時相論理によるプログラム生成は実用的か?

河野 真治 (Shinji Kono)

e-mail:kono@csl.sony.co.jp
Sony Computer Science Laboratory Inc.
3-14-13, Higashi-gotanda, Shinagawa-ku, Tokyo 141, Japan

October 5, 1995

1 時相論理を使って GUI を設計してみる

X-Window や Windows などの GUI を使った Application は、Button, Menu, Entry などの User が入力する Event と、User に対して表示をおこなう Tools Kit からなる。Tool kit が状態を持たなければ、Event ごとに関数を定義すれば、全体は状態を持たない関数型プログラミング構造となる。このような場合はプログラミングは容易である。例えば、メニューからファイルを選んで見る (変更しない) ような場合に相当する。

しかし、実際には Toolkit の状態に依存した Programming が必要である。

Print 中には Cancel Button だけが有効

という簡単かつ良くある例でも Print 中かどうかという状態に依存する。Interface Builder などによって、このような状態に依存するような処理を合成することは今はできない。

実は、このような記述は時相論理的な記述であり、以下のように簡単に記号的な形で記述することができる。

$$\Box(Print \rightarrow (ButtonEvent = Cancel \vee ButtonEvent = None))$$

このような方法で GUI Program を時相論理を使って設計する手法を考えよう。。

GUI Application は次のように三つの部分に分けて記述する。

命題論理 Event の相互関係を定義する

述語論理 Toolkit の処理と Data Dependency を記述する

スクリプト Toolkit の見かけを定義する

今の場合は、

命題論理 Propositional Interval Temporal Logic

述語論理 First Order Interval Temporal Logic (Tokio [6])

スクリプト Tcl/Tk

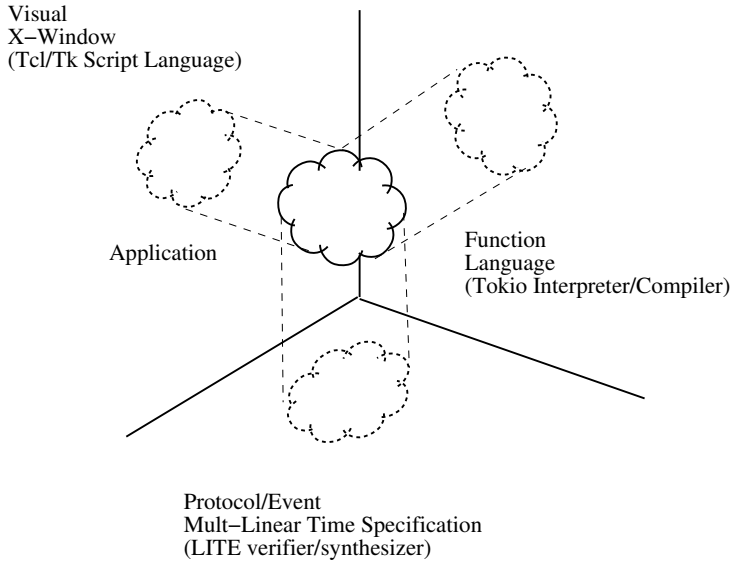

Figure 1: GUI Application の分割記述

を使う。

Toolkit は非常に再利用性が高い Unit だが、簡単な状態処理しかできない。ここでは意図的に Toolkit での Event の相互関係処理をおこなわない。つまり、Toolkit は次のような関数であるとする。

Toolkit 現在の状態 → 次の状態

この関数を述語論理で記述する。

Event を生成する可能性のある Button, Entry, Menu などは、命題論理値を生成する。この Event の値は時間によって異なる真理値を持つ。例えば、時刻 3 には真、時刻 5 には偽などとなる。Event には入力 Event と出力 Event の二つがある。Button などは入力であり、Button の色などが出力となる。出力 Event はユーザによって直接操作されることはない。

