ソフトウェア工学 138−6
（2002．7．11）

組み込みシステム設計における並行正規表現を用いたスレッド抽出法の適用

岡崎 光隆† 青木 利晃†,†† 片山 卓也†
† 北陸先端科学技術大学院大学 情報科学研究科
†† 科学技術振興事業団 さきがけ 21

現在，組み込みシステム開発へのオブジェクト指向技術の適用研究が盛んである．しかし，組み込みシステム特有の問題である，厳しい時間や資源の制約を充足するためには，既存のオブジェクト指向設計技術では十分に対処する事ができない．この問題を解決する手法として，システムの設計をスレッド中心に行う手法がいくつか提案されている．しかし，これらの手法では，分析モデルからスレッドを抽出する具体的な方法が明らかにされていないという問題があった．我々はこれまでに，スレッド抽出問題の解決策として，並行正規表現を用いた手法を提案した．本稿では，この手法を PCM デバイスドライバ開発に適用し，分析モデルの作成，設計モデルの獲得および実装を行った事例を紹介する．

Applying Concurrent regular expression to extract threads in embedded system design

Mitsutaka Okazaki† Toshiaki Aoki†,†† Takuya Katayama†
†Japan Advanced Institute of Science and Technology
††PRESTO Japan Science and Technology Corporation

Recently, object-oriented techniques are going to be adopted in embedded software developments. In embedded software domains, we have to consider non-functional requirements such as real-time properties and resource requirements. To deal with such requirements, we must design a target system in terms of threads. In this design, we need to extract threads from concurrent objects in an analysis model, however the current methodologies do not follow it enough. To solve this problem, we had proposed a formal approach to extract threads from concurrent objects. In this paper, we introduce an experimental application of this approach in PCM device driver development.

1. はじめに

近年の組み込みシステム機器の大規模・複雑化に伴い，オブジェクト指向開発の組み込みシステム開発への適用が期待されている．しかし，既存のオブジェクト指向技術を直接適用した場合，組み込みシステム特有の問題を解決する事が困難である．ここで，組み込みシステム特有の問題とは，リアルタイム性や資源競合などの非機能的な制約を指す．既存のオブジェクト指向設計技術は，厳しい非機能制約を扱う方法を提供していない．

オブジェクト指向開発では，オブジェクトを中心とした観点でシステムを分析し，システムの論理的な振る舞いをモデル化する．このモデル化では，システム全体が，並行に動作するオブジェクトの集合として表現される．ここで，エンタープライズシステムのような，緩やかな非機能制約を持つシステムを仮定すると，分析モデルに出現するオブジェクトを，直接，並行動作するソフトウェアモジュールに対応付けて実装することができる．しかし，組み込みシステムのように，厳しい非機能制約の充足が求められるシステムでは，システムの動作系列，すなわちスレッドに注目した解析を行うことが重要である．

現在，スレッドを中心とした設計モデルを作成し，このモデル上で非機能制約の充足問題を解決する開発法がいくつか提案されている．例えば OCTOPUS 法や[6]，SES アプローチ[3] などがある．しかし，これらの開発法はスレッド抽出後の問題解決に重点が置かれており，オブジェクト中心の分析モデルから，スレッド中心の設計モデルを導出する方法が明確にされていない．この問題に対し，我々は，並行正規表現[2] を用いてモデルの形式化を行い，等価変換の公理系によって，オブジェクト中心のモデルからスレッド中心のモデルへ変換する手法を提案した[1]．しかし，この手法を適用した開発実験が行われておらず，手法の適用可能性や，スレッド中心のモデルと実装との対応関係などが明らかになっていなかった．

本稿では，我々の提案した変換手法を用いて，開発

実験を行った結果を紹介する．この実験の目的は次の通りである．

- 実際の開発に，モデル変換手法を適用し，設計モデルを獲得できる事を示す．
- 獲得した設計モデルを元に，システムの実装が可能である事を示す．

本稿の構成は次の通りである．まず 2 章で，分析モデルと設計モデルを形式化する手法を述べる．3 章でモデル変換のための公理系について述べる．4 章で，PCM デバイスドライバ開発実験の結果を紹介する．5 章で実験結果についての考察を行い，6 章で本稿を統括する．

2. モデルの形式化

モデルを厳密に対応付けるために，本研究では分析・設計の各モデルを形式的に定義している．本節では，それらのモデルの形式化手法について述べる．

本研究では，分析モデルとして，複数のオブジェクトが相互に同期通信を行いながら並行に動作するモデルを仮定する．これに対し設計工程では，複数のスレッドが並行に動作するモデルを用いる．

オブジェクトとスレッドは共に並行動作の主体であるが，それらの間で相互通信を認めるかどうかが異なる．分析モデルではオブジェクトは相互に通信を取り合って協調動作する．これに対し，設計モデルではスレッド間の通信は一切発生しない．

我々は，分析・設計モデルを統一的に扱う体系として，並行正規表現を用いる．並行正規表現[2] は，正規表現に対して並行性と同期通信の概念を拡張した表現である．この拡張は，正規表現に 2 つの演算子，$\|$ と $[\,]$ を導入することで実現されている．

2.1 アクション

本研究では，システムの振る舞いをアクション列の集合で抽象する．アクションとはシステムで行われる最も原始的な処理の単位である．アクションの意味を厳密に定義する事は難しいが，一般には，実世界に発生するイベントやメソッドの呼び出し，プログラムコードの断片等がアクションに対応付けられる．構文的にはアクションは単に識別子としての文字列で定義される．

