Competitio, I. évf. 1. szám, 2002. szeptember (105–113. oldal)

Tarnóczi Tibor*

A logikai programozás alkalmazhatósága a döntéstámogató rendszerekben

A globalizálódó gazdaságban az erősödő verseny egyre inkább előtérbe helyezi a gyors és minőségi vezetői döntéshozatal jelentőségét. A felgyorsult világban lecsökkent a reakcióidő, vagyis a döntési folyamatra kevesebb idő maradt, ugyanakkor a megoldandó problémák komplexitása, bonyolultsága és a döntéshez szükséges információ mennyisége megnőtt. Ilyen környezetben még inkább előtérbe kerül a számítógépes döntéstámogató rendszerek eddig sem jelentéktelen szerepe. A tanulmány bemutatja a logikai programozásban rejlő lehetőségeket, és azt, hogy annak alkalmazása milyen előnyökkel járhat a gazdasági modellezésben, a döntéstámogató rendszerek alkalmazásában.

A globalizálódó gazdaságban az erősödő verseny egyre inkább előtérbe helyezi a gyors és minőségi vezetői döntéshozatal jelentőségét. A felgyorsult világban lecsökkent a reakcióidő, vagyis a döntési folyamatra kevesebb idő maradt, ugyanakkor a megoldandó problémák komplexitása, bonyolultsága és a döntéshez szükséges információ mennyisége megnövekedett. Ilyen környezetben még inkább előtérbe kerül a számítógépes döntéstámogató rendszerek eddig sem jelentéktelen szerepe. Az elmúlt évtizedben a döntéstámogatási rendszerek definíciójában a Mintzberg-féle klasszikus döntésmeghatározást egyre inkább a tudásalapú szemlélet hatotta át, aminek következtében a döntéshozatalt tudásintenzív tevékenységnek tekintették (Holsapple–Whinston, 1996). Ilyen típusú döntéstámogató rendszerek kialakításában komoly szerepe volt a mesterségesintelligencia-kutatásnak, amelynek segítségével lehetővé vált olyan „aktív" döntéstámogató rendszerek kifejlesztése, amelyek a hagyományos algoritmikusfejlesztő eszközökön túlmutató új megoldásokat igényeltek.

Turban a könyvében *(1993)* úgy fogalmazott, hogy a vezetői döntéstámogató rendszerek „interaktív számítógépes rendszerek, amelyek segítik az adatokat és modelleket felhasználó döntéshozókat abban, hogy képesek legyenek strukturálatlan problémákat megoldani". *Holsapple és Winston (1996)* kicsit tovább lépve azt írja, hogy a feladat több és jobb döntés meghozatala adott megismerési, idő és gazdasági korlátok között, úgy, hogy növekedjen a döntéshozók eredményessége és a döntések hatékonysága. Az általuk megfogalmazott célhoz vezető utat a döntéstámogató rendszerek következő öt jellemzőjében határozták meg: (1) a döntéshozó kiegészítése; (2) magas szintű segítségnyújtás a döntéshozatal mindhárom fázisában; (3) a problémamegoldás könnyebbé tétele; (4) segítségnyújtás a strukturálatlan és félig strukturált döntések meghozatalában; (5) a tudás menedzselése.

Holsapple és Winston tehát olyan döntéstámogató rendszerben gondolkodik, amelynek „középpontjában" a tudás áll, és a tudásnak a tárolásához és feldolgozásához megfelelő számítógépes háttérre van szükség. Művükben meghatározzák azokat az alapvető „képességeket", amelyekkel a döntéstámogató rendszereknek rendelkezniük kell. A mesterséges intelligencia kutatásával foglalkozó műhelyek az elmúlt néhány év alatt jó néhány olyan új eredményt hoztak nyilvánosságra (vállalati ontológia, logikai programozás fejlesztése, korlátos programozás), amelyek jelentős mértékben megnövelték azt az eszközrendszert és módszertárat, amely ezen rendszerek hatékonyságát és problémamegoldó képességét továbbfokozni képesek.

