Vývoj programového vybavení v programovacím jazyce Modula-2.

Jindřich Černohorský

VŠB-TU, FEI, Katedra měřicí a řídicí techniky, 17. listopadu, 708 33 Ostrava, Česká Republika

Abstract

Programming languages differ in the degrees to which their constructs can express architectural decisions. Modula-2 is one of the most powerful languages available today in this respect. The largest structural unit is the compilation unit, a module. Modules define visibility boundaries. Modules combine items which together form an abstraction. The data contained in a module can be exported to other modules through module interface. In this way the large software systems can be structured and developed. There are other facilities in Modula-2 important from point of view of development of real time systems: low level programming facilities and coroutines. Using coroutines the system can be designed and structured into the set of concurrently working processes thus reflecting parallel nature of real world environment. Some issues of software development using Modula-2 are discussed in this contribution.

Úvod

V uplynulých padesáti letech vznikly desítky programovacích jazyků (nejméně 700). Velká většina z nich je zapomenuta. Zdá se, že z hlediska vývoje rozsáhlých programových systémů jsou dnes pro přežití jazyka podstatné dvě okolnosti:

1. V jaké míře reflektuje některou z efektivních a úspěšných návrhových metodologií (strukturovaný návrh, modulární návrh, OOP návrh, COP návrh).
2. Do jaké míry podporuje programování v prostředí nejrozšířenějších operačních systémů.

Jazyk MODULA-2, druhé ze tří "dětí" (Pascal,Modula,Oberon) profesora Niklause Wirtha (ETH Zürich) byl poprvé implementován v roce 1979 na počítači DEC. Původním záměrem autora bylo vytvořit jazyk, který by bylo možno použít pro napsání operačního systému.

Modula-2 vznikla z Pascalu, s nímž se shoduje v řadě rysů. Nadto má však další vlastnosti vhodné a nezbytné jak pro systémové programování, tak pro programování úloh reálného času a tedy i pro systémy řízení technologických procesů.

Přínos v tomto směru představují zejména:

1. Výsledná architektura programu je založena na dokonale propracovaném systému členění kódu do modulů, což umožňuje efektivně implementovat modulární návrhové techniky.

2. Poměrně přísná typová kontrola, která je jedním ze základních nástrojů bezpečného programování a implementaci bezpečných systémů.

3. Prostředky pro programování na úrovni stroje, tzv. "low level programming". Zbavují programátora nutnosti uchylovat se k asembleru nebo k různým nebezpečným trikům.

4. Podpora multithreadingu realizovaná pomocí korutin. Programy lze navrhovat a realizovat jako paralelní procesy, včetně procesů pro obsluhu přerušení. Tato možnost je významná jak z hlediska návrhu, tak i z hlediska implementace systémů pracujících s reálným časem.

V českých zemích nepatří Modula-2 mezi známé nebo oblíbené jazyky a její vyznavači jsou považováni tak trochu za zvláštní druh sekty. Může to svědčit o jejich vkusu: buď velmi dobrém anebo velmi špatném. V žádném případě však ne o průměrném. Vzhledem k uvedeným vlastnostem najdou odvahu ji použít především ti, kteří hledají vhodný nástroj pro vývoj aplikací, v nichž by se s jazykem bez uvedených vlastností pracovalo velmi obtížně.

Z vlastní zkušenosti lze potvrdit, že síla Moduly spočívá v její schopnosti reflektovat rysy modulárního a paralelního návrhu a že je velmi vhodná pro týmový vývoj aplikací. Jako velmi výrazné projekty vytvořené v jazyce Modula lze uvést produkty firmy ALCOR Zlín : Inview, Control Panel a SafetyLab. Problematika použití Moduly a modulárního návrhu je velmi široká a na malém prostoru tohoto příspěvku je možno podat jen informace nejpodstatnější a nejstručnější.

Moduly

Vytvořit správnou softwarovou architekturu je jedna z nejobtížnějších věcí. Vytvořit nevhodnou je jedna z nejdražších. Jestliže programovací jazyk umožňuje explicitně vyjádřit strukturální prvky návrhu, stává se nejúčinnějším vývojovým (CASE) nástrojem, protože překladač může ověřovat konformitu programu s architekturou návrhu, která je jeho základem. Čím více toto může dělat během kompilace, tím lépe.