定義 2.1 (アクション)
小文字で始まる文字列をアクションとする．例えば $\mathtt{a}, \mathtt{b}, \mathtt{c}\ldots$ や $\mathtt{open}, \mathtt{close}, \mathtt{post}, \mathtt{get}\ldots$ 等はアクションである．また，アクションの集合を Σ で記述する．

アクションが発生する時間順序に沿って，アクションを並べたものをアクション例と呼ぶ．例えばあるシステムの動作を観測した結果，3 つのアクション $\mathtt{login}, \mathtt{work}, \mathtt{exit}$ が連続して発生する事が分かったとする．この時，システムの振る舞いはアクション列 $\mathtt{login} \cdot \mathtt{work} \cdot \mathtt{exit}$ で表現される．

システムの振る舞い全体はアクション列の集合で表現される．例えば，図 1 の状態遷移図で振る舞いが定義されたシステムについて考える． このシステムは，

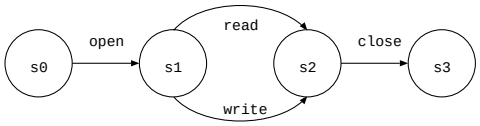

図 1 状態遷移図

初期状態 s0 から，アクション open の後で，状態 s1 に遷移する．さらにアクション read もしくは write のいずれかの後に状態 s2 に遷移し，close の後に状態 s3 に達する．このような振る舞いをアクション列で表現すると，$\mathtt{open} \cdot \mathtt{read} \cdot \mathtt{close}$ および $\mathtt{open} \cdot \mathtt{write} \cdot \mathtt{close}$ の 2 つのアクション列になる．このシステムの振る舞い全体は，これらのアクション列の集合で次のように表す．$\{\mathtt{open} \cdot \mathtt{read} \cdot \mathtt{close}, \mathtt{open} \cdot \mathtt{write} \cdot \mathtt{close}\}$

システムの振る舞いをアクション列の集合として直接記述すると，記述量が多くなり，視認性も悪くなる．本研究では，アクション列の集合の記述に，並行正規表現を用いる．並行正規表現は，対応するアクション列の集合を表現する能力に関して，状態遷移図と等価である．このため，オブジェクト指向分析で振る舞いの表現として用いられる状態遷移図の，形式的記述法としても適当である．

2.2 並行正規表現

本研究では，システムの振る舞いを，アクションの有限集合 Σ 上の並行正規表現で記述する．ここで，Σ 上の並行正規表現とは，$\{\epsilon, \bot\} \cup \Sigma$ に対し，和 $(+)$，連結 $(.)$，閉包 $(*)$，並行 $(\|)$，同期複合 $([\,])$ の 5 種の演算を，有限回行う事によって得られる表現である．ただし，$\bot$ は要素を一つも含まない集合，ϵ は空列を表わす．並行正規表現の構文を以下に定義する．

定義 2.2 (並行正規表現)

1. $\mathtt{c} \in \Sigma \cup \{\bot, \epsilon\}$ ならば c は並行正規表現である．
2. P と Q が並行正規表現ならば，$\mathtt{P}+\mathtt{Q}, \mathtt{P}.\mathtt{Q}, \mathtt{P}^*$ も並行正規表現である．
3. P と Q が並行正規表現かつ $\mathtt{S} \subseteq \Sigma$ ならば，$\mathtt{P}[\mathtt{S}]\mathtt{Q}$ は並行正規表現である．
4. P と Q が並行正規表現ならば $\mathtt{P}\|\mathtt{Q}$ は並行正規表現である．

この並行正規表現の定義は，オリジナルの定義と比較して，$[\,]$ 演算子を $[\mathtt{S}]$ のように拡張している．これは 3 節で紹介する等価変換の公理系において，$[\,]$

演算子の分配法則を実現する目的で導入した拡張である．なお，$S = \phi$ の時は $[S]$ の S を省略して $[]$ と表記する．また，演算子の結合優先順位は $*, ., +, \|, []$ の順で高いものとする．結合順位が分かりにくい場合は，適宜括弧を付けて表現する．

2.3 オブジェクトとスレッド

本研究では，オブジェクトとスレッドは内部的な並行性を持たないものとし，それぞれの振る舞いを正規表現を使って記述する．

正規表現とは，並行正規表現のサブセットであり，和 $(+)$，連結 $(.)$，閉包 $(*)3$ 種の演算のみを使った表現である．ここで定義する正規表現は一般的な正規表現の定義と等しい．

正規表現 P が表現するアクション列の集合を L(P) と表記する．L(P) は正規表現の言語とも呼ばれる．L(P) の定義は以下の通り．

定義 2.3 (正規表現の言語)
P，Q をそれぞれ Σ 上の正規表現とする．このとき，言語 L(P) は以下のように再帰的に定義される．P^i は P の i 回の連結を意味し，$P^0 = \epsilon$ である．

1. $L(\epsilon) = \{\}$, $L(\perp) = \phi$, $L(a) = \{a\}$
2. $L(P.Q) = \{a \cdot b | a \in L(P), b \in L(Q)\}$
3. $L(P + Q) = L(P) \cup L(Q)$
4. $L(P^*) = \bigcup_{i=0,1,\ldots} L(P^i)$

例えば，図 1 の状態遷移図が意味する振る舞いは，正規表現では open.(read+write).close と記述できる．

2.4 同期複合演算子

その振る舞いがそれぞれ正規表現 A，B で定義された 2 つのオブジェクトを考える．これらのオブジェクトが相互に通信しながら並行動作する事を，並行正規表現では，A [S] B と表記する．[S] は**同期複合演算子**と呼ばれ，S はアクションの集合である．特に，$S = \phi$ の時は S を省略して [] と表記する．言語 L(A [S] B) = の定義を以下に示す．

