通信プロセスモデルによるAIBO OPEN-Rプログラムのデッドロックフリー解析手法

末次 亮[†] 結縁祥治[†] 阿草清滋[†]

本論文では AIBO プログラムの振舞いを通信プロセスモデルに基づいて表現し，デッドロックフリー解析を行う手法を提案する．AIBO プログラムの標準的開発環境 OPEN-R で開発される並行オブジェクトは 2 種の信号のやりとりによって同期される．この 2 種の同期信号のやりとりを AIBO プログラムの抽象的な振舞いとしてとらえ，ソースコードの構造を保ったまま π 計算によってモデル化する．モデル検査器 Mobility Workbench を用いて，ロボットにおける致命的な欠陥であるデッドロックの可能性を解析する．モデル検査時の状態爆発の問題に対応するため全体の振舞いを分割し，解析時の状態数を削減する．本論文における解析ではシステムのデッドロックを検出できるだけではなく，デッドロック発生の原因となるコンポーネントの特定が可能である．本論文ではヘッドセンサに反応してメッセージを表示する AIBO プログラムを用いた解析例を示す．

A Deadlock Free Analysis for AIBO OPEN-R Programs Based on Communicating Processes

Ryo Suetsugu,[†] Shoji Yuen[†] and Kiyoshi Agusa[†]

We propose a compositional analysis method to ensure deadlock freeness of AIBO programs based on communicating processes. Concurrent objects of AIBO programs with the OPEN-R API are synchronized by two types of signals: ready and notify. Focusing on these signals, we describe the abstract behavior of AIBO by the π-calculus along with the source code structure concerning synchronization between concurrent objects. We use the Mobility Workbench to check the deadlock capability. We present a decomposing method to reduce the combination of states of concurrent objects. When a deadlock possibility is pointed out, our method enables not only to detect deadlock of the whole system, but also to point out which component may cause the deadlock. We illustrate our method by an example of an AIBO program to display messages when its head sensor pushed.

1. はじめに

機能要求の複雑化や開発期限の短期化に対応するため，組み込みソフトウェアはオブジェクト指向モデリングによって開発される．センサやアクチュエータの制御などの各機能をコンポーネントとして実現し，合成することでシステム全体の制御を行う．組み込みソフトウェアは一般に資源の限られた環境上に開発される．少ない資源の上でのコンポーネントの合成を実現するために，コンポーネント間を単純な非同期メッセージ通信で接続する．

一方，非同期メッセージ通信ではメッセージの送信後，受信側のメッセージの処理の開始時点が判明しないため，個々のオブジェクト間の制御を注意深く行わないと，実行のタイミングによってデッドロックなどの予期しない振舞いが発生する．

本論文では，並行オブジェクトによって記述された AIBO ロボットを制御するソフトウェアを通信プロセスによってモデル化し，並行オブジェクトの振舞いが同期信号によってデッドロックしないことを解析する手法を示す．デッドロックはロボットなどの組み込みソフトウェアでは致命的な結果をもたらす場合が多いため，特にデッドロックフリー性に着目する．

メッセージ通信フローとソースコードの対応関係を把握するために AIBO プログラムをそのソースコードの構造を保ったまま代表的な通信プロセス計算である π 計算[5),7)] でモデル化する．AIBO の標準的な制御プログラムは OPEN-R 開発環境において C++ によって記述される．OPEN-R 開発環境において開発される AIBO プログラムの並行オブジェクトの同期は ready と notify の 2 種の信号で制御される．意図し

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

ない順序によるこれらの信号のやりとりはデッドロックの原因となる．本研究では，この同期信号に注目して，AIBO プログラムにおいてデッドロックの発生を防ぐ解析手法を提案する．

解析を行うツールとして π 計算の振舞いに対するモデル検査器 Mobility Workbench [8] を用いる．モデル検査では一般に状態数が爆発し，現実的な時間およびメモリ量で検証を行うことができない場合が多い．デッドロックが発生しない性質を保存しながら全体の振舞いをコンポーネント群に分割する手法を示し，解析時の状態数を削減する．

本論文の構成は以下のとおりである．2 章で本論文の準備として π 計算について説明する．3 章で AIBO プログラムの開発環境である OPEN-R について説明し，OPEN-R オブジェクト間のメッセージ通信プロトコルについて説明する．4 章で AIBO プログラムの π 計算への変換を定義し，メッセージ通信フローの解析を行う．5 章で実際に AIBO プログラムを π 計算に変換し，デッドロックフリー解析の例を示し解析手法の評価を行う．6 章で関連研究を示し，本研究と比較を行う．

2. π 計 算

π 計算におけるプロセスは名前を用いたリンクを通してハンドシェイク方式の通信によって他のプロセスと相互作用を行う．動作の可能性はプレフィックスによって表現される．π 計算の動作は以下の文法で与えられる．

$$\pi ::= \overline{x}y \mid x(z) \mid \tau$$

各動作の直感的な意味は以下のとおりである．

- $\overline{x}y$：リンク x による名前 y の送信
- $x(y)$：リンク x を通じた名前 y への受信
- τ：ハンドシェイクによる内部通信の発生

動作 $\overline{x}y$ または $x(y)$ において x をサブジェクト，y をオブジェクトと呼ぶ．動作 π におけるサブジェクトとオブジェクトをそれぞれ $subj(\pi)$，$obj(\pi)$ と記述する．

プロセス P を以下の BNF で与える．

$$P ::= M \mid P_1|P_2 \mid \boldsymbol{\nu}z\,P \mid\ !P$$
$$M ::= 0 \mid \pi.P \mid M_1{+}M_2$$

π 計算の動作的意味として，プロセス間のラベル付き遷移関係 $\xrightarrow{\alpha}$ は構造動作意味定義によって**図 1** のように与えられる．構造合同関係 $\equiv$ は**図 2** のように与えられる．$P \xrightarrow{\pi} P'$ はプロセス P が動作 π を行った後 P' となることを表す．この動作意味論において，プロセスを構成する演算子は以下のような意味を持つ．

