ICOT Technical Memorandom: TM-0990

TM-0990

時相論理証明器の並列化について

松本　一教、本位田　真一（東芝）

December, 1990

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

計算機アーキテクチャ　83－1
（１９９０．７．１８）

時相論理証明器の並列化について

松本一教　本位田真一

株式会社東芝
システム・ソフトウェア技術研究所

時相論理の証明器を応用するプログラム検証や生成では、証明を高速に行うことが不可欠である。本稿ではSCTLという簡潔な体系を用いることで、証明器の並列化による高速化が達成できることを示す。SCTLの証明器はタブロー法にもとづいており、論理式のモデルの候補を生成する部分と、候補を絞りこむ部分から構成される。第1の部分に関しては実際にマルチPSIによる実装を行ってその効果を確認した。第2の部分に関しては、その実装の基本となるデータ構造に関する提案を行う。

On Parallel Theorem Prover for Temporal Logic

Kazunori MATSUMOTO and Shinichi HONIDEN

Systems and Software Lab., TOSHIBA Corp.,

70 Yanagi-cho, Saiwai-ku, Kawasaki 210, Japan.

In this papaer, parallel theorem prover for Simplified Computaional Tree Logic(SCTL) is studied. SCTL was recently proposed by Emerson. He also gave the polynomial time theorem prover for SCTL. We give the parallel version of SCTL prover which is based on Tableaux method and consits of two main stages like Emerson's original algorithm. We have implemented it on Multi PSI system. Basic data structures and experimental results are also shown in this paper.

はじめに

並列プログラムの検証や自動生成のために、古典論理に時間の概念を導入した時相論理*(Temporal Logic)*が有効であることが知られている。例えば、時相論理によるプログラム生成として*Manna[84]*や*Emerson[82]*の方法がよく知られている。それらの方法では定理の自動証明にもとづいており、プログラム生成のために要する計算時間に関する問題があった。そこで筆者らは、時相論理の定理証明器を並列化することにより高速化することを試みている。本稿では、*SCTL*という時相論理の定理証明器の並列化に関するいくつかの実験結果などについて報告する。

1. 分岐時間時相論理*(Branching Time Temporal Logic)*

この体系では図*1*に示すように、各時点で複数の分岐があるような構造として時間を扱う。したがって、"どのパスにおいても"や"どれかのパスでは"といった記述が必要となる。実用的には、このような体系を用いることによって、非決定的なプログラムを容易に取り扱えるという利点があり、*Emerson[82]*による*CTL (Computational Tree Logic)*などがこうした体系の*1*つとして提案されている。*CTL*では、**AX**(どの分岐の次のノードでも)、**EX**(ある分岐の次のノードでは)のような記号を導入している。

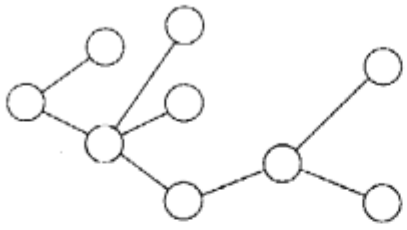

図*1*. 分岐時間のモデル

*CTL*論理式に対する決定手続きとして、タブロー法という方法が知られている。まず、*Emerson[82]*にしたがってその概要をまとめておく。

論理式Gに対して、$Blocks(G) = \{H_1,...,H_n\}$は$=/G$ と $=/H_1$ または, ...,または $=/H_n$ が同値、を満たす論理式の集合である。また、$Tiles(H) = \{G_1,...,G_n\}$は、$=/H$ と $=/G_1$ かつ, ...,かつ $=/G_n$ が同値、を満たす論理式の集合である

このような*Blocks*および*Tiles*への分解を形式的に行うのがタブロー法であり、与えられた論理式$f = \{G_0\}$に対して$Blocks(G_0)$をまず構成し、その各要素に対して$Tiles(H_i)$を構成していく。この構成は必ず停止する。また、構成規則に従って、*Blocks*の各要素を*OR-successor*、*Tiles*の要素を*AND-successor*と見なして作った*2*部グラフを*Tree*-オートマトンとして走らせることにより、論理式fのクリプケモデルが得られることも知られている*(Emerson[85])*。このタブロー法は、図*2*のアルゴリズムのように、大きく*2*つの部分に分けることができる。*stage1*では、与えられた論理式のモデルとなる可能性のあるグラフ*(Tree*-オートマトン*)*が生成され、*stage2*では*stage1*で生成されたグラフの中からモデルとはなり得ない部分が除去される。

