時間付き π 計算によるリアルタイムオブジェクト指向言語の形式的記述

桑 原 寛 明[†]　結 縁 祥 治[†]　阿 草 清 滋[†]

本稿では実時間システムの実装に用いられるリアルタイムオブジェクト指向言語に対し，時間に関して拡張した π 計算による記述への変換規則を与える．π 計算を時間経過とタイムアウトを表現できるよう拡張することで実時間性を直接的に記述できる．実時間システムはその性質により動作テストだけで正しく動作することを確認することは難しい．実時間システムを構成するソフトウェアプログラムを変換規則に基づき π 計算による記述に変換することで，システムの動作の解析や検証を形式的に行うことができる．

A Formal Description of a Real-Time Object-Oriented Language by the π-Calculus with Time

Hiroaki Kuwabara,[†] Shoji Yuen[†] and Kiyoshi Agusa [†]

In this paper, we present a collection of translation rules that map the syntactic constructions of a real-time object-oriented programming language which is used to implement real-time systems into expressions in a time-extended π-calculus. A time-extended π-calculus enables us to describe time progression and timeout. It is difficult to check that real-time systems work correctly by testing. We can analyze and verify the behavior of real-time systems formally by translating software programs of real-time systems into π-calculus expressions.

1. はじめに

現在実時間システムが至るところで利用されている．携帯電話や交通管理システム，ブレーキングシステムなどその応用範囲は非常に広い．ペースメーカーのように誤動作が人命にかかわるようなシステムもあり，そのようなシステムには非常に高い信頼性が要求される．

実時間システムは動作に時間制約を持つ並行システムであり[1)]，複数のプロセスから構成される．プロセスは並行に動作し，相互にメッセージ通信を行うことでシステムの動作を決定する．一方，オブジェクト指向では，システムを相互作用により計算を行う複数のオブジェクトの集合であるとしてモデル化する．相互に通信を行う単位として一つのプロセスを一つのオブジェクトとみなすことでオブジェクト指向開発が適用できる．

実時間システムは与えられた制約を満たすことが要求される．システムの動作の正しさは，結果の正しさだけではなく結果が得られるまでに経過した時間の長さにも依存する．動作の実行時間は実行時のオブジェクト生成やメッセージ通信，並行性など動的な要因に影響される．そのため実装後に実際に動作させてテストを行うだけではシステムが常に正しく動作することを確認することは難しい．実時間性や並行性により非決定的な動作が存在し動作系列は一通りではない．存在しうるすべての動作系列においてテストを行い，動作が正常であること保証することは困難である．実時間システムが正常に動作することを確認するためには，システムの時間を含む動作を定式化し，形式的に検証することが必要である．

本論文では，実時間システムの動作を制御するソフトウェアの実装に着目し，リアルタイムオブジェクト指向言語から π 計算による記述への変換規則を定義する．プログラムはオブジェクト指向言語を使って作成する．多くの場合，言語の構文は BNF を利用して定義されている．一方，意味は自然言語で表現されている．そのため解析対象を単にソースプログラムとするだけでは動作や時間に関する性質を調べることは困難である．変換規則を用いて実装段階で作成されたプログラムから π 計算による記述を得て，その記述に対し

† 名古屋大学大学院情報科学研究科情報システム学専攻
Dept. of Information Engineering, Graduate School of Information Science, Nagoya University

て形式的手法を用いて解析や検証を行う．

π 計算[2),4)] はプロセス代数の一種で並行計算モデルである．プロセス間通信に利用するリンク自身をメッセージとする通信や，プロセスの動的生成を表現することが可能であり，高い表現能力を持っている．そのためオブジェクト指向におけるクラスのインスタンス生成やメッセージ通信の表現に適している．しかし，π 計算には時間の概念がなく実時間性を直接的に表現することは難しい．そこで π 計算を時間に関して拡張し，時間拡張した π 計算を用いて変換規則を与える．時間拡張した π 計算の性質を利用してプログラムから変換して得られた記述を解析する．

本論文の構成は以下の通りである．2 章で π 計算に対する時間拡張について述べる．3 章で簡単なリアルタイムオブジェクト指向言語 OOL_{RT} を示し，4 章で OOL_{RT} から π 計算による記述への変換規則を与える．5 章で OOL_{RT} によるプログラムとその変換について簡単な具体例を挙げる．最後にまとめと今後の課題を述べる．

2. π 計算に対する時間拡張

π 計算[3)] は並行に動作するプロセスの振る舞いと，プロセス間の相互作用を記述する．π 計算ではプロセス間通信に用いるリンクを名前と呼び，名前が π 計算の最も基本的な要素である．名前自身をメッセージとして送受信することにより計算を表す．本節では，時間遷移を表すための構文と動作意味を与える．

2.1 構　　文

Milner の π 計算に時間経過アクション t を導入する．タイムアウトまでの時間を $[n]$ と表し，n 単位時間でタイムアウトする時間待ちを $t[n]$ と記述する．まずアクションガードつきプロセスを以下のように定義する．

定義 **1**　$\pi ::= x(\widetilde{y}) \mid \overline{x}\langle\widetilde{y}\rangle \mid \tau$ とする時

- 任意のプロセス P に対し $\pi.P$ はアクションガードつきプロセスである
- P_1, P_2 がアクションガードつきプロセスであるとすると

  $$P_1|P_2,\ P_1+P_2,\ \boldsymbol{\nu} a\, P_1,\ !\,P_1$$

  はアクションガードつきプロセスである　□

ここで $\mathcal{N}$ を名前の集合とし，$\overline{N} = \{\overline{x} | x \in \mathcal{N}\}$，$\mathcal{L} = \mathcal{N} \cup \overline{\mathcal{N}}$，$a, x \in \mathcal{N}$ とする．$\widetilde{y}$ は名前の並びを表す．また π 計算のプロセス式全体の集合を $\mathcal{P}$ と書き，アクションの集合を $Act = \mathcal{L} \cup \{\tau\}$ と記す．プロセス式を以下のように定義する．

