論　文

様相論理による並行プログラムの積重ね式検証法

正　員　内平　直志†

PQL: Modal Logic for Compositional Verification of Concurrent Programs

Naoshi UCHIHIRA †, *Member*

あらまし　有限状態遷移グラフで表現できる並行プログラムの検証には時相命題論理が有効である．しかし，プログラムの規模が大きくなるに従って，検証に要する計算コストが指数関数的に増加する点が実用化への最大のボトルネックであった．この問題の有効な解決策の一つが積重ね式検証法 (compositional verification) である．並行プログラムの等価性判定基準である bisimulation を利用し，各構成要素から個々の検証項目に必要な情報だけを抽出し，検証対象を再構成することによって，検証コストの爆発を回避する．本論文では，時相論理とプロセス論理を融合した様相論理に基づく検証項目記述言語 PQL を提案し，PQL を用いた「内部遷移による無限ループ」を考慮した積重ね式検証手法を示す．

キーワード：検証，並行プログラム，様相論理，遷移システム，等価性

1. まえがき

FA 装置，自動車，航空機，家電製品などにおける装置組込み計算機のソフトウェアの多くは並行プログラムである．それが単一プロセッサの場合でもソフトウェア的にはマルチタスクによる並行プログラムである場合が多い．近年，これらの装置の高機能化に伴ってこの種のソフトウェア開発のための計算機支援環境は不可欠となっている．特に，デバッグは並行プログラムの開発において最も困難なフェイズであり，テストケースに依存しない論理的検証手法に対する期待は大きい．装置組込み計算機における並行プログラムは，その骨格が有限状態グラフで表現できるケースが多く，時相論理，プロセス論理，CCS，ACP などの理論を用いた自動検証が適用可能である[9],[1]．このような背景から，我々は分枝時間時相命題論理である CTL (Computation Tree Logic)[2] を用いた自動検証システム PTSV[14] を開発し，ロボットの制御プログラムの検証に適用してきた．この CTL による検証法を要約すると次のようになる．まず，並行プログラムからその骨格を表す平たんでグローバルな有限状態グラフを生成し，次にそのグラフ上のすべての状態をトレースしながら与えられた仕様が満たされているかを検証する．しかし，CTL による検証では，

（1）検証に要する計算コストが爆発的に増加する (グローバルなグラフが爆発的に大きくなってしまう)

（2）動作の順序関係に関する直接的記述ができない (CTL では「動作」の結果生じた「状態」に対する検証しかできない)

といった問題点があった．

(1)に対する有望な解決策の一つに積重ね式検証法 (compositional verification)[8],[3],[12] がある．積重ね式検証法とは，直接グローバルな有限状態グラフを検証するのではなく，検証項目ごとにローカルな有限状態グラフを生成/検証する方法である．ここで，ローカルな有限状態グラフとは，検証項目の検証に必要十分な部分だけを残し，その他の部分は縮約したグラフである．この積重ね式検証ではグローバルな有限状態を直接生成しないため，有限状態グラフの組合せ的爆発が効果的に抑制できる．Mishra と Clarke[8] は，このアイディアに基づき，回路の検証に CTL による階層的検証法を適用した．しかし，この手法は検証項目に制限があり，洗練された一般的方法ではなかった．また，Clarke ら[3]は CTL の検証に CCS[7] の枠組みを導入した積重ね式検証法を提案した．このとき，CTL が「状

† (株)東芝システム・ソフトウェア技術研究所，川崎市
Systems & Software Engineering Laboratoty, TOSHIBA CORPORATION, Kawasaki-shi, 210 Japan

態」に注目した論理であるのに対し，CCSは「動作」の順序関係に注目した理論であり，この「状態」と「動作」をいかに同じ枠組みで扱うかがポイントとなる．これは，同時に(2)に対する解決策にもなっている．しかし，Clarkeらの検証法[3]では「状態」と「動作」を自由自在に混合して記述することはできなかった．一方，CCSの特性をプロセス論理(Hennessy-Milner Logic : HML)で表現しようとする研究[4],[5]があった．この発展形として，時相論理とプロセス論理を融合した新しい論理が研究されている．この論理では，「状態」と「動作」を自由自在に混在して記述でき，なおかつ積重ね式検証に対応できるので，検証項目の記述言語として有望である．具体的には，Stirlingらによる時相論理とプロセス論理を融合したGeneral Temporal Logic (GTL)[11],[12]がある．GTLは線形時間時相命題論理，分枝時間時相命題論理，HMLなどの命題プロセス論理をサブセットとして含む汎用的な論理である．更に，Stirlingら[12]はGTLによる積重ね式検証法も提案している．

しかし，GTLを用いた検証法には以下の問題がある．積重ね式検証のポイントは，「検証したい項目に必要な情報だけを残す」ように構成要素を抽象化(縮約)する方法であり，GTLによる積重ね式検証では，「検証したい項目に必要な情報」をCCSの観測等価(observation equivalence)に基づき定式化している．すなわち，抽象化(縮約)が観測等価な範囲であれば検証に影響のないことが保証される．しかし，観測等価では，観測不可能な内部遷移による無限ループ(divergence)が無視されてしまう．これは，観測不可能な内部遷移に関して，ある種の公平性(観測可能で実行可能な動作があるならば，観測不可能な内部遷移の無限ループにより表面上のデッドロックに陥ることはない)を仮定しているためである．積重ね式検証の場合，観測可能にするかしないかは単に検証に必要か否かで決定されるので，観測可能な動作と不可能な動作で公平性に関する違いが生じるのは問題である．例えば，図1に示す状態遷移システム T_a において，「いつかは必ずend状態に到達するか？」という質問(検証項目)に対する答は，動作bが観測可能な場合はbの無限ループの可能性を考えてNOだが，bが観測不可能な場合はYESになる．これは，bが観測不可能な場合，観測等価に基づくならば図1の T_a と T_b は等価とみなされるからである．しかし，積重ね式検証の観点からはbが観測不可能か否かにかかわらずbのループの存在は認識されるべきである．観測不可能な内部遷移によるループの扱いについてはdivergenceとして定式化がなされており[6],[13]，divergenceの特性を表現可能なようにHMLを改造したIntuitionistic HML (IHML)[10]が提案されている．Stirlingら[12]はGTLにこのIHML的要素を付加し拡張する可能性(GTL+IHML)を示唆している(具体的提案はない)．しかし，GTL+IHMLを実現したとしても同様の問題は残る．すなわち，IHMLが基づいている等価性(partial bisimulation preorderによる等価性[10])ではどの動作を観測不可能にするかによって直感に合わない抽象化が行われる場合がある．例えば，図1の T_a と T_c において，bが観測不可能な場合にはdivergenceが発生し，IHMLが基づいている等価性では T_a と T_c は等価になる．これは，「aが発生したあとにcに対して反応しなくなることがある」という観測的意味での等価性である．しかし，「aが発生したあとにcが発生不可能な状態が存在するか？」という検証項目に対しては，bが観測可能か不可能かにかかわらず T_a の場合はNO，T_c の場合はYESとなり区別できるのが自然であり，両者を等価とする等価性では「検証したい項目に必要な情報」の損失が生じる．すなわち，GTL+IHMLの積重ね式検証では T_c を T_a に縮約可能であり，必要以上の抽象化が行われてしまう．以上の議論から，従来の様相論理(HML, GTL, GTL+IHMLなど)が基づいている等価性(観測等価，partial bisimulation preorderによる等価)は，プロセス間の同期通信に主眼を置いたものであり，積重ね式検証における抽象化のための等価性(抽象化して再構成しても検証結果が同じ)には不適当である(必要以上の抽象化が行われる)ことが結論できる．すなわち，積重ね式検証法にとっての「観測不可能」は単に「注目していない」ことを意味し，ある動作を観測不可能にしたとしても挙動の特性は保存されなければならない．

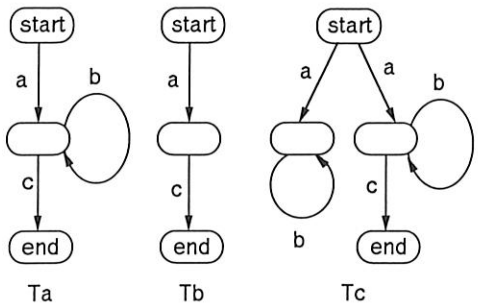

図1　観測不可能な内部遷移による無限ループの扱い
Fig. 1　Transition systems with/without divergence.

