

Az rdbg nyomkövető (előadásvázlat)

Nepusz Tamás

2003. október 28.

Mi is az rdbg ?

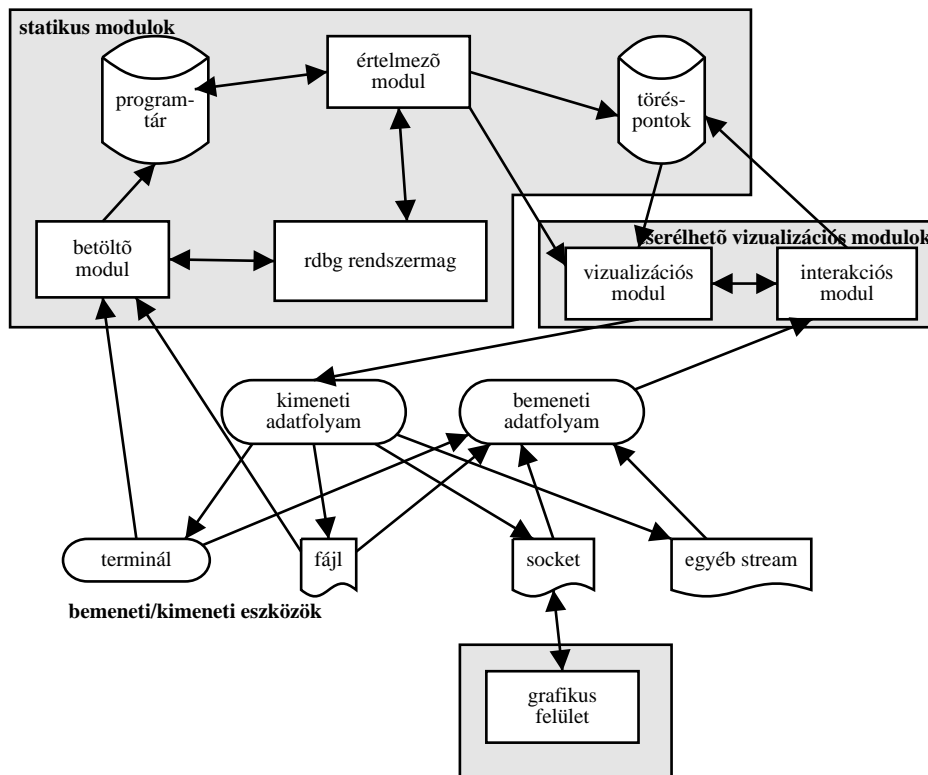
- Platformfüggetlen nyomkövető segédprogram SICStus Prolog-hoz
- Szöveges és grafikus felületet kínál Prolog nyelvű programok végrehajtásának követéséhez
- A szöveges felület magja gyakorlatilag egy Prolog nyelven íródott Prolog értelmező
- A grafikus felület egy erre a magra épülő Java alkalmazás, amely a nyomkövetési információkat fagráf formájában jeleníti meg.
- Alapját Szeredi Péter redukciós Prolog nyomkövetője képezi

Az rdbg szolgáltatásai

- A Deklaratív programozás tárgy tanítása során ismertetésre kerülő összes Prolog nyelvi konstrukció támogatása, az ezekre épülő programok futásának megjelenítése
- A redukciós lépések során bekövetkező egyesítések követése
- Vágók és a $\rightarrow / 2$ végrehajtásának részletes megjelenítése
- Könyvtári eljárások, `assert` és `retract` metapredikátumok kezelése
- Gyűjtőeljárások (`findall/3`, `bagof/3`, `setof/3`) futása során létrehozásra kerülő új keresési terek szemléltetése
- Töréspontok elhelyezése és kezelése

Az rdbg rendszer felépítése

1. **Statikus modulok:** a rendszer állandó magját képezik. Az *rdbg rendszermag* a felhasználó kéréseinek megfelelően vezérli az *értelmező*, illetve a *betöltő modul*. A betöltő modul az rdbg tudásbázisának bővítését végzi (új predikátumok felvétele fájlból vagy standard inputról), az értelmező modul pedig Prolog célsorozatok végrehajtását végzi a tudásbázis tartalma alapján. Az értelmezés során *nyomkövetési eseményeket* generál, és ezeket a vizualizációs modulnak adja tovább. Járulékos tároló szolgál a rendszerben beállított töréspontok adatainak tárolására.
2. **Cserélhető vizualizációs modulok:** az értelmező modul által generált nyomkövetési eseményeket a *vizualizációs modul* megjelenítendő karaktersorozatokká alakítja és elküldi a kimeneti adatfolyamra. Azokon a pontokon, ahol a felhasználó beavatkozására van szükség, az *interakciós modul* beolvassa a bemeneti adatfolyamról a felhasználó interakcióját jelképező karaktersorozatot, majd értelmezi azt. A vizualizációs és az interakciós modul lecserélhető, ezáltal lehetőség nyílik másfajta megjelenítési formák és interakciós lehetőségek implementálására (például: a nyomkövetés során keletkezett fák `TeXdraw`, `Postscript`, `XGML` (eXtended Graph Modeling Meta Language) formátumban történő exportálása. A grafikus felület is ezt a lehetőséget használja ki)



