ソフトウェア基礎論　27－7
プログラミング言語　19－7
(1988. 12. 9)

Ｏｃｃａｍプログラム検証への時相論理の応用

Application of Temporal Logic to Verification of Occam Programs

佐藤賢二*　元村直行**　荒木啓二郎*
Kenji Satou　Naoyuki Motomura　Keijiro Araki
*九州大学工学部情報工学科　**㈱安川電機製作所
Kyushu University　YASKAWA Electric Mfg. Co., Ltd

あらまし　Ｏｃｃａｍで記述されたプログラムの検証を時相論理を用いて行う一手法を提案する．本手法では，制限されたＯｃｃａｍプログラムにいくつかの変換規則を適用することにより，簡単な文法を持つ別の言語に変換し，その後，時相論理を用いて各種性質を証明する．この変換を行うことで，一時的な並列実行という複雑な計算列が直列化され，時相論理による検証が可能になる．また，本手法では，Ｏｃｃａｍプログラムに存在する様々な形態のデッドロックを統一的に扱うことができる．尚，我々の目的の一つは，㈱安川電機製作所で開発された２腕衝突回避システムの検証を行うことである．

Abstract　A verification technique for Occam programs using temporal logic is presented. In this technique, first, some transformation rules are applied to restricted Occam programs so that these programs may be transformed into another language which has a simple syntax. Then, program properties are proved by temporal logic. This transformation serializes complex execution sequences such as temporary parallelism, and makes it possible to verify Occam programs using temporal logic. In this technique, various forms of deadlocks existing in Occam programs can be uniformly dealt with. One of our purposes is the verification of the transputer system which operates two robots without a collision. This system is developed in YASKAWA Electric Mfg. Co., Ltd.

1．はじめに

研究の背景

並行プログラミング言語を用いた分散処理システム研究の一例として，㈱安川電機製作所では実時間での２腕衝突回避問題を解くための分散システムをＯｃｃａｍ[3]とトランスピュータを用いて構築した[5]．このシステムではdummy情報を用いて一定の通信の順序パターンを形成することでデッドロックを回避しているが，デッドロックに陥らないことの形式的証明がなされているわけではない．又，デバッグ中に経験した幾つかのライブロックはいずれも厄介な問題であった．このような並行プログラム特有の性質の証明に時相論理を用いることの有効性が，いくつかの文献で報告されている[1][4]．

本稿では，２腕衝突回避プログラムを検証する前段階として，Ｏｃｃａｍで記述されたプログラムの検証を時相論理を用いて行う一手法を提案する．この手法ではまず，プリミティブプロセスとコンストラクタから成るＯｃｃａｍプログラムにいくつかの変換規則を適用することで簡単な文法を持つ別の言語に変換する．次に，変換されたプログラムの検証を，時相論理を用いて行う．この変換を行うことで，Ｏｃｃａｍでは扱いにくかった停止性やデッドロックフリー性の証明を形式的に行えることを，簡単な例を用いて示す．

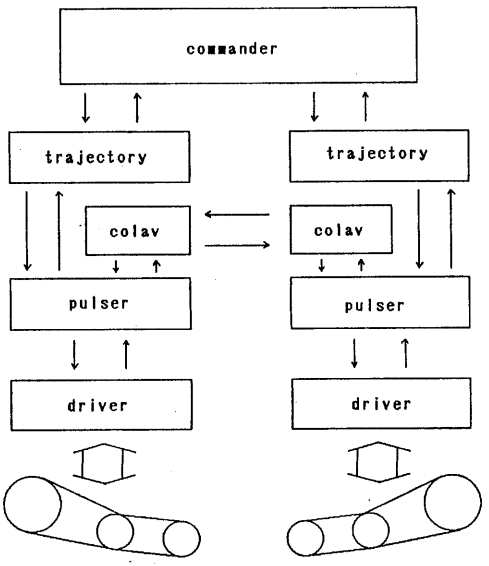

図１．システム構成図

２腕衝突回避システム

システムのソフトウェア構成を図１に示す．トランスピュータは現在，commanderに１個，trajectoryとpulserに１個，driverに１個，colavに１個（計７個）使用している．パルスモータで駆動される２台のスカラ型マニピュレータは２つのリンクと手首を持ち，衝突の危険性がある位置に置かれている．マニピュレータは各々のコントローラ（trajectoryとpulserとcolavとdriverを合わせたもの）で自律的に制御される．各々のcolavは内部にマニピュレータのモデル（ワイヤフレームモデル）を持ち，これをセグメントタイム毎に更新し，その度に相手のcolavにも送信する．衝突の危険性の発見や回避処理はこのモデルに基づいて行う．コントローラは，通常はcommanderから発せられたpoint to point（以下ＰＴＰと記す）の動作指令に従ってマニピュレータを動作させるが，衝突の危険ありと判断した場合はプライオリティの低い方が回避動作を行う．