定義 **2**　時間拡張した π 計算のプロセス式 P は以下の構文によって定義される．

$$\pi ::= x(\widetilde{y}) \mid \overline{x}\langle\widetilde{y}\rangle \mid \tau \mid t[n]$$
$$P ::= M \mid P_1|P_2 \mid \boldsymbol{\nu} a\, P \mid !\,P$$
$$M ::= \mathbf{0} \mid \pi.P \mid M_1 + M_2$$

ただし $!P$ はアクションガードつきプロセスでなければならない．　□

プレフィックスが $t[n]$ であるプロセスは n 単位時間待機する．例えばプロセス $t[5].P$ は 5 単位時間後にプロセス P へ遷移する．時間経過アクションと非決定的選択を用いることで，イベントの発生を一定時間待機しイベントが発生したか否かで動作が異なるプロセスを記述することができる．$a.P + t[5].Q$ というプロセスはアクション a の発生まで 5 単位時間待機する．5 単位時間経過する前に a が発生すればプロセス P へ，発生しなければタイムアウトしプロセス Q へ遷移する．

他のプレフィックスと演算子の意味を以下に示す．

- $x(y)$：x を通して $\widetilde{y}$ を受信
- $\overline{x}\langle\widetilde{y}\rangle$：$x$ を通して $\widetilde{y}$ を送信
- τ：外部からは不可視な内部アクション
- $P+Q$：プロセス P, Q のうち実行可能なプロセスを選択し実行する．いずれのプロセスも実行可能であれば一方を非決定的に選択する
- $P|Q$：プロセス P, Q を並行に動作させる
- $\boldsymbol{\nu} a\, P$：プロセス P 中の自由な変数 a を束縛する
- $!P$：無限回の繰り返しを表す．無限個のプロセス P が $|$ 演算子によって結合しているとみなす

2.2 時間動作意味

従来の π 計算の動作意味を時間に関して拡張する．この拡張では時間は離散時間であるとする．時間の経過は時間経過規則に基づく遷移によってのみ発生し，他の規則による遷移では時間は経過しないとする．

$\mathcal{P}$ 上の遷移関係 $\{\xrightarrow{\alpha} \mid \alpha \in Act\} \cup \rightsquigarrow$ は図 **1** の遷移規則及び図 **2** の時間経過規則によって定義される．プレフィックス $t[0]$ は τ と同等であるとみなし，タイムアウトは τ 遷移であるとする．

ラベル付き遷移は

- プロセスが他のプロセスとどのように通信するか
- プロセスが単位時間ごとにどのように遷移するか

を示す．$P \xrightarrow{\alpha} P'$ は入力，出力，内部アクションのいずれかのアクション α により P から P' に遷移することを表し，$P \rightsquigarrow P'$ は P が 1 単位時間の経過により P' に遷移することを表す．

時間はすべてのプロセスにおいて同時に経過する．時間経過によりプレフィックス $t[n]$ は $t[n-1]$ となり，タイムアウトまでの時間が 1 単位時間短くなったこと

$$\text{ACT}: \frac{}{\alpha.P \xrightarrow{\alpha} P}$$

$$\text{SUM-L}: \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'} \qquad \text{SUM-R}: \frac{Q \xrightarrow{\alpha} Q'}{P+Q \xrightarrow{\alpha} Q'}$$

$$\text{PAR-L}: \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \text{PAR-R}: \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$$

$$\text{COMM-L}: \frac{P \xrightarrow{\overline{x}\langle y\rangle} P' \quad Q \xrightarrow{x(y)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad \text{COMM-R}: \frac{P \xrightarrow{x(y)} P' \quad Q \xrightarrow{\overline{x}\langle y\rangle} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\text{RES}: \frac{P \xrightarrow{\alpha} P'}{\boldsymbol{\nu} x\, P \xrightarrow{\alpha} \boldsymbol{\nu} x\, P'} \text{ if } \alpha \notin \{x, \overline{x}\} \qquad \text{REP}: \frac{P \mid !P \xrightarrow{\alpha} A}{!P \xrightarrow{\alpha} A}$$

$$\text{TIMEOUT-L}: \frac{}{t[0].P + M \xrightarrow{\tau} P}$$

$$\text{TIMEOUT-R}: \frac{}{M + t[0].P \xrightarrow{\tau} P}$$

図 1 遷移規則
Fig. 1 The transition rules

$$\text{PASS}_\text{T}: \frac{}{t[n].P \rightsquigarrow t[n-1].P} \text{ if } n > 0$$

$$\text{OUT}_\text{T}: \frac{}{\overline{x}\langle y\rangle.P \rightsquigarrow \overline{x}\langle y\rangle.P} \qquad \text{IN}_\text{T}: \frac{}{x(y).P \rightsquigarrow x(y).P}$$

$$\text{SUM}_\text{T}: \frac{P \rightsquigarrow P' \quad Q \rightsquigarrow Q'}{P+Q \rightsquigarrow P'+Q'} \qquad \text{PAR}_\text{T}: \frac{P \rightsquigarrow P' \quad Q \rightsquigarrow Q'}{P \mid Q \rightsquigarrow P' \mid Q'} \text{ if } P \mid Q \not\xrightarrow{\tau}$$

$$\text{RES}_\text{T}: \frac{P \rightsquigarrow P'}{\boldsymbol{\nu} x\, P \rightsquigarrow \boldsymbol{\nu} x\, P'} \qquad \text{REP}_\text{T}: \frac{P \rightsquigarrow P'}{!P \rightsquigarrow !P'} \text{ if } P \mid P \not\xrightarrow{\tau}$$

$$\text{STRUCT}_\text{T}: \frac{P \rightsquigarrow P'}{Q \rightsquigarrow Q'} \text{ if } P \equiv Q,\ P' \equiv Q'$$

