プログラミング－言語・基礎・実践－ 18－17
（1994. 7. 22）

並列オブジェクト指向言語A-NETLデバッギングシステムの構築

古谷 泰重　吉永 努　馬場 敬信

宇都宮大学工学部

我々は，並列オブジェクト指向トータルアーキテクチャA-NETの研究の一つとして，並列オブジェクト指向言語A-NETLにより記述された並列プログラムをデバッグするシステム“A-NETLデバッガ”を開発中である．並列プログラムのデバッグ方式は，従来型のブレークポイント方式に加え，イベントベース方式，静的解析方式などが提案されている．我々は，これらの方式を検討した結果，ブレークポイント方式とイベントベース方式を採用することに決定した．本稿では，これらの方式をA-NETLデバッガに適用するための設計方針と実現方法について述べる．

Program Debugging System for the Parallel Object-Oriented Language A-NETL

Yasushige Furuya　Tsutomu Yoshinaga　Takanobu Baba

Department of Information Science, Utsunomiya University
Utsunomiya-shi 321 Japan

As a part of the A-NET project we have been developping a program debugging system, called “A-NETL Debugger”, for a parallel object-orented language A-NETL. Concerning several techniques for debugging parallel programs we have adopted a breakpointing and an event-based view. This paper describes the design principles and the implementation.

1 はじめに

我々のプロジェクトでは，並列オブジェクト指向を核概念として，計算機，プログラミング言語，応用の3つの立場から統合的に検討したトータルアーキテクチャA-NET[1, 2] の開発を行っている．この研究の一環として，プログラム記述言語である並列オブジェクト指向言語 A-NETL を定義した．さらに，A-NETL によるプログラム開発支援システムの1つとしてデバッギングシステム “A-NETL デバッガ [3]” を開発中である．

並列プログラムのデバッギングには従来の逐次プログラムに対する技術では解決できない様々な問題があり，近年盛んに研究が行われている [4, 5]．これらの研究では，逐次デバッガの拡張方式，イベントベース方式，静的解析方式などが提案されている．また，デバッグ情報の表示法としては，従来型のテキスト形式に加え2次元ダイアグラム形式やアニメーション表示形式などのアプローチがある．

我々はこれらのアプローチを検討した結果，オブジェクト内部の状態を効率よく解析するために従来型のブレークポイント方式，オブジェクト間通信による影響を解析するためにイベントベース方式を採用することに決定した．また，増大するデバッグ情報を効率よく視覚化するための表示法として，テキスト形式．2次元ダイアグラム形式，アニメーション表示形式を使い分ける．

A-NETL デバッガは，ホスト計算機上にマスターデバッガ，各ノード上にローカルデバッガを配置することにより，マスター/ローカル間でメッセージをやり取りし，詳細なものから大局的なデバッグまでを可能する．各ノード上のローカルデバッガは，ファームウェアレベルと OS レベルで機能分担して構成される．

本稿では，A-NETL 言語の特徴と A-NETL デバッガの設計方針を述べる．さらに設計方針に沿った実現方法について述べる．

2 並列プログラムデバッギングに対するアプローチ

2.1 並列オブジェクト指向言語 A-NETL

並列オブジェクト指向 A-NETL[6] は，アクタ理論に基づき A-NET 並列計算機上での実行を前提とした並列処理記述言語である．

A-NETL ではオブジェクトが並列処理の単位であり，1つのプログラムは複数のオブジェクトにより構成される．各オブジェクトは，同期/非同期的なメッセージパッシングにより互いに協調しながら並列処理を行う．オブジェクトには，静的/動的の2種類が存在し，静的オブジェクトには同種のオブジェクトを複数定義するインデックス付きオブジェクトがある．オブジェクトの宣言部には，状態変数宣言部とメッソド宣言部とがある．

メッセージ送信には，past（非同期）型・now（同期）型・future（非同期+返答メッセージ待ち）型の3タイプが存在する．メッセージは受信側オブジェクトにおいて到着順に実行される．

2.2 並列プログラムにおけるデバッグ問題

効率的な並列プログラムを開発するには，問題に潜む並列的な現象をそのまま制御フローとして自然に記述できる必要がある．並列プログラムは，並列に動作するプロセス間の同期や通信を考慮したプログラムスタイルとなる．並列に動作するオブジェクト間の同期や通信に伴うデバッグ時の諸問題としては以下のものが挙げられる．

