修 士 論 文

並行オブジェクトのスレッドへの変換法の研究

指導教官　片山卓也 教授

審査委員主査　片山卓也 教授

審査委員　二木厚吉 教授

審査委員　渡辺卓雄 助教授

北陸先端科学技術大学院大学
情報科学科 情報システム学専攻

910020　岡崎光隆

平成 13 年 2 月 15 日

目 次

第 1 章

はじめに

1.1 研究の背景

各種の機器に組み込まれ，その制御を行う計算システムを組み込みシステムと呼ぶ．近年の半導体技術の進歩に伴い，組み込みシステムの大規模化，複雑化は急激に進んできた．しかし，これまで組み込みシステム開発に用いられてきた手法は，比較的小規模なシステムに対する開発法であり，現在の手法で将来も開発を続ける事の限界が指摘されている．この問題に対して，オブジェクト指向のアプローチを組み込みシステムの開発に適用する研究が有力視されている．

オブジェクト指向の開発法は，大規模なシステムのモデリングやソフトウェアの再利用に関して優れた実績を持ち，広く普及している．開発法全体は大きく分析，設計，実装の 3 工程に分けられる．分析工程では，システムの論理的構造をオブジェクト中心の観点でまとめ，分析モデルを作成する．設計工程では分析モデルを具体的に実現する方法を決定し，設計モデルを作成する．実装工程は設計モデルを元に実装が行われる．

組み込みシステムには他のシステムに比べ，厳しい時間や資源の制約が存在するという特徴がある．組み込みシステムの開発法で特に重要となるのは，時間制約や，資源競合の問題の解決である．オブジェクト指向の開発法では，これらの問題は設計工程で扱われる．分析工程ではシステムは各オブジェクトに論理的に並行な動作が仮定された上でモデル化されるが，一般に組み込みシステムが実現される際には，同期した事象は一つのスレッドにまとめられ，非同期な事象のみが並行に動作するという実装が行われる．時間の経過や資源の消費はスレッドの実行に沿って発生する概念であり，システムの実現方法の影響を強く受ける問題である．したがって，これらの問題はシステムの実現方法が考慮される設計工程で扱われる．

しかし，従来のオブジェクト指向の開発法は，組み込みシステムに特化した開発法ではない．このため，設計工程で時間や資源の制約を扱う方法は不十分であり，現実的な組み込みシステムに対応できないという問題がある．

これに対し，近年，従来の開発法を拡張し，組み込みシステムに特化させた開発法が提案されてきた．[1, 2]．これらの開発法は，設計工程で組み込みシステムに適したスレッド中心のモデルを作成する事に特色がある．

SES アプローチ

本研究では組み込みシステムの開発法である SES アプローチ [2] に焦点を当てる．SES アプローチは，SES モデルと呼ばれるスレッド中心の設計モデルを構成する．SES モデルは，SES(Synchronus Execution Sequence) と呼ばれるスレッドを単位とする．SES は，その実行に際して，ハードウェア処理による待ち時間や，資源の競合によるブロックといった処理を含まない．このため，各 SES における処理時間の見積もりは容易であり，実時間性を考慮しやすいという特徴がある．

図 1.1 は SES とオブジェクトの関係である．o_i はオブジェクト識別子，p_i はオブジェクトで行われる処理である．SES に含まれる各処理はすべて短時間のうちに処理される．

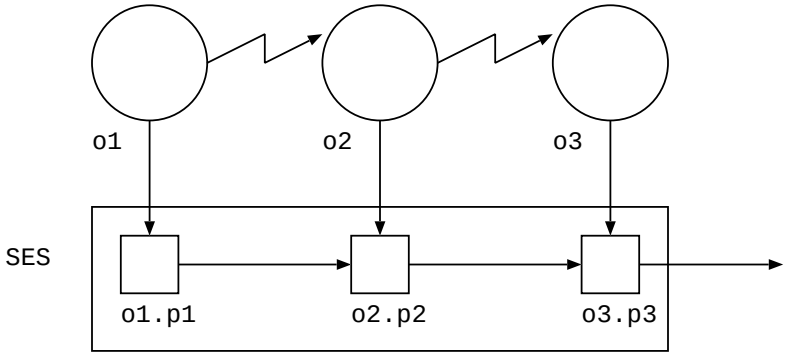

図 1.1: SES とオブジェクトの関係

SES モデルは，図 1.2 のように SES と状態遷移図を組み合わせたものである．このモデルでは，状態遷移図で SES の実行順序を制御し，システム全体の振る舞いを定義する．状態遷移図は，システムの全域的な状態と，それらの間の遷移関係である．状態遷移図の各状態には，その状態で実行すべき SES の集合が関連づけられている．

SES モデルの状態遷移はイベントの観測によって発生する．図 1.2 のモデルでは，例えば

初期状態 $s0$ では SES1, 2, 3 の 3 本の SES が並行に実行されている．この状態で，イベント $e2$ が観測されると，システムの状態は $s1$ に移り，SES1, 2, 3 の実行は停止され，$SES4, 5$ が並行に実行される．

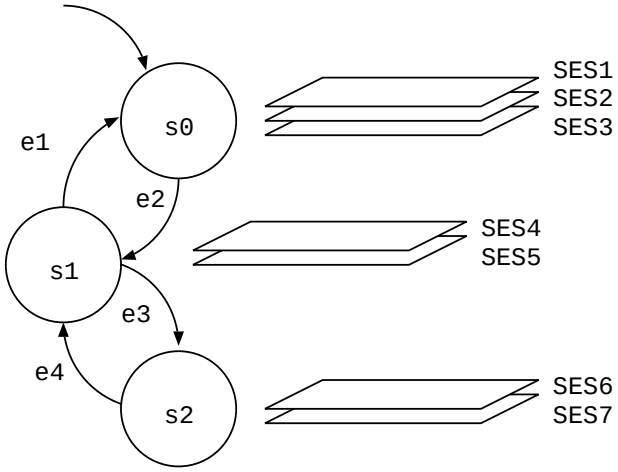

図 1.2: SES モデル

1.2 研究の目的

現在の SES アプローチには，分析モデルから，SES モデルを構成する手法が形式化されてない．このため，分析工程から設計工程に移行する際の効率と堅実性に欠けるという問題がある．そこで本研究では，分析モデルの振る舞いから具体的なスレッドを獲得する方法を確立し，SES モデルを構成する手法を形式化する事を目的とする．

1.3 本論文の構成

本論文の構成は以下の通りである．第 2 章では，分析モデルの定義を行う．分析モデルの振る舞いは正規表現に並行性と同期に関する演算を拡張したスレッド式を用いて形式化される．第 3 章では形式化された分析モデルの振る舞いを元に，スレッドを獲得する手法を提案する．これは分析モデルの振る舞いを示すシステム式から，スレッドの振る舞いを示すスレッド式へ等価変換を行う方法である．第 4 章では，分析モデルの記述とスレッドへの変換の例を示す．第 5 章では本論文で提案した変換の手法について考察し，第 6 章で本論文を総括する．

第 2 章

分析モデルの定義

2.1 概要

分析モデルとは，システムの論理的な振る舞いを記述したモデルである．本研究では，分析モデル全体を並行に動作するオブジェクトの集合としてモデル化する．各オブジェクトの振る舞いは有限状態機械を用いて特徴付けられ，この状態機械が生成する言語が，オブジェクトの可能な振る舞い全体の集合を表現する．ただし，表記の見通しの良さから，本稿ではオブジェクトの振る舞いの形式的な表現として，正規表現を拡張した記法を用いる．正規表現はその生成する言語に関して状態機械と等価な表現力を持つ表現である．

分析モデル全体の振る舞いは，正規表現に並行演算子と同期複合演算子を拡張した並行正規表現を用いて定義する．本節における並行正規表現は，[3] で導入された並行正規表現に基づきその一部を拡張したものである．

オブジェクト同士の相互通信の方法には，同期通信を仮定する．同期通信とは，送信側，受信側のオブジェクトが共に通信の完了まで待機する通信方法である．同期通信の仮定の下では，分析モデル全体の振る舞いは正規集合と対応付けられる．並行演算子は複数の処理が並行に動作した場合の振る舞いを，同期複合演算子は複数のオブジェクトが同期しながら並行に動作した場合の振る舞いを，それぞれ正規集合と対応付けて表現する演算子である．