図 2 時間経過規則
Fig. 2 The rules for time progression

を示す．プレフィックスでない $t[n]$ の n の値は変化しない．PAR_T 規則と REP_T 規則において遷移は τ 遷移が発生しない場合にのみ発生する．

2.3 時間的性質

時間拡張した π 計算が時間決定性及び最大進行性を満たすことを示す．

定理 1 (時間決定性) プロセス P に対し，$P \rightsquigarrow P'$ かつ $P \rightsquigarrow P''$ ならば $P' = P''$ である．

証明： P の構造に関する帰納法による．

定理 2 (最大進行性) プロセス P に対し，$P \xrightarrow{\tau}$ ならば $P \not\rightsquigarrow$ である．

証明： P の構造に関する帰納法による．

3. リアルタイムオブジェクト指向言語 OOL_{RT}

OOL_{RT} は簡単なオブジェクト指向言語 OOL[4] からデータ型の取り扱いを省き，時間待ちとタイムアウトのための構文を追加した言語である．OOL_{RT} ではメンバ変数への直接的なアクセスは不可能でありカプセル化を実現している．

OOL_{RT} の構文を図 3 に示す．ここで P はプログラム名，E は式，A はクラス名，M はメソッド名，S は文，X は変数名を表す．

$$\begin{aligned}
Pdec &::= \text{program } P \text{ is } Cdec_1, \ldots, Cdec_n \\
&\quad\ \text{with } E \\
Cdec &::= \text{class } A \text{ is } Vdecs,\ Mdecs \\
Mdecs &::= Mdec_1, \ldots, Mdec_m \\
Mdec &::= \text{method } M(X_1, \ldots, X_l) \text{ is } S \\
Vdecs &::= \text{var } X_1, \ldots, X_k \\
S &::= X := E \\
&\mid\ E \\
&\mid\ S_1; S_2 \\
&\mid\ \text{wait } E \\
&\mid\ \text{return } E \\
&\mid\ \text{if } E \text{ then } S_1 \text{ else } S_2 \text{ endif} \\
&\mid\ \text{while } E \text{ do } S \text{ done} \\
&\mid\ \text{within } E \text{ do } S_1 \text{ timeout } S_2 \text{ done} \\
E &::= \text{true} \mid \text{false} \mid \text{nil} \\
&\mid\ Numeral \\
&\mid\ \text{new}(A) \\
&\mid\ X \\
&\mid\ E!M(E_1, \ldots, E_l)
\end{aligned}$$

図 3 OOL_{RT} の構文
Fig. 3 The syntax of OOL_{RT}

プログラム プログラムはクラスの定義 $Cdec_i$ とプログラム開始時に実行される式 E からなる

クラス クラス定義 $Cdec$ はクラス名 A とメンバ変数の宣言 $Vdecs$，メソッドの定義 $Mdecs$ からなる

メソッド メソッド定義 $Mdec$ はメソッド名 M と引数 X_i，メソッド本体 S からなる

時間待ち 実行を一時的に停止する．wait E が実行されると長さ E だけ実行を停止しその後再開する

タイムアウト 時間制約のある処理を

within E do S_1 timeout S_2 done

と記述する．処理 S_1 が E で指定された時間内に終了しなければ処理 S_2 の実行を開始する

インスタンス生成 new(A) によりクラス A のインスタンスを生成する

4. OOL_{RT} から π 計算への変換規則

OOL_{RT} の構文要素それぞれに対し π 計算による記述への変換規則を与える．時間に関する要素があるため時間拡張した π 計算を用いる．構文要素 A に対応する変換規則を $[\![A]\!]$ と表す．

プログラム $Pdec$ に対する π 計算記述への変換規則を図 4 に示す．$\widetilde{k}$ はプログラム中で定義される各クラスの名前を示す．Boolean はブール値を表すクラスでありプログラムには標準で組み込まれ

$$
\begin{aligned}
[\![Pdec]\!] &\equiv (\boldsymbol{\nu}\widetilde{k})([\![Cdec_1]\!](\widetilde{k}) \mid \cdots \mid [\![Cdec_n]\!](\widetilde{k}) \mid \mathrm{Boolean}(k_{Bool}) \mid (\boldsymbol{\nu}l)[\![E]\!](l, \mathsf{nil}, \widetilde{k})) \\
[\![Cdec]\!](\widetilde{k}) &\equiv !\,(\boldsymbol{\nu}a)(k(l).\overline{l}\langle a\rangle.(\boldsymbol{\nu}\widetilde{x}, \widetilde{m})([\![Vdecs]\!](\widetilde{x}, \widetilde{c} \cup \{a\}) \mid [\![Mdecs]\!](\widetilde{m}, a, \widetilde{x}, \widetilde{k}))) \\
[\![Vdecs]\!](\widetilde{x}, \widetilde{c}) &\equiv [\![\mathsf{var}\ X_1]\!](x_1, c_1) \mid \cdots \mid [\![\mathsf{var}\ X_k]\!](x_k, c_k) \\
[\![\mathsf{var}\ X]\!](x, c) &\equiv (\boldsymbol{\nu}l)(x(r,u).(\overline{r}\langle c\rangle.\overline{l}\langle c\rangle \mid u(c').\overline{l}\langle c'\rangle) \mid !\,l(c).x(r,u).(\overline{r}\langle c\rangle.\overline{l}\langle c\rangle \mid u(c').\overline{l}\langle c'\rangle))
\end{aligned}
$$