(1) 振舞いの非決定性

非決定性を含むプログラムとは，実行のたびにプログラムの挙動が変化する可能性を持つプログラムのことである．これは，メッセージの到着順の相違による．

また，デバッガ自体が観測のために行う操作が非決定性を生み出す（これをプローブ効果と呼ぶ）場合もある．

(2) 大域的な時間概念の欠如

A-NET 計算機のノード間では，同一のクロックで制御を行うことは想定しておらず，A-NETL プログラミングモデルにおいても実時間を用いたオブジェクトの振舞いを記述する機能は持っていない．また，オブジェクト間の実行順序関係も送信側と受信側の順序関係のみが明確であり，通信を行っていないオブジェクト間の関係を明確に規定することは不可能である．よって，実時間を基準にしたプログラム全体の振舞いの表現が困難となる．

(3) デバッグ情報の増大

逐次プログラムにおいてデバッグの対象となるのは，単一プログラムの内部状態に関するもののみであった．しかし，並列プログラムでは複数オブジェクトの内部状態をデバッグする必要があり，さらにメッセージ通信形のデバッグも行わなければならない．このため，デバッガの収集する情報量が増大し，従来型のデバッグ情報の表示法では対応ができなくなる．

2.3 並列プログラムのデバッギング法

現在までに各研究機関において，研究・開発されている並列プログラムのデバッガの手法は次のように分類される

(1) 従来型手法（ブレークポイント方式）

並列動作する各プロセスに逐次デバッガを割り当て，逐次デバッガの集合を1つの並列デバッガとみなすものである．この手法では，一般の逐次デバッガと同種のブレークポイント機能をサポートしている．

従来型手法は，並列プログラムの各プロセスにおける手続きレベルや命令レベルの解析では非常に有効である．しかし，競合状態の観察などのプロセス間の相互作用から生じる事象を解析することができない．また，プローブ効果の問題は，ほとんど解決されていない．

(2) イベントベース方式

プログラム実行中に特定の振舞いを“イベント履歴”と呼ばれるログとして記録し，そのログをもとにプログラムの振舞いを解析する方式である．この方式では，プロセス間通信レベルでの振舞いを記録することにより，従来型手法では困難であったプロセス間相互作用から生じる事象を解析できる．

イベントとして定義する事象は，計算機や言語の性質により異なる．DISDEB [7] システムではメモリアクセスがイベントであり，Radar[8] システムではメッセージ通信がイベントである．

イベントベース方式におけるプローブ効果の影響は，従来型手法に比べると改善されているものの，完全に排除することは不可能である．また，解析可能な振舞いはイベント記録時のものであるため，非決定性に関しては何度かテストを行いその都度対処しなければならない．

(3) 静的解析方式

静的解析方式は，従来のイメージとは異なり，プログラムを実行せずにプログラムの構造的エラーを検出するものである．この方式は，PIMOS[9] システムやHyperDEBU[10] システムにおける変数解析機能に用いられている．

静的解析方式では，データフロー解析を並列プログラムに適用し，並列プログラムの構造を解析するものである．この方式は，プログラムを実行することなしに解析を行うため，プローブ効果を完全に排除できる．また，同期エラー解析機能により競合状態が発生する可能性のある箇所をあらかじめ予測することができるため，非決定性の問題も回避できる．

しかし，起こり得る振舞いすべてについて解析を行うため，解析に必要な計算量がプログラム規模や動的性質により指数的に増大してしまう．

3 A-NETL デバッガの設計

3.1 基本設計

前節に述べた並列プログラムのデバッグに関するアプローチを検討し，効率がよくユーザフレンドリなデバッグを実現するため，A-NETL デバッガでは以下のような基本設計を採ることとした．

(1) 機能分散

A-NETL デバッガは，実現すべき機能を明確に区分することにより，それぞれの区分ごとにモジュールを構成した分散型の構造を採る．機能区分は，ユーザインターフェイスと各ノードの統括，OS レベルの処理，ファームウェアレベルの処理と階層化される．

(2) デバッグ方式

オブジェクト内部レベルの状態を観察するための手法としてブレークポイント方式を採用する．この方式では，プローブ効果を避けることは不可能であるが，オブジェクトごとの論理的な誤り検出や各種変数の観察には有効な手段である．

また，オブジェクト間通信レベルの振舞いを解析するための手法として，イベントベース方式を採用する．この方式により，非決定的な振舞いの1パターンの実行過程を決定的に解析する．イベント履歴記録処理はプローブ効果となるが，記録する事象と情報を限定し記録処理を軽くすることにより，最小限