$\{w | w \in (\Sigma_A \cup \Sigma_B)^*, w/(\Sigma_A \cup S) \in L(A), w/(\Sigma_B \cup S) \in L(B)\}$

ここで，Σ_A は式 A に含まれるアクションの集合を意味し，w/Σ は w 中のアクションを Σ に含まれるアクションのみに制限する事を意味する．例えば，$a \cdot c \cdot a \cdot d \cdot a \cdot b/\{a, c\} = a \cdot c \cdot a \cdot a$ である．

同期複合演算では，2 つのオブジェクトに共通して含まれる記号を同期通信の表現であるとみなす．同期通信とは，お互いの処理が完了するまで，双方のオブジェクトが共に待機する通信方法である．また，同期通信は，通信相手に無視される事があってはいけない．同期複合演算子を含む式の言語の例を，次に示す．

$L(a \,[]\, a) = \{a\}$

$L(a.x.c \,[]\, a.y.c) = \{a \cdot x \cdot y \cdot c, a \cdot y \cdot x \cdot c\}$

2.4.1 分析モデルの形式化

システムの分析モデルは，並行オブジェクト式で表現する．並行オブジェクト式は，$\Sigma \cup \{\epsilon, \perp\}$ に対し，和，連結，閉包，同期複合の 4 つの演算演算子から成る並行正規表現である．振る舞いがそれぞれ $O_1, O_2, \ldots O_n$ で定義された n 個のオブジェクトがあるとする．これらのオブジェクトが相互通信しながら並行動作するシステムは，並行オブジェクト式 $O_1 [] O_2 [] \ldots [] O_n$ で記述される．なお，ここでは，同期複合演算子に結合法則が成り立つので括弧を省略している．

2.5 並行演算子

振る舞いがそれぞれ正規表現 A，B で定義された 2 つのスレッドが，互いに独立に並行動作するシステムを，並行正規表現を用いて，A‖B と表記する．ここで ‖ を**並行演算子**と呼び，その言語の定義は以下の通りである．

$$L(A \| B) = \begin{cases} L(A \,[]\, B) & \text{if} \quad \Sigma_A \cap \Sigma_B = \phi \\ \phi & \text{else} \end{cases}$$

2.6 設計モデルの形式化

システムの設計モデルは，並行スレッド式で記述される．並行スレッド式は $\Sigma \cup \{\epsilon, \perp\}$ と，和，連結，閉包，並行の 4 つの演算子のみから成る並行正規表現である．例えば，式 a.(b‖c).d は並行スレッド式であり，その言語は $\{a \cdot b \cdot c \cdot d, a \cdot c \cdot b \cdot d\}$ である．

3. モデル変換

分析モデルから設計モデルへの変換は，分析モデルの振る舞いを表現する並行オブジェクト式を，それと全く等しい振る舞いを持つ並行スレッド式に変換することである．本研究では，この変換を式の等価変換の公理系を用いて達成する．

3.1 等価変換の公理系

本稿で使用する公理を表 1 に示す．ただし，この表に挙げた公理の他に，正規表現の代数的性質として一般に知られている法則も公理とみなすが，誌面の都合で割愛している．表中の A, B, ... は任意の並行正規表現を，x, y, ... は単一のアクションを示す．また，Σ_A は A に出現する全てのアクションの集合を意味する．

公理系には，公理の適用規則と循環解決規則の 2 つの規則がある．式 A に規則を 1 回適用した結果，式 B を得ることを $A \Leftrightarrow B$，0 回以上適用した結果，式 B を得ることを $A \overset{*}{\Leftrightarrow} B$ と書くことにする．

(公理の適用規則) 式 A, B, P に対し，公理より A = B が言えるならば，$P \Leftrightarrow P_{(A/B)}$ または $P \Leftrightarrow P_{(B/A)}$ ただし，$P_{(X/Y)}$ は，式 P 中の X の出現を Y に置換した式を意味する．

同一性	$A\ [S]\ A = A$
零元	$A\ [S] \perp = \perp$
単位元	$S \cap \Sigma_A = \phi$ならば, $A\ [S] \epsilon = A$
交換法則	$A\ [S]\ B = B\ [S]\ A$
結合法則	$(A \square B) \square C = A \square (B \square C)$
分配法則	$(A + B)\ [S]\ C =$ $(A\ [S \cup (\Sigma_B \cap \Sigma_C)]\ C) + (B\ [S \cup (\Sigma_A \cap \Sigma_C)]\ C)$
同期	$(x.B)\ [S]\ (x.C) = x.(B\ [S \cup \{x\}]\ C)$
同期失敗	$x, y \in S \cup (\Sigma_x.A \cap \Sigma_y.B)$ かつ $x \neq y$ ならば, $x.A\ [S]\ y.B = \perp$
展開	$x \neq y$ かつ $x \notin S \cup \Sigma_B$かつ $y \notin S \cup \Sigma_A$のとき, $x.A\ [S]\ y.B = x.(A\ [S]\ y.B) + y.(x.A\ [S]\ B)$
スレッド抽出	$x, y \in (\Sigma_{x.B} \cap \Sigma_{y.D}) \cup S$ かつ $(\Sigma_{A.x.B} \cup S) \cap \Sigma_C = (\Sigma_{C.y.D} \cup S) \cap \Sigma_A = \phi$ならば $(A.x.B)\ [S]\ (C.y.D) = (A \square C).(x.B)\ [S]\ (y.D)$
最適化	$A\ [S]\ B = A\ [S \cap (\Sigma_A \cup \Sigma_B) \cap (\overline{\Sigma_A \cap \Sigma_B})]\ B$
スレッド化	$\Sigma_A \cap \Sigma_B = \phi$ならば, $A \square B = A \| B$