1. ábra. Az rdbg rendszer felépítése

3. **Adatfolyam interfészek:** az rdbg és a külvilág közti kapcsolatot biztosítják. Az adatfolyam interfészek valójában két Prolog stream alias-t takarnak (`rdbg_input` és `rdbg_output` néven), amelyek tetszőleges valódi stream-hez (így a standard be/kimenethez, fájlhoz, hálózati socket-hez) köthetők, így a kimenet átirányítása egyszerűen megoldható anélkül, hogy a rendszermaggal ezt külön tudatni kellene.
4. **Grafikus felület:** az a különálló Java alkalmazás, amely az rdbg kimenetének grafikus formában való megjelenítését végzi. Amint az ábrán látható, az rdbg magjával egy hálózati socket-en keresztül kommunikál, mégpedig egy saját belső protokoll szerint, így a grafikus felület használatához az alapértelmezett vizualizációs és interakciós modul helyett ezt a belső protokollt implementáló modulpárra van szükség, amely az rdbg disztribúció része, és megfelelő beállítások esetén az rdbg grafikus felülete automatikusan betölti

Klózok tárolási formátuma

- Minden klózhoz egyetlen `clause/7`, azonos funktorú klózok csoportjához egy-egy `indexing/3` struktúra
- A `clause/7` tárolja a klóz adatait, az `indexing/3` pedig az azonos funktorú klózokhoz tartozó indexelési információkat
- Példaprogram:

```
fib(1,F) :- !, F=1.
fib(2,F) :- !, F=1.
fib(N,F) :-
    N > 2, N1 is N-1, N2 is N-2,
    fib(N1, F1), fib(N2, F2), F is F1+F2.
```

Az első klóz az alábbi formátumban kerül tárolásra az `rdbg` belsejében:

```
clause(fib(1,_127), user, [!(Vágó),_127=1|CélFarok], CélFarok,
      [var('F',0,_127)], Vágó, 1).
```

Az egyes argumentumok jelentése:

- `fib(1,_127)` – az eredeti klózfej, amelyben a Prolog a beolvasáskor automatikusan belső változónevekre cseréli a meglévő változókat (ld. `read_term/2,3`).
- `user` – a modul, amelyben a klózt tároltuk (ily módon az `rdbg` is támogatja a programok moduláris felépítését)
- `[!(Vágó), _127=1|CélFarok]` – a klóz törzsében lévő konjunkciók, nyílt végű listává alakítva. Minden vágót egy `!/1` struktúrává alakítunk, amelynek egyetlen paramétere a `clause/7` hatodik argumentuma, ennek jelentőségét ld. a vágókezelésnél. A nyitott végű listának az a jelentősége, hogy amikor egy klózfejet a célsorozatban helyettesítünk a klóz törzsével, akkor az eredeti célsorozat farkát egyesítve a `CélFarok` változóval pont az új célsorozatot kapjuk, és így megúsztuk az `append/3` használatát.
- `CélFarok` – a nyílt végű klózlista farka külön elhelyezve, hogy ne kelljen végigmennünk a listán, ha egy redukciós lépés során a klózlista farkát akarjuk egyesíteni az eredeti célsorozat farkával
- `[var('F',0,_127)]` – ez a lista őrzi meg a célsorozatban lévő eredeti változónevek és a jelenlegi változók közti összerendelést, így az `rdbg` kimenetén nem a SICStus belső változónevei látszanak majd. Az első paraméter a változónév, a harmadik maga a változó, a második paraméter jelentőségét ld. később
- `Vágó` – a vágókhoz hozzárendelt változó külön elhelyezve, az egyszerűbb hozzáférés kedvéért. Jelentőségét ld. a vágókezelésnél
- `1` – a klóz sorszáma. A klózokat 1-től növekvő sorrendben sorszámozzuk, de esetleg `retract`-ok esetén előfordulhatnak „lyukak” a sorszámozásban, sőt az `asserta` használatkor negatív klózaazonosítók is bejöhhetnek.