Button などの Event を $\mathrm{Event}_1, \mathrm{Event}_2 ... \mathrm{Event}_n$ それを時相論理式で組み合わせた入力条件を p_i としよう。f_iによってその条件に対応する Toolkit の処理を表す。

f_iは、現在の状態 → 次の状態という関数であり、実際には次の時刻を表す@という演算子のみを含む一階述語論理式を使って記述する。

p_iには様相論理演算子を含まない 普通の論理式を使う。例えば、 「青ボタンが押されていればボールがはずむ」などという記述を意図している。

$$
\begin{aligned}
Toolkit_i \quad \equiv \quad & p_{i1}(\mathrm{Event}_1, \mathrm{Event}_2, ..., \mathrm{Event}_n) \quad \rightarrow f_{i1} \\
\wedge \quad & p_{i2}(\mathrm{Event}_1, \mathrm{Event}_2, ..., \mathrm{Event}_n) \quad \rightarrow f_{i2} \\
& ... \\
\wedge \quad & p_{im}(\mathrm{Event}_1, \mathrm{Event}_2, ..., \mathrm{Event}_n) \quad \rightarrow f_{im}
\end{aligned}
$$

このように記述された Toolkit はオートマトンとして容易に実装することができる。

このように記述された Toolkit を複数同時に動作させる。そのためには「いつも p が成り立つこと」を表す$\Box\, p$という演算子を使う。Event の相互作用を表す時相論理式を $schedule(\mathrm{Event}_1, \mathrm{Event}_2 ... \mathrm{Event}_n)$ とすると GUI Application は以下のような時相論理式で表すことができる。

$$schedule(\mathrm{Event}_1, \mathrm{Event}_2, ..., \mathrm{Event}_n) \wedge \Box\ \mathrm{Toolkit}_1 \wedge \Box\ \mathrm{Toolkit}_2 \wedge ... \wedge \Box\ \mathrm{Toolkit}_m$$

$schedule$ には例えば「start button が押されてから stop button が押されるまで青である」などという条件が来る。このスケジューラ部分は状態間の制約が来るので自明な実装は存在しない。この部分を自働検証系を使ってサポートしようというのが今回の狙いである。

このような分割は一種のプログラミング方法論であり、一意的に分割されるわけではない。出力 Event の Button の色などは一階述語部分で記述しても良いし、命題論理部分で記述してもよい。

2 Interval Temporal Logic

ここで使う時相論理は離散時間 (descrete time) と時区間の概念を持っている区間時相論理 (Interval Temporal Logic) というもので、普通の論理演算子や命題変数/述語変数の他に以下のような演算子を持っている。

chop 二つの時区間を一つの時刻を重ねてつなげる。$p\& \ q$で表す。

next 次の時刻から始まる時区間を表す。時区間の長さが 0 の時に無条件に真となる定義するものを$\bigcirc \ p$ で表す。時区間の長さが 0 の時に無条件に偽となる定義するものを$@p$ で表す。

empty 長さ 0 の時区間を表す。

proj 単位時間の変更を表す。単位時間を p という時区間とする、より coarse grained な時間区間 qを定義する時 $p\, proj\ q$と表す。

この論理は命題変数が時刻にだけ依存する場合 (命題変数が local という性質を持つという) は決定可能であることが知られている。ただし、計算量は変数の数に対して指数的、時相論理式の長さ対して指数的とかなり大きい。量化記号を入れても計算量は増えるが決定可能である。しかし、逆にいえば変数の数と式の大きさが小さければ比較的簡単に検証できるともいえる。

量化記号が入ると一階述語論理に近いと考える人もいるだろう。実際、このまま一階述語論理にも tableau method を適応できる場合もあるが、その場合は限られている。もっと弱い時相論理、例えば、Linear Time Temporal Logic に関しては多項式オーダの決定手続きが知られている [5]。

さらに以下のような演算子を定義する。

$$
\begin{array}{lcll}
more & \equiv & \neg empty & \% \text{ 空でない時区間} \\
\Diamond P & \equiv & T\& P & \% \text{ いつか } P \\
\Box P & \equiv & \neg\Diamond\neg P & \% \text{ いつも } P \\
skip & \equiv & @empty & \% \text{ 単位時間} \\
length(n) & \equiv & \underbrace{@@...@}_{n} empty & \% \text{ 長さ n} \\
less(n) & \equiv & \underbrace{\bigcirc\bigcirc...\bigcirc}_{n} F & \% \text{ 長さ n 以下} \\
P\&\&Q & \equiv & (P \wedge \neg empty)\&Q & \% \text{ 必ず時間の進む chop} \\
*\ P & \equiv & (P\ proj\ T) \vee (empty \wedge P) & \% \text{ closure 繰り返し} \\
+P & \equiv & P\& *\ P & \% \text{ 一回以上の繰り返し} \\
fin(P) & \equiv & \Box(empty \Rightarrow P) & \% \text{ 最後で P になる} \\
keep(P) & \equiv & \Box(\neg empty \Rightarrow P) & \% \text{ 最後を除いて P になる} \\
stable(P) & \equiv & keep(@P = P) & \% \text{ 一つの時区間で P が安定} \\
halt(P) & \equiv & empty \Leftrightarrow P & \% \text{ P で最後になる}
\end{array}
$$