* *Tarnóczi Tibor egyetemi adjunktus, a Debreceni Egyetem Közgazdaságtudományi Karának oktatási igazgatója.*

Mesterséges intelligencia alatt általában a számítógépek azon tulajdonságainak összességét értik, amelyek révén képesek olyan tevékenységek végzésére, amelyek intelligensnek látszanak. Ez azt a reményt ébreszti bennünk, hogy az intelligensebb számítógépek több segítséget nyújthatnak számunkra, és még jobban képesek lesznek az elvárásainknak megfelelő válaszok megadására. Ugyanakkor azt is tudjuk, hogy az intelligencia elég nehezen meghatározható fogalom, ebből következően a mesterséges intelligencia sem mondható jól definiált területnek. Vannak, akik magas fokon álló software engineeringet értenek alatta, valamiféle bonyolult szoftvertechnológiát, amely képes nagyon komplikált problémák egyszerű megoldására. Mások a problémamegoldás nem-numerikus módszereit értik mesterséges intelligencia alatt. A mesterséges intelligencia technikáit és alapelveit látszólag nehezebb megérteni, mint a számítógép-tudomány legtöbb területét, mivel a mesterséges intelligencia lehetőséget biztosít olyan komplex problémák megoldására, amelyeknél a hagyományos módszerek nem nyújtanak elég hatékony segítséget. A komplexitás annak is határokat szab, hogy valaki mennyire képes megérteni egy mesterséges intelligenciával rendelkező program pontos működését, mert ezek a programok egyfajta szimulációként foghatók fel, ahol a programozó beállítja a program viselkedésének feltételeit, de pontosan nem tudja, mi fog történni, amikor a program elindul. Ez teljesen új gondolkodási módot igényel a programozóktól, a fejlesztőktől.

Ha azt akarjuk, hogy a számítógép intelligensen „gondolkodjon", segítenünk kell neki, azaz meg kell adnunk számára mindazt a tudást, amivel rendelkezünk. Ez nagyon bonyolult feladat, mert sokszor a rendelkezésünkre álló ismeret alkalmazása túlságosan evidensnek tűnik számunkra, és nem egyszerű azt a számítógép számára úgy átadni, hogy ugyanúgy fel tudja használni, ahogyan mi tesszük.

A tudásnak nagyon sokféle típusa lehetséges. Itt most két típussal foglalkozunk: a tényekkel és a következtetési eljárással. A tények a világról alkotott megállapítások, a következtetési eljárások pedig a tények közötti következtetési lánc követését biztosító tevékenységek. A tények megfogalmazása sokkal egyszerűbb, mint a következtetési eljárásoké. A következtetési mechanizmus kifejlesztéséhez megfelelő nyelvezetre is szükség van, amely többek között lehet az elsőrendű predikátum logika.

Az elsőrendű logika ontológiai halmazában a legfontosabb, hogy a világot objektumok alkotják, amelyek a többi objektumtól megkülönböztető saját azonosítókkal és tulajdonságokkal rendelkeznek Az objektumok között különböző relációk létezhetnek. Az elsőrendű logika képes tényeket kifejezni az objektumokról, és lehetővé teszi különböző szabályok alkalmazását. Az elsőrendű logika abban az értelemben univerzális, hogy képes kifejezni bármit, ami programozható. Nem lehet ugyan teljes mértékben azt állítani, hogy a világ ténylegesen objektumokból és relációkból épül fel, de ez a fajta felosztás segítséget nyújthat a világról történő érveléshez *(Russel–Norvig 1995).*

A logikai programozás

Az előzőekben felvetett problémák megoldásában jelentős szerepe van a logikai programozásnak. A logikai programnyelvekben – a nevükből is következően – fontos szerepet játszanak a különböző rendű logikák. A logikai programozás az elsőrendű logika részhalmazán alapuló programozási paradigmaként is felfogható, és az előzőekből következően a logikai programozás alapgondolata az, hogy programjainkat a logika nyelvén, állítások (tények) formájában írjuk meg, és egy tételbizonyítási algoritmussal (szabályokkal) hajtjuk végre. Ebben a programnyelvben a program- és adatstruktúrák azonos szerkezetűek, ami azt is jelenti, hogy a program képes akár önmagát is módosítani *(Futó 1999).*

A logikai programozás különböző problémák gyakorlati megoldásában nyújthat segítséget. Ezek a gyakorlati problémák általában két nagy csoportba oszthatók *(Van Roy et al. 1999)*: (1) algoritmikus problémák (ilyenek lehetnek a szabályalapú szakértői rendszerek), és (2) keresési problémák (ilyenek lehetnek a különböző optimalizálási problémák – tervezés, ütemezés, természetes nyelvek elemzése, tételbizonyítás). A logikai programozás előnyei a két utóbbi típusnál használhatók ki igazán, de az első típus esetében is hatékonyabb megoldásokhoz vezethetnek.

