ICOT Technical Memorandom: TM-1205

TM-1205

並列論理型言語 KL1 のデバッグ環境

中尾　浩一 (応用技術(株))、和田　久美子
清原　良三、近山　隆

August, 1992

ICOT
Mita Kokusai Bldg. 21F
4 28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03)3456-3191 ~ 5
Telex ICOT J32964

Institute for New Generation Computer Technology

並列論理型言語 KL1 のデバッグ環境

中尾浩一[1] 和田久美子[2] 清原良三[2] 近山隆[2]
1: 応用技術 (株) 2:(財) 新世代コンピュータ技術開発機構

KL1 は，第五世代プロジェクトの核言語として開発された並列論理型プログラミング言語である．PIMOS は KL1 を効率良く実行するために設計されたオペレーティングシステムであり，その上で種々の応用プログラムの開発を進めている．並列プログラムのデバッギングは，逐次プログラムのデバッギングに比べて困難である．それは，(1) 複数の実行が交錯し合う．(2) 再現性のないバグが発生する．(3) デッドロックが発生する．などの理由による．したがって，並列言語のデバッグシステムを開発する場合には，これらの問題点を十分に考慮しなければならない．本稿では，並列プログラムのデバッギングに有効な機能を検討し，PIMOS が提供している KL1 のデバッグ環境を紹介する．

Debugging Facilities for KL1

Koichi Nakao
Applied Technology Co.,Ltd.
1-2-23 Minami-mori-machi, Kitaku, Osaka 530, Japan

Kumiko Wada Ryozo Kiyohara Takashi Chikayama
Institute for New Generation Computer Technology
1-4-28 Mita, Minato-ku, Tokyo 108, Japan

KL1 is a concurrent logic programming language, designed as a kernel language in the Japanese FGCS project. The operating system PIMOS is designed to run KL1 programs efficiently. Many application programs have been developed on it. In concurrent languages, debugging is more difficult than in sequential languages. The reasons are the following: (1) Execution of many processes mingles with one another. (2) Some bugs can not be replicated. (3) Deadlocks may occur, etc. When we develop a debugging system for a concurrent language, these problems have to be made into consideration. This paper discusses which debugging functions are effective for concurrent languages and describes debugging facilities of PIMOS.

1 はじめに

KL1 は並列論理型言語 FlatGHC[1] を基本として，オペレーティングシステムの記述には不可欠な，プログラムの実行制御を行なうためのメタ機能の付加など，種々の拡張を施したプログラミング言語である．

PIMOS[2][3] は KL1 を並列推論計算機上で効率良く実行・制御するためのオペレーティングシステムであり，PIMOS 自身も KL1 で記述されている．資源管理部を中核として，シェルやコンパイラ，デバッガといった基本的なプログラミングシステムを備えている．現在，PIM[4][5] 上で第三版が稼働中であり，種々の応用プログラムの開発を進めている．

本稿では PIMOS のデバッガの開発経験をもとに，並列プログラムのデバッギングに有効と思われる機能を紹介する．第 2 章では KL1 プログラムのデバッギングについて問題点を検討する．第 3 章では PIMOS のデバッグ支援機能を紹介する．最後に第 4 章でまとめと今後の課題について述べる．

2 KL1 プログラムのデバッギング

2.1 並列プログラムのデバッギング

並列プログラムのデバッギングは，逐次プログラムのデバッギングと比較するとかなり困難である．それは以下のような理由による．

- 複数の計算が同時に進行していること
- 複数の計算が互いに干渉し合うこと
- 再現性のないバグが発生すること
- デッドロックが発生すること

複数の計算の同時進行

逐次プログラムの実行では，プログラム中のある部分の計算が実行を終えるまでは他の部分の実行が開始されることはない．一方，並列プログラムの実行では複数の計算が同時に進行しているため，特定の部分の実行を観察しようとしても，不必要な部分の実行が交錯してしまいデバッグは困難となる．

トレース機能について考えてみると，不必要な実行はトレースをしなければよい．こうすることで，ユーザは本来のデバッグ対象に注意を集中してデバッグを行なうことができる．