の影響にとどめる．記録されたイベント履歴は，ブラウジングとリプレイに使用される．

図1に，これらの方式を用いたデバッグ処理の流れを示す．

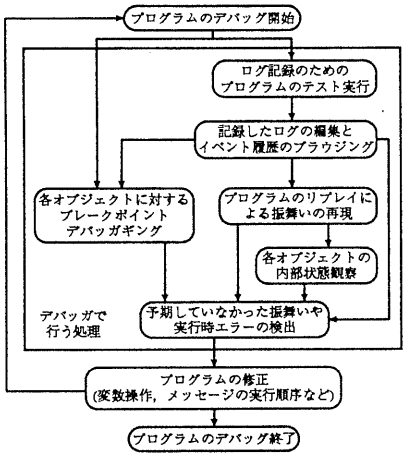

図1: A-NETLデバッガを用いたデバッグ処理の流れ

(3) デバッグ情報の視覚化

高並列システムにおけるデバッグ処理では，必要とされる情報量が非常に多く，大量の情報の中から意味のある情報を選択して表示する技術が必要となる．この問題に対して，A-NETLデバッガでは図2に示すようにプログラムの振舞いに関する情報をマルチウィンドウを用いて階層化する．

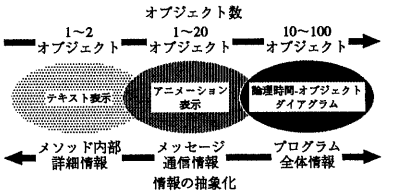

図2: 3形式の表示法を用いたデバッグ情報提供

3.2 A-NETLデバッガの構造

A-NETLデバッガは，図3に示すようにA-NET計算機のインターフェイスとなるホスト計算機上のマスターデバッガと，A-NET計算機の各ノード上に配置されるローカルデバッガからなる分散デバッガ構造を採用する．

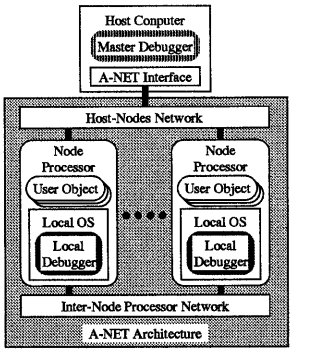

図3: A-NETLデバッガの構造

A-NET計算機における外界とのインターフェイスは，ホスト計算機上のA-NET計算機インターフェイスモジュールのみである．よって，デバッグ操作を行うには，このA-NET計算機インターフェイスを介して行う必要がある．

また，イベントの検出やブレークポイント設定などの実際のデバッグ処理は，各ノード内で行う必要がある．A-NETLデバッガでは，実際のデバッグ処理を各ノード上のローカルデバッガが行い，ホスト計算機上のマスターデバッガが各ローカルデバッガを制御してデバッグ情報を表示する．

(1) マスターデバッガ

マスターデバッガは，A-NETLプログラム開発環境とともに，ホスト計算機となるUNIXワークステーション上で動作するX Windowシステムを利用したアプリケーションとして実現される．

(2) ローカルデバッガ

ローカルデバッガは各ノード上に配置されており，PE上に割り付けられている各オブジェクトに対するリプレイ時の実行制御機能やブレークポイントデバッグ機能を実現するモジュールである．ローカルデバッガは，デバッグ用マイクロプログラムであるファームウェアモジュール（以下FWレベル）とローカルOSのデバッグ用システムメソッドであるOSレベルモジュール（以下OSレベル）により構成されている．

4 A-NETL デバッガの実現

前節で述べた基本設計に基づき，A-NETL デバッガを実際に構築するための実現方法を述べる．

4.1 ブレークポイント

A-NETL の提供するブレークポイントデバッギングは，各オブジェクトの特定のメソッドのソースコード 1 行分を実行した後に，プログラムを中断する機能である．また，プログラム中断の際には，ユーザが指定した変数の内容を確認することが可能である．

A-NETL の行うブレークポイントの検出はマイクロプログラムで実現される．この方式では，プログラムにデバッグ用のコードを挿入する方式に比べてオーバヘッドを軽減することが可能である．また，プログラムコード自体を変更する必要がないので，デバッグ時と通常実行時の振舞いの差異を小さくすることができる．