これらの時相論理式は命題変数の真偽の時系列σ(有限でなくてもいいが終端がなければならない) を与えると真理値が決まる。正式には時系列を使って時相論理演算子の定義をおこなう。例えば、$\Box\ p$ は p が真である状態を任意の長さつなげた時系列に対して真である。

決定手続きは Tableau method の拡張であり、時相論理式が受け入れる時系列集合をオートマトンの形で出力することができる。

3　一階述語論理と命題論理の組み合わせ

最初に述べた GUI Application の記述のうち命題論理の部分は自動検証系によってオートマトンに変換される。この時、入力である Event と出力となる Event を区別する必要がある。時相論理式レベルでは入出力は区別されない。オートマトンは@演算子を使って述語論理で容易に記述できるし、命題論理変数の状態を Toolkit を記述する一階述語論理式に余分な引数として付け加えてやることによって、すべての記述を述語論理に落すことができる。

一階述語時相論理の実行は Prolog と同じようにおこなわれる。時相論理演算子については不動点演算子つまり再帰と同じように取扱う。従って、このままでは否定の実行や無限に続く再帰に関しては完全には実行されない。

このシステム [3, 2] は時相論理だけなく、状態遷移記述を直接に入力に使うことができるので、X-Window の Tcl/Tk などだけでなく、SIS などの CAD ツールなどにも接続できる。

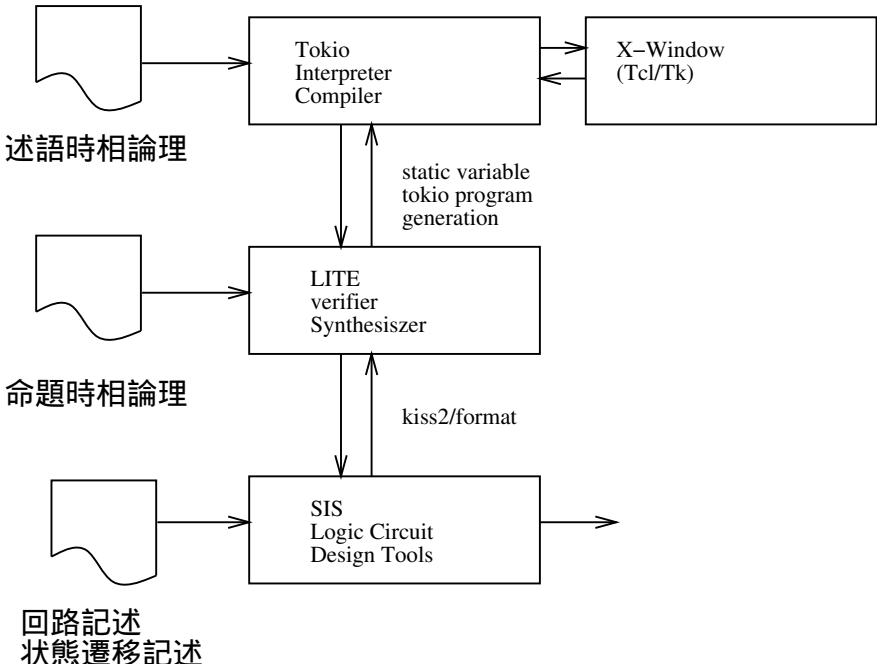

Figure 2: 時相論理プログラム合成システム

この枠組は古くは Wolper[5] に遡るが GUI に使えること、projection を持った ITL を使っていること、証明図が直接出力できることなどが新しい。実際のシステムでは内平, 本位田らの MENDEL[4] という Petri-Net と命題時相論理を結合するものがあり、このシステムはその Petri-Net 部分を述語時相論理に置き換えたものと見ることもできる。