本論文では，上記の問題を解決するために，GTL＋IHML とは異なるアプローチから，観測不可能な内部遷移によるループを明示的かつ簡単に扱える，「状態」と「動作」の積重ね式検証を目的とした並行プログラムの検証項目記述言語 PQL (Process Query Language) を提案する．PQL の特徴は，時相オペレータおよび内部遷移によるループの有無を，論理式の最大/最小不動点として統一的に扱う点である．PQL は積重ね式検証に適した新しい等価性判定基準をもつと共に，上記の統一性による簡明な自動検証手続きをもつ．

以下，2.では並行プログラムの表現法および並行プログラムの等価性判定基準を定義する．3.では，検証項目記述言語 PQL を定義し，PQL の識別能力が並行プログラムの等価性判定基準と同等であることを示す．これは，与えられた並行プログラムをサイズの小さい等価なプログラムに縮約しても PQL の検証結果が保存されることを意味する．4.では 3.の結果を利用した積重ね式検証法を提案し，その有効性を実験結果により示す．

2. 並行プログラムの表現

並行プログラムはいくつかのプロセスから構成される．各プロセスはハンドシェイク型の同期通信を行いながら並行に動作する．プログラムおよび個々のプロセスは以下で定義する遷移システムとして表現される．

2.1 遷移システム

[定義 1](遷移システム)

$T=(S, P, A, \pi, \delta, s_0)$

S：状態集合

P：状態属性の集合

A：動作集合

$Act=A\cup\{\tau\}$

τ：観測不可能な内部遷移

$\pi: S\to 2^P$ 真偽値関数(各状態で成り立つ状態属性の集合)

$\delta: S\times Act\to 2^S$ 非決定性遷移関数

s_0：初期状態

但し，有限分岐条件(任意の $s\in S, a\in A$ に対して $\delta(s, a)$ は有限集合) を仮定する．

後に述べるように，自動検証を目的とする場合は S, P, A が有限な遷移システムのみを対象とする．この場合は有限分岐条件は成り立つ．また，Milner[7] の定義にならって，$\delta(s, a)\ni s'$ を $s\xrightarrow{a}s'$，$s(\xrightarrow{\tau})^*\xrightarrow{a}(\xrightarrow{\tau})^*s'$ を $s\overset{a}{\Rightarrow}s'$ と表す．また，$\hat{a}$ は，$a\neq\tau$ のとき a を，$a=\tau$ のとき ε を表すとする (ε は空列，すなわち，$s\overset{\hat{\tau}}{\Rightarrow}s'=s\overset{\varepsilon}{\Rightarrow}s'=s(\xrightarrow{\tau})^*s'$．図 2 に遷移システムの簡単な例を示す．

$T=(\{s_0, s_1, s_2, s_3\}, \{p1, p2\}, \{a, b\}, \pi, \delta, s_0)$ *where*
$s_0\xrightarrow{a}s_1$, $s_0\xrightarrow{b}s_2$, $s_1\xrightarrow{b}s_3$, $s_2\xrightarrow{a}s_3$, $s_3\xrightarrow{\tau}s_0$,
$\pi(s_0)=\{p_1, p_2\}$, $\pi(s_1)=\{p_1\}$, $\pi(s_2)=\{p_2\}$,
$\pi(s_3)=\phi$

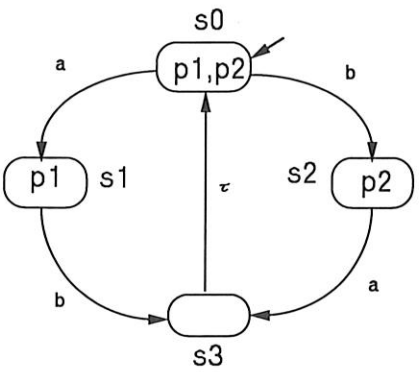

図 2 遷移システムの例
Fig. 2 Example of a transition system.

2.2 遷移システムの等価性

内部遷移によるループを区別する遷移システムの新しい等価性判定基準 ($\pi\tau\omega$ 等価) を定義する．まず，CCS で用いられている等価性判定基準の一つである *bisimulation*[7] をプロセスの状態集合，τ を含む動作集合，遷移関数の三つ組 (S, Act, δ) に対して定義する．

[定義 2](*bisimulation*) (S, Act, δ) において，2 項関係 $R\subset S\times S$ が *bisimulation* であるとは，任意の s, $t\in S$ に対して，

$sRt\Leftrightarrow$

- $\forall a\in Act.\ \forall s'\in S.$ (if $s\xrightarrow{a}s'$ then $\exists t'\in S.\ t\overset{\hat{a}}{\Rightarrow}t'\wedge s'Rt'$)
- $\forall a\in Act.\ \forall t'\in S.$ (if $t\xrightarrow{a}t'$ then $\exists s'\in S.\ s\overset{\hat{a}}{\Rightarrow}s'\wedge s'Rt'$)

bisimulation では，内部遷移によるループが明示的に区別できない．そこで，*bisimulation* を内部遷移によるループが明示的に区別できるように拡張した *τω-bisimulation* を定義する．

[定義 3](*τω-divergence*) (S, Act, δ) において，$s\in S$ のとき，

$s\uparrow\overset{def}{=}\forall n>0.\ \exists s'\in S.\ s(\xrightarrow{\tau})^n s'$

[定義 4](*τω-bisimulation*) (S, Act, δ) において，2 項関係 $R\subset S\times S$ が *τω-bisimulation* であるとは，任意の s, $t\in S$ に対して，

$sRt \Leftrightarrow$

- $\forall a \in Act.\ \forall s' \in S.$ (if $s \xrightarrow{a} s'$ then $\exists t' \in S.\ t \overset{\hat{a}}{\Rightarrow} t' \wedge s'Rt'$)
- $\forall a \in Act.\ \forall t' \in S.$ (if $t \xrightarrow{a} t'$ then $\exists s' \in S.\ s \overset{\hat{a}}{\Rightarrow} s' \wedge s'Rt'$)
- $s \uparrow$ iff $t \uparrow$

更に，(S, Act, δ) に状態属性の集合と真偽値関数を加えた五つ組 (S, Act, δ, P, π) に対して，状態属性の等価性を考慮した *πτω-bisimulation* を定義する．

[定義 5]($\pi\tau\omega$-*bisimulation*) (S, Act, δ, P, π) において，2 項関係 $R \subset S \times S$ が *πτω-bisimulation* であるとは，任意の s，$t \in S$ に対して，

$sRt \Leftrightarrow$

- $\pi(s) = \pi(t)$
- $\forall a \in Act.\ \forall s' \in S.$ (if $s \xrightarrow{a} s'$ then $\exists t' \in S.\ t \overset{\hat{a}}{\Rightarrow} t' \wedge s'Rt'$)
- $\forall a \in Act.\ \forall t' \in S.$ (if $t \xrightarrow{a} t'$ then $\exists s' \in S.\ s \overset{\hat{a}}{\Rightarrow} s' \wedge s'Rt'$)
- $s \uparrow$ iff $t \uparrow$

[定義 6]($\approx$, $\approx_{\tau\omega}$, $\approx_{\pi\tau\omega}$) (S, Act, δ, P, π) において，s_1，$s_2 \in S$ に対して，$(s_1, s_2) \in R$ なる *bisimulation* が存在するならば，$s_1 \approx s_2$ と表す．$s_1 \approx_{\tau\omega} s_2$，$s_1 \approx_{\pi\tau\omega} s_2$ も同様に定義する．

[定理 1]($\approx$, $\approx_{\tau\omega}$, $\approx_{\pi\tau\omega}$ の関係) (S, Act, δ, P, π) の任意の s_1，$s_2 \in S$ に対して，$s_1 \approx_{\pi\tau\omega} s_2$ ならば $s_1 \approx_{\tau\omega} s_2$，$s_1 \approx_{\tau\omega} s_2$ ならば $s_1 \approx s_2$ である．

(証明) 定義より明らか． □

$s_1 \approx s_2$ は観測等価 (observation equivalence) と呼ばれている．本論文では，$s_1 \approx_{\pi\tau\omega} s_2$ を「*πτω* 等価」と呼ぶ． □

[定義 7]($\pi\tau\omega$ 等価) 二つの遷移システム $T_1 = (S_1, P, A, \pi_1, \delta_1, s_{01})$，$T_2 = (S_2, P, A, \pi_2, \delta_2, s_{02})$ があるとき，$(S_1 \cup S_2, Act, \delta_1 \cup \delta_2, P, \pi_1 \cup \pi_2)$ において，$s_{01} \approx_{\pi\tau\omega} s_{02}$ ならば，T_1 と T_2 は *πτω* 等価であると呼び，$T_1 \approx_{\pi\tau\omega} T_2$ と表す．