Az indexelési információkat tároló `indexing/3` struktúra:

```
indexing(fib(_,_), user, t(1/0,[1,3],-1,
                        t(var,[1,2,3],-1,t(other,[3],0,t,t),t),
                        t(2/0,[2,3],0,t,t))).
```

Tároljuk az alábbiakat:

- Az a funktor, amelyre az indexelési információ vonatkozik (jelen esetben `fib/2`). Elvileg tárolhatnánk egy `'/2` alakú struktúráként is, de a SICStus első argumentumos indexelése szempontjából ez így előnyösebb.
- A modul, amelyben a megadott funktorú klózok találhatóak
- Az indexelési információ maga egy AVL-fával megvalósított asszociációs lista formájában. Az AVL-fát a SICStus `assoc` könyvtára hozza létre. A fenti lista pl. az alábbi hozzárendeléseket tartalmazza:

Kulcs	→	Érték
1/0	→	[1, 3]
2/0	→	[2, 3]
other	→	[3]
var	→	[1, 2, 3]

Az `rdbg` indexelése a SICStus-éhoz hasonlóan az első argumentum funktora alapján történik: ha a funktor `1/0` (tehát maga az `1` szám), akkor az első és harmadik klózok jöhetnek szóba, ha a funktor `2/0`, akkor a második és harmadik klózok. Ha a hívásban az első argumentum helyén változó (`var`) szerepel, akkor értelemszerűen az összes klózt végig kell próbálgatnunk, minden más esetben pedig csak a harmadik klóz jöhet szóba. A kiválasztott klózokat az `rdbg` ki is jelzi:

```
% leash, verbose
|| ?- fib(2, F).
G0:   fib(2,F) ?
|   Indexing selects clauses: [2,3]
|
|   Tried clause 2 of fib/2 ... success
...
```

Lekérdezések tárolási formátuma

- A lekérdezésben szereplő célok egy zárt végű listává alakulnak. A diszjunkciók megmaradnak ; / 2 struktúráként, és a bennük lévő konjunkciókat is csak akkor bontjuk ki listává, ha ténylegesen szükség lesz rájuk. A vágók helyett ! (0) jelenik meg, de a felhasználó ezt nem érzékeli.
- Példa: $p, !, q, (r, s; t) \rightarrow [p, !(0), q, (r, s; t)]$
- Szükség van egyéb járulékos információk letárolására is: *speciális vágókezelési primitív* (ld. vágókezelésnél) és *futási környezet*
- Az értelmező predikátumok a futási környezetet és az aktuális célsorozatot leíró listát passzolgatják egymásnak

A futási környezet tartalma

- A célsorozat végrehajtása során keletkezett változók listája, ahol minden listaelem egy ugyanolyan `var / 3` struktúra, mint amit a `clause / 7`-ben lehetett látni. Különbség: itt már van szerepe a középső paraméternek, jelentése: annak a célsorozatnak az invokációs száma, ahol a változó először megjelent a célsorozatban (ld. első előadáson a végrehajtási szintek azon a bizonyos három főlián). A `clause / 7`-ben ez alapértelmezésben azért nulla, mert így a lekérdezésben átadott (tehát a felhasználó által biztosított) változókhoz tartozó `var / 3` struktúrákat a `clause / 7`-ekből közvetlenül átrakhatjuk a futási környezetbe
- A jelenleg érvényben lévő választási pontok listája. A lista kétféle elemet tartalmazhat: vagy egy egyszerű invokációs számot (jelentése: az adott invokációs számú célsorozat első helyén lévő célnál választási pont maradt), vagy egy invokációs számot egy ! / 1 struktúra belsejében (jelentése: az adott invokációs számú célsorozatnál lévő választási pontot egy vágó megszüntette). A lista első eleme mindig a legutóbbi választási pont (tehát lista elejére pakolunk és onnan vesszünk ki)
- Az aktuális célsorozat első helyén lévő cél még nem illesztett, de illeszthető klózainak listája. Pl. ha az indexelés az első, harmadik és negyedik klózokat választotta ki, és az elsővel próbálkozunk éppen, akkor a lista: [3 , 4]. Néha ez a lista egyetlen x elemet is tartalmazhat, jelezve azt, hogy a célnál van még illeszthető klóz, de klózazonosítót nem tudunk rá értelmezni pl. azért, mert a cél beépített eljárás, vagy diszjunkció.
- A megnyitott keresési terek nyitási helyeinél lévő célsorozatok invokációs számainak listája. Keresési alterek megoldásgyűjtő eljárások hívásakor vagy a \+ / 1 predikátum használatakor jönnek létre, tehát ezen lista elemei mindig olyan célsorozatokat jelölnek, amelynek az első eleme megoldásgyűjtő eljárás vagy a \+ / 1.
- Az aktuális alapértelmezett modul (kezdetben `user`).

Az értelmezés folyamata (vágókezelés nélkül)

- Bekérjük a felhasználótól a célsorozatot, és átalakítjuk a belső listaformátumra. A diszjunkciókat nem bontjuk ki, a vágókat megjelöljük ! / 1 struktúrákkal.
- Ha a célsorozat üres, akkor végeztünk, egyébként vesszük a célsorozat első helyén álló célt.