- 0 は停止したプロセスを表す．
- $\pi.P$ は動作 π を行えば P となるプロセスを表す．
- $+$ は選択を表す．$P{+}Q$ のとき，プロセス P，Q のうち動作可能な一方のプロセスを選択する．
- $|$ はプロセスの並行合成を表す．$P|Q$ のとき，プロセス P，Q は並行に動作する．プロセス P，Q のプレフィックスが同じ名前のときハンドシェイクで通信を行うことができる．
- $\boldsymbol{\nu}zP$ はプロセス P において名前 z による外部との通信を制限する．
- $!P$ はプロセス P を必要なだけ起動する．各プロ

$$\text{OUT}: \frac{}{\overline{x}y.P \xrightarrow{\overline{x}y} P} \qquad \text{INP}: \frac{}{x(z).P \xrightarrow{xy} P\{y/z\}}$$

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

$$\text{PAR-L}: \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\ bn(\alpha) \cap fn(Q) = \emptyset$$

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

$$\text{CLOSE-L}: \frac{P \xrightarrow{\overline{x}(z)} P' \quad Q \xrightarrow{xz} Q'}{P \mid Q \xrightarrow{\tau} \boldsymbol{\nu}z(P' \mid Q')}\ z \notin fn(Q)$$

$$\text{RES}: \frac{P \xrightarrow{\alpha} P'}{\boldsymbol{\nu}x\,P \xrightarrow{\alpha} \boldsymbol{\nu}x\,P'}\ \text{if } \alpha \notin \{x, \overline{x}\}$$

$$\text{OPEN}: \frac{P \xrightarrow{\overline{x}z} P'}{\boldsymbol{\nu}zP \xrightarrow{\overline{x}(z)} P'}\ z \neq x \qquad \text{REP-ACT}: \frac{P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P' \,|!P}$$

$$\text{REP-COMM}: \frac{P \xrightarrow{\overline{x}y} P' \quad P \xrightarrow{x(y)} P''}{!P \xrightarrow{\tau} (P' \mid P'')\,|!P}$$

$$\text{REP-CLOSE}: \frac{P \xrightarrow{\overline{x}(z)} P' \quad P \xrightarrow{xz} P'}{!P \xrightarrow{\tau} (\boldsymbol{\nu}z(P' \mid P''))\mid !P}\ z \notin fn(P)$$

$$\text{R-STRUCT}: \frac{P \equiv Q \xrightarrow{\alpha} Q' \equiv P'}{P \xrightarrow{\alpha} P'}$$

図 1 π 計算の動作規則
Fig. 1 Structural operational semantics of the π-calculus.

SC-SUM-ASSOC	$M_1 + (M_2 + M_3)$	$\equiv$	$(M_1 + M_2) + M_3$
SC-SUM-COMM	$M_1 + M_2$	$\equiv$	$M_2 + M_1$
SC-SUM-INACT	$M + 0$	$\equiv$	M
SC-COMP-ASSOC	$P_1 \mid (P_2 \mid P_3)$	$\equiv$	$(P_1 \mid P_2) \mid P_3$
SC-COMP-COMM	$P_1 \mid P_2$	$\equiv$	$P_2 \mid P_1$
SC-COMP-COMM	$P \mid 0$	$\equiv$	P
SC-RES	$\boldsymbol{\nu}z\boldsymbol{\nu}wP$	$\equiv$	$\boldsymbol{\nu}w\boldsymbol{\nu}zP$
SC-RES-INACT	$\boldsymbol{\nu}z0$	$\equiv$	0
SC-RES-COMP	$\boldsymbol{\nu}z(P_1 \mid P_2)$	$\equiv$	$P_1 \mid \boldsymbol{\nu}zP_2,\ \text{if } z \notin \text{fn}(P_1)$
SC-REP	$!P$	$\equiv$	$P\,|!P$

図 2 構造合同の公理
Fig. 2 Axioms of structural congruence.

セス P は並行に動作する．

プロセス $x(z).P$，$\boldsymbol{\nu}zP$ において名前 z はプロセス P において束縛される．名前 z がどのプロセスにおいても束縛されないとき z は自由であるという．プロセス P において自由な名前の集合を $fn(P)$ と記述し，束縛された名前の集合を $bn(P)$ と記述する．プロセスの最後の 0 は省略する．たとえば，$x(y).0$ は単に $x(y)$ と書くことにする．さらに，n が文脈より明らかなとき $P_1 + \cdots + P_n$ を $\Sigma_i P_i$ と書く．

3. AIBO プログラミング

3.1 OPEN-R

OPEN-R [1] は AIBO プログラムのためのオブジェクト指向開発環境である．

OPEN-R では 2 種の設定ファイルを用いてメッセージ通信の定義を行い，並行オブジェクトを接続する．“stub.cfg” は各オブジェクトに与えられ，各オブジェクトにおけるメッセージ通信のためのポートを定義する．“CONNECT.CFG” はプログラム全体に唯一与えられるファイルであり，各オブジェクトのポートの接続を記述する．

stub.cfg ではオブジェクトの通信ポートの一覧を記述し，以下の 2 つの属性を記述する．

通信の種類：
そのポートを介して送受信されるメッセージの種類とポート名を定義する．

メソッド定義：
そのポートを介して通信が行われるときに起動されるメソッド名を定義する．ただしメソッドの本体はオブジェクトのメンバ関数として記述する．

CONNECT.CFG ではポートの組としてポート間の接続を定義し，接続されたポート間でメッセージ通信が行われる．

3.2 同期のためのメッセージ通信フロー

OPEN-R オブジェクトの各ポートは “サブジェクト” もしくは “オブザーバ” のいずれかの属性を持つ．メッセージ通信は以下のプロトコルで行われる．まずオブザーバ型のポートからサブジェクト型のポートへ ready メッセージを送信する．サブジェクト型のポートが ready メッセージを受信すると接続されたオブザーバ型のポートへデータとともに notify メッセージを返信する．