このアルゴリズムの単なる並列化は容易であ

```
decision procedure for CTL formula
begin
        do stage1          /* モデルの候補者をすべて生成する(2部グラフとして表現)*/
        do stage2          /* 候補者の中からモデルとはなり得ない部分を削除 */
        if graph = ∅ then the formula is unsatisfiable
        else
            the graph is the collection of all models of the formula
end.
```

図*2*. タブロー法の概要

る。例えば、$Blocks(G) = \{H_1,...,H_n\}$が得られたとき、各$H_i$に対する$Tiles(H_i)$の構成を並列的に実行すれば良い。しかし、

(1) 必要なプロセッサ数が、指数的なオーダになること

という問題がある。また、プロセッサの台数に制限がある場合には、

(2) 論理式によって生成されるグラフの大きさが異なるため、負荷分散が困難である

という問題もある。

このような理由により、*CTL*のタブロー法をそのまま並列化することはあまり効果が見込めない。そこで、筆者らは最近*Emerson[89]*により提案された*SCTL*という体系を用いることにした。この体系は、*CTL*よりも表現力の点で劣るのであるが、タブロー法による検証に要する時間が短いという利点がある。また、次章で述べるように並列化が容易で、効果的である。

られている。まず、*Emerson[89]*から*SCTL*で許される論理式の表現についてまとめておく。

(1) $\vee a$ *(initial assertion)*

(2) $\mathbf{AG}(\vee a)$ *(invariance assertion)*

(3) $\mathbf{AG}(P \Rightarrow \mathbf{AX}(\vee a) \wedge (\wedge_i \mathbf{EX}(\vee b_i))$ *(successor assertion)*

(4) $\mathbf{AG}(P \Rightarrow \mathbf{AF}(\vee a))$ *(leads-to assertion)*

(5) $\mathbf{AG}(P \Rightarrow \mathbf{A}((\vee a)\ \mathbf{U}\ (\vee b)))$ *(ensures assertion)*

の形をした*CTL*論理式だけに制限したような体系が*SCTL*である*(*実際にはさらに細かい制約がある。直接文献を参考にされたい*)*。ここに、a, bは命題変数の集合を、Pは命題変数を表す。

```
Algorithm of stage1
begin
    for all successor assertions AG(P => AX(∨a) ∧ (∧_i EX(∨b_i))), do in parallel
      begin
      create an AND-node P
      for all sub-formula EX(∨b_i), do in parallel
        begin
        create a OR-node EX(∨b_i)
        create an arc from P to EX(∨b_i)
        for all e ∈ b_i, do in parallel
          begin
          create an AND-node e
          create an arc form EX(∨b_i) to e
          end
        end
      end
end.
```

図*3. stage1*の並列化アルゴリズム

2. SCTL(Simplified CTL)

*Manna[84]*や*Emerson[82]*の体系では、古典命題論理を完全に包含しているため、その計算量は少なくとも***NP-complete***なものになってしまう。*SCTL*では、命題論理に関する表現能力を最少限にとどめることにより、効率化がはかこのような*SCTL*論理式に対する決定手続きは、*CTL*の場合と同様なタブロー法を特殊化して用いることができる。まず、*successor assertions*のみを用いて、*stage1*を実行し、残りの形式の論理式は*stage2*で処理する。*stage1*の並列化アルゴリズムを図*3*に示したが、これは$O(n)$のプロセッサにより$O(1)$の時間

で実行できることがわかる。

3. stage1に関する実験

前章で示したアルゴリズムを実際に実現して、その性能を実験した。実験にはマルチ*PSI*(*[MPSI89]*)および*KL1*(*[KL189]*)を使用した。なお実験のための論理式として、相互排除(*mutual exclusion*)の問題をプロセス数を変化させて用いた。

3.1 相互排除の問題

この問題では、プロセス*i*が*non-critical section*(NC_i), *trying section*(TR_i), *critical section*(CS_i)の3つの状態を、この順序で遷移するものとして考える。そうして、例えば図4に示したような論理式として、各プロセスの状態に関する要求を記述する。図4では簡単のために*CTL*論理式で記述したが、この問題は*SCTL*でも記述することができる(*Emerson[89]*)。

