5U-3

Tokio による論理回路の検証　2
—述語論理検証系の構想—

河野真治・藤田昌宏＊・田中英彦・元岡達
東京大学工学部　＊富士通研

1.時相論理型言語Tokio

Tokio 〔1〕は，時間に関する演算子をふくむ論理（時相論理）に基づくプログラミング言語である。時間に関する演算子は二つあり，LTTL (Linear Time Temporal Logic) のnext (次の時刻，@で表す) と，ITL (Interval Temporal Logic) の chop (前半と後半に分割する操作，&&で表す) である。Tokio は，この論理を，Unification, Reduction, Backtrack により，実行する。

Tokio により論理回路のBehavior Descriptionが可能であることを，これまで示してきた〔2〕。Tokio の記述に対する時相論理に基づく検証系，合成系をつくることにより，総合的な論理設計支援が可能となる。

検証系としては，①Tokio を使う自動検証と，②定理証明のサポートシステム，③Register Transfer Level での検証を考えている。

2.一階述語論理検証の必要性

論理回路は，ALU などの機能素子を含んでおり，同期回路に対する検証のような命題論理による検証だけなく，関数を含む述語論理による検証が望まれる。また，ハードウェアには，多くの並列性が存在し，並列プログラミングとしての難しさがあり，プログラミングサポートとしても，論理に基づく検証が有効である。

述語時相論理は，命題論理とことなり，Undecidable であり，一般的な機械検証は不可能である。しかし，ハードウェア記述やプログラミングの場合は，基になる正しいアルゴリズムおよび論理（仕様）がすでにあり，直接的に，Tokio による実装の記述がその仕様を満たすことを示ばよい。実装が，仕様に近い程検証は容易になる。

3.Tokio の実行と検証の関係

Tokio の実行は，時相論理のresolutionであり，その実行は，Tokio で記述された論理式を満たす状態（各時刻の変数の値や，述語の真理値）を探すことである。したがって，Tokio の実行そのものも検証となっている。しかし，この実行は，有限時間内に止るとは限らない。また，Tokio の論理式は拡張されたclauseであり論理式のsubsetである。時相論理の自動証明系として，Manna のNon clausal Decuction 〔3〕があるが，Tokio は，unification を実装している点が特徴である。

図1 は，哲学者問題のTokio による仕様記述である。

```
ph_think(L,R,I)          :- true
        && ph_left(L,R,I).
ph_left(L,R,I)           :- L=I
        && ph_eat(L,R,I).
ph_eat(L,R,I)            :- L=I,R=I
        && ph_left_off(L,R,I).
ph_left_off(L,R,I)       :- R=I
        && ph_think(L,R,I).

ph_think(L,R,I)          :- empty.
ph_left(L,R,I)           :- empty.
ph_eat(L,R,I)            :- empty.
ph_left_off(L,R,I)       :- empty.

diningPhilosopher(A,B,C) :-
        ph_think(A,B,1),
        ph_think(B,C,2),
        ph_think(C,A,3).
```

図1　Dining Philosopherの記述

```
starve(A,B,I) :- A=3,B=2.

?- diningPhilosopher(A,B,C),
        #starve(A,B,1),length(5).
```

図2　質問の例

Tokio は，この仕様を満たす状態を探す。哲学者が仕様を満たすことができない状態になると，Tokio は，時間を一つ戻して，可能な状態を探す。このような時間の逆行は，実装の記述では，許されない。

