P R O S T Ř E D Í P R O T V O R B U P R O G R A M Ů

Ing. Branislav Lacko, CSc
Výpočetní středisko, TOS Kuřim

Motto: E.B. Daly: Vývoj software vyžaduje odpovídající technologii, řízení a organizační strukturu.

1. Tvorba programů - status quo

Způsob tvorby programového vybavení pro samočinné číslicové počítače prošel složitou cestou vývoje, který v současné době vyústil do několika charakteristických trendů:

- vypracování izolovaných jednotlivých programů nahradila realizace velkých programových systémů, zahrnujících stovky integrovaných programů
- individuální programování se přeměnilo v týmovou práci, vyžadující složité metody řízení velkých projektů
- příležitostné vypracování programů přerostlo v jejich nepřetržitou proudovou výrobu
- amatérský přístup k navrhování programů byl postupně opouštěn, až se dosáhlo vytvoření specializované odbornosti, které používá přiměřeně dokonalých metod.

Světový prudký vzrůst výroby počítačů přinesl s sebou vzápětí celospolečenskou objednávku potřebného programového vybavení, kterou se nepodařilo dodnes uspokojit.

Vzniklou situaci, označovanou někdy termínem "programová krize", komplikují i další přirozené požadavky, které jsou vznášeny v souvislosti s vypracováním programů. Zejména je nutno připomenout následující požadavky:

- snížení nákladů na vývoj programového vybavení
- zkrácení doby vývoje programů
- zajištění vysoké spolehlivosti programů
- zvýšení užitkové hodnoty vypracovaných programů.

Zatímco první dva požadavky souvisí s problémem zvýšení produktivity programátorské práce, druhé dva požadavky tvoří jádro metodiky programování.

Nastalé situace vyprovokovala řadu návrhů, jak řešit problémy,

které přináší současná programátorské praxe, jež v určitých podmínkách přinesly pozoruhodné výsledky např. několikanásobné snížení výskytu chyb /1 odhalená chyba na 10 000 řádků/, nebo snížení nákladů na chyby v programech /z 3600 dolarů na 3 dolary za chybu/, zvýšení produktivity programátorské práce /desetinásobné zvýšení produktivity/. Přitom tyto návrhy vycházejí většinou z rozsáhlé analýzy současného stavu v programování a z rozboru faktorů, které ovlivňují konečný výsledek při tvorbě programového vybavení.

Krátký přehled těchto nejdůležitějších faktorů uvádí E. Daly:

- velikost vývoje, daná počtem vypracovávaných programů
- počet projektů, které probíhají současně
- rozsah vývojových prací daný rozličnými typy aktivit
- prostředí, ve kterém se programy tvoří /podnikové výpočetní středisko, specializované organizace na tvorbu programového vybavení, použité nástroje a techniky podporující tvorbu programů/
- úroveň řízení
- vliv lidského činitele.

Jak vyplývá z publikovaných zpráv o různých experimentech z této problémové oblasti, převážná většina řešitelů se zaměřuje na vypracování speciálních programovacích nástrojů podporujících tvorbu programů. Zdá se, že je to onen příslovečný hlavní článek, který integruje řešení všech dalších, souvisejících problémů.

Nezbytnost potřeby nástrojů, podporujících tvorbu programového vybavení, byla pociťována již v samých počátcích používání počítačů. Vývoj těchto prostředků začal u jazyků symbolických adres, programů vypisujících obsah paměti /DUMP/ a programů sledujících provádění instrukcí /TRACE/ a pokračoval vývojem programovacích jazyků vyšší úrovně, speciálními testovacími programy a generátory testovacích dat.

Automatizované řešení těchto problémů se v podstatě ubíralo dvěma cestami, jak uvádějí Král a Demner:

- jazykový přístup k řešení se vyznačuje rozšiřováním vybraného jazyka o vhodné prostředky, které automatizují další prvky programovacího procesu
- integrační přístup k řešení se snaží skloubit vzájemnou komunikaci a jednotným způsobem ovládání potřebné množství re-

lativně samostatných nástrojů, které automatizují různé činnosti programátora.

Podle potřeby jsou v praxi tyto dva přístupy vhodně kombinovány.

Prostředky, navrhované v současné době pro podporu tvorby programů, jsou charakteristické zejména tím, že se klade zvýšený důraz na zajištění vysoké spolehlivosti finálních programových produktů. Pro zajištění této vlastnosti vytvořených programů se ukázalo nezbytné opřít koncepci navrhovaných podpůrných prostředků pro tvorbu programů o dobře propracovanou metodologii programování, které je v těchto systémech považována vždy za výchozí základnu.

Přes všechnu snahu podpořit tvorbu programového vybavení všemi dostupnými prostředky, ukazuje bezprostřední situace v současném programovém vybavení počítačů řadu skutečností, které zaslouží kritiku.

2. Specifikace požadavků na programy

Předpokladem pro úspěšnou realizaci programů je přesné specifikace vlastností programových produktů.

Programátorské praxe potvrdila, že na realizaci programů je nutno myslet již při analýze. Cíle, které má programový systém splňovat, musí být stanoveny způsobem, založeným na systémovém přístupu. Sebelepší postup při programování nezachrání špatně provedenou analýzu. Jak uvádějí autoři Soi a Gopal, téměř 60% chyb při realizaci programového vybavení je zaviněno špatným analytickým zadáním.

Vztah mezi systémovou analýzou a programováním při projektování počítačových aplikací se mění. Adamec uvádí pro každou generaci počítačů charakteristický určitý způsob komunikace mezi počítačovými specialisty a uživateli, charakter zpracovávaných úloh a vztah programových produktů k počítačovému okolí. Přitom nejstarší pragmatické koncepční schema je postupně nahrazováno metodickým schematem a vývoj směřuje k integrovanému koncepčnímu schematu postupu tvorby programových produktů.

Pragmatické koncepční schema vychází z klasické dělby etap při tvorbě programu, jak ukazuje obr. 1.

Metodické koncepční schema vychází z dekompozice celého procesu tvorby programu na postupné dílčí operace a vazby mezi nimi. Přitom se dodržuje zásada, že k následující operaci může dojít až po

provedení všech činností v předcházející etapě. Množina operací bývá různá podle toho, zda při dekompozici převládalo hledisko tvorby programu nebo hledisko tvorby dokumentace.

Perspektivním se jeví rozpracování metodiky programování podle integrovaného schematu, které vychází z fungování konečného automatu obr. 2. Aplikaci myšlenek z oblasti konečných automatů doporučují Selter a Chitten.