各モジュールの機能の概要を以下に述べる．

commander:

マンマシンインターフェイスとしてシステムのイニシャライズ，キャリブレーション，直線動作やランダムムーブなど，オペレータからの各種コマンド入力を処理する．ランダムムーブは，各々のコントローラに対して独立なＰＴＰ指令列を乱数により発生させるものである．

trajectory:

commanderからの動作指令を実行する．目標点は２つのマニピュレータの中間を原点とする世界座標系で送られて来るので，それを自分の腕座標系に変換し，各関節の回転角を計算する．

pulser:

trajectoryからのコマンドとcolavからのコマンドのどちらをdriverへ送信するかの判断を行う．通常はtrajectoryからの指令を選択するが，衝突の危険がある時はcolavからの指令を選択する．また，パルスの払い出しによって更新された手首の位置と姿勢を計算しcolavに送信する．

colav:

自分と相手のマニピュレータのモデルとステータスを持ち，その幾何学的な分析により

・衝突の危険性の発見
・衝突回避のための処理

を行う．

driver:

pulserから送られて来る，以下の２つのコ

マンドを実行する.

・マニピュレータのイニシャライズ

・目標点, 目標姿勢への移動

マニピュレータはその動作に応じて次の様なステータスを持つ.

1) waiting
動作指令を待っている状態

2) jogging
キャリブレーション等のために, オペレータによるマニュアル操作で動作している状態

3) moving
目標点に向かって動いている状態

4) working
目標点において作業をしている状態

プライオリティは1)-4)の順で高くしてあり, 衝突の危険性が発見された後の回避処理を決定する情報となる.

コントローラ内部のモジュールのうち, trajectoryとpulserとcolavの間の通信パターンを図2に示す. このように複数のプロセスが互いに通信し合うプログラムでは, デッドロックやライブロックに陥る危険性があるので, 形式的証明が必要となる.

	phase0	phase1	phase2
trajectory		to.pulser ! arm.state | pulse.ready | dummy	fr.pulser ? ack | nack
pulser	to.colav ! position	PAR fr.trajectory ? arm.state | pulse.ready | dummy fr.colav ? avoid | wait | dummy	PAR to.trajectory ack | nack to.colav ! position | arm.state | position. arm.state | dummy
colav	SEQ fr.pulser ? position PAR to.patrner ! position fr.partner ? position	to.pulser ! avoid | wait | dummy	SEQ fr.pulser ? position | arm.state | position. arm.state | dummy PAR to.partner ! 同上 fr.partner ? 同上

図2. trajectory, pulser, colav間の通信

Ｏｃｃａｍ

Ｏｃｃａｍは，抽象的言語ＣＳＰ[2]に基づく並行プログラミング言語である．並行性・チャネルを用いた同期通信・非決定性などの特徴があり，トランスピュータ上で実行されることを意識した言語構成になっている．

Ｏｃｃａｍプログラムは，最小の実行単位であるプリミティブプロセスを，コンストラクタと呼ばれる結合子で結合していくことにより作られる．この結合の仕方によって，逐次実行（SEQ）並行実行（PAR）条件分岐（IF）繰り返し（WHILE）非決定的選択（ALT）などの実行制御方式を指定できる．

時相論理

様相論理から派生してきた時相論理は，プログラムの時間的性質（安全性・生存性・先行性など）を形式的に記述し証明するのに適している．Fred Kröger は文献[1]の中で，簡単な文法を持つ並行プログラミング言語（以下，ＴＬ言語と呼ぶ）を取り上げ，この言語で記述したプログラムの検証を行っている．また，文献[1]では証明を行う形式的な時相論理体系（公理＋推論規則）も併せて提示されていて，この体系は健全で完全であることが証明されている．