4.1.1 ブレークポイントの検出

ブレークポイントデバッグモードに入ると，対象オブジェクトの実行モジュールは，A-NETL デバッガの制御下におかれ，指令が出るまでフリーズされる．

ブレークポイント検出処理は，マスターデバッガからの要求により開始される．まず，フリーズされている実行イメージを解放して通常実行させる．ローカルデバッガの FW レベルでは，ブレークポイント情報リストを参照し，実行中のプログラムカウンタが登録されているブレークポイント対応していれば，得られた情報を検出情報配列に格納して OS レベルにトラップする．このとき，実行イメージは再びフリーズされる．

OS レベルでは，検出情報配列の内容をもとに観察すべき変数の内容を引数としたメッセージをマスターデバッガに返す．マスターデバッガでは，受け取った情報をもとに変数内容をウィンドウに表示する．

4.1.2 インデックス付きオブジェクトの扱い

A-NETL には同種のオブジェクトを定義するインデックス付きオブジェクトが存在する．A-NETL デバッガは，これらのオブジェクトに対するブレークポイントの設定を容易に行うための環境を提供する．

インデックス付きオブジェクトのプログラムコードはすべて同一であり，オブジェクトの特定は添字により行われる．このオブジェクト群に同様の設定を行う際に，個別の設定をオブジェクト 1 つずつに行うものでは使いにくい．そこで，A-NETL デバッガでは，同種のインデックス付きオブジェクトに対するブレークポイントの設定を 1 箇所で行うことにより，すべてのインデックス付きオブジェクトに設定情報を反映させることができる．

この機能を実現するためのプログラムコード配置に関する工夫もなされている．インデックス付きオブジェクトの実行部コードは，各ノード内で同一のアドレスに置かれている．また，同一ノードに複数のインデックス付きオブジェクトが載る場合でも，実行部コードは共有される．そのため，ブレークポイントは，全く同一のプログラムカウンタを設定することができる．

4.2 イベント履歴

A-NETL デバッガで採用したイベントベース方式は，プログラムの振舞いを計算機上で再現させ，その上で詳細部の解析を行うためのものである．そのため，イベントとして定義する事象は，プログラム全体の振舞いを忠実に再現するのに必要かつ十分な情報を持つものでなければならない．また，リプレイを行うために，イベントを単にプログラム実行過程の一事象ととらえるだけでなく，リプレイの実行単位としての意味を持たせる必要がある．

4.2.1 イベントの定義

A-NETL デバッガでログとして記録する事象には表 1に示すものがある．

表 1: A-NETL デバッガで記録されるログ

分類	事象
メッセージ通信関連	メッセージ送信
	メッセージ受信
ユーザオブジェクト	起動
実行イメージ遷移関	終了
連	中断

A-NETL デバッガでは，ローカルイベントとグローバルイベントの2つのイベントを定義する．ローカルイベントはログとして記録した事象1つ1つを表すイベントであり，発生した事象を解析する際の単位として用いられる．グローバルイベントはローカルイベントの集合であり，リプレイ処理における同期単位として用いられる．

4.2.2 論理時間の規定

2.2項で述べたようにA-NETLプログラミングモデルでは，プログラム全体にわたる統一された時間概念が明示的には存在しない．よって，A-NETL デバッガではプログラム実行過程の解析を行うために論理時間の概念を導入し，プログラム全体にわたる時間表現を行う．A-NETL デバッガではイベントに時間的意味を持たせ，ローカルイベントの発生時刻を表すローカル論理時間，グローバルイベントの発生時刻を表すグローバル論理時間の2つの論理時間を用いる．図4は，論理時間とイベントの関係を示している．

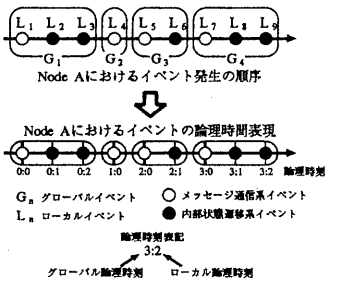

図 4: 論理時間とイベントの関係

4.2.3 イベント履歴記録

ユーザがイベント履歴生成処理実行を選択した際に，マスターデバッガはA-NET 計算機上の各ノードに制御メッセージを送信し，全てのノードをイベント記録モードを変更する．その後にプログラムを実行し，各ローカルデバッガにおいて実行過程のログを記録する．