表 1 等価変換の公理

(循環解決規則) 式 S, A, B に対し，$S \overset{*}{\Leftrightarrow} A.S + B$ ならば $A.S + B \Leftrightarrow A^*.B$

提案した公理系は健全であり，すなわち，$P \overset{*}{\Leftrightarrow} P'$ ならば，$L(P) = L(P')$ が成立する．並行オブジェクト式 O の等価変換を繰り返すことで，$L(O) = L(T)$ なる並行スレッド式 T を獲得する事ができる．

4. PCM デバイスドライバの開発

PCM デバイスドライバの開発に，公理系によるモデルの変換手法を適用する実験を行った．開発した PCM デバイスドライバは，リアルタイムに波形合成演算を行い，単一の D/A コンバータから複数チャンネルの PCM 音声を出力するソフトウェアである．チャンネルとは，周波数と音量の制御が可能な PCM 音声ストリームの抽象である．

4.1 実装環境

ドライバを実装する環境は，中央演算装置と以下のハードウェアで構成されているものとする．

- ハードウェアクロック 1 基
- D/A コンバータ 1 基

さらに，以下の機能を持つ OS が動作していると仮定する．

- I/O アクセス
- 割り込み通知
- セマフォ

I/O アクセスはドライバから D/A コンバータへのデータ書き出しに使用する．また，OS の割り込み通知機能により，ドライバはクロック割り込みをイベントとして検知する．セマフォは同期処理の実現に使用される．ドライバの実装言語には C 言語を用いた．

4.2 ドライバ仕様

ドライバソフトウェアに対する要求仕様を次に示す．

- 最大 3 チャンネルの PCM 音声を再生可能
- チャンネル毎に異なる再生周波数を設定可能
- チャンネル毎に異なる再生音量を設定可能
- 音声再生中でも周波数と音量を動的に変更可能

ドライバの API 仕様を表 2 に示す．

エントリ	機能	引数
PLAY	再生開始	チャンネル番号 再生データ先頭アドレス 再生データ終了アドレス
STOP	再生中止	チャンネル番号
FREQ	周波数指定	チャンネル番号, 周波数
VOL	音量変更	チャンネル番号, 音量

表 2 API 一覧

4.3 分析工程

4.3.1 クラスの定義

仕様に基いてシステムを分析し，作成したクラスを図 2 に示す．

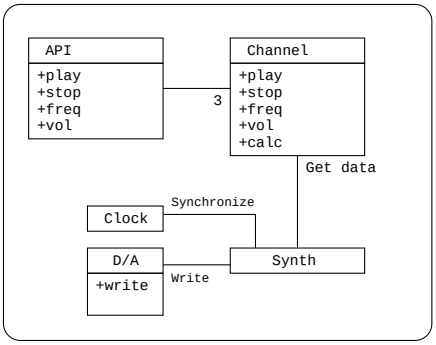

図 2 クラス図

クラス API はドライバの API エントリを表現するクラスである．API の各メソッドはそれぞれ API のエントリと一対一で対応する．

クラス Channel は個々のチャンネルを表現するクラスである．play,stop,freq,vol はそれぞれ，そのチャンネルの PCM 音声について，再生開始，停止，周波数変更，音量変更を行うメソッドである．calc は，そのチャンネルが出力する PCM 波形を計算するメソッドである．一回の呼び出し毎に，1 クロックに相当する時間分の波形を計算する．

Synth は各チャンネルの出力を合成する装置を表現するクラスである．また，Clock はハードウェアクロックと対応するクラスである．Synth クラスのインスタンスは，このクロックが生成する信号に同期して，D/A コンバータに書き込み処理を行う．

D/A は D/A コンバータを表現するクラスである．write は D/A コンバータに書き込みを行うメソッドである．書き込まれた値によって D/A コンバータの出力が変化する．

4.3.2 振る舞いの定義

オブジェクトの振る舞いを表 3 の通り定義した．オブジェクト API,SYN,CLK,DA はそれぞれクラス

オブジェクト	振る舞い
API	$(\mathtt{play}.(\mathtt{p}_0+\mathtt{p}_1+\mathtt{p}_2)+$ $\mathtt{stop}.(\mathtt{s}_0+\mathtt{s}_1+\mathtt{s}_2)+$ $\mathtt{freq}.(\mathtt{f}_0+\mathtt{f}_1+\mathtt{f}_2)+$ $\mathtt{vol}.(\mathtt{v}_0+\mathtt{v}_1+\mathtt{v}_2))^*$
$\mathtt{CH}_0$	$(\mathtt{p}_0+\mathtt{s}_0+\mathtt{f}_0+\mathtt{v}_0+\mathtt{calc})^*$
$\mathtt{CH}_1$	$(\mathtt{p}_1+\mathtt{s}_1+\mathtt{f}_1+\mathtt{v}_1+\mathtt{calc})^*$
$\mathtt{CH}_2$	$(\mathtt{p}_2+\mathtt{s}_2+\mathtt{f}_2+\mathtt{v}_2+\mathtt{calc})^*$
SYN	$(\mathtt{clk}.\mathtt{calc}.\mathtt{write})^*$
CLK	$\mathtt{clk}^*$
DA	$\mathtt{write}^*$