それぞれのポートに対してメソッドが関連付けられる．メソッドはポートにおけるメッセージの受信時に起動する．AIBO のスケジューラは非決定的にポートを選びメソッドを起動する．

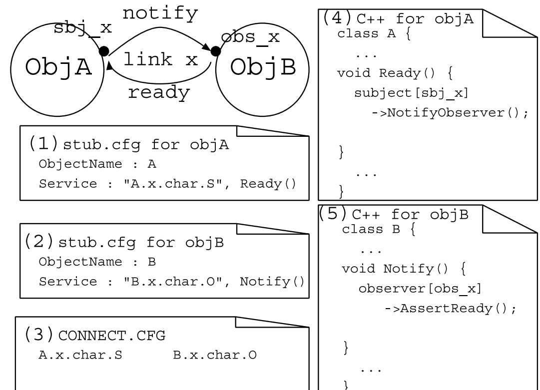

図 3 メッセージ通信定義の記述例
Fig. 3 An example description of message passing.

2 つの並行オブジェクト間のメッセージ通信フローの例を**図 3** に示す．Ready() メソッドはオブジェクト A のポート x に関連付けられ，Notify() メソッドはオブジェクト B のポート x に関連付けられている．

リンク x のサブジェクトポートを “sbj_x” と記述し，オブザーバポートを “obs_x” と記述する．Ready() メソッドにおいて記述されている文 `subject[sbj_x]->NotifyObserver()` はオブジェクト A のポート x に接続されているポートに対して notify メッセージを送信する．Notify() メソッドにおいて記述されている文 `observer[obs_x]->AssertReady()` はオブジェクト B のポート x に接続されている受信可能なポートに対して ready メッセージを送信する．

AIBO ロボットの電源が ON になったときに動作が開始され，このときすべてのオブザーバポートから ready メッセージを送信する．

4. オブジェクト同期に関する振舞いの形式的記述

AIBO のスケジューラは stub.cfg において定義されているメソッドの中で起動可能なメソッドを非決定的に起動する．ここで ready および notify メッセージに着目し AIBO プログラムにおける並行オブジェクトの実行をモデル化する．

4.1 π 計算への変換

AIBO プログラムから π 計算のプロセスへの変換を以下の 4 段階で与える．

Step 1 C++ のソースコードをスライシングし，さらに分岐文の条件節の削除を行うことでメッセージ通信の振舞いを抽出する．

Step 2 スライスされたソースコードを用いて並行オブジェクトを π 計算のプロセスへ変換する．

```
 1  void Notify() {
 2    if (state == MOVING) {
 3       state = WAITING;
 4    } else {
 5       int pressure = getHeadPressure();
 6       if (pressure > 1000) {
 7           subject[sbj_x]->NotifyObserver();
 8       } else {
 9           observer[obs_y]->AssertReady();
10       }
11       state = MOVING;
12     }
13  }
```

図 4 スライス前のプログラム例
Fig. 4 A program example before sliced.

```
 1  void Notify() {
 2    if (                ) {
 3
 4    } else {
 5
 6       if (                ) {
 7           subject[sbj_x]->NotifyObserver();
 8       } else {
 9           observer[obs_y]->AssertReady();
10       }
11
12    }
13  }
```

図 5 スライス後のプログラム例
Fig. 5 A sliced program example.

Step 3 CONNECT.CFG におけるポート間の接続情報を用いてスケジューラを π 計算のプロセスとして定義する．

Step 4 以上で得られたすべてのプロセスを合成する．

以上の各段階の詳細は以下で説明する．

4.1.1 メッセージ通信の抽出

π 計算のプロセスへの変換の前処理として C++ のソースコードに対してスライシングを行い，メッセージ通信動作を抽出する．オブザーバ型のポートを格納する配列とサブジェクト型のポートを格納する配列をスライシング基準としてスライシング技法[9] を適用する．

AIBO プログラムにおいてメッセージ送信のための分岐は外界のイベントに基づいて判定される．たとえば**図 4** のように記述されたあるポートに関連付けられたメソッドにおいて 6 行目の分岐文の条件節はヘッドセンサの入力値に基づいて判定される．通常外界のイベントは非決定的に発生するため，条件節の真偽値は動作時に決定する．後の変換において分岐文は非決定的選択として変換するためにスライスされたプログラムにおいて残された分岐文の条件節を削除する．

図 5 は図 4 を配列 observer および subject をスライシング基準としてスライスし，分岐文の条件節を削除したコード片である．さらに条件分岐を非決定的選択に置き換えると，7 行目の NotifyObserver() と 9 行目に対する AssertReady() との非決定的選択が並行オブジェクト間の同期に関するメッセージ通信として抽出される．

4.1.2 並行オブジェクトの変換

“stub.cfg” と C++プログラムのメソッドの変換 $[\![\ ]\!]$ を示す．正しい動作をする AIBO プログラムにおいて，その意味から NotifyObserver() および AssertReady() は繰返し文に現れない．このためスライスされたプログラムにおいて繰返し文は出現しない．したがって，スライス後のメソッドは `subject[sbj_a]->NotifyObserver()`, `observer[obs_a]->AssertReady()` およびこれらの条件分岐と逐次合成で構成される．

さらにこのように構成される文 S は返り値を持たないメソッド `void` $m()\{S\}$ に出現する．

以下，`subject`，`observer` は，それぞれサブジェクト型，オブザーバ型のポートを格納する配列であるとする．また，end はメソッドの終了を表し，メソッドを構成する文の他の部分には現れない名前とする．

$[\![\texttt{subject}[sbj_a]\texttt{->NotifyObserver()}]\!] = \nu x\ \overline{sbj_a}\ x$

$[\![\texttt{observer}[obs_a]\texttt{->AssertReady()}]\!] = \overline{obs_a}\ A_m$

