PROGRAMOK, PROGRAMSPECIFIKÁCIÓK

TARTALOMJEGYZÉK

PROGRAMOK, PROGRAMSPECIFIKÁCIÓK

Szlávi Péter, szlavi@ludens.elte.hu
ELTE TTK Informatika Szakmódszertani Csoport

Absztrakt

*Az előadásban vázlatos áttekintést adunk a **programspecifikációk**ról (amely alatt csak a **Neumann-elvű programok** specifikációját értjük), érintve annak **történet**ét is, vázolva két, legfontosabb irányzatát: a **modell-elvű**t és az **algebrai**t. Részletesebben bemutatjuk az ELTE Informatika Tanárszakán folyó oktatás ez irányú elképzeléseit. Közben összehasonlítjuk az egyik legismertebb modell-elvű formalizmussal, a **VDM-SL**-lel.*

1. Epizódok a programspecifikációk történelméből

E fejezetben egy szubjektív történeti beszámolót olvashatunk a programspecifikációkról. Nem törekszünk teljességre, hanem inkább arra, hogy a történetből a számunkra legfontosabb „gondolatokhoz” kapcsolódó események domborodjanak ki.

Őskor: *Algoritmus = program*

A programspecifikáció fogalma alighanem egyidős magával a program fogalmával. A „korai” specifikációk nem különböztek a programot „embrionális” állapotában leíró *algoritmus*tól, vagy annak esetleg egy felsőszintű megfogalmazásától, hiszen célja: a feladatmegoldásának pontosítása. Erre a célra megfelelt kezdetben valamilyen rajzos eszköz is, amely kicsiny „szókincsével” kellően egyértelműen fejezte ki a megoldás menetét.[1]

A szoftverkrízis kora: *Munkamegosztás* ⇒ *precízebb leírások, strukturált programozás* ***Algoritmikus absztrakció***

A szoftverkrízis idején, amikoris már a programkészítés ipari méreteket öltött, nagy jelentőségre tett szert, hogy mennyi időt kell eltölteni az effektív *kód*olás mellett a program*teszt*eléssel, a *helyesség* belátásával. Kiderült, hogy döntő jelentősége van annak, hogy milyen módon fogalmazzák meg a problémát, amely megoldásához keresik a programot. Az ok kézenfekvő: már nem ugyanaz a személy végzi a feladatmegfogalmazást, mint aki a megoldás algoritmusát tervezi meg, s megint más az, aki a konkrét programozási nyelvre ülteti át, sőt egy „negyedik résztvevő” végzi a programhelyesség ellenőrzését. Így a résztvevők csak akkor képesek minőségi munkát végezni, ha van olyan eszközük, amely kellően objektívan és precízen képes megfogalmazni a feladatot, amelyhez a munkájuk „viszonyítási pontjaként” tudnak

[1] Szándékosan nem említjük itt a nyelv *szintaxis*ának specifikációs eszközeit (pl. az ALGOL 60 leírásához kitalált BNF-et), hiszen a programspecifikáció alatt a működéssel, a *szemantiká*val kapcsolatos leírást értjük.

nyúlni. Ekkor fogalmazódnak meg a *strukturált programozás* elvei (felülről-lefelé programtervezés + strukturált programszerkezetek: szekvencia, elágazás, ciklus). A korszakot nyugodtan nevezhetjük az *algoritmikus absztrakció* korának is.

Az "első (absztrakt) fecske" = a VDL

Még a 60-as évek első felében az IBM úgy határozott, hogy egy új programozási nyelvet fejlesztenek ki, amely a FORTRAN-t és a COLBOL-t lesz hivatott fölváltani. A nyelv, amely először a „New Programming Language" nevet kapta[2], végül is egyszerűen Programming Language I (PL/I) néven vált híressé. Mivel eredendően „jól megtermettnek" tervezték, úgy döntöttek, leírására valamilyen formális technikát próbálnak alkalmazni. Sokak munkájának eredményeképpen született meg egy ún. *operációs szemantikadefiníciós nyelv*, az ULD3[3], amelyet a programozótársadalom mint Vienna Definition Language-t (VDL-t) tart számon. [Varga 81] Ezt tekinthetjük a specifikációs nyelvek első teljesnek mondható –bár „korai", ti. igazán nem bevált– változatának.

feladat *= mit* ***program*** *= hogyan* ***kapcsolat*** *= P/P-technika (helyesség-bizonyítás)*

A krízist követő kutatások eredményeként –többek közt– tisztázódott a programleírás (amely a *mit*re kérdésre adja meg a választ) és programmegvalósítás (amely a *hogyan*ra koncentrál) elválasztásának elkerülhetetlen volta. Az algoritmus megadására sokféle pszeudokód született (ALGOL 60, APL, LISP stb.), amelyet egy-egy szoftverkutató intézmény berkeiben fejlesztettek ki és használtak előszeretettel. Ezek egyike-másika nagyobb népszerűsége folytán publikációs nyelvvé is vált. (Pl. a 60-s évek ALGOL 60 korszakát a Pascal-éra követte.) [Horowitz 87] A „mit" és a „hogyan" szétválásával egy időben a *feladat* és a feladat egy lehetséges megoldását jelentő *program* fogalmi kettősének eltávolodás is megtörtént. Az eltávolodás ugyan lezajlott, de nem maradtak minden kapcsolat nélkül. A helyességbizonyítást célzó P/P-technikák[4] e kapcsolat létesítésére kiválóan alkalmasak. A bizonyítás során –többnyire– az algoritmikus struktúrák közé illesztett állítások közötti egymásra következést kell belátni, amelyre többféle nevezetes módszer (Floyd-féle invariánsok, Manna-féle ún. részcél, Hoare deduktív [l. függelék] stb.) létezik. [Varga 81] [Manna 81] (L. 1. függelék: „Helyességbizonyítás Hoare deduktív módszerével") Ez az absztrakciós lépés, és az addigra megerősödött formális programozási iskolák munkája nyomán kialakult programleíró eszközök és módszerek teremtették meg később a *programozási tétel*ek fogalmát.