$\mathbf{AG}(\sim(CS_1 \wedge CS_2))$
(いかなるときにも,CS_1とCS_2が同時に*critical-section*に入ることはない)

$\mathbf{AG}(TRY_1 \rightarrow \mathbf{AF}CS_1)$
$\mathbf{AG}(TRY_2 \rightarrow \mathbf{AF}CS_2)$
(いかなるときでも、*trying-section*に入れば次には*critical-section*に入れる。*Starvation*に陥らないないことを示す)

図4. 相互排除の記述例の一部

このようにして記述された論理式を先に示したようなタブロー法によって検証するのであるが、逐次実行した場合には図5に示すような実行時間を要する。

プロセス数	2	3	4
実行時間(*msec.*)	45	348	3950

図5. プロセス数と逐次処理時間

それでは、図3のアルゴリズムに従って、*SCTL*決定手続きの並列化について検討する。筆者らが使用したマルチ*PSI*には*CPU*が16台しかないため、すべてを並列的に実行しても意味がない。そこで、先のアルゴリズムの最も外側の*for*ループのみを並列実行することとし、内側のループは逐次的に実行した。このように逐次処理を導入したとしても、*SCTL*論理式による仕様記述には図6のように*CPU*の個数以上の*successor assertions*が含まれるため、プロセッサ割当ての問題は依然として残っている。このことに関して、2つの方式による実験を行った。

プロセス数	2	3	4	5
*suc. assertions*数	9	27	81	243

図6. 仕様記述中の*successor assertions*の数

方式*1*.
論理式を読込みながら、次々にプロセッサへと処理を渡して行く方式。処理を渡すプロセッサは、以前に選択されてから最も時間の経過しているものを選ぶ。

方式*2*.
まず全論理式を読込み、均等な長さに区切ってから各プロセッサへと処理を渡す。

これらの方式での実験結果を図7に示す。

プロセス数	4	5
方式*1*(*sec.*)	3.3	32.6
方式*2*(*sec.*)	1.5	9.7

図7. 各方式による実行時間(*5-CPUs*の場合)

方式2では、まず全ての論理式を読込むため、読込み時間は決定手続きを行うことができない。しかし、いったん読み込みが終了すればプロセッサ間の通信はほとんど行われないという特長がある。一方、方式*1*では、論理式の断片を読込むたびに処理が並列的になされるので、待ち時間は少ないが、通信の回数は方式2より多いという特徴がある。

結局、方式2のようにデータをまとめて通信回数を減らしたほうが効率的であることが解ったので、次にこの方式2をプロセッサ数を変化させて

実験してみた。結果を図8に示す。

この実験結果から、SCTLの決定手続きのstage1に関してはほぼ理想的な並列処理が行えることが解った。

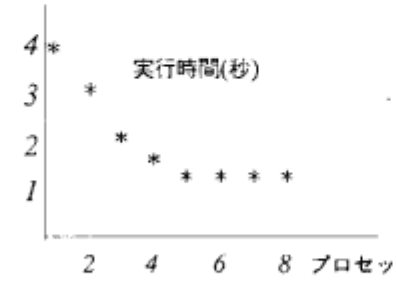

図8(a). stage1の並列実行、プロセス数 = 4)

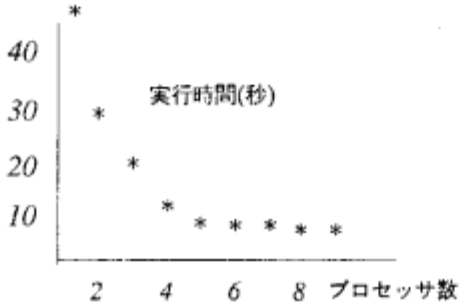

図8(b). stage1の並列実行、プロセス数 = 5)