Az előzőekből következően a logikai programozás szabályalapú formalizmussal rendelkezik, vagyis a program szabályok „gyűjteményét" tartalmazza, amelyeket keresési eljárások segítségével tudunk aktiválni. Ez a programozási elv a logikai programozás automatikus tételbizonyítási alapelvéhez kapcsolódik. Valójában minden szabály vagy keresés egy Horn-klauzula és a számítási eljárás a döntési módszer speciális formájaként fogható fel. Ezek a programok nagyon tömörek, többek között mert a felügyelet (a kontroll) implicite benne van a programban. Ezzel szemben az eljárás-alapú programok sokkal „barátságosabbnak" tűnnek azok számára, akik a hagyományos programnyelveket ismerik *(Apt – Smaus 2001)*.

A problémák megoldására a logikai programozás különböző típusait használhatjuk:

1. *Determinisztikus logikai programozás*: az algoritmus vezérlési folyamata teljes mértékben ismert és a programozó által meghatározott; nincsen szükség keresési eljárásokra; alapvetően szekvenciális algoritmikus problémákhoz használható; szintaktikailag a funkcionális programozáshoz hasonlít, de azt kiterjeszti, mert a logikai változók lehetnek "határozatlanok" is; lehetővé teszi inkomplett adatstruktúrákon alapuló hatékony programozási technikák alkalmazását.
2. *Nem-determinisztikus logikai programozás*: keresési eljárásokat használ a megoldás megkeresésére; a program minden részében jelen vannak a keresési eljárások; a program algoritmusa nem teljes mértékben ismert.
3. *Konkurens logikai programozás*: lehetővé teszi, hogy a programot részekre bontsuk, a különböző részek önállóan fussanak és szükség esetén együttműködjenek, a környezetükkel kommunikáljanak; klasszikus példája az aszinkron típusú termelő-fogyasztó modell.

A logikai programok egyik jellemzője, hogy működésük során viszonylag nagy a memóriaigényük. Különösen így van ez a nem-determinisztikus és a konkurens programozás, illetve ezek kombinációjának alkalmazása esetében. A számítógépek kapacitásának és sebességének rohamos fejlődése napjainkra lehetővé tette, hogy a memória ne jelentsen szűk kapacitást ezen fejlesztések számára. A számítástechnika és az informatika nagyarányú fejlődésének köszönhetően az elmúlt évtizedben a logikai programnyelvek is hatalmas fejlődésen mentek keresztül, és mára már az egyik legkomolyabb fejlesztőeszközként vehetők számításba. [1]

Mindez nem jelenti azt, hogy a hagyományos procedurális programnyelvek nem fejlődtek volna, és nem lennének alkalmasak ugyanezen feladatok megoldására. Az ezen nyelvekhez tartozó fejlesztőrendszerek is hatalmas fejlődésen mentek keresztül (pl.: C++, JAVA), és egyre alkalmasabbá váltak ilyen típusú feladatok megoldására, csak a fejlesztés időigényesebb, és sokkal nagyobb munkaráfordítással lehet egy ugyanolyan „tudással" bíró programot elkészíteni.

A logikai programozás fejlődésében az egyik komoly lépcsőfokot a Mozart programozási nyelv kifejlesztése jelentette, amely a kutatás több eredményét sikeresen ötvözi, egyben nagyon magas funkcionalitást és minőséget biztosít a UNIX és Windows platformokon futtatható rendszernek. A Mozart az Oz programozási nyelven alapszik, amely egységes egészként támogatja a deklaratív, objektumorientált, korlátos és konkurens programozást. Emellett lehetőség van hálózati és bizton-