πτω 等価は観測等価に *τω-divergence* に関する条件と状態属性の等価性を加えたものであり，観測等価より強い (分別能力が高い) 等価性である．

また，divergence に関しては，Milner[6]，Stirling[10]，Walker[13] らの研究があるが，彼らの partial bisimulation preorder ($\trianglelefteq$) により定義できる等価性 ($T_1 \approx_{preorder} T_2 \overset{def}{=} T_1 \trianglelefteq T_2 \wedge T_2 \trianglelefteq T_1$) より，*πτω* 等価の方が強い等価性である．図 1 の例では，b を観測不可能にした場合，$T_a \approx T_b$ だが $T_a \not\approx_{\pi\tau\omega} T_b$ であり内部遷移によるループを区別できる．また，$T_a \approx_{preorder} T_c$ だが $T_a \not\approx_{\pi\tau\omega} T_c$ であり，1.で指摘した問題を起こさない．

2.3 遷移システムの合成

並行プログラムはいくつかの同期通信し合うプロセス (遷移システム) から合成される．この遷移システムの合成に関するオペレータ (composition, relabelling) を導入する．また，これらのオペレータに関して *πτω* 等価性が保存されることを示す．

〈合成 (composition)：$T_1 \mid T_2$〉

$P_1 \cap P_2 = \phi$ であるような遷移システム $T_1 = (S_1, P_1, A_1, \pi_1, \delta_1, s_{01})$ と $T_2 = (S_2, P_2, A_2, \pi_2, \delta_2, s_{02})$ から合成された遷移システム $T = T_1 \mid T_2$ は次のように定義される．

$T = (S_1 \times S_2, P_1 \cup P_2, A_1 \cup A_2, \pi, \delta, (s_{01}, s_{02}))$

ここで，$\pi : S_1 \times S_2 \to 2^{P_1 \cup P_2}$ は，

$\pi(s_1, s_2) = \pi_1(s_1) \cup \pi_2(s_2)$ for all $s_1 \in S_1$, $s_2 \in S_2$

であり，

$\delta : S_1 \times S_2 \times (A_1 \cup A_2 \cup \{\tau\}) \to 2^{S_1 \times S_2}$ は，

$$\delta((s_1, s_2), a) = \begin{cases} \{(s_1', s_2) \mid s_1' \in \delta_1(s_1, a)\} & \text{if} \quad a \in A_1 \wedge a \notin A_2 \\ \{(s_1, s_2') \mid s_2' \in \delta_2(s_2, a)\} & \text{if} \quad a \notin A_1 \wedge a \in A_2 \\ \{(s_1', s_2') \mid s_1' \in \delta_1(s_1, a), s_2' \in \delta_2(s_2, a)\} & \text{if} \quad a \in A_1 \wedge a \in A_2 \\ \{(s_1', s_2') \mid (s_1' \in \delta_1(s_1, \tau), s_2' = s_2) \vee (s_2' \in \delta_2(s_2, \tau), s_1' = s_1)\} & \text{if} \quad a = \tau \end{cases}$$

である．

$T_1 \mid T_2$ の直感的意味は，T_1 と T_2 を同じ名前の動作に関して同期をとりながら並行に実行させた場合の並行プログラムである．

〈ラベル名変更 (relabelling)：$T[f]$〉

$T = (S, P, A, \pi, \delta, s_0)$ をラベル名変更関数 $f = (f_A, f_P)$，$f_A : A \cup \{\tau\} \to 2^{A' \cup \{\tau\}}$ (但し，$f_A(\tau) = \{\tau\}$)，$f_P : P \to P' \cup \{true\}$ (ここで，$f_P(p) = true$ は状態属性 p を観測不可能にすることを意味する) で変換した遷移システム $T' = T[f]$ は次のように定義される．

$T' = (S, P', A', \pi', \delta', s_0)$

ここで，$\delta' : S \times (A' \cup \{\tau\}) \to 2^S$ は，$\delta'(s, a') = \{s' \mid s' \in \delta(s, a), a' \in f_A(a)\}$ であり，$\pi' : S \to 2^{P' \cup \{true\}}$ は，$\pi'(s) = f_P(\pi(s))$ である．直感的には，$f_A(a) = \{a'\}$，$f_P(p) = p'$ は単に a を a' に p を p' に名前変更することを意味し，$f_A(a) = \{a_1, a_2\}$ は a の遷移と同じものを一つ追加生成したのち，それぞれ a_1，a_2 と名前変更することを意味する．

遷移システム T の動作/状態属性名を関数 f に従って変更する．具体的には，同一のプロセスを複数個コ

ピーして同時に使用するようなケースにおいて，動作名/状態属性名の重複を回避するために名前変更を行う．また，遷移システム T の動作および状態属性を観測不可能にする場合にも用いる $(f_A(a)=\{\tau\},\ f_P(p)=true)$．

この relabelling 関数 f_A を次のように記すこともできる．

$[\{l_{11}', \cdots, l_{1k_1}'\}/l_1, \cdots, \{l_{n1}', \cdots, l_{nk_n}'\}/l_n]$

これは，$f_A(l_i)=\{l_{i1}', \cdots l_{ik_i}'\}$ for all $i\in\{1, \cdots, n\}$ なる関数を表すとする．f_P も同様である．

[定理 2](合成に関する $\pi\tau\omega$ 等価の保存)

$T_{11}\approx_{\pi\tau\omega}T_{12}$，$T_{21}\approx_{\pi\tau\omega}T_{22}$ ならば，$T_{11}\mid T_{21}\approx_{\pi\tau\omega}T_{12}\mid T_{22}$，$T_{11}[f]\approx_{\pi\tau\omega}T_{12}[f]$

(証明) *relabelling* に関しては自明．*composition* に関しても，文献(7)の定理 7.2 と同様に，$R=\{((s_{11}, s_{21}), (s_{12}, s_{22}))\mid s_{11}\approx_{\pi\tau\omega}s_{12}, s_{21}\approx_{\pi\tau\omega}s_{22}, s_{11}\in S_{11}, s_{12}\in S_{12}, s_{21}\in S_{21}, s_{22}\in S_{22}\}$ が $\pi\tau\omega$-*bisimulation* であることが容易に示せる． □

3. 検証項目記述言語

遷移システム T に対する検証項目記述言語 PQL (Process Query Language) を定義する．PQL の特徴は，

・状態と動作に関する制約を混在させて記述可能，

・観測不可能な内部遷移による無限ループを明示的に記述可能，

・正規表現を含む高度な記述能力

である．まず，τ 遷移を観測可能とした場合の様相論理 SPQL (Strong PQL) を定義し，SPQL のマクロ言語として τ 遷移を観測不可能にした PQL を定義する．

3.1 SPQL (Strong Process Query Logic)

SPQL は，不動点オペレータを用いて時相論理とプロセス論理を融合した様相論理である．

[定義 8](SPQL 式)

[構文]

P：状態属性の集合

A：動作集合

SPQL 式は，以下のように再帰的に定義される．ここで，自由状態論理変数とは μ オペレータで束縛されない状態論理変数である．

〈状態論理式〉

・状態論理変数 Z は状態論理式．

・$p\in P$ および $true$ は状態論理式．

・f_1，f_2 が状態論理式ならば，$f_1\wedge f_2$，$\neg f_1$ は状態論理式．

・f が状態論理式で，その中に現れる Z が自由状態論理変数で，各 Z にかかる $\neg$ のネストが偶数ならば，$\mu Z.f$ は状態論理式．

・g がパス論理式ならば，$\exists g$ は状態論理式．

〈パス論理式〉

・$a\in A$ はパス論理式

・g_1，g_2 がパス論理式ならば，$g_1\wedge g_2$，$\neg g_1$ はパス論理式

・f が状態論理式ならば，Xf，Tf はパス論理式

〈SPQL 式〉

・自由状態論理変数を含まない状態論理式は SPQL 式

[意味] 遷移システム $T=(S, P, A, \pi, \delta, s_0)$ に対する SPQL 式 f のモデル (f を満たす状態) の集合 $V[\![f]\!]$ を次のように定義する．ここで，f が状態論理式，g がパス論理式，SF が状態論理式の集合，PF がパス論理式の集合とする．また，f が成り立つ T の状態の集合を $V[\![f]\!]\subset S$，g が成り立つパスの集合を $R[\![g]\!]\subset S\times S$ と表す．

$V: SF\to 2^S$，すなわち，$V[\![f]\!]$ は自由状態論理変数を含まない状態論理式 f が成り立つ状態の集合であり，以下のように定義される．ここで，$(\lambda Z.f_1)f_2$ は f_1 に現れる自由状態論理変数 Z を状態論理式 f_2 で置き換えた状態論理式を表し，$[\![S']\!]^{-1}$ は，$V[\![f]\!]=S'$ なる架空の状態論理式 f を表す (例えば $[\![\phi]\!]^{-1}=false$, $[\![S]\!]^{-1}=true$ である)．