2.2 オブジェクト

2.2.1 オブジェクトの振る舞い

図 2.1 にオブジェクトの例を示す．角丸の四角がオブジェクトを，その内部の状態機械がオブジェクトの振る舞いを表現している．状態機械の初期状態は，始点に状態を持たない矢印が指し示す状態である．二重丸の状態は状態機械の最終状態を意味する．

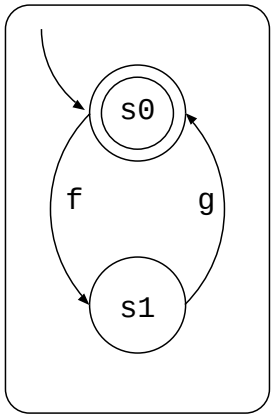

図 2.1: オブジェクトの例

状態機械の遷移は，オブジェクトで何らかの不可分な処理が行われる事の抽象である．図 2.1 のオブジェクトの状態機械は，初期状態 $s0$ から処理 f を行った後に状態 $s1$ に移り，さらに処理 g を行うと，再び状態 $s0$ に移る，といった振る舞いを表現している．この状態機械が生成する言語は，オブジェクトが行う処理を順に並べた列の集合であり，オブジェクトの可能な振る舞いがすべて表現された集合である．

2.2.2 オブジェクトの定義

本節では，オブジェクトの振る舞いを有限集合 Σ 上の正規表現として定義する．Σ 上の正規表現とは，$\{\epsilon, \perp\} \cup \Sigma$ に対して，和，連結，閉包の 3 種の演算を有限回行う事によって得られる表現である．また，正規表現によって生成される言語を正規集合または正規言語という．なお，$\perp$ は語を一つも含まない言語 (つまり $\{\perp\} = \phi$)，ϵ は空列を表わすものとする．

有限集合 Σ 上の正規表現を以下のように定義する．

定義 2.2.1 (正規表現のシンタックス)

1. $c \in \Sigma \cup \{\bot, \epsilon\}$ は Σ 上の正規表現である．
2. P と Q が Σ 上の正規表現であれば，和 $P + Q$ と連結 $P.Q$ も Σ 上の正規表現である．
3. P が Σ 上の正規表現であれば，閉包 P^* も Σ 上の正規表現である．

正規表現 P の意味は P の生成する言語である．P の生成する言語は $L(P)$ で表す.

定義 2.2.2 (正規表現の意味)

P，Q をそれぞれ Σ_P, Σ_Q 上の正規表現とする．このとき，関数 L を以下のように再帰的に定義する．

1. $P \in \Sigma_P$ のとき，$L(P) = \{P\}$
2. $L(P.Q) = \{ab | a \in L(P), b \in L(Q)\}$ (連結)
3. $L(P + Q) = L(P) \cup L(Q)$ (和)
4. $L(P^*) = \bigcup_{i=0,1,...} L(P^i)$ (閉包)

ただし，L^i は P の i 回の連結を意味する．
オブジェクトの振る舞いを表す正規表現を特にオブジェクト式と呼ぶ.

定義 2.2.3 (オブジェクト式)

任意の Σ に対して，Σ 上の正規表現はオブジェクト式である．

例えば，オブジェクトの振る舞いが図 2.2 の状態機械で表現されているとする．この状態機械の生成する言語は $\{ab, afgb, afgfgb, ...\}$ であるから，これをオブジェクト式で表わすと，$a.(f.g)^*.b$ となる．

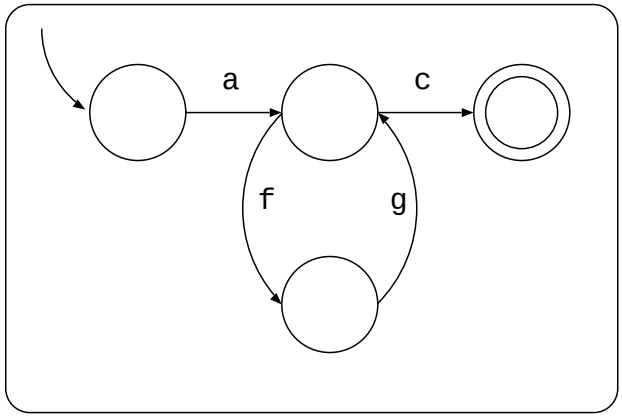

図 2.2: 状態機械の例

2.3 オブジェクトの並行動作

前節ではオブジェクトの振る舞いを正規表現によって記述し，正規集合と対応付けたが，複数の処理の並行動作による振る舞いもまた同様に正規集合と対応付けて考える事ができる．

2.3.1 並行性

本研究では、2 つの処理 a,b が並行に行われるという概念を，処理 a と b が a, b もしくは b, a のどちらの順序で発生してもよいことと定める．例えば，処理 a と処理 b が並行に行われた場合，その全体の振る舞いは集合 $\{ab, ba\}$ で表現される．また，処理の連結 $a.b$ と $c.d$ が並行に行われた場合も考えてみよう．$a.b$ と $c.d$ が示しているのは，a の後に b が発生する事と，c の後に d が発生する，という事実である．$a.b$ と $c.d$ が並行に動作した場合の振る舞いは，a, b, c, d がすべて並行に動作したと仮定した場合の振る舞いのうち，a の後に b が，c の後に d が出現するという制約が守られている振る舞いである．したがって，振る舞いの集合は $\{abcd, acbd, acdb, cabd, cadb, cdab\}$ となる．

2.3.2 並行演算子

並行に動作する複数の処理の振る舞いは，それらの処理がインターリーブされた列の集合である．この集合は並行演算子 $||$ によって計算される．

オブジェクト式に並行演算子を拡張した式をスレッド式と呼び，以下のように定義する．

定義 2.3.1 (スレッド式)

1. オブジェクト式はスレッド式である．
2. A, B がスレッド式ならば $A.B$ はスレッド式である．
3. A, B がスレッド式ならば $A + B$ はスレッド式である．
4. A がスレッド式ならば A^* はスレッド式である．
5. A, B がスレッド式かつ $\Sigma_A \cap \Sigma_B = \phi$ ならば $A||B$ はスレッド式である．

スレッド式の例を挙げる．

$$a||b, \quad (a.b.c)||(d.e.f)$$

$\Sigma_A \cap \Sigma_B = \phi$ という制約に注意．たとえば以下の式はスレッド式では無い．

$$a||a, \quad (a.b.c)||d.e.(f + a)$$

スレッド式の生成する言語を計算する関数 L_t の定義は以下の通り.

定義 2.3.2 (スレッド式の意味)
Σ を任意の記号の集合とする．このときスレッド式の意味関数 L_t を以下のように再帰的に定義する.

1. オブジェクト式 A に対して，$L_t(A) = L(A)$
2. スレッド式 A, B に対して $L_t(A.B) = \{ab | a \in L_t(A), b \in L_t(B)\}$
3. スレッド式 A, B に対して $L_t(A + B) = L_t(A) \cup L_t(B)$
4. スレッド式 A に対して $L_t(A^*) = \bigcup_{i=0,1,\ldots} L_t(A^i)$
5. $a \in \Sigma$ に対して，$L_t(a||\epsilon) = \{a\}$
6. $a, b \in \Sigma,\ P, Q \in \Sigma^*$ に対して，$L_t(a.P||b.Q) = L_t(a.(P||b.Q)) \cup L_t(b.(a.P||Q))$
7. スレッド式 P, Q に対して，$L_t(P||Q) = \{w | X \in L_t(P), Y \in L_t(Q), w \in L_t(X||Y)\}$

例えば，$L_t(ab||ac) = \{abac, aabc, aacb, acab\}$ である.

並行演算子には以下の性質が成り立つ.

1. $L_t(A||B) = L_t(B||A)$
2. $L_t(A||(B||C)) = L_t((A||B)||C)$
3. $L_t(A||\{\perp\}) = \{\perp\}$
4. $L_t((A + B)||C) = L_t((A||C) + (B||C))$

性質 2 より，$(A||B)||C, A||(B||C)$ の括弧を省略しても，式の意味に曖昧さが生じないので，これらを単純に $A||B||C$ と書く事がある.
なお，任意の正規表現 A, B に対して $L_t(A||B)$ は正規集合である [3].