[1] *Ennek látszólag ellentmond az, hogy a logikai programnyelvek alkalmazása ma még Magyarországon nem olyan széleskörű, mint az a világban tapasztalható, pedig a magyar fejlesztők élenjártak a Prolog programnyelv implementációinak kidolgozásában (Márkusz, 1988) és ma is megtalálhatók a legismertebb fejlesztői csoportok tagjai között (pl.: SICStus Prolog, ld. Swedish Institute of Computer Science, 2002). A széleskörű elterjedést gátolja a nyelv jelentős mértékű különbözősége a hagyományos procedurális programnyelvektől.*

sági igényeket is kielégítő programok írására is. [2] A rendszer kiváló lehetőséget biztosít a fejlesztők számára, komplex és nagyméretű feladatok gyors és nagyon hatékony megoldásához. A Mozart-Oz rendszer ideális osztott korlátos rendszerek fejlesztésére is. Összefoglalva azt mondhatjuk, hogy ez olyan magas szintű programozási nyelv, amely az intelligens, hálózatorientált, valósidejű, párhuzamos, interaktív és proaktív alkalmazások fejlesztését támogatja. Konkrétabban az alábbiakról van szó:

– absztrakt adattípusok, osztályok, objektumok és az öröklődés által kombinálja az objektumorientált programozás kiemelkedő lehetőségeit;

– nagyon jó lehetőséget biztosít a funkcionális programozás számára is (first-class procedures, lexical scoping, threads);

– kiváló lehetőséget biztosítanak a logikai és korlátos programozáshoz kapcsolódó logikai változók, diszjunktív szerkezetek és programozható keresési stratégiák.

– a Mozart-Oz konkurens nyelv, amely dinamikusan képes létrehozni egymással kommunikáló szekvenciális eljárások tetszőleges számú láncolatát, amelyek egyben adatáramlási láncot is képeznek;

– támogatja a halózaton keresztüli működést, ahol a különböző Oz oldalak egymáshoz tudnak kapcsolódni és megosztott változókon, objektumokon, osztályokon és eljárásokon keresztül képesek az együttműködésre, és képesek úgy viselkedni, mintha egyetlen alkalmazásként dolgoznának. Az oldalak automatikusan képesek szétkapcsolódni, amikor a működésüket befejezik;

– a megosztott környezetben képes megfelelő biztonságot is biztosítani *(Haridi – Franzén 2001)*.

Ez a fejlesztőrendszer nagy segítséget nyújtott számomra a logikai programozás mélyebb megismeréséhez, a korlátos programozás lehetőségeinek tanulmányozásához, és osztott rendszerek fejlesztési lehetőségeinek vizsgálatához *(Van Roy – Haridi 2001)*. Ez a rendszer mutatta meg számomra a korlátos programozásban rejlő lehetőséget, és ösztönzött további kutatómunkára ezen a területen.

A tanulmány további részében mégsem ennek a rendszernek az alkalmazási lehetőségeivel foglalkozom, mert a számomra igazán fontos korlátos programozás esetében igazán jól kidolgozott eljárások csak az egész értékekre vannak. [3]

A PROLOG programozási nyelv

A logikai programnyelvek egyik legelterjedtebb és napjainkban legszélesebb körben használt képviselője a PROLOG (PROgramming in LOGic) programozási nyelv, amely az elsőrendű logika, a predikátumkalkulus olyan megszorítása, amelyhez hatékony tételbizonyítási módszer adható. A nyelvet és az első interpretert Alain Colmerauer francia kutató fejlesztette ki 1972-ben, az elméleti háttér kifejlesztésében együttműködve Robert Kowalskival.

A Prolog gyökerei a logikában és az automatikus dedukcióban találhatók meg, amiről Colmerauer és Roussel a következőt írták *(Colmerauer – Roussel 1996)*: „Az nem kérdés, hogy a Prolog alapjában véve egy Robinson-féle tételbizonyítási program. A mi hozzájárulásunk az volt, hogy a tételbizonyítást programnyelvvé alakítottuk át." Ebből is következően a Prolog programozási nyelv a logikai programozási elmélet gyakorlati realizációjaként fogható fel, amely természetesnyelv-feldolgozó alkalmazásként indult és nem sokkal utána, mint általános célú programozási nyelv jelent meg. Eredetileg a Prolog Philippe Roussel – Colmerauer egyik munkatársa – által került implementálásra, mint interpreter *(Apt 2001)*.