ここで，ポート obs_a に関連付けられるメソッドが，オブジェクト A のメソッド m であるとする．

$[\![S;]\!] = [\![S]\!]$

$[\![S_1; S_2]\!] = [\![S_1]\!]; [\![S_2]\!]$

ここで，

$$P;P' = \begin{cases} \Sigma_i \pi_i.(P_i;P') & P = \Sigma_i \pi_i.P_i \text{ のとき} \\ P' & P = 0 \text{ のとき} \end{cases}$$

$[\![\texttt{if()}\{S_1\}\texttt{else}\{S_2\}]\!] = [\![S_1]\!] + [\![S_2]\!]$

メソッド m がオブジェクト A に属し，オブザーバ型のポートに関連付けられるとき：

$[\![\texttt{void}\ m()\{S\}]\!] = !A_m.([\![S]\!];\overline{end})$

メソッド m がオブジェクト A に属し，サブジェクト型のポート sbj_x に関連付けられるとき：

$[\![\texttt{void}\ m()\{S\}]\!] = !A_m(x).([\![S]\!];\overline{end})$

以上のように各メソッドを変換すると，メソッドの本体 S は前置演算，選択および 0 によって表される．このためプロセス上の演算 ‘;’ が未定義となることはない．オブジェクト A のポートに関連付けられたメソッドを $m_1, \cdots, m_n$，その本体を $S_1, \cdots, S_n$ とするとオブジェクト A に対応するプロセス Obj_A は各メソッドの変換の並行合成として与えられる．

$$Obj_A = [\![\texttt{void}\ m_1()\{S_1\}]\!] | \cdots | [\![\texttt{void}\ m_n()\{S_n\}]\!]$$

4.1.3 スケジューラの記述

スケジューラはメッセージ通信に従ってメソッドを起動する．本論文ではスケジューラを ‘end’ トークンを管理し，AIBO プログラムがシングルスレッドで動作することを保証するための “ロック” としてモデル化する．

スケジューラの振舞いは CONNECT.CFG と stub.cfg から変換する．それぞれのリンクに対してスケジューラは ready 信号または notify 信号を受信するまで待機する．ポートが信号を受信するとスケジューラは動作中のメソッドが ‘end’ トークンを返信するまで待機する．スケジューラはポートにおける信号の受信時に該当ポートに関連付けられたメソッドを起動する．

オブジェクト A とオブジェクト B 間のあるリンクを x とし，ポート obs_x に関連付けられたメソッドを m() とすると，リンク x に対するスケジューラは

$$\mathrm{Scheduler_x} = $$
$$!(\mathrm{obs_x}(a).\mathrm{end}.\overline{\mathrm{B_m}}\ a \ \ | \ \mathrm{sbj_x}(a).\mathrm{end}.\overline{a})$$

として変換する．“$\overline{\mathrm{B_m}}$” はオブジェクト B のメソッド m() の起動を表す．

AIBO プログラムがリンクを n 個持つとき，スケジューラに対するプロセスは以下のようにすべてのリンク x_i を変換したプロセスの並行合成として定義する．

$$\mathrm{Scheduler} = \mathrm{Scheduler_x}_1 \mid \cdots \mid \mathrm{Scheduler_x}_n$$

4.1.4 AIBO プログラムの変換

AIBO プログラムが n 個のオブジェクトを持ち，m 個のオブザーバポートを持つとき，プログラム全体をプロセス AIBO_System として以下のように変換する．

$$\begin{aligned}&\mathrm{AIBO_System} = \\ &\quad (\boldsymbol{\nu}\mathrm{A_x}_1\ \cdots\ \mathrm{A_x}_m \quad \mathrm{A_Ready}_1 \cdots \mathrm{A_Ready}_m\ \mathrm{end}) \\ &\quad (\overline{\mathrm{A_x}_1}\ \mathrm{A_Ready}_1.\cdots.\ \overline{\mathrm{A_x}_m}\ \mathrm{A_Ready}_m.\ \overline{\mathrm{end}}\ | \\ &\qquad \mathrm{Scheduler}\ \mid\ \mathrm{Obj}_1\ \mid\ \cdots \mid\ \mathrm{Obj}_n)\end{aligned}$$

AIBO の実行環境では，すべての ready メッセージが AIBO の起動時に送信される．AIBO_System の最初のプロセスは初期化手続きを表している．

システムがセンサやアクチュエータのような外界のイベント源を持つとき，これらのイベントも π 計算のプロセスとして変換しシステムと並行合成する．センサのようなプログラムへのデータ入力機器はサブジェクトポートと見なして変換する．アクチュエータのようなプログラムからデータを受ける出力機器はオブザーバポートと見なして変換する．

4.2 デッドロックフリー解析

4.2.1 デッドロックフリー性

本論文のモデルでは AIBO プログラムは実行環境を含めて変換されており，閉じているため AIBO プログラムの実行は τ 遷移のみによって表される．AIBO プログラムにおいてデッドロックしているプロセスとは τ 遷移が存在しないプロセスである．デッドロックフリー性を以下のように定義する．

定義 1. デッドロックフリー性
プロセス P を AIBO プログラムから変換された π 計算の閉じたプロセスとする．$P \xrightarrow{\tau} {}^{*}P'$ であるすべてのプロセス P' に対して $P' \xrightarrow{\tau} P''$ であるプロセス P'' が存在するとき P はデッドロックフリーであるという． □

4.2.2 分解手法

2 つのプロセス間に通信が存在しないときその 2 つのプロセスの遷移の順序は無視できる．本論文では並行オブジェクトの状態の組合せを削減するために互いに通信しないプロセス群に分割してから解析を行う．

通信ポートが重複しないサブプロセスに対して，デッドロックフリーの性質は分割できる．たとえば

$$P = \overline{a}\ \mid\ !(a.(d + \overline{a}))$$
$$Q = \overline{b}\ \mid\ !(b.(\overline{e} + \overline{b}))$$

