat obsah většího počtu generovaných vět. Zatím žádný ze známých generátorů negeneruje soubor dat podle zadaného testovacího kritéria. Zde by měla být učiněna náprava a dodané programové produkty by měly odrážet pokroky v této oblasti. Řada generátorů neumožňuje jednoduše vytvořit řadu vět, u kterých prpgramátor jednoduchým seznamem konstat určí obsah jednotlivých položek vět. To je nutné v případě, kdy věty spolu tvoří logicky svázané skupiny, které simulují komplexní případy z praxe.

2. Přesnější indikace systémových chyb a reakce programátora na ně

Bohužel ještě existuje mnoho takových operačních systémů, které se chovají tak, že při výskytu chyby (např. nesouhlas velikosti bloku na MP s deklarovanou velikostí věty v programu) ukončí svoji práci s lakonickým oznámením ABNORMAL TERMINATION s následným uvedením hexadecimálního obsahu čítače instrukcí. Lokalizací místa paměti a následného zjištění, že adresa se nachází uvnitř standardního modulu pro manipulaci s magnetickými páskami, lze nyní zkoumat řadu příčin (získaných zkušenostmi, tj. nikde v dokumentaci explicitně neuvedených!), které takovou reakci mohou zavinit a zkoušet, která z nich nastala. S uvedeným příkladem si nezadá jiný, kdy výpočet je přerušen s oznámením, že na určité hexadecimální adrese se objevila nepovolená kombinace bitů pro zpracování. (Zasvěcený programátor ví, že uvedená adresa je odlišná od adresy proměnné, jejíž obsah chybu způsobil). Je přirozený požadavek, aby indikace chyb nejen oznamovala skutečnost, že došlo k nějaké chybě, ale aby byla signalizována chyba co nejpřesněji, s informacemi umožňující její lokalizaci a detekci, a aby tyto

informace byly hlášeny prostřednictvím symbolických objektů zdrojového textu jazyka, ve kterém byl program napsán.

Staronová forma pro jazyk COBOL (viz ANS X3.23 - 1974) zavádí nový stavebnicový segment DEBUG, který poskytuje prostředky pro ladění, zejména sledování hodnot údajů a průchodu cesty, kterou probíhá zpracování programu. Prostřednictvím příkazu USE FOR DEBUGGING se provádí sledování tak, že získané informace se ukládají do speciálního registru označeného jako DEBUG-ITEM. V tomto registru lze identifikovat řádek zdrojového textu, symbolický název objektu (paragraf, proměnné, název souboru) podle okolností a případně obsah proměnných. Navíc je možno znakem "D" v 7. sloupci formuláře označit řádky, které se kompilují a provádějí pouze v případě, že je uvedena klauzule WITH DEBUGGING MODE v paragrafu SOURCE-COMPUTER. Je opravdu škoda, že na tato nová doporučení nijak nereagují tvůrci kompilátorů jazyka COBOL u počítačů řady JSEP a SMEP. Programátor by měl mít k dispozici i v jazyku vyšší úrovně možnost převzít na sebe ošetření všech nebo vybraných chyb, které odhalí operační systém a předepsat jak postup při jejich výskytu, tak i způsob pokračování po chybě. S tím také souvisí problém, aby programátor dostal přesnou zprávu popisující chybu. Takové prostředky lze i dodatečně vytvořit v rámci výpočetního střediska. V našem výpočetním středisku byl aplikován modul CHYBA (21), který takovou problematiku řeší. V důsledku použití modulu klesl počet testovacích průchodů až o 50%.

Protokolování výpočtu

Přirozeným požadavkem z hlediska uživatele je, aby operační systém poskytl kromě explicitně požadovaných výstupních sestav i protokol o průběhu výpočtu. Protokol o průběhu výpočtu může být velmi efektivním nástrojem při testování programu. Protokol by měl být pro každou práci (job) vytištěn v celistvosti na tiskárně. Měl by obsahovat:

- opis zadaných řídících příkazů pro kontrolu, jak byl výpočet skutečně zadán
- systémové výpisy opatřené časovými údaji (začátek a ukončení práce, start a ukončení programů apod.)
- hlášení chyb
- příkazy operátora k těžící úloze
- výpisy programátora z programu do protokolu a na konzolu operá-

tora
- zprávy a indikace o použitých paměťových nosičích (disky,pásky)
- zprávy o průběhu vstup/výstupních operací (počty vět).