4. Stage2の並列化

前章でstage1の並列化については殆ど理想的にできることが解った。次にstage2の並列化について考察する。stage2では、次に示すような削除規則によって、stage1で作成されたグラフのノードを削除していく。

rule1.(invariance assertionの処理)
グラフからinvariance assertionに現れないANDノードを削除する

rule2.(ensures assertionの処理)
rule4と類似だが本稿では省略する。

rule3.(削除の影響伝搬)
子ノードが1つでも削除されたANDノードおよびすべての子ノードが削除されたORノードを削除する。

rule4.(eventuality に関する処理)
ensures assertionsが満足されていないノードを削除する。

stage2の削除ルールについて、対象とする論理式ごとに並列処理を行うことができる。図10にその概念的なアルゴリズムを示した。

さて、図10のようなアルゴリズムを実現する場合、並列的に処理されたグラフを高速にマージするための方法が必要になってくる。そのためにはグラフを表現するためのデータ構造が重要である。そこで、筆者らは隣接リスト表現[Gibbons88]を改良し、ビット毎のAND操作でグラフのマージ操作ができるデータ構造を用いることにした。

このデータ構造では、各ノードに対してその子ノードのリストをビット列の形で表現する。すなわち、グラフのノード数をnとするとき、各ノードn(i)に対して、長さn+1のビット列を対応させる。このビット列のj+1番めのビットは、ノードn(i)からn(j)に至るアークが存在するときに限って1に、存在しなければ0にセットしておく。また、最初のビットはノードn(i)が削除されている(0)、削除されていない(1)を表現している(図9)。

グラフGとG'のマージを考える。添字iによって

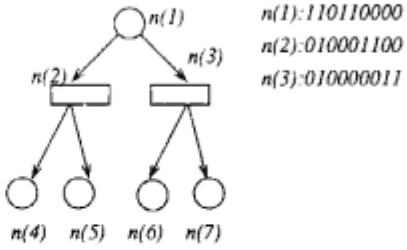

図9. 隣接リストのビット列表現

各グラフにおけるノードiのビット列表現を表し、&でビット列のビット毎のANDをとることを表すことにする。すると、先の削除ルール3を考慮して、
iがANDノードのとき

begin

Gを*stage1*で得られたグラフとする

for all $x \in a$, *where a is in an invariance assertion* **AG**(∨a), ***do in parallel***

begin

Gからノードxを削除したグラフG_xを作る

end

すべてのGxをマージし、グラフG'を作る

for all ensures assertion **AG**(P => **A**((∨a) **U** (∨b))), ***do in parallel***

begin

G'から**A**((∨a) **U** (∨b))の条件を満たさないノードを削除したG_{ab}を作る

end

すべてのG_{ab}をマージする

end.

図*10. stage2*の並列処理の概要

$$G_i \,\&\, G'_i = G_i$$
$$G_i \,\&\, G'_i = G'_i$$

ならノードiはいずれのグラフでも削除されていない。そうでなければ、*rule3*によってこのノードは削除されるので、すべて0のビット列がマージ結果である。

iが*OR*ノードのとき

$$G_i \,\&\, G'_i = \text{すべて0のビット列}$$

ならば*rule3*によりこのノードを削除する。そうでなければ、ビット毎の*AND*をとった結果のビット列がマージした結果である。

このように*AND*ノード、*OR*ノードのいずれに対しても容易に2つの処理結果をマージできる。

おわりに

論理体系を*SCTL*という効率的なものに制限することにより、並列化による効率的な処理の見通しを得た。現在、*stage2*まで含めた並列化アルゴリズムを実装中であるが、現状の*KL1*ではビット列の処理に難点があるという問題があるため、*C*言語による実装も並行して行っている。

今後は*SCTL*の記述能力を詳細に検討して、このような定理証明にもとづく方法の実用化をめざして行く予定である。

なお、本研究は第5世代コンピュータプロジェクトの再委託テーマの一部として行われている。

参考文献

[Emerson82] Emerson.E.A., Clark. E. M., Using Brancing Time Logic to Synthesize Synchronization Skeltons, Science of Computer Programming, Vol.2, 1982.

[Emerson85] Emerson, E.A., Automata, Tableaux, and Temporal Logics, I.NCS-193, 1985.

[Emerson89] Emerson, E.A., et al., Efficient Temporal Reasoning, ACM POPL 1989.

[Manna84] Manna, Z., Wolper, P., Synthesis of Communicating Processes from Temporal Logic Specifications, ACM TOPLAS, Vol.6, No.1, 1984.

[KL189] ICOT第4研究室, KL1ユーザーズマニュアル, 1989.

[MPSI89] 古市昌一,杉野栄二, マルチPSI/V2コンソールシステム操作説明書, 1989.

[Gibbons88] Gibbons, A. and Rytter. W., Efficient Parallel Algorithms, Cambridge Press, 1988.