Modula může definovat statickou strukturu programu, což má pro návrh architektury programu mimořádný význam. Největší strukturální jednotkou je kompilační jednotka, *modul*. Prakticky můžeme definovat modul jako segment programu, který komunikuje se svým okolím (tj. s ostatními moduly) pouze prostřednictvím dobře definovaného rozhraní, jeho začlenění do většího programového systému může být provedeno bez znalosti jeho vnitřní struktury a jeho správnost může být ověřena bez úvah o jeho začlenění do širšího programového systému.

Moduly definují hranice viditelnosti objektů. Co je uvnitř modulu (konstanty, proměnné, procedury, typy) není "vidět", (tj. není možno užívat, modifikovat, volat) mimo modul, pokud to není modulem explicitně exportováno. Modulem exportované prvky se nazývají rozhraní modulu. Modul může importovat rozhraní jiných modulů.

Moduly obsahují kombinace programových prvků, které dohromady tvoří nějakou abstrakci, např. systém ovládání souborů. Taková abstrakce obvykle zahrnuje nejenom abstraktní datové typy nebo třídy, ale celý soubor vzájemně propojených typů objektů, konstant, globálních proměnných a procedur. Dostupnost konstruktu modul je tak jednou z největších výhod Moduly.

Program v Module může být popsán pomocí grafu modulů. Je jím orientovaný acyklický graf ukazující, kdo importuje od koho. Volání ve směru importu se nazývá "upcall" v obráceném směru jde o normální volání procedury. Upcall představuje určitý implementační problém : odvoláváme se na něco, co bude třeba vytvořeno i někdy v budoucnosti „neznámým programátorem". Objektové orientované techniky to řeší virtuálními metodami.

Pokud se nemění rozhraní, může být poměrně volně modifikována implementace modulu, aniž je nutno rekompilovat ostatní moduly. Rozhraní a implementace modulu mohou být i v určitém smyslu rozšířeny - například přidáním dalších procedur - aniž bychom tím omezili platnost již existujícího kódu.

Typová kontrola

Druhou významnou statickou strukturální vlastností Moduly je její zacházení s typy. Každá proměnná je určitého typu, který říká, jakých hodnot může typ nabývat a jaké operace jsou přípustné s proměnnou tohoto typu. Kromě základních předdefinovaných typů umožňuje Modula definovat tři druhy strukturovaných typů: *pole, záznamy* a *množiny*. Strukturované typy lze libovolným způsobem kombinovat do nejrůznějších hierarchických datových struktur. Spolu s množinou typových ukazatelů (každý ukazatel je svázán se svým základním typem) to dává možnost konstruovat velmi flexibilní datové struktury.

Modularizace a modulární návrh

V modulárním programování spíše než efektivita programu, je důležitějším požadavkem jeho flexibilita. Podstatou modulárního návrhu je *modularizace*, tj. proces dekompozice programu na moduly resp. proces seskupení požadovaných funkcí programu do jednotlivých modulů.

Jedním z nejdůležitějších cílů modularizace je vytvořit poměrně nezávislé, volně svázané moduly. Neboť: čím méně vztahů je mezi moduly, tím menší je nebezpečí šíření chyb mezi nimi. A také změny v jednom modulu případně vyvolají změny v menším počtu modulů.

Modularizace navazuje na fázi analýzy systému, jejímž výsledkem je souhrn funkcí, které mohou být realizovány jako funkce, procedury, procesy, objekty atp., tj. jako menší jednotky než je celý program nebo modul.

Přitom vzniká otázka: Jaké bude rozhraní mezi těmito jednotkami a na základě jakých kritérií mají být sdružovány do modulů? Optimální odpověď na tuto otázku má dát právě modularizace. Při modularizaci nám mohou pomoci některá pravidla a postupy, jak moduly správně konstruovat. Zde jsou některé z nich.