ＴＬ言語では，プログラムは複数の並列コンポーネントから成り，各々の並列コンポーネントは逐次的な while プログラムである．並列コンポーネントには非循環コンポーネントと循環コンポーネントの２種類があり，前者は逐次実行後終了し，後者は無限ループである．通信は共有変数によって行われ，

await 論理式（then 実行文）

という文で同期が取られる．並行性はインターリーブモデルに基づいている（すなわち，任意の並列コンポーネント中でロケーションカウンタが指しているラベルに続く文が１ステップ毎に実行される）．また，文には総て一意なラベルが付けられ，現在ロケーションカウンタがラベルλにあることを atλ，次に実行されるのがラベルλを持つ文であることを λ という命題変数で表す．

以上の枠組みにおいて，プログラムの性質は次の時相論理式で表現される．

部分正当性

$start_{\pi} \wedge P \rightarrow \Box(at\alpha_{\theta}^{(1)} \wedge \cdots \wedge at\alpha_{\theta}^{(p)} \rightarrow Q)$

相互排除

$start_{\pi} \wedge P \rightarrow \Box \neg((at\alpha_{i} \vee \cdots \vee at\alpha_{j}) \wedge (at\beta_{k} \vee \cdots \vee at\beta_{l}))$

デッドロックフリー性

$start_{\pi} \wedge P \rightarrow \Box(at\alpha_{i} \wedge at\beta_{k} \rightarrow B_{1} \vee B_{2})$

正当性及び停止性

$start_{\pi} \wedge P \rightarrow \Diamond(at\alpha_{\theta}^{(1)} \wedge \cdots \wedge at\alpha_{\theta}^{(p)} \wedge Q)$

本論文中の表記法および用語についての詳細は文献[1]を参照されたい．

2．変換の方針

Ｏｃｃａｍでは，プログラムの実行中に一時的に並行処理をしたり，その後逐次処理に戻ったりすることができる（図３）．時相論理を使って並列プログラムの検証を行う場合，普通はインターリーブを行うことで各プロセスの計算列を直列化するのだが，Ｏｃｃａｍの場合は一時的な並列実行なので簡単には直列化できない．また，プリミティブプロセスをコンストラクタで結合することでプログラムが構成されるので，ある行のある文が実際にどういうアクションを行うかはその行だけでは決められない．
このような理由で，並列コンポーネントのインターリーブというすっきりした計算モデルを持つＴＬ言語に変換することを考えた．

変換の対象となるＯｃｃａｍプログラムには，以下の制限を付けるが，これらの制限は本質的なものではなく，Ｏｃｃａｍの記述能力を損なうものではない．

５つのプリミティブプロセス（代入，入

力，出力，ＳＫＩＰ，ＳＴＯＰ）および５つのコンストラクタ（ＳＥＱ，ＰＡＲ，ＷＨＩＬＥ，ＩＦ，ＡＬＴ）から成る

レプリケータやプロシージャＰＲＯＣなど，展開可能なものは展開されている

ＣＡＳＥコンストラクタはＩＦコンストラクタの一種と考える．又，データ構造についてもここでは特に扱わない．

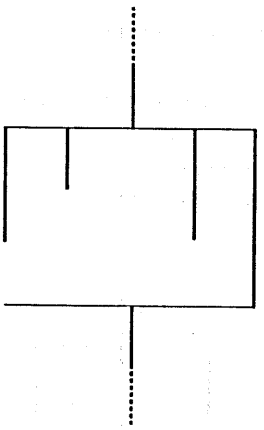

図３．Ｏｃｃａｍプログラムの実行

前述した様に，Ｏｃｃａｍプログラムはプリミティブプロセスをコンストラクタで結合したものである．すなわち，Ｏｃｃａｍにおけるプログラム（プロセス）は

①プリミティブプロセスはプロセスである

②プロセスをコンストラクタで結合したものはプロセスである

のように帰納的に定義される．よって，

①プリミティブプロセスがＴＬ言語で表現可能であり，かつ

②表現可能なプロセスをコンストラクタで結合したプロセス（コンストラクト）がＴＬ言語で表現可能である

ならば，全てのＯｃｃａｍプログラムはＴＬプログラムで表現可能であるといえる．よって，上の条件を満足するよう変換規則を定めた．

変換したプログラムの性質は，前記した時相論理体系を拡張したもので証明する．

３．変換規則

プリミティブプロセスおよびコンストラクトについて変換規則を提示する．この変換はＯｃｃａｍからＴＬ言語への１方向の変換である．

プリミティブプロセス

代入プロセス

ＴＬ言語の代入文に変換する．

入出力プロセス

各プロセス間で一意な名前を持つチャネルを用いた通信を，共有変数による通信と await文による同期で表現する．

出力プロセス

```
channel ! x
```

を，次の様なＴＬプログラム片に変換する．

α_n :chan:=x;fchan:=true;