Na základě některého z výše uvedených schemat jsou již vypracovány rozličné metody tvorby programového vybavení. Autoři Peters a Tripp provedli srovnání pěti metod:

- Strukturovaného návrhu /autoři Constantine-Yourdon-Myers/
- Jacksonovy metody
- Logické konstrukce programu /navržené J. Warnierem z Francie/
- Metody MSR /Meta Stepwise Refinement/ od Ledgarda a Schneidermana
- Metody HOS /Higher Order Software/ od Hamiltona a Zeldina.

Analýzou vlastností jednotlivých metod dospěli k názoru, že žádná ze zkoumaných metod neřeší komplexně všechny problémy při tvorbě programového vybavení. Toto zjištění je vedlo k vytvoření vlastního koncepčního schematu pro tvorbu programového vybavení. Tvoří ho trojrozměrný prostor s osami TIME, LOGIC a FORMALISM. Na osách jsou vyděleny jednotlivé úseky viz obr. 3. V tomto prostoru lze vytvořit těleso, které je složeno z 6 x 4 x 5 = 120 komponent, představující procesy při vývoji software.

Důležitým prvkem v každé metodě je postup, kterým se v procesu analýzy a programování odstraňují chybné závěry. Přibáň rozlišuje:

- Profylaktický postup, založený na apriorní analýze procesu návrhu, na základě které se stanoví hierarchie úloh tak, aby hloubka konfliktních situací /počet úrovní, přes které se uzavírá smyčka zpětné vazby/ byl minimální a konflikty se daly řešit v mezích úrovně, v níž byly zjištěny a řešeny.
- Postpriorní postup, který zjišťuje prameny konfliktu a projektování vrací se na tu úroveň, kde konflikt vznikl. Pak se postup opakuje.

Uživatelé většinou specifikují jen funkční požadavky na požadované programové vybavení. Analýza musí tyto požadavky transformovat na specifikaci datových struktur a specifikaci programů, které jsou pak realizovány v procesu programování. Selter doporučuje pro tyto

účely použít metodu, založenou na teorii konečných automatů /tím navazuje na myšlenku Chittena/ využívající binárních matic při stanovení struktury dat, která vyplývá z funkčních stavů programového vybavení. Pro zvýšení efektivnosti metody automatické kontroly elementů dekompozice /úplnost, kontradikce apod./ byl jím navržen jazyk SSL, který má být ověřen v praxi.

Praktické příklady využití konečných automatů v programování uvádí Hořejš pro testování programů nebo Šmejkal.

Velký význam pro vývoj metodiky programování mělo odhalení důležitosti vztahu mezi pojmy algoritmus a datová struktura. Prof. N. Wirth vyjádřil tento vztah stručně zápisem:

program = algoritmus + datové struktury

Hoare považuje dokonce pojem datové struktury ze primární a uvádí, že člověk nejprve vyděluje z reálného světa objekty svých úvah a teprve pak uvažuje o operacích, které s nimi chce provést.

Poměrně nízká spolehlivost programových produktů je mimo jiné zapříčiněna skutečností, že teprve finální produkt celého vývoje - programový systém - je podrobován zkouškám. Přitom platí obecná zásada, že čím později se chyba odhalí, tím větší jsou náklady na její odstranění. Vznikl přirozený požadavek soustavně provádět verifikaci každého kroku od formulace zadání až k předání programů do rutinního provozu. Konkrétně byla taková metodika vypracována firmou Computer Sciences Corp. Základem metodiky je rozklad funkčních požadavků na řadu tzv. dekompozičních elementů /DE/. Každý element zobrazuje:

1/ proces, který se provádí při realizaci zadané funkce
2/ vstupní data nebo podněty
3/ výstupní data nebo reakce

Z jednotlivých elementů se sestrojují tzv. grafy řízení systému /SVD/ - obr. 4b.

Pro zachycení vlastností každého DE elementu slouží speciálně uspořádaný formulář s pěti poli /viz obr. 4a/. Obsah polí tvoří:

1/ vstupní data, řídící zprávy, příkazy operátora
2/ identifikace DE
3/ popis funkce
4/ výstupní data, zprávy pro operátora, jiné reakce
5/ odkaz na zdroj požadavků pro tuto funkci.

Na základě DE a CVD se sestrojí binární matice vztahů N x N, kde N je počet elementů DE_i, kde prvek matice m /i,j/ = 1 je-li DE_i bezprostředně svázán s DE_j. Dále binární matice dostupnosti M x M, kde M = N, u které prvek m /i,j/ = 1 existuje-li cesta z DE_i k DE_j. Tyto matice se pak používají v analýze požadavků programového systému a zejména pro verifikaci postupu při návrhu programového systému. Uvádí se, že při experimentu, který zahrnoval 5 velkých programových systémů, bylo odhaleno 75 - 85% chyb již ve stadiu specifikací funkcí vyvíjených systémů.

Metodický koncept průběžného testování jak ho navrhuje A. S. Fujii zachycuje obr. 5.

Řada prací a snah v oblasti metodiky programování je zaměřena na řešení problému - formalizovat popisné prostředky, které se používají pro zachycení postupných specifikací produktů v každém vývojovém kroku.

Vyřešení úplné, bezrozporné a přesné specifikace požadavků je považováno za klíčový problém na cestě k efektivnímu vývoji programového vybavení.

Z řady pokusů uveďme metodu VDM /Vienna Development Method/, která představuje formální metody tvorby programů. Notační systém VDM se nazývá META - IV. Část VDM, která se zabývá tvorbou programů předpokládá, že se nejprve vytvoří formální model programu, který se má implementovat. Vývoj programu se skládá podle VDM. z fáze S_0, ve které se vytvoří úplné formální specifikace. Za fází S_0 následuje obvykle několik fází S_i, ve kterých se vytvořené specifikace transformují na resp. směrem k realizaci. Smyslem formálního návrhu je otestovat alternativní návrhy a vybrat nejvhodnější návrh. Dále pak má formální popis sloužit:

1. Jako podklad pro realizaci
2. Jako základ při vytváření uživatelské příručky
3. K dokazání správnosti implementace
4. Pro ověření správnosti užití software.

Metoda VDM se zatím používá jen v oblasti systémového programového vybavení. Pro aplikační programy je zatím nevhodná. Její formální aparát vyžaduje speciální matematickou přípravu, která zatím není u analytiků a programátorů běžná.