図 4　プログラム，クラス定義，変数宣言に対する π 計算記述への変換規則
Fig. 4　Translation rules for $Pdec$, $Cdec$ and $Vdecs$

$$
\begin{aligned}
[\![Mdecs]\!](\widetilde{m}, a, \widetilde{x}, \widetilde{k}) &\equiv !\,a(n).\overline{n}\langle\widetilde{m}\rangle.([\![Mdec_1]\!](m_1, \widetilde{x}, \widetilde{k}) \mid \cdots \mid [\![Mdec_m]\!](m_n, \widetilde{x}, \widetilde{k})) \\
[\![Mdec]\!](m, \widetilde{x}, \widetilde{k}) &\equiv (\boldsymbol{\nu}g, \widetilde{y}, \widetilde{r}, \widetilde{u}) \\
&\qquad (m(\widetilde{v}, r, l).\overline{y_1}\langle r_1, u_1\rangle.\overline{u_1}\langle v_1\rangle.\cdots.\overline{y_l}\langle r_l, u_l\rangle.\overline{u_l}\langle c_l\rangle.\overline{g}\langle r, l\rangle \\
&\qquad \mid [\![\mathsf{var}\ Y_1]\!](y_1, \mathsf{nil}) \mid \cdots \mid [\![\mathsf{var}\ Y_l]\!](y_l, \mathsf{nil}) \mid g(r,l).[\![S]\!](l, r, \widetilde{x} \cup \widetilde{y}, \widetilde{k}))
\end{aligned}
$$

図 5　メソッド定義に対する π 計算記述への変換規則
Fig. 5　Translation rules for $Mdecs$

る．Boolean クラスの名前は k_{Bool} で $k_{Bool} \in \widetilde{k}$ である．

クラス定義　$Cdec$ に対する π 計算記述への変換規則を図 4 に示す．名前 k がここで定義されているクラスを示し，名前 a がこのクラスのインスタンスの一つを示す．インスタンス生成の時以下のように動作する．

$$
\begin{aligned}
&[\![Cdec]\!](\widetilde{k}) \\
&\xrightarrow{k(l)} (\boldsymbol{\nu}a)\,(\overline{l}\langle a\rangle.(\boldsymbol{\nu}\widetilde{x}, \widetilde{k})\,(\cdots))) \mid [\![Cdec]\!](\widetilde{k}) \\
&\xrightarrow{\overline{l}\langle a\rangle} (\boldsymbol{\nu}\widetilde{x}, \widetilde{k})\,(\cdots) \mid [\![Cdec]\!](\widetilde{k})
\end{aligned}
$$

k によるメッセージの受信がインスタンスの生成要求を意味する．このときメッセージとして渡される名前 l を通して新しいインスタンスを示す名前 a を返す．a を用いてメソッドにアクセスする．$\boldsymbol{\nu}$ 演算子により個々のインスタンスを示す名前はすべて異なる．$\widetilde{x}$ はメンバ変数の名前，$\widetilde{m}$ はメソッドの名前，$\widetilde{c}$ は各メンバ変数の初期値を表す．メンバ変数にはインスタンス自身を示す $this$ が含まれ初期値は a である．

変数宣言　$Vdecs$ に対する π 計算記述への変換規則を図 4 に示す．$\widetilde{x}$ が各変数の名前，$\widetilde{c}$ が各変数の初期値を示す．それぞれの変数にアクセスするには変数を示す名前 x を通して 2 つの名前 r, u を渡す．値を参照する場合は r を通して値を受信する．値を更新する場合は u を通して新しい値を送信する．

メソッド定義　$Mdecs$ 及び $Mdec$ に対する π 計算記述への変換規則を図 5 に示す．すべてのメソッドは必ずいずれかのインスタンスに属している．そのインスタンスを名前 a によって示す．メソッド呼び出し時の動作例を以下に示す．

$$
\begin{aligned}
&[\![Mdecs]\!](\widetilde{m}, a, \widetilde{x}, \widetilde{k}) \\
&\xrightarrow{a(n)}\ \overline{n}\langle\widetilde{m}\rangle.([\![Mdec_1]\!](m_1, \widetilde{x}, \widetilde{k}) \mid \cdots) \\
&\qquad \mid [\![Mdecs]\!](\widetilde{k}, a, \widetilde{x}, \widetilde{k}) \\
&\xrightarrow{\overline{n}\langle\widetilde{m}\rangle}\ ([\![Mdec_1]\!](m_1, \widetilde{x}, \widetilde{k}) \mid \cdots) \\
&\qquad \mid [\![Mdecs]\!](\widetilde{k}, a, \widetilde{x}, \widetilde{k}) \\
&\xrightarrow{m(\widetilde{v}, r, l)} \cdots\cdots
\end{aligned}
$$

a を通して要求を受けるとそのインスタンスに属するメソッドを示す名前すべてを返す．その中から呼び出し元が希望するメソッドを示す名前 m を選択し，その名前を通して引数 $\widetilde{v}$ などを送信することでメソッド呼び出しとする．$\widetilde{y}$ は実引数を表す名前，$\widetilde{r}, \widetilde{u}$ は実引数にアクセスするための名前である．また m を通して渡される名前 r は return 文により値を返すために利用される．

文　文要素 S に対する π 計算記述への変換規則 $[\![S]\!](l, r, \widetilde{x}, \widetilde{k})$ を図 **6** に示す．l は文の実行完了を示す名前，r は return 文による値の返却を示す名前，$\widetilde{x}$ はメンバ変数，$\widetilde{k}$ はプログラム中で定義されているクラスを示す名前である．

$[\![\mathsf{wait}\ E]\!]$ では E を評価するサブプロセスと時間待ちを行うサブプロセスが並行に動作する．E の評価値をローカルな名前 l' を通して時間待ちを行うサブプロセスに送信し時間待ちが発生する．