Komplexní řešení této problematiky poskytuje snad jen operační systém OS, který navíc dává možnost předepsat úroveň podrobnosti protokolovaných skutečností. Ostatní, zejména tuzemské operační systémy, jsou poznamenány celou řadou nedokonalostí, které ztěžují práci programátora a provozu počítače. Zejména se zapomíná na skutečnost, že programátor může často chtít vypsat informaci do protokolu, ale nikoliv do konzolu operátora. Podobně se zapomíná, že protokolovací výpisy se vyžadují nejen v období testování, ale i při rutinných výpočtech, a je tudíž neprakticé je prokládat mezi uživatelskými sestavami.

Ekonomické hledisko

Tvůrci operačních systémů často spoléhají na skutečnost, že neplatí náklady, které spotřebuje běžící operační systém u jednotlivých uživatelů. Proto si usnadňují svoji práci na úkor uživatelů. Jako příklad je možno uvést sledování průběhu programu v jazyku COBOL, kdy po příkazu READY TRACE se tisknou názvy prováděných paragrafů - vždy jeden název na jeden řádek. Přitom je známo, že maximální délka názvu paragrafu (sekce) je 30 znaků a standartní systémový výstup je 127 znaků na řádku. Podobný případ nastává v okamžiku, kdy je v programu zapsán příkaz cyklu pomocí příkazu PERFORM v jazyku COBOL. V příkazu nelze citovat opakované příkazy. Ty musí být popsány v samostatném paragrafu. Fráze: PERFORM NULUJ-TAB 1ØØ TIMES způsobí 100 nezajímavých řádků s textem NULUJ-TAB. Přitom by stačilo testovat shodnost po sobě jdoucích názvů a uvést informaci: NULUJ-TAB a 99x znak*. Výše uvedené a jiné nedokonalosti vedou ke stohu potištěného papíru a je možné je vysvětlit jedině tím, že tvůrci programového vybavení si nikdy nedali práci s vyhodnocením testovacích prostředků z praktického hlediska.

Problém minimalizace výstupu by měl být řešen i možností omezit dobu provádění testovacího výpočtu. Běžně dostupné limitování časem (reálným nebo CPU-TIME) není nejvhodnější, protože obvykle chybí podklady pro jeho objektivní exaktní stanovení, když chování programu není ještě zcela známo. Přijatelnější se je-

ví limit na výstupní počet stran resp. řádků. Takové limity lze snadno předepsat u operačních systémů, které pracují metodou SPOOLINGU. Určitý problém přináší požadavek programátorů, že je nezajímá počáteční průběh výpočtu, ale např. posledních dvě strany protokolu o sledování výpočtu než se výpočet ukončil z jiné příčiny (supervisor odhalil programovou chybu). Docházíme tím k požadavku dynamického překrývání výsledků testování, aby byl k dispozici určitý nevelký soubor informací k sedanému časovému okamžiku přepisován systémem "round-robin".

V každém případě by však měl být testovací chod zabezpečen prostřednictvím údaje CPU-TIME-LIMIT proti zacyklení.

Nastavení testovacího režimu

Pro praktické použití je výhodné, když operační systém má zabudovány standardizované prostředky pro nastavení testovacího režimu. Většina operačních systémů takové prostředky nemá a programátoři si musí zadání a indikaci tohoto režimu sami naprogramovat. V takovém případě je nutné standardizovat určité řešení v celém výpočetním středisku, aby každý program neměl způsob nastavení testovacího režimu jiný. Rovněž je nutno zakázat programátorům slučovat zadávání testovacího režimu se zadáváním jiných parametrů. Programátor by měl mít možnost rozlišit případ, kdy chce v rámci jednoho výpočtu (jobu) zadat tento režim pro určitý program nebo pro všechny volané programy. Testovací režim by mělo být možno zadat i z konsoly operátora bez nutnosti úpravy řídících příkazů.

Závěr

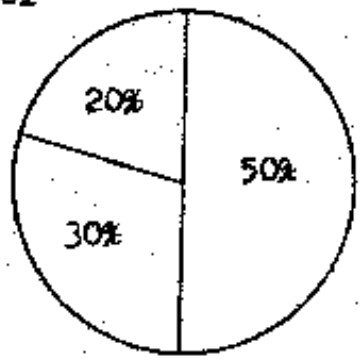
Předložený příspěvek chtěl sdělit některé praktické otázky, které se týkají testování programů a navázat tak na referáty minulých seminářů, které se touto problematikou zabývaly. Pro oblast testování vznikl dnes naléhavý požadavek sloučit teorii a praxi. Teoretické přístupy, které rozpracovávají nové myšlenky testování programů jsou v malé míře dopracovávány do podoby prakticky použitelných programových produktů. Na druhé straně implementované testovací prostředky nástrojů poplatné zastaralým praktikám minulých generací počítačů a nijak neodrážejí pokroky, kterých se na tomto poli počítačové vědy dosáhlo.