表 3 振る舞いの定義

API,Synth,Clock,D/A のインスタンスである．また，ドライバには最高 3 本の PCM 音声チャンネルの再生が要求されている．そこで，Channel クラスのインスタンスを $\mathtt{CH}_0,\mathtt{CH}_1,\mathtt{CH}_2$ の 3 つ作成し，それぞれの PCM 音声チャンネルに割り当てている．

アクション play,stop,freq,vol はそれぞれ API エントリの呼び出しに対応する．$\mathtt{p}_i,\mathtt{s}_i,\mathtt{f}_i,\mathtt{v}_i$ はそれぞれオブジェクト $\mathtt{CH}_i$ の play,stop,freq,vol メソッドの呼び出しと対応する．

calc は，$\mathtt{CH}_0,\mathtt{CH}_1,\mathtt{CH}_2,$SYN の全てで同期して行われるアクションである．アクション calc においては，オブジェクト $\mathtt{CH}_0,\mathtt{CH}_1,\mathtt{CH}_2$ がそれぞれのチャンネルの出力値を計算する．オブジェクト SYN はそれらの出力を合成し，D/A コンバータへの出力値を計算する．clk はクロック割り込みイベントの発生と対応し，write は D/A コンバータへの書き込み操作と対応する．

各オブジェクトが表現する振る舞いの直観的な意味は次の通り．

- API は，各 API エントリの呼び出しを受けると，それぞれのチャンネルに処理を委譲する．たとえば，API エントリ play が呼び出されると，いずれかのチャンネルの play メソッド $(\mathtt{p}_0,\mathtt{p}_1,\mathtt{p}_2)$ が呼び出される．
- $\mathtt{CH}_i$ は，演奏開始，演奏停止，周波数設定，音量設定，出力値の計算のいずれかの動作を繰り返し行う．
- SYN はクロックイベントに同期して (clk)，出力値を計算し (calc)，D/A コンバータに書き込みを行う (write)．
- CLK はクロックイベントの生成 (clk) を繰り返す．
- DA は書き込み操作 (write) を任意の回数受理する．

以上のオブジェクトは全て相互通信しながら並行動作する．システム全体の振る舞いは，以下の並行オブジェクト式で定義される．

$\mathtt{API}\ []\ \mathtt{CH}_0\ []\ \mathtt{CH}_1\ []\ \mathtt{CH}_2\ []\ \mathtt{SYN}\ []\ \mathtt{CLK}\ []\ \mathtt{DA}$

なお，ここではオブジェクト名を，対応する振る舞いの略記として用いている．たとえば，CLK [] DA は $(\mathtt{clk}^*\ []\ \mathtt{write}^*)$ を意味する．

4.4 設計工程

前節で分析した PCM デバイスドライバの分析モデルを元に，PCM デバイスドライバの設計を行う．まず，等価変換の公理系を用いて並行オブジェクト式の等価変換を行い，並行スレッド式を導出する．

4.4.1 準　　備

安直に公理を適用すると計算が爆発するので，定理を利用して計算量を削減する．まず，次の補題を証明する．

補題 4.1

$\bigcup_{i=0}^{n}\mathtt{a}_i \stackrel{def}{=} \mathtt{a}_0+\mathtt{a}_1+\ldots\mathtt{a}_n$ とする．

$\mathtt{P}=\bigcup_{i=0}^{n}\mathtt{a}_i, \mathtt{Q}=\bigcup_{i=0}^{m}\mathtt{b}_i, \Sigma_P\cap\Sigma_Q=\phi$ のとき，

$$\mathtt{P.X}\ []\ \mathtt{Q.Y} \stackrel{*}{\leftrightarrows} (\mathtt{P}.(\mathtt{X}\ []\ \mathtt{Q.Y}))+(\mathtt{Q}.(\mathtt{P.X}\ []\ \mathtt{Y}))$$

証明

P,Q の定義から，

$\mathtt{P.X}\ []\ \mathtt{Q.Y}=(\bigcup_{i=0}^{n}\mathtt{a}_i).\mathtt{X}\ []\ (\bigcup_{i=0}^{m}\mathtt{b}_i).\mathtt{Y}$

.,[] の分配法則を繰り返し適用して，

$\stackrel{*}{\leftrightarrows} (\bigcup_{i=0}^{n}\bigcup_{j=0}^{m}(\mathtt{a}_i.\mathtt{X}\ []\ \mathtt{b}_j.\mathtt{Y}))$

$\stackrel{*}{\leftrightarrows} (\bigcup_{i=0}^{n}\mathtt{a}_i.(\bigcup_{j=0}^{m}(\mathtt{X}\ []\ \mathtt{b}_j.\mathtt{Y})))+$
$(\bigcup_{j=0}^{m}\mathtt{b}_i.(\bigcup_{i=0}^{n}(\mathtt{a}_i.\mathtt{X}\ []\ \mathtt{Y})))$

$\stackrel{*}{\leftrightarrows} (\bigcup_{i=0}^{n}\mathtt{a}_i.(\mathtt{X}\ []\ (\bigcup_{j=0}^{m}\mathtt{b}_j).\mathtt{Y}))+$
$(\bigcup_{j=0}^{m}\mathtt{b}_i.((\bigcup_{i=0}^{n}\mathtt{a}_i).\mathtt{X}\ []\ \mathtt{Y}))$

$\equiv (\mathtt{P}.(\mathtt{X}\ []\ \mathtt{Q.Y}))+(\mathtt{Q}.(\mathtt{P.X}\ []\ \mathtt{Y}))$

定理 4.2

$\mathtt{P}=\bigcup_{i=0}^{n}\mathtt{a}_i, \mathtt{Q}=\bigcup_{i=0}^{m}\mathtt{b}_i$ かつ $\Sigma_P\cap\Sigma_Q=\phi$ のとき，

$$(\mathtt{P}^*[]\ \mathtt{Q}^*) \stackrel{*}{\leftrightarrows} (\mathtt{P}+\mathtt{Q})^*$$

