2K-6

時相論理の仕様から状態遷移表の効率的合成

藤田　昌宏　　田中　英彦　　元岡　達
（東京大学　工学部）

1.　はじめに

我々は既に、時相論理（Temporal Logic）［1］を用いて仕様を記述し、処理系にPrologを用いてハードウェア論理設計を検証することを提案し、ゲート回路や状態遷移レベルのハードウェア記述言語であるDDLの記述に対する具体的手法を示した［2、3］。

本稿では、同期部について、時相論理による仕様から状態遷移表現の自動合成について述べる。状態遷移表現は、時相論理の決定手続き［1］を用いて、仕様である時相論理の記述を現在に対する条件と次の時刻以降に対する条件に展開することで合成することできるが、この手法では、処理時間は最悪の場合、時相演算子の数に対して、指数的に増大する。そこでここでは、仕様記述の性質を利用した効率的な合成法を示す。

2.　時相論理を用いたハードウェア同期部の仕様記述

一般にシステムは、実際に論理・算術演算を行なう演算部と、各演算間のタイミングを制御し同期をとる同期部に分けることができる［3］。

時相論理は、∧、∨、→、～等の古典論理の演算子に、○（next）、□（always）、▽（sometime）、U（until）の4つの時相演算子を加えたものであり、各演算子は次のような意味をもつ。

○P：　次の時刻（同期回路では、次のクロック）にPが成り立つ

□P：　現在から将来ずっとPが成り立つ

▽P：　現在から将来の少くとも1時刻でPが成り立つ

P　U　Q：　現在から考えて、Qが成り立つまではPでありつづける。（ここではQが必ずしも成立することを要求しないweak until［1］を用いる。）

まず、『信号Pがactiveになると次の時刻に信号Qがactiveになる』は、

□（P→○Q）　…①

と表現できる。条件①では、『Pがactiveになると次にQがactiveになる』ことは保証するが、他の場合にもQがactiveになるかもしれない。これを『Qがactiveへ変化するのは、Pがactiveになった次の時刻であり、かつその時に限る』とするには、次の条件を①に付け加える。

□（～Q→（（○～Q）　U　P））　…②

信号の因果関係の表現は、①と②を合せたものが基本となる。実際の仕様記述では、①と②を合せたものを単位として、各信号の立上り、立下り条件を記述していくため、簡単な時相論理の式のANDで表現される。なお、仕様記述例については、参考文献［4］を参照されたい。

3.　合成アルゴリズム

ここでは、時相論理の決定手続きに基づく状態遷移表現の自動合成アルゴリズムについて簡単に説明する。詳細については、参考文献［1］を参照されたい。

合成は次のようにして行なえる。まず、時相論理の各演算子を表1の規則にしたがって、現在に関する条件と次の時刻以降に対する条件に展開する。つぎに次の時刻以降に対する条件についても、一番外側の○演算子を取り除いたものを同じように展開して、次の時刻に対する条件と次の次の時刻以降に対する条件に分ける。このような展開を既に処理したことのある条件と一致するまで繰り返していく。展開の結果得られる全ての条件を処理すれば、各時刻での条件が状態に、各展開過程が遷移に対応した状態遷移表現が得られている。　各時相演算子の展開規則は表1のようになっている。表1は時相論理の公理から得られるもので例えば、＜1＞は『ずっとPであることは、現在Pであり、かつ、次の時刻からみてもずっとPである』ということを表している。また＜2＞は、『いつかPであるということは、現在Pであるか、または、現在はPでなくかつ、次の時刻からみていつかPである』ということを意味する。しかし、＜2＞においていつも～P∧○▽Pを選択していると、ずっと～Pとなってしまい▽Pを満さない。したがって、～P∧○▽Pは『いつかはPを選択する』という条件付きで選択しなければならない。これは'eventuality'と呼ばれるもので、～P∧○▽Pの後ろの｛P｝はこの'eventuality'を示している。

この表を用いることによって、任意の時相論理の式を現在に対する条件と次の時刻以降に対する条件に展開できる。しかし、処理時間は時相演算子の数に対して、最悪の場合指数的に増大する。一方、2．で述べたように仕様は簡単な式のANDで表現されている。そこで、この性質を利用した効率的な合成法について次に述べる。

