1H-4 並列オブジェクト指向言語DinnerBellのデバッグ環境

渡部眞幸，河野真治，青柳龍也，田中英彦
東京大学　工学部

1.DinnerBellのデバッグ支援システム

現在われわれは、並列オブジェクト指向言語DinnerBellのデバッグを効率良く進めるためのシステムを実装中である。

DinnerBellは並列言語ではあるが、各オブジェクトは特定のものを除いて状態を持たず、その限りではプログラムは deterministicに進行する。nondeterministicな部分は、メッセージのJoin機構によって実現されるため、このメッセージJoinのみに注目してメッセージの同期関係を比較的容易に抽出することができる。

本デバッガは、プログラムの実行履歴を蓄積しておき、その情報を用いて次の２つの方向からデバッグを行うことを目指している。すなわち、

① deterministicな部分については、Shapiro の Algorithmic Debugging 〔1〕の方式に基づき、計算木中の誤りnodeをDivide and Queryによって探索する。

② nondeterministicな部分については、抽出されたメッセージの同期関係をPetri-Net で表現し、その安全性や活性の問題を解くことによって、同期機構の誤りやデッドロックを検出する。

図１に全体の構成を示しておく。

2.履歴情報の蓄積

履歴情報は、runtime でプログラムを実行した際にPrologのfact形式で蓄積される。一方、tokenizer を通してプログラムソースをmethod単位に分割したものを、やはりPrologのfact形式で蓄積する。

DinnerBellでは、コンテクストと呼ばれるDestinaton、Message（含Argument）、Reply の三つ組を実行の基本単位とする。プロセッサはプールから適宜取り出したコンテクストをreduceし、生成したコンテクストをプールに戻すという動作を繰り返す。このコンテクストに着目し、履歴情報として、

(1) D、A、Rにそれぞれ変数ではない確定値を持つ、reduce後のコンテクスト（コンテクストのインスタンス）のリスト

(2) コンテクスト１のreduceによりコンテクスト２が生成された場合、１→２というコンテクスト間の生成関係（コントロールフロー）

(3) コンテクスト１の実行結果（Reply ）を、変数を通じてコンテクスト２のD、AまたはRが参照している場合、１→２というコンテクスト間のデータ参照関係（データフロー）

を蓄積する。例えば整数の階乗を計算させるプログラムを実行した場合には

```
context(1,('Fact',0),'fact:',('INT',5),('INT',120)).
context(2,('INT',5),'-:',('INT',1),('INT',4)).
context(3,('Fact',0),'fact:',('INT',4),('INT',24)).
context(4,('INT',24),'*:',('INT',5),('INT',120)).

parent(1,2).
parent(1,3).
parent(1,4).

refer(1,4).
refer(3,2).
refer(4,3).
```

等々のfactが得られる。一方、tokenizer を通して得られるソースのメソッドリストは、

```
method('Node',[('Cdr','Cdr'),('Car','Car')],[[reverse('Tail')|answer(_47494)]],[(pseudo(sender,1),[answer(('Cdr',[reverse(('Node',[and('Tail'),cons('Car')])])])])|_47489],_47489).
```

という形で与えられ、ソースプログラムの復元、Petri-Net fragmentの生成に用いられる。

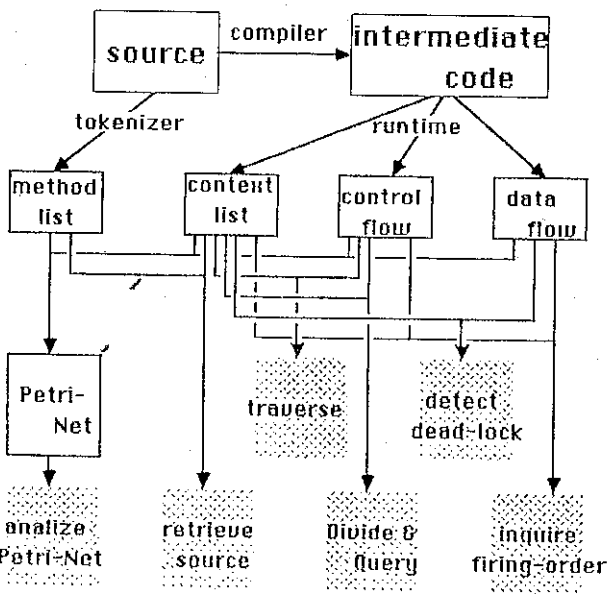


図１　DinnerBellデバッグ支援システム

3.計算木上のバグ探索

deterministicな部分に関しては、蓄積された履歴情報からコンテクストをnodeとし、コントロールフローをarc とした計算木を構成する。プログラマは、計算木上をtraverseして実際にreduceされたコンテクストを系統立てて眺めることができる。計算木上ではShapiro のAlgorithmic Debugging の方式に基づき、Divide and Queryによって誤りコンテクストの探索を行う。すなわち、計算木を２つのsubtree に分割し、分割点のnode（コンテクスト）のReply が正しいか否か Queryを発する。プログラマのAnswerに従って一方のsubtree を選び、この手順を繰り返して誤りnodeを同定する。

4. data-dependencyを利用したデバッグ

runtime は、コンテクスト間のデータフローに関しても情報を蓄積する。この情報はfragmentからPetri-Netを合成する際に用いられるが、それ以外にもこの情報を直接用いた一より消極的ではあるが簡便な一デバッグ機能を付加している。

データフローはコントロールフローと共に、コンテクストを発火する順序を直接決定している。したがって、

An Environment for Debugging of Concurrent Object Oriented Language 'DinnerBell'

Masaki WATANABE, Shinji KONO, Tatsuya AOYAGI and Hidehiko TANAKA
The Univercity of TOKYO