$V[\![p]\!]=\{s\in S\mid p\in\pi(s)\}$

$V[\![\exists g]\!]=\{s\in S\mid \exists(s, s')\in R[\![g]\!])\}$

$V[\![\neg f]\!]=S-V[\![f]\!]$

$V[\![f_1\wedge f_2]\!]=V[\![f_1]\!]\cap V[\![f_2]\!]$

$V[\![\mu Z.f]\!]=\cap\{S'\subseteq S\mid V[\![(\lambda Z.f)[\![S']\!]^{-1}]\!]\subseteq S'\}$

$R: PF\to 2^{S\times S}$，すなわち，$R[\![g]\!]$ はパス論理式 g が成り立つ長さ 1 のパス (エッジ) の集合であり，以下のように定義される．

$R[\![a]\!]=\{(s, s')\mid s\xrightarrow{a}s'\}$

$R[\![\neg g]\!]=(S\times S)-R[\![g]\!]$

$R[\![g_1\wedge g_2]\!]=R[\![g_1]\!]\cap R[\![g_2]\!]$

$R[\![Xf]\!]=\{(s, s')\mid \exists a.(s\xrightarrow{a}s'\wedge a\neq\tau\wedge s'\in V[\![f]\!])\}$

$R[\![Tf]\!]=\{(s, s')\mid \exists a.(s\xrightarrow{a}s'\wedge a=\tau\wedge s'\in V[\![f]\!])\}$

また，以下の便宜的オペレータを導入する．

・$false\overset{\text{def}}{=}\neg true$

・$f_1\vee f_2\overset{\text{def}}{=}\neg(\neg f_1\wedge\neg f_2)$

・$\nu Z.f\overset{\text{def}}{=}\neg\mu Z'.\neg(\lambda Z.f)(\neg Z')$

各オペレータの直感的意味を以下に示す．

$\wedge$ (論理積)，$\vee$ (論理和) $\neg$ (否定)，

Xf：τ 以外の遷移の直後に f が真，

Tf：τ 遷移の直後に f が真，

$\exists g$：あるパスで g が真，

$\mu Z.f$：μ は最小不動点オペレータである．自由状態論理 f に現れるすべての状態変数 Z に再帰的に $\mu Z.f$ がバインドしたときの最小不動点を表す．例えば，$\mu Z.(f \vee \exists XZ)$ の意味は「あるパスでいつかは f が成り立つ」である．

$\nu Z.f$：ν は最大不動点オペレータである．例えば，$\nu Z.(f \vee \exists XZ)$ の意味は「あるパスでいつかは f が成り立つ，または無限長パスがある（有限長パスで判定できない）」である．

この SPQL 式の最小/最大不動点オペレータの重要な性質を以下の補題にまとめる．ここで，$\lambda x.y$ に対して $(\lambda x.y)^1 z=(\lambda x,y)z$，$(\lambda x.y)^{k+1}z=(\lambda x.y)(\lambda x.y)^k z$ とする．

[補題 1]（単調性）

$S_1 \subset S_2 \Rightarrow V[\![(\lambda Z.f)[\![S_1]\!]^{-1}]\!] \subset V[\![(\lambda Z.f)[\![S_2]\!]^{-1}]\!]$

（証明） f において Z にかかる否定のネストは偶数だから SPQL 式の定義により単調性は自明． □

[補題 2]（最小/最大不動点オペレータの性質）

（1） $V[\![\mu Z.f]\!]=\lim_{k\to\infty}(\lambda S'.\ V[\![(\lambda Z.f)[\![S']\!]^{-1}]\!])^k\phi$
$=\lim_{k\to\infty} V[\![(\lambda Z.f)^k false]\!]$
$V[\![\nu Z.f]\!]=\lim_{k\to\infty}(\lambda S'.\ V[\![(\lambda Z.f)[\![S']\!]^{-1}]\!])^k S$
$=\lim_{k\to\infty} V[\![(\lambda Z.f)^k true]\!]$

（2） $s\in V[\![\mu Z.f]\!] \Leftrightarrow \exists k.\ \forall h\geq k.\ s\in V[\![(\lambda Z.f)^h false]\!]$

（3） 状態集合 S が有限の場合，μ を含む任意の f に対して，$V[\![f]\!]=V[\![f^*]\!]$ で，μ を含まない有限長の f^* が存在する．

（証明）（1） $S^k=(\lambda S'.\ V[\![(\lambda Z.f)[\![S']\!]^{-1}]\!])^k\phi=V[\![(\lambda Z.f)^k false]\!]$，$S^\omega=\lim_{k\to\infty}S^k$ とする．単調性により，$\forall S'\subset S.(V[\![(\lambda Z.f)[\![S']\!]^{-1}]\!]\subset S' \Rightarrow S^\omega\subset S')$．また，明らかに $V[\![(\lambda Z.f)[\![S^\omega]\!]^{-1}]\!]\subset S^\omega$ であるから，$V[\![\mu Z.f]\!]=\cap\{S'\subseteq S \mid V[\![(\lambda Z.f)[\![S']\!]^{-1}]\!]\subseteq S'\}=S^\omega$ である．$V[\![\nu Z.f]\!]$ についても同様．（2） $\phi\subset S^1\subset S^2\cdots\subset S^\omega$ より，$s\in S^\omega \Leftrightarrow \exists k.\ \forall h\geq k.\ s\in S^h \Leftrightarrow \exists k.\ \forall h\geq k.\ s\in V[\![(\lambda Z.f)^h fasle]\!]$．（3） 状態集合 S が有限の場合，各 $s\in S$ ごとの k の最大値をとれば $\exists k.(V[\![\mu Z.f]\!]=V[\![(\lambda Z.f)^k false]\!])$ である．故に，f の最も外殻の（自由状態変数を含まない）μ オペレータ部分論理式 $\mu Z.f_{sub}$ を $(\lambda Z.f_{sub})^k false$ で置き換えた論理式 f' で，$V[\![f]\!]=V[\![f']\!]$ を満たすものが存在する．この操作を μ オペレータがなくなるまで繰り返すことにより，$V[\![f]\!]=V[\![f^*]\!]$ で，μ オペレータを含まない有限長の論理式 f^* が構成できる． □

[定義 9]（モデル） 状態 $s\in S$ が $s\in V[\![f]\!]$ のとき，s を SPQL 式 f のモデルと呼び $s\models f$ と記す．同様に，遷移システム $T=(S,P,A,\pi,\delta,s_0)$ で $s_0\in V[\![f]\!]$ ならば，T は SPQL 式 f のモデルであると呼び $T\models f$ と記す．

[定理 3]（SPQL 式のモデル判定の決定可能性） 任意の遷移システム $T=(S,P,A,\pi,\delta,s_0)$ と SPQL 式 f に対して，S が有限集合ならば $T\models f$ が成り立つかどうかを機械的に判定するアルゴリズムが存在する．

（証明） $V[\![\mu Z.f]\!]$，$V[\![\nu Z.f]\!]$ を求めるアルゴリズムを示せばよい．補題 2（1）から，$[\![\mu Z.f]\!]=\lim_{k\to\infty}(\lambda S',V[\![(\lambda Z.f)[\![S']\!]^{-1}]\!])^k\phi$ であり，S が有限ならば有限の k で収束するので判定可能である．$V[\![\nu Z.f]\!]$ に関しても同様． □

実際の SPQL の効率的モデルの判定手続きは，CTL のモデル判定手続き[2]を拡張することで実現できる．

3.2 PQL (Process Query Language)

SPQL を用いて PQL を定義する．SPQL では τ 遷移が観測可能であり，Tf のように式の中に明示的に記述しなければならなかった．PQL は τ 遷移を観測不可能とした言語であり，式中に τ 遷移を明示的に記述できない．

[定義 10]（PQL 式）

[構文]

P：状態属性の集合

A：動作集合

・状態論理変数 Z は状態論理式

・$p\in P$ および $true$ は状態論理式

・f_1，f_2 が状態論理式のとき，$f_1\wedge f_2$，$\neg f_1$ は状態論理式

・f が状態論理式で $a\in A$ のとき，$\langle a\rangle^+ f$，$\langle a\rangle^- f$，$\langle . \rangle^+ f$，$\langle . \rangle^- f$，$\langle\rangle^+ f$，$\langle\rangle^- f$ は状態論理式

・f が状態論理式でその中に現れる Z が自由状態論理変数で f における各 Z にかかる否定のネストが偶数のとき，$\mu Z.f$ は状態論理式

・f が状態論理式でその中に自由状態論理変数を含まないとき，f は PQL 式

PQL 式の全体を L_{PQL} で表す．

[意味] PQL 式は以下のルールで SPQL 式に変換可能であり，SPQL 式として解釈される．

・$\langle a\rangle^- f \Leftrightarrow$
$\mu Z_1.(\exists((a\wedge X(\mu Z_2.(f\vee\exists TZ_2)))\vee TZ_1))$