- Ha az első cél gyűjtőeljárás vagy `\+/1`, akkor új keresési alteret nyitunk, ezt jelöljük a futási környezetben, és az újonnan nyitott alterben indítunk egy újabb értelmezést az eljárásnak átadott predikátumra vonatkozóan (közben gyűjtjük a megoldásokat, ha kell).
- Ha a cél beépített eljárás, könyvtári eljárás, vagy olyan felhasználói predikátum, amit az `rdbg`-n kívül konzultáltunk be (tehát összefoglalva olyan eljárás, amelyre vonatkozóan nincsenek `clause/7` és `indexing/3` struktúráink), akkor átadjuk az eljárást a SICStus Prolog-nak, és ha ez sikeres, akkor a célsorozat listájának fejét eldobva a maradékkal folytatjuk az értelmezést. Természetesen ilyenkor a végrehajtás részleteiről nem kapunk információkat, mert azt a SICStus végzi.
- Ha a cél olyan eljárás, amit az `rdbg` ismer, akkor az `indexing/3` alapján kiválasztjuk a megfelelő klózokat, és ha szükség van rá, választási pontot csinálunk. Sorra vesszük a klózokat, és mindegyik klózzal megpróbálunk egy redukciós lépést végrehajtani. Ha sikerült, akkor az új célsorozattal haladunk tovább, egyébként pedig visszalépünk, amíg van választási pontunk vagy maradék klózunk.
- Sikeres végrehajtás esetén a futási környezet változólistáján végighaladva megkapjuk a változóegyesítéseket. A felhasználó csak azokra a változókra kíváncsi, amelyek a 0 invokációs számnál (tehát az átadott lekérdezésnél) bukkantak fel először, ezért ezek értékét kiírjuk.

A vágókezeléshez szükséges módosítások

- Minden célsorozatban a vágókhoz tartozó szám azt az invokációs számot jelöli, ahol a vágó a célsorozatban megjelent. Úgy is fogalmazhatunk, hogy a vágó szülő klózának invokációs számát tároljuk, így ha a vágó végrehajtásra kerül, akkor a fában az eddig az invokációs számig terjedő összes választási pontot meg kell szüntetni.
- Ha egy vágó kerül a célsorozat első helyére, akkor meghívásra kerül a `need_to_cut_back_to/1` eljárás, ami a következőképpen működik: ha az átadott invokációs szám kisebb, mint az `rdbg:cut` speciális vágókezelő blackboard primitív értéke, vagy a blackboard primitív még nem létezik, akkor az invokációs számot beírja a primitívbe. Ily módon a primitív mindig azt a legkisebb invokációs számot tartalmazza, ameddig vissza kell vágni a végrehajtási fában.
- Minden választási pontnál az értelmező eljárások megnézik, hogy létezik-e az `rdbg:cut` blackboard primitív. Ha igen, akkor meghívásra kerül a `cut_handling/1` eljárás, melynek egyetlen argumentuma az aktuális invokációs szám, működése pedig a következő:
 - Lekérdezi az `rdbg:cut` primitív értékét, legyen ez `InvToCut`, az átadott paraméter pedig `Inv`
 - Ha `InvToCut < Inv`, akkor a választási pontnál vágni kell, ilyenkor az eljárás sikeresen lefut, ebből a hívó eljárás tudni fogja, hogy vágásra van szükség, és a többi választási pontot figyelmen kívül hagyja.
 - Ha `InvToCut = Inv`, akkor szintén vágni kell, de egyúttal töröljük az `rdbg:cut` értékét, hogy további vágások ne érvényesüljenek
 - Ha `InvToCut > Inv`, akkor már nem kell vágni, de az `rdbg:cut` értékét törölni kell.

Ha a `cut_handling/1` sikeresen lefut, akkor az őt hívó `interp_...` eljárásban egy olyan ágra kerül a vezérlés, ami az értelmezett eljárás többi ágát figyelmen kívül hagyja.

- A `->/2` operátor kezelése a vágóéhoz hasonló, csak nem az őt szülő klózig vág vissza, hanem addig a pontig, ahol a `->/2` operátor feltételrészre a célsorozat elejére került. Ez úgy érhető el, ha a `->/2` operátoros struktúrákat a célsorozatban változatlan formában hagyjuk (`[..., (Feltétel -> Cél), ...]`), és amikor egy ilyen `->/2` struktúra a célsorozat elejére kerül, megnézzük az invokációs számot (legyen ez `Inv`), és a célsorozatot átalakítjuk így: `[Feltétel, -(Inv), Cél, ...]`. A `-(Inv)` innentől kezdve ugyanúgy kezelhető, mintha `!(Inv)` lenne.