VDM

73-74 körül Cliff Jones egy publikációjában (a Hursley Technical Reportban) az ALGOL 60-nak egy *funkcionális szemantikai leírásá*t adja. Itt je-

[2] ami ellen a National Physics Laboratories névrövidítés-egyezésre hivatkozva tiltakozott

[3] az ULDx (x=1, 2) specifikációs nyelveket a természetes nyelvek leírására használták

[4] P/P=Pre-/Postcondition, azaz az elő-utófeltételeken alapuló megközelítést 3 „nagy név" fémjelzi: Floyd, Hoare és Dijkstra.

lentek meg egy későbbi –számunkra különösen fontos– specifikációs eszköz (a VDM) egyes aspektusai. Ez az ideje a PL/I szabványosításának is. E munka, amelyben Bekiç, Bjørner, Henhapl, Jones és Lucas vettek részt, eredménye lett az 1974-es „Technical Report", azaz a PL/I formális leírása, s egyben a Vienna Development Method (VDM) kiindulópontja. Cliff Jones és Dines Bjørner vállalta magára annak bebizonyítását, hogy a VDM valami másra is alkalmas, mint fordítóprogramok technikai riportjainak elkészítése. [LNCS 61] Dines Bjørner csoportja a dán Műszaki Egyetemen fáradhatatlanul folytatta a VDM használatát nyelvek leírására. Ő és kollégái voltak a CHILL nyelv leírásának felelősei, és az Ada nyelv szemantikáját leíró dokumentum létrehozásában is nagy részben az ő munkájuk fekszik. [LNCS 98]

Programozási tételek

A programozási tételek szerencsésen ötvözik a „mit" és a „hogyan" fogalmakat a cél, ti. a feladatmegoldás érdekében. Sémákat jelentenek a programozó számára, amelyekből magabiztosan építheti föl a megoldó programot. A feladatot (és a programot is) olyan absztrakt gépnek tekintik, amelynek bemenetére és kimentére fogalmaznak meg elvárásokat. A programozási tételek egy állítást fogalmaznak meg (innen az elnevezése is), aminek helyességét elegendő egyszer bebizonyítani, s utána „csak" alkalmazni kell. Az állítás mindig efféleképpen hangzik: ha a *bemenet*re kerülő adatok teljesítik a rájuk kirótt ún. *előfeltétel*eket, akkor a mellékelt (a tételhez tartozó) absztrakt *algoritmus kimenet*i adatai teljesíteni fogják a terminálásnál az ún. *utófeltétel*t, ami nem más, mint az elérendő cél. A feltételek leírására –általában– a matematikai logika és a halmazelmélet eszköztárát használják föl. Az absztrakt algoritmus nyelvezete lehet bármely *strukturált program*leíró eszköz. (Pl. struktogram, valamilyen természetes nyelvű alapszavakból konstruált pszeudokód, vagy egy alkalmasan leegyszerűsített programozási nyelv.)

Programtranszformációk

A programozási tételeken alapuló programfejlesztés tehát a tervezés[5], majd helyességbizonyítás lépéskettőse helyett –elméletileg legalábbis– a tételek specifikációból történő generálás és a programtranszformálás lépéskettősét illeszti. A *programtranszformáció*k is bizonyíthatóan helyes sémák, itt azonban nem a feladat és a program kerül párba, hanem két program. Most is lehetnek a két programpár ekvivalenciájának (azaz a transzformáció elvégzésének) *előfeltétel*ei. [Harangozó 98]

[5] Az elkövetkezőkben nem különböztetjük meg az algoritmust és a neki megfelelő valamely programozási nyelven készült kódot, mondván a kódolás oly mechanikusan kapható meg az algoritmusból, hogy –majdhogy nem– egyenlőség jel tehető közéjük.

Az adat-absztrakció első aranykora:

A típus-fogalom

Jackson diagram

A következő fontos mérföldkőnek látszik a specifikáció fejlődése szempontjából –bár időben a fentiekkel párhuzamosan futnak az események– az adatok és a program viszonyának „egyensúlyba kerülése". A program végrehajtását szervező *struktúrák absztrahálódása*[6] már késznek látszik a 60-s évek végén, amíg ugyanez az adatok vonatkozásában még meglehetősen gyermeki naivitású. Ekkoriban a típusfogalom alig több, mint a memória egy tartományának „sablonos" leírása, csupán egy ábrázolási séma. Felismerik, hogy a típus ugyanolyan építő eszköze az adatok világának, mint az eljárások (függvények) az algoritmusok világának. Tehát ha kell, a feladatnak megfelelően a „semmiből" létrehozandókként kell tudni velük bánni. A kétféle absztrakció közötti közeledés első jelének tekinthetjük a Jackson-féle diagramok megszületését.

Az adat-absztrakció második aranykora:

A típus és a modul

Objektumok és osztályok

A *típus*fogalom „többszálú" fejlődése során születnek meg a következő meghatározó fogalmak: osztály, egységbe zárás, öröklés. Valóban az *objektum* az a fogalom, amely –megítélésem szerint– jelentőségében a strukturált programozás elvei, vagy a programozási tételek jelentőségével vetekedhet. A típus- (*objektumosztály*-) leírás stációját lehet tekinteni a specifikáció következő lépésének. Dahl annak a véleményének ad hangot [LNCS 428/1-11], hogy az objektum-elvűséget 2 fő cél motiválta: egyrészt jó strukturális megfelelés elérése a diszkrét eseményeket szimuláló program és a modellezendő rendszer között, másrészt hatékonyan újrafölhasználható programkomponensek konstruálása. A típusmegvalósítás eszközeként bevezetik a *modul*[7] fogalmat, amely képes elrejteni fekete dobozként a programozó elől azon „részletkérdéseket", amelyhez valójában nincs köze, s így nagyban növeli a típus használatának biztonságosságát.

Típusok specifikálása:

algebrai specifikációk

A probléma most tehát nem az, hogy hogyan fogalmazható meg a programvégrehajtás *jó végcél*ja, hanem, hogy a típushoz (az objektumosztályhoz) asszociált (a vele egybezárt) műveletek, mint egy majdani programhoz fölhasználandó eszközkészlet elemei, mikor tekinthetők jónak, egymással „jól harmonizálódónak". Erre az „összhangzat-leírásra" a korábban említett, elő- és utófeltételekre alapuló, ún. modell-bázisú specifikációs eszközök, módszerek nem igen alkalmazhatók. Az ötlet kézenfekvő: az operációk együttműködésének elvárásait kell állításokkal megadni, vagyis egyfajta *axiómarendszer*t kell mellékelni.[8] S ha egy típust reprezentáló-implementáló modult elkészítettünk, az a feladat marad, hogy bebizonyítsuk: a megvalósított műveletek kielégítik axiómarendszerünket. Ehhez illeszkedő specifikációs eszközöket nevezik *algebrai*aknak.

---

[6] A vezérlő szerkezetek, az eljárások-függvények-operátorok, mind az algoritmikus absztrakció eszközei többé-kevésbé megtalálhatók a programozási nyelvek szókincsében.

[7] A modul kifejezés helyett esetleg mást (pl. unit, package) használnak, de a lényeg mindig ugyanaz: az *egységbe zárás* és a „nem publikus" dolgok *elrejtése.*

[8] Ennek a specifikációs módszernek a kidolgozója: Guttag, még 1975-ből.