とするとき $fn(P) \cap fn(Q) = \emptyset$ であり，本論文では τ 遷移のみを考慮するため P，Q はそれぞれデッドロックフリーである．さらに $P \mid Q$ もデッドロックフリーである．一方，

$$Q' = \overline{b}\ \mid\ !(b.(\overline{d} + \overline{b}))$$

のとき $fn(P) \cap fn(Q') = \{d\}$ であり，Q' はデッドロックフリーである．P と Q' が共通の自由な名前 d で通信すると $P \mid Q'$ はデッドロック状態となる．

分割のための条件は以下の補題で表される．

補題 2. P，Q をそれぞれ AIBO プログラムから変換された閉じた π 計算のプロセスとする．$fn(P) \cap fn(Q) = \emptyset$ とし，P と Q のそれぞれのプレフィクスのサブジェクトが $fn(P)$ と $fn(Q)$ に含まれているとする．P と Q がデッドロックフリーであるとき，並行合成したプロセス $P \mid Q$ はデッドロックフリーである． □

補題 2 よりプロセス $P \mid Q$ のデッドロックフリー解析において，サブプロセス P と Q の間でメッセージ通信が発生しないとき，P と Q のそれぞれのデッドロックフリー解析を行えば十分である．

本論文における変換で得られた π 計算のプロセスの分割アルゴリズムを示す．分割は共通の自由な名前において直和分割することで得られる．

AIBO プログラムを変換して得られた π 計算のプロセスにおいてトークンを表す名前 end が送信可能であるとき他の名前は送信可能ではない．メソッドを変換して得られたプロセスは必ず名前 end で送信する．デッドロックフリーであるプロセスにおいて名前 end が送信可能であるときスケジューラを変換して得られたプロセスは名前 end で受信できる．分割されたそれぞれのプロセスにおいて名前 end で受信可能である場合，並行合成されたプロセスにおいても名前 end で受信可能である．このため名前 end のみを共有している場合，並行合成についてデッドロックフリー性を保存する．分割する際に名前 end は分割の基準から除外できる．

プロセスの分割は以下の手順で行う．

(1) π 計算のプロセスのそれぞれのサブプロセスにおいて end 以外の自由な名前を見つける．
(2) end 以外の共通の自由な名前を持つプロセスを並行合成する．
(3) (2) を並行合成されるプロセスがなくなるまで繰り返す．
(4) (3) で得られたそれぞれのプロセス群に対し，4.1.4 項で示した AIBO プログラムの変換と同様に，初期化手続きを表すプロセスを並行合成し，自由な名前を束縛する．

アルゴリズムが停止したときに分割されているプロセスが互いに名前 end 以外の自由な名前を共有しないプロセス群である．このアルゴリズムで分割されたそれぞれのプロセス群に対してそれぞれデッドロックフリー解析を行えば十分である．

5. 例

本章では AIBO のサンプルプログラムを π 計算のプロセスに変換し，デッドロックフリー解析を行い，評価を行う．サンプルプログラムの振舞いの概要を**図 6** に示す．AIBO は頭センサが押下されるまで待機し，ある一定の圧力で押下されれば無線により遠隔地の計算機にメッセージを送信する．

サンプルプログラムは “SensorObserver”，“Display” の 2 つの並行オブジェクトからなる．それぞれのオブジェクトは以下のステップで動作する．

SensorObserver

(1) 頭センサから圧力データを受信する．
(2) 頭センサから得たデータがある一定値以上であれば Display にメッセージを送信する．データがある一定値未満であれば (1) へ戻る．
(3) Display からメッセージの返信を待ち，返信があれば (1) へ戻る．

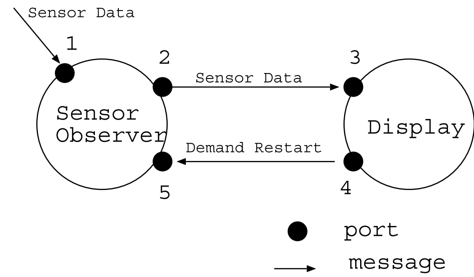

図 6 プログラム例
Fig. 6 A sample program.

Display

(1) SensorObserver からのメッセージを待ち，メッセージを受信すると遠隔地の計算機にメッセージを表示する．
(2) SensorObserver へメッセージを送信し (1) へ戻る．

5.1 変 換

サンプルプログラムについてオブジェクトの一部およびスケジューラの変換を示し，プログラム全体の変換を示す．サンプルプログラムの一部を以下に示す．

Display.cc

```
void Display::DisplayData() {
    subject[sbjReadyS]->NotifyObserver();
    observer[obsDisplay]->AssertReady();
}
```

Display オブジェクトの stub.cfg

```
ObjectName : Display
NumOfOSubject   : 1
NumOfOObserver  : 1
Service : "Display.Display.char.O", DisplayData()
Service : "Display.ReadyS.char.S", Ready()
```

SensorObserver オブジェクトの stub.cfg

```
ObjectName : SensorObserver
NumOfOSubject   : 1
NumOfOObserver  : 2
Service : "SensorObserver.Sensor.OSensorFrameVectorData.O",
                                                  Observe()
Service : "SensorObserver.Start.char.O",StartObserve()
Service : "SensorObserver.Ready.char.S", Ready()
```

CONNECT.CFG

```
OVirtualRobotComm.Sensor.OSensorFrameVectorData.S
    SensorObserver.Sensor.OSensorFrameVectorData.O
Display.ReadyS.char.S SensorObserver.Start.char.O
SensorObserver.Ready.char.S Display.Display.char.O
```

以上の 4 つのソースコードを用いて以降の章でサンプルプログラムの変換の一部を説明する．

5.2 オブジェクトの変換

オブジェクト Display の DisplayData() メソッドの変換を示す．Display_DisplayData_Method として以下のように変換する．

$$\begin{array}{l}\text{Display_DisplayData_Method} = \\ !\ \text{Display_DisplayData}. \\ \boldsymbol{\nu}\text{SensorObserver_StartObserve} \\ \quad \overline{\text{Display_ReadyS}}\ \text{SensorObserver_StartObserve}. \\ \overline{\text{Display_Display}}\ \text{Display_DisplayData}. \\ \overline{\text{end}}\end{array}$$