2.4 オブジェクト間の通信

2.4.1 通信記号

各オブジェクトはそれぞれ並行に動作するが，相互に通信を行いつつ，他のオブジェクトとの協調動作を行う．本研究では，通信記号によって，オブジェクト間の相互通信を抽象する．複数のオブジェクトの振る舞いの表現に，共通して出現する記号が存在した場合，その記号を通信記号と呼び，オブジェクト間の相互通信の表現であるとする.

例えば，ある 2 つのオブジェクトの振る舞いが，それぞれオブジェクト式 abc,bcd で表現されているとすれば，通信記号は $\{a,b,c\} \cap \{b,c,d\} = \{b,c\}$ である．

通信記号は，送信側のオブジェクトと受信側のオブジェクトを区別しない．単にオブジェクトがある通信に関与した，という事実を表現した抽象度の高い表現である．

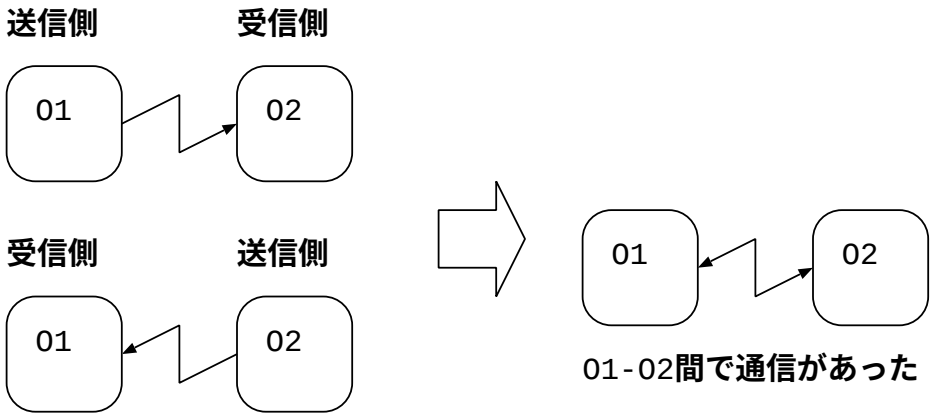

図 2.3: 通信記号の扱い

例えば図 2.3 に示したように，O_1 から O_2 にイベントが送信された，あるいは O_2 から O_1 にイベントが送信されたというような状況があったとする．通信記号を用いた表現では，この 2 者は区別されず，どちらも単に O_1 と O_2 の間で通信が発生した，という表現になる．

2.4.2 同期通信

オブジェクトが並行動作した場合の全体の振る舞いは，相互通信の方法に依存する．本研究では，この通信方法に同期通信を仮定する．同期通信とは以下の制約を満たすオブジェクト間の通信方法である．

1. 通信は決して無視されない
2. 通信完了までオブジェクトは次の処理に移らない

制約 (1) は，あるオブジェクトから通信要求が発生した時に，その要求は必ず他のオブジェクトに受信されるという事である．(2) は，双方の通信の完了時刻が一致するという事である．これはオブジェクトの振る舞いに次の制約を与える事に相当する．

- オブジェクトの振る舞いを表す複数の状態機械において，共通のラベルを持つ遷移があった場合，それらの遷移は複数の状態機械で必ず同時に発生する．

この制約が満たされることは，オブジェクトの振る舞い集合について次の関係が成り立つ事である．

定義 2.4.1 (同期通信)
B_1, B_2 をそれぞれオブジェクト O_1, O_2 の振る舞いの集合，S を O_1, O_2 間の通信記号の集合とする．このとき，オブジェクト O_1, O_2 間の通信方法が同期通信ならば，かつそのときにかぎり $B_1/S = B_2/S$.

ただし，w/Σ は w 中の記号を Σ に含まれる記号のみに制限する事を意味する．
例えば $xyzx/\{x, y\} = xyx, xyzx/\{y, z\} = yz$.

通信方法が同期通信であるとき，通信記号を特に同期記号と呼ぶ．また，2 つのオブジェクト間で同期通信が行われたとき，2 つのオブジェクトは同期した，という．

2.4.3 同期通信の例

複数のオブジェクトが相互通信して動作した場合の振る舞いの例を示す．図 2.4 は，電源をオンにすると電源オフまでの期間，音を鳴らしながらライトを点滅させる痴漢撃退装置のシステムのモデルである．このシステムは電源をオンにすると，電源オフまで音を鳴らすオブジェクト on.beep*.off と電源をオンにすると電源オフまでライトを点滅させるオブジェクト on.flash*.off で構成される．この 2 つのオブジェクトは並行に動作するが，電源のオンとオフが同期する．このため，電源のオンとオフは両オブジェクトに共通する記号 on と off で表現されている．これらの遷移は 2 つのオブジェクトで必ず同時に発生しなくてはならない．この制約の元で，全体の振る舞いは，

- on が発生する
- beep と flash はそれぞれ並行に，任意の回数 (0 回以上) 発生する．
- off が発生する．

となる．この振る舞い全体は集合
{on.off,on.beep.off,on.flash.off,on.beep.flash.off,on.flash.beep.off,...}

で表現できる．この集合を正規表現と並行演算子を用いて表現するとすれば，

$\mathrm{on}.(\mathrm{beep}^* \| \mathrm{flash}^*).\mathrm{off}$

となる．

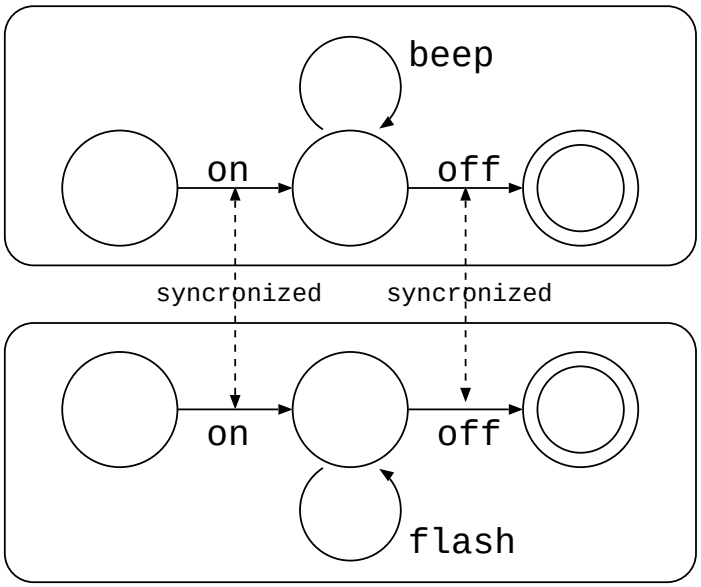

図 2.4: 痴漢撃退装置

なお，同期できない事があるオブジェクトの存在に注意が必要である．図 2.5 に，$O_1 : ac, O_2 : ab, O_3 : ba$ という 3 つのオブジェクトを示した．

ここで，O_1 と O_2 はつねに同期して動作可能である．しかし，O_2 と O_3 は同期できない．遷移ラベル a, b は O_2 と O_3 の両方に存在するラベルであるから，O_2 と O_3 で同時に遷移する必要がある．しかし，初期状態から可能な遷移は O_2 では a,O_3 では b のみである．したがって，a と b はどちらも遷移不可能であり，すなわち同期不可能である．

同期通信モデルは，同期可能である事を前提とするモデルである．このように同期に失敗が生じるモデルはモデル自体が誤っているものとして扱う．誤ったモデルの振る舞い全体の集合は空集合と対応付けられる．

2.4.4 同期複合演算子

複数のオブジェクトが同期しながら並行に動作した場合の振る舞いもまた振る舞いの集合に対応づけることが可能である．この対応付けのために，同期複合演算子 $[S]$ を導入する．