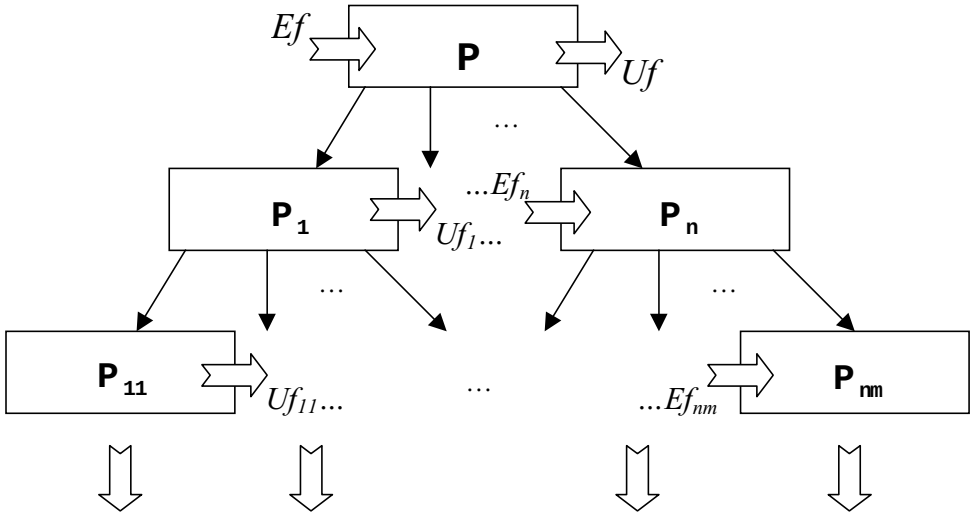
2. Feladatmegoldás – a kétféle megközelítés

Sarkosan szokás szembeállítani a programkészítés kétféle stratégiáját, a *felülről-lefelé* és az *alulról-felfelé* történő építkezés. A programgyártás módszertanában elsőként szokták említeni a strukturált programépítést, amely egyik fontos aspektusa a felülről-lefelé programkifejtés elve. Ez az elv azonban következetesen csak jól körülhatárolt, viszonylag kisméretű problémánál alkalmazható. Így az a vélemény, amely egyedül üdvözítőként említi az előbbit, hibás túlzásnak, a programozási tevékenység leegyszerűsítésének tekinthető. Ha bonyolult vagy adatok tekintetében komplex feladat megoldása a cél, akkor óhatatlanul fölvetődik az igény a másik irányú tervezésre is. Hiszen meg kell teremteni a strukturált programozás kivitelezéséhez oly fontos „mikrokörnyezetet", azon típusokat (objektumosztályokat), amelyek szókincsére támaszkodhatunk a felülről-lefelé problémalebontás (a programszintézis) során. Gyakorlatban nem számíthatunk ui. arra, hogy minden lehetséges „objektumosztály alap" készen rendelkezésünkre áll. (L. a következő oldali illusztrációt.)

A specifikáció szempontjából kialakult irányzatok közül tehát egy-egy feladat megoldásához többnyire több alkalmazására is szükség lehet. Egyik a feladat „egészének" logikáját igyekszik kézben tartani, valamilyen *modell-alapú* (*operációs*) specifikációval, míg a másik a „biztos alapként" fölhasznált típusok (objektumosztályok) helyességével foglalkozik célszerűen valamilyen *algebrai* leírás alkalmazásával. A következőkben e két specifikációkategória egy-egy képviselőjével foglalkozunk. (Vannak más specifikációs eszközök, sőt más „filozófiájúak" is.)

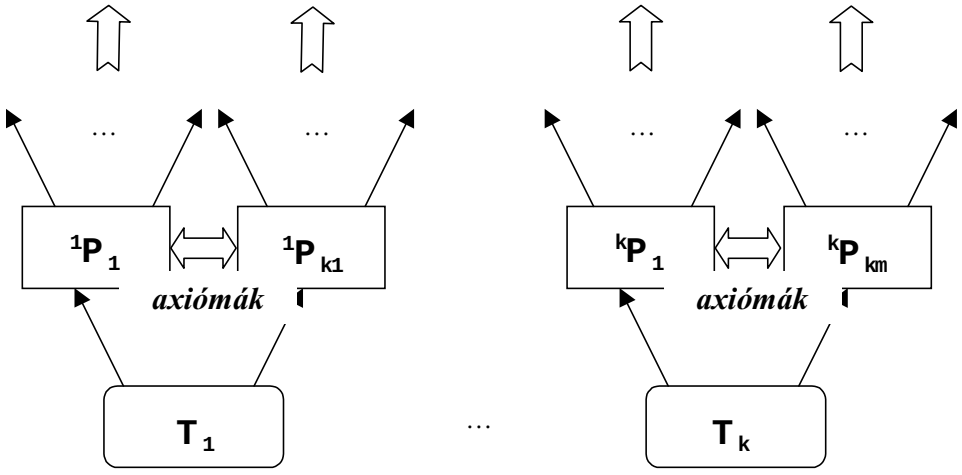
1 Felülről-lefelé algoritmustervezés

P = (fő)program; P_i = i. program (eljárás); ⇒ egymásra következés



2 Alulról-felfelé algoritmustervezés

T_i = i. típus; ^iP_j = i. típus j. operációja; ⇔ kapcsolat (axiómák)

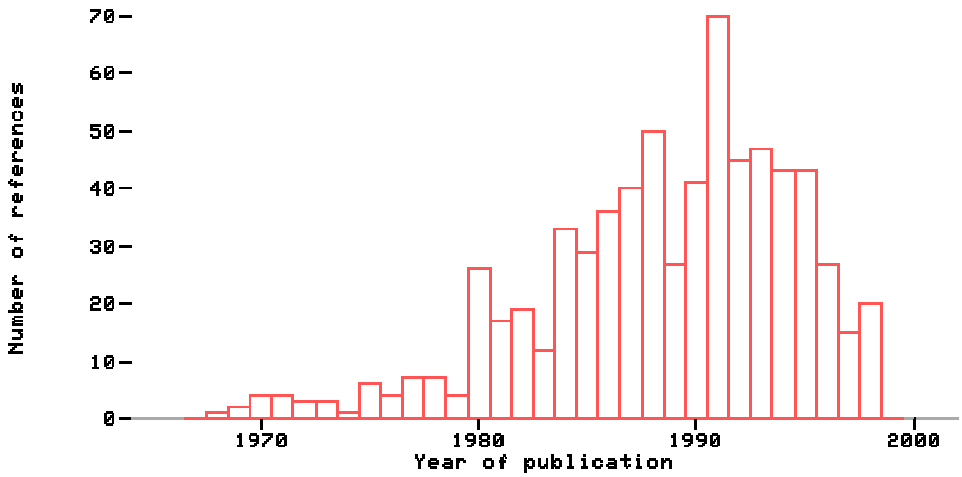


'Struktúra szerinti feldolgozás' elve
1 ⇔ 2
(mechanizálva: Jackson diagram)

3. „VDM-univerzum”

A VDM egy népszerű képviselője az ún. modell-alapú specifikációs módszereknek, sőt több – amint hamarosan kiderül.