4 簡単な生成例

この例は勝手に動いているボールを Run と Stop の二つのボタンによって制御する例である。時相論理部分は以下のようになる。

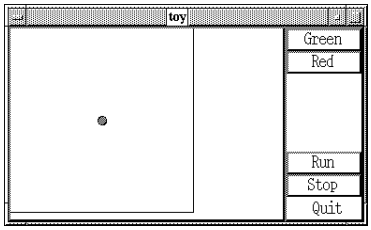

Figure 3: Toy Example

```
st_variables([stop,start,quit],[red,gree,move])).
% 赤ボタンと青ボタンが排他動作
[]((red,not(green);not(red),green)),

% 赤は止まれ、青は動け
[]((green->move)), []((red->not(move))),

%    stop から、ずっと赤
  stop-> keep(red)
%    start から、ずっと青
  start-> keep(green)
% 繰り返しは+で表す
+((
  (stop-> keep(red)),
  (start-> keep(green)),
% 時間の区間の区切りは stop または start または quit
  next(halt((stop;start;quit)))
  )),
% quit ボタンが押されたら終り
halt(quit).
```

4.1 動作記述の例

ツールキット部分は Tokio によって記述される。ここで出て来る時相論理演算子は次の時刻を表す@と、いつもを表す□ だけである。ボールの動作等は一階述語で記述されているのがわかる。

```
% ツールキットを [](いつも) を使ってプロセスとして起動する
toy1(W,R,G) :-
s1, % generated scheduler
[]((button_red(R),button_green(G),bounce(W))).  % output

button_red(Out) :-
        *red =0, out(Out,"",red).
```

```
button_red(Out) :-
        *red =1, out(Out,"Red",red).
bounce([_, View, Obj, _Xout, X, _Yout, Y, Xd, Yd, Ydd, Xlim, _]) :-
*move = 1,*quit = 0,
Y1 is Y+Yd,
        moveto(View,Obj,X,Y),
calc_xd(X, Xd, Xd1, Xlim),
calc_yd(Y, Yd, Ydt, Y1, Yt1),
@X = X+Xd,@Xd = Xd1, @Y = Yt1, @Yd = Ydt+Ydd.
```

4.2 タイミング記述の例 (2)

この動作記述をそのまま使って、同さタイミングだけを変更することもできる。ここでは、信号が勝手に点滅する例を記述しよう。

```
% 状態遷移記述を入力とすることもできる
% 信号は勝手に点滅する

st(s0,(green,not(red)),s1).  % 状態 s0
st(s1,(not(green),red),s0).  % 状態 s1

% projectin によって状態遷移のクロックを長く (length 5) する
    proj(
        (length(5),         % クロックの定義
        % 大きなクロックの間、信号は変わらない
        ((stable(red),stable(green))&skip)
        ),( % 大きなクロックでの状態遷移の実行
        st(s0),halt(quit)
        )
    )
% stop で止まり、start で動く
    +(((stop-> keep(not(move))),
       (start-> keep(move)),
       next(halt((stop;start;quit)))))
```

4.3 利点

命題論理との組み合わせで改善されるのは否定の実行の部分である。命題論理パートに関しては 時相論理演算子と否定と自由に組み合わせることができる。従って実行可能な部分が述語論理を直接実行する場合よりも広がる。実行可能な範囲内でも前もってオートマトンを展開する分だけ高速に実行できる可能性がある。つまり余計なバックトラックを減らすことができる。

最終的には時間を含まない述語論理に落ちているので Prolog のプログラム理論的な利点をそのまま引き継いでいる。例えば、プログラムの意味を集合として与えることができる。特に有限時間内ならば全ての可能なスケジュールを試すことができる。

述語論理部分のプログラミングは再帰がループに書きかわっている分だけ容易になっている。

矛盾した仕様に対しては空のオートマトンが前もって出力される。また、失敗する状態に関しては前もって枝刈りをして状態を減らすことができる。

命題論理部分に関して述語論理部分とは独立にテストすることができる。実際、自動検証系をフルに使ってテストすることができる。テスト条件をオートマトンで記述したり時相論理を使って記述することもできる。

4.4 欠点

プログラミングという視点から見ると二つの部分に分けて記述するのは冗長な気がする。特に自明な部分に関しても他の記述との整合性を考えて二重に記述する必要がある。何もしない Toolkit に関しては何もしないことを$@A = A$のような形で明示的にしていしないといけない。

