5D−2

並行プログラムの挙動に対する静的解析

島崎敦　　原田賢一
慶應義塾大学

1 はじめに

本研究では、動的なイベントの履歴をチェックする方法の欠点（刺し針効果など）を克服するために、静的に並行プログラムの挙動(イベント間の順序関係)を解析する手法を提案する。この手法を考えた場合、動的解析の欠点を回避できるばかりでなく、動的解析ではある一つの挙動についてのみしか調べることができなかったのが、すべての考えられる挙動に対して解析を行なうことができるという利点が生まれる。

2 概要

本研究では、Ada風の同期機構を持つプログラムを対象とし、動的解析で行なわれている方法と同様に、プログラムの実行によって引き起こされると予想されるイベント系列に関する記述の集合と、実際の並行プログラムの挙動との整合性を静的にチェックする。プログラムの挙動を記述する方法として、挙動仕様(Behaviore Specification)[1]を本研究用に手直ししたものを用いる。また、静的解析の方法として、実行順序計算[2]による方法と到達可能な状態計算[3]による方法を組み合わせて用いる。

解析は、一度に作成される状態数を削減し、また、エラー状況をつかみやすくするために、すべての条件分岐の組合せごとに次の1～3までを繰り返す。

1. フローグラフから、解析に必要な到達可能な状態を求める。
2. その状態から、イベント間の順序関係を表すグラフを作成する。
3. そのグラフ上で、実行順序計算を行ない、挙動仕様に対してチェックを行なう。

3 システムの構成

本システムでは、図1に示すような解析手順をとる。本研究では、特定の並行処理言語の使用は避け、プログラムにおける制御構造をモデル化したものとしてフローグラフを用いる。与えられた挙動仕様を、yaccで記述された変換プログラムに通すことによって、与えられた並行プログラム(フローグラフ)の挙動をチェックする解析関数を作成する。並行プログラムから状態, グラフを作成する関数と、この解析関数をコンパイルし、プログラムを与えてやることによって解析結果を得ることができる。

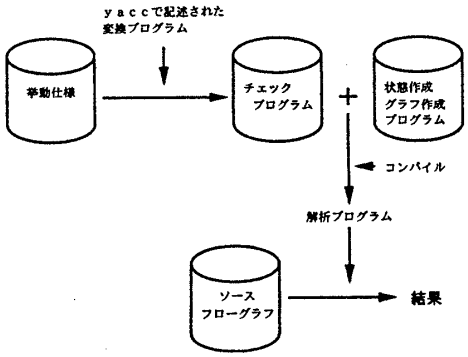

図 1: 挙動仕様の解析手順

4 挙動仕様

本システムでは、静的に解析を行なう都合上、プロセスの状態などについて言及できず、よって、プログラムの流れに注目をおいて挙動仕様を決定した。その文法を図2に示す。それぞれ";" はイベントの逐次実行、"|" はイベントの並行実行、"+" はイベントの選択実行、"[]" はイベントの非決定的実行、"!","@" はイベントの繰り返し、"^" は指定されたイベント以外のイベントの実行を示す。なお、イベント自身の定義は、将来的に現在のシステムを実際の並行処理言語に対する解析システムとすることを考え、現在はフローグラフ上の文（ノード）に直接イベント名を指定して付加する方法をとっている。これは、実際の並行処理言語を対象とした時に、実際の文をイベントの対象として考えることを想定している。

5 状態とグラフの作成

文献[3]の研究に基づき、解析に必要な状態をフローグラフから作成する。解析に必要な状態とは、その後作成するイベント間の順序関係を表したグラフを作成するのに必要な状態のことである。ここでは、グラフを作成するた

Static analysis for checking event sequences in concurrent programs

Atsushi Shimazaki　　Kenichi Harada

Keio University

```
bs:-      BEGIN be END
        ;

be:-      be ; be
        | be | be
        | be + be
        | be [] be
        | (! be )
        | (@n be )
        | ^ EVNET
        | EVENT
        ;
```

図 2: 挙動仕様

めに、考えられるあらゆる同期を求めることを目的としている。また、後々エラーの状態を解析するにあたって、いつ如何なる条件でエラーが発生するのかをユーザに提示できるようにすることも目的としているため、フローグラフ上の考えられるパス毎に状態を作成し、解析を行なう方法をとる。

グラフは、文献 [2] で使用されたグラフとほぼ同じものであり、イベント間の順序関係を表す。作成された状態から、同期をとるノードを調べ、それらの実行順序を反映するよう各ノードを実行順序を表すエッジで結ぶ。

6 解析

解析には、文献 [2] で用いられた実行順序集合を使用する。本研究では、$BEFORE(n), AFTER(n)$ の実行順序集合を使用する。BS における ";" の解析は次のように表される。

$$AFTER(E_1) \cap BEFORE(E_2) = \phi$$
かつ
$$E_2 \in AFTER(E_1)$$
ならば
$$E_1; E_2 = true$$

一方、"|" は、次のように表される。

$$AFTER(E_1) \cup BEFORE(E_1) \not\ni E_2$$
ならば
$$E_1 | E_2 = true$$

これらを組合せ、グラフ上のイベント（ノード）に適用することにより、挙動仕様のすべてを解析することができる。

7 実装上の問題

すべてのパスの組合せで、すべての同期を求めることから、状態作成時において膨大なメモリと時間を必要とすることが考えられる。条件分岐の組合せ毎に解析を行なうことで、メモリの削減はある程度達成されるが、以前として時間の問題は残る。ツールとして考えた場合、その対策として、

1. 記号実行を行なうことにより、不必要なパスの解析を省く。
2. ユーザからの指示により、指定したパスの場合についてだけ解析を行なう。

といった方法が考えられる。また、あり得ないパスを解析することにより生じる不要なエラーも多数発生するが、この対策として、

1. あり得ない条件の組合せをユーザが指示することにより、自動的にそのパスを選択して発生したエラーを取り除く。
2. その組合せについては、挙動仕様や、イベントなどを変更した後も解析対称外とし、再度指示する手間を省くとともに、解析時間を減少させる。

などが考えられる。

8 終りに

並行プログラムの挙動に対する静的解析方法に関する報告を行なったが、以前ツールとして考えた場合、ユーザインタフェースとエラーの解析に関しては不十分である。今後、これらの点について改良を行なっていく予定である。

参考文献

[1] Baiardi, F. DeFrancesco, N., and Vaglini G.: Development of a debugger for a concurrent language. *IEEE Trans. Softw. Eng. SE-12*,4(1986),pp.547-553.

[2] Bristow, G., Drey, C., Edwards, B., and Riddle, W.: Anomaly detection in concurrent programs. In *Proceedings of the 4th International Conference on Software Engineering*. IEEE, 1979 ,pp.265-273.

[3] Taylor, R. N.: A general-purpose algorithm for analyzing concurrent programs. *CACM Vol.26* ,No.5(1983), pp.362-376.