Tokio による状態遷移の記述は，いくつかの方法があり，時間の逆行を許す記述と，そうでない記述が可能である。

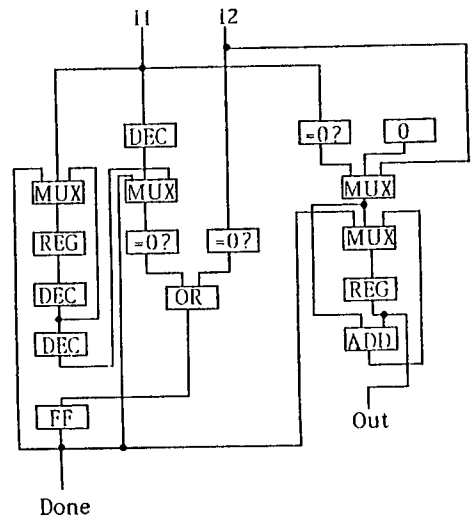

図3　Multiplierの回路

```
mux(Switch,In1,In2,Out) :-
        if Switch = 0
                then Out = In2
                else Out = In1.

reg(In,Out) :- @Out = In.
flipflop(In,Out) :- @Out = In.

dec(In,Out) :-
        if In = 0
                then Out = 0
                else Out = In-1.

adder(In1,In2,Out) :- Out = In1+In2.

zero_test(In,Out) :-
        if In = 0
                then Out = 1
                else Out = 0.

or_gate(In1,In2,Out) :-
        Out1 is In1 \/ In2,
        Out = Out1.

mult_imp(In1,In2,M,N,Done) :-
        #zero_test(In1,B4),
        #(L10 = 0),
        #mux(B4,L10,In2,L9),
        #zero_test(In2,B2),
        #dec(In1,L6), #adder(L9,M,L8),
        #dec(N,L3),
        #mux(Done,L9,L8,L7),
        #reg(L7,M),
        #mux(Done,In1,L3,L1),
        #reg(L1,N), #dec(L3,L4),
        #mux(Done,L6,L4,L5),
        #zero_test(L5,B1),
        #or_gate(B1,B2,B3),
        #flipflop(B3,Done).
```

図4　Multiplierの実装の記述

4.検証の方法

一階述語論理の自動検証では，そのSearch Spaceを縮める方法が問題となる。時相論理演算子の入らない部分については，通常の証明系と同じである。時間に関する部分では，論理式から，その式を真とする状態を見積ることができるがどうかが問題となる。Tokio では，chop をふくむ直接および間接のrecursion があると，その式は，無限の状態をもちえる。このような場合を除く簡単な方法は，全体の時間の区間の長さを制限してしまうことである。このような制限をつけても，設計の仕様にたいする反例をみつけるためには有効である。

```
#p :- p,next(#p).

p :- #p.
next(p) :- #p.
next(next(p)) :- #p. ....

halt(p) :-
     #(if p then empty
                else not empty).

stable(P) :- #(@P=P).
```

図5　用いた規則

もう一つ，Tokio をRegister Transfer Level での記述に用いた場合に，必要となる検証が存在する。Tokio には，staticとよばれるRegisterを表す変数が存在する。この変数は，実際にはシーケンサーから，アクセスされるために，記述内で，矛盾なくアクセスがあるかどうかチェックする必要がある。つまり，同一時刻内で，同じRegisterに対して，ことなる値の書込みがあってはならない。この場合は，RTL の記述が決定的であるので，自動的に行うことが可能である。

5.Tokio の実行による検証例

図1のDining Philosopherの仕様は，Philosopher の状態遷移に対して，時間の制限がない。図2は，この仕様にたいして，長さが5のstarvationがあるうるかどうかの質問である。starvationは，自分のForkが他人にとられている時におきる。Tokio は，仕様からそのような可能性があることをみつけることができる。

6.推論規則による変形による検証

図3は，Hardware Multiplier の実装例である。この例は，〔4〕による。この実装をTokio で記述したのが図4である。これに対して図5の規則により，変形を加えて，図6の記述を得ることができる。図6では，実際に加算に繰返しをしていることが理解できる。この変形は，手作業である。LCF 〔5〕は，これにたいするサポートシステムである。Tokio にたいしても同様なシステムを考えることができる。

```
mult_imp4(In1,In2,M,N,Done) :-
        #stable(In1),#stable(In2),
        @{
                M = In2,
                N = In1,
                Done = 0,
                halt(Done=1),
                intN(In2,Done,M,N)
        }.

intN(In2,Done,M,N) :-
        @M = M+In2,
        @N = N-1,
        (if N-2=0 then @Done=1 else @Done
        next(intN(In2,Done,M,N)).
```

図6　変換後の記述

参考文献

〔1〕8.2,Proc.of the LOgic Prog.Conf.,ICOT,1985

〔2〕情報処理学会第31回全国大会，2j-9,1985

〔3〕Nonclausal Temporal Deduction
Proc. of the Logics of Programs Conf.1985

〔4〕Executing Temporal Logic Programs
Proc.of the NSF.SERC Workshop on the
Sematics of Concurrency, 1984

〔5〕Edinburgh LCF
Lecture Note in Computer Science'78