Prolog に較べて命題論理パートは合成結果を予測することが難しい。もちろん、論理式通りの結果が合成されるわけだが、それが実際にどういう意味なのかを理解することが難しい。

合成に余計な時間がかかるのはうれしくない。命題変数が多い場合は状態遷移数が増える。記述が非決定性を多く含む場合は状態数が多くなる。

4.5 生成されたプログラムの質

合成された結果は仕様を満たす可能な実行をすべて含んでいる。ということは、非常に効率の悪い結果も効率の良い結果も均等に合成されている。可能な実行の内、どれを実行するかは述語論理/Prolog に落した時に決まる。実際には depth first に実行される。

depth first に実行されると分かっていれば生成されたオートマトンをさらに小さくすることができる。例えば、リアルタイムスケジューリングが特定の決まった繰り返しで実現できるならば、その繰り返しを実現するオートマトンだけを使えば良い。しかし、これには次に述べるような入出力に関する問題がある。

一般的にいって生成されるプログラムは馬鹿であるが、それを depth first に実行すると実用的な結果を与えることが多い。そのような特殊な実行のみを生成するオートマトンを採用すると、結果的にオートマトンを小さくすることができる。ということは、最初から効率を考えて合成していく方法をとることによってより大きなプログラムを合成できる可能性があることがわかる。

4.6 入出力の区別が必要な理由

生成されたオートマトンは入出力の時系列に対して真偽を返すだけである。これは特性関数 (characteristic function) と呼ばれる。

$$\text{入力列} \wedge \text{出力列} \rightarrow \text{True/False}$$

これをプログラムと見るためには入力を与えて出力を返すようなオートマトンを作らなくてはならない。

$$\text{入力列} \rightarrow \text{出力列}$$

一般的にすべての入力列に対して出力列が存在するとは限らない。

それは、入力列を $\forall$ でくくることで検証することができる。しかし、仕様そのものをすべての入力列に対して記述すること自身がかなり冗長であり難しい。もちろん、すべての入力列に対して仕様を満たす出力を要求することは論理的な視点からは正しいのだが、大抵の場合、tribial な出力列を付け加えることが必要なだけなので単にめんどくさい作業になる。例えば、停止ボタンと作動ボタンと二つあったとする。停止ボタンと作動ボタンが排他的であるという仮定を入力に対して置くと仕様記述を小さくすることができる。しかも、その生成された結果に対して停止ボタンと作動ボタンを同時に押した入力に対しておこなうべきことは、どちらかだけが押されるという条件を満たすまで実行を遅らせるだけである。

また出力列が存在する場合も、複数可能な出力列から一つを選択する必要がある。ところが、ある特定の出力を選択した場合には将来の入力列に対して特定の仮定を設けることになる場合がある。例えば「5 秒実行」ボタンに対して 10 秒実行する時系列を選択することは、その時点では正しいが 5 秒後以降は間違う可能性がある。Prolog にコンパイルする場合はバックトラックが使えるので条件は若干緩いが本質的にはこの問題は残っている。つまり、

$$\forall \text{入力列} \forall \text{途中までの入出力列} \rightarrow \exists \text{出力列}$$

を保証するようなオートマトンを実際には構築しなくてはならない。これはやさしくない問題だが、入力列が状態依存しないような仕様の場合は自明になる。つまり 5 秒実行するか 10 秒実行するかをユーザにその時点で教えなければ破綻することはない。実際には「ユーザに状態を覚えておいてもらうような仕様」は普通作らないので大抵の場合は、この条件は満たされている。Real-time Scheduling のような問題は「どう頑張っても間に合わない入力列」というのは残るのが許される立場 (Soft Real-time) と許さない立場 (Hard Real-time) がある。Hard Rea-time では実現できる仕様や入力列に対して仮定を置くことが許されているのが普通であるので少しは救いがある。

4.7 生成されたプログラムの量

今回作成した検証系は決定性オートマトンを生成する。従って非決定性をたくさん含むような仕様に対しては巨大なオートマトンを生成してしまう。また運良く小さい場合でも生成されたオートマトンは冗長性を含んでいる。例えば仕様の中に「同等だが異なる項」があると冗長性が入る。通常は仕様には組み合わせると同等になる項は多量にあるので生成された結果には一般的に冗長性がある。