A VDM viszonylag hosszú múltra tekint vissza: már a 70-s években is folyt vele kapcsolatos kutatás. A kutatás „intenzitását” mutatja be az alábbi grafikon. Amint látható az előzmények még a 60-as évek végére nyúlnak vissza, s fénykorát a 90-es évek elején élte.



VDM= definíciós és fejlesztési technika-gyűjtemény

Mára már a „VDM nem csupán egy a sok specifikációs nyelv közül. Helyesebb számítógépes rendszerek *formális definíciós* ***és*** *fejlesztési technikáinak gyűjteményé*nek tekinteni, amely egy *VDM-SL*-nek nevezett specifikációs nyelvből, adat- és műveletfinomítási szabályokból, valamint bizonyításelméletből áll” – írja Verhoef a VDM-ről feltett gyakori kérdésekre adott egyik válaszában. [Verhoef]

Tehát a VDM-SL mind *specifikációs*, mind pedig *fejlesztő nyelv*. Ebből következik egyik fő jellemvonása, amelyben különbözik a programozási nyelvektől: a VDM-SL specifikáció általában „lazán” specifikált, vagyis nem egy konkrét entitást (értsd ez alatt a specifikálandót, bármi is legyen az) definiál, hanem inkább egy olyan osztályát entitásoknak, amelyeket ki kell elégítsen. A formális szemantikában ezeket a specifikáció *modell*jének hívják.

A VDM-SL leírása *absztrakt szintax*issal készült, amely a nyelv struktúráját absztrakt szinten mutatja be. A standard két *konkrét szintaxis*t definiál. Az egyik tisztán matematikai, amely publikációs célú, a másik az ISO 646 szintaxis az elektronikus adattovábbítás korlátjait veszi figyelembe.

Jellegzetes nyelvezetére egy példát látunk a 2. függelékben.

A VDM-kutatásról

A fejezet címével arra a sokrétűségre, kidolgozottságra akarok utalni, amely a VDM-kutatást jellemzi, s mintegy önálló „galaxist” alkot a programozás tudományának nagy univerzumában (a rokon területekbe át-átnyúló „galaxiskarokkal”).

Számos helyen alakult a VDM-mel kapcsolatos *kutató centrum* (Dániában, Angliában, Németországban, USA-ban…). Ezekről kaphatunk némi képet, az alábbi „tematikus" csoportosítás révén, amelyek a legfontosabb centrumok honlapjainak címét tartalmazza[9]:

- VDM-ről „általában": „http://imada.ou.dk/vdm/vdm.html"
- A VDM-szabvány: „http://ftp.sei.cmu.edu/technology/standards/"
- Példagyűjtemény: „http://www.ifad.dk/pub/vdm/examples/examples.html"
- Bibliográfia: „ftp://imada.ou.dk/pub/docs/"
- Newsgroup: „comp.specification.misc"
- Levelezési lista: „mailbase@mailbase.ac.uk" *(subject: „join vdm-forum xy")*
- VDM újság: „http://www.ifad.dk/newsletter/..."

A VDM-szoftverek

A *VDM-szoftver*ek választéka is meglepően széles skálán mozog. A legegyszerűbbek képesek ellenőrizni egy specifikáció helyességét a VDM nyelvezete szempontjából, illetve igényes formában képesek nyomtatni. Ezek a publikációs célú eszközök (pl. SpecBox). A jobbak sokoldalúbb ellenőrzésekre képesek: szintaxis, típusmegfelelés, statikus szemantika (pl. Centaur-VDM environment). A szoftverek csúcsát jelentik azok, amelyek a kiterjedt ellenőrzés után képesek valamilyen programozási nyelvre lefordítani a specifikációt. Ilyen pl. az IFAD VDM-SL Toolbox, amely C++-kódgenerátorral és forrásnyelvi nyomkövetővel rendelkezik. Ráadásul többfajta platformon is képes futni: Sun SPARC SunOS/Solaris, HP 700 HPUX, PC/586 Win95, WinNT, Linux, DEC Alpha UNIX.

[9] Általában a felsorolt WWW-cím mellett létezik legalább egy anonymus FTP-elérés is.

4. Algebrai típus-specifikációk

Az előző részben az egyik népszerű modell-alapú specifikációs eszközről, a VDM-ről esett szó, most a másik nagyon fontos specifikációs „kategóriára", az *algebrai*ra vetünk néhány pillantást.

Az algebrai specifikációk ígéretes területe az absztrakt típusok definiálása. Az algebrai jelző arra utal, hogy formalizmusa nagyban épít az absztrakt algebra jelöléseire, formalizmusára. Guttag [Gehani 86/55-74] szerint a specifikáció rendszerint három részből áll: *szintaktikus* specifikációból, *szemantikus* specifikációból és *szűkítési* specifikációból. Az első biztosítja a szintaktikai és típusellenőrzési információkat az asszociált operációk neveinek, értelmezési tartományainak és értékkészleteinek megadásával (*szignatúra*). A következő az operációk „egymásra hatását" írja le axiómák segítségével. A szűkítési specifikáció foglalkozik az előfeltételekkel és kivételes helyzetek feltételeivel. Meg kell jegyeznünk, hogy gyakorta ez utóbbit a szerzők nem tartják igazán fontosnak, s ezért elhagyják a specifikációjukból. Az algebrai specifikáció technikájának bemutatására álljon itt két részlet Gehani egy cikkéből [Gehani 86/69-70, 173-185], amely során egy verem, ill. egy relációs adatbázis leírását adja: ***1.*** és ***2. specifikáció***.

```
type Stack[elem:Type, n:Integer]
  external operations newstack, push, pop, top, isnew, depth
  syntax
    newstack: → Stack
    push:  Stack x elem → Stack
    pop:   Stack → Stack
    top:   Stack → elem
    isnew: Stack → Boolean
    depth: Stack → Integer
    …
  semantics
    declare stk: Stack, elm:elem
    axioms
      pop(push(stk,elm)) = stk
      top(push(stk,elm)) = elm
      isnew(newstack) = true
      isnew(push(stk,elm)) = false
      depth(newstack) = 0
      depth(push(stk,elm)) = 1+depth(stk)
    …
  restrictions
    pre(pop,stk) = ~isnew(stk)
    isnew(stk) ⇒ failure(top,stk)
    failure(push,stk,elm) ⇒ depth(stk)≥n
end Stack.
```

1. specifikáció

```
type database
  external operations new, add, delete, empty, in, get
  internal operation insert
  syntax
    new: → database
    add: database x name x relation → database ∪ {ERROR}
    empty database → boolean
    in: database x name → boolean
    …
  semantics
    var d: database; n, m: name; r: relation
    axioms
      empty(new) = true
      empty(insert(d,n,r)) = false
      in(new,n) = false
      add(d,n,r) = if in(d,n) then ERROR else insert(d,n,r)
      …
end database.
```

2. specifikáció

Megjegyzések:

- A függvények mellékhatás nélküliek!
- Az `insert` operáció lényegét tekintve azonos az `add`-dal, csak nem végez ellenőrzést a reláció már meglétét illetően. Az `internal` kulcs-szó utal arra, hogy „kívülről nem látható”, csak „technikai szükséglet” az axiómák könnyebb megfogalmazása érdekében.
- Nem definiáltak az alaphalmazok: `database`, `name`, `relation`.