複数の計算の干渉

一般に，同時に進行している複数の計算は，お互いにデータの依存関係を持っていることが普通である．このような実行をトレースする場合には，これらの計算の実行順序を制御できれば，干渉による影響を緩和できる．また，これら複数の計算の実行トレースが交錯しないように，トレース出力をよく整理し，わかりやすく表示することが大切である．

```
producer(Stream) :- true |
    Stream = [msg|NextStream],      (送信)
    producer(NextStream).
consumer([Msg|Stream]) :- true |    (受信)
    consumer(Stream).

?- producer(S), consumer(S).        (ゴール)
```

図 1: ストリーム通信の例

再現性のないバグ

並列システムではプログラムを実行する度に，計算の実行順序は違っているのが普通である．現在のところ画期的な解決策は見つかっていない．しかしながら，プログラムの静的解析機能を強化することでかなりのバグは検出できる [9]．クローズ間の入出力関係を解析すると，リダクションの失敗やコニフィケーションの失敗，非決定的な記述によるバグなどを検出することが可能である．

デッドロック

変数チェックなどの静的な解析を行なえば，ミスタイプなどの単純なバグは検出できる．しかし，このような静的解析では検出できない複雑な場合も多い．プログラム実行時に動的にデッドロックを検出する機能は必須である．

2.2 ストリーム通信

ストリーム通信は KL1 の基本的なプログラミング手法である．KL1 では，通信は共有変数により実現するが，プロセス間の継続的な通信を行なうために，共有変数をメッセージと新たな共有変数の組にしたデータ構造に具体化する．この通信形態をストリーム通信と呼ぶ．図 1 にストリーム通信の例を示す．

通常，KL1 プログラムの実行は，ストリーム通信によりプロセス間でメッセージのやり取りを行ないながら進行していく．各プロセスの振舞いは時間の推移とは無関係であり，プロセスの状態はやり取りしたメッセージにより移り変わっていく．したがって，プロセス間のストリームを流れるメッセージを観察することは，プログラムの実行状況を把握するための有効な手段となる．

2.3 負荷分散

マルチプロセサシステム上でプログラムを効率良く実行するためには，計算負荷を各々のプロセサにうまく分散させることが大切である．特に，PIMのような疎結合型のマルチプロセサシステムでは，プロセサ間の通信コストを無視できないため，計算負荷をうまく分散させ，なおかつ，プロセサ間通信のオーバヘッドを最小限に抑えることが，プログラムの性能向上のポイントとなる．効率の良い負荷分散アルゴリズムを見つけ出すことは，並列処理の最も重要な研究課題の一つである．このため，負荷分散など，プログラムの性能改善を支援するための機能は不可欠である．

3 PIMOSのデバッグ支援機能

PIMOSではKL1プログラムをデバッグするための機能として，トレーサやインスペクタ，性能評価支援ツールなど，一連のデバッグ機能を提供している．

3.1 トレーサ

プログラムのデバッグのために実行をトレースする機能で重要なことは，必要な部分以外のトレースを抑制し，本来のデバッグ対象に注意を集中できるようにすることである．

3.1.1 トレース方式

通常の逐次型言語では適当な時間間隔についてトレースを行なえば，特定のゴールの実行に限定したトレースを行なうことができる．一方，並列言語であるKL1の実行では複数の実行が同時に進行しているため，時間間隔を限ったトレースを行なっても不必要なゴールの実行までトレースしてしまい，特定のゴールの実行に限定したトレースを行なうことができない．

そこでPIMOSではゴールの親子関係だけに注目したトレース方式を実現している．トレースは実行木の中でトレースが必要なゴールの枝に対してのみ適用する．不必要な部分のトレースは抑制されるので，ユーザは本来のデバッグ対象に注意を集中してデバッグを行なうことができる．トレースの対象は「リダクション」である．トレースの実行は一つのリダクションの終了毎に，その結果生じたサブゴールを報告する形で進められる．