Nyomkövetési módok

- `leash` („póráz”) – ez a nyomkövető alapértelmezett üzemmódja. Minden célsorozatnál és megoldásnál megáll, és interakcióra vár a felhasználótól.
- `unleash` („póráz nélkül”) – a nyomkövető csak akkor áll meg, ha ráfut egy töréspontra vagy befejezi a célsorozat végrehajtását.
- `nodebug` (nincs nyomkövetés) – a nyomkövető csak a megoldást jelentő eseményeket jeleníti meg, és csak ezeknél az eseményeknél áll meg (tehát még töréspontnál sem). Természetesen a megoldás kiírásánál lehetőség van beavatkozni, például „visszarakni a póráz” a nyomkövetőre.

Részletességi szintek

- `verbose` („szószátyár”) – alapértelmezés, a nyomkövető minden eseményt kiír, amely a vizualizációs modulban definiálva van
- `quiet` („csendes”) – a nyomkövető csak a célsorozatokról és a megoldásokról informál bennünket, az egyéb részleteket (pl. indexelési események, vágás, megoldásgyűjtés kezdete stb.) elrejt
- `muzzle` („szájkosár”) – a nyomkövető nem jelenít meg semmit, kivéve, ha az esemény a beállított üzemmódnak megfelelően beavatkozást igényel (tehát pl. `leash` üzemmód esetén megjelennek a célsorozatok és a megoldások, mert ezeknél a felhasználónak lehetősége van beavatkozni, és jó, ha tudja, mire kell reagálnia).

A vizualizációs modul működése

- A nyomkövetés során az értelmező modul eseményeket generál, amelyeket a vizualizációs modulnak ad tovább. Egy esemény nem más, mint egy egyszerű Prolog struktúra, amely a vizualizációs modul `visualize/1` eljárásnak egyetlen paraméterében adódik át. A struktúra neve az esemény fajtájára utal, az első paramétere pedig mindig a futási környezet. A további paraméterek az esemény fajtájától függenek.
- A `visualize/1` eljárás teendői:
 1. Lekérdezi az aktuális nyomkövetési módot és részletességi szintet
 2. Megnézi, hogy van-e olyan töréspont, amely a bekövetkezett eseményre van beállítva, és ha igen, visszakapcsol `leash` módba (kivéve, ha előzőleg `nodebug` üzemmódban voltunk, mert ebből nem fogunk kijönni).
 3. Megállapítja, hogy az aktuális üzemmód és részletességi szint alapján meg kell-e jeleníteni a szóban forgó eseményt. Ha igen, kiszedi az `Env` paramétert az eseményből, és meghívja a vizualizációs kampóeljárást az esemény maradékával és külön az `Env` paraméterrel. Ennek az a szerepe, hogy így könnyebb hozzáférni az `Env` struktúrához a kampóeljárásán belül. Ha nincs szükség vizualizációra, akkor kilép.
 4. Miután a kampóeljárás visszatért, egy visszalépéssel hatályn kívül helyezi a kampóeljárásban esetlegesen végrehajtott egyesítéseket. Erre azért van szükség, mert így egy hibásan megírt kampóeljárás változóegyesítései nem borítják fel a program működését.
 5. Megállapítja, hogy az aktuális üzemmód és részletességi szint alapján az esemény igényel-e felhasználói beavatkozást. Ha igen, meghívja az állandó részben szereplő `interact/2` predikátumot az aktuális eseménnyel és futási környezettel.

Az `interact/2` eljárás teendői:

1. Meghívja az interakciós kampóeljárást az átadott két paraméterrel, valamint egy járulékos `Again` nevű változóval, amely kezdetben lekötetlen, és az interakciós kampóeljárásnak kell lekötnie az alábbi három érték valamelyikére:
 - `yes` – az interakció lekezelése után nem szabad továbbhaladni a nyomkövetéssel, hanem újabb interakciót kell bekérni

- `redisplay` – az interakció lekezelése után nem szabad továbbhaladni a nyomkövetéssel, hanem újból meg kell jeleníteni az eseménynek a vizualizációs modul által generált kimenetét (a vizualizációs kampóeljárás újbóli meghívásával), majd újabb interakciót kell bekérni
 - `no` – nincs szükség újabb interakcióra, a nyomkövetés haladhat tovább
2. Az `Again` változó értékétől függően el kell végezni az esemény ismételt megjelenítését (`Again=redisplay`) és az interakció újbóli bekérését (`Again=yes` vagy `Again=redisplay`).
- Saját vizualizációs modul létrehozásához az alapértelmezett vizualizációs és interakciós kampóeljárásokat kell átírni az általunk megírt eljárásokra. Ezt használjuk ki a grafikus változathoz szükséges saját protokoll implementálásakor.
 - Példa:
 - Az átadott esemény: `goal (Env, [member(X, [1,2,3]), (Y = 1; Y = 2), Z is X+Y])`
 - A szöveges változat vizualizációs modulja által adott válasz:
`G0: member(X,[1,2,3]), (Y=1; Y=2), Z is X+Y ?`
 - A grafikus változat vizualizációs modulja által adott válasz:
`goal 0 'member/2' yes member(X,[1,2,3]), (Y=1; Y=2), Z is X+Y`
`interact`