3. Věnovat pozornost lidskému činiteli

Metody, které se opírají o metodiku analýzy a programování, mají pro praktický návrh programovacích systémů velký význam. Na druhé straně Peters a Tripp správně poznamenávají, že jsou to konec konců analytici a programátoři, kteří vyvíjejí programové vybavení a nikoliv programovací metody. V procesu tvorby velkých programovacích systémů mají proto neméně důležitou roli řízení /plánování, dělbe práce, kontrola/, organizace pracovního týmu a lidský činitel - tedy "netechnické" faktory /viz Brooks: Mythical Man-Month/.

Organizaci práce při tvorbě programového vybavení je v poslední době věnována zvýšená pozornost /např. tým hlavního programátora navržený Bakerem a Millsem, maticová struktura řešitelských týmů v projektu BETA/. Realizace těchto myšlenek obvykle nepřináší požadavky na implementování specifického programového vybavení. Obvykle je pouze vyžadována určitá problémově orientovaná organizace prostředků pro manipulaci s knihovnou a textové editory.

Zdůraznění požadavku lepšího řízení programátorských týmů však přineslo požadavek na dokonalejší zajištění informací o postupu prací na jednotlivých programových komponentách a na požadavek automatického vedení evidence vykonaných prací a hotových úkolů. Řada implementovaných programovacích systémů má u každého elementu speciální stavový vektor, který udává rozpracovanost příslušného prvku /např. údaje o zdrojovém textu: částečně vytvořen, zcela zkompletován, zkompilován s odhalenými chybami, zkompilován bez chyb a vytvořením modulu/. Stavy bývají opatřeny někdy navíc datumem.

Často jsou vytvářeny vektory dva: vektor plánovací a vektor skutečného průběhu. Jsou-li k dispozici takové údaje a jsou-li pravidelně aktualizovány, je možno zajistit na požádání řadu užitečných přehledů, které usnadní vedoucím týmů a projektů jejich činnost. Zdá se, že pro úspěšnou realizaci velkých programových systémů je takovéto nahrazení ruční evidenční práce automatizovaným systémem nejen otázkou racionalizace administrativní práce, ale zejména základem pro poskytování přesných a rychlých informací, které jsou nezbytné pro účinné řízení.

Poznamenejme, že výše uvedený systém lze rozhatit o řadu dalších užitečných informací /počet příkazů, počet proměnných apod./, ve vazbě na identifikaci programátora zajišťovat podklady pro kvalifikované úvahy při plánování projektních prací a údaje pro kontrolu programátorské produktivity.

I když otázky produktivity programátorské práce nejsou dosud uspokojivě zodpovězeny, již sám fakt, že programátoři vědí o existenci možnosti kontroly jejich práce, má za následek ve většině případů vzrůst produktivity programování. Pokud by se programovací systém realizoval v budoucnu na bázi exaktního modelu programovacích prací, dá se očekávat, že by význam prognostické funkce využitelné pro plánování podstatně vzrostl. To, že se těmto otázkám věnuje v současných programovacích systémech jen minimální nebo žádná pozornost, pouze potvrzuje ryze pragmatický přístup při stanovení jejich koncepce.

4. Uživatelské hledisko

Vývoj prostředků pro podporu programování je nutno vidět i z hlediska narůstající složitosti v jejich používání.

Počítače I. generace měly v podstatě jen několik primitivních prostředků, které ulehčovaly základní etapy ladění programu.

Teprve u počítačů II. generace se objevily komponenty, speciálně navržené pro ladění programů. Obvykle představovaly relativně samostatné programy, které tvořily malou část z celkového objemu standardního programového vybavení. Proto se je uživatelé naučili ovládat za krátkou dobu.

U počítačů III. generace došlo k nárůstu kvantity, nikoliv kvality programového vybavení pro tuto oblast. Setkáváme se zde s velkým počtem programů, které dělíme obvykle do několika skupin:

- překladače programovacích jazyků
- programy pro údržbu knihoven /ukládání do knihoven, opravy elementů v knihovnách, opravy obsahu knihoven, výpisy obsahu knihoven, výpisy elementů knihoven, rušení elementů v knihovnách, manipulace s knihovnami/
- testovací programy /sledování výpočtu, výpisy obsahu paměti, generování testovacích dat/
- sestavovací a zaváděcí programy
- jiné programy /makroprocesory, programy na typografickou úpravu zdrojových textů apj./

Některé z uvedených programů závisí na programovacím jazyku, jiné jsou vytvořeny speciálně pro určitý programovací jazyk. Přitom každý program je navržen tak, že poskytuje celou řadu možností. Nejsou řídké případy, kdy pro určitou akci přípravy programů existuje několik různých způsobů, kterými je možno ji provést.

Dokumentace celé takové množiny programů spočívá obvykle v prosté specifikaci funkcí jednotlivých programů a prostředků pro jejich ovládání, ale i ta čítá několik stovek až tisíc listů.

V tomto okamžiku je nutno se vžít do postavení uživatele, který obdrží s počítačem programové vybavení tohoto druhu.

Jak má bez zkušeností využít předložené silné potenciální možnosti? Jaký postup má v přípravě programů zvolit, aby dosáhl optimálních výsledků? Kolik času a nákladů ho stojí jeho začátečnické chyby a postupné získávání zkušeností!? Kolik výpočetních středisek postupuje tak, že problém reorganizace knihovny /likvidace nepotřebných verzí, uspořádání aktuálních verzí a úzkých archivních verzí/ začne řešit v okamžiku, kdy operační systém poskytne poprvé zprávu, že v knihovně není volné místo! Přitom vhodný způsob organizace práce s knihovnou se začne řešit až když po delší době ztrátový čas převyšuje čas produktivní!

Poznamenejme, že většina operačních systémů nemá funkci reorganizace knihovny řešenou samostatným programem, ale že ji uživatel musí dosáhnout kombinací řady standardních programů.

Doporučení, jak postupovat při řešení otázek tvorby programů, by se měla stát nedílnou součástí instalačních pokynů, dodávaných se systémem.

Je možno bez nadsázky říci, že naprostý nedostatek těchto informací patří mezi hlavní důvody, proč počáteční období nasazování počítačů u nás trvá tak dlouho.

V zahraničí se na tuto oblast pomoci uživatelům specializuje řada poradenských firem, které modifikují dodané programové vybavení z hlediska efektivního používání při ladění programů.

Práce s knihovnami představuje nezbytný prvek v současné přípravě programů s nikoliv možný přepych. Tomu by měly odpovídat i implicitní volby vstupů do kompilátorů resp. výstupů z kompilátorů a dalších programů. Často se však setkáváme s pravým opakem, kdy operační systém svou koncepcí naopak preferuje práci bez knihoven.