---

[2] *A rendszer kifejlesztői: Universität des Saarlandes; Swedish Institute of Computer Science; Université Catholique de Louvain*

[3] *Finite Domain Constraint Programming [http://www.mozart-oz.org/documentation/fdt/index.html]*
Problem Solving with Finite Set Constraints [http://www.mozart-oz.org/documentation/fst/index.html]
Constraint Extension Tutorial [http://www.mozart-oz.org/documentation/cpitut/index.html]

A Prolog első implementációi interpreterként kerültek kifejlesztésre, ezért az terjedt el a Prologról, hogy lassú. Ezen a problémán segített David H. Warren, aki 1983-ban létrehozott egy „absztrakt gépet", amelyet ma Warren absztrakt gépnek (WAM) neveznek. Ez nemcsak a programnyelv gyorsaságát növelte, hanem megoldotta a platform függetlenségét is. A WAM a Prolog implementációk alapjává vált [4], de alkalmazásával találkozhatunk más interpreter típusú programnyelveknél is. Napjainkban elmondhatjuk, hogy a WAM-nak köszönhetően a logikai programnyelvek a végrehajtási sebességben is felveszik a versenyt más programnyelvekkel, különösen érvényes ez a hasonló rendszerű programnyelvekre (pl.: LISP).

A logikai programozásnak két további fontos sajátossága van: Az első az, hogy a „tiszta" formája a deklaratív programozást támogatja. A deklaratív programozás lehet procedurális (a módszerrel kapcsolatos) és deklaratív (a jelentéssel kapcsolatos). Egy deklaratív program procedurális interpretációja egy futtatható algoritmus leírásának (egy formulának) tekinthető. Véleményem szerint a deklaratív program könnyebben megérthető és könnyebben is fejleszthető.

A Prolog nyelvet nem könnyű elsajátítani, mivel nem tartalmaz olyan vezérlési eljárásokat, mint a hagyományos funkcionális programnyelvek, és a strukturált programozási koncepció alkalmazásához sem nyújt igazi segítséget. A programozók többsége az előre definiált változó típusokhoz van szokva, ugyanakkor a Prolog nyelvben a változók típusa csak futási időben, értékadáskor kerül meghatározásra. Ez nagy szabadságot ad a programozónak, de ugyanakkor sokkal nagyobb körültekintést is igényel. Jelentős különbség, hogy a változók nem a hagyományos módon kapnak értéket, hanem egyfajta illesztési technika alkalmazásán keresztül. Lényeges eltérést jelent az is, hogy a Prolog változói logikai változók, amelyek csak egyszer kaphatnak értéket (egyszer rendelhető hozzájuk a jellemzőjük), amit nem lehet megváltoztatni. Ebből és a logikai programozás egyéb tulajdonságaiból következően megváltozik az értékadás hagyományos felfogása is.

Lényeges eltérés az is, hogy a logikai programnyelvekben nem találhatók meg a hagyományos ciklusszervezési eljárások, azokat más formában, a logikai programozás szintaktikájához illeszkedően kell megoldani. Ahogyan az (1)-es példában látható, ugyanolyan néven több „eljárás" (szabály, tény) is lehet, és ezeket az „eljárásokat" a rendszer „logikai vagy"-gyal fűzi össze. Ezen túlmenően az ugyanazon néven szereplő eljárások argumentumainak a száma is különbözhet. Az eltérő argumentumszám lényegében már új „eljárást" jelent.

Sokáig lehetne sorolni azokat a jellemzőket, amelyek a logikai programnyelveket jellemzik, de úgy gondolom, hogy ez a néhány példa is mutatja, hogy egy egészen más típusú programnyelvről van szó. Ahhoz, hogy igazán megértsük a logikai programozás előnyeit, sokkal jobban el kell mélyedni a nyelv tanulmányozásában.