証明

$\mathtt{P}^*[]\ \mathtt{Q}^* \stackrel{*}{\leftrightarrows} (\epsilon+\mathtt{P}.\mathtt{P}^*)\ []\ (\epsilon+\mathtt{Q}.\mathtt{Q}^*)$

$\stackrel{*}{\leftrightarrows} \epsilon+\mathtt{P}.\mathtt{P}^*+\mathtt{Q}.\mathtt{Q}^*+(\mathtt{P}.\mathtt{P}^*[]\ \mathtt{Q}.\mathtt{Q}^*)\ldots(1)$

ここで，補題より，

$(\mathtt{P}.\mathtt{P}^*[]\ \mathtt{Q}.\mathtt{Q}^*) \stackrel{*}{\leftrightarrows} (\mathtt{P}.(\mathtt{P}^*[]\ \mathtt{Q}.\mathtt{Q}^*)+\mathtt{Q}.(\mathtt{P}.\mathtt{P}^*[]\ \mathtt{Q}^*))$

であることを利用すると，

$\stackrel{*}{\leftrightarrows} \mathtt{P}.\mathtt{Q}.\mathtt{Q}^*+\mathtt{Q}.\mathtt{P}.\mathtt{P}^*+(\mathtt{P}+\mathtt{Q}).(\mathtt{P}.\mathtt{P}^*[]\ \mathtt{Q}.\mathtt{Q}^*)$

この結果を (1) に適用すると，

$\epsilon+\mathtt{P}.\mathtt{P}^*+\mathtt{Q}.\mathtt{Q}^*+(\mathtt{P}.\mathtt{P}^*[]\ \mathtt{Q}.\mathtt{Q}^*)$

$\stackrel{*}{\leftrightarrows} \epsilon+(\mathtt{P}+\mathtt{Q}).(\epsilon+\mathtt{P}.\mathtt{P}^*+\mathtt{Q}.\mathtt{Q}^*+(\mathtt{P}.\mathtt{P}^*[]\ \mathtt{Q}.\mathtt{Q}^*))$

循環解決規則を適用し，

$\epsilon+\mathtt{P}.\mathtt{P}^*+\mathtt{Q}.\mathtt{Q}^*+(\mathtt{P}.\mathtt{P}^*[]\ \mathtt{Q}.\mathtt{Q}^*) \stackrel{*}{\leftrightarrows} (\mathtt{P}+\mathtt{Q})^*$

ゆえに，$(\mathtt{P}^*[]\ \mathtt{Q}^*) \stackrel{*}{\leftrightarrows} (\mathtt{P}+\mathtt{Q})^*$

定理 4.3

$a \neq b, b \notin \Sigma_X$ とすると，

$(a.b + X)^*[]\ b^* \equiv (a.b + X)^*$

証明

$(a.b + X)^*[]\ b^*$

$\overset{*}{\leftrightarrow}\ (\epsilon + (a.b + X).(a.b + X)^*)\ []\ (\epsilon + b.b^*)$

$\overset{*}{\leftrightarrow}\ \epsilon + ((a.b + X).(a.b + X)^*[b]\epsilon) + (\epsilon[b]\ b.b^*)((a.b + X).(a.b + X)^*[b]\ b.b^*)$

$\overset{*}{\leftrightarrow}\ \epsilon + (a.b.(a.b + X)^*[b]\epsilon) + (X.(a.b + X)^*[b]\epsilon) + \bot + (a.b.(a.b + X)^*[b]\ b.b^*) + (X.(a.b + X)^*[b]\ b.b^*)$

ここで，

$(a.b.(a.b + X)^*[b]\epsilon) \overset{*}{\leftrightarrow} a.(b.(a.b + X)^*[b]\epsilon) \overset{*}{\leftrightarrow} \bot$

だから，

$\overset{*}{\leftrightarrow}\ \epsilon + \bot + a.b.((a.b + X)^*[b]\ b^*) + X.(((a.b + X)^*[b]\epsilon) + ((a.b + X)^*[b]\ b.b^*))$

ここで，[] 演算子の分配法則より，

$((a.b + X)^*[b]\epsilon) + ((a.b + X)^*[b]\ b.b^*)$

$\overset{*}{\leftrightarrow}\ ((a.b + X)^*[b]\ (\epsilon + b.b^*) \overset{*}{\leftrightarrow} ((a.b + X)^*[b]\ b^*)$ だから，

$\overset{*}{\leftrightarrow}\ \epsilon + a.b.((a.b + X)^*[b]\ b^*) + X.(((a.b + X)^*[b]\ b^*))$

$\overset{*}{\leftrightarrow}\ \epsilon + (a.b + X).((a.b + X)^*[b]\ b^*)$

$\overset{*}{\leftrightarrow}\ \epsilon + (a.b + X).((a.b + X)^*[]\ b^*)$

循環解決規則を適用し，

$((a.b + X)^*[]\ b^*) \overset{*}{\leftrightarrow} (a.b + X)^*$

4.4.2 並行スレッド式の導出

結論から言えば，並行オブジェクト式

$API\ []\ CH_0[]\ CH_1[]\ CH_2[]\ SYN\ []\ CLK\ []\ DA$

は，等価変換によって，並行スレッド式

$API \| SYN$

に変換可能である．以下にその変換の概要を示す．

まず，$CH_0[]\ CH_1[]\ CH_2$ を計算する．

$CC_i = (p_i + s_i + f_i + v_i)$ とおくと，定理 4.2 より，

$CH_i \equiv (CC_i + calc)* \overset{*}{\leftrightarrow} CC_i^*[]\ calc^*$ となるから，