5. Egy „hazai” specifikációs nyelvről és fölhasználásáról

A következőkben röviden ismertetjük azt az eszközt, amelyet az ELTE Informatika Tanárszak Programozási módszertan tantárgyában használunk specifikációs nyelvként. A formalizmus definiálásakor arra törekedtünk, hogy –amennyire lehet– könnyen érthető legyen, precizitása a „józan ész” határai között maradjon, mivel nem volt célunk olyan eszközzé fejleszteni, amelyet automatikusan valamilyen nyelvre fordítani lehet.

5.1. Összehasonlítás a VDM-mel

Válasszuk a bemutatás módszeréül: egy ismert formalizmussal való egybevetést. Viszonyításként a VDM-mel állítjuk párba, pontosabban annak azzal a változatával, amelyhez a legigényesebb szoftver tartozik, így könnyen ellenőrizhető a helyesség. Azaz az IFAD VDM-SL. Példaként álljon itt egy-két programozási tétel mind a két formalizmussal:

IFAD VDM-SL

```
module DecisionPrTh
  parameters
    types
      elem
    functions
      t: elem -> bool
  exports
     all
  definitions
    functions
      decision(X: seq of elem)
                             ex: bool
        post ex = exists i in
              set inds X & t(X(i));
end DecisionPrTh
```

3. specifikáció

ELTE

Eldöntés $(H^*, \mathcal{T}(H,\mathbf{L})) : \mathbf{L}$

Be: $N \in \mathbf{N}$, $X \in H^N$, $T: H \to \mathbf{L}$
Ki: $VAN \in \mathbf{L}$
Ef: -
Uf: $VAN \equiv \exists i \in [1,N] : T(x_i)$
Def: -
Algoritmus:

```
  i:=1
  Ciklus amíg i≤N és nem T(X(i))
    i:+1
  Ciklus vége
```

4. specifikáció

Megjegyzések: 1) a VDM példában egy *modul* szerepel, amely nem feltétele a specifikációnak, de minden esetre hasznos lehetősége a IFAD VDM-SL-nek. 2) van olyan specifikálási lehetősége is a VDM-nek, amelyben az algoritmust *explicite* megadjuk, ekkor azonban a fenti *implicit* nem szerepelhet. 3) Az IFAD-változat az ISO 646 szabványt kielégítő konkrét szintaxissal készült. 4) Az ELTE-változatban az első sorát mint egy függvény *szignatúrá*ját kell tekintenünk, amely lehetővé teszi, hogy egy későbbi algoritmikus hivatkozásnál tömören építhessük abba bele. A „Be:”-től „Def:”-ig terjedő rész egy „klasszikus” *állapottér* megadás, amely halmazok, függvények deklarálását és esetleg definiálását (Be, Ki, Def), valamint szűkítését (Ef, Uf) tartalmazza.

Következő példa legyen egy típuskonstrukciós eszköz megadása mind a két formalizmussal (persze sok-sok kihagyással). L. ***5.*** és ***6. specifikáció***t.

IFAD VDM-SL

```
module P_queue
  parameters
    types Entry
    functions
      priority: Entry -> nat
  exports
    operations
      ENTER: Entry ==> ();
      EXTRACT: () ==> Entry |
                      QUEUE_EMPTY
  instantiations
    entries as Multiset(elem ->
                        Entry)
      types bag
      functions
        empty_bag: () -> bag;
        plus_bag: elem * bag -> bag;
        …
  definitions
    state P_queue of
      QUEUE: Entries'bag
      init mk_P_queue(q) ==
                 q=Entries'empty_bag
    end
    operations
      ENTER(e:Entry)
        ext  wr QUEUE
        post QUEUE=Entries'
                  plus_bag(q,QUEUE~);
      EXTRACT() e: Entry |
                     QUEUE_EMPTY
        ext  wr QUEUE
        post if QUEUE=Entries'
                        plus_bag then
                  e=QUEUE_EMPTY and
                          QUEUE=QUEUE~
             else
               …
end P_queue
```

5. specifikáció

ELTE

```
Típus PrSor(Elem, Prior):
        Asszociált műveletek:
  Üres: PrSor
  Üres?(PrSor):Logikai
  Elsõ(PrSor):(Elem x Prior) ∪
                                {NemDef}
  Sorba(PrSor,Elem,Prior):PrSor ∪
                                {NemDef}
  …
              Axiómák:
1º  Létrehozás után prioritási sor
üres.
      s=Üres  ⇔  Üres?(s)
2º  Az a prioritási sor, amelyben
legalább egy elem van, az nem Üres.
      nem Üres?(Sorba(s,e,p))
3º  Az üres prioritási sornak nincs
elsõ eleme.
      Elsõ(Üres)=NemDef
…
ExportModul PrSor(Típus TElem:
                  Típus TPrior):
  Eljárás Üres(Változó s:PrSor)
    Ef: -   Uf: s=()
  Függvény Üres?(Konstans s:PrSor):
                                Logikai
    Ef: -   Uf: Üres?((s1,…,sn))=n=0
  Függvény ElsőElem(Változó s:PrSor):
                                  Telem
    Ef: …   Uf: …
  Függvény ElsőPrioritás(
           Változó s:PrSor):TPrior
    Ef: …   Uf: …
  …
Modul vége.
Modul PrSor(Típus TElem:
            Típus TPrior):
  Reprezentáció
  …
  Implementáció
  …
  Inicializálás
  …
Modul vége.
```

6. specifikáció

Megjegyzés: 1) A példából látszik, hogy az ELTE-specifikáció három elkülöníthető részből áll (algebrai leírásból + algoritmikus „elvárások” megadásából + algoritmikus megvalósításból), amíg az IFAD egyetlen egységet alkot. 2) A nálunk használatos formalizmus elvi lehetőséget kínál a „leg-

felsőbb szintű" (DA)[10] elvárások és a „praktikusabb" algoritmikus (P/P) elvárások egymásnak megfelelését vizsgálására; ugyanerre nincs lehetőség a másik módszerben (bár kétségkívül a P/P-terén jobban kidolgozott). 3) Az ELTE-s specifikációban az ExportModul első sorát mint típus-deklarációs mintát kell értelmezni, azaz pl. az alábbi deklaráció eleget tesz ennek a mintának: **Változó** ps:**PrSor**(**Szöveg: Egész**). 4) Az operációk elő- és utófeltételei lehetőséget kínálnak ellenőrző információk elhelyezésére.