A prologban történő programozás szemléltetésére példaként nézzük tehát a következő egyszerű eljárást:

```
plusz(X, Y, Z) :- number(X), number(Y), Z is X + Y.        (1)
plusz(X, Y, Z) :- number(X), number(Z), Y is Z - X.
plusz(X, Y, Z) :- number(Y), number(Z), X is Z - Y.
```

Ezután bármely két számot megadva a program kiszámítja a harmadik számot. Pl.:

```
plusz(2, 3, Z).        Z = 5        (2)
plusz(X, 3, 5).        X = 2
plusz(2, Y, 5).        Y = 3
```

[4] *Létezik egy másik absztrakt gép is: a BAM - Berkeley Abstract Machine.*

A program az egyes utasításokat logikai állításként (tényként) kezeli, és egy eljáráson (szabályon) belül csak akkor hajtja végre a következőt, ha az előző 'igaz' volt. Ha egy eljáráson belül valamely állítás 'hamis', akkor a végrehajtást a következő ugyanolyan nevű eljáráson folytatja. Ezt a nyelvbe beépített következtetési mechanizmus biztosítja. Ha egyik eljárás sem vezet 'igaz' eredményre, akkor a program befejezi a működését és jelzi, hogy nincsen megoldás.

A Prolog fejlesztésekben komoly előrelépést jelentett az Ötödik generációs projekt meghirdetése Japánban (1982-1991).

A Prolog programozási nyelvvel 1992-ben ismerkedtem meg, abban az időben még kevésbé elterjedt programozási nyelv volt. Abban az időben a Turbo Prolog (Borland International Inc.) DOS operációs rendszer alatt futó változata állt rendelkezésemre (Rich–Robinson, 1988), amelyben egy nagyon egyszerű döntéstámogató rendszert készítettem. Ez az időszak a nyelvvel történő ismerkedést jelentette, és az is elmondható, hogy a programok sok vonatkozásban csak a procedurális nyelvről a logikai programnyelvre átültetett eljárásokból épültek fel. Vagyis messze nem voltam tisztában a logikai programozás igazi lehetőségeivel. A DOS operációs rendszer alatt rendelkezésre álló memória nem biztosította komolyabb Prolog program futtatását sem.

Ahhoz, hogy megfelelő szinten megismerjem a logikai programozás technikáját és lehetőségeit, jó néhány évre volt szükség, emellett nagyon nagy segítséget adott az LPA Prolog fejlesztőrendszer [5] 1997-ben történt megvásárlása és használata. Annak ellenére, hogy egy nagyon jól használható rendszer, nem tartalmaz eljárásokat a korlátos programozás alkalmazására.

A közelmúltban a logikai programozás területén sok energiát fektettek a különböző programnyelvi implementációk hatékonyságának, gyorsaságának és képességeinek a növelésébe, ezért napjainkban nagyon sok Prolog fejlesztőrendszer létezik. A számomra szükséges lehetőségeket a Ciao Prolog [6], a SICStus Prolog [7] és az ECLiPSe [8] tartalmazták. A SICStus Prolog és az ECLiPSe rendszerek sok hasonlóságot mutatnak, a korlátos programozás vonatkozásában talán a SICStus valamelyest több lehetőséget biztosít.

A korlátos [9] logikai programozás

A korlátos programozás alternatív megközelítése egy olyan programozásnak, amelyben a programozási folyamat bizonyos igények (korlátok) sorozatával korlátozott, és ezen igények általános vagy tárgykör (domain) specifikus módszerek által kerülnek kielégítésre. Az általános módszerek rendszerint a keresési intervallumot csökkentő és speciális keresési technikákhoz tartoznak. Ezzel ellentétben a tárgykörspecifikus módszerek rendszerint speciális cél-algoritmusok és „csomagok" formájában jelennek meg, amelyeket korlátmegoldó (constarint solver) rendszernek szoktak nevezni (ilyenek lehetnek: lineáris egyenletrendszereket megoldó programok; lineáris programozási programcsomagok; különböző egyesítő algoritmusok).