$[\![\mathsf{within}\ E\ \mathsf{do}\ S_1\ \mathsf{timeout}\ S_2\ \mathsf{done}]\!]$ では初めに E を評価しその値をローカルな名前 l'' を通して送信する．受信側では S_1 を実行するサブプロセスとタイムアウトまで時間を計測するサブプロセスが並行に動作する．時間内に S_1 が完了しなければタイムアウトの発生により S_2 の実行が選択さ

$$
\begin{aligned}
[\![X := E]\!](l, r, \widetilde{x}, \widetilde{k}) &\equiv (\boldsymbol{\nu} l')([\![E]\!](l', \widetilde{x}, \widetilde{k}) \mid (\boldsymbol{\nu} r, u)(l'(v).\overline{x_X}\langle r, u\rangle.\overline{u}\langle v\rangle.\overline{l}))\\
[\![E]\!](l, r, \widetilde{x}, \widetilde{k}) &\equiv (\boldsymbol{\nu} l')([\![E]\!](l', \widetilde{x}, \widetilde{k}) \mid l'(v).\overline{l})\\
[\![S_1; S_2]\!](l, r, \widetilde{x}, \widetilde{k}) &\equiv (\boldsymbol{\nu} l')([\![S_1]\!](l', r, \widetilde{x}, \widetilde{k}) \mid l'.[\![S_2]\!](l, r, \widetilde{x}, \widetilde{k}))\\
[\![\mathsf{wait}\ E]\!](l, r, \widetilde{x}, \widetilde{k}) &\equiv (\boldsymbol{\nu} l')([\![E]\!](l', \widetilde{x}, \widetilde{k}) \mid l'(n).t[n].\overline{l})\\
[\![\mathsf{return}\ E]\!](l, r, \widetilde{x}, \widetilde{k}) &\equiv (\boldsymbol{\nu} l')([\![E]\!](l', \widetilde{x}, \widetilde{k}) \mid l'(v).\overline{r}\langle v\rangle)\\
[\![\mathsf{if}\ E\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{endif}]\!](l, r, \widetilde{x}, \widetilde{k}) &\equiv (\boldsymbol{\nu} l', l_1, l_2)\\
&\quad ([\![E]\!](l', \widetilde{x}, \widetilde{k}) \mid (\boldsymbol{\nu} t, f)(l'(v).\overline{v}\langle\widetilde{n}\rangle.\overline{n_1}\langle t, f\rangle.(t.\overline{l_1} \mid f.\overline{l_2}))\\
&\quad \mid l_1.[\![S_1]\!](l, r, \widetilde{x}, \widetilde{k}) \mid l_2.[\![S_2]\!](l, r, \widetilde{x}, \widetilde{k}))\\
[\![\mathsf{while}\ E\ \mathsf{do}\ S\ \mathsf{done}]\!](l, r, \widetilde{x}, \widetilde{k}) &\equiv (\boldsymbol{\nu} g, l', l_1, l_2)\\
&\quad (\overline{g} \mid !\, g.([\![E]\!](l', \widetilde{x}, \widetilde{k})\\
&\qquad \mid (\boldsymbol{\nu} t, f)(l'(v).\overline{v}\langle\widetilde{n}\rangle.\overline{n_1}\langle t, f\rangle.(t.\overline{l_1} \mid f.\overline{l_2}))\\
&\qquad \mid l_1.[\![S]\!](g, r, \widetilde{x}, \widetilde{k}) \mid l_2.\overline{l}))\\
[\![\mathsf{within}\ E\ \mathsf{do}\ S_1\ \mathsf{timeout}\ S_2\ \mathsf{done}]\!](l, r, \widetilde{x}, \widetilde{k}) &\equiv (\boldsymbol{\nu} l', l'', r')\\
&\quad ([\![E]\!](l'', \widetilde{x}, \widetilde{k})\\
&\quad \mid l''(n).([\![S_1]\!](l', r', \widetilde{x}, \widetilde{k})\\
&\qquad \mid ((r'(v).\overline{r}\langle v\rangle \mid l'.\overline{l}) + t[n].[\![S_2]\!](l, r, \widetilde{x}, \widetilde{k}))))
\end{aligned}
$$

図 6 文要素に対する π 計算記述への変換規則
Fig. 6 Translation rules for statements

$$
\begin{aligned}
[\![\mathsf{true}]\!](l, \widetilde{x}, \widetilde{k}) &\equiv (\boldsymbol{\nu} l', t, f)(\overline{k_{Bool}}\langle l', t, f\rangle.\overline{t}.l'(b).\overline{l}\langle b\rangle)\\
[\![\mathsf{false}]\!](l, \widetilde{x}, \widetilde{k}) &\equiv (\boldsymbol{\nu} l', t, f)(\overline{k_{Bool}}\langle l', t, f\rangle.\overline{f}.l'(b).\overline{l}\langle b\rangle)\\
[\![\mathsf{nil}]\!](l, \widetilde{x}, \widetilde{k}) &\equiv \overline{l}\\
[\![Numeral]\!](l, \widetilde{x}, \widetilde{k}) &\equiv \overline{l}\langle Numeral\rangle\\
[\![\mathsf{new}(A)]\!](l, \widetilde{x}, \widetilde{k}) &\equiv (\boldsymbol{\nu} l')(\overline{k_A}\langle l'\rangle.l'(c).\overline{l}\langle c\rangle)\\
[\![X]\!](l, \widetilde{x}, \widetilde{k}) &\equiv (\boldsymbol{\nu} r, u)(\overline{x_X}\langle r, u\rangle.r(v).\overline{l}\langle v\rangle)\\
[\![E!m(E_1, \ldots, E_n)]\!](l, \widetilde{x}, \widetilde{k}) &\equiv (\boldsymbol{\nu} h, h_1, \ldots, h_n)\\
&\quad ([\![E]\!](h, \widetilde{x}, \widetilde{k}) \mid [\![E_1]\!](h_1, \widetilde{x}, \widetilde{k}) \mid \cdots \mid [\![E_n]\!](h_n, \widetilde{x}, \widetilde{k})\\
&\quad \mid h(v).h_1(v_1).\cdots.h_n(v_n).(\boldsymbol{\nu} r, l', n)(\overline{v}\langle n\rangle.n(\widetilde{m}).\overline{m_M}\langle v_1, \ldots, v_n, r, l'\rangle\\
&\qquad \mid r(v).\overline{l}\langle v\rangle \mid l'.\overline{l}))
\end{aligned}
$$