Z tohoto hlediska musí systémoví programátoři věnovat při návrhu operačního systému velkou pozornost organizaci knihoven, protože sebemenší nedostatek se projeví nárůstáním ztrátových časů, které jsou spotřebovány při ukládání elementů v knihovnách nebo při vyhledávání elementů v knihovnách.

Uživatel operačního systému musí nutn[illegible] sledovat i nákladové hledisko při analýze výdajů za přípravu programů. Kromě nákladů na strojový čas jsou tu další náklady – náklady na tabelační papír, magnetická média, děrná média. V tomto směru projektanti operačních systémů zneužívají situace, že se na těchto nákladech nepodílejí. Množství nehospodárně potištěného tabelačního papíru u počítačů III. generace tuto skutečnost potvrzuje.

Problematické se jeví z hlediska uživatelů, též používání testovacích prostředků. Uživatelé dnes většinou používají programovacích jazyků vyšší úrovně. Ve standardním programovém vybavení však uživatel najde většinou jen testovací prostředky orientované na jazyk symbolických adres – ASSEMBLER, což jsou ty prostředky, které používali systémoví programátoři při ladění operačního systému.

Také kompilátory programovacích jazyků vyšší úrovně mnohdy nepřispívají ke zjednodušení přípravy programů v důsledku množství chyb a nízké efektivnosti. Ve výpočetních střediscích dodavatele software se pracuje s jazykem symbolických adres. Proto se v jeho překladači odstraní téměř všechny chyby, protože se na ně narazí ihned při rozpracovávání dalších programů operačního systému. Ostatní překladače se vyzkouší jen formálně. Někdy se nevyzkouší vůbec, to když se jedná o nějaký kompilátor, který se přebírá proto, aby zvýšil sortiment poskytovaných překladačů. Používání překladačů s chybami tak ztěžuje přípravu programů ve výpočetním středisku.

Výše uvedené problémy nijak nepřispívají ke zvýšení efektivity tvorby programů [illegible], že [illegible] vytvářejí koncepci programového vybavení pro [illegible] i [illegible] své vlastní práce, nikoliv z pohledu práce [illegible].

3. Interaktivní příprava programů

Jak již bylo v úvodu [illegible], vypracování programu [illegible] počítačů a v technologii [illegible] je [illegible] strané [illegible] programování je interaktivní způsob přípravy [illegible] programátorské práce. [illegible] testovacího [illegible] testování, dovolují navíc výrazně zvýšit produktivitu [illegible] úrovně. Doposud tento způsob práce [illegible] pro-

gramové prostředky, které u nás nejsou většinou k dispozici. Požadavek interaktivní přípravy programů je však natolik aktuální, že počítače JSEP 2 a počítače SMEP 2 by měly být dodávány výhradně s touto možností.

Nutnou podmínkou pro úspěšné rozšíření interaktivní přípravy programů bude i výchova programátorů v tomto směru, kteří nemají v této oblasti praktické zkušenosti.

Vzhledem na skutečnost, že problematice interaktivní přípravy programů byla v poslední době věnována řada speciálních příruček, není tato problematika zde dále rozváděna.

6. Volba programovacího jazyka

Programovací jazyk je vedle metodiky programování druhou základní složkou, která rozhodující měrou určuje vlastnosti systémů pro tvorbu programů.

Starší programovací systémy byly orientovány na možnost používat několik různých programovacích jazyků, podle volby uživatelů. Byly to většinou problémově orientované jazyky ALGOL, FORTRAN, COBOL, BASIC a později i jazyk PL/I.

V současné době převládá směr, kdy je programovací systém orientován na jeden programovací jazyk univerzálního typu. Je to jeden z modernějších programovacích jazyků, který umožňuje využití progresivních myšlenek strukturovaného programování, modulového programování, tvorby datových typů atd.: jedná se o jazyky PASCAL, MODULA, SIMULA 67, ALGOL 68 a nejnovější Ada. Při této koncepci bývá programovací systém uspůsoben pro použití jazyka symbolických adres, který umožňuje odstranit "úzké" místo nosného jazyka.

Zdá se, že přes lákavé zdání výhody universálnosti multijazykových programovacích systém, může poskytnout monojazykové koncepce programovacích systémů celou řadu nezanedbatelných výhod jak pro uživatele, tak pro dodavatele programového vybavení:

- snížení nákladů na údržbu jediného kompilátoru
- zvýšení spolehlivosti kvalitnější údržbou jediného kompilátoru
- redukce potřebné paměťové kapacity pro podpůrné moduly
- lepší vlastnosti kompilátoru v důsledku přesnější programovanosti
- možnost cílevědomé orientace metodiky programování a standardizace.

Stává se, že programovací systémy obsahují specializované systémové jazyky /CDL 2, LISP, LINGUS, SYSTRAN/. Obvykle jsou to programovací systémy určené pro specifické okruhy aplikací např. tvorba software nebo problémy umělé inteligence.

7. Systém SSD firmy FUJITSU

Jako příklad progresivně pojatého programovacího systému může posloužit systém SSD /Software for Structured Development/, vyvinutý a používaný japonskou firmou FUJITSU pro tvorbu programového vybavení.

Systém byl navržen podle následujících koncepčních zásad:

a/ postupný proces projektování po etapách musí zabezpečit vytváření hierarchických struktur programových systémů

b/ rozšířit na maximum automatické vyhledávání chyb, aby se co nejvíce chyb nalezlo již v prvních etapách návrhu

c/ zajistit trvalou a úplnou informaci o každé komponentě navrhového programového produktu

d/ dialogový způsob komunikace se systémem při všech funkcích

e/ použít systému jako prostředku pro dokumentaci programů.

Struktura finálního programového systému, který je prostřednictvím SSD vyvíjen, je popsána prostřednictvím svých komponent a vztahy mezi nimi.

Základní komponentou je modul. Systém se skládá ze souboru komponent. Popis systému se nazývá specifikace projektu. Z něj se generují automaticky požadavky na specifikované komponenty.

Každá komponenta má dvě části:

- specifikaci požadavků na funkce komponenty
- implementaci specifikovaných požadavků nebo podrobnější specifikaci dalších potřebných komponent.

První část vygeneruje systém SSD vždy automaticky, druhou část doplní projektant. Takto se vytvoří stromová struktura systému. Projektovaný systém je hotov, když všechny větve jsou ukončeny implementací specifikovaných požadavků v programovém modulu. Translátor.