3.1.2 KL1のトレース機能

トレーサの実現にはKL1のメタレベル機能である荘園の機能を利用している．

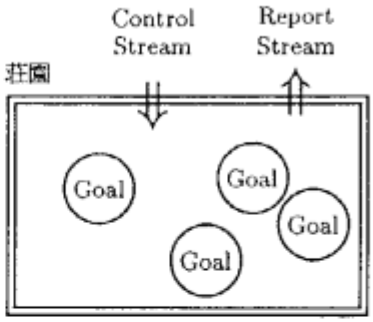

図2: 荘園の概念図

荘園の例外処理機構

荘園はKL1プログラムの実行や資源の管理を行なうための基本機能である．荘園はKL1ゴールの集まりであり，コントロールストリームとレポートストリームを介して，その内部のゴールの実行を監視/制御できる実行管理単位である．コントロールストリームには実行の中断，再開，放棄，資源状態の問い合わせなどのメッセージを送ることができる．レポートストリームからは，デッドロックやリダクションの失敗などの例外事象の発生，資源の不足，実行の終了などが報告される．図2に荘園の概念図を示す．

荘園内の実行中に例外が発生すると，次のようなメッセージが報告される．

```
exception(Type, GoalInfo, NewGoal)
```

Type: 例外名．例外の種類を表す情報である．

GoalInfo: 例外を起こしたゴールの情報．ゴール名と例外発生時の引数の情報などである．組込述語の例外では，それを呼び出している述語の情報も報告される．

NewGoal: 例外処理後に実行を継続するための未定義変数．例外を起こしたゴールの代わりに実行する代替ゴールを指定する．シェルやデバッガなど例外処理を行なうメタプログラムが具体化する．

トレース機構

図3のように，トレース実行中の荘園にはトレースが指定されているゴールと抑制されているゴールが混在している．トレースが指定されているゴールのリダクションでは，そのサブゴール (GoalInfo) はトレース例外としてトレーサに報告される．この時，どのゴールのリダクションであるのかを識別するために，トレース指定時にトレーサが与えた識別子 (Id) も一緒に報告される．なお，トレースの実行は apply_tracing(Goal,Id) というメタ述語で行なう．

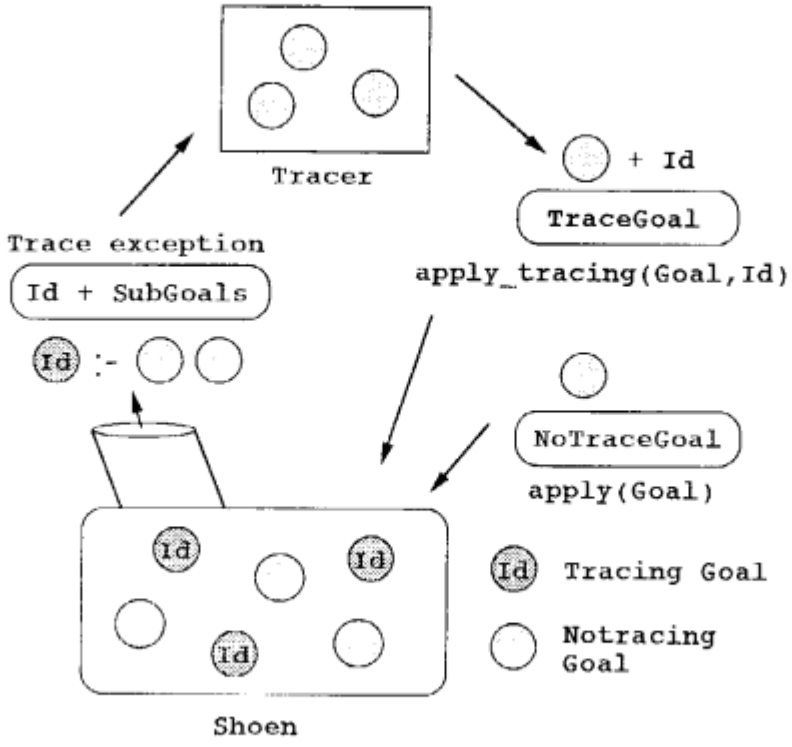

図 3: トレースのメカニズム

また例外復帰用の変数 (NewGoal) にこれを指定することでトレース実行を継続できる.

3.1.3 スパイ

スパイは, あらかじめプログラムが動作を停止する位置 (ブレークポイント) を設定し, デバッグを行なう操作に相当する. スパイではブレークポイントとして述語を指定する. 以後これをスパイ対象と呼ぶことにする.

