IZKUŠNJA S PROLOGOM KOT JEZIKOM ZA SPECIFIKACIJO INFORMACIJSKIH SISTEMOV

DAMJAN BOJADŽIJEV,
NADA LAVRAČ,
IGOR MOZETIČ

UDK: 681.3.0.6:007

INSTITUT JOŽEF STEFAN, JAMOVA 39, LJUBLJANA

Visokonivojski programski jezik PROLOG smo preizkusili kot specifikacijsko orodje pri razvoju zahtevne aplikacije. Članek najprej opisuje problem specifikacije kompleksnih programskih paketov. Nato obravnava prednosti uporabe logike kot formalnega specifikacijskega jezika in PROLOGa kot enostavnega izvedljivega logičnega formalizma specifikacij. Opisano je naše izkušnje pri pretvorbi ISAC specifikacij računalniško podprte konkretnega skladiščnega sistema v PROLOGov program in povratni učinek testiranja PROLOGovega programa na spremenjanje in poglabljanje specifikacij. Podana je naša ocena PROLOGa kot zelo uporabnega specifikacijskega jezika - ob tem navajamo njegove konkretno ugotovljene prednosti in slabosti.

EXPERIENCE WITH PROLOG AS AN INFORMATION SYSTEMS SPECIFICATION LANGUAGE. We have used the high-level programming language PROLOG as a specification tool for the development of a complex application system. The article first describes the problem of specifying complex program packages. Then it describes the advantages of using logic as a formal specification language and PROLOG in particular as a simple runnable formalism. We describe our experience with transforming the ISAC specifications of a computerised warehouse system into a PROLOG program. We also mention the feedback from testing and demonstrating the PROLOG program to the change and further development of specification. The advantages and some weaknesses of PROLOG as a specification language are presented, supporting our opinion that PROLOG is a very useful and efficient specification tool.

1. UVODNI OPIS PROBLEMA

Programski jezik PROLOG smo preizkusili kot specifikacijsko orodje pri razvoju računalniškega sistema za podporo poslovanja skladiščno prodajnega centra Slovenijales v Črnučah. Serva smo se lotili programiranja poenostavljenega modela skladiščnega poslovanja v PROLOGu zaradi želje po testiranju hitrosti programiranja ter obnašanja PROLOGovega programa pri simulaciji nekega realnega procesa. Ker pa se je izkazalo, da lahko PROLOG uporabimo kot zelo učinkovit specifikacijski jezik (s stališča hitrosti programiranja), smo nadaljevali s programiranjem vse realnejšega modela skladišča. Pričakujemo, da nam bo podrobno razdelani program v PROLOGu trdno vodilo pri implementaciji sistema v izbranem programskem jeziku (PASCAL).

2. PROBLEM SPECIFIKACIJE PROGRAMSKIH SISTEMOV

Razvoj korektnega programskega paketa poteka vsaj skozi štiri faze [3]:

- razumevanje problema,
- formalna specifikacija,
- programiranje,
- preverjanje pravilnosti.

V fazi razumevanja problema si načrtovalec v interakciji z uporabnikom ustvari intuitivno sliko problema in izbere pristop k njegovemu reševanju. V naslednji fazi mora jasno in nedvoumno izraziti svojo intuitivno interpretacijo problema v izbranem specifikacijskem jeziku. Na osnovi specifikacije se izdela program, katerega pravilnost pa je potrebno preveriti.

V idealnem primeru bi potekala realizacija sistema zaporedno po opisanih fazah, pri čemer bi se v vsaki fazi vztrajalo pri testiranju in odpravljanju napak do take mere, da se ne bi bilo treba vračati na prejšnje. V praksi pa ne moremo dovolj podrobno vsebinsko preveriti formalnih specifikacij, da bi ugotovili, če ustrezajo uporabnikovim željam. Specifikacije namreč ponavadi niso izvedljive, ročno preverjanje pa je mukotrpen posel. Zato raje začnemo s programiranjem kljub zavesti, da specifikacije najbrž še niso dokončne. V primeru sprememb to povzroči preprogramiranje sistema, kar utegne biti zelo zamudno in neustvarjalno opravilo.