Systém SSD zajišťuje prověrku úplnosti specifikací projektu a implementovaných modulů, jakož i kontrolu rozpracovanosti specifikací na každé úrovni.

Schema základních částí SSD ukazuje obr. 7.

Implementačním jazykem modulů /SIL/ se může stát jazyk FORTRAN, PL/I, ALGOL 60 nebo SPL /jazyk pro systémové programování firmy FUJITSU/.

Speciální jazyk pro projektování /SDL/ umožňuje realizovat výše uvedenou koncepci systému SSD tím, že zajišťuje:

- možnost automatického prověření formálně zapsaných specifikací požadavků na funkce jednotlivých komponent
- vytvoření hierarchické struktury komponent
- popis všech tří typů komponent.

Při použití jazyka SDL zahrnují údaje popisující projekt:

- popisy komponent
- údaje o vazbách mezi komponentami.

Tyto údaje jsou vytvářeny pomocí elementů a příkazů jazyka SDL. Všechny údaje jsou uloženy v databázi.

8. Prostředí programové podpory jazyka Ada

Tvůrci jazyka Ada a organizátoři jeho zavedení si od počátku uvědomovali, že plná výhoda nového programovacího jazyka bude realizována pouze tehdy, až bude k dispozici úplné a promyšlené prostředí jeho programové podpory. Na základě této motivace vznikl návrh APSE /Ada Program Support Environment/.

Uveďme zde přehled obecných požadavků na APSE jak byl popsán v dokumentu označeném krycím názvem STONEMAN:

"Požadavky na prostředí programové podpory jazyka Ada /APSE/ je nutno vyvíjet v kontextu charakteristik popsaných pro začleněné počítače /mikroprocesory/. Je jasné, že je třeba vyvinout prostředky pro podporu těch, kteří systém vyvíjí a udržují, právě tak jako pro vedoucího. Jazyk Ada se snaží o portabilitu programů a programátorů. Svět začleněných počítačů je dynamický, pokud se týká mobility programátorů. Je tedy nutné, aby programátoři nacházeli konzistentní prostředí při přesunech od jednoho projektu k jinému. Prostředky nejenom musí být přenositelné z jednoho počítače na druhý, ale musí programátorovi poskytovat konzistentní formu styku s počítačem.

Prostředí musí podporovat celý vývojový cyklus programového vybavení. Musí poskytovat koordinovaný a úplný soubor prostředků, které je možno použít na všechny stadia vývoje programu, a které

poskytují konzistentní podporu nejenom pro vývoj, ale pro pokračující změny a aktualizace v pozdějších stadiích vývojového cyklu.

Soubor prostředků nejenom že musí podporovat vhodné funkce, ale musí být integrován v konzistentním prostředí. Jednotlivé prostředky musí být schopny nejenom mít přístup k aplikačnímu programu, ale také komunikovat navzájem. Toto propojení mezi prostředky musí být nezávislé na hostitelském počítači.

APSE musí usnadňovat vývoj a údržbu programů pro projekty systémů se začleněným počítačem používající programy v jazyku Ada, s účelem zlepšení dlouhodobé cenové efektivní spolehlivosti programového vybavení. Musí podporovat všechny funkce požadované týmem realizujícím projekt. Tyto funkce zahrnují kontrolu vedení projektu, dokumentaci a záznamy a dlouhodobou kontrolu konfigurace a verze. Je nutno zajistit podporu pro projekty během celého vývojového cyklu programového vybavení od požadavků a návrhu přes implementaci k dlouhodobé údržbě a modifikaci. Musí odrážet priority pro kvalitu programového vybavení v aplikacích začleněného počítače; tj. spolehlivost, výkon, vývoj, údržbu a schopnost reakce na měnící se požadavky.

APSE musí být navrhnuto tak, aby využívalo, ale nevyžadovalo, moderní vysoce kapacitní a vysoce výkonné technické vybavení hostitelského systému. Musí to být značně odolný systém, který se umí chránit před chybami ze strany uživatele i systému, který je schopen se dostat z nepředvídaných situací a který může poskytovat svým uživatelům smysluplné diagnostické informace. Musí usnadňovat snadný přenos podpory projektu z jednoho hostitelského počítače na jiný.

APSE musí poskytovat dobře koordinovaný soubor užitečných prostředků s jednotnými propojeními mezi prostředky a s komunikací prostřednictvím společné databáze, která slouží jako zdroj informací a místo pro ukládání produktů pro všechny prostředky. Tam, kde je to vhodné, budou prostředky navrhovány tak, aby měly oddělitelné komponenty provádějící omezené funkce, které je možno skládat, uživatel si je může vybírat a komunikují prostřednictvím společné databáze. Musí usnadňovat vývoj a integraci nových prostředků a zlepšování, aktualizaci a náhradu prostředků.

Struktura APSE by měla být založena na jednoduchých jednotných pojmech, kterým je přímo rozumět, snadno se používají a je jich málo. Všude tam, kde je to možné, budou koncepce jazyka Ada použity v APSE. Vysoká priorita bude dána při návrhu požadavkům lidského inženýrství. Cílem zajistit pro uživatele prospěšné připojení s adekvátními dobami

odezvy pro interaktivní uživatele a dobazi odpovědí pro uživatele dávkového zpracování. Komunikace mezi uživateli a prostředky bude podle konvencí jednotného protokolu."

APSE má tři hlavní rysy:

- databázi
- propojení mezi uživatelem a systémem
- soubor prostředků.

Databáze slouží jako centrální depozitář pro informace týkající se každého projektu po celý vývojový cyklus projektu.

Propojení zahrnuje řídící jazyk, který představuje propojení pro uživatele a také propojení systému s databází a souborem prostředků.

Soubor prostředků zahrnuje prostředky pro vývoj programů, jejich údržbu a řízení konfigurace podporované prostředím APSE.

Portabilita APSE je umožněna tím, že jsou definovány požadavky na dvě nižší úrovně v rámci APSE:

- jádro /the Kernel APSE - KAPSE/
- minimální soubor prostředků /MAPSE/

Tento přístup je ilustrován na obr. 8, kde jednotlivé úrovně zahrnují tyto funkce:

úroveň 0: Technické vybavení a programové vybavení hostitelského počítače podle potřeby.

úroveň 1: Jádro prostředí programové podpory jazyka Ada /KAPSE/, které poskytuje databázi, podpůrné funkce pro komunikaci a pro dobu výpočtu umožňující provádění programů v jazyce Ada /včetně prostředku MAPSE/ a představující propojení, které je nezávislé na počítači a umožňuje portabilitu.