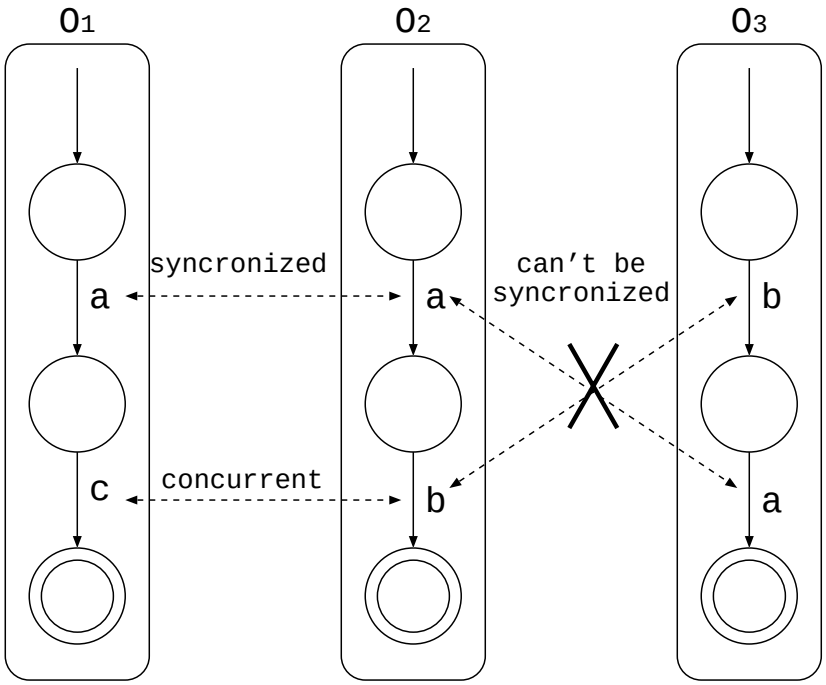

図 2.5: 同期の可／不可

スレッド式に同期複合演算子を拡張した式をシステム式または並行正規表現と呼ぶ．システム式は以下のように再帰的に定義される．

定義 2.4.2 (システム式)

- スレッド式はシステム式である．
- A, B がシステム式ならば，$A.B$ はシステム式である．
- A, B がシステム式ならば，$A + B$ はシステム式である．
- A がシステム式ならば，A^* はシステム式である．
- A と B をシステム式，S を任意の記号の集合とすると，$A[S]B$ はシステム式である．

Σ_A 上のスレッド式 A，Σ_B 上のスレッド式 B でそれぞれ振る舞いが定義されている 2 つのオブジェクトを考える．ここで，2 つのオブジェクトの間では，同一のラベルを持つ遷移は必ず同期して動作すると仮定する．この仮定の下で，これらのオブジェクトが協調動作した際に観測される振る舞いの全体をオブジェクト A, B 同期複合と呼び，$A[S]B$ と表記する．S は同期記号として扱うべき記号の集合である．

システム式に対応する振る舞い集合を計算するための関数 L_s は以下のように再帰的に定義される．

定義 2.4.3 (システム式の意味)

1. A がスレッド式のとき，$L_s(A) = L_t(A)$
2. システム式 A, B に対して $L_s(A.B) = \{ab | a \in L_s(A), b \in L_s(B)\}$
3. システム式 A, B に対して $L_s(A + B) = L_s(A) \cup L_s(B)$
4. システム式 A に対して $L_s(A^*) = \bigcup_{i=0,1,...} L_s(A^i)$
5. $S \cap (\Sigma_A \cup \Sigma_B) \subseteq \Sigma_A \cap \Sigma_B$ のとき，
   $L_s(A[S]B) = \{w | w/\Sigma_A \in L_s(A), w/\Sigma_B \in L_s(B), w/(\Sigma_A \cup \Sigma_B) = w\}$
6. それ以外の場合 $L_s(A[S]B) = \{\bot\}$

ただし，w/Σ は w 中の記号を Σ に含まれる記号のみに制限する事を意味する．

なお，同期複合演算子には以下の略記を認める．

$$A[\ ]B$$

と表記した場合は，

$$A[\phi]B$$

の略記である．

また，厳密には S={a,b,c..} の時，

$$A[S]B = A[\{a, b, c..\}]B$$

と書く必要があるが，特に {} を省略しても誤解の無い場合は，これを

$$A[a, b, c]B$$

と表記する．

演算子 $[S]$ には以下の性質が成立する．

1. $L_s(A[S]B) = L_s(A[S \cap \overline{(\Sigma_A \cap \Sigma_B)}]B)$
2. $L_s(A[S]A) = L_s(A)$
3. $L_s(A[S]B) = L_s(B[S]A)$
4. $L_s((A[S_1]B)[S_2]C) = L_s(A[S_1](B[S_2]C))$
5. $L_s(A[S]A^*) = L_s(A)$
6. $L_s((A + B)[S]C) = L_s((A[S \cup (\Sigma_B \cap \Sigma_C)]C) + (B[S \cup (\Sigma_A \cap \Sigma_C)]C))$

性質 4 より，$(A[S_1]B)[S_2]C, A[S_1](B[S_2]C)$ の括弧を省略しても式の意味に曖昧さが生じないので，これらを単純に $A[S_1]B[S_2]C$ と書く事がある．

2.4.5 演算子の優先度

本節で定義された演算子は，閉包，連接，和，並行，同期複合の順で結合力が高いものとする．本論文ではたとえば以下のように括弧を省略して表記する事がある．ただし，括弧を省略可能であっても，見にくいものに対しては適宜括弧を用いて記述する．

- $a.b^* \Leftrightarrow a.(b^*)$
- $a + b[\ ]c + d \Leftrightarrow (a + b)[\ ](c + d)$

2.4.6 その他の定義

同期に関する諸概念はシステム式の上では以下のように定義される．

定義 2.4.4 (同期記号)
A, B, L をシステム式，S を記号の集合とする．このとき，L の同期記号の集合 $C(L)$ を次のように定義する．

1. $L = A[S]B$ のとき，$C(L) = (\Sigma_A \cap \Sigma_B \cup C(A) \cup S \cup C(B)$
2. それ以外のとき，$C(L) = \phi$

例 1. $a.b[\ ]b.c$ の同期記号

$$\begin{aligned} C(a.b[\ ]b.c) &= (\{a, b\} \cap \{b, c\}) \cup C(a.b) \cup \phi \cup C(b.c) \\ &= \{b\} \cup \phi \cup \phi \cup \phi \\ &= \{b\} \end{aligned}$$

例 2. $a[b]c$ の同期記号

$$\begin{aligned} C(a[b]c) &= (\{a\} \cap \{b\}) \cup C(a) \cup \{b\} \cup C(b) \\ &= \phi \cup \phi \cup \{b\} \cup \phi \\ &= \{b\} \end{aligned}$$

例 3. $(a[b]c)[d]c$ の同期記号[1]

$$\begin{aligned} C((a[b]c)[d]c) &= (\{a,c\} \cap \{c\}) \cup C(a[b]c) \cup \{d\} \cup C(c) \\ &= \{c\} \cup \{b\} \cup \{d\} \cup \phi \\ &= \{b,d,c\} \end{aligned}$$

定義 2.4.5 (同期複合可能性)
オブジェクト式 A と B について，$A[\ ]B \neq \phi$ ならば，A と B は同期複合可能である，と呼ぶ．逆に，$A[\ ]B = \phi$ ならば A と B は同期複合不可能である．

2.5 分析モデルの振る舞いの定義

分析モデルの振る舞い全体はシステム式によって以下のように定義される．

定義 2.5.1 (分析モデルの振る舞い)
オブジェクト式 $O_1, ...O_n$ から成る分析モデル全体の振る舞いは，システム式 $O_1[\ ]O_2...[\ ]O_n$ で表現される．

[1] $\Sigma_{A[S]B} = \Sigma_A \cup \Sigma_B$ であることに注意．

第 3 章

スレッドへの変換

3.1 概要

前節では，並行正規表現によって，複数のオブジェクトが同期して動作した際の振る舞いをシステム式によって形式的に定義した．本節では，このシステム式からスレッドの情報を抽出する手法について論じる．

3.2 スレッドモデル

分析モデルを形式的に表現したように，抽出すべきスレッドの側も形式的に表現しておく必要がある．スレッド中心の観点でシステムの振る舞いを表現するモデルをスレッドモデルと呼び，本研究ではこれをスレッド式で形式的に表現する．スレッドモデルの例を図 3.1 に示した．