ある副作用を持つオブジェクトに対してメッセージを送ってくるコンテクストを集めて、それらの間に発火順序が定まっているかどうかを data-dependencyや control-dependency をたどって確認することができる。例えば、display への出力を実行するコンテクスト

```
Ⅰ  StdOut   write:  1
Ⅱ  StdOut   write:  2
Ⅲ  StdOut   write:  3
```

の間に Ⅰ→Ⅱ→Ⅲ というdependencyが存在すれば、display には”１２３”と出力されることが、プログラムの上で保証されていることになる。

今１つ、データフロー情報を直接用いたデバッグ機能に、デッドロックの検出がある。これは、data-dependency がループになっているとデッドロックを引き起すという前提に立って、参照関係のfactからそのようなループを検出する。

5.Petri-Net 表現を用いたデバッグ

DinnerBellのプログラムをPetri-Net で表現することによって、プログラム中のメッセージJoinの部分が抽出され、その同期機構が明確化されることについては前回の発表〔２〕で述べた。ここでは具体的にどのようにPetri-Net を構成していくかについて述べる。

基本的には、プログラム中のメソッドはPetri-Net のtransitionと同等である。そこでまず、tokenizer を通して得られたソースプログラムのメソッドリストから図２に示したようなPetri-Net のfragmentを作り出す。ここでtransitionへの入力place にあたるのはプログラムではhead部のメッセージであり、出力place はbody部で新たに送り出されるメッセージである。しかし、これだけではメソッド間のコントロールフローだけしか表現されず、データフローの情報は欠落してしまう。データフローに関しては全てをPetri-Net 上で明らかにする必要はないが、メソッド発火の条件であるdestination の値の受け渡しは、発火の順序関係を明らかにしておくために必要である。そこで、あるメソッドのdestination が他のメソッドの返す値を参照している場合には、そのデータの流れをそれぞれのメソッドの入力place 、出力place として付け加えておく。

このようにして作られたfragmentをruntime から得られた実行履歴に従って接続していく。その際、同一のメソッドであっても異なるオブジェクト（インスタンス）内で処理されたものは区別しなければならない。したがって先に得られたfragmentは履歴中に現れたインスタンスの数だけ複製される。

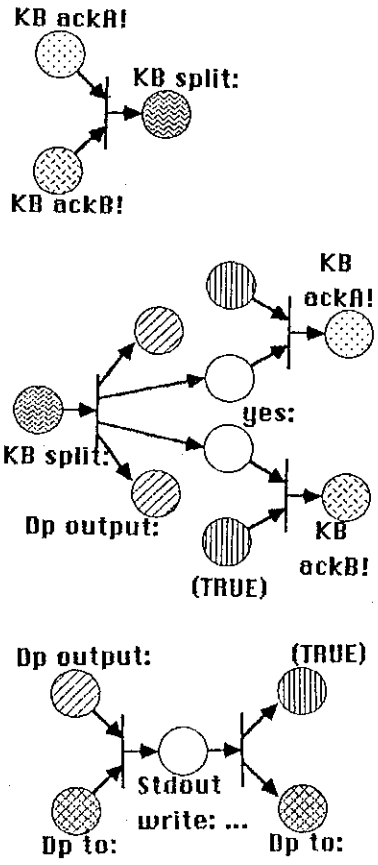
接続に関しては、同一インスタンスに与えられる同一メッセージのplace を１つにまとめる、のが原則である。注意が必要なのは、複数のインスタンスから１つのインスタンスに同一メッセージが送られる場合、発火されたメソッドがメッセージの送り手に答えを返さなければならないときには、送り手を識別する必要からこれらのメッセージを１つのplace にまとめることはできない。その点にだけ注意を払えば、あとは全く機械的に接続を進めることができる。図の例では、KeyBoardとDisplay のインスタンスが２つずつ作られ、４本のループがDisplay class の ‘to：’ place を介して２本ずつまとめられる様子を示している。

6.まとめ

DinnerBellは、並列言語といってもJoin機構以外の部分は単一代入則に従って deterministicに実行されるため、nondeterministicな部分と切り離してそれぞれに特徴的なデバッグの方法を採用した本システムの効果は大いに期待される。両者の役割は相補的であり、統合的なデバッギングシステムを構築している。

参考文献

〔１〕 B.Y.Shapiro, “Algorithmic Program Debugging,” M.I.T.Press, Cambridge (1982).
〔２〕 渡部，河野，田中，情報処理学会第35回全国大会，3R-6，(1987).



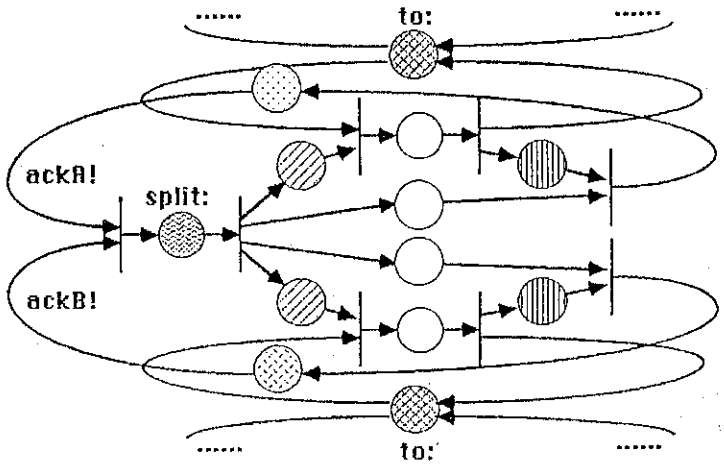


図２ Petri-Net のfragmentとその合成