- Každý modul by měl vykonávat úlohu, která představuje do určité míry uzavřený problém. Funkce, obsažené v modulu by měly tvořit v jistém smyslu logickou jednotku. Není dobré, jestliže ucelené algoritmy resp. postupy jsou rozděleny mezi více různých modulů.Této vlastnosti se říká *uzavřenost modulu*. "Síla", která drží modul "pohromadě" je dána tzv. soudržností modulu, o kterou by se mělo usilovat.

- *Soudržnost modulu* je míra vzájemné závislosti funkcí uvnitř modulu. Soudržnost je trojího druhu: *Funkční, datová a sekvenční*.

- *Funkčně soudržné moduly* obsahují funkce, které jsou nezbytné a postačující pro řešení určitého úkolu. *Datově soudržné moduly* jsou tvořeny funkcemi, které všechny využívají téchže datových struktur. *Sekvenčně soudržné moduly* jsou tvořeny funkcemi, které musí být provedeny postupně po sobě a kde výsledek předchozí funkce je potřebný pro funkci následující.

- V případě práce s paralelními procesy vstupuje do hry *soudržnost časová*, která je vodítkem pro dekompozici systému na procesy a řídí se především časovými hledisky, jako jsou *cykličnost, doba odezvy, časová návaznost* atp.

- Z požadavku na jednoduchou komunikaci mezi moduly vyplývá požadavek na *minimalizaci rozhraní* a *viditelnosti*. Tím se vlastně minimalizují předpoklady, které moduly dělají vzájemně jeden o druhém. Modularizace by měla být provedena tak, aby rozhraní mezi moduly byla co nejjednodušší a explicitně definovatelná.

- Vlastní komunikace mezi moduly by měla být správně prováděna výhradně procedurami a s minimem parametrů. Moduly, které komunikují přístupem ke společné datové oblasti, jsou těsně svázány, čímž je ztíženo hledání eventuálních chyb.

- Požadavek na *nezávislost modulů* znamená, že dekompozice musí zaručovat, že moduly se nijak vzájemně interně neovlivňují, a že může být nějaký modul nahrazen jiným, aniž by narušil fungování celého systému za předpokladu, že respektuje existující rozhraní.

- Každý modul by měl být co nejmenší v tom smyslu, že věcně i typograficky by neměl obsahovat nic navíc, co by ztěžovalo orientaci v něm. *Modul je dokonalý tehdy, když už z něj nelze nic ubrat a ne když už k němu nelze nic přidat.*

- K ocenění vlastnosti modulu je možno použít dvou čísel. *Importní číslo modulu* je počet modulů importovaných modulem. Velké číslo může naznačovat, že modul musí vykonávat mnoho koordinačních a rozhodovacích úloh a obsahuje příliš málo úrovní abstrakce. Malé číslo je podnětem k zamyšlení, nemá-li být modul ještě dekomponován na menší části. *Číslo využitelnosti modulu* je počet modulů, v nichž je modul importován. Čím větší jsou čísla využitelnosti modulů v systému, tím snadnější údržba, protože jenom malý počet podobných komponent musí být během údržby opraven. To platí jen za předpokladu, že takové moduly mají silnou modulovou soudržnost.

Testování modulů

Každý modul by měl být konstruován tak, aby jeho správnost mohla být co nejjednodušeji ověřena na základě jeho rozhraní bez znalosti toho, jak bude zakomponován do výsledného systému. Avšak samotný knihovní modul nemůže být testován, protože nemá exekutivní formu. Je proto třeba vytvořit vhodné testovací okolí (moduly), nemá-li být celý systém testován vcelku. To musí být co nejjednodušši, aby bylo zaručeno, že v něm není chyba.

Testování interakcí mezi moduly

Po otestování jednotlivých modulů je možno testovat subsystémy. Subsystémem rozumíme funkční kombinaci jednotlivých modulů. Účelem je otestovat interakce mezi moduly subsystému, především pak správnost komunikace mezi moduly. I zde musí být vytvořeno vhodné testovací prostředí. Subsystémy jsou dále spojovány a testovány ve větších celcích. Tato forma testování se nazývá *hierarchické testování* nebo *integrační testování*.