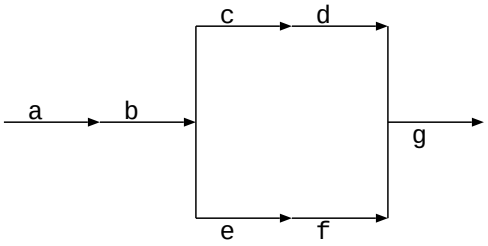

図 3.1: スレッドモデル

このスレッドモデルは，a,b が順に行われた後，c, d と e, f がそれぞれ順に，並行に行わ

れ，その後，gが行われる，といった振る舞いを表現している．これをスレッド式で記述すると，$(a.b.(c.d||e.f).g)$ となる．

3.2.1 スレッドモデルの特徴

スレッドモデルには次のような特徴がある，

- スレッドモデルの表現では，本質的に並行な事象のみが並行に動作するスレッドで表現され，並行な動作が要求されない事象は一本のスレッドにまとめられる．一般に並行に動作するスレッドが少なければ，実時間制約や資源競合などの見積もりは容易になるので，スレッドモデルのこの特性は有益である．なお，分析モデルと対応するスレッドモデルでは，同時に並行な動作が求められるスレッドの数は最大でもオブジェクトの個数以下である．
- あるシステムの振る舞いに対応するスレッドモデルは複数個存在する．したがって，あるスレッドモデルを元に作成したSESモデルが時間や資源の制約を満たさなかった場合でも，その他の適切なスレッドモデルを選択する余地がある．

3.2.2 スレッドモデルの定義

スレッドとは，連続的に実行が可能な一連の処理の流れである．ただし，スレッドは他のスレッドとの同期処理を含まない．

スレッドモデルは複数のスレッドを組み合わせて構成されるモデルである．スレッドモデルの振る舞いはスレッド式で表現される．スレッド式は正規表現に並行演算子を拡張した式であり，その定義は2.3.1で示した．

例えば次のような表現がスレッドモデルである．

$$a||b, \quad (a+b)^*||c.(d||e), \quad a.b.c||(d.e.f)^*||h.i.j$$

分析モデルでは，各オブジェクトがそれぞれ並行に動作する事を仮定していた．このため，一見，分析モデルと等価な振る舞いを実現する際には，オブジェクトの個数分並行に動作するスレッドが生じると思われるかもしれない．しかし，スレッドモデルのスレッドは，オブジェクト一対一に対応する概念ではなく，並行に動作するスレッドの数はオブジェクトの個数以下で済む場合がある．

たとえば，$a.x.b$ と $a.y.b$ と $a.b$ という 3 つのオブジェクトがあった場合，そのすべてが並行に動作する訳ではない．これら 3 つのオブジェクトの振る舞いは，システム式 $a.x.b[\ ]a.y.b[\ ]a.b$ で表現され，その可能な振る舞い全体の集合は $\{axyb, ayxb\}$ である．この集合はスレッド式 $a.(x\|y).b$ でも表現する事ができる．この式からは，並行に動作すべき部分は $x\|y$ である事が分かる．x, y がそれぞれ並行に動作するスレッドである．

3.3 スレッド導出公理

本研究では，分析モデルからスレッドを抽出するという行為を，分析モデルからスレッドモデルへの変換で表現する．変換の形式的手法としては，公理系によるシステム式からスレッド式への等価変換を提案する．ここでいう等価とは二つの式が同じ振る舞いの集合と対応する事を指す．

提案する公理系は，正規表現における等価変換，並行演算における等価変換，同期複合演算における等価変換の 3 種の公理から成る．

3.3.1 正規表現における等価変換の公理

A, B, C はそれぞれ正規集合とする．

公理 3.3.1 (同一性)
$A + A = A$

公理 3.3.2 (和の交換法則)
$A + B = B + A$

公理 3.3.3 (和の結合法則)
$(A + B) + C = A + (B + C)$

公理 3.3.4 (連結の分配法則)
$(A + B).C = A.C + B.C$

公理 3.3.5 (連結の結合法則)
$(A.B).C = A.(B.C)$

公理 3.3.6 (閉包の展開)
$A^* = \epsilon + A.A^*$

3.3.2 並行演算における等価変換の公理

A, B, C はそれぞれ正規集合とする.

公理 3.3.7 (並行演算子の零元)
$A||\{\perp\} = \{\perp\}$

この公理は，振る舞いが存在しないようなスレッドが存在するシステムは誤りである事を意味する.

公理 3.3.8 (並行演算子の単位元)
$A||\{\epsilon\} = A$

公理 3.3.9 (並行演算子の交換法則)
$A||B = B||A$

スレッド A と B が並行に動作する事は，B と A が並行に動作する事と等しい.

公理 3.3.10 (並行演算子の結合法則)
$A||(B||C) = (A||B)||C$

公理 3.3.11 (並行演算子の分配法則)
$(A + B)||C = (A||C) + (B||C)$

公理 3.3.12 (並行演算子の展開)
$a.A||b.B = a.(A||b.B) + b(a.A||B)$

3.3.3 同期複合演算における等価変換の公理

A, B, C はそれぞれ正規集合,S は記号の集合とする．また，Σ_R は正規集合 R に出現する記号の集合を示す.

公理 3.3.13 (同期失敗)
$S \cap (\Sigma_A \cup \Sigma_B) \not\subseteq \Sigma_A \cap \Sigma_B$のとき, $A[S]B = \perp$

ある同期記号が，A と B のどちらか一方にだけ存在する場合，A と B は同期不可能なので，振る舞いが空集合になるという事を示している．以下のような例が挙げられる.

例． $a.b[\{b\}]c = \perp,\ a.b[\{b, d\}]c.d = \perp$

公理 3.3.14 (同期演算の零元)
$A[S]\{\bot\} = \{\bot\}$

この公理は，振る舞いが存在しないようなオブジェクトとはいかなるオブジェクトも同期不可能である事を意味している．

公理 3.3.15 (同期演算の単位元)
$S \not\subseteq \Sigma_A$のとき, $A[S]\{\epsilon\} = A$

同期記号を含まないオブジェクトであれば，空列と同期可能であり，その振る舞い全体の集合はそのオブジェクトの振る舞いそのものである．

公理 3.3.16 (同期演算子の交換法則)
$A[S]B = B[S]A$

いかなる場合においても，オブジェクト A と B が同期した場合の振る舞いと，B と A が同期した場合の振る舞いは等しい．

公理 3.3.17 (同期演算子の結合法則)
$(A[S_1]B)[S_2]C = A[S_1](B[S_2]C)$

公理 3.3.18 (同期演算子の分配法則)
$(A + B)[S]C = (A[S \cup (\Sigma_B \cap \Sigma_C)]C) + (B[S \cup (\Sigma_A \cap \Sigma_C)]C)$

公理 3.3.19 (スレッド化)
$\Sigma_A \cap \Sigma_B = \phi$のとき, $A[S]B = A||B$

A と B がどちらも同期記号を含んでいないならば，A と B は並行に動作する二つのスレッドである．

公理 3.3.20 (デッドロック検出)
$X, Y \in (S \cup (\Sigma_A \cap \Sigma_B))^*$かつ $X \neq Y$ のとき, $X.A[S]Y.B = \{\bot\}$

オブジェクト A が同期記号の列 X の同期待ち，オブジェクト B が同期記号の列 $Y(\neq X)$ の同期待ちをしている場合，A と B は同期不可能である．したがって，振る舞い全体の集合は空集合となる．

例. $x.a.b.c[x,y]y.d.e.f = \{\bot\}$

公理 3.3.21 (非同期部分の抽出)

$X \in \Sigma^*$かつ $(\Sigma_A \cup \Sigma_C) \cap S = \phi$のとき, $(A.X.B)[S](C.X.D) = (A||C).(X.B)[S](X.D)$

2 つのオブジェクトが，将来ある同期記号 X で同期可能である事がわかっている場合，X 以前の非同期記号の列は，並行に動作するスレッドとみなす事ができる．

例. $a.b.x.c[\ ]d.e.x.f = (a.b||d.e).(x.c[\ ]x.f)$

公理 3.3.22 (同期部分の抽出)

$X \in \Sigma^*$のとき, $(X.A)[S](X.B) = X.(A[S \cup \Sigma_X]B)$

2 つのオブジェクトが X で同期可能であれば，X はスレッドとみなせる．

例. $x.y.a.b[\ ]x.y.c.d = x.y.(a.b[x,y]c.d)$

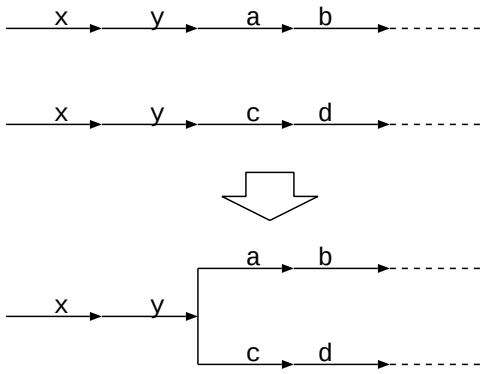

図 3.2: 同期部分の抽出

公理 3.3.23 (同期記号の最適化)