Problematika testování se musí stát v každém výpočetním středisku stejně důležitým problémem jako problematika vypracovávání nových programů. Tvorbu programů a testování prpgramů nelze od sebe ani oddělit ani jedno nebo druhé zanedbat.

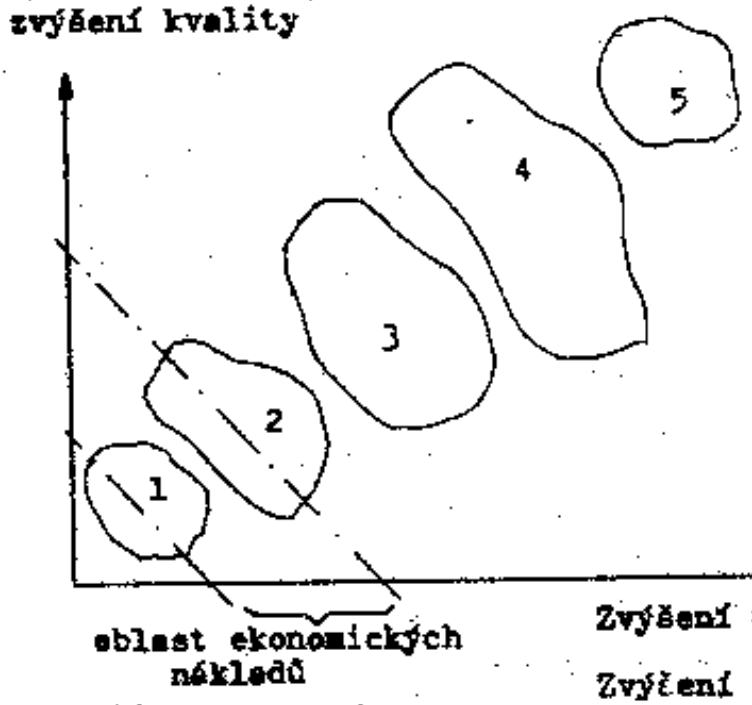
Seznam literatury

1. Hořejš J.: Ladění programů. Sborník ze semináře SOFSEM 76, VVS Bratislava 1976, s. 4-12
2. Lacko B.: Spolehlivost programů. Sborník ze semináře PROGRAMOVÁNÍ 77, DT ČSVTS Ostrava 1977, s. 15-43
3. Vykydal M.: Testování programů. Sborník ze semináře PROGRAMOVÁNÍ 80, DT ČSVTS Ostrava, s. 102-117
4. Novák D.: Jak hledat chyby v programech. Mechanizace a automatizace administrativy, roč. XVI, 1976, č. 10, s. 389-341
5. Tvrdík J.: Typy úsudku v programování. Sborník ze semináře PROGRAMOVÁNÍ 81, DT ČSVTS Ostrava, s. 103-111
6. Lampert M.: Testovanie programovýoh systémov,Sborník ze semináře SOFASR 78, díl III.,s. 125.132
7. Mikulecký P.: Problémy programovania v predikátovom počte. Informačné systémy r. 7, 1978,č. 2, s. 123-134
8. Hájek P., Kolářek P.,Kůrka P.: Systémy dynamické logiky v programování. Sborník SOFSEM 79. VVS Bratislava 1979
9. Schaefer L.: Improving System Testing. Datamation 23, 1977, č. 9 s. 115-117
10. Jolie P.: Improving Performance the Easy Way. Datamation, 23, 1977, č. 4, s. 135-137
11. Cooke L.: Express Testing. Datamation 24, 1978, č. 9 s. 219-222
12. Miller E.: Program Testing Tools and their Use. Software Reliability, 1977, s. 183-216
13. Miller E.: Testing for Software Reliability. Software Reliability, 1977, s. 217-241
14. Frybertová-Pokorná J.: Testovací kritéria. Diplomová práce KAM PF UJEP, Brno 1981
15. Brown A.R., Sampson W.A.: Ladenie programov. ALFA Bratislava 1981 (překlad z angličtiny)
16. Debugging Techniques in Large System. Ed. Rustin R. Courant Comp. Sci. Symp. 1, Prentice Hall 1971
17. Vítek M.: Logika a programování. Sborník referátů s. 5. se-