Hierarchie modulů

Praxe ukazuje, že vnitřní struktury velkých programových systémů mají podobnou konstrukci s ohledem na velikost aplikace. V konečném návrhu můžeme rozlišit 4 hierarchické úrovně modulů:

1. Moduly na nejnižší úrovni definují struktury dat, datové typy a přístupové operace k jednotlivým složkám těchto typů. Půjde například pro manipulaci s řetězci, matematické funkce, operace se soubory, vstupně výstupní operace , atp.
2. Nadřazená úroveň zahrnuje moduly, jejichž funkce manipulují se skupinami elementů různých typů úrovně 1. Například moduly pro práci s okny. Obě tyto úrovně obsahují tedy funkce, které zapouzdřují datově orientované funkce.
3. Moduly další úrovně obsahují implementaci obecně orientovaných subsystémů. Zde se kombinují elementy různých datových typů úrovní 1. a 2. Cílem je vytvořit obecně orientované flexibilní subsystémy, jako například subsystém zpracování textů ve víceoknovém prostředí.
4. Nejvýše položené moduly obsahují řídící funkce, které kombinují funkce nižších úrovní způsobem, aby byla naplněna systémová specifikace, tj. realizují problémově orientované funkce.

Struktura programu v Module

Program v jazyku Modula-2 je tvořen jedním programovým modulem (*hlavní modul*) a libovolným počtem *knihovních modulů*. Hlavní modul i knihovní moduly dovážejí z jiných knihovních modulů různé entity, jako jsou konstanty, typy, proměnné, třídy, objekty a procedury.

Programové i knihovní moduly mohou obsahovat další, menší, vnořené moduly, které nejsou viditelné vně modulu v němž jsou vnořeny a nazývají se *lokální moduly*. Tyto lokální moduly se definují pouze pro účely nadřazeného modulu,

v němž jsou definovány. Z entit deklarovaných uvnitř modulu jsou z vnějšku viditelné a použitelné pouze ty, které modul exportuje.

Knihovní moduly

Knihovní modul se skládá ze dvou částí: z *definičního modulu*, a z *implementačního modulu*.

Definiční modul obsahuje pouze definice entit, tj. konstant, typů, proměnných a rozhraní procedur, které modul exportuje; definuje rozhraní knihovního modulu vzhledem ke zbytku programu. Implementační modul definuje, jak jsou služby (tj. podprogramy, funkce, datové typy, objekty atp.) popsané v definiční části implementovány. Kromě toho může implementační modul obsahovat definice dalších procedur, typů, dat atp., které nejsou v definičním modulu exportovány a jsou pak pro jiné moduly neviditelné.

Vzhledem k jiným modulům jsou objekty deklarované v definiční části modulu *nelokální*, objekty deklarované v implementační části modulu jsou *lokální*. Vzhledem k objektům v témž modulu jsou *globální*. Na rozdíl od lokálních proměnných procedur si lokální a nelokální proměnné modulů zachovávají svou hodnotu po celou dobu běhu programu.

Definice typu v definičním modulu může opomenout specifikaci typu a uvést pouze jeho jméno. Pak je třeba typ specifikovat v implementační části modulu a jeho struktura bude vně modulu nedostupná, neprůhledná. Takovému typu, jehož strukturu neprozrazujeme mimo hranice modulu, říkáme *skrytý typ*.

Definice skrytého typu je základem pro vytváření programových abstrakcí. Ta je však proveditelná technicky jedině tehdy, je-li skrytý typ typu ukazatel. Proto struktura typu i typ ukazatel na něj se definují v implementační části modulu, v definiční části se pak uvede pouze jméno ukazatele jako jméno skrytého typu. Ukážeme si nyní na jednoduchém výřezu z modulu pro práci s komplexními čísly.

```
IMPLEMENTATION MODULE COMPL;
.....
  TYPE KOMPLEX = POINTER TO KomplexniCislo;
  TYPE KomplexniCislo = RECORD
     x,y : LONGREAL
  END
  PROCEDURE SumCx( C1,C2: KOMPLEX ) : KOMPLEX;
....
END COMPL.

DEFINITION MODULE COMPL;
.....
  TYPE KOMPLEX ;   (* Zde se zviditelňuje jméno skrytého typu *)

  PROCEDURE SumCx( C1,C2: KOMPLEX ) : KOMPLEX;
  VAR Temp : KOMPLEX;
  BEGIN
```

```
    Temp^.x := C1^.x + C2^.x
    Temp^.y := C1^.y + C2^.y
    RETURN ( Temp )
  END SumCx

.....
END COMPL.
```