スパイ機能も荘園の例外処理機構により実現している. スパイではスパイ対象ゴールが生成された時に例外が報告される. スパイの実行は apply_spying(Goal,Id,Target) というメタ述語で行なう. 実行木中の任意の部分木 (Goal) 毎に異なるスパイ対象を設定できる点が特徴である. スパイ対象を指定するための引数 Target には述語名やアリティなどの指定にワイルドカードを使用できる.

スパイを利用するとデバッグ対象にたどり着くまでの不必要なトレースの手間を省くことができる. スパイ機能はデバッグ時間の短縮とプログラマの集中力の維持に貢献している. 図4 にスパイの実行イメージを示す. トレーサでは二つのタイプのスパイを実現している.

フォークスパイ: スパイ対象ゴールの生成を報告する. すなわち, スパイ対象をそのボディ部で呼び出しているゴール (親ゴール) のリダクションを報告する.

リダクションスパイ: スパイ対象ゴールのリダクションを報告する. フォークスパイ実行後, 検出されたスパイ対象ゴールを更にトレースすることで実現している.

3.1.4 ゴールの実行制御とトレース出力の整理

通常, トレース対象は複数であることが多い. このような場合, これらの実行をそのままトレースしたのでは, 複数の実行の様子が交錯して報告されるためデバッグは困難になる. この問題を緩和させる機能として, トレーサでは次のような機能を提供している.

ゴールの実行保留

トレースで報告されたサブゴールの実行を一時的に保留する機能である. 実行が保留されているゴールはトレーサのコマンドによりその一覧を見ることができ, 任意の時点で実行を開始することができる. この機能を利用することで動的にゴールの実行順序を制御することができる. ゴール間のデータの依存関係に注目して, 選択的, 段階的にゴールの実行を進めていくことができる. 例えば Producer-Consumer モデルでは, 先に Producer を実行し, 後から Con-

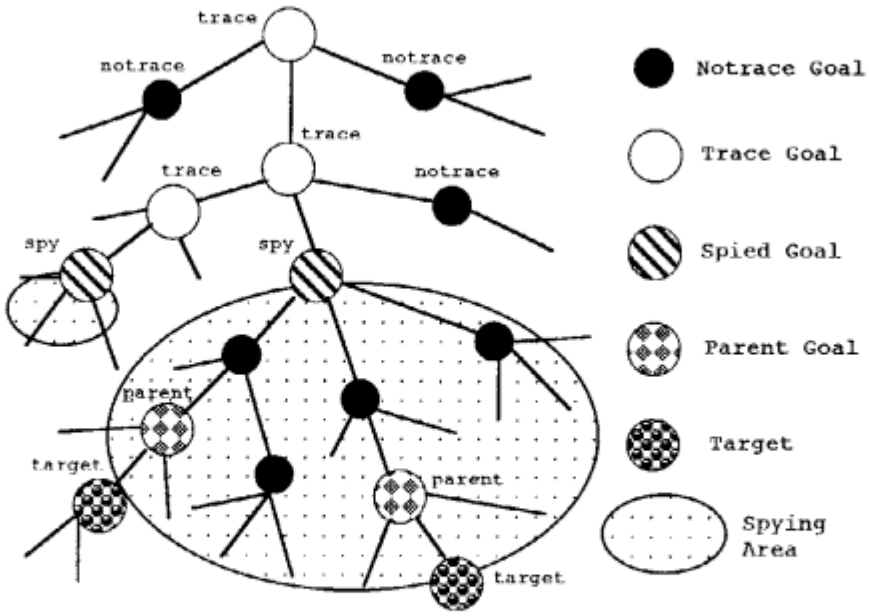

図 4: スパイの実行イメージ

sumer を動かしてみるというような使い方ができる．主に，データ駆動型のプログラムをデバッグする時に利用する．

マルチウインドウを利用したトレース

任意のプロセス毎に専用のウインドウを割り付けてトレースを行なう機能である．個々のプロセスが並列に実行されると，それらに対応したトレーサへのトレース出力も並列に行なわれる．トレース出力のマージも自由である．また，個々のトレーサへのコマンドの入力順序を変えるだけで，簡単に実行の順番を制御することができる．主に，要求駆動型のプログラムをデバッグする時に利用する．