プログラムの終了判定は，ユーザがインターフェイス上で記録終了を選択することにより行い，マスターデバッガは制御メッセージを全ローカルデバッガに送信し記録処理を終了させる．その後，マスターデバッガは，記録されたログを整形してイベント履歴を生成する．

(1) ログの記録

各ノードのローカルデバッガは，マスターデバッガからのイベント記録開始要求を受け取ると，システムをイベント記録モードに変更する．プログラムが実行されると，各ノードにおいて表1に定義した事象をログとして記録する処理を行う．この処理は，各事象の検出処理とログデータの記録処理に分類される．

各事象の検出処理　ログデータとして記録する事象には，ローカルOS レベル/ファームウェアレベルの2種類がある．メッセージの受信や実行イメージの遷移はローカルOS の処理中に検出され，メッセージ送信や実行イメージ終了などの機械命令レベルの事象は，ファームウェアにおいて検出される．

イベント記録モードにおいて各事象を検出すると，ローカルデバッガのOS レベルに処理が移行しログの記録処理を行う．

ログデータの記録処理　ログデータの記録処理では，検出された事象に関するデータをイベントデータとして整形し，システム変数であるイベント履歴辞書に一時格納される．格納されたイベントデータは，ガーベッジコレクション発生時にファームウェアモジュールにより，事象の発生順にログファイルに格納される．

整形されたイベントデータには，事象発生時にタイムスタンプが付加される．イベント記録モードでは，送信メッセージにもタイムスタンプが付加されているので，ログデータ記録後にメッセージの送信側と受信側との実行順序関係を規定できる．

(2) イベント履歴の生成

実行過程ログの記録が終了すると，マスターデバッガにおいてイベント履歴生成処理を開始する．

まず，マスターデバッガは履歴情報ファイルを生成し，テスト実行時のプログラム実行環境に関する情報を格納する．その後，各ローカルデバッガが記録したログデータを一定量取り込み，ログのデータ構造を生成する．次に，各ノードごとに記録した順にログを比較し，ログのタイムスタンプからイベント発生時の各ノード間のグローバル論理時間，ロー

カル論理時間を割り出す．割り出された論理時間ごとにログデータを整理し，ローカルイベントデータを生成して各ノードに対応する履歴データファイルに格納する．

この処理を全てのログデータがなくなるまで繰り返すことによりイベント履歴が生成される．

4.3 リプレイ

A-NETLデバッガにおけるリプレイは，マスターデバッガとローカルデバッガが同期を取りながらプログラムを再実行させることにより実現される．この同期の単位はグローバル論理時間であり，各ノードにおいて1グローバル論理時間の実行過程を1実行サイクルとする．

マスターデバッガは，ユーザがリプレイ実行を選択した際に，全ノードのローカルデバッガに1グローバルイベント分のリプレイ実行要求メッセージを送信する．各ノードのローカルデバッガは，そのノードに配置されているオブジェクトの実行過程をイベント履歴に基づいて矯正しながら実行し，1グローバルイベント分の実行終了を検出するとマスターデバッガにリプレイ実行終了メッセージを送信する．マスターデバッガは全ノードから実行終了メッセージを受け取ると，それらのメッセージ引数から実行による各ノードの状態遷移に関する情報を得て，リプレイ用情報表示ウィンドウに表示する．以上のような処理をインクリメンタルに実行していくことによりリプレイを実現する．

(1)1サイクル実行開始処理

各ノードのローカルデバッガがマスターデバッガから1サイクル実行要求を受け取ると，1サイクル実行開始処理を行う．この処理では，イベント履歴に記録されている振舞いとリプレイにおける再実行時の振舞いの時間的なズレを検出し補正するために，現在ノードの状態が1サイクル実行可能か否かを判断する．

1サイクル実行可能であれば，ユーザプログラムの実行イメージを実行可能状態にスケジューリングして，1サイクル分の実行を開始し，実行過程観察処理に移行する．1サイクル実行不可能であれば，プログラムを実行せずに1サイクル実行終了処理に移る．

(2) 実行過程監視処理

実行過程監視処理では，プログラムの再実行時におけるローカルイベントの発生を検出して，各ノードのシステム変数であるイベント履歴辞書に格納されているローカルイベントの発生順序との差分を監視する．この処理は，ローカルデバッガのFWレベルにおいて再実行時のイベント発生を検出した場合はOSレベルにトラップし，OSレベルにおいて検出された場合は差分監視用メソッドを呼ぶ出すことで実行される．