<1> □P=P∧○□P
<2> ▽P=P∨(～P∧○▽P(P))
<3> P1 U P2=
P2∨(P1∧～P2∧○(P1 U P2)
<4> ～□P=～P∨(P∧○(～□P)(～P))
<5> ～▽P=～P∧○(～▽P)
<6> ～(P1 U P2)=(～P1∧～P2)
∨(～P2∧○(～(P1 U P2))(～P1))

表1 時相演算子の展開規則 ただし、P、P1、P2は任意の時相論理の式

4. 効率的合成法

2．で述べたように実際の仕様は簡単な式のANDで表現され、かつ同じような形の式がかなり多い。したがってよく使われる形の式はあらかじめ展開して状態遷移表現に直しておき、実際の展開においては、その展開された状態遷移表現を組合せて合成することができれば、かなり効率化を図ることができる。

実際の仕様Tは、2．で述べたたように時相論理の式T1、T2、…、Tnに対し、

T=T1∧T2∧…∧Tn …③

の形で記述できる。このことからTに対する状態遷移表現は、T1、…、Tnの各々の状態遷移表現をまとめ、可能な全ての状態遷移をさせることによって得ることができることが分かる。

したがって、状態遷移表現の自動合成は次の合成アルゴリズムのように効率的に行なうことができる。

<合成アルゴリズム>

（ステップ1） 仕様は③の形で記述されているものとする。まず、各Tiは3．で示した方法によって状態遷移表現に展開しておく。

（ステップ2） 各Tiに対する状態遷移表現から図1にしたがってTに対する状態遷移表現を作る。

（ステップ3） 各状態遷移についてeventualityを調べ、満さないものがあればその遷移を削除する。

（ステップ4） 合成された状態遷移表現を簡単化する。

本アルゴリズムは、Prologを用いて実装されている。詳細については、参考文献[4]を参照されたい。

ステップ2において、Ti全部を一度に展開するのではなく、ステップ2、4を繰り返しながら1つずつ展開していくこともできる。このように展開を行なうとした時の処理時間をTaを計算する。合成するハードウェアの入力信号数をNi、出力信号数をNoとし、さらに合成結果として得られる状態数をNsとする。まず、ステップ2、4の繰り返しの回数Nrは、

Nr∝No …④

また、ステップ2、4の一回の繰り返しに要する時間Tcは、合成途中でも状態数はNsの数倍を越えないと考えられるため、

$Tc \propto Ns^2 * (No + Ni)$ …⑤

したがって、Taは

$Ta \propto Tc * Nr \propto Ns^2 * No * (No + Ni)$ …⑥

となり、多項式時間で処理できることが分かる。

一方、合成された状態遷移表現の簡単化（合成アルゴリズムのステップ4）は、もとの状態の数Nsに対し、Ns^2程度の手間でできる。

5. おわりに

時相論理で仕様記述されたハードウェア同期部の効率的な状態遷移表現への自動合成法について述べた。簡単な式のANDの形で仕様記述することにより、十分実用的な時間で処理することができる。今後は、演算部も含めた自動合成[5]へと発展させたい。

参考文献

[1] P. Wolper : "Synthesis of Comminicating Processes from Temporal Logic Specifications" Dept. of Computer Science, Stanford University, No.STAN-CS-82-925, 1982
[2] 情報処理学会、研究会資料、DA21-4
[3] 藤田他：ハードウェア状態遷移表現のPrologによる検証"、情報処理学会論文誌、Vol. 25、No.4、1984
[4] 電子通信学会、研究会資料、EC83-59
[5] 電子通信学会、電子計算機研究会、7月、1984

```
procedure synthesis ;
var L: set of state-set of all Ti ;
   s,s': set of state ;
 begin
   L := {set of initial states of all Ti} ;
   while L ≠ ∅ do begin
   choose a set of state, say s from L and delete it from L ;
    for all sets of input, possible in s do begin
     with this set of input, state-transition s for all Ti ;
     let L' be the set of new sets of states ;
     for each s' ∈ L' do begin
      s' is a successor of s ;
      if s' has not been visited then
        add s' to L ;
     end ;
    end ;
  end ;
 end .
```

図1 個々の状態遷移表現から全体に対する状態遷移表現を求めるプロシージャ