3.2 デッドロック検出

PIMOS では変数チェッカによる静的なデッドロック検出機能と，ガーベジコレクション (GC) 時に動的に検出する機能を提供している．

3.2.1 変数チェッカ

クローズ単位でそれぞれの変数の出現回数をカウントし，出現が一度きりの変数を検出する静的解析ツールである．プログラミング時の変数名の綴り間違いにより，デッドロックが発生することは多いが，そのようなバグは変数チェッカにより簡単に検出できる．PIMOS では単体の変数チェッカのほか，使い勝手の向上のためコンパイラのプリプロセサとしても提供している．

3.2.2 GC による動的検出

デッドロックの動的検出 [8] は GC 実行時 (実行中のインクリメンタルな GC も含む) に行なわれる．一般に，一つのゴールがデッドロックに陥ると，そのゴールが生成すべきデータに依存するゴール群もデッドロックに陥る．このためデッドロックゴールは複数存在するのが普通である．デバッグ時にはデッドロックの連鎖の中で，根本原因となっているゴールの情報が必要である．動的検出機能ではデッドロックの因果関係の中で原因となっているゴールだけを報告する．

3.3 ストリーム通信に関するデバッグ機能

3.3.1 変数モニタによるストリームの監視

変数モニタは KL1 データを監視し，その具体化を待って報告するツールである．主に，プログラムの進行に伴う未定義変数の具体化状況を知りたい場合に利用する．変数モニタをトレーサから起動して，トレース中のゴール間のストリームを流れるメッセージを監視することができる．ストリーム通信の監視では，連続してメッセージの流れを観察する場合が多いので，変数モニタは監視中のデータがリストに具体化した場合，その Cdr 部に対して引き続き監視を継続する．

3.3.2 ストリーム通信の試行

PIMOS のデバッガは，変数プールと呼ぶ任意の KL1 データを保持しておくためのプールを用意して

いる．変数プールは，ゴールの引数に用いた変数やトレース出力中に現れた変数を保存して，それらのデータを後の実行に利用するといった使い方をする．

この変数プールを利用することで，ストリーム通信を会話的に実行することができる．本体となるプロセスをデバッガのバックグラウンドで起動し (例えばウインドウプロセス)，そのプロセスへのストリームを変数プールへ格納しておくと，メッセージを順次送出していきながら，プロセスの振舞いを観察するという操作を行なえる．

3.4 性能評価支援機能

並列プログラムでは，プログラムの性能向上のために，負荷分散アルゴリズムを設計することは非常に重要である．PIMOS では KL1 プログラムの性能評価を支援する機能として，パフォーマンスメータや，プログラムの実行状況をグラフィック表示するためのチューニングツールを提供している．

3.4.1 Runtime Monitor

プログラム実行時に各プロセサの稼働状況を表示するパフォーマンスメータである．一定時間間隔毎の各プロセサの稼働率をカラーもしくは白黒の濃淡グラフで表示する．図5 に動作例を示す．縦軸はプロセサ，横軸は時間を表す．過去からの稼働状況の遷移をひと目で把握できる点が特徴である．GC の発生 (図中の△) も確認できる．なお，Runtime Monitor のオーバヘッド (Runtime Monitor 実行時のプロセサの稼働率) は 5 %以下である．このほか，パフォーマンスメータとして，全プロセサの平均稼働率を棒グラフで表示するものも提供している．

3.4.2 ParaGraph

ParaGraph[6][7] は，デバッガ上で計測した実行ログ情報をもとに，KL1 プログラムの実行過程をグラフィック表示するプロファイリングツールである．プロファイル情報は，何が (What)，いつ (When)，どのプロセサで (Where)，どれくらい処理されたかに着目して表示される．逐次プログラムの実行では，Where は一定であり，When については実行順序が決まっているためそれ程重要ではないが，並列プログラムの実行では，これら三つの次元の情報は不可欠である．

計測は一定時間間隔毎に行なわれる．When は各時間間隔に 1 から順に通し番号を割り付けたものであり (サイクル)，Where はプロセサを示す．What は計測項目を示し，次のものがある．