úroveň 2: Minimální prostředí programové podpory jazyka Ada /KAPSE/, které poskytuje minimální soubor prostředků, které jsou jak nutné, tak postačující pro vývoj a pokračující podporu programů v jazyku Ada. Tyto prostředky budou napsány v jazyku Ada a podporovány jádrem KAPSE.

úroveň 3: Prostředí programové podpory jazyka Ada, která jsou konstruována jako rozšíření minimálního prostředí MAPSE pro zajištění úplnější podpory určitých aplikací nebo metodologií.

Model poskytuje konzistentní propojení pro uživatele prostřednictvím KAPSE, které definuje propojení k hostitelskému systému. Dal-

ší prostředky napsané v jazyku Ada je možno snadno přidávat a posléze přenášet na jiné APSE. Tedy na KAPSE je možno se dívat jako na virtuální počítač pro programy v jazyku Ada, včetně prostředků napsaných v jazyku Ada.

Je možno předvídat, že mnoho systémů, které nabízejí použití jazyka Ada jako programovacího jazyka, vznikne bez plného nebo žádného zřetele k požadavkům STONEMANu. Například překladače budou implementovány v rámci existujících prostředí podpory hostujících na existujících operačních systémech, spolu s existujícími programovými prostředky a technikami a možná s implementací některých dalších prostředků majících vztah k jazyku Ada. V případech, kde se na projekty programového vybavení vynakládají velké běžné investice, přičemž tyto projekty byly původně napsány v jiných jazycích a je nutné pro ně pokračovat v dlouhodobé údržbě a zlepšovat ji, může být řešením implantovat prostředky vyvíjované v APSE do existujícího prostředí pro údržbu. To by mohlo poskytnout cenný přenos technologie na úrovni prostředí a ne na úrovni jazyka.

V jiných případech budou konstruovat prostředí podpory jazyka Ada, která budou nabízet použití jazyka spolu s řadou prostředků, které se obsahem a/nebo strukturou znatelně liší od prostředků navrhovaných STONEMANem. Například v některých ústavech zabývajících se výzkumem programování mohou být vytvořeny vysoce integrované vývojové systémy konstruované "shora dolů", které vyhovují požadavkům APSE, ale neodrážejí strukturu KAPSE/MAPSE.

Takové systémy reprezentují zcela platné přístupy k použití jazyka Ada. Dlouhodobým cílem STONEMANu je však vyvinutí integrovaného systému s maximální úrovní portability. Aby se docílilo tohoto dlouhodobého cíle portability programových prostředků a aplikačních systémů na nich závislých, plánuje se, že konvence a nakonec standardy budou vyvíjeny na úrovni propojení KAPSE.

2. Shrnutí požadavků na programovací systém

Základní principy práce programovacího systému musí být založeny na určitém konceptuálním schematu programování a musí se opírat o uznávanou metodiku programování.

Programovací systém musí napomáhat tvorbě programů tím, že na ně bude použit od začátku návrhu až po závěrečné práce na programovém produktu.

S ohledem na současnou situaci při tvorbě programů musí programovací systém umožnit týmovou práci na rozsáhlých systémech programů.

Uživatel musí mít možnost programovací systém efektivně v krátké době využívat. Měl by mít k dispozici dokonalou dokumentaci, obsahující nejen popis programovacího systému, ale i doporučení pro uživatele z hlediska účinného a hospodárného používání. Dokumentace by měla být napsána podle obecně platných, uznávaných, pedagogických a didaktických zásad.

Kompilátor /kompilátory/ vyššího programovacího jazyka by měl mít následující charakteristické rysy:

- v plném rozsahu respektovat mezinárodní normu ISO
- dobrou diagnostiku chyb zdrojového textu
- prostředky pro ladění programu
- doplňkové příkazy, určující způsob a průběh kompilace
- začleňování podprogramů v jazyku symbolických adres
- separátní kompilaci modulů
- používání separátně přeložených modulů
- rozšiřování souboru standardních podprogramů
- poskytovat efektivně generovaný kód při malé spotřebě času na kompilaci
- možnost podmíněné kompilace.

Programovací systém by měl zahrnovat rozsáhlé netriviální možnosti práce se symbolickými zdrojovými texty např. sestavení nového textu programu z částí jiných programů, textové editory pro opravy, makroprocesor, předkompilátory pro vybrané programovací problémy a prostředky pro vypracování vlastních atypických předkompilátorů a generátorů programů.

Současná situace v přípravě programů si vynucuje, aby byla k dispozici široká škála testovacích prostředků pro statickou i dynamickou analýzu programů, umožňujících testování v symbolických objektech programovacího jazyka i testování strojového kódu, které by se mohly odstupňovaně používat. Generátory testovacích dat musí dovolit předepsat teoreticky zdůvodněná kriteria výběru množiny dat testovacího souboru.

Programovací systém by měl zajišťovat centrální i dílčí správu knihovny /knihoven/ včetně manipulací se skupinami elementů, důsledný odstupňovaný úklid neaktivních prvků a zabezpečení archivními kopiemi.

Podpůrné programy programovacího systému by měly brát v úvahu organizační záležitosti při programování a umožnit vést deník prací a jeho sumáře, evidenci sledu prací a jejich kompletaci, kontrolu činnosti programátorů a poskytovat podklady pro účtování a plánování programovacích prací.

V maximální míře by měly být využity archivované texty zdrojových programů pro dokumentaci. Tiskové programy by měly umožnit výpisy jednotlivých programů, jejich částí i skupin programů, pořizovat křížové reference, provádět tisk vývojových diagramů apod.

Pokud je to možné, měly by být programové prostředky programovacího systému orientovány důsledně na použití interaktivního režimu.

S ohledem na praktické používání by měl být celý systém navržen jako otevřený, aby bylo možno dodatečně doplnit komponenty, požadované uživatelem. S praktickým využíváním souvisí i vlastnost portability, aby uživatel mohl přenést celý systém do jiného prostředí. Dalšími požadavky uživatele jsou vysoká spolehlivost systému a jeho celková výkonnost.

Programovací systém by neměl klást maximalistické požadavky na technické vybavení, na druhé straně by však měl efektivně využívat všechny složky počítačového systému.

Při koncipování vlastností programovacího systému je nutno mít na paměti hledisko jednoduchosti a logické výstavby celého systému. Neefektivní komponenty a komponenty s malou nadějí na využití by měly být z návrhu odstraněny.