mináře "Používání jazyka COBOL", DT ČVTS Pardubice, 1977, s. 115-124
18. Myers G.: Software Reliability (Principles and Practices). John Wiley Sons, New York 1976 (Ruský překlad: Mir, Moskva 1980)
19. Materna P.: Úvod do jednoduché teorie typů. Sborník se semináře DATASEM 81, DT ČVTS Praha 1981, s. 32-40
20. Lacko B.: Metodické pokyny pro analytiky a programátory (SOFTIK č. 23), Interní publikace TOS Kuřim 1981 (II.vyd.)
21. Katolický M.: Modul CHYBA. Popis ZN 34/81
22. Baker F., Mills H.: Chief Programmer Teams. Datamation, 19, 1973, č. 12, s. 58 - 61



OBR. 1



1 Testování programů
2 Testování podporované strukturovaným programováním
3 Formální prověrka programů
4 Symbolické vyhodnocení správnosti
5 Dokazování správnosti

?	Vyskytuje se	Nevyskytuje se
Co?		
Kdy?		
Kde?		
V jakém rozsahu?		

Modifikace programu
Tvorba programu
Vytvoření testovacích dat
Program
Testovací data
TESTOVÁNÍ
Výsledky
Analýza výsledků
STOP
Další
Testovací metoda

Obr. 4

Příloha: Metodické pokyny pro testování programů

1. Zkontroluj text programu po formální, syntaktické i sémantické stránce, než ho předáš do provozu výpočetního střediska. Zaměř se především na fatální chyby, které způsobí přerušení kompilace.
2. Použij simulace výpočtu k ověření správnosti algoritmu a ke snížení spotřeby strojového času při ladění.
3. Seznam se dobře s dostupnými testovacími prostředky, abys z nich mohl vybrat optimální nástroj pro řešení konkrétní situace při testování.
4. Testovací postupy a testovací data vyvíjej souběžně s analýzou a programováním.
5. Doplň kontrolní příklad /testovací data/ o situace, které by dokonale ověřily správnou funkci programu z hlediska programového řešení. Prostřednictvím testovacích dat by měly být ověřeny všechny části programu.
6. Pro testování se musí použít speciálně připravených dat, pro která známe předem správné výsledky /nikoliv rozsáhlých, nahodilých souborů z praxe!/.
   Připrav nejprve relativně malý soubor dat, který prověří jen základní funkce programu. Teprve v dalších testech rozšiř soubor testovacích dat pro komplexní testy.
7. Testování se musí provádět se skutečnými programy, se kterými se bude provádět později rutinní zpracování.
8. Uschovej připravená testovací data po celou dobu, nezbytnou k ověřování programu.
9. Pečlivě stanov cíl a strategii testování. Uvaž, zda zde zvolený testovací prostředek /prostředky/ umožňuje efektivně dosáhnout žádaných výsledků.
   Snaž se minimalizovat objem získaných informací z testovacího chodu na nezbytně potřebné množství.
10. Každý zásah do programu, který děláš pro odstranění chyby na základě provedených testů, prověř, zda jím nenarušuješ syntaktickou, sémantickou a logickou správnost programu.
11. Ověř funkci programu v rámci celého programového systému.

12. Dohodni postup ověřování správné funkce programu /programového systému/ duplicitním zpracováním.
13. V průběhu testování si připrav podklady pro stanovení časových odhadů délky trvání výpočtu, čerpání operační paměti, potřeby sekundárních pamětí apod.
14. Pamatuj, že testováním lze dokázat existenci chyb, nikoliv jejich nepřítomnost v programu.
15. Jestliže program nevykazuje očekávané chování, použij následující postup:
    - snaž se získat analyzovatelný výstup promyšleně sestavenými testovacími údaji a správnou volbou testovacích prostředků
    - definuj očekávanou funkci programu ze znalosti testovacích dat a struktury programu
    - pomocí testovacích prostředků lokalizuj, kde se hodnoty proměnných nebo průběh výpočtu odchyluje od předpokladů
    - anylýzou obdrženého výsledku testovacího chodu:
      a/ Urči chyby, jejich příčiny a postup odstranění.
      b/ Sestav hypotézu důvodu odchylného chování programu od předpokládaného, kde nejsi schopen identifikovat chybu.
      K tomu použij tabulku:

?	VYSKYTUJE SE	NEVYSKYTUJE SE
CO		
KDY		
KDE		
ROZSAH		

    - připrav další testovací chod:
      a/ zpřesni testovací data
      b/ přístup k detailnímu sledování vymezených úseků programu pro kontrolu provedených zásahů a potvrzení vytvořených hypotéz.