まずこのプロセスは Display_DisplayData を待つ．これはオブジェクト Display のメソッド DisplayData() の起動を意味する．次にポート ReadyS メッセージをリンク先のポートに関連付けられたメソッド StartObserve() の起動用のポインタとともに送信する．さらに Display メッセージを自身に関連付けられたメソッド DisplayData() の起動用のポインタとともに送信する．最後にメソッド DisplayData() の実行の終了を意味する end メッセージを送信する．

オブジェクトはメソッドから変換されたプロセスの並行合成として変換される．オブジェクト Display は以下のように変換される．

$$\begin{array}{l}\text{Display} = \\ \text{Display_DisplayData_Method} \\ \mid \text{Display_Ready_Method}\end{array}$$

ここで Display_Ready_Method はオブジェクト Display のサブジェクトポートに関連付けられたメソッド Ready() から変換される．

5.3 スケジューラの記述

AIBO の実行環境のスケジューラはプロセス Scheduler として CONNECT.CFG によって以下のように得られる．

$$\begin{array}{l}\text{Scheduler} = \\ !(\ \text{Display_ReadyS}(x).\text{end}. \\ \quad \overline{\text{SensorObserver_StartObserve}}\ x \\ \mid \text{SensorObserver_Start}(x). \\ \quad \text{end}.\overline{x}) \\ \mid \text{Scheduler}_1 \\ \mid \text{Scheduler}_2\end{array}$$

ここで Scheduler_1 および Scheduler_2 はそれぞれ CONNECT.CFG における 1 行目および 3 行目の接続定義から以下のように得られる．

$$\begin{array}{l}\text{Scheduler}_1 = !(\ \text{on}(x).\text{end}. \\ \qquad \overline{\text{SensorObserver_Observe}}\ x \\ \qquad \mid \text{SensorObserver_Sensor}(x). \\ \qquad \text{end}.\overline{x}) \\ \text{Scheduler}_2 = !(\ \text{SensorObserver_Ready}(x).\text{end}. \\ \qquad \overline{\text{Display_DisplayData}}\ x \\ \qquad \mid \text{Display_Display}(x). \\ \qquad \text{end}.\overline{x})\end{array}$$

5.4 AIBO プログラム全体の変換

サンプルプログラム全体はプロセス AIBO_System として以下のように変換される☆．

$$\begin{array}{l}\text{AIBO_System} = \\ (\boldsymbol{\nu}\text{Display_Display Display_DisplayData} \\ \text{SensorObserver_Sensor Display_DisplayData} \\ \text{SensorObserver_Start SensorObserver_StartObserve}) \\ (\overline{\text{Display_Display}}\ \text{Display_DisplayData}. \\ \overline{\text{SensorObserver_Sensor}}\ \text{SensorObserver_Observe}. \\ \overline{\text{SensorObserver_Start}}\ \text{SensorObserver_StartObserve}. \\ \overline{\text{end}} \\ \mid \text{Scheduler} \mid \text{Display} \\ \quad \mid \text{SensorObserver} \mid \text{Sensor})\end{array}$$

ここでプロセス Sensor は以下のように定義されるプロセスである．

$$\text{Sensor} = !(\text{sensor}.\overline{\text{on}}\ \text{sensor}.\overline{\text{end}})$$

OPEN-R ではオブジェクトとセンサの間のメッセージ通信はオブジェクト間におけるメッセージ通信と同様に非同期な ready-notify メッセージ通信のプロトコルで行われる．プロセス Sensor はメソッドから変換されるプロセスの特別な形として得られる．

5.5 Mobility Workbench によるデッドロックフリー解析

AIBO プログラムに対する π 計算のプロセスを Mobility Workbench の入力形式として記述しデッドロックフリー解析を行った．ここで Mobility Workbench では複製を表す演算が存在しないため，以下のように複製プロセスをプロセスの再帰で表す必要がある．

自由な名前を持たない場合，Q をプロセス定数として入力プレフィクスを持たない Scheduler 以外の複製プロセス $!P$ を，$Q = \{\{P\}\}$ と定義する．ここで変換 $\{\{\ \}\}$ は以下のように定義される．

$\{\{P_1 \mid P_2\}\} = \{\{P_1\}\} \mid \{\{P_2\}\}$

$\{\{\boldsymbol{\nu} z P\}\} = \boldsymbol{\nu} z \{\{P\}\}$

π をアクションとするとき，

$\{\{\pi.P\}\} = \pi.\{\{P\}\} \qquad (\pi \neq \overline{end})$

$\{\{P_1 + P_2\}\} = \{\{P_1\}\} + \{\{P_2\}\}$

☆ 完全な記述は以下の URL でアクセスできる．
http://www.agusa.i.is.nagoya-u.ac.jp/person/sue/download/mwb/aibo.mwb