・$\langle a\rangle^+ f \Leftrightarrow$

$\nu Z_1.(\exists((a\wedge X(\nu Z_2.(f\vee\exists TZ_2)))\vee TZ_1))$

・$\langle . \rangle^- f \Leftrightarrow$

$\mu Z_1.(\exists(X(\mu Z_2.(f\vee\exists TZ_2))\vee TZ_1))$

・$\langle . \rangle^+ f \Leftrightarrow$

$\nu Z_1.(\exists(X(\nu Z_2.(f\vee\exists TZ_2))\vee TZ_1))$

・$\langle\rangle^- f \Leftrightarrow \mu Z.(f\vee\exists TZ)$

・$\langle\rangle^+ f \Leftrightarrow \nu Z.(f\vee\exists TZ)$

・上記以外の PQL 式 f についてはそのまま

$\langle a\rangle^- f$ の直感的意味が「動作 a を実行したあとに f が成り立つ状態になり得る」のに対し，$\langle a\rangle^+ f$ の直感的意味は「動作 a を実行したあとに f が成り立つ状態になり得る，または τ 遷移の無限ループに陥り得る」である．後者は，τ 遷移の無限ループを「有限回の遷移では判定できない．判定できないものは真である」と解釈するわけである．また，$\langle . \rangle f$ は $\exists a\in A.\langle a\rangle f$ を意味し，$\langle\rangle f$ は $\langle\varepsilon\rangle f$ を意味する．

PQL の記述能力は CTL より強力 (CTL で記述可能なことは PQL で記述可能) であり，並行プログラムの動作順序に関するさまざまな検証項目の記述が可能である．しかし，PQL 式は必ずしも可読性がよいとは言えないため，便宜的に以下のマクロオペレータを導入する．apat，apet，epat，epet はそれぞれ CTL の時相オペレータ AG，AF，EG，EF に相当する．すなわち，a は「∀(for all)」，p は「path」，e は「∃(exists)」，t は「time」を表している．

・$f_1\vee f_2 \overset{\text{def}}{=} \neg(\neg f_1\wedge\neg f_2)$

・$\nu Z.f \overset{\text{def}}{=} \neg\mu Z'.\neg(\lambda Z.f)(\neg Z')$

・$[[a]]^+ f \overset{\text{def}}{=} \neg\langle a\rangle^-\neg f$

・$[[a]]^- f \overset{\text{def}}{=} \neg\langle a\rangle^+\neg f$

・$[[.]]^+ f \overset{\text{def}}{=} \neg\langle . \rangle^-\neg f$

・$[[.]]^- f \overset{\text{def}}{=} \neg\langle . \rangle^+\neg f$

・$[[]]^+ f \overset{\text{def}}{=} \neg\langle\rangle^-\neg f$

・$[[]]^- f \overset{\text{def}}{=} \neg\langle\rangle^+\neg f$

・$apat\ f \overset{\text{def}}{=} [[]]^+\nu Z.(f\wedge[[.]]^+ Z)$
(すべてのパスで常には f が成り立つ)

・$apet\ f \overset{\text{def}}{=} [[]]^+\mu Z.(f\vee(\langle . \rangle^- true\wedge[[.]]^+ Z))$ (すべてのパスでいくつかは f が成立り立つ)

・$epat\ f \overset{\text{def}}{=} \langle\rangle^-\nu Z.(f\wedge([[.]]^+ false\vee\langle . \rangle^- Z)$ (あるパスでいつも f が成り立つ)

・$epet\ f \overset{\text{def}}{=} \langle\rangle^-\mu Z.(f\vee\langle . \rangle^- Z)$
(あるパスでいつかは f が成り立つ)

・$external\_deadlock \overset{\text{def}}{=} [[.]]^+ false$
(観測不可能な内部ループが存在するかもしれないが観測上はデッドロック)

・$internal\_divergence \overset{\text{def}}{=} \langle\rangle^+ false$
(観測不可能な内部ループが存在する)

・$internal\_deadlock \overset{\text{def}}{=} [[.]]^+ false\wedge\neg\langle\rangle^+ false$ (観測不可能な内部ループが存在しない完全なデッドロック)

また，以下の例で示すように，正規表現と同等の動作の順序関係の記述も可能である．

[例 1] PQL 式による検証項目の記述例

$apat(etep\langle a\rangle^- true)$

意味：動作 a はデッドロックフリーである．

$\nu Z_1.([[b]]^+ false\wedge[[c]]^+ false\wedge[[a]]^+(\nu Z_2.([[a]]^+ false\wedge[[b]]^+ Z_2\wedge[[c]]^+ Z_1)))$

意味：動作 a，b，c の生起順序は正規表現 $((ab^*c)^*)$ で規定される．

PQL の重要な性質として，遷移システムの識別能力が $\pi\tau\omega$ 等価と同等であることが証明できる．すなわち，$\pi\tau\omega$ 等価な遷移システムは PQL で記述された任意の検証項目に対して判定結果が同じである．

[定義 11] ($\approx_{PQL}$)

$T_1\approx_{PQL} T_2 \overset{\text{def}}{=} \forall f\in L_{PQL}.(T_1\models f \text{ iff } T_2\models f)$

[定理 4] (PQL と $\pi\tau\omega$ 等価の関係) 任意の遷移システム T_1，T_2 に対して，

$T_1\approx_{\pi\tau\omega} T_2 \Leftrightarrow T_1\approx_{PQL} T_2$

(証明) 付録参照. □

[例 2] PQL 式による簡単な検証例

図 1 の b を観測不可能としたときの三つの遷移システムに対する三つの PQL 式の検証結果を表 1 に示す．ここで，YES は遷移システムが PQL 式のモデルであることを表し，NO はモデルでないことを表す．この結果は遷移システムを区別する PQL 式が存在することを意味するが，これは，$T_a\not\approx_{\pi\tau\omega} T_b$，$T_a\not\approx_{\pi\tau\omega} T_c$，$T_b\not\approx_{\pi\tau\omega} T_c$ を裏づけている．

4. 積重ね式検証法

定理 4 の結果を利用した並行プログラムの積重ね式検証法を示す．

表 1 PQL の簡単な検証例

PQL 式	T_a	T_b	T_c
$\ll a\gg^- \ll c\gg^- true$	YES	YES	YES
$[[a]]^- \ll c\gg^- true$	NO	YES	NO
$\ll a\gg^- \neg \ll c\gg^- true$	NO	NO	YES

4.1 検証スコープ

まず，積重ね式検証法に必要な「検証スコープ」を導入する．一般に，一つの検証項目は検証対象であるプログラムの部分的な側面についての記述であることが多い．そこで，各検証項目がプログラムのどの部分に注目して記述されたかを，明示的に記述することにする．これを検証スコープと呼ぶ．具体的には，遷移システム $T=(S,P,A,\pi,\delta,s_0)$ の検証スコープ VS は注目している状態属性集合 $P'\subset P$ と動作集合 $A'\subset A$ のペア $VS=(P',A')$ で表される．このとき，検証項目を (VS,f) で表す．f に現れる状態属性の集合 P_f および動作の集合 A_f は検証スコープに含まれなければならない．検証時には，検証スコープに現れない状態属性および動作はすべて観測不可能とみなし，それぞれ τ，*true* で置き換える．この VS に基づくラベル名変更関数を l_{VS} と表す．すなわち，$T\models(VS,f)$ は $T[l_{VS}]\models f$ と定義できる．この検証スコープは，検証項目の書きやすさのためのマクロ記法である．すなわち，検証項目記述者にとって，$(T[l_{VS}]\models f)$ と考えるよりも，VS の指定も検証項目記述の一部として $T\models(VS,f)$ と表現するのが自然である．例えば，検証項目 $f=\langle\cdot\rangle^{-}true$ においては，f には直接現れないが検証者の意図としては観測可能として認識している動作が存在するわけで，それを検証項目の一部として明記するのが自然である．

4.2 検証手順

積重ね式検証法とは，ローカルな検証の積み重ねとしてグローバルな検証を達成する方法である．具体的には，$T\models(VS_1,f_1)\wedge(VS_2,f_2)\wedge\cdots\wedge(VS_n,f_n)$ を検証する代わりに，$T_1\models f_1\wedge T_2\models f_2\wedge\cdots\wedge T_n\models f_n$ を検証することである．ここで，T_i は T を検証項目 f_i の検証スコープ VS_i に射影したもの，つまり検証スコープ外の部分を観測不可能とみなし，可能な限り縮約したものであり，$T[l_{VS_i}]\approx_{PQL}T_i$ を満たすものである．VS_i が局所的であればあるほど T_i も局所的になる．T_i を T の f_i への射影遷移システムと呼ぶ．