Eseményfajták

- `waiting (Env)` – a nyomkövető célsorozat begépelésére vár a felhasználatól. Ezen esemény felüldefiniálásával lehetőség van pl. parancssor előtti prompt-ok, járulékos információk kiírására. Mivel itt még nem igazán értelmezhető a futási környezet, ezért az `Env` paraméter tetszőleges lehet.
- `goal (Env, Goals)` – a végrehajtás során új célsorozat adódott, amely a `Goals` lista elemeinek konjunkciójából épül fel.
- `exit (Env, Goal, Inv)` – befejeződött az `Inv` invokációs számú célsorozat végrehajtása (akár sikeresen, akár sikertelenül), a befejezés a `Goal` cél végrehajtásakor történt meg. Szemléletesebben ez az esemény akkor következik be, amikor a végrehajtási fában véglegesen kilépünk az `Inv` invokációs számú csomóponttal mint gyökérrel képzett részfaból.
- `pred (Env, Goal, Unif, Type)` – végrehajtottuk a `Goal` célt, a végrehajtás eredményeképpen létrejött egyesítések listája `Unif`. Ha a végrehajtás sikertelen volt a fejlesztés megüszülése miatt, akkor `Unif` nem lista, hanem a `failure` atom. A cél típusa `Type`, ahol `Type` az alábbiak egyike:
 - `bip` – beépített predikátum (**b**uilt-**i**n **p**redicate)
 - `meta` – metapredikátum
 - `clause (ClauseId)` – `rdbg` által ismert predikátum, amelynek jelenleg a `ClauseId`. klóz értelmezése történt meg
 - `imported_from (Module)` – a `Module` könyvtárban vagy modulban szereplő külső predikátum, amit az aktuális modul importált magának
- `indexing (Env, ClauseList)` – az indexelés során kiválasztott klózok listája `ClauseList`.
- `success (Env, Vars)` – a végrehajtás az aktuális ágon sikeres, a megoldást tartalmazó illesztések listája `Vars`. Ha `Vars` üres, akkor a megoldás egyszerűen `yes`.
- `failure (Env)` – a végrehajtás az aktuális ágon sikertelen, visszalépés következik, ha még lehetséges
- `disjunction1 (Env, Branch1, Branch2)` – diszjunkció első ágának végrehajtása következik, a diszjunkció két ágán lévő klózok listája `Branch1` és `Branch2` (mint tudjuk, a többszörös diszjunkciók is kétszeres diszjunkciókra vezethetők vissza)

- `disjunction2(Env, Branch1, Branch2)` – diszjunkció második ágának végrehajtása következik, a diszjunkció két ágán lévő klózek listája `Branch1` és `Branch2`.
- `cut(Env, Inv)` – vágás történt visszafelé egészen az `Inv` invokációs számú célsorozatig.
- `assert(Env, Pred, ClauseId, Type)` – a `Pred` predikátumhoz új klózt vettünk hozzá `ClauseId` klózzazonosítóval. Az `assert` megvalósítása még nem teljesen SICStus-kompatibilis, ott ugyanis a programban meg kell adni előre a `dynamic/1` fordító direktíva használatával, hogy milyen nevű predikátumokat fogunk `assert`-álni, erre az `rdbg`-ben még nincs szükség.
- `retract(Env, Pred, ClauseId)` – a `Pred` predikátum `ClauseId` azonosítójú klózat egy `retract` vagy `retractall` hívás eltávolította.
- `not_provable_start(Env, Goal)` – új keresési alterben megkezdődött annak levezetése, hogy `Goal` nem bizonyítható (ld. `\+/1`)
- `not_provable_end(Env, Goal, Result)` – befejeződött annak levezetése, hogy `Goal` nem bizonyítható. `Result=failure`, ha a cél nem bizonyítható, vagy egy lehetséges megoldás egyesítéseinek listája, ha bizonyítható.
- `findall_start(Env, Template, Goal, Solutions)` – elkezdődött a `Goal` célsorozat összes megoldásának összegyűjtése, a megoldásokat a `Template` sablonra illesztjük (`Template` tehát a `findall/3`, `bagof/3`, `setof/3` első paramétere), és az összegyűjtött listát `Solutions`-ban fogjuk visszaadni.
- `findall_end(Env, Template, Goal, Solutions)` – befejeződött a `Goal` célsorozat összes megoldásának összegyűjtése. A paraméterek jelentése ugyanaz, mint a `findall_start/4` eseménynél.
- `findall_success(Env, Vars, Template)` – a megoldásgyűjtés során egy végrehajtási ág sikeresen lefutott. Funkcióját tekintve megegyezik a `success/2` eseménnyel, csak itt kiegészült a `Template` paraméterrel.
- `message(Severity, Msg)` – tetszőleges, eseményhez nem köthető üzenet kiírása. `Severity` a hibaüzenet fajtája:
 - `error` – hibaüzenet
 - `warning` – figyelmeztető üzenet
 - `informational` – információt közlő üzenet
 - `help` – felhasználó által kért segítség
- `exception(Ex)` – a végrehajtás során az `Ex` kivétel bekövetkezett
- `bp(Breakpoint, EventType, Result)` – a `Breakpoint` törésponttal kapcsolatos `EventType` parancs végrehajtásának eredménye `Result`. `EventType` lehet `add`, `enable`, `disable` és `remove` a töréspont hozzáadásakor, engedélyezésekor, letiltásakor és törlésekor. `Result` lehet `success` (sikeres végrehajtás), `failure` (sikertelen végrehajtás) vagy `already` (sikeres végrehajtás, de érdemi változás nem történt, mert pl. engedélyezéskor már előzőleg engedélyezve volt a töréspont). A `Breakpoint` változó egy `bp/4` struktúra, leírását ld. később