$A[S]B = A[S \cap \overline{(\Sigma_A \cap \Sigma_B)}]B$

同期記号 $\Sigma_A \cap \Sigma_B$ は同期記号の集合 S に含まれなくても同期記号とみなされる．

例. $x.y[x]x.z = x.y[\ ]x.z$

システム式の変換は，次に示す公理の適用規則と循環解決規則で行われる．

3.3.4 公理の適用規則

A, B をシステム式とする．公理により，$A = B$ であるとき，システム式 P の任意の位置の A の出現を B に，B の出現を A に置換してよい．

システム式 S に公理の適用規則を任意の回数適用した結果，式 T が得られる事を $S \Rightarrow T$ と表現する．

3.3.5 循環解決規則

閉包を含むシステム式は，公理の適用のみではスレッド式に変換する事ができない．公理に加えて以下の推論規則が必要である．

定義 3.3.1 (循環解決規則)
A, B をシステム式であると仮定し，

$$S \Rightarrow T \text{ かつ } T = A.S + B \text{ ならば } T \Rightarrow A^*.B$$

第 4 章

例題

4.1 分析モデルの例

本節では分析モデルの記述例として集合住宅の入り口などで利用されている，単純なオートロック式ドアの仕様を示す.

オートロック式ドアの仕様

オートロック式ドアシステムは，施錠可能なドアとカードリーダーから成るシステムである．オートロック式ドアは次のような動作をする.

1. ドアが施錠状態にあるとき，カードリーダにカードキーを挿入すると，ドアが開錠される.
2. ドアが開錠状態にあるとき，ドアを一度開いて閉じると，ドアは自動的に施錠される.
3. ドアが開場状態で，ドアが開かないまま一定時間が過ぎた場合，ドアは自動的に施錠される.

各オブジェクトの振る舞い

このシステムは，施錠可能なドア，カードリーダ，タイマーの 3 つのオブジェクトで実現される．

図 4.1 は施錠可能なドアオブジェクト (Door) の振る舞いを状態遷移機械で示したものである．

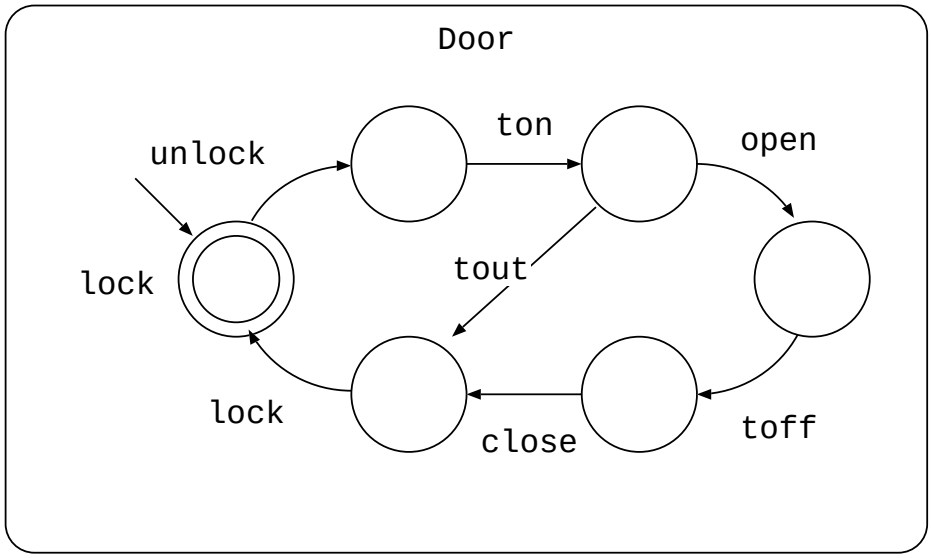

図 4.1: ドアオブジェクト

各遷移の意味は以下の通りである．

lock	ドアを施錠
unlock	ドアを開錠
ton	タイマーを開始
toff	タイマーを停止
tout	タイマーオブジェクトからのタイムアウト通知
open	ドアが開いた
close	ドアが閉じた

図 4.2 はタイマーオブジェクト (Timer) の振る舞いを示したものである．このオブジェクトは，タイマー開始要求が来て一定時間経過すると，タイムアウトを通達する．ただし，タイムアウトより前にタイマー中止の要求がくると，タイムアウトを発生させずに初期状態に戻る．

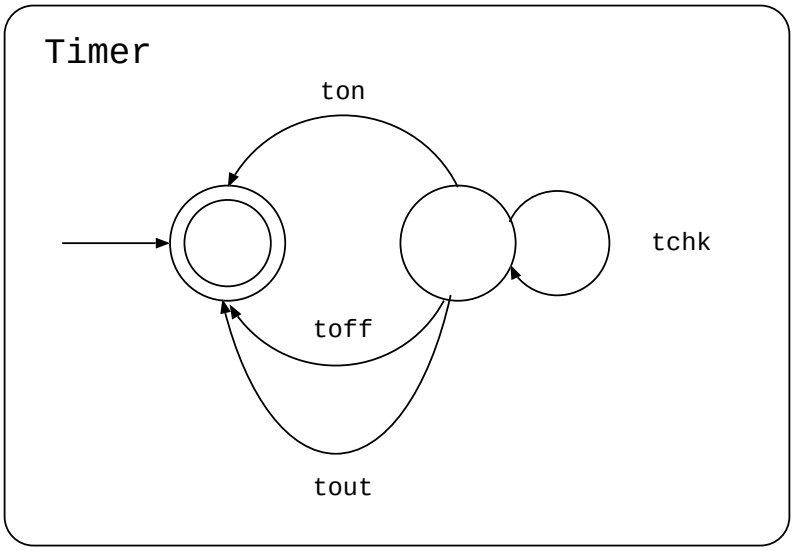

図 4.2: タイマーオブジェクト

各遷移の意味は以下の通り．

ton　タイマー開始
toff　タイマー中止
tchk　時間の計測
tout　タイムアウトを通達

図 4.3 はカードリーダオブジェクト (Card Reader) の振る舞いを示したものである．このオブジェクトは，カードが挿入されるとドアに開錠指示を出し，ドアが施錠されるまで待つ．という動作を繰り返すオブジェクトである．

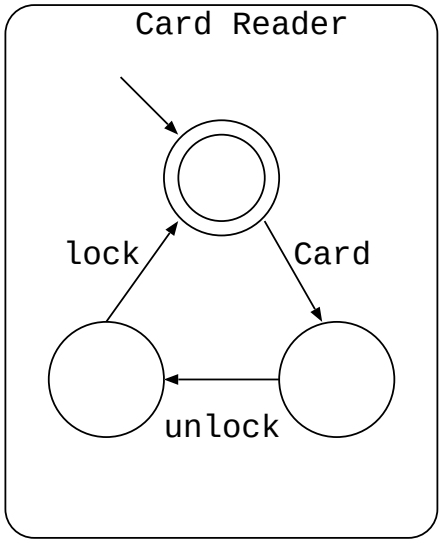

図 4.3: カードリーダオブジェクト

各遷移の意味は以下の通り．

card	カードが挿入された
lock	ドアが施錠された
unlock	ドアを開錠

分析モデル全体の振る舞いは以下のようになる．

1. Card Reader オブジェクトにカードが挿入されると遷移 cardin が発生する．
2. Card Reader はドアの開錠処理 unlock を発生させ，この遷移は Door オブジェクトの遷移 unlock と同期しする．結果，ドアが開錠される．
3. Door オブジェクトで遷移 ton が発生し，Timer オブジェクトの ton がこれに同期して，タイマーが作動する．
4. タイマー動作中は Timer オブジェクトは tchk を繰り返し，タイムアウトすると tout で遷移する．
5. ドアが空かないまま Timer オブジェクトがタイムアウトすると，tout によって，Door オブジェクトは閉錠操作に移行する．7 へ．
6. ドアが開かれると，Door オブジェクトで遷移 open が発生する．ドアが開けられると toff でタイマーは中止される．
7. ドアが閉じられると，Door オブジェクトで遷移 close が発生する．
8. Door オブジェクトが lock で遷移し，ドアの鍵が掛けられる．
9. Card Reader オブジェクトで lock で同期して遷移するので，再びカードを受け付けられる状態に戻る．

オートロック式ドアのシステム式

各オブジェクトをオブジェクト式で表現すると次の様になる．

Door	(unlock.ton.(tout + open.toff.close).lock)*
Card Reader	(card.unlock.lock)*
Timer	(ton.(tchk)*.(tout + toff))*

したがって，分析モデル全体は，システム式

(unlock.ton.(tout + open.toff.close).lock)*
[ ](card.unlock.lock)* [ ] (ton.(tchk)*.(tout + toff))*

で表現される．

4.2 スレッドの導出

ここでは，4.1 節で例示した，オートロック式ドアの分析モデルから，スレッド式を獲得する例を示す.