V zákaznickém modulu je pak možno například následující použití

```
  FROM COMPL IMPORT KOPMLEX, SumCx;
  VAR  a,b,c : KOMPLEX;
                          c:=SumCx( a,b );
```

Při použití skrytého typu nevidíme strukturu typu a na všechno, co chceme s proměnnými skrytého typu provádět musíme mít operace, které vytvoříme v implementační části modulu COMPL a v definiční části povolíme jejich vývoz.. Používání skrytých typů vytváří předpoklady pro velmi dobrou modifikovatelnost implementace.

„Low-level programming" aneb programování (skoro) na úrovni stroje

Při systémovém programování je často třeba provádět takové operace jako např.:

- číst a zapisovat údaje na konkrétní adresy v paměti
- pracovat s jednotlivými bity, bitovými poli, a byty.
- číst a zapisovat z a do registrů periferních zařízení

Tyto operace budou vždy závislé na konkrétní HW architektuře. Proto je součástí každé implementace jazyka MODULA-2 standardní modul SYSTEM, který exportuje strojově závislé typy a procedury pro programování na úrovni stroje. Zde lze počítat například:

- *Typ BITSET* umožňující přístup k jednotlivým bitům slova. Ten je jediným předdefinovaným strukturovaným typem v MODULE-2. Odpovídá definici množinového typu TYPE BITSET = SET OF [0..W-1]. Přitom W závisí na konkrétním typu stroje, obvykle W = 16 (délka slova INTEGER nebo CARDINAL).

- *Typy WORD, BYTE a LONGWORD*, umožňující operace se byty, slovy, a dvojslovy bez ohledu na typ v nich obsažených dat. Jako formální parametr procedury je typ WORD kompatibilní s libovolným skutečným parametrem, který má délku 16 bitů. (Jsou to CARDINAL, INTEGER, BITSET a SHORTADR). Formální parametr typu ARRAY OF WORD pak koresponduje se skutečným parametrem libovolného typu, například s typem záznam. To nám umožňuje psát procedury velké obecnosti. Podobné (stejné) vlastnosti mají i ostatní typy modulu SYSTEM, tj. typy BYTE a LONGWORD, které koresponduji s typy SHORTINT, SHORTCARD, CHAR a BOOLEAN, resp. s typy LONGCARD, LONGINT a REAL.

- *Typ ADDRESS* pro uložení a *proceduru ADR*, pro vytvoření strojových adres. Hodnoty typu ADDRESS jsou adresami slov v operační paměti. Hodnota ADDRESS se chová (v závislosti na použitém paměťovém modelu) v souladu s deklarací TYPE ADDRESS = POINTER TO WORD , resp. TYPE

ADDRESS = POINTER TO LONGWORD. Typ ADDRESS je kompatibilní s libovolným typem ukazatele, čili je-li VAR ptr: POINTER TO T; (*T je libovolný typ*) a VAR addr: ADDRESS, můžeme přiřazovat ptr:=addr; nebo addr:=ptr;. Pro aritmetické operace na typech ADDRESS, jsou k dispozici funkce AddAdr a SubAdr.

- Funkce umožňující převedení typů na jiný typ umožňující se v odůvodněných případech vyhnout se typové kontrole, což je na strojové úrovni nezbytné

- Možnost spojit určitou proměnnou (identifikátor) s konkrétní strojovou adresu a tím umožnit stejný formální přístup k proměnným na *absolutních adresách* paměti jako k proměnným alokovaným překladačem.