ここで要求されているのは単なる状態数の最小ではなくて前節で述べたような実行系列の探索をおこなう時に便利なように最小化しなくてはならない。表現自身が決定性オートマトンであることも必要ない。蓄積量が少なければ非決定性オートマトンを使っても良い。しかし、この場合は可能な実行系列を探すことが難しくなる可能性がある。

現時点で有効であるとわかっているのは時相論理式の sub term に対する BDD (Binary Decision Diagram) であり、状態遷移をそのまま格納するのに較べて有利である。これは su term の組み合わせに依存する状態数に較べて変数の指数乗に依存する状態遷移数の方が多いからである。通常、入力変数に対してはすべての可能な遷移があるのが普通なので状態遷移数がどうしても多くなる。状態遷移は状態から生成することができるので格納する必要はない。しかも入力状態が決まっていれば状態から一般的な場合よりもより少ない状態遷移を生成することができる。ただし、実験はしてないが、これを実行時におこなうのが望ましいとは思えない。

BDD は状態圧縮の一つの手段であり他の圧縮技術があればそれを採用することを妨げるものはなにもない。

しかしより圧縮を可能にするのはオートマトンの分割である。合成をおこなう際にもより小さな仕様から合成する方が圧倒的に有利なわけであり、最初から適切に分割してある仕様から合成をおこないたい。オブジェクト指向プログラミングは、そういう視点から見て正しい部分と正しくない部分がある。単なる分割では生成されたタイミング制御部分が小さくなるとは限らない。通信変数が増える分だけ大きくなる場合もある。頻繁に通信する部分と疎に通信する部分とを分割するようなプログラミングが望ましい。実際、自動検証が容易なプログラムが良いプログラムだと考えることもできるし、実際にわかりやすいプログラムになる。しかし、具体的にどういう分割が望ましいかは良く分かっていない。また、BDD をうまく圧縮するような分割を見つけるアルゴリズムも分かってない。

4.8 生成された特性関数の実際

比較的制約のきついReal-time schedulingの例を考えよう。

$$
\begin{array}{l}
\% \text{ 非同期プロセス} \\
(((T\ proj\ (length(5) \wedge \Box(dc))) \wedge length(15))\&T) \wedge \\
\% \text{ 繰り返しプロセス} \\
((length(3)\ proj\ @\Diamond(ac)) \wedge T) \wedge \\
((length(5)\ proj\ @\Diamond(bc)) \wedge T) \wedge \\
((length(5)\ proj\ @\Diamond(cc)) \wedge T) \wedge \\
\%\text{排他制御} \\
\Box(((ac \wedge \neg(bc) \wedge \neg(cc) \wedge \neg(dc)) \vee \\
(\neg(ac) \wedge bc \wedge \neg(cc) \wedge \neg(dc)) \vee \\
(\neg(ac) \wedge \neg(bc) \wedge cc \wedge \neg(dc)) \vee \\
(\neg(ac) \wedge \neg(bc) \wedge \neg(cc) \wedge dc) \vee \\
(\neg(ac) \wedge \neg(bc) \wedge \neg(cc) \wedge \neg(dc))))
\end{array}
$$

は以下のような特性関数を生成する。ここで横軸は入力状態、縦軸は出力状態を表す。点は遷移可能な状態の組を表す。 特徴的なのは直線が現れることで、これが非決定性の少ない逐次的な実行

