Vol.2011-SE-173 No.1
2011/7/21

# LTSAを用いたシーケンス図の詳細化関係の検証

浅田和真[†1]　横川智教[†1]
宮崎仁[†2]　佐藤洋一郎[†1]

本論文では，UMLのシーケンス図で記述され，hMSCに基づいて統合されたソフトウェアの設計記述を対象として，詳細化関係の検証を行う手法を提案する．検証にはモデル検査ツールLTSAを用いる．本手法では，シーケンス図の振る舞いをLTSAの入力記法であるFSPによって表現する．さらに，与えられたhMSCに基づいて複数のシーケンス図を合成し，FSPによる表現を求める．また，本手法を例題システムへと適用してその有効性を示す．

# Refinement Check of Sequence Diagrams Using LTSA

Kazuma ASADA,[†1] Tomoyuki YOKOGAWA,[†1]
Hisashi MIYAZAKI[†2] and Yoichiro SATO[†1]

During a software development phase where a product is progressively elaborated, it is difficult to guarantee that the refined product retains its original behaviours. In this paper, we propose a method to detect refinement errors in UML sequence diagrams using LTSA (Labeled Transition System Analyzer). In this method, behaviors of sequence diagrams are represented by FSP (Finite State Process), which is an input language of LTSA. This method can integrate multiple sequence diagrams using hMSC (high-level Message Sequence Charts) and can translate it into a FSP description. In addition, this method supports several combined fragment operators of UML 2.0. We applied the method to some examples of refined sequence diagrams, and checked the correctness of refinement. The result shows this method can detect refinement errors in practical time.

## 1. はじめに

大規模なソフトウェアシステムの開発においては，システムは機能単位で複数のコンポーネントへと分割される．その上で，各機能を実現するためのコンポーネントの振る舞いを段階的に詳細化していき，得られた各コンポーネントの設計を統合することにより目的とするシステム設計を得る．しかしながら，コンポーネントの設計の詳細化および統合の際に，当初の設計において想定された振る舞いが保存されることは保証できない．詳細化における誤りは実装時のバグや要求仕様との不整合を引き起こし，開発コストが増大する要因となる．

UML[1)]のシーケンス図は，コンポーネント間の相互作用に着目してシステムの振る舞いを記述するための記法であり，このようなシステム開発において広く用いられているものの一つである．そのため，シーケンス図を対象とする詳細化関係の形式的検証はシステムの品質向上のために欠かせない技術であり，形式的検証を目的としたシーケンス図の意味論および詳細化関係の定義は既に提案されている[2),3)]．しかしながら，シーケンス図を検証可能な形式で表現し，その詳細化関係を検証するための試みは十分なされているとはいえない．

プロセス代数は，並行システムを形式的に記述するための手法であり，プロセス間の通信や同期等の相互作用をモデル化するために用いられる．また，プロセス代数では，プロセス記述の操作および解析を行うための代数的規則が提供されている．さらに，CSPに対するFDR[4)]やFSPに対するLTSA[5)]のように，いくつかのプロセス代数に対しては解析ツールが開発されている．

本論文では，形式的検証手法の一種であるモデル検査[6)]を用いて，設計間の詳細化関係の検証を行う手法を提案する．この手法では，シーケンス図で記述されたシステムの振る舞いをプロセス代数により表現し，モデル検査ツールLTSAを用いて詳細化関係の検証を行う．ここでは，文献[3)]で提案されたシーケンス図の意味論および詳細化関係の定義を採用している．本手法が対象とするシーケンス図は，UML2.0から導入された結合フラグメント記法の一部を含み，さらにhigh-level Message Sequence Chart (hMSC)[7)]を用いて統合可能である．

本手法では，まず，シーケンス図で記述されたソフトウェアシステムの振舞いを，LTSAの入力言語であるプロセス代数記法FSPによって表現する．そして，与えられたhMSCの記述に基づいて，各シーケンス図に対応するプロセスを合成し，統合されたプロセスを求め

る．最後に，詳細化および統合後のプロセスが，元のプロセスの振る舞いを全て保存していることを LTSA によって自動検証する．これにより，設計間の詳細化関係の自動検証を実現できる．