Töréspontok kezelése

- Egyelőre kétfajta töréspont létezik: *belső töréspont* és *egyszerű töréspont* (SICStus-ban *spypoint*).
- A belső töréspontokkal nem kell foglalkozni, az `rdbg` helyezi el átmenetileg olyankor, ha a felhasználó a nyomkövetés egyes részeit egy `skip` vagy `out` paranccsal át akarja ugrani.
- Az egyszerű töréspont a `spy/1` és `nosp/1` hívásokkal helyezhető el, kezelésük megegyezik a SICStus hasonló nevű hívásaival.

- Kis buherálással lehetőség van más jellegű töréspontok implementálására is, ugyanis az `rdbg` töréspontrendszerének „filozófiája” nem más, mint egy olyan utasítás a nyomkövetőnek, hogy adott nyomkövetési esemény(ek) bekövetkezésekor kapcsolja vissza a nyomkövetőt `leash` üzemmódba. A belső és az egyszerű töréspontok ezen kritérium speciális esetei. Lehetséges lenne például, hogy a Prolog vágókezelés működésének szemléltetésére implementáljunk egy olyan töréspontot, amely egy tetszőleges vágó működésekor aktiválódik: egyszerűen a `cut` eseményre vonatkozóan kell beállítani a töréspontot. Ez lényegesen egyszerűbbé teheti a Prolog nyelv oktatása során a vágó bemutatását (nem kell „kézzel” lépkedni a vágás bekövetkeztéig), és a helytelen vágóhasználatból eredő hibák kijavítását is.
- Töréspont felvételekor az `rdbg` belső töréspont-tárjába egy `bp / 4` struktúra assert-álódik. A struktúra elemei:
 - A töréspont egyedi azonosítója (*breakpoint ID*, `BID`). Ez egy pozitív egész szám, az újonnan létrehozott töréspontok mindig a legkisebb, nem foglalt pozitív egész számot kapják azonosítóként.
 - A töréspont típusa, azaz az `rdbg` (belső töréspont) és a `spy` (egyszerű töréspont) atomok egyike. Ezen a típusparaméteren keresztül lehet másfajta töréspontokat is felvenni, ha valaki megírja a hozzájuk tartozó predikátumokat.
 - A töréspontot leíró struktúra. Ez függ a töréspont típusától. `rdbg` (belső töréspont) esetén egy lista, amely megadja, hogy mely nyomkövetési eseményekre vonatkozik a töréspont. Az eseményeket *teljes* struktúraként kell tárolni, azaz az összes paraméterrel együtt. Ha valamelyik paraméter értéke a töréspont aktiválása szempontjából lényegtelen, akkor behelyettesítetlen változóval lehet kiváltani. Ha például a felhasználó a 4-es invokációs számmal jelzett célsorozaton egy `out` parancsot ad meg, akkor az `rdbg` felvesz egy, az `[exit(_, _, 4)]` eseménylistával fémjelzett belső töréspontot, amely persze eltávolításra kerül, ha az esemény bekövetkezik. Ha ugyanezen a célsorozaton egy `skip` parancs történik, akkor az `rdbg` veszi a célsorozat második elemét (legyen ez `Kövcél`), és eseménylistának a `[goal(_, [Kövcél|_]), exit(_, _, 4)]` listát választja. Ha megvalósítottam volna a fent leírt, vágásokra „érzékeny” töréspontfajtát, akkor az egy `[cut(_, _)]` eseménylistára hangolt belső töréspont is lehetne.
Ha a töréspont egyszerű, akkor ez a paraméter nem egy eseménylista, hanem egy `Modul : Cél` alakú struktúra, amely azt a predikátumot jelöli, amelynél a töréspontnak aktiválódnia kell. A töréspont vizsgálatakor ezt az `rdbg` automatikusan konvertálja egy `[goal(_, [Modul : Cél|_])]` listává, hogy a fentebb leírt töréspont-kezelési kritériumnak eleget tegyen.
 - A töréspont élettartama, amely megadja, hogy hány aktiválódás után kell törölni a töréspontot. Ez vagy egy pozitív szám (ilyen esetben a szám minden aktiválódás után eggyel csökken, és ha eléri a nullát, a töréspont törlődik), vagy az `inf` atom, amely a perzisztens (automatikusan nem törlődő) töréspontot jelenti. A belső töréspontok élettartama mindig 1, az egyszerű töréspontok perzisztensek, és csak felhasználói kérésre törlődnek.
- A töréspontok engedélyezhetőek és tilthatóak, ennek jelölésére assert-áljuk az `enabled_bp / 1` struktúrákat. Egyetlen paramétere a töréspontazonosító.
- Töréspont csak akkor aktiválódhat, amikor nyomkövetési esemény történt, ezért a vizualizációs modul `visualize / 1` predikátuma a `can_activate_bps / 3` hívással végignézi, hogy van-e aktiválható töréspont. Bemenő paraméter: az esemény, kimenő paraméterek: az aktiválható töréspontok és az aktiválható, engedélyezett töréspontok listája. Mellékhatásként csökkenti az engedélyezett, aktiválható töréspontok számlálóját, és a lejárt számlálóval rendelkezőket meg is szünteti. Ha van aktiválható, engedélyezett töréspont, akkor visszakapcsol `leash` üzemmódba.