$CH_0[]\ CH_1[]\ CH_2 \equiv$

$CC_0^*[]\ calc^*[]\ CC_1^*[]\ calc^*[]\ CC_2^*[]\ calc^*$

[] の交換法則を使って，

$\overset{*}{\leftrightarrow}\ CC_0^*[]\ CC_1^*[]\ CC_2^*[]\ (calc^*[]\ calc^*[]\ calc^*)$

calc は同期して一つにまとまるので，

$\overset{*}{\leftrightarrow}\ CC_0^*[]\ CC_1^*[]\ CC_2^*[]\ calc^*$

次に，$API\ []\ CC_i^*$ を計算する．CC_i^* に定理 4.2 を繰り返し適用し，$CC_i \overset{*}{\leftrightarrow} p_i^*[]\ s_i^*[]\ f_i^*[]\ v_i^*$ を得る．また，$(API \overset{*}{\leftrightarrow} (play.p_i + X))$ なる X が存在することと，定理 4.3 より，

$API\ []\ p_i^* \overset{*}{\leftrightarrow} (play.p_i + X)^*[]\ p_i^* \overset{*}{\leftrightarrow} (play.p_i + X)^*$

s_i, f_i, v_i についても同様に計算できるので，結局，

$API\ []\ CC_i^* \overset{*}{\leftrightarrow} API\ []\ p_i^*[]\ s_i^*[]\ f_i^*[]\ v_i^* \overset{*}{\leftrightarrow} API$

したがって，

$API\ []\ CC_0^*[]\ CC_1^*[]\ CC_2^* \overset{*}{\leftrightarrow} API$

以上の結果を用いて，

$API\ []\ CH_0[]\ CH_1[]\ CH_2[]\ SYN\ []\ CLK\ []\ DA$

$\overset{*}{\leftrightarrow}\ API\ []\ calc^*[]\ SYN\ []\ CLK\ []\ DA$

$\overset{*}{\leftrightarrow}\ API\ []\ calc^*[]\ (clk.calc.write)^*$

$\overset{*}{\leftrightarrow}\ API\ []\ (clk.calc.write)^*$

$\overset{*}{\leftrightarrow}\ API \| SYN$

4.4.3 スレッド抽出

ここまでの結果より，スレッド API と SYN を並行動作させる事で，システム全体の振る舞いが表現できる事が分かった．しかし，これらのスレッドと実装コードの間にはまだ隔たりがある．そこで，これらのスレッドをさらに分解し，実装コードと直接対応付け可能なスレッドを抽出する必要がある．

並行スレッド式 API‖SYN をよく観察すると，API,SYN はともに一定のアクション列を周期的に繰り返すが，個々の周期の発火条件は，常に外部からのイベントの到着である事が分かる．

例えば，スレッド API の中で，外部からのイベントを表現するアクションは play, stop, freq, vol である．これはドライバ外部から API のエントリが呼び出された事に対応する．また，スレッド SYN においては，clk が外部からのイベントである．このアクションはドライバ外部からクロック割り込み通知が到着した事を意味する．

設計モデルに出現しているスレッドの周期的な振る舞いは，ドライバが，周期的に到着する外部イベントに応答する事を意味している．ドライバソフトウェア自体は，外部から到着するイベントに応答する非周期的なスレッドの集合で構成されていれば十分である．

以上の考察から，我々は，外部からのイベントに対応するアクション play, stop, freq, vol, clk を先頭に持つ，5 本のアクション列を抽出した．これらのアクション列を実装すべきスレッドとする．表 4 に抽出したスレッドを示す．

PLAY	$play.(p_0 + p_1 + p_2)$
STOP	$stop.(s_0 + s_1 + s_2)$
FREQ	$freq.(f_0 + f_1 + f_2)$
VOL	$vol.(v_0 + v_1 + v_2)$
CLK	clk.calc.write

表 4 スレッドの抽出結果

このうち，スレッド PLAY, STOP, FREQ, VOL はスレッド API において和演算で結合されていた．このため互いに並行動作する事はない．それ以外のスレッドの組みは互いに並行動作する可能性がある．

4.5 実装工程

4.5.1 基本方針

前節で得られた 5 本のスレッドを，一つの関数に直接対応させて実装した．関数名はスレッドの名前をそのまま用いている．

PLAY,STOP,FREQ,VOL については API エントリと直接対応するので，その引数は API の仕様に準じている．アクション play,stop,freq,vol は，外部からの API エントリ呼び出しによって発生する．

スレッド CLK はクロックに同期して呼び出されるスレッドである．このため，割り込みに同期して，関数 CLK が呼ばれるようにする必要がある．これは，ドライバの初期化時に，OS の割り込み通知機能に対して，関数 CLK を登録する事によって実現する．

それぞれのスレッドの中身，すなわち関数内部で実行される詳細な内容は，個々のアクションの意味を考えて適切に実装した．実装の主要部分を付録に示す．

4.5.2 相互排他スレッド

設計の結果から，PLAY,STOP,FREQ,VOL は互いに並行動作しない事が定められている．しかし，スレッドをそのまま関数に対応付けて実装すると，外部イベントの発生頻度によっては，関数の再入呼び出しが発生する．関数への再入を認める事は，すなわちスレッドが並行動作することを意味する．これは，設計モデルの振る舞いと異なる，誤った振る舞いである．