α_{n+1} :await fchan=false;

ここで，chanは各プロセス間で一意な変数，fchanは同様に一意なブール変数とし，プログラム中ではこのチャネル channelを使う入出力プロセス以外の部分には現れないものとする．また，以下の入力プロセスについても同様とする．

入力プロセス

```
channel ? y
```

を同様に変換する．

β_m :await fchan=true then

y:=chan;fchan:=false;

ＳＫＩＰプロセス

ＳＫＩＰプロセスは何もせず，実行すると即座に停止するプロセスである．これはＴＬ言語では x:=x;（xは任意の変数）という代入文に変換する．この変換は，文献[2]にある規則

(x:=x)＝ＳＫＩＰ

に照らして妥当であることが言える．

ＳＴＯＰプロセス

ＳＴＯＰプロセスは何もせず，実行すると永久に停止しない（即ち，一種のロック状態になる）プロセスである．これをＴＬ言語の await false; という同期文に変換する．

コンストラクト

ＳＥＱコンストラクト

幾つかのプロセスを逐次的に実行することは，ＴＬ言語では各プロセスを変換したプログラム片を順に並べることで表現する．

ＰＡＲコンストラクト

図３の様に一時的に並行実行されることを，図４の様に「並行実行されるプロセスの数だけ並列コンポーネント（P1,…,P4）を用意し，ＰＡＲコンストラクト（main）の実行が開始される時まで待機させておく」という形で表現する．これは，Ｏｃｃａｍのプロセスが「並列実行されるプロセス数は静的に決まっているものの，実行される（実行開始→実行→停止）回数は動的に変化する」ことに起因する．

すなわち，以下のＯｃｃａｍプログラム

```
:
PAR
  P1;
  P2;
  :
  :
  Pn;
:
```

を，以下に挙げるｎ＋１個の並列コンポーネントに変換する．

・ﾒｲﾝﾌﾟﾛｸﾞﾗﾑ

```
:
PARSTART:=true;
    T1:=false;‥‥;Tn:=false;
await T1=true∧‥‥∧Tn=true
  then PARSTART:=false;
:
```

・ﾌﾟﾛｸﾞﾗﾑm（1 ≦ m ≦ n）

```
loop
  await PARSTART=true;
      :
      : 変換されたプロセスPm
      :
  Tm:=true;
  await PARSTART=false
end
```

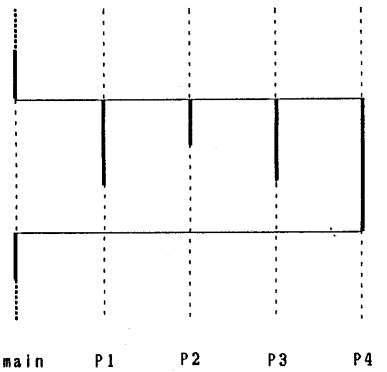

図４．変換規則（PAR）適用後

ブール変数 T1,…,Tn,PARSTARTはプログラムの他の部分には現れないものとする．

ＷＨＩＬＥコンストラクト

ＷＨＩＬＥコンストラクトは，ＴＬ言語のwhile文に変換する．

ＩＦコンストラクト

ＩＦコンストラクトは，ＴＬ言語では入れ子になった if then else文に変換する．ただし，条件式１…条件式ｎが全て偽であった場合，ＩＦコンストラクトはＳＴＯＰプロセスと同じ状態になるので，ＴＬ言語でそれと同

じ意味を持たせるために最後に await false を置く．即ち，以下のＯｃｃａｍプログラム

```
:
IF
  条件式１
    プロセス１
  条件式２
    プロセス２
      :
      :
  条件式ｎ
    プロセスｎ
:
```

は，

```
:
if 条件式１=true then 文の列１
else if 条件式２=true then 文の列２
else if     :
            :
            :
else if 条件式ｎ=true then 文の列ｎ
else await false fi fi… fi fi;
:
```

というＴＬプログラムに変換される．

ＡＬＴコンストラクト

ＡＬＴコンストラクトは非決定的選択を実現するが，ＴＬ言語には非決定性を含む文は無い．従ってＴＬ言語の文法を拡張して，非決定的マッピングを実行する新しい文を導入する．