Beépített és könyvtári eljárások kezelése

- Probléma: ezekről nem áll rendelkezésünkre klózlista és indexelési információ \implies nem tudjuk előre, hogy a végrehajtásukkor kell-e választási pontot csinálnunk, vagy sem
- Megoldás: `SICStus call_cleanup / 2` használatával:
`call_det(Goal, Det) :- call_cleanup(Goal, Det=true).`

- A `call_cleanup` végrehajtja az első argumentumban átadott célt, majd ha az már többféleképpen nem futhat le, végrehajtja a második argumentumban átadott célt is. Így ha `Det` kezdetben behelyettesítetlen, akkor az utolsó lefutáskor fog `true`-ra helyettesítődni, tehát tudni fogjuk, hogy mikor kell megszüntetni a választási pontot.

- Példák:

```
| ?- call_det(member(X, [1,2,3]), Det).
X = 1 ? ;
X = 2 ? ;
X = 3, Det = true ? ;
no
```

Bizonyos esetekben a `call_det/2` látszólag hibás eredményt ad:

```
| ?- call_det(member(2, [1,2,3]), Det).
true ? ;
no
```

Ennek magyarázata: a 2, mint megoldás megjelenésekor a `member/2`-nél még ténylegesen van választási pontunk (hiszen a listát nem néztük még végig), tehát `Det` helyesen nem helyettesítődik be. Amikor viszont a `member/2` a lista végére ér, és `Det` behelyettesítődik, azonnal bekövetkezik egy visszalépés is, ezért nem látjuk `Det`-et a kimeneten.

Gyűjtőeljárások és a `\+/1` kezelése

- A `findall/3`, `bagof/3`, `setof/3` és a `\+/1` hívások közös jellemzője, hogy a paraméterként megkapott predikátum megoldásait egy, az eredeti keresési tértől független térben keresik, és a program további futása ezen keresés eredményétől függ.
- `rdbg`-ben ez úgy jeleníthető meg, hogy az eredeti keresés során felrajzolódó gráfban a gyűjtőeljárás vagy a `\+/1` csomópontjához egy algráfot rendelünk, amely az argumentumban lévő predikátum megoldásainak keresését jeleníti meg
- Külön esemény jelzi az új keresési tér megnyitását és bezárását (gyűjtőeljárásoknál `findall_start`, `findall_end`, a `\+/1` esetén pedig `not_provable_start` és `not_provable_end`)
- A gyűjtőeljárások esetén egy megoldás megtalálásakor a megoldás mellett át kell adni a vizualizációs modulnak az eredeti sablont is (a `findall/3`, `bagof/3`, `setof/3` első paraméterét), hogy megjeleníthessük a felhasználónak a megoldások sablonra való illesztését (tehát azt az elemet, amit a megoldáslistában majd eltárolunk). Erre egy külön `findall_success` esemény szolgál. A `\+/1` végrehajtásához ilyen eseményre nincs szükség, megteszi egy egyszerű `success` is.