ＣＩＬプログラミング環境

CIL programming environment

天沼敏幸*,　　鈴木隆之*,　　奥西稔幸**,　　向井国昭**
Toshiyuki AMANUMA, Takayuki SUZUKI, Toshiyuki OKUNISHI, Kuniaki MUKAI
* 三菱電機東部コンピュータシステム株式会社
Mitsubishi Electric Computer systems (Tokyo) Corp.
** （財）新世代コンピュータ技術開発機構
Institute for New Generation Computer Technology

CIL is a logic based programming language to describe a natural language processing system influenced by situation semantics. The CIL programming environment has been implemented on PSI as the base language of LTB. LTB is a language tool box for Japanese processing system.

We report the overview of CIL programming environment focused on debug environment. CIL debug environment, which is called debug aide, is a screenful source image tracer on extended procedure box model.

The leash command of CIL debug aide is different from that of a standard Prolog system. The latter specifies a next step point of gate, but the former specifies a box relationship of next gate. This method is suitable for debugging on the Box model. These leash commands and a retry command supply a programmer-driven Divide-and-Query debugging.

１．はじめに

ＣＩＬ(Complex Indeterminates based Language)は状況意味論をモデルとした自然言語の意味記述を目的とする論理型プログラミング言語である。　汎用日本語処理系ＬＴＢ（Language Tool Box）の意味表現言語として、そのプログラミング環境を逐次型推論マシンＰＳＩ上に構築した。本稿ではＣＩＬの言語紹介と処理系の概要および特にそのデバッグ支援環境について述べる。

２．ＣＩＬの紹介

２．１　言語仕様概要

ＣＩＬは、Ｐｒｏｌｏｇのデータ構造に部分項を追加し、制御機構に遅延実行制御とそれを応用した制約機能を追加した言語である。言語仕様の面ではそれらの機能を扱う組み込み述語とそれらの機能を容易にプログラミングするためのシンタックス・シュガーを追加している。また、シンタックス・シュガーは汎用的なマクロ機能として、ユーザ定義が可能である。

２．２　処理系概要

ＣＩＬ処理系はプログラミング環境として、コマンド・インタプリタ，エディタ，インタプリタ，デバッグ支援，インスペクタ，コンパイラから成る。コマンド・インタプリタは処理系に対するコマンドの実行とゴール列の入力を行う。エディタは、シンタックス・チェックを行いながら編集を行うスクリーン・エディタである。インタプリタはソース・プログラム・レベルで簡易実行を行う。デバッグ支援は簡易実行時にシンタックス・シュガーを含めたソース・プログラム・イメージで追跡を行うプログラマ主動型のスクリーン・デバッガである。インスペクタは無限構造を持つ部分項を含むデータ構造や遅延実行制御によって生成される変数のデーモンを調べるためのユーティリティである。コンパイラは最適化フェーズを持ち、デバッグ済みのプログラムを高速に実行するためにＣＩＬプログラムをＰＳＩの記述言語であるＥＳＰに変換するトランスレータである。

3．デバッグ環境

3．1　概要

ＣＩＬプログラムを通常のプロセジャ・ボックス・モデルに基づいたライン・トレーサを用いてデバッグを行うとデバッグが不必要に困難なものとなる。その原因には、デーモンおよび条件付項の条件部，制約の実行がヘッド・ユニフィケーション中やボディの実行中に起動される事や、シンタックス・シュガーによるソース・プログラムとトレーサで表示されるイメージとのギャップが大きい事、リーシュ・コマンドがボックス・モデルに適合していない事が挙げられる。

快適なデバッグ環境を実現するために、デバッガとしてスクリーン・トレーサを採用した。また、ＣＩＬ独自の述語の充足過程をトレースするモデルとして、プロセジャ・ボックス・モデルを少し改良したモデルを用いた。そして、リーシュ・コマンドをボックス・モデルに対応した使い勝手の良いコマンドに改良した。

3．2　スクリーン・トレーサ

イメージ・ギャップを避けるために、実行中の具体化されたゴールの表示とそれに対応するソース・プログラム中のゴールをスクリーン上で反転表示を行う。さらにデバッグ中にスクリーン上で編集することも可能である。デバッグ支援に限らずＣＩＬ処理系の操作仕様はウインドウ／メニュとマウスによって行う。図-1にそのウインドウの例を示す。