以下に，射影遷移システムの構成法を示す．ここでは，2個のプロセス（遷移システム）T_1 と T_2 から構成される並行プログラム $T=T_1\mid T_2$ の構成手順を示す．一般に n 個のプロセスから構成される場合も2個の場合の組合せとして定式化できる（$T=(\cdots((T_1\mid T_2)\mid T_3)\cdots\mid T_n)$）．検証項目は (VS,f) とする．ここで，直接 T を合成するとき，T の大きさのオーダは T_1 と T_2 の大きさの和ではなく積になり，これが状態数の爆発の原因であった．これを回避する射影遷移システム T_f の合成手順は，

$$T_f=red((red(T_1[l_1])\mid red(T_2[l_2]))[l_{VS}])$$

である．ここで，l_1，l_2 は，f の検証スコープと T_1 と T_2 の同期通信に無関係な部分を観測不可能にするラベル名変更関数である．すなわち，検証スコープに現れる動作と状態属性，各プロセスが外部と同期をとる動作，以外の動作と状態属性をすべて観測不可能な τ 遷移および *true* とする．また，$red(T)$ は縮約関数であり，$T'=red(T)$ は，$T\approx_{\pi\tau\omega}T'$ を満たし，T よりもサイズの小さい遷移システムである．縮約関数の具体例は次節で述べる．

合成された射影遷移システム T_f に対して定理3に基づきモデル判定を行う．このとき，$T[l_{VS}]\approx_{\pi\tau\omega}T_f$ であり，定理4より，f の検証結果は同じ（$T[l_{VS}]\approx_{PQL}T_f$）であることが保証される．本検証法では，合成の過程で縮約を繰り返すことによって途中過程で生成される遷移システムの最大瞬間状態数を抑制できるため，比較的大規模な並行プログラムの検証も可能になる．

本検証法を用いた自動検証システムの入出力は次のようになる．

[入力]

（1） 並行プログラム（遷移システムの合成式によって表される．合成に必要なラベル名関数も含む）

（2） 検証項目のリスト（各検証項目は検証スコープと PQL 式のペアで表される）

[出力]

（1）各検証項目に対する回答（YES/NO）

各射影遷移システムの構成過程における，縮約のためのラベル名変更および縮約，合成，および射影遷移システムの検証はシステムが全自動で行う．

4.3 縮約関数

縮約（reduction）とは，与えられた遷移システム T から，T と $\pi\tau\omega$ 等価でサイズの小さい遷移システム T' を生成することである．ここでは，二つの縮約方法を説明する．

（1） *πτω-bisimulation* による縮約

T において最大な *πτω-bisimulation* を求め R とする．R で関係づけられるノードをまとめて一つのノードにした遷移システムを $red(T)$ とする．このとき，$T'=red(T)$ は $T'\approx_{\pi\tau\omega}T$ を満たすもののうちで状態数が最小の遷移システムである．

（2） 書換えルールによる縮約

πτω-bisimulation による縮約では，状態数が最小の遷移システムが求まる．しかし，T が大きい場合の

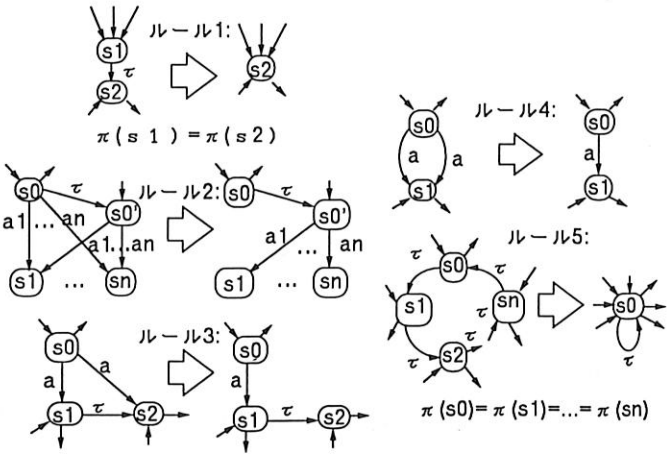

図3 縮約のための書換えルール
Fig. 3 Reduction rules.

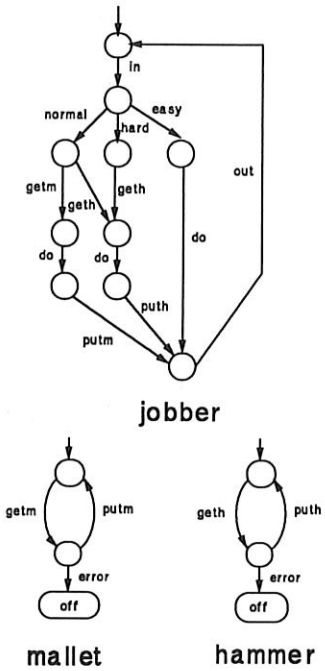

図4 例題：ジョブショップ
Fig. 4 Example : jobshop.

$\pi\tau\omega$-*bisimulation* の計算コストが膨大になる．そこで，τ 遷移に注目した遷移システムの書換えを行うヒューリスティックルール（図3）を用意し，そのルールを可能な限り適用して遷移システムの縮約を行う．ここで，各ヒューリスティックルールによる書換えは $\pi\tau\omega$ 等価性を保存するので縮約された遷移システム $T'=red(T)$ は $T'\approx_{\pi\tau\omega}T$ を満たす．$\pi\tau\omega$-*bisimulation* による縮約と比べると縮約率は小さいが，大規模な遷移システムに対しても高速な縮約が可能である．また，（2）の方法によってある程度小さくした後に（1）の方法で更に縮約する折衷法もある．

4.4 検証実験

4.4.1 ジョブショップ問題

Milnerの本[7]にある簡単な例で積重ね式検証の効果を確認する．検証したい並行プログラム *jobshop* は，四つのプロセス（*jobber*×2, *hammer*, *mallet*）から以下のような合成式によって与えられるとする．各プロセスの遷移システムは図4に示す．

$jobshop=(((jobber[l_{j1}] \mid hammer[l_h]) \mid mallet[l_m]) \mid jobber[l_{j2}])[l_{js}]$

ここで，

$l_{j1}=([\tau/easy, \tau/hard, \tau/normal, \tau/do, in1/in, out1/out, geth1/geth, puth1/puth, getm1/getm, putm1/putm], [])$.

$l_{j2}=([\tau/easy, \tau/hard, \tau/normal, \tau/do, in2/in, out2/out, geth2/geth, puth2/puth, getm2/getm, putm2/putm], [])$.

$l_h=([\tau/error, \{geth1, geth2\}/geth, \{puth1, puth2\}/puth], [h\_off/off])$.

$l_m=([\tau/error, \{getm1, getm2\}/getm, \{putm1, putm2\}/putm], [m\_off/off])$.

$l_{is}=([\tau/geth1, \tau/geth2, \tau/puth1, \tau/puth2, \tau/getm1, \tau/getm2, \tau/putm1, \tau/putm2, in/in1, in/in2, out/out1, out/out2], [])$.

この並行プログラムに対して，以下の検証項目が与えられたとする．

（1） デッドロックが起こり得るか？ $(([], []), epet\ external\_deadlock)$

（2） 観測不可能な内部ループはあるか？ $(([], []), epet\ internal\_divergence)$

（3） *in* が2回 out が2回のパターンはあるか？
$(([in, out], []), \langle in\rangle^{-}\langle in\rangle^{-}\langle out\rangle^{-}\langle out\rangle^{-}true)$

（4） mallet が使用不可能な状況でも hammer が使用可能ならばデッドロックは起こらない．

$(([], [m\_off, h\_off]),$
$apat((m\_off \wedge \neg\langle\rangle^{-}h\_off)$
$\supset \neg external\_deadlock))$

以上の検証項目に対して，検証システムは，

(第1段階) 各検証項目ごとに検証項目に関係のない部分を可能な限り縮約した射影遷移グラフを合成する．

例えば検証項目(3)に対しては，射影遷移グラフ($jobshop_{f_3}$)を以下の手順で合成する．この手順は元の合成手順と検証スコープから自動的に生成される．

$$jobshop_{f_3} = red(red((red((red((red(jobber[l_{j1}]) \mid red(hammer[l_{h'}]))[l_{j1h}]) \mid red(mallet[l_{m'}]))[l_{j1hm}]) \mid red(jobber[l_{j2}]))[l_{j1hmj2}])[l_{js'}])$$

ここで，

$l_{h'}=([\tau/error, \{geth1, geth2\}/geth, \{puth1, puth2\}/puth], [true/off])$.