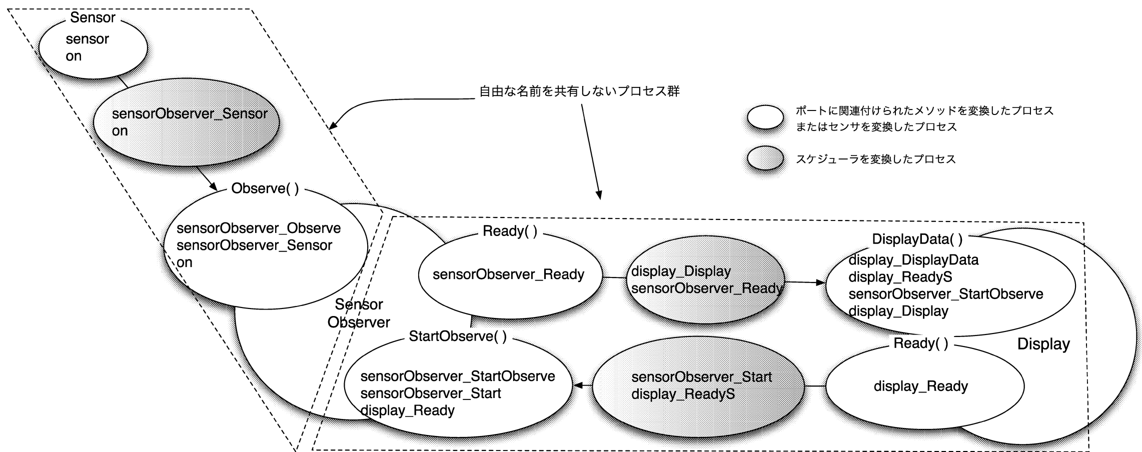

図 7　サンプルプログラムを変換して得られるそれぞれのプロセスにおける自由な名前
Fig. 7　Free names of processes translated from a sample program.

$\{\{\overline{end}.0\}\} = \overline{end}.Q$
AIBO プログラムの変換の定義より，複製プロセス $!P$ のサブプロセスとして複製プロセスは現れない．

Scheduler 以外のプロセスの複製は動作 $\overline{end}$ で終了する．Scheduler については，$\{\{0\}\}$=Scheduler とする．Scheduler によってオブジェクト同期に関するメソッドの動作が唯一起動される．AIBO プログラムから変換されたプロセスのすべてのサブプロセス $!P$ について，$fn(P) \cap \overline{fn(P)} = \emptyset$ である．したがって変換 $\{\{\ \}\}$ によって変換されたプロセスがデッドロックしないとき，変換前の複製演算を持つプロセスもデッドロックしない．

図 7 に Mobility Workbench 記述におけるプロセスの概要を示す．図 7 において楕円はポートに関連付けられたメソッドを変換したプロセス，センサを変換したプロセスもしくはスケジューラを変換したプロセスである．楕円内の文字列は各プロセスにおいて end 以外の自由な名前である．2 つの並行オブジェクトと 3 つのメッセージ通信リンクを持つサンプルプログラムは 9 つのプロセスとして変換された．

4.2.2 項の方法で図 7 の破線のように 2 つのプロセス群に分割した．2 つのプロセス群は名前 end のみを共通の名前として含む．本論文の変換によって得られる π 計算のプロセスにおいて，名前 end はあるメソッドが起動信号を受信するとブロックされる．メソッドはそのメソッドの動作の終了時に end を送信する．この振舞いは分割されたプロセス群でも同様である．したがって，分割されたそれぞれのプロセスについて end がブロックされ続けないならば，もとの全体プロセスについても end はブロックされ続けることはないため，補題 2 の手法を適用してもデッドロックしない性質は保たれる．

CPU 3.2 GHz，メモリ 1 GB の計算機で直接デッドロックフリー解析を行ったところ状態爆発のために解析を行うことができなかった．分割された 2 つのプロセス群に対してデッドロックフリー解析を行ったところ，それぞれ **15.1 秒**，**229.3 秒**で解析可能となった．

5.6　考　　察

本節では，本論文における π 計算の適用可能性について考察する．

5.6.1　ソースコードと変換で得られたプロセスの対応

変換においては OPEN-R 並行オブジェクトのソースコードの構造を保つことを目標とした．π 計算による変換では名前渡しによりメソッドの呼び出しについて，そのソースコードの構造を保存して変換することができる．たとえばスケジューラプロセスのサブプロセスとして $sbj_x(a).end.\overline{a}$ が存在し，アクション $\nu x\ \overline{sbj_x}\ x$ を持つプロセスが n 個存在するとする．CCS [6] のような名前渡しを行うことができない通信プロセスモデルで変換する場合，送信アクションを区別するために $\nu x\ \overline{sbj_x_1}\ x$，$\nu x\ \overline{sbj_x_2}\ x$，$\cdots$，$\nu x\ \overline{sbj_x_n}\ x$ とし，それらに対応するスケジューラのサブプロセスとして $sbj_x_1(a).end.\overline{a}$，$sbj_x_2(a).end.\overline{a}$，$\cdots$，$sbj_x_n(a).end.\overline{a}$ のように可能な組合せをすべて列挙する必要があり，CONNECT.CFG におけるそれぞれの接続定義に対して 1 対 n の対応となる．π 計算では名前渡しが可能であるため CONNECT.CFG における定義と 1 対 1 の対応をとることができる．この点から π 計算はソースコードの構造を保った変換に適している．

5.6.2　変換の妥当性

本論文における AIBO プログラムの π 計算への変換は，AIBO プログラムの実行におけるデッドロックが多くの場合非同期通信プロトコルの意図しない順序によって発生することに着目して定義した．本論文の変換では，ready 信号，notify 信号を発生する文をそれぞれ送信アクションとして変換する．メソッドは起動信号を表すアクションを受信し，ソースコードのオブジェクト同期に関する動作を保存し，制御としてより非決定的に振る舞うように変換し，さらにメソッドの終了信号を表すアクションを送信するプロセスとして変換する．スケジューラは π 計算における非決定的な意味付けによる一般的振舞いを行う．π 計算のプロセスにおける τ 遷移は各種信号の送受信に対応し，非同期通信プロトコルの振舞いを模倣する．この点で本論文における変換で得られる π 計算のプロセスの τ 遷移は実際のオブジェクトの同期についての振舞いをすべてカバーしている．したがって本論文の変換で得られた π 計算のプロセスが τ 遷移による振舞いにおいてデッドロックしないとき，対象となる AIBO プログラムはその振舞いにおいてデッドロックしない．

6. 関連研究

Alur らは階層的ハイブリッドモデル[2] から AIBO プログラムのコード生成を行った[3]．連続量と不変条件が階層的ハイブリッドモデルに記述される．Alur らは到達可能性検証によって階層的ハイブリッドモデル上で不変条件違反が発生しないことを示し，逐次的に動作するプログラムコードを生成する手法を示した．