プロセサの稼働状況: 計算時間，メッセージの送受信処理時間，GC 時間，アイドル時間が各サイクルに占める割合．

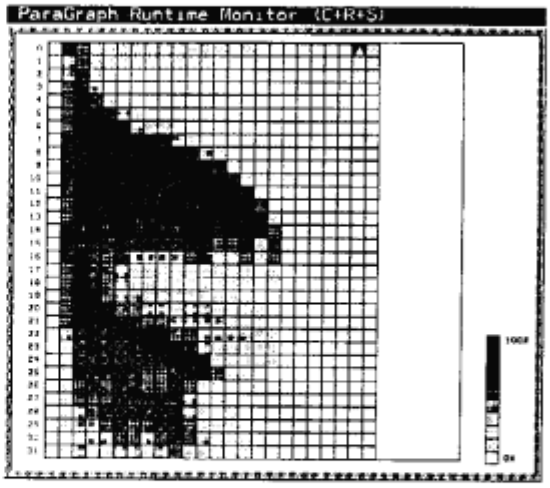

図 5: Runtime Monitor の動作例

プロセサ間通信の発生状況: メッセージ毎の送受信回数．

プログラムの実行状況: 述語毎のリダクション数とサスペンション数．

無限ループの検出

ParaGraph は本来，デバッグのためのツールではなく，プログラムの性能向上を支援するためのチューニングツールであるが，暴走するプログラムのバグの検出にも利用できる．

前述の通り，ParaGraph にはサイクル毎に実行された述語のリダクション数を計測する機能がある．暴走するプログラムを計測し，暴走が疑われる状態になったところで実行を放棄すると，その時点までの実行情報が計測される．この情報をグラフィック出力すると暴走部分のリダクション数が突出するので，暴走していた述語がおかり暴走の原因を推測することができる．図6 に ParaGraph を用いて暴走した述語を検出した例を示す．縦軸がリダクション数，横軸は時間である．リダクション数の時間推移から暴走している述語がひと目でわかる．

3.5 その他のデバッグ支援機能

3.5.1 インスペクタ

任意の KL1 データの内容を調べるための調査システムである．会話的にデータ構造を調査して表示する機能を持っている．インスペクタはデバッガのトップレベルやトレーサの中から起動し，複雑な構造を持つデータやトレース中のゴールの引数の中身などを調べる時に利用する．

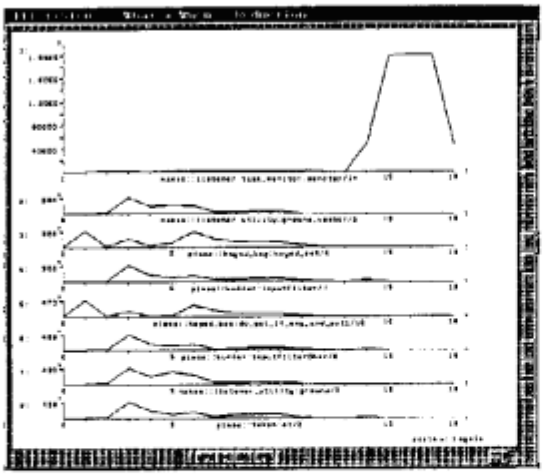

図 6: ParaGraph で暴走を検出した例

3.5.2 クロスリファレンス

モジュール間の参照関係を解析し，指定されたモジュールや述語を参照している (呼び出している) モジュールを表示する．

3.5.3 ユーザ定義マクロ

デバッグ機能ではないが，マクロ機能を利用することはバグの予防につながる．マクロはプログラムの記述性 / 可読性向上のために不可欠な機能である．PIMOS では算術演算やユニフィケーション，定数表記のためのシステムマクロを用意しているが，ユーザが独自のマクロ定義を行なうこともできる．

デバッガでは，ユーザが定義したプログラムのコンパイル時に用いるユーザ定義マクロを，ゴール (項) 入力時に利用するための機能を提供している．

3.5.4 コンパイラとの連係によるデバッグ情報の獲得

デバッグモードによりコンパイルを行なうことで，トレース実行時にデバッグ情報を獲得できる．現在は，ゴールが選択したクローズの番号 (ソースプログラム中の記述で，上のクローズから順番に割り付けた番号) を提供しているが，ソースプログラム上の様々な情報を獲得する拡張も検討している．