$l_{m'}=([\tau/error, \{getm1, getm2\}/getm, \{putm1, putm2\}/putm], [true/off])$.

$l_{j1h}=([\tau/geth1, \tau/puth1], [])$.

$l_{j1hm}=([\tau/getm1, \tau/putm1], [])$.

$l_{j1hmj2}=([\tau/geth2, \tau/puth2, \tau/getm2, \tau/putm2], [])$.

$l_{js'}=([in/in1, in/in2, out/out1, out/out2], [])$.

l_{j1} と l_{j2} は変更なし．

縮約を行った場合($jobshop_{f_3}$)と縮約を行わない場合($jobshop$)の大きさ(状態数)および合成/縮約の途中過程で生成される遷移システムの状態数の最大値を表2に示す．$jobshop_{f_3}$における縮約法に関しては，前述の二つの縮約関数のそれぞれを用いた結果を示す．縮約を行うことによって検証サイズを約10分の1にすることができた．

(第2段階) 各検証項目 f に対して $jobshop_{f} \models f$ の判定を行う．各検証項目に対する判定結果を表3に示す．

4.4.2 加工装置制御プログラム

本検証法を用いて，中規模の加工装置制御プログラムのプロトタイプを検証した結果を示す．この装置は，16個のプロセスから構成される並行(マルチタスク)プログラムで制御される．各要素プロセスの大きさと全体構成(プロセス間の同期関係)をそれぞれ表4と図5に示す．各プロセスの状態数が小さいのは同期制御部分だけに注目しているためである．この並行プログラムに対してデッドロック(([],[]), *external_deadlock*)の検証を行ったところ，表5に示す効果を得た．実際には，制御プログラムの各プロセスはペトリネットベースのプロトタイピング言語 MENDEL[15] で記述され，そこから遷移システムが自動生成される．設計者は，プロトタイプの検証によって，詳細設計に入る前にタイミングに関する致命的な設計上のバグを検出できる．実際のシステムの動きを自然に記述しようとするとき，検証の観点からは冗長な動作，状態が多数混在するの

表2 縮約効果(jobshopの例)

縮約の有無	状態数	最大瞬間状態数
jobshop (縮約無)	164	164
$jobshop_{f_3}$ (縮約(1))	17	72
$jobshop_{f_3}$ (縮約(2))	90	110

表3 検証結果(jobshopの例)

検証項目	結果
(1)	YES
(2)	YES
(3)	YES
(4)	YES

表4 装置制御プログラム(各要素プロセスの状態数)

要素プロセス名(個数)	遷移システムの状態数
(p1) 移動用アーム(1)	6
(p2) 検査装置(1)	3
(p3) 第1加工装置(1)	5
(p4) 第2加工装置(2)	3
(p5) 第3加工装置(1)	5
(p6) 設置用アーム(2)	6
(p7) 取出用アーム(2)	6
(p8) 第1開閉器(2)	2
(p9) 第2開閉器(2)	3
(p10) 移動用コンベア(2)	3

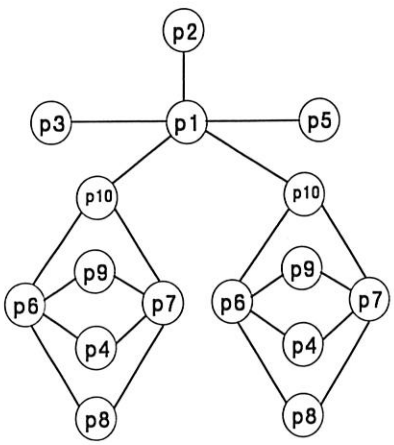

図5 装置制御プログラム(プロセス構成図)
Fig. 5 Machine Control Program (Process Structure).

表5 装置制御プログラム(縮約の効果)

縮約の有無	状態数	最大瞬間状態数
$machine$ (縮約無)	16741	16741
$machine_{red}$ (縮約(1))	4	1276
$machine_{red}$ (縮約(2))	1425	4218

で，縮約は非常に効果的である．

5. むすび

様相論理に基づく並行プログラム検証項目記述言語PQLとその積重ね式検証法を提案し，例題を用いてその有効性を確認した．本検証法を実際の検証に適用する場合，以下の課題が残っている．

(1) 一般の設計者にとってPQL式を記述することは必ずしも容易ではない．

(2) 検証結果が仕様と異なる場合，バグの存在はわかってもバグの具体的場所がわからない．

(3) 合成作業(composition. relabelling)が面倒である．

現在，これらの課題を解決する(1)PQL式生成インタフェース，(2)デバッガ，および(3)合成作業の支援機能を備えた検証システム「検太郎」を開発中である．

謝辞 本研究の機会を与えて下さいました東芝システム・ソフトウェア技術研究所の西島誠一所長，河野毅部長，大筆豊部長に感謝致します．また，本研究を含め日ごろから御指導頂いている同研究所の本位田真一博士に感謝致します．また，本システムを装置制御プログラムなどの実際問題へ適用するにあたって，同社生産技術研究所の隅田敏氏との議論が有益でありました．

文献

(1) ed. Baeten J. : "Applications of Process Algebra", Cambridge Univ. Press (1990).

(2) Clarke E. M., Emerson E. A. and Sistla A. P. : "Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications", ACM TOPLAS, 8, 2 (1986).

(3) Clarke E. M., Long E. E. and McMillan K. L. : "Compositional Model Checking", Proc, 4th Logic in Computer Science (1989).

(4) Hennessy H. and Milner R. : "Algebraic Laws for Nondeterminism and Concurrency", J. ACM, **32**, 1 (1985).

(5) Hennessy H. and Stirling C : "The Power of the Future Perfect in Program Logics", Information and Control, **67** (1985).

(6) Milner R. A : "Modal Characterization of Observable Machine-behaviour", LNCS, **112** (1981).

(7) Milner R : "Communication and Concurrency", Prentice Hall (1989).

(8) Mishra B. and Clarke E. M : "Hierarchical Verification of Asynchronous Circuit Using Temporal Logic", TCS, **38** (1985).

(9) ed. Sifakis J. : "Automatic Verification Methods for Finite State Systems", Springer-Verlag LNCS 407 (1989).

(10) Stirling C : "Modal Logic for Communicating Systems", TCS, **49** (1987).

(11) Stirling C : "Temporal Logics for CCS", LNCS, **354** (1989).

(12) Stirling C. and Walker D : "CCS, Liveness, and Local Model Cheking in Linear Time Mu-calculus", LNCS, **407** (1989).

(13) Walker D. J : "Bisimulation and Divergence", Information and Computation, **85** (1990).

(14) 内平直志，西村一彦，隅田 敏，川田秀司："時間論理による並行ロボット制御プログラムの検証とデバッグ"，第3回人工知能学会全国大会(1989).

(15) 内平直志，本位田真一："知的分散システム上のペトリネットに基づく並行プログラミング言語"，信学技報，**CPSY89-34** (1989).

付録

定理4の証明

定理を証明するためにいくつかの定義，補題を用意する．

[定義12] $(\approx_{\pi\tau\omega^k})$ (S, Act, δ, P, π) において，任意の $s,\ t \in S,\ k \geq 0$ に対して

$s \approx_{\pi\tau\omega^0} t \overset{\text{def}}{=}$
- $\pi(s)=\pi(t)$
- $s\uparrow$ iff $t\uparrow$

$s \approx_{\pi\tau\omega^{k+1}} t \overset{\text{def}}{=}$
- $\pi(s)=\pi(t)$
- $\forall a.\ \forall s'.$(if $s \xrightarrow{a} s'$ then $\exists t'.\ t \overset{\hat{a}}{\Rightarrow} t' \wedge s' \approx_{\pi\tau\omega^k} t'$)
- $\forall a.\ \exists t'.$(if $t \xrightarrow{a} t'$ then $\exists s'.\ s \overset{\hat{a}}{\Rightarrow} s' \wedge s' \approx_{\pi\tau\omega^k} t'$)
- $s\uparrow$ iff $t\uparrow$

[補題3] $(\approx_{\pi\tau\omega}$ と $\approx_{\pi\tau\omega^k}$ の関係)

(1) $s \approx_{\pi\tau\omega} t \Leftrightarrow \bigwedge_{k=1}^{\infty} (s \approx_{\pi\tau\omega^k} t)$

(2) $s \not\approx_{\pi\tau\omega} t \Leftrightarrow \bigvee_{k=1}^{\infty} (s \not\approx_{\pi\tau\omega^k} t) \Leftrightarrow \exists k.\ (s \not\approx_{\pi\tau\omega^k} t)$

(証明) 定義より明らか． □

[補題4] $(\langle\rangle^- f,\ \langle\rangle^+ f,\ \langle a\rangle^- f,\ \langle a\rangle^+ f$ の意味)