λ：ｎｄｍ（論理式１，…，論理式ｎ）

そして，このｎｄｍのアクションを定義する一連の公理を証明体系に付加する．

λ∧Ｂ１∧¬Ｂ２∧¬Ｂ３∧…∧¬Ｂｎ
　→ ○（Ｇ１∧¬Ｇ２∧¬Ｇ３∧…∧¬Ｇｎ）

λ∧¬Ｂ１∧Ｂ２∧¬Ｂ３∧…∧¬Ｂｎ
　→ ○（¬Ｇ１∧Ｇ２∧¬Ｇ３∧…∧¬Ｇｎ）
　:
　:
λ∧Ｂ１∧Ｂ２∧¬Ｂ３∧…∧¬Ｂｎ
　→ ○（exor（Ｇ１，Ｇ２）
　　∧¬Ｇ３∧…∧¬Ｇｎ）
　:
　:
λ∧Ｂ１∧…∧Ｂｎ
　→ ○ exor（Ｇ１，…，Ｇｎ）

λ∧Ｐ → ○Ｐ
　※ただし，ＰはＢ１，…，Ｂｎを含まないＰ-論理式（文献[1]参照）である．

Ｂ１，…，Ｂｎ，Ｇ１，…，Ｇｎは論理式，exorは排他論理和を表す．

ＴＬ言語と証明体系に以上の拡張を施した上で，ＡＬＴコンストラクト

```
:
ALT
  c1 ? x
    プロセス１
  bool1 & c2 ? y
    プロセス２
  bool2 & SKIP
    プロセス３
:
```

は，以下のＴＬプログラムに変換する．

```
:
await fc1=true∨(bool1∧fc2)=true
                ∨bool2=true
  then ndm(fc1, bool1∧fc2, bool2);
if G1 then x:=c1;
           fc1:=false;
           文の列１;
else if G2 then y:=c2;
                fc2:=false;
                文の列２;
else if G3 then 文の列３;
:
```

G1, G2, G3はプログラムの他の部分には現れないものとする．下線部はプロセス1および2のガード部の入力プロセスに対応していることに注意されたい．

停止性に関して

上記した変換規則を全て適用した後，最大のコンストラクト（即ち，プログラム）を変換したＴＬプログラムの末尾に

$$\lambda_0^{main}: \text{stop}$$

を置く．変換されたＯｃｃａｍプログラム全体の停止性は，次の論理式で定義される．

$$start_\psi \wedge P \rightarrow \Diamond(at\lambda_0^{main})$$

文献[1]での停止性の定義と異なっている理由は，ＰＡＲコンストラクトを変換した際にできた並列コンポーネントを「エンドラベルλ_eに到達しなくても」停止したとみなしているからである．

4．変換例

1回だけ通信を行う2つのプロセスを並行動作させるＯｃｃａｍプログラム（図5）

```
PAR
  c ! x
  c ? y
```

は，以下のＴＬプログラムϕに変換される．

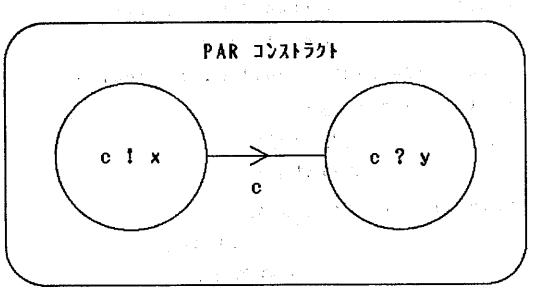

図5．Ｏｃｃａｍプロセス

```
φ ≡ initial true;
    cobegin
      α0: PARSTART:=true;T1:=false;
                          T2:=false;
      α1: await (T1∧T2)=true
              then PARSTART:=false;
      α2: stop
      ||
      loop
        β0: await PARSTART=true;
        β1: c:=x;fc:=true;
        β2: await fc=false;
        β3: T1:=true;
        β4: await PARSTART:=false
      end
      ||
      loop
        γ0: await PARSTART=true;
        γ1: await fc=true
              then y:=c;fc:=false;
        γ2: T2:=true;
        γ3: await PARSTART:=false
      end
    coend
```

プログラムϕのデッドロックフリー性は，ϕ中の3つの並列コンポーネントから1つずつawait文を取り出す組合せ（1×3×3個）について

$$start_\psi \rightarrow \Box(at\alpha_i \wedge at\beta_j \wedge at\gamma_k \rightarrow B_1 \vee B_2 \vee B_3)$$

が導出できることを示せばよい．一例として，

$$start_\psi \rightarrow \Box(at\alpha_1 \wedge at\beta_0 \wedge at\gamma_0 \rightarrow ((T1 \wedge T2)=TRUE) \vee (PARSTART=TRUE))$$

を導出する．まず，ϕ中でPARSTARTに対する代入文はα_0とα_1にしか出現しないことより，

$$start_\psi \rightarrow \Box(at\alpha_1 \rightarrow (PARSTART=TRUE)) \quad \cdots(1)$$

が導出される.