5.2. A specifikáció szerepe a programkészítésben [11]

***Programozási tétel**ek (PrT) feltérképezése az **elő-** és az **utó feltétel** alapján*

Nagyon sok szó eset már a szakirodalomban a specifikáció nélkülözhetetlen voltáról, így most inkább arról szólnék, hogy a specifikáció-alapú programkészítést mi miként képzeljük el. A feladat (P/P) specifikációhoz fordulunk több ízben is a programozás során. Kiinduló pontunk nyilvánvalóan ez, kérdés mégis: hogyan lehet „legveszélytelenebbül", a legtöbb információt kinyerni belőle. Válasz: fedezzük föl benne az esetleg rejtve megbújó *programozási tételkombináció*kat. Mivel az alaptételek száma nem túl nagy (<20), és pusztán a feladat bemeneti adatainak valamint a tételek bemeneti adatainak összevetése jócskán szűkíti a kört, ezért a megfelelő, helyesebben az elsőként alkalmazandó kiválasztása nem nagy gond. A folytatáshoz az *utófeltétel* némi rutin átalakítására esetleg szükség lehet, de nem reménytelen vállalkozás. Így transzformálgatva, redukálgatva az utófeltételt kapjuk az utófeltétel tételekre bontott ekvivalens változatát.

***Algoritmus-szekvenciák** képzése*

Ha az utófeltételt partícionáltuk, a tételek aktualizált *algoritmusainak szekvenciáj*át kell képeznünk. L. ***1. algoritmus***t.

Feladat: („Egyesítés" tétel)

Két sorozat egyesítése.

Specifikáció:

Egyesítés(H^*,H^*):H^*

Be: $N \in \mathbf{N}$, $X \in H^*$, $M \in \mathbf{N}$, $Y \in H^*$

Ki: $Db \in \mathbf{N}$, $Z \in H^*$

Ef: Halmazfölsorolás(X) $\wedge$ Halmazfölsorolás(Y)

Uf: $Db = M + \sum_{i=1}^{N} \chi(T(x_i)) \wedge Z \in H^{DB} \wedge$ Halmazfölsorolás(Z)

$\wedge \forall i \in [1,DB]: z_i \in X \vee z_i \in Y$

Def: $T(x) := x \notin Y$

[10] A szakirodalom a specifikáció ezen algebrai fajtáját *adatabsztrakció*nak (Data Abstraction) nevezi innen a fenti rövidítés.

[11] Ezt a fejezetet egy hálózaton „publikált" anyagra építettük: [Szlávi WWW].

Algoritmus:

Tételek egymásra építésével megfogalmazva:

```
Z:=Másol(1..N,X,ι); Db:=N
Z:+Kiválogat(M,Y,nem Eldönt(N,X,=))
```

Megjegyzések:

- ι az ún. identikus leképezés H-ról H-ra.
- „**:+**” sorozat egymásutánírás (Db kezdőértékét átvéve)
- Eldönt(N,X,=) eldönti, hogy a paramétere eleme-e az X(1..N) sorozatnak.

1. algoritmus

Optimalizálás *programtranszformációkkal*

Mivel ezek sok hatékonysági kívánni valót hagynak maguk után, ezért – bár egy megoldást már találtunk– nem állunk itt meg. *Programtranszformációk* segítségével ezek a „redundanciák” kiküszöbölhetők. L. ***2. Algoritmus***t. Vegyük észre, hogy eddig csak garantáltan helyes lépést használtunk, ha a feltételek mindig jól figyelembe vettük, akkor a kapott algoritmus jó megoldást ad. (Igaz, egyáltalán nem „eredeti” megoldást találtunk, de most a helyességet, s nem az eredetiséget tartottuk szem előtt.) [Harangozó 98]

Feladat:

adjuk meg a sorozat K. T tulajdonságú elemét, ha van ilyen.

Specifikáció:

Be: $N,K\in\mathbf{N},\ X\in H^*,\ T:H\to\mathbf{L}$

Ki: $VAN\in\mathbf{L},\ SORSZ\in\mathbf{N}$

Ef: $K\in[1,N]$

Uf: $$VAN\equiv\left(\sum_{i=1}^{N}\chi(T(x_i))\geq K\right)\wedge$$

$$VAN\Rightarrow SORSZ\in[1,N]\wedge T(x_{sorsz})\wedge\sum_{i=1}^{SORSZ}\chi(T(x_i))=K$$

Levezetés:

- Tételkombinációk azonosítása az elő- és utófeltétel alapján: *megszámolás + keresés*.
- A „megoldásfüggvény”:

```
(Van,SORSZ):=Keresés(N,X, i∈[1,N]:Megszámolás(i,X,T)≥K)
```

Az algoritmus 0. változatban (csak lényeg):

```
I:=1
Ciklus amíg I≤N és nem Megszámol(I,X)≥K
 I:=I+1
Ciklus vége
VAN:=(I≤N); Ha VAN akkor SORSZ:=I
```

... és a „Megszámol” függvény:

```
Függvény Megszámol(Konstans J:Egész,X:HTömb): Egész
 ...
 DB:=0
```

```
      Ciklus L=1-tõl J-ig
        Ha T(X(L)) akkor DB:=DB+1
      Ciklus vége
      Megszámol:=DB
    Függvény vége.
```

Programtranszformációkkal „egyszerűsíthető” [Szlávi PrT]*:*

```
    I:=0; DB:=0; VAN:=Hamis
    Ciklus amíg I<N és nem VAN
      I:+1; DB:=Megszámol(I,X)
      VAN:=DB≥K
    Ciklus vége
    Ha VAN akkor SORSZ:=I
```

A „Megszámol” függvény „behelyettesítése” után kapjuk az „elvárt” megoldást.

```
    I:=0; DB:=0; VAN:=Hamis
    Ciklus amíg I<N és nem VAN
      I:+1; Ha T(X(I)) akkor DB:+1
      VAN:=DB≥K
    Ciklus vége; Ha VAN akkor SORSZ:=I
```

2. algoritmus

Kódolás

A késznek vehető algoritmusból generáljuk ezután az adott programozási nyelvű kódot. Ahogy eddig is jó néhány sémát használtunk, semmi meglepő nincs abban, hogy most is ekként teszünk: *kódolási szabályok* mechanikus alkalmazásával kapjuk a kódot. Ebben a lépésben az inputnál kell adott esetben újból a *bemenet*nél, illetőleg az *előfeltétel*eknél előírtakat újból elővenni.

Tesztelés

A módszertani szempontból utolsóként érdekes lépéshez, a *tesztelés*hez is segítséget nyújt a specifikáció. Pontosabban a *fekete doboz* módszer szerintihez. A határesetek a *bemenet*- és az *előfeltétel*-részből lehet „generálni”, míg a legmagasabb szintű ekvivalencia-osztályok az *utófeltétel* alapján képezhető.