Dejansko je najtežja in najkreativnejša faza v razvoju programskega sistema prav izdelava sprejemljivih formalnih specifikacij iz intuitivne interpretacije problema. Zato je koristno izbrati tako orodje, s katerim je konstruiranje in preverjanje specifikacij čimbolj olajšano. To preverjanje seveda ne more pomeniti dokazovanja pravilnosti specifikacij glede na našo intuitivno sliko, temveč le ugotavljanje ali se na testiranih primerih obnašajo tako, kot smo si želeli in predvideli.

3. POMEN FORMALNIH IN IZVEDLJIVIH SPECIFIKACIJ

Uporaba formalnih specifikacij [10] navaja načrtovalca sistema k odstranjevanju nejasnosti, dvoumnosti in skritih protislovij prvotnih specifikacij, podanih v pogovornem jeziku. Zato ni čudno, da logika že dalj časa služi kot preverjeno formalno specifikacijsko orodje. Logika je lahko razumljiva, opisna in s svojim mehanizmom dokazovanja izrekov omogoča odkrivanje protislovij, ki so se ohranila v formalnih specifikacijah.

če je specifikacijski jezik tudi direktno (strojno) izvedljiv, lahko brez dodatnih naporov preverimo, kako se bo sistem obnašal v posameznih primerih. Tedaj postane metoda "pa poslejmo!" zares operativna, preverjanje intuitivnega razumevanja problema in korektnosti predlagane rešitve v komunikaciji z uporabnikom pa dobi konkretno, "otipljivo" izhodišče. Izvedljivost specifikacij tako omogoča simulacijo obnašanja sistema pred njegovo dokončno, "zaresno" implementacijo, kar prinese vrsto očitnih prednosti.

V primeru specifikacij, podanih v kakšnem logičnem formalizmu, je njihova izvedljivost dana z mehanizmom dokazovanja izrekov [5]. Vprašanje, kakšno bo obnašanje sistema v posameznem primeru, se namreč prevede na vprašanje, ali je formalni zapis pravtnega vprašanja izrek logičnega sistema, katerega tvorijo specifikacijski stavki. Tako lahko preverimo, ali se bo sistem v določenem primeru res obnašal tako, kot si to želimo:

(Ali) vhodu I ustreza izhod I ?

Na enak način lahko izvemo, kakšno obnašanje predvidevajo specifikacije pri določenem vhodu:

(Kateri so izhodi X, da) vhodu I ustreza X ?

Tako lahko enostavno odkrijemo nepravilnost ali nepopolnost specifikacij in jih ustrezno spremenimo.

Opisani način preverjanja specifikacij je sotovo hitrejša in učinkovitejša alternativa običajnemu preverjanju programov, saj

- od uporabnika dobimo povratne informacije dovolj zgodaj
- omejimo se zgolj na vsebinske spremembe
- pri tem se nam ni potrebno ozirati na učinkovitost.

Ker se v komunikaciji z uporabnikom prepričamo v ustreznost (popravljene) formalne rešitve, nam pri preverjanju pravilnosti končnega programa, v nasprotju z običajnim preverjanjem programa, preostane le še odkrivanje programerskih napak.

Uporaba logičnih formalizmov ne narekuje izbire programskega jezika pri končni implementaciji. Dovolj podrobna logična specifikacija se sicer s stališča uporabnika razlikuje od končne verzije sistema načelno samo po učinkovitosti. Drugače rečeno, logična specifikacija je že lahko znosno učinkovita programska rešitev načrtovanega sistema. Izvedljivost logičnih specifikacij tako zabriše tradicionalno razliko med specifikacijo in ustreznim programom [4], [11].