Závěr

Poslední průzkumy, prováděné v oblasti tvorby programů, zjišťují dva důležité fakty:

- v současné době práce na údržbě dosud vypracovaných programů převyšují několikanásobně předpokládané finanční částky a vyžadují velké programátorské kapacity
- v procesu přípravy automatizace připadá největší procento nákladů na vývoj programů.

Obě zjištění jen zdůrazňují důležitost, kterou je nutno věnovat těm částem operačního systému, které souvisí s problematikou tvorby programů.

Význam programovacích systémů roste tím více, čím klesají ná-

klady na technické vybavení počítačů a zkracuje se doba výroby počítačů a naopak rostou náklady a čas pro realizaci programového vybavení počítačů.

Význam programovacích systémů byl v plném rozsahu doceněn v oblasti tvorby programového vybavení pro mikroprocesory, kde vznikají speciální mikroprocesorové vývojové systémy.

Průkopnickou prací v oblasti programovacích systémů u nás bylo vytvoření programovacího systému BPS ve VVS Bratislava, jehož portabilní řešení ho umožnilo přenést na další počítače. Protože BPS byl předmětem referátů na řadě akcí ČSVTS a byl popsán v našem odborném tisku, nebyl zde zevrubně popisován.

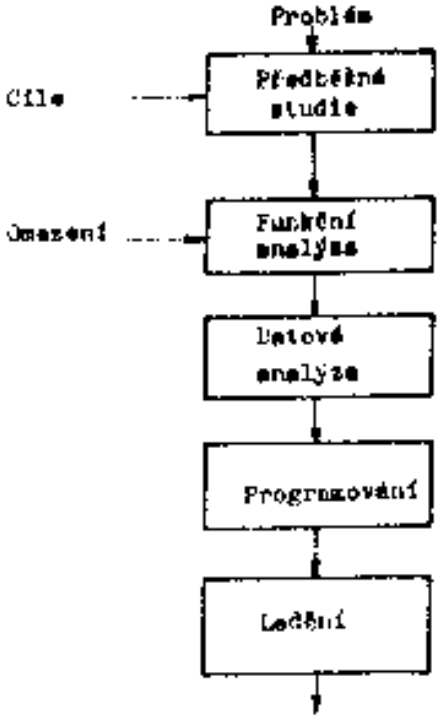
Ve světě se věnuje této problémové oblasti velká pozornost. Poslední novinkou jsou specializované stanice typu CASE - Computer Aided Software Engineering, které se intenzivně vyvíjejí u řady firem v USA a Japonsku.

První komerční zařízení tohoto typu vyvinula firma Convergent Technologies Inc. v USA. Očekává se, že v tomto roce bude dána do výroby pracovní stanice Lilith CASE, jejímž autorem je profesor Niklaus Wirth, autor jazyka Pascal. Zatím existuje prototyp.

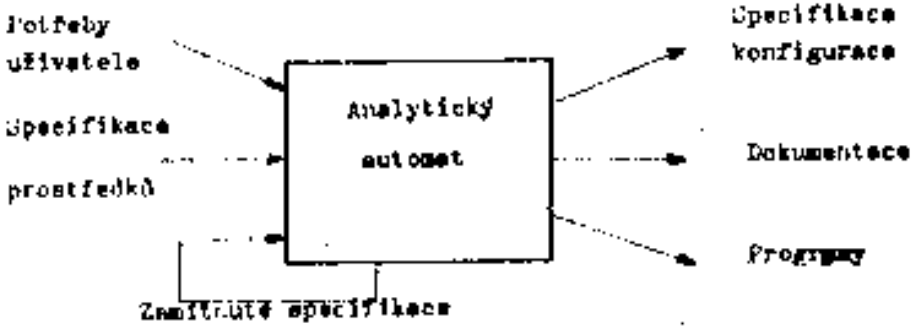
Další zařízení typu CASE připravují firmy Tektronix a Rolm v USA a firma Hitachi v Japonsku.

Tyto práce ukazují, vývoj prostředí pro tvorbu programů neustrnul, ale stále pokračuje.

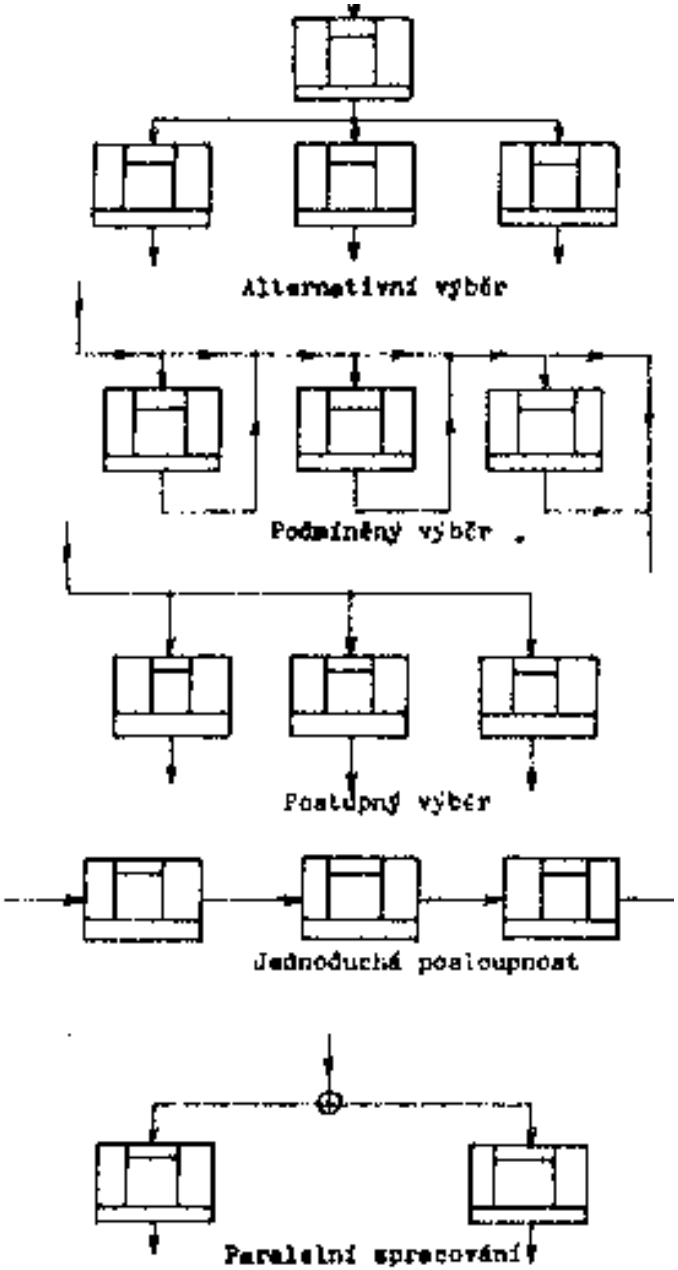
Aby nebyl překročen dohodnutý rozsah příspěvku do sborníku, není uveden seznam použité literatury. Případným zájemcům poskytne autor potřebné informace podle jejich požadavků.