再実行時のローカルイベントと履歴内のローカルイベントの発生順序に差分が生じた場合，OSレベルは履歴データと等価なプログラムの振舞いを得るために，ローカルOSのスケジューリングに関与し，イベント履歴に記録されているローカルイベントの発生順序をもとに再実行時の実行過程を矯正する．

(3)1サイクル実行終了処理

1サイクル実行終了処理では，リプレイ処理の終了検出やマスターデバッガへの1サイクルの実行終了通知，次回の実行サイクルのための準備をローカルデバッガのOSレベルにおいて行う．この処理に移行した際に，OSレベルではその時点で実行中であるイメージを一旦退避させ，ユーザプログラムの実行をフリーズする．

実行過程監視処理から移行してきた場合，履歴データファイル内に格納されている次回の実行サイクルに対応するグローバルイベントデータを読み込んでローカルデバッガ内に格納し，イベントの発生時間を表すグローバル論理時刻を論理時間カウンタに格納する．

最後に，マスターデバッガへ1サイクル実行終了メッセージを送信する．このとき，イベント履歴内の全事象を実行していれば，リプレイ実行が終了したこともあわせて送信する．

(4) オブジェクト状態観察処理

オブジェクト状態観察処理では，ローカルデバッガがユーザの指定したオブジェクトの状態に関する情報を解析し，その内容をマスターデバッガに通知する処理をOSレベルにおいて行う．

OSレベルでは，マスターデバッガからのメッセージにより解析対象の情報を得る．実行中のオブジェクトからこれらの情報を取得し，得られた情報を引数としたメッセージをマスターデバッガに返す．

この処理は1サイクル実行終了処理の後に行われるため，ユーザプログラムの実行がフリーズされた状態であり，プログラムの振舞いにまったく影響を及ぼさない．

5 おわりに

本稿では，A-NETLデバッガの設計方針と実現方法について述べた．現在A-NETLデバッガは，UNIXワークステーション上にプロトタイプを開発中であり，残念ながら評価を行うまでには至っていない．今後は，プロトタイプを完成させるとともに，本稿で述べた問題点を中心にデバッギングシステムとしての評価を行う予定である．具体的な評価項目としては，以下のものが挙げられる．

1. プローブ効果の軽減などのデバッグ時に生じる問題をどの程度解決できているか．
2. デバッグによるオーバヘッドが使用に耐え得るか．
3. ユーザの必要とする情報が提供でき，使いやすいシステムとなっているか．

これらの評価結果からシステム性能の改善もあわせて行っていく．

謝辞

本研究は一部，文部省科研費（試験研究(B)課題番号04555077，重点領域研究（超並列）課題番号04235104，奨励研究課題番号06780233）の補助を受けている．

参考文献

[1] 馬場: “超並列マシンへの道”, 情報処理学会誌 Vol.32 No.4, pp.348-364(1991).

[2] 馬場, 吉永: “並列オブジェクト指向トータルアーキテクチャA-NETにおける言語とアーキテクチャの統合”, 電子情報通信学会論文誌 Vol.J75-D-I,pp.563-574(1992).

[3] 片平, 吉永, 馬場: “並列オブジェクト指向言語A-NETLのデバッギングシステム”, SWoPP '92, 92-PRG-8, pp.155-162(1992).

[4] C.E.McDowell and D.P.Helmbold: “Debugging Concurrent Programs”, ACM Computing Surveys, 21, 4, pp.598-622(1989).

[5] 山田: “並列処理システムにおけるプログラムデバッギング”, 情報処理学会誌 Vol.34 No.9, pp.1170-1178(1993).

[6] 吉永, 馬場: “トポロジカルなプログラミングが可能な並列オブジェクト指向言語A-NETL”, 電子情報通信学会論文誌 D-I 採録決定(1994).

[7] Lazzerni,B., and Prete,C.A.: “DISDEB: An interactive high-lebel debugging system for a multi-microprocessor system”, Micro Process. Microprogram., 18 401-408(1986).

[8] LeBlanc,T.J., and Robbins,A.D.: “Event-driven monitoring of distoributed programs”, Proc.International Conference on Distributed Computing Systems, IEEE(1986).

[9] 中尾 他: “並列論理型言語KL1のデバッグ環境”, SWoPP '92, 92-PRG-8, pp.131-138(1992).

[10] 舘村 他: “PIE64 のマルチウィンドウデバッガ HyperDEBU における並列プログラムの実行制御”, SWoPP '90, 90-ARC-83, pp.193-198(1990).