4. PROLOG KOT SPECIFIKACIJSKI JEZIK

Logični formalizmi, med njimi standardno predikatni račun, so bili, vsaj do pojava PROLOGa, širše sprejeti predvsem kot specifikacijski jeziki. Možnost direktne uporabe npr. predikatnega računa kot programskega jezika je v praksi ostala neizkoriščena zaradi neučinkovitosti ustreznih dokazovalcev izrekov. Ta situacija se je spremenila s pojavitvijo PROLOGa [2], ki implementira šibkejši, a izrazno dovolj naraven in uporaben podjezik predikatnega računa. Predikatni račun je v PROLOGu omejen na tim. Hornove stavke, ki izražajo posojne trditve (implikacije) tipa

če Pi in P2 in ... in Pn, potem P

kar se npr. v DEC-10 PROLOGovi sintaksi [6] zapiše kot

P :- P1, P2, . . . , Pn.

V posebnem primeru n=0 so to brezpogojne trditve ali dejstva, kar se v PROLOGovem zapisu izrazi s

P.

PROLOGova omejitev na Hornove stavke je koristna predvsem zato, ker ob enostavni kontrolni strukturi omogoča učinkovito izvajanje sklepov, ki sledijo iz postavljenih trditev. S tem pridobi Hornova logika status visoko-nivojskega, deklarativnega programskega jezika. Logika Hornovih stavkov je tudi bližja standardnim formalizmom računalniške znanosti po svoji proceduralni interpretaciji. Ker namreč Hornovi stavki reduciranje reševanje problema P na reševanje podproblemov P1,...,Pn, lahko v funkciji Pi-jev (in v načinu njihovega aktiviranja) prepoznamo elemente funkcije (in delno načina klicanja) podprogramov v klasičnem programskem jeziku. S tem se načelno zmanjša razlika med specifikacijo problema v PROLOGu in njegovo implementacijo v nekem drugem programskem jeziku.

5. OPIS PROBLEMA IN PROGRAMIRANJE POENOSTAVLJENEGA MODELA V PROLOGU

Skladiščno prodajni center Slovenijales v črnučah potrebuje računalniško podporo poslovanja velikega paletnega skladišča, v katerem se skladišči veliko število različnih vrst artiklov. To skladišče bo prvo v bodoči mreži Slovenijalesovih računalniško vodenih skladišč. Ker bo za komercialne funkcije ter za planiranje nabave in prodaje skrbel centralni informacijski sistem, je računalniška podpora skladišča omejena na vodenje prevzema in izdaje blaga, inventuro, obravnavanje reklamacij, vodenje zalog (količinsko in prostorsko po paletah) ter komunikacijo s centrom.

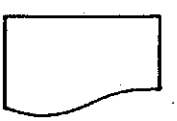
Pri razvoju sistema smo naleteli na precejšnje težave. Omenimo samo slabo definiranost načina poslovanja bodočega skladišča s strani uporabnika, kar je potegnilo za seboj veliko število sprememb specifikacij. Zato smo že po nekajkratnem spreminjanju specifikacij začutili potrebo po poenostavljenem in lahko spremenljivem modelu poslovanja skladišča.

V začetni fazi specificiranja problema smo kot orodje uporabili ISAC metodologijo. Ta metodologija se je izkazala kot koristna pri komunikaciji z uporabnikom zaradi možnosti grafične predstavitve problema. ISAC grafi namreč omogočajo strukturiran pregled nad aktivnostmi ter pretoki materialov in informacij v sistemu [9].