図 7 式要素に対する π 計算記述への変換規則
Fig. 7 Translation rules for expressions

れる．

式　式要素 E に対する π 計算記述への変換規則 $[\![E]\!](l, \widetilde{x}, \widetilde{k})$ を図 **7** に示す．名前 $l, \widetilde{x}, \widetilde{k}$ が示すものは文要素と同じである．$[\![\mathsf{new}(A)]\!]$ の k_A はクラス A を示す名前，$[\![X]\!]$ の x_X は変数 X を示す名前である．

インスタンス生成 $[\![\mathsf{new}(A)]\!]$ では k_A を通してクラス定義のプロセスに名前 l' を渡し，l' を通してインスタンス c を受信する．

メソッド呼び出し $[\![E!m(E_1, \ldots, E_n)]\!]$ では，初めに呼び出し先のインスタンス E と引数 E_i を評価する．次に呼び出し先のインスタンスを示す名前 v を通して名前 n を送信し，n を通してそのインスタンスが持つメソッドの名前 $\widetilde{m}$ を受信する．そして $\widetilde{m}$ に含まれる目的のメソッドを示す名前 m_M に対し引数を送信する．return 文によって返される値は r を通して受信する．値を返さないメソッドの完了は l' を通して通知される．

Boolean クラス　Boolean クラスに対する π 計算記述への変換規則[5] を図 **8** に示す．k が Boolean クラスを示す名前，b_t が真，b_f が偽を示す名前である．BoolClass は Boolean クラスにアクセスするためのインターフェースであり，名前 k を通してアクセスする．

5. 変　換　例

本節では OOL_{RT} プログラムを変換することによってどのような π 計算による記述が得られるか具体的な

$$\begin{aligned}
\mathrm{Boolean}(k) &\equiv (\boldsymbol{\nu} b_t, b_f)(\mathrm{BoolClass}(k, b_t, b_f) \mid \mathrm{Bool}_t(b_t, k) \mid \mathrm{Bool}_f(b_f, k)) \\
\mathrm{BoolClass}(k, b_t, b_f) &\equiv !\, k(l, t, f).(t.\overline{l}\langle b_t\rangle \mid f.\overline{l}\langle b_f\rangle) \\
\mathrm{Bool}_v(b, k) &\equiv (\boldsymbol{\nu}\widetilde{n})(!\,\mathrm{BoolBody}(b, \widetilde{n}) \mid !\,\mathrm{BoolVal}_v(n_1) \\
&\qquad \mid !\,\mathrm{BoolNot}(n_2, n_1, k) \mid !\,\mathrm{BoolAnd}(n_3, n_1, k)) \\
\mathrm{BoolBody}(b, \widetilde{n}) &\equiv b(\widetilde{n'}).(n_1'(t, f).\overline{n_1}\langle t, f\rangle \mid n_2'(l).\overline{n_2}\langle l\rangle \mid n_3(b', l).\overline{n_3}\langle b', l\rangle) \\
\mathrm{BoolVal}_t(n_1) &\equiv n_1(t, f).\overline{t} \\
\mathrm{BoolVal}_f(n_1) &\equiv n_1(t, f).\overline{f} \\
\mathrm{BoolNot}(n_2, n_1, k) &\equiv n_2(l).\overline{n_1}\langle t, f\rangle.(t.\overline{k}\langle l, t, f\rangle.\overline{f} \mid f.\overline{k}\langle l, t, f\rangle.\overline{t}) \\
\mathrm{BoolAnd}(n_3, n_1, k) &\equiv n_3(b', l).\overline{n_1}\langle t, f\rangle.(f.\overline{k}\langle l, t, f\rangle.\overline{f} \mid t.b'(\widetilde{n'}).\overline{n_1'}\langle t', f'\rangle.(t'.\overline{k}\langle l, t, f\rangle.\overline{t} \mid f'.\overline{k}\langle l, t, f\rangle.\overline{f}))
\end{aligned}$$

図 8 Boolean クラスに対する π 計算記述への変換規則
Fig. 8 Translation rules for Boolean class

```
 1: program P is
 2:
 3:   class Timer is
 4:     method start() is
 5:       while true do
 6:         wait 10;
 7:         this!fire()
 8:       done
 9:
10:     method fire() is
11:       ....
12:
13: with
14:   new(Timer)!start()
```

図 9 タイマーのプログラム
Fig. 9 A program of timer

例を示す．

ここでは簡単なタイマーのプログラムを取り上げる．プログラムを図 **9** に示す．$start$ メソッドにより実行を開始し，一定時間毎に $fire$ メソッドを呼び出す．処理内容は $fire$ メソッドに実装するが，この例では省略する．

図 9 のプログラムを図 4，図 5，図 6，図 7 の変換規則を用いて時間拡張した π 計算による記述に変換する．変換結果を図 **10** に示す．

一定時間後に $fire$ メソッドが呼ばれることを変換結果を用いて確認する．プログラム上では `wait 10; this!fire()` の部分であり，ここの変換結果である $[\![\mathsf{wait}\ 10; this!fire()]\!]$ を展開した式と遷移の経過を図 **11** に示す．図中の $\rightsquigarrow^n$ は n 回連続して $\rightsquigarrow$ による遷移が発生したことを表す．α は τ などの適切なアクションを示すとする．