この状況を回避するために，PLAY,STOP,FREQ,VOL の各関数が排他的に動作するような仕組みを設ける．これは OS 持つセマフォ機能を使う事で実現出来る．例えば，関数 VOL を次のように実装した．

```
void VOL(int ch, int value){
  ENTER(sem);
  v[ch] = value;
  LEAVE(sem);
}
```

ここで，sem はセマフォとする．ENTER はセマフォを上げて排他領域へ侵入する事を，LEAVE はセマフォを下げて排他領域から脱出する事を，それぞれ意味するマクロである．排他領域に侵入できるスレッドは一つだけである．ENTER の際に，もし他のスレッドによって，既にセマフォが上げられた状態だった場合，スレッドはブロックされ，セマフォが下がるまで待機する．

PLAY,STOP,FREQ についても同様に，関数の出入り口でそれぞれ排他領域に対する ENTER,LEAVE 操作を実装する．このようにする事で，各スレッドの相互排他動作が実現される．

5. 考　　察

今回の実験のようにごく小さなモデルであっても，モデル変換は難しい問題である．当初，並行オブジェクトモデルから並行スレッドモデルへの変換は，定理を使用せず，純粋に公理の適用のみで行っていた．しかし，この方法では，公理の適用ステップが，手作業で計算可能な範囲を越えてしまい，並行スレッドモデルを得る事ができなかった．

そこで，定理を使って変換ステップを大幅に減らすことで，変換結果を得る事ができた．しかし，今回使用した定理は，この実験で使用したモデルに特化した定理である．異なるモデルの変換に対して，この定理を再利用できるとは限らない．また今回は運良く適切な定理を発見する事ができたが，一般には，このような定理が容易に見つかるとは限らない．

6. ま と め

本稿では，並行正規表現を用いたモデル変換の手法を，PCM デバイスドライバ開発に適用する実験を紹介した．今回の実験により，以下の結果が得られた．

- 分析モデルから設計モデルへの変換が可能である事が確認できた．
- 設計モデルを，外部からのイベントで発火する非周期的なスレッドの組に分解する事で，スレッドと実装を対応付ける事ができた．

今回の適用実験では非常にうまく実装できる形の設計モデルが得られたが，これは対象としたシステムの規模が小さく，システムの複雑度が低かったことに起因する部分が大きい．より規模が大きく，複雑なシステムに対する実験が今後の課題である．ただし，規模が大きいシステムの場合，分析モデルから設計モデルへの変換作業が爆発的に増大する．このため，変換作業を支援する計算機環境の構築を行う予定である．

参 考 文 献

1) 岡崎光隆，青木利晃，片山卓也: 並行動作するオブジェクトからの処理列の抽出法，ソフトウェア科学会 FOSE2001 ソフトウェア工学の基礎 VIII, pp147-150, 2001.

2) Vijay K. Garg, M.T. Ragunath: *Concurrent regular expressions and their relationship to Petri nets*, Theoretical Computer Science 96, pp.285-304, 1992.

3) Toshiaki Aoki and Takuya Katayama: SES Model for Object-Oriented Time Critical System Development, ACIDCA 2000, pp.19-24, 2000.

4) Arto Salomma : *Two Complete Axiom Systems for the Algebra of Regular Events* , Journal of the Association for Computing Machinery, Vol.13, No. 1, pp158-169, 1966.

5) J.Rumbaugh, M.Blaha, W.Premerlani, F.Eddy, W.Lorenson: *Object-Oriented Modeling and Design*, Prentice Hall, 1991.
6) Maher Awad, Juha Kuusela and Jurgen Ziegler : *Object-Oriented Technology for Real-Time Systems* , Prentice Hall, 1996.
7) Robin Milner : *Communication and Concurrency*, Prentice Hall, 1989

付録:実装例 (主要部分を抜粋)

```
#define INPUT_CLOCK 50000 /* クロック周波数 */
#define CH_MAX 3
static int f[CH_MAX], v[CH_MAX];
static short *start_adr[CH_MAX];
static short *current_adr[CH_MAX];
static short *end_adr[CH_MAX];
static double counter[CH_MAX];
static double counter_step[CH_MAX];
static int playflag[CH_MAX];
static SEMAFO sem=0;

/* play.(p_0+p_1+p_2) */
void PLAY(int ch, short *start, short *end){
  ENTER(sem);
  playflag[ch] = 1;
  start_adr[ch] = start;
  end_adr[ch] = end;
  LEAVE(sem);
}

/* stop.(s_0+s_1+s_2) */
void STOP(int ch){
  ENTER(sem);
  playflag[ch] = 0;
  LEAVE(sem);
}

/* freq.(f_0+f_1+f_2) */
void FREQ(int ch, int value){
  ENTER(sem);
  f[ch] = value;
  counter[ch] = 0.0;
  counter_step[ch] = f[ch]/INPUT_CLOCK;
  LEAVE(sem);
}

/* vol.(v_0+v_1+v_2) */
void VOL(int ch, int value){
  ENTER(sem);
  v[ch] = value;
  LEAVE(sem);
}

/* clock.calc.write */
void CLK(){
  short mix = 0;
  int i;

  for(i=0;i<3;i++)
  {
    if(playflag[i])
    {
      /* 波形ポインタを移動 */
      counter[i] += counter_step[i];
      if(counter[i]>=1.0)
      {
        counter[i]-=1.0;
        current_adr[i]++;
      }
      /* 終了チェック */
      if(current_adr[i]==end_adr[i])
        playflag[i] = 0;

      mix += (*(current_adr[i]) * v[i]) >> 4;
    }
  }
  DA_WRITE(mix); /* I/O Access */
}
```