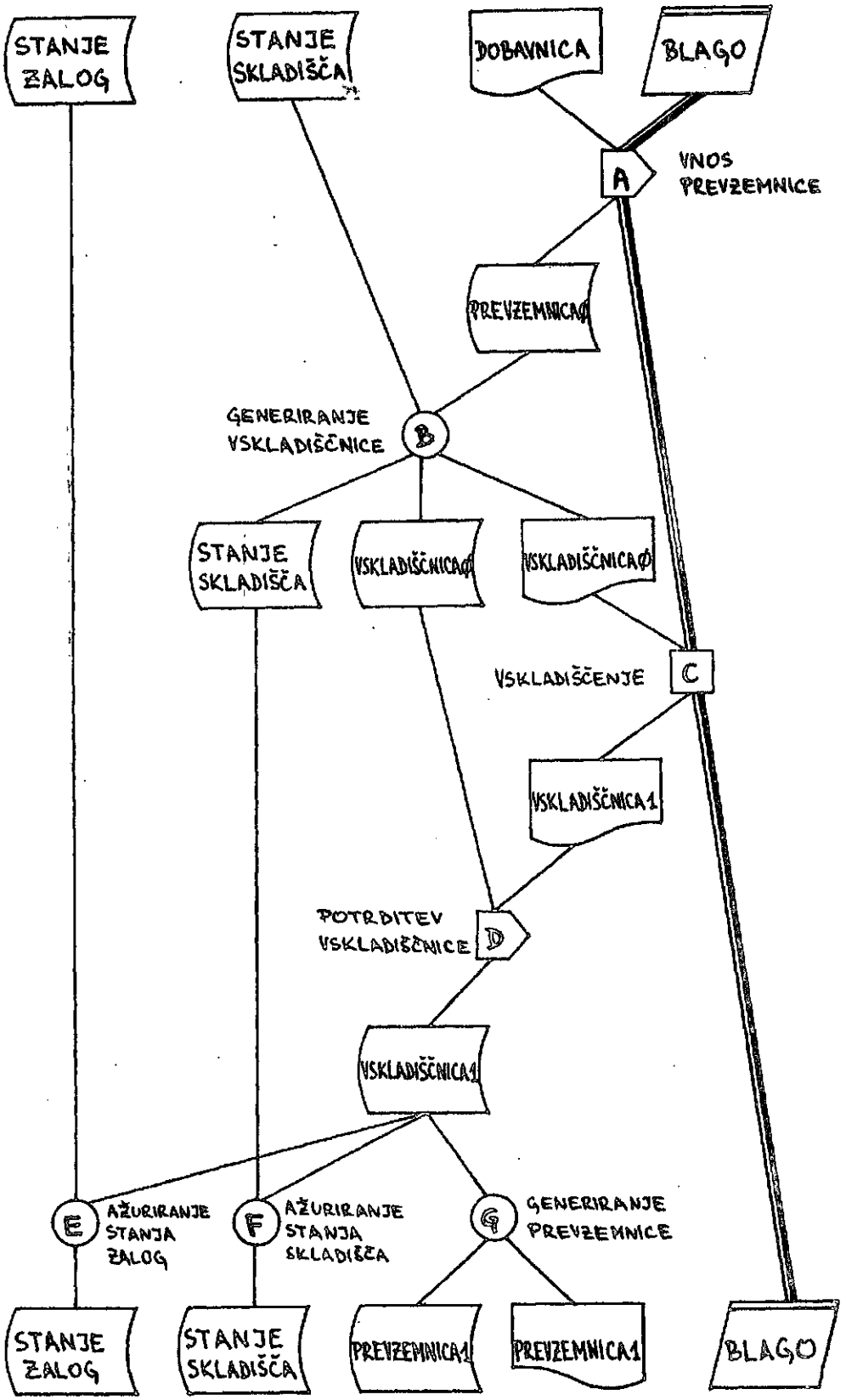
Slika prikazuje del ISAC grafa, ki poenostavljeno modelira prevzem blaga. V grafu smo uporabili naslednje simbole:

- za vhodno - izhodne množice:

datoteka



listina



V akciji A skladiščnik preko terminala vnese šifre prevzetih artiklov. Na osnovi vnesenih podatkov v akciji B sistem predlaga paletna mesta, na katera naj se blago vskladišči. Izpiše se dokument vskladiščnica, na podlagi katerega se blago fizično vskladišči. Če iz kakih razlogov vskladiščenje na predvidena paletna mesta ni bilo mogoče, se spremembe registrirajo v akciji D. Na podlagi tako popravljene vskladiščnice so avtomatsko ažurirata stanje zalog (kumulativne količine artiklov) in stanje skladišča (količine artiklov po paletah), generira in izpiše se prevzemnica.