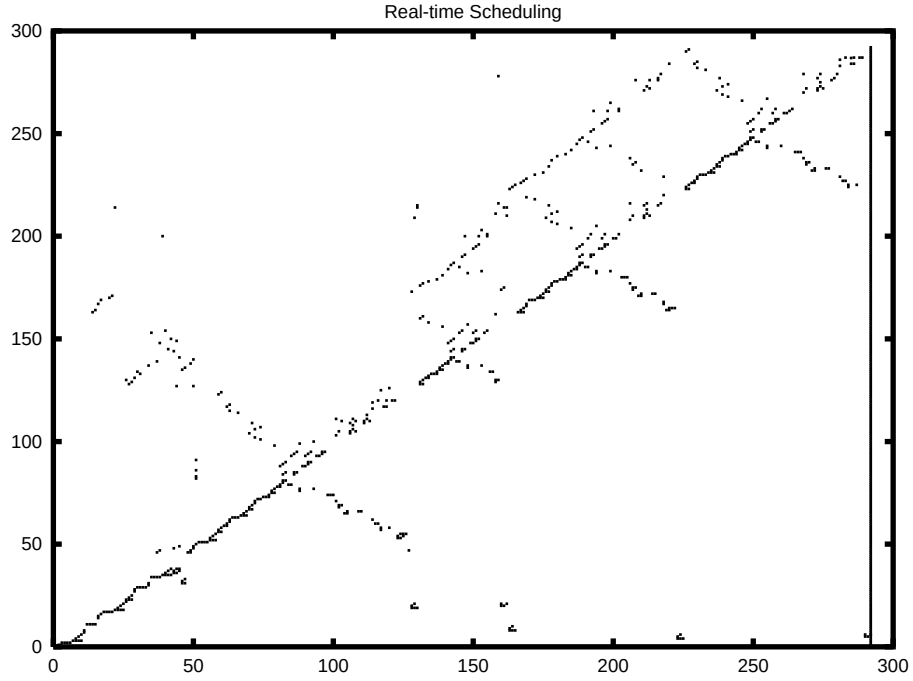

Figure 4: Real-time scheduling の特性関数

を示している。並べ方を変えれば図も代わるが、状態をdepth-firstに生成した順に並べることによる逐次プロセスが直線となる。同じような図形が繰り返しているのは特性関数の冗長性を示している。

次に非決定性を多く含む例として Dining Philosopher をあげよう。

$$
\begin{aligned}
&\%\ \text{five philosophers}\\
&more\wedge\\
&*((\Box((\neg al\wedge\neg ar))\&@(al\wedge\neg ar\wedge@(al\wedge ar\wedge@(ar\wedge\Box(\neg al))))\vee empty))\wedge\\
&*((\Box((\neg bl\wedge\neg br))\&@(bl\wedge\neg br\wedge@(bl\wedge br\wedge@(br\wedge\Box(\neg bl))))\vee empty))\wedge\\
&*((\Box((\neg cl\wedge\neg cr))\&@(cl\wedge\neg cr\wedge@(cl\wedge cr\wedge@(cr\wedge\Box(\neg cl))))\vee empty))\wedge\\
&*((\Box((\neg dl\wedge\neg dr))\&@(dl\wedge\neg dr\wedge@(dl\wedge dr\wedge@(dr\wedge\Box(\neg dl))))\vee empty))\wedge\\
&*((\Box((\neg el\wedge\neg er))\&@(el\wedge\neg er\wedge@(el\wedge er\wedge@(er\wedge\Box(\neg el))))\vee empty))\wedge\\
&\%\ \text{shared resources}\\
&\Box(\neg(ar\wedge bl))\wedge\Box(\neg(br\wedge cl))\wedge\Box(\neg(cr\wedge dl))\wedge\\
&\Box(\neg(dr\wedge el))\wedge\Box(\neg(er\wedge al))
\end{aligned}
$$