Pragmatické koncepční schema
Obr. 1.



Integrované koncepční schema
Obr. 2

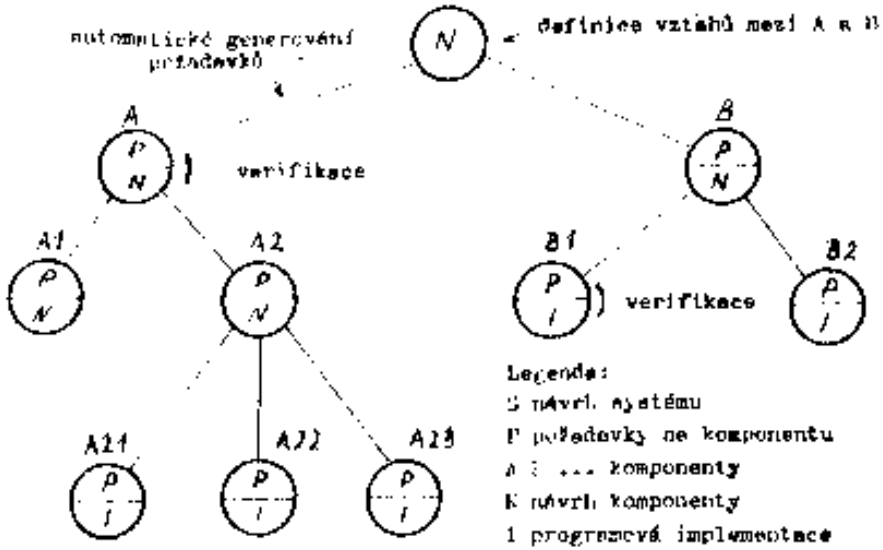


Grafy řízení v systému SDV
Obr. 4.b

Průběžné testování podle [illegible]
Obr. 3

Prostor konceptuálního schematu Peterse a Tripletu
Obr. 3

Formulář elementu ERM
Obr. 4a



Obr. 6

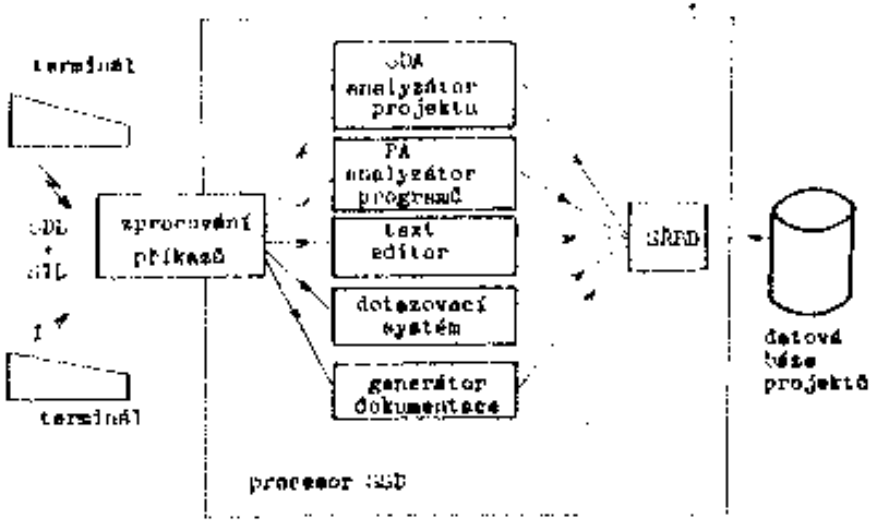
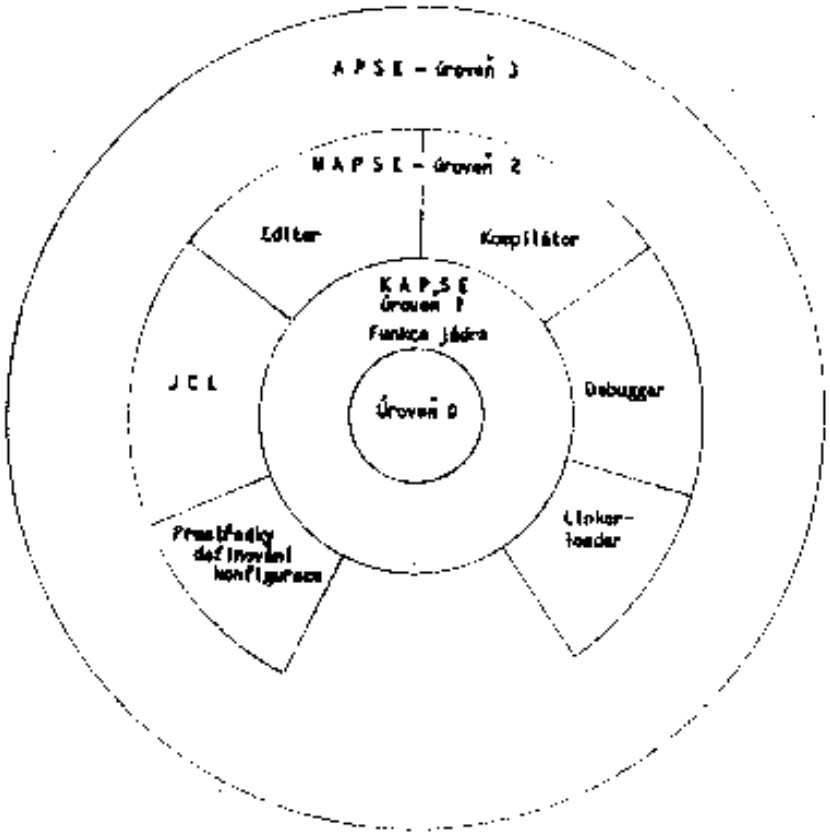


Schéma systému SSD
Obr. 7



Obr. 8

Schematické znázornění úrovní APSE