- za aktivnosti:
  -  avtomatizem
  -  interakcija človek-stroj
  -  ročno

Zaradi nenehnega spreminjanja specifikacij je uporaba ISAC metodologije zahtevala poleg kreativnega še ogromno ročnega dela z dokumentiranjem razvoja v različnih besedilih, tabelah in grafih. Poleg tega nam s tako predstavljeno problematiko ni uspelo poglobiti specifikacij od nekega nivoja dalje.

V tej fazi razvoja sistema smo začutili potrebo po drugačni sistematizaciji funkcij in gradnikov sistema. Ob določitvi gradnikov smo usmerili, da se funkcije lahko realizirajo z različnimi kombinacijami teh gradnikov. Ker se je izkazalo, da je v PROLOGu enostavno realizirati in sestavljati osnovne gradnike, smo se lotili programiranja teh gradnikov in sestavljanja poenostavljenega modela sistema.

Za pisanje programa, ki je obsegal približno 150 vrstic PROLOGove kode, smo porabili pičla dva dneva. Zaradi hitrosti in preprostosti programiranja v PROLOGu smo model razvijali naprej. Osnovne postopke smo približali realnim zahtevam ter uvedli še preostale funkcije sistema. V naslednji fazi smo v programu uporabili tako organizacijo podatkov, kakršna bo realizirana z izbranim sistemom za delo s podatkovnimi bazami (TOTAL). V komunikaciji z uporabnikom smo se približali bodoči realni verziji in razdelali spremljajoče podatkovne strukture. Opisani model, katerega smo realizirali v približno enem mesecu, je obsegal približno 1000 vrstic PROLOGove kode. Pri demonstraciji delovanja modela uporabniku smo (po pričakovanju) dobili veliko novih informacij, ki so narekovale nadaljnje spreminjanje modela.

Naslednja verzija modela je obsegala približno 1400 vrstic programa, ko je pred njeno dokončno zaokrožitvijo (t.j. zamrznitvijo specifikacij) in demonstracijo uporabniku prišlo do novih, resnejših sprememb v zasnovi bodočega skladišča. Končna varianta fizične konstrukcije objekta je namreč narekovala nekatere drugačne rešitve.

Če bo PROLOGov model bodočega sistema dobro razdelan in bo ohranil sedanjo, po vsem sodeč dobro strukturiranost, pričakujemo, da se bomo ob implementaciji sistema v izbranem programskem jeziku lahko pretežno omejili na prepisovanje algoritmov, učinkovito realizacijo baze podatkov, programiranje vhodno - izhodnih procedur, dodajanje podrobnosti in eventualno povečanje učinkovitosti sistema. Program v PROLOGu bi moral služiti tudi kot solidna osnova za dokumentiranje programskega paketa.

6. PREDNOSTI IN SLABOSTI PROLOGA ZA HITRO PROGRAMIRANJE SPECIFIKACIJ

Pri implementaciji modela bodočega skladiščnega sistema so se pokazale marsikatere dobre lastnosti PROLOGa [8].

V veliki meri smo se naslanjali na usodnosti, ki sledijo iz načelne zadostnosti deklarativnega opisa problemov. PROLOG namreč omogoča povsem deklarativno programiranje, ker ima že vgrajeno kontrolno strukturo. Razen možnosti, da omejimo sicer nedeterministično izvajanje programa, PROLOG ne potrebuje nobenih kontrolnih konstruktov (npr. iteracije). Tako se pri definiciji algoritmov lahko ukvarjamo le z vsebino problema, saj je za postopek reševanja že poskrbljeno.