3．3　ボックス・モデル

ボックス・モデルは詳細なトレースを可能とするために、ヘッド・ユニフィケーションを示すヘッド・ボックス，ボディ部のゴールの呼出しを示すボディ・ボックス，遅延実行制御によって起動されるデーモンの呼出しを示すデーモン・ボックス，シンタックス・シュガーによる展開形のゴールを示すエクスパンド・ボックスの四種類のボックスを持つモデルに拡張した。

```
CIL DEBUG AIDEDECAMP (send.cil.2)
Command
 <LEASH>
brother:CR
child:SPC
parent:p
warp:w
no_trace:n
 <CONTROL>
retry:r
fail:f
quit:q
 <SOURCE>
up:u
down:d
edit:e
edit_end
see_end
 <GOAL>
inspect:i
gate:g
spy:s
display
 <CIL>
see
state
(H1) 1 HCALL> send([exp/A]@discource_constraint,B) ? child
(H2) 3 HCALL> discourse_constraint([sit/A,sp/B,hr/C,dl/D,exp/
E]) ?
%
%  send
%
send([exp/X]@discource_constraint, Meaning):-
    sentence(X, [], Meaning), !.
send([exp/X]@number_constraint, Meaning):-
    different(X, Meaning).

discourse_constraint([sit/S, sp/I,hr/You,dl/Here,exp/Exp]):-
    member(sor(speaking, (I,Here), yes), S),
    member(sor(addressing, (You, Here) yes,S).

member(X,[X|_]):-!.
member(X, [_|Y]):-!, member(X,Y).
```

図-1　デバッグ支援ウインドウの例

各ボックスのゲートには、ボックスの最初の呼出しを表すＣＡＬＬゲート，成功を表すＥＸＩＴゲート，バックトラックによる再試行を表すＲＥＤＯゲート，失敗を表すＦＡＩＬゲートがある。さらに、デーモン・ボックスには、デーモンの生成を表すＤＧＥＮゲートとデーモンの消滅を表すＤＤＥＬゲートがある。ゲート名の先頭一文字はボックスの種類を示す。

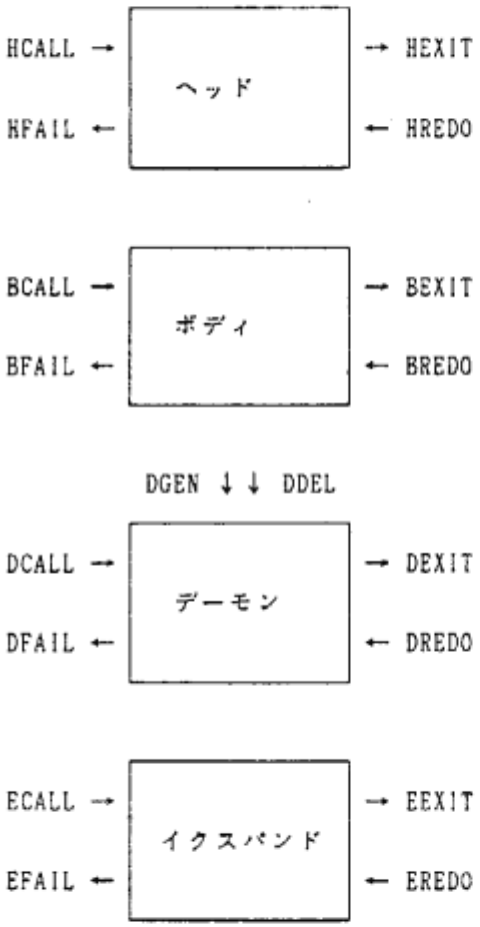

図-2 ボックスとゲート名

このモデルによるトレースは例えば次のようになる。

プログラム

```
a(X):- integer(X?), one(X).
one(1).
```

トレース

```
?-a(X).
… トップ・レベル・ゴール
(H1) 1 HCALL> a(X)
… ヘッド・ボックスの呼出し
(H1) 1 HEXIT> a(X)
… ヘッド・ボックスの成功
(B1) 1 BCALL> integer(X?)
… 遅延実行引数付ボディの呼出し
(E1) 1 ECALL> bind_hook(X,integer(X))
… イクスパンド・ボックスの呼出し
(D1) 1 DGEN > integer(X?), X
… デーモン・ボックスの生成
(E1) 1 EEXIT> bind_hook(X,integer(X))
… イクスパン・ドボックスの成功
(B1) 1 BEXIT> integer(X?)
… 遅延実行引き数付のボディの成功
(B2) 1 BCALL> one(X)
… ボディ・ボックスの呼出し
(H2) 2 HCALL> one(1)
… ヘッド・ボックスの呼出し
(D1) 3 DCALL> integer(1)
… デーモン・ボックスの呼出し
(D1) 3 DEXIT> integer(1)
… デーモン・ボックスの成功
(H2) 2 HEXIT> one(1)
… ヘッド・ボックスの成功
(B2) 1 BEXIT> one(1)
… ボディ・ボックスの成功
X => 1

yes
```