A korlátos logikai programozás (CLP) alapját általában tehát különböző keresési technikák alkalmazásai adják, amelyek egyesítik magukban a matematika, az operációkutatás és a mesterséges intelligencia különböző módszereit és előnyeit, ezáltal biztosítva gyors programfejlesztést és hatékony programvégrehajtást. Ezen technikák magas szintű integrációja lehetővé teszi a programozó számára, hogy a programot könnyen hozzá tudja igazítani a változó követelményekhez és

---

[5] *[http://www.lpa.co.uk/ind_top.htm]*

[6] *[http://www.clip.dia.fi.upm.es/Software/Ciao/] – szabadon használható rendszer*

[7] *[http://www.sics.se/sicstus/] – csak tesztelésre tudtam hozzáférni*

[8] *[http://www.icparc.ic.ac.uk/eclipse/] – a fejlesztő engedélyével a rendszert oktatási és kutatási célra használhatom*

[9] *{constraint} Követelmények megadására használt reprezentáció, mely véges számú változó megengedett értékegyütteseit adja meg. Egy korlát definiálható explicit módon, a megengedett értékegyüttesek felsorolásával, vagy implicit módon (algebrai vagy logikai kifejezésként), vagy egyéb relációként (Futó, 1999).*

a probléma legjobb megoldására koncentrálhasson. A gyors programfejlesztés és -módosítás által lehetővé válik, hogy a problémamegoldás legjobb útját választhassuk ki. A módszer segítségével nagyon összetett problémákat is kezelni tudunk, anélkül hogy azok menedzselhetetlenné válnának.

A korlátozást általános értelemben úgy foghatjuk fel, mint a lehetőségtér szűkítését, amely természetszerűleg megjelenik az emberi tevékenyég szinte minden területén, és ezek a korlátok különböző érdekes sajátosságokkal is rendelkeznek *(Van Hentenryck – Saraswat 1996)*: meghatározhatnak *rész információt* (nem szükséges, hogy a korlát egyedül határozza meg a benne szereplő változók értékét); *additívak* (a korlát megadásának sorrendje nem számít, csak az a lényeges, hogy a korlátok konjunkciója eredményes legyen); ritkán *függetlenek; nem irányítottak; deklaratívak* (azt határozzák meg, hogy milyen kapcsolatnak kell érvényesülnie és nem a kapcsolatot érvényre juttató számítási eljárást adják meg).

A korlátos programozás módszerét már számtalan problémacsoport esetében alkalmazták, úgymint interaktív grafikus rendszerek, operációkutatási problémák, molekuláris biológiai kutatás, üzleti alkalmazások, villamosmérnöki problémák megoldása, áramkörtervezés, numerikus számítási problémák megoldása, természetes nyelvek feldolgozása, számítógépes algebra.

A korlátos logikai programozás előnyét – a korábbi (1) és (2) példához kapcsolódva – lássuk egy nagyon egyszerű példán:

(CLP(Q)): plusz(X, Y, Z) :- Z = X + Y (3)

A (3)-as eljárás ugyanazt a feladatot oldja meg, mint az (1)-es, de egy kicsit egyszerűbben megfogalmazva. A korlátos programozás esetén a rendszer az értékkel nem rendelkező változóhoz rendel egy olyan értéket, amely esetében a feltétel teljesül. A (2)-es eljárások meghívásai ugyanazt az eredményt adják, ami ott is szerepel.

Lássuk most, hogy mit jelent egy lineáris egyenlőtlenségrendszer megoldása a SICStus programnyelvben. A SICStus racionális és valós számokra vonatkozó korlátos logikai programozási modulját használom fel:

```
2 * X +      Y £ 16                                    (4)
   X + 2 * Y
fldinst SYMBOL 163 \f„Symbol" \s 12 11
X + 3 * Y £ 15
30 * X + 50 * Y Æ maximum
```

A SICStus program:

```
clp(q) ?- { 2 * X + Y =< 16, X + 2 * Y =< 11, X + 3 * Y =< 15,      (5)
Z = 30 * X + 50 * Y}, maximize(Z).
```

Az eredmény:

```
X = 7                                                      (6)
Y = 2
Z = 30
```

Ilyen kis feladatnál a megoldási sebesség nehezen hasonlítható össze egy hagyományos lineáris programozási modellmegoldó program megoldási sebességével, de a szakirodalmi adatok azt bizonyítják, hogy a logikai programozással történő megoldás jelentős mértékben gyorsabb. Ennél

is lényegesebb lehet, hogy a különböző modulok kombináltan is felhasználhatók, így a numerikus változók mellett minőségi jellemzők és logikai feltételek is beépíthetők a modellbe, nem beszélve a logikai programozás egyéb előnyeiről.