Ker delovanje PROLOG programa temelji na unifikaciji, eksplicitni prireditveni konstrukti niso potrebni (razen pri vrednotenju aritmetičnih izrazov), kar omogoča hitrejše pisanje jasnih in jedrnatih programov.

K jedrnatosti programov prispeva tudi dejstvo, da deklaracije podatkovnih struktur niso potrebne [1]. Oblika zapisa termov že sama implicitno definira enega od splošnih podatkovnih tipov, za dinamično dodeljevanje pomnilnika med izvajanje programa pa skrbi kontrolna struktura. S funktorji lahko terme povezujemo med seboj v kompleksne podatkovne strukture. Ker ni potrebna vnaprejšna definicija tipov podatkov, lahko z uporabo funktorjev strukture poljubno poslahljamo, ne da bi nam bilo zato potrebno spreminjati višje nivoje programov in ne da bi bilo treba strukturo predvideti v podrobnosti vnaprej, kar je za hitro programiranje bistvenega pomena.

Jedrnatost PROLOG programov in fleksibilnost omenjanja podatkovnih struktur ponazarja prepis grafa na sliki 1 v program:

```
prevzem :-
    vnos_prevzemnice (Prevzemnica_0),
    generiranje_vskladiscnice (Prevzemnica_0, Vskladiscnica_0),
    vskladiscenje,
    potrditev (Vskladiscnica_0, Vskladiscnica_1),
    azuriranje_stanja_skladisca (Vskladiscnica_1),
    azuriranje_stanja_zalog (Vskladiscnica_1),
    generiranje_prevzemnice (Vskladiscnica_1, Prevzemnica_1).
```

V njem so v PROLOGovi minimalni sintaksi posamezne aktivnosti grafa proglašene za procedure, oznake podatkovnih množic grafa pa so prevzete kot formalni parametri teh procedur. Ti označujejo še nespecificirane podatkovne strukture, ki se seveda razgradijo na nižjih nivojih programa. Tako smo npr. proceduro, ki iz prevzemnice generira vskladiščnico izrazili kot rekurzivno relacijo med seznamom artiklov s količinami in seznamom artiklov s pripadajočim seznamom palet in količin na njih:

```
generiranje_vskladiscnice ([],[]).

generiranje_vskladiscnice ([Artikel:Kolicina|Prevzemnica],
                           [Artikel:Palete_Kolicine|Vskladiscnica]) :-
    stanje_skladisca (Prazne_palete_0),
    predlog_praznih_palet (Artikel, Kolicina, Palete_Kolicine,
                           Prazne_palete_0, Prazne_palete_1),
    brisi (stanje_skladisca (Prazne_palete_0)),
    vrisi (stanje_skladisca (Prazne_palete_1)),
    generiranje_vskladiscnice (Prevzemnica, Vskladiscnica).
```

Ker so v PROLOGu vse spremenljivke lokalne stavku, v katerem se nahajajo, je preglednost in berljivost posameznih programov ustrezno večja kot v konvencionalnih programskih jezikih. Stavki ene procedure so med seboj neodvisni, kar pomeni, da jih lahko dodajamo in odvzemamo, ne da bi morali spreminjati preostale. Seveda pa to ne velja, če imamo pri programiranju posameznih stavkov procedure že v mislih njihovo postopkovno interpretacijo. Ta način programiranja smo sicer pogosto uporabljali, saj si z njim prihranimo dodatno pisanje in povečamo preglednost programov. Zaradi preglednosti programov in lokalnosti spremenljivk je lažje razbiti sistem v neodvisne module in ga strukturirati.

Pravilnost programov v PROLOGu je razmeroma enostavno empirično preverjati. Ker ni nobene

formalne razlike med programi in podprogrami, lahko neposredno testiramo in preverjamo posamezne dele sistema. Pri odkrivanju napak si lahko pomagamo z že vgrajenim mehanizmom za sledenje izvajanja programa, pri katerem je možno selektivno izločanje nezanimivih delov. V RT-11 PROLOGu [2], katerega smo uporabljali, je to možno le do neke mere.