3．4　リーシュ・コマンド

リーシュ・コマンドはゲート間の関係指定方式(creep, skip)ではなく、ボックス間の関係指定方式(親，兄弟，子供)に改良した。以下の三つのコマンドがある。

(1) ｂｒｏｔｈｅｒコマンド

現在のレベルと同じレベルのリーシュ・ゲートを次のリーシュ・ポイントとする。その間は下位レベルのトレース・ゲートとリーシュ・ゲートは抑制される。同じレベルのゲートの実行の最後の場合は次の上位レベルのゲートがリーシュ・ポイントとなる。このコマンドはそのボックスが正しく動作するかどうかをその結果のみによって調べるときや次の兄弟ボックスに移るときに使う。

(2) ｃｈｉｌｄコマンド

現在のレベルより下位レベルのリーシュ・ゲートを次のリーシュ・ポイントとする。下位レベルのリーシュゲートが存在しない場合は同じレベルまたは上位レベルのゲートが次のリーシュ・ポイントとなる。このコマンドはあるボックスの結果がおかしい時にそのボックス内を更に調べていく時に用いる。

(3) ｐａｒｅｎｔコマンド

現在のレベルより上位レベルのリーシュ・ゲートを次のリーシュ・ポイントとする。あるボックスの結果がおかしい時にそのボックス内を更に調べていって問題のない所に入っていった時にいったん上位レベルに戻りまた別の下位レベルを調べる必要がある。そのときにこのコマンドを用いる。

4．デバッグ方法と操作性

ＣＩＬのデバッグ環境を実現するにあたっては、分割と確認を繰り返すことによってバグ追及を行う方式に適した操作性となることを考慮した。

ゴールの充足過程において、レベルに着目してボックス間の関係を見ると、ボックスの間には親、子、兄弟の関係がある。ある述語が正しくない場合として二つの現象がある。一つは失敗する場合であり、もう一つは成功したが具体化した変数の値が正しくない場合である。その原因を調べるためには、その述語を親とする一つ下のレベルの兄弟が正しいかをまず確かめる必要がある。正しく動作しない述語が判明すれば、つぎにその述語を親とする一つ下の兄弟を調べる。これを繰り返すことによって、正しく動作しない原因を突き止めることが出来る。

デバッグ支援のリーシュ・コマンドは上記の方式を用いてデバッグを行うことを前提として、Ｐｒｏｌｏｇとは異なる概念が用いられている。リーシュ・コマンドは次のリーシュのレベル番号が上か下か同じかを指定するコマンドとなっている。従来のリーシュ・コマンドでは、バックトラック時のskipが出来ないことや深くトレースしていった時に元に戻る時の操作が煩わしいという問題がある。ＣＩＬのリーシュ・コマンドは、それらの問題を解決している。さらに、操作上誤って調べたい述語と別の述語に入ってしまっても、調べたい述語に戻ることが容易である。

レベル番号に着目したリーシュ・コマンドはボックス・モデルとも一致し、EXITとFAILの両ゲートで再実行を指定するコントロール・コマンドの retry とこのリーシュ・コマンドを使えば Shapiro の Divide_and_Query 方式に近い操作が可能となる。このリーシュ・コマンドは他の逐次式論理型言語のデバッガにも採用できる。

参考文献

1) Ehud Y. Shapiro: Algorithmic Program Debugging, The MIT Press, 1983

2) 淵一博監修:自然言語の基礎理論，共立出版，1986

3) 向井国昭: 自然言語処理のための単一化の拡張と遅延的実行制御，ICOT-TR 215, 1986

4) 向井国昭，奥西稔幸，天沼敏幸: ＣＩＬ言語マニュアル，ICOT-TM 242, 1986

5) 杉村領一: 汎用日本語処理系ＬＴＢ，５Ｇシンポジゥム，1988