これが生成する特性関数は以下のようになる。 ところどころにある黒い島が非決定性の固まり

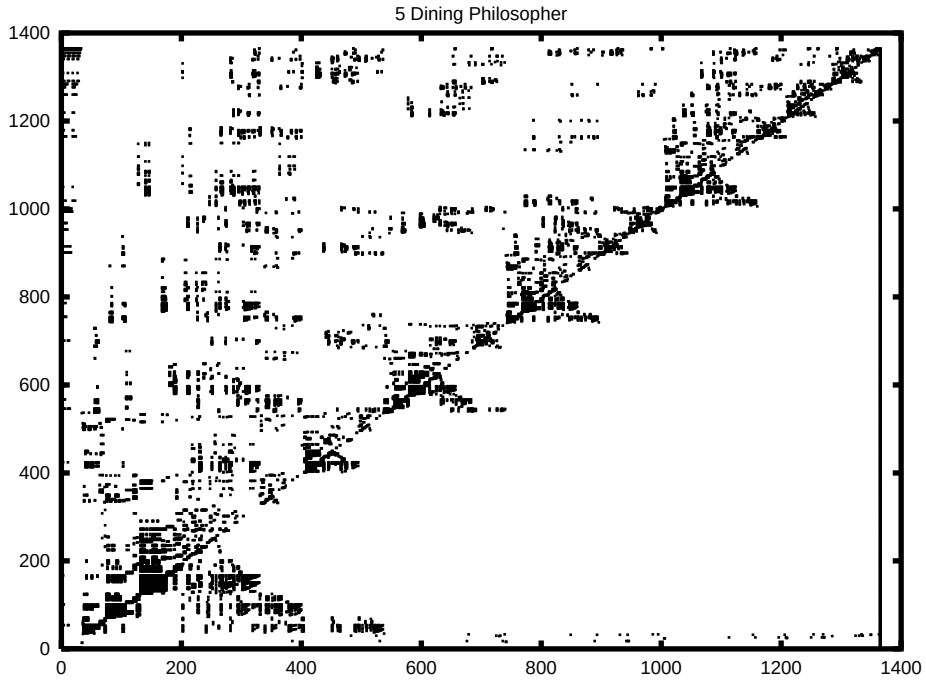

Figure 5: 5 Dining Philosopher の特性関数

である。何人かがデッドロックすると非決定性が少なくなり一点に収束する。ここでも冗長性を示す繰り返しを観察することができる。

5 プログラミングツールとしての時相論理

時相論理の良い点は複雑な仕様をコンパクトに記述できる点であるが、これは一面欠点でもある。ある程度の長さの時相論理の仕様を一部修正すると驚くような大きな変化がある場合がある。一つの方法は前節で述べたように変更しやすいような仕様の分割方法を考えることである。しかし、その分割方法は自明ではない。実際、プログラムを作る時に、どのように構造化あるいはオブジェクトを作っていくかは自明でない。

もう一つ、別な方法として時相論理を直接的な仕様記述ツールとして使うのではなく、オートマトン変更/特性関数変更演算として使うことが考えられる。この視点から見ると ITL の演算子は次のように見ることができる。

chop 二つのオートマトンを直列結合する。

and 二つのオートマトンを並列結合する。

or 二つのオートマトンを非決定的に結合する。

next オートマトンを一つずらす。

proj オートマトンを拡大する。オートマトンの繰り返しを作る。

この変更演算でどのような性質が保存されるかを研究していく方向が一つ考えられる。このような演算は直接オートマトンを計算していけば良いと考えるかも知れない。実際、そのような方法で時相論理を検証する方法もある [1]。しかし、その方法は指数オーダのオートマトンの決定化を繰り返しおこなうために非常に高くついてしまう。Tableau method は指数オーダを要求するが一回で済むという利点がある。実際、変数が少ない場合は一般にオートマトン合成よりも速く結果を得ることができる。

あと、多少、強引で無理な方法だが、これらの時相論理演算子と量化記号を組み合わせて既存のプログラムのパッチ当てをおこなうこともできる。この方法の欠点は、パッチ当てをおこなう際にもとのプログラムの一部を弱くする必要がある。これは量化記号を用いるか状態遷移の一部を取り除くことで可能だが、どちらの方法もプログラムに不規則な非決定性を導入してしまう。これは特性関数という表現を使う場合には特性関数を巨大にする原因となる。

References

[1] Gjalt G. de Jong. An Automata Theroretic Approach to Temporal Logic. In *Computer Aided Verification*. Springer-Verlag, July 1991. 3rd International Workshop, CAV'91.

[2] Masahiro Fujita and Shinji Kono. Synthesis of Contrllers from Interval Temporal Loigc Specification. In *International Conference on Computer Design*, October 1993.

[3] Shinji Kono. A Combination of Clasual and Non Clausal Temporal Logic Program. *IJCAI-93 Workshop on Executable Modal and Temporal Logics*, Aug, 1993.

[4] N. Uchihira. A petri-net-based programming environment and its design methodology for cooperating discrete event systems. *IEICE Trans. Fundamentals*, Vol. E75-A, No. 10, pp. 1335–1347, 1992.

[5] P. Wolper. Synthesis of communicating processes from temporal logic specifications. Technical Report STAN-CS-82-925, Stanford University, 1982.

[6] 河野真治, 青柳龍也, 藤田昌宏, 田中英彦. 時相論理型言語 Tokio の実装. In *Logic Programming Conference '85*, 1985.