Příklady.
Adresa 0:410H obsahuje tzv."equipment flag" a jehož bity 11-9 udávají počet seriových portů systému. Ve spojení bitovými operacemi pak můžeme řešit následující úlohy.

A. Definujeme : CONST SerMask = BITSET{9..11}; SerShift = 9;
VAR EquipFlag[0:410H] : BITSET;
PSP : CARDINAL; (* PočetSeriovýchPortů*)
zjistíme počet seriových portů PSP:
PSP:=CARDINAL(EquipFlag * SerMask) >> SerShift;

B: Podobně lze například pracovat s pamětí VideoRAM. Stačí deklarovat

```
CONST BlinkBit=BITSET{15};      bit blikání
      InstyBit=BITSET{11};      bit jasu
      FgColor =BITSET{8..11};   bity barvy popředí
      BgColor =BITSET{12..15};  bity barvy pozadí
  TYPE Item = RECORD Item       definuje strukturu obsahu slova VideoRAM
         Znak : CHAR;           prvni byte - kód znaku
         Atr  : SHORTCARD;      druhy byte - hodnota atributu
       END;
  TYPE Screen = ARRAY [1..25],[1..80] OF Item;
  VAR page0,page1,  VRAM[ 0B800H:0H ] : Screen;
```

Příkazem VRAM [2,26]:= Item('A', SHORTCARD(23)) lze přímo zapisovat do paměti obrazovky, příkazem page0:=BWS kopírovat celou obrazovku do paměti atp.

- Module SYSTEM definuje velký počet systémově závislých typů a konstant, které jsou ve skutečnosti "vestavěny" do překladače Moduly-2 a dále speciální procedury podporující strojově specifické programování na počítačích rodiny 80x86. Například typ Registers se užívá pro uchování aktuálních hodnot registrů procesoru.

```
TYPE Registers = RECORD
          CASE : BOOLEAN OF
          | TRUE  : AX,BX,CX,DX,BP,SI,DI,DS,ES : CARDINAL;
                Flags                  : BITSET;
          | FALSE : AL,AH,BL,BH,CL,CH,DL,DH   : SHORTCARD;
          END;
        END;
```

- Kromě již uvedených typů umožňuje modul SYSTEM svými prostředky
  - zakazování a povolování přerušení
  - přímou komunikaci s hardwarovými porty.
  - vytváření a použití korutin

Pro práci s porty a přerušením je k dispozici řada funcí a procedur , mezi nimi například

```
PROCEDURE   Out (port:CARDINAL; value:SHORTCARD); zápis bytu na port
PROCEDURE OutW(port:CARDINAL;  value:CARDINAL );  čtení slova z portu
PROCEDURE    In (port:CARDINAL): SHORTCARD;       čtení bytu z portu
PROCEDURE  InW(port:CARDINAL): CARDINAL;          čtení slova z portu
PROCEDURE DI(); zákaz přerušení
PROCEDURE EI(); povolení přerušení
```

Korutina (coroutine) je sekvenční proces, který může být odložen tím, že předá řízení jiné korutině a v okamžiku, kdy je mu předáno řízení zpět, pokračuje ve zpracování přesně v místě, kde bylo její zpracování přerušeno.

V Module-2 je řízení mezi korutinami předáváno pomocí podprogramů TRANSFER a IOTRANSFER (na základě HW přerušení). Korutina (proces), však musí být nejdříve vytvořena a iniciována pomocí procedury NEWPROCESS. Problematice paralelního zpracování v MODULE-2 je v tomto sborníku věnován samostatný příspěvek.

Všechny uvedené možnosti sice přispívají k větší flexibilitě jazyka, avšak jejich intensivní využití vede ke ztrátě portability programu a přes prováděné kontroly přece jenom zvětšují možnost vyrobit závažné, obtížně odhalitelné chyby. Proto je zde třeba co největší opatrnosti. Prevence spočívá ve vhodné modularizaci, s výsledkem lokalizace problematických operací ve speciálním modulu.