Omejene vhodno-izhodne možnosti PROLOGa nas niso ovirale pri realizaciji modela, nasprotno, v začetni fazi smo izkoristili možnost preprostega izpisovanja poljubno kompleksnih podatkovnih struktur. Podatkovne strukture smo namreč pretežno realizirali v obliki termov, ki so direktno izpisljivi.
Omenimo še nekatere lastnosti PROLOGa, ki so se pri našem delu izkazale kot neugodne.

Čisti PROLOG, kot rečeno, nima globalnih spremenljivk (kar smo mu zgoraj štteli v dobro), dovoljuje pa njihovo simulacijo z dodatnimi meta-logičnimi konstrukti. Zanesljiva uporaba teh konstruktov zahteva določeno mero natančnega postopkovnega razmišljanja, ki lahko postane delikatno zlasti pri realizaciji podatkovnih struktur v obliki množice dejstev (namesto enega "seznamskega" dejstva).

Zasledovanje izvajanja programa pri odkrivanju napak je včasih nekoliko težje kot sledenje izvajanja programa v drugih programskih jezikih zaradi njegovega nedeterminizma.

Pri realnih aplikacijah bi se kot PROLOGova hiba izkazalo dejstvo, da PROLOG ne podpira realnih števil. Implementacija, katero smo uporabljali (RT-11 PROLOG), pa ne podpira niti negativnih celih števil, kar je zahtevalo nekaj dodatnega dela pri implementaciji modela.

7. ZAKLJUČEK

Iz povedanega je razvidno, da so naše izkušnje pri uporabi PROLOGa kot specifikacijskega jezika zelo pozitivne. Programiranje v PROLOGu je hitro in zanimivo. PROLOGov model omogoča vodeno poslabljanje specifikacij, simulacijo obnašanja sistema načrtovalcu in uporabniku ter jasno zasnovo realne programske rešitve. Pri bodočih aplikacijah se bomo gotovo mnogo prej lotili programiranja modela v PROLOGu, kar bo omogočilo učinkovitejšo interakcijo z uporabnikom in hitrejšo definicijo problema.

8. LITERATURA

1. I. Bratko, M. Gams: Prolog: Osnove in principi strukturiranja podatkov; Informatica, letnik 4, št. 4, 1980

2. W. F. Clocksin, C. S. Mellish: Programming in Prolog; Springer-Verlag, 1982

3. R. E. Davis: Runnable specification as a Design Tool; Proceedings of the Logic Programming Workshop, Debrecen, Hungary, 1980

4. R. Kowalski: Logic as a Computer Language; Dept. of Computing, Imperial College, London, 1981

5. R. Kowalski: Logic for Problem Solving; AI Series, Elsevier North Holland, 1979

6. L. M. Pereira, F. Pereira, D. Warren: User's Guide for DEC-10 PROLOG; Dept. of Artificial Intelligence, University of Edinburgh, 1978

7. E. Santane-Toth, P. Szeredi: PROLOG applications in Hungary; Proceedings of the Logic Programming Workshop, Debrecen, Hungary, 1980

8. P. Szeredi, K. Balogh, E. Santane-Toth, Zs. Farkas: LDM - a Logic Based Software Development Method; Proceedings of the Logic Programming Workshop, Debrecen, Hungary, 1980

9. P. Tancig et al.: Nekaj izkušenj z ISAC metodologija razvoja informacijskih sistemov - I. del (analiza IS); Zbornik radova IX Jugoslovenskog savetovanja o informacionim sistemima, Beograd, 1980

10. W. M. Turski: Design of Large Programs; Notes for seminar Informatica, Ljubljana, Yugoslavia, 1981

11. G. Winterstein, M. Dausmann, G. Persch: Deriving Different Unification Algorithms from a Specification in Logic; Proceedings of the Logic Programming Workshop, Debrecen, Hungary, 1980