A SICStus programnyelv a következő lehetőségekkel rendelkezik a korlátos programozás területén *(Swedish Institute of Computer Science, 2002)*: logikai (Boole-algebrai) korlátos megoldó modul (clp(B)); korlátos logikai programozás racionális és valós értékeken modul (clp(Q), clp(R)); korlátos logikai programozás véges egész értékeken modul (clp(FD)); korlátkezelő szabályok (CHR);

Az ECLiPSe lehetőségei a korlátos logikai programozás területén *(Imperial College–Parc Technologies, 2001)*: korlátos logikai programozás véges egész értékeken modul (fd); véges halmazokon végzett korlátos programozás (fd_sets); intevallumos korlátos programozás (range, ria); kombinált korlátos programozás (ic); felhasználó által definiált korlátok (propia); korlátkezelő szabályok (CHR); külső lineáris korlátokat megoldó modulok (eplex); lineáris korlátok megoldása Simplex módszerrel (clpqr); lineáris és véges egész értékek megoldását kombináló modul (fdplex).

Láthatjuk tehát, hogy mindkét program nagyon sokféle lehetőséget biztosít korlátos programozási feladatok megoldására.

Az ECLiPSe jóval több, mint egy korlátos logikai programozási rendszer, hiszen matematikai programozási és sztochasztikus programozási feladatok megoldását is támogatja. Az ECLiPSe komoly előnye pedig, hogy a felhasználó számára lehetővé teszi, hogy éppen azokat a módszereket kombinálja a probléma megoldása során, amelyek leginkább szükségesek, és legkönnyebbé teszik a feladat elvégzését. Úgy is felfogható, hogy az ECLiPSe egy komplex modellezési nyelv, amelynek segítségével egyszerűen és gyorsan tudunk bonyolult problémákat megoldani.

Hivatkozások

Apt, K.A. – Smaus, J-G. (2001): Rule-based versus Procedure-based View of Logic Programming. Joint NCC&IIS Bull. Comp. Science 16/2001:75-97. NCC Publisher

Apt, K.A. (2001): The Logic Programming Paradigm and Prolog, July [http://www.home.cs.utwente.nl/~etalle/lp/lp00.pdf.gz]

Colmerauer, A. – Roussel, P. (1996): The Birth of Prolog, History of Programming Languages. pp. 331-367, ACM Press/Adison Wesley

Futó, I. (1999, Szerk.): Mesterséges intelligencia, Aula Kiadó, Budapest

Haridi, S. – Franzén, N. (2001): Tutorial of Oz. [http://www.mozart-oz.org/documentation/tutorial/index.html]

Holsapple, C. W.–Whinston, A. B. (1996): Decision Support Systems; A Knowledge-based Approach. West Publishing Company

Imperial College – Parc Technologies LTD. (2001): ECLiPSe Constraint Library Manual, International Computers Limited and Imperial College. London, November [http://www.icparc.ic.ac.uk/eclipse/]

Márkusz, Zs. (1988): Prologban programozni könnyı. Novotrade, Budapest

Rich, K. M. – Robinson, P. R. (1988): Using Turbo Prolog. Borland Osborn/McGraw Hill

Russel, S. J. – Norvig, P. (1995): Mesterséges intelligencia modern megközelítésben. Panem-Prentice Hall, Budapest

Swedish Institute of Computer Science (2002): SICStus Prolog User's Manual. Intelligent Systems Laboratory [http://www.sics.se/sicstus]

Turban, E. (1993): Decision Support and Expert Systems: Management Support Systems. Macmillan Publishing Company

Van Hentenryck, P. – Saraswat, V. (1996): Constraint Programming. The Constraint

Programming Working Group. ACM-MIT SDCR Workshop
Van Roy, P. – Haridi, S. (2001): Concepts, Techniques, and Models of Computer Programming with Practical Applications in Distributed Computing and Intelligent Agent. Draft [http://www.mozart-oz.org]
Van Roy, P. – Brand, P. – Duchier, D. – Haridi, S. – Henz, M. – Schulte, C. (1999): Logic Programming in the Context of Multiparadigm Programming: the Oz Experience. International Conference on Logic Programming, Las Cruces, New Mexico, November
Wallace, M. – Novello, S. – Schimpf, J. (1997): ECLiPSe: A Platform for Constraint Logic Programming. William Penney Laboratory, Imperial College, Dublin, August