Něco o překladačích

Jednou z nejznámějších je implementace jazyka TopSpeed Modula-2 fy Jensen & Partners International nyní Clarion Software. Kromě překladače Moduly-2 vyvinula tato firma překladače pro PASCAL, C, a C++. Všechny překladače používají stejné vývojové prostředí, umožňují zmíněný paralelismus, mají prvky OOP, včetně vícenásobné dědičnosti a patří mezi nejlépe optimalizující systémy na světě. Je možno vyvíjet programy pracující v protected modu.Velmi dobré je i vývojové prostředí včetně VIDu - Visual Interactive Debugger. Bohužel po pohlcení firmou Clarion Software se vývoj zastavil, včetně vývoje podpory pro práci pod Windows a OS-2, které zůstaly na úrovni roku 1994.

Velmi známé jsou překladače společnosti GradenPoint Modula, které mohou pracovat na těchto paltformách (překlad se generuje do jayzka C) :
HP 9000-8xx a HP 9000/7xx pod HP-UX, Pracovní stanice Silicon Graphics (IRIS) Tandem(architektura mips) , Pracovní stanice IBM RS6000 Power Series , SunSparc (SunOS a Solaris 1.xx), Pracovní stanice Appolo (DN3x00 a DN4x00)

Vlastní kód se pak generuje pro architektury:
Stanice DEC po ULTRIX, DEC Alpha APX pod OSF-1, Sun Sparc (Solaris 2.xx) a Intel ipa386/486 pod Unix SV4.2, WindowsNT, IBM OS/2, Linux(shareware), MS-DOS Go32 DOS Extender (shareware) MS-DOS interpret (freeware).

V tuzemsku existuje v Brně firma CERSOFT, která vyvinula vynikající vývojové prostředí Multi Micro Modula-2 umožňující v jazyku Modula-2 vývoj aplikací pro celou řadu mikroprocesorů , jako např. Motorola, Toshiba, Zilog, Intel8051 a jeho deriváty. Vývojové prostředí je nejen srovnatelné s obdobnými produkty světových firem, ale v řadě aspektů je přesvědčivě předčí. Kromě Moduly-2 je zde dispozici i překladač Obreonu-2 a řada dalších produktů jako např. Processor Expertů.

Závěr

Přednosti jazyka Modula-2 spočívají v jeho schopnosti reflektovat modulární resp. objektově orientované návrhové techniky. Disponuje dále nástroji, které jsou potřebné pro implementaci řídicích systémů a systémů pracujících v reálném čase. Současnou nevýhodou je malá podpora pro nejrozšířenější operační systém Windows95. V přímé návaznosti na jazyk Modula-2 však vznikl programovací jazyk Oberon-2 a vývojové prostředí Oberon-2, cílené na vývoj technikami komponentově orientovaného programování. To vychází z technik modulárního a objektově orientovaného programování. Technologie Oberonu je orientována na použití pod Windows95, WindowsNT, Mac OS v budoucnu i na další nejrozšířenější architektury. Mezi nimi jsou pak aplikace velmi dobře přenositelné. Lze očekávat, že vyznavači modulárních návrhových a implementačních postupů přesunou svůj zájem od jazyka Modula-2 k Oberonu, který se jeví jako velmi perspektivní pro rozsáhlé aplikace s předpokládanou dlouhodobější dobou vývoje a rozvoje provozované v prostředí Windows nebo Mac OS. Prostorem vhodným pro Modulu-2 pak zůstanou aplikace menšího a středního rozsahu s vysokými požadavky na bezpečnost a modifikovatelnost návrhu a především oblast programování mikroprocesorů.

Literatura

[1] Wirth, N.: Programming in Modula-2, 4.vydání, Springer Verlag, 1989
[2] Pomberger, G.: Software Engineering and Modula 2, Prentice Hall, 1986
[3] The Concepts Behind Oberon/F, Oberon/F Release 1.2, Oberon Microsystems, Inc. Switzerland
[4] Pressman,R.: Software Engineering, Mc Graw Hill, 3.vydání, 1992
[5] Burns,A.,Wellings,A.: Real-Time Systems and Their Programming Languages. Addison-Wesley, 1994