システム式,

(unlock.ton.(tout + open.toff.close).lock)*
[ ](card.unlock.lock)* [ ] (ton.(tchk)*.(tout + toff))*

から出発するが，変換過程で式が複雑化するので，状況により以下の記号 C,D,T を，以下の式の省略形として使用する.

- D=unlock.ton.(tout + open.toff.close).lock
- C=card.unlock.lock
- T=ton.tchk*.(tout + toff)

この省略形を使うと，システム式は

D*[ ]C*[ ]T*

で表される.

まず始めに (D*[ ]C*) の部分を変換する.

閉包の展開公理より,

$\mathrm{D}^*[\ ]\mathrm{C}^* \Rightarrow (\epsilon + \mathrm{D}.\mathrm{D}^*)[\ ]\mathrm{C}^*$

同期複合演算子の分配法則より

$\Rightarrow (\epsilon[\mathrm{unlock}, \mathrm{lock}]\mathrm{C}^*) + (\mathrm{D}.\mathrm{D}^*[\ ]\mathrm{C}^*)$

閉包の展開公理より,

$\Rightarrow (\epsilon[\mathrm{unlock}, \mathrm{lock}]\epsilon + \mathrm{C}.\mathrm{C}^*) + (\mathrm{D}.\mathrm{D}^*[\ ]\mathrm{C}^*)$

同期複合演算子の分配法則より

$\Rightarrow (\epsilon[\mathrm{unlock}, \mathrm{lock}]\epsilon) + (\epsilon[\mathrm{unlock}, \mathrm{lock}]C.C*) + (\mathrm{D}.\mathrm{D}^*[\ ]\mathrm{C}^*)$

$\epsilon[\mathrm{S}]\epsilon = \epsilon$ と同期失敗の公理より

$\Rightarrow \epsilon + \{\bot\} + (\mathrm{D}.\mathrm{D}^*[\ ]\mathrm{C}^*) \Rightarrow \epsilon + (\mathrm{D}.\mathrm{D}^*[\ ]\mathrm{C}^*)$

閉包の展開公理より,

$\Rightarrow \epsilon + (\mathrm{D}.\mathrm{D}^*[\ ](\epsilon + \mathrm{C}.\mathrm{C}^*))$

同期複合演算子の分配法則より,

$\Rightarrow \epsilon + (\mathrm{D}.\mathrm{D}^*[\ ]\mathrm{C}.\mathrm{C}^*) + (\mathrm{D}.\mathrm{D}^*[\mathrm{unlock}, \mathrm{lock}]\epsilon)$

$\epsilon[\mathrm{unlock}, \mathrm{lock}]\mathrm{C}.\mathrm{C}^*$の時と同様にして,

$\Rightarrow \epsilon + (\mathrm{D}.\mathrm{D}^*[\ ]\mathrm{C}.\mathrm{C}^*) + \epsilon + \{\bot\} \Rightarrow \epsilon + (\mathrm{D}.\mathrm{D}^*[\ ]\mathrm{C}.\mathrm{C}^*)$

D，C の省略を元に戻して,

$\Rightarrow \epsilon + (\mathrm{unlock.ton.}(\mathrm{tout} + \mathrm{open.toff.close}).\mathrm{lock.C}^*[\ ]\ \mathrm{card.unlock.lock.C}^*)$

非同期部分の抽出公理より

$\Rightarrow \epsilon + (\mathrm{card.}(\mathrm{unlock.ton.}(\mathrm{tout} + \mathrm{open.toff.close}).\mathrm{lock}).\mathrm{D}^*[\ ]\mathrm{unlock.lock.C}^*))$

同期部分の抽出公理より

$\Rightarrow \epsilon + (\mathrm{card.unlock.}(\mathrm{ton.}(\mathrm{tout} + \mathrm{open.toff.close}).\mathrm{lock}).\mathrm{D}^*[\mathrm{unlock}]\mathrm{lock.C}^*))$

同期演算子の分配法則より

$\Rightarrow \epsilon + (\mathrm{card.unlock.}((\mathrm{ton.tout} + \mathrm{ton.open.toff.close}).\mathrm{lock}).\mathrm{D}^*[\mathrm{unlock}]\mathrm{lock.C}^*))$

非同期部分の抽出公理より

$\Rightarrow \epsilon + (\mathrm{card.unlock.}(\mathrm{ton.tout} + \mathrm{ton.open.toff.close}).(\mathrm{lock.D}^*[\mathrm{unlock}].\mathrm{lock.C}^*))$

同期部分の抽出公理より

$\Rightarrow \epsilon$ + (card.unlock.(ton.tout + ton.open.toff.close).lock.(D*[unlock, lock]C*))

同期記号の最適化公理より

$\Rightarrow \epsilon$ + (card.unlock.(ton.tout + ton.open.toff.close).lock.(D*[ ] C*))

和の分配法則より

$\Rightarrow \epsilon$ + (card.unlock.ton.(tout + open.toff.close).lock.(D*[ ] C*))

したがって,

(D*[ ] C*) $\Rightarrow \epsilon$ + (card.unlock.ton.(tout+ open.toff.close).lock.(D*[ ] C*))

であるから，循環解決の推論規則を適用し,

(D*[ ] C*)$\Rightarrow$(card.unlock.ton.(tout+open.toff.close).lock)*

次に，(D*[ ] C*)[ ]T*を変換する.
B を card.unlock.ton.(tout+open.toff.close).lock の省略形とすると,

(D*[ ] C*)[ ]T* $\Rightarrow$B*[ ]T*

(D*[ ]C*) の時と同様にして,

B*[ ]T* $\Rightarrow \epsilon$ + (B.B*[ ]T.T*)

B,T の省略を元に戻せば,

(B.B*[ ]T.T*) =

(card.unlock.ton.(tout + open.toff.close).lock.B*) [ ](ton.tchk*.(tout+ toff).T*)

非同期部分の抽出公理より,

$\Rightarrow$card.unlock.(ton.(tout + open.toff.close).lock.B*) [ ](ton.tchk*.(tout+ toff).T*)

同期部分の抽出公理より,

$\Rightarrow$card.unlock.ton((tout + open.toff.close).lock.B*) [ton](tchk*.(tout+toff).T*)

和の分配法則より,

$\Rightarrow$card.unlock.ton.((tout.lock.B* + open.toff.close.lock.B*) [ton](tchk*.(tout+toff).T*))

同期複合の分配法則より

$\Rightarrow$card.unlock.ton.(

((tout.lock.B*)[ton, toff](tchk*.(tout+toff).T*)) +

((open.toff.close.lock.B*)[ton, toff](tchk*.(tout+toff).T*))

)

非同期部分の抽出公理より

$\Rightarrow$card.unlock.ton.(

(tchk*.((tout.lock.B*)[ton, toff]((tout+toff).T*))) +

((tchk*||open).((toff.close.lock.B*)[ton, toff]((tout+ toff).T*)))

)

和の分配法則より

$\Rightarrow$card.unlock.ton.(

(tchk*.((tout.lock.B*)[ton, toff](tout.T*+toff.T*))) +

((tchk*||open).((toff.close.lock.B*)[ton, toff]((tout.T*+toff.T*)))

)