※(1)の導出

$A \equiv at\alpha_1 \rightarrow (PARSTART=TRUE)$と置くと,

(1.1) $start_\phi \rightarrow A$

…実行開始時は$at\alpha_1$が偽なので, Aは真である.

(1.2) A invof {α_0, α_1, α_2を除く全てのラベル}

…これらのラベルに関して, Aは不変式である.

※A invof λは,

$\lambda \wedge A \rightarrow \bigcirc A$

を表す.

(1.3) A invof α_0

…α_0実行後PARSTART=TRUEが真になるので, Aも真になる. よって

$\alpha_0 \wedge A \rightarrow \bigcirc A$

が成立する.

(1.4) A invof α_1

…α_1実行後は$at\alpha_1$が偽なので, Aは真である. よって

$\alpha_1 \wedge A \rightarrow \bigcirc A$

が成立する.

(1.5) $start_\phi \rightarrow \Box A$

…(1.1)-(1.5)および推論規則

(inv) A → B,

B invof {エンドラベルを除く, φ中の全ラベルの集合}

$\vdash A \rightarrow \Box B$

より.

次に,

$(at\alpha_1 \rightarrow (PARSTART=TRUE))$

$\rightarrow (at\alpha_1 \wedge at\beta_0 \wedge at\gamma_0$

$\rightarrow ((T1 \wedge T2)=TRUE)$

$\vee (PARSTART=TRUE))$

…(2)

はトートロジーであるから, この証明体系では公理の一つである. よって,

$B \equiv at\alpha_1 \wedge at\beta_0 \wedge at\gamma_0$

$\rightarrow ((T1 \wedge T2)=TRUE)$

$\vee (PARSTART=TRUE)$

と置くと,

(3) $\Box A \rightarrow \Box B$

…公理

$\Box (A \rightarrow B) \rightarrow (\Box A \rightarrow \Box B)$

および(2)を前提とし, モーダスポーネンスを用いて導出.

(4) $start_\phi \rightarrow \Box B$

…(1)および(3)より.

(証明終わり)

又, φの停止性は, デッドロックフリー性が証明されればφのメイン・コンポーネント(αでラベル付けられている)が非循環コンポーネントであることより即座に導かれる.

5. まとめ

構造的なOccamプログラムを時相論理で検証する一手法を提案した. この手法の欠点は, 特にPARコンストラクタ, ALTコンストラクタおよび入出力プロセスを多く含むプログラムにおいて変換後のプログラムが複雑になる点であろう. 多くのawait文の組合せがあると, 人手で検証するのは難しくなる. 他方, 本手法の長所としては, Occamでは様々な形態のロック(入出力プロセスによるデッドロック, STOPプロセス実行によるロック, IFコンストラクトのSTOP状態によるロック等)が存在するのに対して, 変換後は全てawait文によるロックとなって統一的に取り扱うことができることである.

今後はこれらの変換の(意味保存, 性質保存という意味での)正当性の保証を行い, 2腕衝突回避プログラム検証への応用を試みる.

謝辞

本研究を進めるにあたって御支援頂く㈱安川電機製作所の武田淳男氏に感謝致します. また, 日頃貴重な御助言を頂く九州大学の牛島和夫教授, 吉田紀彦助手, 平原正樹助手並びに研究室の皆様に感謝致します.

参考文献

[1]Fred Kröger:Temporal Logic of Programs, Springer-Verlag, 1987

[2]C.A.R.Hoare:Communicating Sequential Processes, Prentice Hall International, 1985

[3]C.A.R.Hoare (Ed.):occam2 Reference Manual, INMOS Limited, 1988

[4]Pierre Wolper:Specification and Synthesis of Communicating Processes using an Extended Temporal Logic, Proc. 9th Annual ACM Symp. Principles of Programming Languages, pp.20-33, Jan.1982

[5]元村直行: 衝突回避へのトランスピュータの応用, 情報処理学会 マイクロコンピュータ研究会報告, 47-3, 1987