初めの状態から τ 遷移が一回発生し 2 つのプロセス $t[10].\overline{l'}$，$l'.(\boldsymbol{\nu} l'\ P)$ が並行に動作する．この時，2 つ目のプロセスは l' でブロックされているため実際に動作するのは 1 つ目のプロセスだけであり，10 単位時間の時間待ちとなる．その後，x_{this}, r, h, v といった名前による遷移が発生しさらに m_{fire} による遷移が発生する．m_{fire} は $fire$ メソッドを表しており，一定時間経過した後に呼び出されることがわかる．

図 11 はプログラムから変換して得られた式なので複雑だが，システムの設計段階では同じ動作を

$$t[10].\overline{m_{fire}}.\cdots\cdots$$

のようにより直接的に表現する．この場合遷移は

$$\begin{aligned}
& t[10].\overline{m_{file}}.\cdots\cdots \\
\rightsquigarrow^{10}\ & t[0].\overline{m_{file}}.\cdots\cdots \\
\xrightarrow{\tau}\ & \overline{m_{file}}.\cdots\cdots \\
\xrightarrow{m_{fire}}\ & \cdots\cdots
\end{aligned}$$

となり，10 単位時間経過した後に $fire$ メソッドを呼び出していることがわかる．

図 11 の遷移と比較すると，m_{fire} による遷移の前に 10 単位時間経過する点が同じである．システム全体の遷移を考えると，図 11 の x_{this}, r などによる遷移は通信の相手となるプロセスがあるため τ 遷移である．そこで時間経過，m_{fire} のような注目するアクション，τ 以外のアクションによる遷移列が，設計段階に記述されるシステムの動作を表す式と，プログラムから変換して得られる式とで同じになるか調べることで動作の等価性を検証する．この検証により実装が設計に従って正しく行われたかどうか確認する．

6. おわりに

本論文では実時間システムの開発に適用するために π 計算を時間に関して拡張し，リアルタイムオブジェクト指向言語に対して時間拡張した π 計算による記述への変換規則を定義した．本論文の変換規則を用いて実時間システムのソフトウェアプログラムを形式的表

$$
\begin{aligned}
P &= (\boldsymbol{\nu} k_{Timer}, k_{Bool}) \\
&\quad ([\![Timer]\!](\widetilde{k}) \mid \mathrm{Boolean}(k_{Bool}) \mid (\boldsymbol{\nu} l)\, [\![\mathsf{new}(Timer)!start()]\!](l, \mathsf{nil}, \widetilde{k})) \\
[\![Timer]\!](\widetilde{k}) &= !(\boldsymbol{\nu} a)\, (k_{Timer}(l).\overline{l}\langle a\rangle.(\boldsymbol{\nu} x_{this}, m_{start}, m_{fire}) \\
&\qquad\qquad (!a(n).\overline{n}\langle \widetilde{m}\rangle.([\![Mdec_{start}]\!](m_{start}, \widetilde{x}, \widetilde{k}) \\
&\qquad\qquad\qquad \mid [\![Mdec_{fire}]\!](m_{fire}, \widetilde{x}, \widetilde{k})))) \\
[\![Mdec_{start}]\!](m, \widetilde{x}, \widetilde{k}) &= (\boldsymbol{\nu} g)\, (m(r,l).\overline{g}\langle r, l\rangle \mid g(r,l).[\![Body_{start}]\!](l, r, \widetilde{x}, \widetilde{k})) \\
[\![Body_{start}]\!](l, r, \widetilde{x}, \widetilde{k}) &= (\boldsymbol{\nu} g, l', l_1, l_2) \\
&\quad (\overline{g} \mid !g.([\![\mathsf{true}]\!](l', \widetilde{x}, \widetilde{k}) \\
&\qquad \mid (\boldsymbol{\nu} r, f)\, (l'(v).\overline{v}\langle \widetilde{n}\rangle.\overline{n_1}\langle r, f\rangle.(t.\overline{l_1} \mid f.\overline{l_2})) \\
&\qquad \mid l_1.[\![\mathsf{wait}\ 10;\ this!fire()]\!](g, r, \widetilde{x}, \widetilde{k}) \\
&\qquad \mid l_2.\overline{l})) \\
[\![\mathsf{wait}\ 10;\ this!fire()]\!](l, r, \widetilde{x}, \widetilde{k}) &= (\boldsymbol{\nu} l')\, ([\![\mathsf{wait}\ 10]\!](l', r, \widetilde{x}, \widetilde{k}) \mid l'.[\![this!fire()]\!](l, r, \widetilde{x}, \widetilde{k})) \\
[\![\mathsf{wait}\ 10]\!](l, r, \widetilde{x}, \widetilde{k}) &= (\boldsymbol{\nu} l')\, ([\![10]\!](l', \widetilde{x}, \widetilde{k}) \mid l'(n).t[n].\overline{l}) \\
[\![10]\!](l, \widetilde{x}, \widetilde{k}) &= \overline{l}\langle 10\rangle \\
[\![this!fire()]\!](l, r, \widetilde{x}, \widetilde{k}) &= (\boldsymbol{\nu} l')\, ([\![this!fire()]\!](l', \widetilde{x}, \widetilde{k}) \mid l'(v).\overline{l}) \\
[\![this!fire()]\!](l, \widetilde{x}, \widetilde{k}) &= (\boldsymbol{\nu} h)\, ([\![this]\!](h, \widetilde{x}, \widetilde{k}) \\
&\qquad \mid h(v).(\boldsymbol{\nu} r, l', n)\, (\overline{v}\langle n\rangle.n(\widetilde{m}).\overline{m_{fire}}\langle r, l'\rangle \mid r(v).\overline{v} \mid l'.\overline{l})) \\
[\![this]\!](l, \widetilde{x}, \widetilde{k}) &= (\boldsymbol{\nu} r, u)\, (\overline{x_{this}}\langle r, u\rangle.r(v).\overline{l}\langle v\rangle) \\
[\![Mdec_{fire}]\!](m, \widetilde{x}, \widetilde{k}) &= (\boldsymbol{\nu} g)\, (m(r,l).\overline{g}\langle r, l\rangle \mid g(r,l).[\![....]\!](l, r, \widetilde{x}, \widetilde{k})) \\
[\![\mathsf{new}(Timer)!start()]\!](l, \widetilde{x}, \widetilde{k}) &= (\boldsymbol{\nu} h)\, ([\![\mathsf{new}(Timer)]\!](h, \widetilde{x}, \widetilde{k}) \\
&\qquad \mid h(v).(\boldsymbol{\nu} r, l', n)\, (\overline{v}\langle n\rangle.n(\widetilde{m}).\overline{m_{start}}\langle r, l'\rangle \mid r(v).\overline{l}\langle v\rangle \mid l'.\overline{l})) \\
[\![\mathsf{new}(Timer)]\!](l, \widetilde{x}, \widetilde{k}) &= (\boldsymbol{\nu} l')\, (\overline{k_{Timer}}\langle l'\rangle.l'(c).\overline{l}\langle c\rangle)
\end{aligned}
$$