同期演算子の分配法則より

$\Rightarrow$card.unlock.ton.(

(tchk*.(((tout.lock.B*)[ton, toff](tout.T*))+ ((tout.lock.B*)[ton, toff, tout](toff.T*)))+

((tchk*||open).((toff.close.lock.B*)[ton, toff]((tout.T*+toff.T*)))

)

同期失敗の公理より

$\Rightarrow$card.unlock.ton.(

(tchk*.(((tout.lock.B*)[ton, toff](tout.T*))+{$\perp$})) +

((tchk*||open).((toff.close.lock.B*)[ton](tout.T*+toff.T*)))

)

同期演算子の分配法則より

$\Rightarrow$card.unlock.ton.(
(tchk*.((tout.lock.B*)[ton, toff](tout.T*))) +
((tchk*‖open).(((toff.close.lock.B*)[ton, toff](tout.T*))+
((toff.close.lock.B*)[ton, toff, tout](toff.T*))))
)

同期失敗の公理より

$\Rightarrow$card.unlock.ton.(
(tchk*.((tout.lock.B*)[ton, toff](tout.T*))) +
((tchk*‖open).({$\perp$}+((toff.close.lock.B*)[ton, toff, tout](toff.T*))))
)

同期部分抽出の公理より

$\Rightarrow$card.unlock.ton.(
(tchk*.tout.((lock.B*)[ton, toff]T*)) +
((tchk*‖open).toff.((close.lock.B*)[ton, toff, tout]T*))
)

非同期部分抽出の公理より

$\Rightarrow$card.unlock.ton.(
(tchk*.tout.lock.(B*[ton, toff]T*)) +
((tchk*‖open).toff.close.lock.(B*[ton, toff, tout]T*))
)

同期記号の最適化公理より

$\Rightarrow$card.unlock.ton.(
(tchk*.tout.lock.(B*[ ]T*)) +
((tchk*‖open).toff.close.lock.(B*[ ]T*))
)

和の分配法則より

$\Rightarrow$card.unlock.ton.((tchk*.tout)+((tchk*‖open).toff.close)).lock.(B*[ ]T*)

したがって,

(B.B*[ ]T.T*) ⇒card.unlock.ton.((tchk*.tout)+((tchk*||open).toff.close)).lock.(B*[ ]T*)

ゆえに,

(B*[ ]T*) ⇒ϵ + card.unlock.ton.((tchk*.tout)+((tchk*||open).toff.close)).lock.(B*[ ]T*)

循環解決規則により,

(B*[ ]T*) ⇒(card.unlock.ton.((tchk*.tout)+((tchk*||open).toff.close)).lock)*

よって，分析モデルに対応するスレッド式は,

(card.unlock.ton.((tchk*.tout)+((tchk*||open).toff.close)).lock)*

第 5 章

考察

5.1 スレッドモデルとSESの対応

本節では，スレッドモデルからSESを構成する事について考える．1章で述べたようにSESは途中に同期待ちを含まないスレッドである．これに対し，スレッドモデルでは，スレッド同士の同期待ちは発生しないが，システム外部との同期待ちが発生する個所がある．

第4章で挙げた例題では，最終的にスレッドモデルとして，次のスレッド式が得られた．

$$(\mathrm{card.unlock.ton.((tchk^*.tout)+((tchk^*||open).toff.close)).lock})^*$$

ここで，記号 unlock,ton,tout,toff,lock は分析モデルでは同期記号であったが，スレッド式ではシステム内での同期の問題はすべて解決しているので，同期記号ではない．つまり同期待ちが発生しない事象である．したがって，これらの記号が並んだ列，例えばunlock.ton の連結などは一つのSESと見なすことが可能な記号列である．

しかし，記号 card,open,close は分析モデルでは非同期記号であったが，実際はシステム外部との通信を表現した記号である．したがって何らかの同期待ちが発生する可能性があるため，SESを構成する際には同期記号として扱う必要がある．これらの記号を区切りとして，その前後の系列は異なるSESに割り当てられなくてはならない．

今後，この考察を進め，スレッドモデルからのSESの抽出を行い，SESモデル全体の構成を行う手法を確立する．

5.2 変換法の完全性の問題

変換の非一意性

システム式に対応するスレッド式は複数存在する場合がある．

例えば，$a.x.b, c.y.d, x.y$ の 3 つのオブジェクトから成る分析モデルを考える．このモデル全体の振る舞いはシステム式 $a.x.b[\ ]c.y.d[\ ]x.y$ で表現できる．このシステム式からスレッド式を求めると以下のようになる．

$$\begin{aligned}
a.x.b[\ ]c.y.d[\ ]x.y &= (a.x.b[\ ]x.y)[\ ]c.y.d \\
&= a.(x.b[\ ]x.y)[\ ]c.y.d \\
&= a.x.(b[x]y)[\ ]c.y.d \\
&= a.x.(b\|y)[\ ]c.y.d \\
&= (a.x.b.y + a.x.y.b)[\ ]c.y.d \\
&= a.x.b.y[\ ]c.y.d + a.x.y.b[\ ]c.y.d \\
&= (a.x.b\|c).(y[\ ]y.d) + (a\|c)(y.b[\ ]y.d) \\
&= (a.x.b\|c).y.(\epsilon[y]d) + (a.x\|c).y.(b[y]d) \\
&= (a.x.b\|c).y.d + (a.x\|c).y.(b\|d)
\end{aligned}$$

しかし，この式に対しては公理の適用順によって別解が存在する．

$$\begin{aligned}
a.x.b[\ ]c.y.d[\ ]x.y &= a.x.b[\ ](c.y.d[\ ]x.y) \\
&= a.x.b[\ ](c\|x).(y.d[\ ]y) \\
&= a.x.b[\ ](c\|x).y.(d[y]\epsilon) \\
&= a.x.b[\ ](c\|x).y.d \\
&= a.x.b[\ ]c.x.y.d + x.c.y.d \\
&= a.x.b[\ ]c.x.y.d + a.x.b[\ ]x.c.y.d \\
&= (a\|c).(x.b[\ ]x.y.d) + a.(x.b[\ ]x.c.y.d) \\
&= (a\|c).x.(b[\ ]xy.d) + a.x.(b[x]c.y.d) \\
&= (a\|c).x.(b\|y.d) + a.x.(b\|c.y.d)
\end{aligned}$$

複数存在するスレッド式の中からどれを選択する事が適切かという評価基準は今後 SES モデルとスレッド式の対応付を行う過程で取り組まねばならない課題である．

また，期待されるスレッド式を導出できるような公理の適用戦略を考える必要がある．

完全性

変換法の適用によって，あるシステム式と等価なスレッド式を全て求める事が可能である時，変換法は完全であるという．現在，３章で提案した変換法が完全であるかどうかの問題は未解決である．

第 6 章

まとめ

6.1 まとめ

本論文では，分析モデル全体を並行に動作するオブジェクトで定義し，その振る舞いの形式化を行った．この形式化のために，正規表現に並行演算子と同期複合演算子の二つの演算子を拡張した．また，スレッドを中心としたスレッドモデルを提案し，分析モデルからスレッドモデルへの変換手法について述べた，この変換手法は，公理系を用いたシステム式からスレッド式への等価変換として形式化し，変換の実例を挙げた．

6.2 今後の課題

スレッド式と SES の対応を明らかにし，SES モデルを構成する手法を形式化する．このためには SES モデル側の形式化も必要である．また，変換法の完全性の問題について検討し，必要であれば変換法の拡張を行う．

謝辞

本論文を執筆するに当たり終始ご指導賜りました片山卓也教授，青木利晃助手に感謝申し上げます．また，本研究に対してご意見を頂いたり，質問や議論にも快く応じて下さいました片山研究室の皆さんに感謝いたします．

参考文献

[1] Maher Awad, Juha Kuusela and Jurgen Ziegler : *Object-Oriented Technology for Real-Time Systems* , Prentice Hall, 1996.

[2] 青木利晃, 内藤壮司, 片山卓也: オブジェクト指向組み込みシステム開発のための SES-Ba sed アプローチ , 日本ソフトウェア科学会学会誌 コンピュータソフトウェア Vol.16 No. 2, pp.67-71, 1999.

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

[4] J.Rumbaugh, M.Blaha, W.Premerlani, F.Eddy, W.Lorenson: *Object-Oriented Modeling and Design*, Prentice Hall, 1991.