4 おわりに

KL1 プログラムのデバッグ機能の概要を述べた．最後に，これまでデバッグ機能の開発に携わってきて感じていることをまとめる．

実用になるものを作るには処理系の協力が不可欠

PIMOS のトレーサは言語処理系が提供する非圧の機能を利用して実現しているが，このほかトレーサを実現する方法としては，ソフトウエアでメタインタプリタを作成する方法も考えられる．これは，処理系に負担がかからないという利点があるが，実行効率を考えると本格的なプログラムのデバッグは困難である．

例えば，処理系にスパイ機能がまだ搭載されていなかった頃 (トレース機構は実装されていた)，トレース例外を利用してスパイを開発したことがある．これはこれで役に立ったのだが，とても実用規模のプログラムをデバッグできるものではなかった (処理系サポートのスパイではオーバヘッドは 50 %程度である)．実用に耐え得るシステムを開発するには処理系の協力は不可欠である．

暗黙の同期機構のありがたさ

KL1 でプログラムを作成すると自然に並列プログラムが書けてしまう．「自然に」というのは，特にプログラマが実行の同期に注意を払わなくとも，KL1 では暗黙の内に同期がとれてしまうという意味である．

このため，KL1 のような同期機構を持たない並列言語では一番問題になり易い，同期によるバグにはほとんど苦労をしていない．また，PIMOS を種々の PIM 上に移植したが，その時もほとんどバグは出なかった．暗黙の同期機構はプログラム開発上の大きな利点である．

KL1 は高機能

実行制御，例外処理，トレースにスパイ，プロファイル機能などデバッガの基本的な機能はすべて KL1 の機能としてユーザに提供されている．ユーザインタフェイスに関わる部分は，ユーザが自由に作成できるので，色々と応用の幅は広い．現在，トレース機構を利用してプログラムの可視化ツールやアルゴリズミックデバッガなども試作されている．様々な方面の研究に役立てて頂きたい．

今後の課題

プログラムの静的解析ツールの強化: 現在，プログラムの静的解析ツールとしては変数チェッカを提供しているが，これだけでは不十分である．第 2.1章で述べたような本格的な静的解析ツールは必要である．

デバッガのユーザカスタマイズ機能の充実: デバッガが管理している色々な資源，例えば，変数プールやスパイの環境などを，ユーザがプログラムレベルから利用できるように改良したい．

また，現在デバッガが行なうトレースなどの出力は，デバッガが自身で定めたフォーマットにより出力している．これをユーザが自由にカスタマイズできるように，出力フォーマットをユーザが指定するためのポートレート機能を導入したい．

有効機能の調査: 本稿で述べたようなPIMOSのデバッグ機能は，どの程度ユーザのデバッギングに役立っているか，また，ユーザはそのほかにどのような機能を必要としているか，調査をする必要がある．その結果を踏まえた上で機能の整理を進め，再度仕様を検討してみることは大切である．

参考文献

[1] K. Ueda. Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. Technical Report TR-208, ICOT, 1986.

[2] T. Chikayama, et al. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp.230-251, ICOT, 1988.

[3] T. Chikayama. Operating System PIMOS and Kernel Language KL1. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pp.73-88, ICOT, 1992.

[4] A. Goto, et al. Overview of the Parallel Inference Machine Architecture (PIM). In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp.208-229, ICOT, 1988.

[5] K. Taki. Parallel Inference Machine PIM. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pp.50-72, ICOT, 1992.

[6] 相川聖一他「負荷分散支援ツール ParaGraph」情報処理学会研究報告 91-PRG-3 pp.95-102, 1991年7月

[7] S. Aikawa, et al. ParaGraph: A Graphical Tuning Tool for Multiprocessor Systems. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pp.286-293, ICOT, 1992.

[8] Y. Inamura and S. Onishi. A Detection Algorithm of Perpetual Suspension in KL1. In *Proceedings of 7th International Conference on Logic Programming*, 1990.

[9] K. Ueda and M. Morita. A New Implementation Technique for Flat GHC. In *Proceedings of 7th International Conference on Logic Programming*, 1990.