Modul-implementálások

Megjegyzés: ha az utófeltételben már komplexebb szerkezetek fölbukkannak, akkor értelemszerűen a fentiek kiegészülnek a megfelelő modulok (specifikáció-alapú) implementálásával.

5.3. Problémák

Ebben a fejezetben felsorolom azokat a problémákat, amelyek a gyakorlati (tehát nem „oktatástiszta” feladatok) feladatok megoldása során felvetődnek, s némelyikre könnyű megoldás nem ismert rá.

- Utófeltétel-partícionálás

  Például a ***2. algoritmus***ban:

  Uf: $\mathrm{VAN} \equiv (\sum_{i=1}^{N} \chi(T(x_i)) \geq K) \wedge$

  $\mathrm{VAN} \Rightarrow \mathrm{SORSZ} \in [1,N] \wedge T(x_{sorsz}) \wedge \sum_{i=1}^{SORSZ} \chi(T(x_i)) = K$

  fölfedezni a *megszámolás* és a *keresés* tételt.
- Programtranszformációk mechanikus alkalmazása a hatékony algoritmus megtalálásához
- Futás közbeni feltétel ellenőrzések

  Programozásnyelvi támogatások – C, Pascal preprocesszálás útján beszerkesztett invariánsok ki-, ill. bekapcsolása.
- Modulok helyességének ellenőrzése (axiómák és elő-utófeltételek „kompatibilitásának" ellenőrzése)

  Egy-egy modul konkrét ellenőrzése cél Prológ program segítségével.

6. Konklúzió

Előadásomban szándékom szerint némi bepillantást nyújtottam a neumann-i programozási nyelveken történő programfejlesztés specifikáció-alapú módszereibe. Közülük egyről kicsit részletesebben is szóltam, azzal a szándékkal, hogy a specifikáció hasznos gyakorlati szerepét ecseteljem. Ez a módszer egyike azoknak, amely a programkészítés operációs és algebrai specifikációját együttesen alkalmazza.

7. Irodalomjegyzék

[Varga 81] Varga L.: *Programok analízise és szintézise.* Akadémiai Kiadó, 1981

[Horowitz 87] E. Horowitz: *Magasszintű programozási nyelvek.* Műszaki Könyvkiadó, 1987

[Manna 81] Z. Manna: *Programozáselmélet.* Műszaki Könyvkiadó, 1981

[Harangozó 98] Harangozó É., Szlávi P., Zsakó L.: *Joining Programming Theorems; a Practical Approach to Program Building*. Annales Univ. Sci.,1998

[LNCS 61] *VDM – Meta language*. Springer-Verlag, 1978

[LNCS 98] *Towards a Formal description of Ada*. Springer-Verlag, 1980

[LNCS 428] D. Bjørner, C.A. Hoare, H. Langmaack (szerk.): *VDM'90, VDM and Z – Formal Methods in Software Development. Third International Symposium of VDM Europe Kiel*, Springer-Verlag, 1990

[Verhoef] M.,Verhoef: *VDM Frequently Asked Questions*. „marcel@dutct05.tudelft.nl", „vdm-forum-request@mailbase.ac.uk", 1995

[Gehani 86] N. Gehani, A.D. McGettrick (szerk.): *Software Specification Techniques.* Addison-Wesley, 1986

[Szlávi WWW] Szlávi P.: Programozási tételek szerepe a gyakorlati programozásban. http://izzo.inf.elte.hu/~szlavi, 1997

[Szlávi PrT] Szlávi P.: Programtranszfotmációk. http://izzo.inf.elte.hu/~szlavi, 1997

[Fitzgerald] J. Fitzgerald, G. Larsen: *Modelling Systems: Practical Tools and Techniques in Software Development*. Cambridge University Press, 1998

FÜGGELÉK

1. Helyességbizonyítás Hoare deduktív módszerével.

Állítások:

$\{p(x,y)\}$ B $\{q(x,y)\}$ (p igaz B előtt, q B után.)

Levezetési szabályok:

$$\frac{\alpha}{\beta}$$

(α -- feltétel, β -- következmény; olvasd: „α-ból β levezethető”)

Értékadás-axióma:

$\{p(x,g(x,y))\}$ $y \leftarrow g(x,y)$ $\{p(x,y)\}$

2. Feltételes szabályok:

$$\frac{\{p \wedge t\}\ B1\ \{q\} \text{ és } \{p \wedge \neg t\}\ B2\ \{q\}}{\{p\}\ \text{if t then B1 else B2}\ \{q\}}$$

…

3. While-szabály:

$$\frac{\{p \wedge t\}\ B\ \{p\}}{\{p\}\ \text{while t do B}\ \{p \wedge \neg t\}}$$

4. Kompozíciós szabály:

$$\frac{\{p\}\ B1\ \{q\} \text{ és } \{q\}\ B2\ \{r\}}{\{p\}\ B1; B2\ \{r\}}$$

5. Következmény szabályok:

$$\frac{\{p\} \supset \{q\} \text{ és } \{q\}\ B\ \{r\}}{\{p\}\ B\ \{r\}} \qquad \frac{\{p\}\ B\ \{q\} \text{ és } \{q\} \supset \{r\}}{\{p\}\ B\ \{r\}}$$

Értékadás-szabály:

$$\frac{\{p(x,y)\} \supset \{p(x, g(x,y))\}}{\{p(x,y)\}\ y \leftarrow g(x,y)\ \{p(x,y)\}}$$

$\{\varphi(X,N)\}$

```
1    i:=1
2    Ciklus amíg i≤N és nem T(X(i))
3      i:+1
4    Ciklus vége
5    Van:=i≤N
```

$\{\psi(X,N,Van)\}$

$\varphi(X,N)$: - -- *az előfeltétel*

$\psi(X,N,Van)$: $Van \equiv \exists j \in [1,N] : T(x_j)$ -- *az utófeltétel*

1 i:=1

Legyen $R(X,N,i) : i \in [1,N+1] \wedge \forall j \in [1,i) : \neg T(x_j)$

1. lemma: $\varphi(X,N) \supset R(X,N,1)$ (triviális)

Értékadás-szabály 1-re és 1. lemma $\Rightarrow$

$\{\varphi(X,N)\}$
`i:=1`
$\{R(X,N,i)\}$

3 i:+1

2. lemma: $R(X,N,i) \wedge i \leq N \wedge \neg T(x_i) \supset R(X,N,i+1)$

ui.: $(i \in [1,N+1] \wedge \forall j \in [1,i) : \neg T(x_j)) \wedge i \leq N \wedge \neg T(x_i)$
$\equiv (i \in [1,N+1] \wedge \forall j \in [1,i+1) : \neg T(x_j)) \wedge i \leq N$
$\supset R(X,N,i+1)$

Értékadás-szabály 3-ra és 2. lemma $\Rightarrow$

$\{R(X,N,i) \wedge i \leq N \wedge \neg T(x_i)\}$
i:+1
$\{R(X,N,i)\}$ (*azaz R ciklusinvariáns*)