(1)
$s\models\langle\rangle^- f \Leftrightarrow \exists t=\tau^*.\exists s'.(s\xrightarrow{t}s'\wedge s'\models f)$
(2)
$s\models\langle\rangle^+ f \Leftrightarrow \exists t=\tau^*.\exists s'.(s\xrightarrow{t}s'\wedge s'\models f)$
$\vee\exists t=\tau^\omega.(s\xrightarrow{t})$
(3)
$s\models\langle a\rangle^- f \Leftrightarrow \exists t=\tau^* a\tau^*.\exists s'.(s\xrightarrow{t}s'\wedge s'\models f)$
(4)
$s\models\langle a\rangle^+ f \Leftrightarrow \exists t=\tau^* a\tau^*.\exists s'.(s\xrightarrow{t}s'\wedge s'\models f)$
$\vee\exists t=\tau^* a\tau^\omega.(s\xrightarrow{t})$
$\vee\exists t=\tau^\omega.(s\xrightarrow{t})$

ここで，$\exists t=\tau^* a\tau^*$ は，$\exists t\in\{\tau^i a\tau^j \mid i,j\geq 0\}$ を意味する．

(証明) (1)補題2より，$s\in V[\![\langle\rangle^- f]\!]=V[\![\mu Z.(f\vee\exists TZ)]\!]\Leftrightarrow\exists k.s\in V[\![(\lambda Z.(f\vee\exists TZ))^k false]\!]=V[\![(\exists T)^{k-1}f]\!]\Leftrightarrow\exists t=\tau^*.(s\xrightarrow{t}s'\wedge s'\in V[\![f]\!])$.

(2) 補題2より，$V[\![\langle\rangle^+ f]\!]=V[\![\nu Z.(f\vee\exists TZ)]\!]=\lim_{k\to\infty}V[\![\lambda Z.(f\vee\exists TZ)^k true]\!]=\lim_{k\to\infty}\bigcup_{i=0}^{k-1}V[\![(\exists T)^i f]\!]\cup V[\![(\exists T)^k true]\!]$ によって，$s\in V[\![\langle\rangle^+ f]\!]\Leftrightarrow\exists i.(s\in V[\![(\exists T)^i f]\!])\vee s\in V[\![(\exists T)^\omega true]\!]\Leftrightarrow\exists t=\tau^*.(s\xrightarrow{t}s'\wedge s'\in V[\![f]\!])\vee\exists t=\tau^\omega.(s\xrightarrow{t})$．(3) (1)と同様に，$s\in V[\![\langle a\rangle^- f]\!]=V[\![\mu Z_1.(\exists(a\wedge X(\mu Z_2.(f\vee\exists TZ_2)))\vee TZ_1)]\!]\Leftrightarrow\exists i,j.s\in V[\![(\exists T)^i\exists(a\wedge X(\exists T)^j f)]\!]$．故に，$s\in V[\![\langle a\rangle^- f]\!]\Leftrightarrow\exists t=\tau^* a\tau^*.(s\xrightarrow{t}s'\wedge s'\in V[\![f]\!])$.

(4) (2)と同様に，$V[\![\nu Z.(f\vee\exists TZ)]\!]=\lim_{k\to\infty}(\bigcup_{i=0}^{k-1}V[\![(\exists T)^i f]\!]\cup V[\![(\exists T)^k true]\!])$．よって，$s\in V[\![\langle a\rangle^+ f]\!]=V[\![(\nu Z_1.(\exists(a\wedge X(\nu Z_2.(f\vee\exists TZ_2))\vee TZ_1)]\!]\Leftrightarrow\exists i,j.(s\in V[\![(\exists T)^i\exists(a\wedge X(\exists T)^j f)]\!])\vee\exists i.(s\in V[\![(\exists T)^i\exists(a\wedge X(\exists T)^\omega true)]\!])\vee s\in V[\![(\exists T)^\omega true]\!]$．故に，$s\in V[\![\langle a\rangle^+ f]\!]\Leftrightarrow\exists t=\tau^i a\tau^j.(s\xrightarrow{t}s'\wedge s'\in V[\![f]\!])\vee\exists t=\tau^i a\tau^\omega.(s\xrightarrow{t})\vee\exists t=\tau^\omega.(s\xrightarrow{t})$. □

(定理4の証明)

[$T_1\approx_{\pi\tau\omega}T_2\Leftarrow T_1\approx_{PQL}T_2$ の証明]

$T_1\not\approx_{\pi\tau\omega}T_2$ ならば，$\exists f.(T_1\models f\wedge T_2\not\models f)$ を示せばよい．このとき，補題3より $\exists k.(s_{01}\not\approx_{\pi\tau\omega^k}s_{02})$ だから，k に関する帰納法で解く．$s\not\approx_{\pi\tau\omega^k}t$ の場合，可能な四つのケースに関してそれぞれ $s\models f\wedge t\not\models f$ なる f を構成できる．

(ケース1) $\pi(s)\neq\pi(t)\Rightarrow f=p\,s.t.p\in\pi(s).p\notin\pi(t)$.

(ケース2) $s\uparrow\wedge\neg(t\uparrow)\Rightarrow f=\langle\rangle^+ false$. (補題4より)

(ケース3) $s\xrightarrow{a}s'\wedge\neg\exists t'.(t\xRightarrow{\hat{a}}t')\Rightarrow f=\langle a\rangle^- true$.(定義より)

(ケース4) $s\xrightarrow{a}s'\wedge\forall t_i'.(t\xRightarrow{\hat{a}}t_i'\Rightarrow s'\not\approx_{\pi\tau\omega^{k-1}}t_i')$ このとき，帰納法により，$\forall t_i'.\exists f_i.(s'\models f_i\wedge t_i'\not\models f_i)$ である．そこで，$f=\langle a\rangle^-\bigwedge_i f_i$.

[$T_1\approx_{\pi\tau\omega}T_2\Rightarrow T_1\approx_{PQL}T_2$ の証明]

$\forall f.(T_1\approx_{\pi\tau\omega}T_2\wedge T_1\models f\Rightarrow T_2\models f)$ を示す．ここで，補題2(3)より，μ オペレータを含まない有限長の PQL 論理式だけを考えればよい．すなわち，μ オペレータを含まない PQL 論理式に関する構造帰納法で証明できる．ここでは，$f=\langle a\rangle^+ f'$，$f=\langle a\rangle^- f'$ の場合のみを証明する．他の論理式に関しても，自明あるいは同様に証明できる．

$f=\langle a\rangle^- f'$：$s_1\approx_{\pi\tau\omega}s_2, s_1\models\langle a\rangle^- f'$ のとき，補題4より，$\exists t_1=\tau^* a\tau^*.(s_1\xrightarrow{t_1}s_1'\wedge s_1'\models f)$，$s_1\approx_{\pi\tau\omega}s_2$ より $\exists t_2=\tau^* a\tau^*.(s_2\xrightarrow{t_2}s_2'\wedge s_1'\approx_{\pi\tau\omega}s_2')$．構造帰納法により，$s_2'\models f'$，よって，$s_2\models\langle a\rangle^- f'$

$f=\langle a\rangle^+ f'$：$s_1\approx_{\pi\tau\omega}s_2$，$s_1\models\langle a\rangle^+ f'$ のとき，補題4より，(1)$\exists t=\tau^* a\tau^*.(s_1\xrightarrow{t}s_1'\wedge s_1'\models f)$，または(2)$\exists t=\tau^\omega.(s\xrightarrow{t})$，または(3)$\exists t=\tau^* a\tau^\omega.(s\xrightarrow{t})$．(1)の場合は，$f=\langle a\rangle^- f'$ と同様．(2)の場合は，$s_1\uparrow$ であり，$s_1\approx_{\pi\tau\omega}s_2$ より $s_2\uparrow$，よって，$s_2\models\langle a\rangle^+ f'$．(3)の場合は，$\exists t_1=\tau^* a\tau^*.(s_1\xrightarrow{t_1}s_1'\wedge s_1'\uparrow)$，$s_1\approx_{\pi\tau\omega}s_2$ より $\exists t_2=\tau^* a\tau^*.(s_2\xrightarrow{t_2}s_2'\wedge s_2'\uparrow)$．よって，$s_2\models\langle a\rangle^+ f'$ □

(平成3年3月1日受付，7月19日再受付)

内平　直志

昭57東工大・情報科学卒．同年(株)東芝入社．現在同社システム・ソフトウェア技術研究所勤務．時相論理，ペトリネット，CCSを用いた並行プログラムの合成/検証の研究開発に従事．昭61年度情報処理学会論文賞受賞．情報処理学会，ソフトウェア科学会各会員．