図 10 タイマーのプログラムの π 計算による記述
Fig. 10 π-calculus expressions of a timer program

$$
\begin{aligned}
& (\boldsymbol{\nu} l')\, ((\boldsymbol{\nu} l'')\, (\overline{l''}\langle 10\rangle \mid l''(n).t[n].\overline{l'}) \mid l'.(\boldsymbol{\nu} l'')\, (P \mid l''(v).\overline{l})) \\
\xrightarrow{\tau}\ & (\boldsymbol{\nu} l')\, (t[10].\overline{l'} \mid l'.(\boldsymbol{\nu} l'')\, (P \mid l''(v).\overline{l})) \\
\rightsquigarrow^{10}\ & (\boldsymbol{\nu} l')\, (t[0].\overline{l'} \mid l'.(\boldsymbol{\nu} l'')\, (P \mid l''(v).\overline{l})) \\
\xrightarrow{\tau}{}^{2}\ & (\boldsymbol{\nu} l'')\, (P \mid l''(v).\overline{l}) \\
\xrightarrow{\alpha}{}^{3}\ & (\boldsymbol{\nu} l'')\, ((\boldsymbol{\nu} r, l', n)\, (\overline{v}\langle n\rangle.n(\widetilde{m}).\overline{m_{fire}}\langle r, l'\rangle \mid r(v).\overline{l''}\langle v\rangle \mid l'.\overline{l''}) \mid l''(v).\overline{l}) \\
\xrightarrow{\alpha}{}^{2}\ & (\boldsymbol{\nu} l'')\, ((\boldsymbol{\nu} r, l')\, (\overline{m_{fire}}\langle r, l'\rangle \mid r(v).\overline{l''}\langle v\rangle \mid l'.\overline{l''}) \mid l''(v).\overline{l}) \\
\xrightarrow{\overline{m_{fire}}}\ & (\boldsymbol{\nu} l'')\, ((\boldsymbol{\nu} r, l')\, (r(v).\overline{l''}\langle v\rangle \mid l'.\overline{l''}) \mid l''(v).\overline{l}) \\
\text{ここで } P &= (\boldsymbol{\nu} h)\, ((\boldsymbol{\nu} r, u)\, (\overline{x_{this}}\langle r, u\rangle.r(v).\overline{h}\langle v\rangle) \mid h(v).(\boldsymbol{\nu} r, l', n)\, (\overline{v}\langle n\rangle.n(\widetilde{m}).\overline{m_{fire}}\langle r, l'\rangle \mid r(v).\overline{l''}\langle v\rangle \mid l'.\overline{l''}))
\end{aligned}
$$

図 11 $[\![\mathsf{wait}\ 10; this!fire()]\!]$ の遷移
Fig. 11 The transition of $[\![\mathsf{wait}\ 10; this!fire()]\!]$

現に変換する．この表現と設計段階で与えられた動作記述の遷移列を比較して等価性を検証することにより，実装が設計に従っているか確認することができる．本論文では OOL_{RT} に対し変換規則を与えた．しかし，この変換規則は OOL_{RT} に本質的に依存しているわけではない．クラス定義，インスタンス生成，メソッド呼び出し，制御構造など同等の記述能力を持つオブジェクト指向言語に対する一般的な規則となっている．

実時間システムはその性質により正しく動作することを動作テストによって確認することが難しい．形式的記述を用いることでシステムの動作の解析や検証を機械的に処理することができる．このことにより動作確認を主にテストのみによって行う従来手法による開発に比べ信頼性が向上する．

π 計算を時間に関して拡張することで実時間システムの動作を形式的に記述できるようになった．形式的

記述を利用して正しく動作するか機械的に検証することが可能であるが，具体的な検証手法については今後の課題である．実際の実時間システム開発においてプログラムを時間拡張した π 計算による記述に変換し検証しようとする場合，システム規模が大きくなるに従ってより多くの式が生成される．実際の開発に適用するための環境を整えることは今後の課題である．

参 考 文 献

1) Gomma, H.: *Designing Concurrent, Distributed, and Real-Time Applications with UML*, Addison Wesley (2000).
2) Milner, R.: *Communicating and Mobile Systems: the π-Calculus*, Cambridge University Press (1999).
3) Milner, R., Parrow, J. and Walker, D.: A Calculus of mobile processes, Part I/II, *Information and Computation*, Vol. 100, pp. 1–77 (1992).
4) Sangiorgi, D. and Walker, D.: *The π-calculus: A Theory of Mobile Processes*, Cambridge University Press (2001).
5) Walker, D.: Objects in the π-Calculus, *Information and Computation*, Vol. 116, No. 2, pp. 253–271 (1995).