2 **Ciklus …**
3 i:+1
4 **Ciklus vége**

While-szabály 2-4-re $\Rightarrow$

$\{R(X,N,i)\}$
`Ciklus amíg` `i≤N` **`és nem`** `T(X(i))`
`i:+1`
`Ciklus vége`
$\{R(X,N,i) \wedge (i=N+1 \vee T(x_i))\}$

1 i:=1
2 **Ciklus …**
3 i:+1
4 **Ciklus vége**

Kompozíciós szabály 1-4-re $\Rightarrow$

$\{\varphi(X,N)\}$
`i:=1`
`Ciklus amíg` `i≤N` **`és nem`** `T(X(i))`
`i:+1`
`Ciklus vége`
$\{R(X,N,i) \wedge (i=N+1 \vee T(x_i))\}$

5 Van:=…

3. lemma: $R(X,N,i) \wedge (i=N+1 \vee T(x_i)) \supset (\exists j \in [1,N] : T(x_j)) \equiv i \leq N$

ui.: $(R(X,N,i) \wedge T(x_j)) \vee (R(X,N,i) \wedge i=N+1)$
(*sorrendcsere és disztributivitás*)
$\equiv (i \in [1,N+1] \wedge \forall j \in [1,i) : \neg T(x_j) \wedge T(x_i))$
$\vee (i \in [1,N+1] \wedge \forall j \in [1,i) : \neg T(x_j) \wedge i=N+1)$
$\equiv (...)$
$\vee (i \in [1,N+1] \wedge i=N+1 \wedge \forall j \in [1,i) : \neg T(x_j))$ (*átrendezve*)
$\supset (\exists i \in [1,N] : T(x_j))$
$\vee (i=N+1 \wedge \forall j \in [1,i) : \neg T(x_j))$
(*A,B egymást kizáróak*: $A \oplus B \supset A \equiv \neg B$)
$\supset \exists i \in [1,N] : T(x_j) \equiv \neg(i=N+1 \wedge \forall j \in [1,i) : \neg T(x_j))$
($A \wedge B \supset A$)
$\equiv \exists i \in [1,N] : T(x_j) \equiv \neg(i=N+1) \equiv i \leq N$

```
1 i:=1
2 Ciklus …
3   i:+1
4 Ciklus vége
5 Van:=…
```

Értékadás-szabály 5-re és 3. lemma ⇒

$\{R(X,N,i) \wedge (i=N+1 \vee T(x_i))\}$

`Van:=i≤N`

$\{(\exists j \in [1,N] : T(x_j)) \equiv Van\}$

Kompozíciós szabály 1-5-re ⇒

$\{\varphi(X,N)\}$

```
i:=1
Ciklus amíg i≤N és nem T(X(i))
  i:+1
Ciklus vége
Van:=i≤N
```

$\{(\exists j \in [1,N] : T(x_j)) \equiv Van\}$

$\equiv \{\psi(X,N,Van)\}$ (*az ekvivalencia szimmetriája*)

Ezzel a parciális helyességet láttuk be.

2. Egy VDM-SL specifikációs példa [Fitzgerald].

A következő rövid példa a VDM zamatát igyekszik megízleltetni. Egy italárusító automata nagyon absztrakt („idealizált") specifikációját mutatja be: ez –egyelőre– nem specifikált fajtájú italokat kínál föl, a vásárló csak „korrekt" érméket dobhat be, kiválaszthat egy italt, majd az automata kiadja a választott italt és az –esetlegesen– visszajáró pénzt.

```
types
      Drink = token            -- egy még nem specifikált típus
      Money = ℕ                -- természetes szám
state Vending-Machine-1 of
      BALANCE: Money           -- a bedobott pénz mennyisége
      PRICES: Drink →m Money   -- az egyes italok ára
      inv   mk-Vending-Machine-1(-,p) ≜ ∀d:Drink • d∈dom p
            -- minden italfajtának van ára;
            -- p jelenti az „árjegyzéket"
      init  mk-Vending-Machine-1(b,-) ≜ b=0
            -- kezdõállapot: nincs pénz bedobva;
            -- a b bedobott pénzt jelenti
operations
      INSERT-MONEY(cash:Money)
      -- a vásárló bedob valamennyi pénzt
      ext wr      BALANCE: Money
      -- a mûvelet megváltoztatja a BALANCEt
      post  BALANCE=BALANCE~ +cash
      -- a mûvelet növeli a bedobott összeget
      GET-DRINK(choice:Drink) return:Drink×Money
      -- a vásárló választ italt,
      -- és kap italt meg visszajáró pénzt
      ext wr      BALANCE: Money
          -- a mûvelet megváltoztatja a BALANCEt
          rd      PRICES: Drink →m Money
          -- és olvassa az ártáblázatot
      pre   BALANCE≥PRICES(choice)
      -- a mûvelet végrehajtásának feltétele
      post  (BALANCE=0) ∧
            (return=mk-(choice,BALANCE~ -PRICES(choice)))
      -- a mûvelet hatása: kiüriti a BALANCEt,
      -- kiadja a kért italt és a visszajárót
```

Néhány magyarázatot fűzűnk a fenti programhoz.

- A példa a legtipikusabb, ún. flat-VDM specifikációs példa.
- A Drink és a Money egy-egy típus.
- A „--" jellel bevezetett szöveg megjegyzés.
- A BALANCE és a PRICES a modell paraméterei, amelyek a teljes modellben elérhetők, tehát *globális* adatként viselkednek.
- A „PRICES: Drink →m Money" paraméter egy véges leképezést definiál az italok és az árak között.
- Az „**inv** mk-Vending-Machine-1(-,p) ≜ ∀d:Drink • d∈**dom** p" egy mindig igaz, ún. invariáns állítást fogalmaz meg a modellben.
- Az „**init** mk-Vending-Machine-1(b,-) ≜ b=0" állítás a modell kezdőértékét határozza meg.

- Ez a két állítás egyben példa az ún. *explicit* függvénydefiniálásra. Erre utal a „$\triangleq$” jel.
- Az INSERT-MONEY és a GET-DRINK a modell két művelete. Az utóbbi függvényszerűen használható. Ez a szignatúrájából olvasható ki: „GET-DRINK(choice:Drink) return:Drink**×**Money”.
- Az **ext**-tel bevezetett rész deklarálja, hogy a műveletek mely modell-paraméterre hogyan hatnak (**wr**/**rd**). Észrevehetünk itt némi redundanciát, hisz változatlanul meg kell ismételni a paramétermegadást. [1/134]
- A **post** és a **pre** részek az egyes műveletek utó- és előfeltételeit fogalmazzák meg egy-egy állítás formájában. Ezek *implicit*e adják meg a kritériumot ahelyett, hogy pl. a kiszámítás módját rögzítenék.