本研究ではソースコードを直接検証の対象として扱い，AIBO プログラムの振舞いを表すモデルとして π 計算を用い，検証の対象をデッドロックフリー性の検査としている．

7. おわりに

本論文では OPEN-R AIBO ロボットプログラムを通信プロセスモデルによって ready-notify プロトコルに着目してデッドロックを検出するためのメッセージフローの解析を行った．同期に関するオブジェクト全体の振舞いを抽象的に与えることで，実行のタイミングに起因する致命的なデッドロックの発生を事前に検出することが可能となった．

一般に並行システムでは発生する状態が爆発的に増加する．しかし AIBO プログラムではインタリーブされるオブジェクトが多いことに着目すると，デッドロックフリー解析は分割されたそれぞれのプロセスに対して行うことが可能となる．通信ポートの組合せを調べることで比較的容易に分割検証を行うことができる．

本研究の変換はソースコードの構造を保存する．モデル上でデッドロックが検出されたとき遷移を追跡することで原因となったコンポーネントを特定できる．

一般に資源の少ない組み込みシステムでは AIBO プログラムと同様に非同期なメッセージ通信によって制御が行われる．このような組み込みシステムに対して本手法と同様な変換定義を与えることによってデッドロックフリー性の解析が可能となる．

より詳細に通信の状況を静的解析することによって柔軟な分割手法を提案することは今後の課題である．4 章において単純な場合として名前 ‘end’ はコンポーネント間のまとまった通信をブロックしないという特性のために分割の基準から除外できることを説明した．このように，ある特性を名前の持つ型として定式化し，変換時に型付けすることで静的に分割可能性を解析できる．また，一連の通信に対してセッション型[4] を導入することによって互いに通信が干渉しないことを示し，セッション単位で分割できる場合がある．型の導入は今後の課題である．

さらに，デッドロックフリー性以外の性質について分割した証明手法を示すことは今後の課題である．本手法では，プログラムの抽象化の点においてすでにデッドロックフリー性に依存した抽象化を行っている．デッドロックフリー性以外の性質を同様の手法で示すためには，その性質に依存したプログラムの抽象化が必要である．

謝辞　本研究を進めるにあたり，多くの助言をくださり，様々な相談にのっていただいた名古屋大学阿草・結縁研究室の皆様ならびに有益なコメントをくださった査読者の皆様に感謝いたします．本研究の一部は，科学研究費補助金基盤研究（B）17300006，基盤研究（C）16500027，19500026，栢森情報科学振興財団および文部科学省 21 世紀 COE プログラム「社会情報基盤のための音声・映像の知的統合」の助成による．

参考文献

1) OPEN-R SDK. http://open-r.aibo.com
2) Alur, R., Dang, T., Esposito, J., Hur, Y., Ivan, F., Kumar, V., Lee, I., Mishra, P., Pappas, G. and Sokolsky, O.: Hierarchical modeling and analysis of embedded systems (2002).
3) Alur, R., Ivancic, F., Kim, J., Lee, I. and Sokolsky, O.: Generating Embedded Software from Hierarchical Hybrid Models, *ACM SIG-*

PLAN Notices*, *Proc. 2003 ACM SIGPLAN conference on Language, compiler and tool for embedded system* (2003).
4) Honda, K., Vasconcelos, V.T. and Kubo, M.: Language Primitives and Type Discipline for Structured Communication-Based Programming, *ESOP '98: Proc. 7th European Symposium on Programming*, London, UK, pp.122–138, Springer-Verlag (1998).
5) Milner, R.: *Communicating and Mobile Systems: The π-Calculus*, Cambridge University Press (1999).
6) Milner, R.: *Communication and concurrency*, Prentice Hall International (UK) Ltd., Hertfordshire, UK (1995).
7) Sangiorgi, D. and Walker, D.: *The π-calculus: A Theory of Mobile System*, Cambridge University Press (2001).
8) Victor, B. and Moller, F.: The Mobility Workbench—A Tool for the π-Calculus, *CAV '94: Computer Aided Verification*, Dill, D. (Ed.), Lecture Notes in Computer Science, Vol.818, pp.428–440, Springer-Verlag (1994).
9) Weiser, M.: Program Slicing, *the Fifth International Conference on Software Engineering*, pp.536–545 (1981).

(平成 19 年 1 月 9 日受付)
(平成 19 年 6 月 5 日採録)

末次　　亮（学生会員）

2004 年名古屋大学工学部電気電子・情報工学科卒業．2006 年同大学大学院情報科学研究科情報システム学専攻博士前期課程修了．現在，同大学院情報科学研究科情報システム学専攻博士課程後期課程在学中．並行計算モデルに基づく形式的検証手法の組み込みシステムへの適用に興味を持つ．

結縁　祥治（正会員）

1990 年名古屋大学大学院博士課程満了．名古屋大学大学院工学研究科助手，1998 年同情報メディア教育センター助教授を経て，現在，同大学院情報科学研究科教授．博士（工学）．並行計算モデルのソフトウェアへの応用面の観点で，通信プロセスモデルの研究に従事．形式的な手法によるモデル化に基づくソフトウェア検証に興味を持つ．

阿草　清滋（正会員）

1947 年生．1970 年京都大学工学部電気工学第二学科卒業．1972 年同大学大学院工学研究科電気工学第二専攻修士課程修了．同博士課程へ進学．1974 年より同情報工学科助手．同講師，助教授を経て 1989 年より名古屋大学教授．現在，同大学院情報科学研究科教授．工学博士．専門分野はソフトウェア工学，ソフトウェア開発方法論，知的開発環境，ソフトウェアデータベース，仕様化技法，再利用技法，マンマシンインタフェース．電子情報通信学会，ソフトウェア科学会，IEEE，ACM 各会員．