## 2. シーケンス図の詳細化

### 2.1 シーケンス図

シーケンス図は UML で定義された図の 1 つで，オブジェクト間のメッセージ通信系列を時系列に沿って表現するために用いる．図 1 にシーケンス図の例を示す．シーケンス図はオブジェクトの時系列に沿った振る舞いを表すライフラインとオブジェクト間で通信されるメッセージから構成される．オブジェクトはオブジェクト名が書かれた矩形で記述し，ライフラインはそこから下方向に引かれた破線で記述する．オブジェクト間のメッセージ通信はライフライン間の矢印で記述し，メッセージ名をラベルとして付ける．メッセージは同期メッセージと非同期メッセージに分類される．同期メッセージは黒く塗りつぶされた矢印で記述し，非同期メッセージは開いた矢印で記述する．

シーケンス図の意味論は，以下に示すように，メッセージの送受信処理を表すイベントオカレンスの順序関係に基づいて定義される．シーケンス図の動作は，ラベル付き半順序集合 $p = (E, L, \leq_E, \lambda)$ として表現される．ここで $E$ はイベントオカレンスの集合，$L$ はラベルの集合，$\leq_E$ は $E$ に関する半順序関係，そして $\lambda : E \to L$ は $E$ へのラベル付け関数である．ラベル $L$ は，シーケンス図の各メッセージ $m \in M$ の送信を表すラベル $m!$ および受信を表すラベル $m?$ から構成される．シーケンス図 $S$ の動作が $p$ によって表されていることを，$p \models S$ と書く．

$p$ の全順序への拡張 $p = (E', L' \leq'_E, \lambda')$ から得られるイベントオカレンスの列 $t = e_1 e_2 \cdots e_n$ をトレースという．ここで，$E' = E, L' = L, \lambda' = \lambda$ であり，$\leq'_E$ は全順序かつ $\leq'_E \supseteq \leq_E$ である．トレース $t$ の各イベントオカレンスを $\lambda'$ によってラベル付けして得られるラベルの列を $t^\lambda$ と書き，実行とよぶ．例として，図 1 のシーケンス図は実行 $m1!m1?m2!m2?$ をもつ．$p \models S$ であり，かつ $p$ から実行 $t^\lambda$ が得られることを，$t^\lambda \models S$ と書く．$t^\lambda \models S$ は，シーケンス図 $S$ において $t^\lambda$ で表されるメッセージ処理の系列が実行可能であることを表している．

### 2.2 結合フラグメント

結合フラグメントは，シーケンス図におけるメッセージ処理の分岐や繰り返し等を表現するためのもので，メッセージを囲むフレームによって記述する．本手法では，UML2.0 で導入された結合フラグメント記法のうち，並列 ($par$)，弱シーケンス ($seq$)，そして強シーケンス ($strict$) の 3 種を扱うことができる．

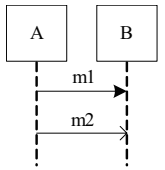



**図 1** シーケンス図

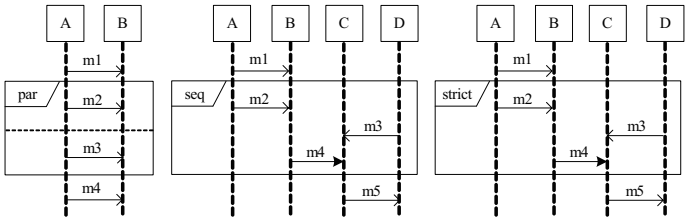



**図 2** 結合フラグメント

並列 ($par$) はフレーム内の破線で区切られた領域内のメッセージ通信が交互実行されることを表現する．図 2(a) の例では，$m2!m2?$ と $m3!m3?$ の処理が独立して行われる．弱シーケンス ($seq$) は，フレーム内のイベントオカレンスが通常のシーケンス図と同様の順序関係をもつことを表現する．図 2(b) の例では，二つのメッセージ通信 $m2$ および $m3$ は交互実行される．それに対して強シーケンス ($strict$) は，フレーム内の全てのメッセージ通信が，図中の縦方向の座標に従った線形順序で行われることを表現する．図 2(c) の例では，メッセージ通信 $m3$ は $m2$ の後に処理され，$m4$ は $m2$ の後に処理される．

### 2.3 hMSC

hMSC は複数のシーケンス図に対して，それらを実行する時間的な順序関係を記述するための図である．単体のシーケンス図は hMSC と区別して bMSC (basic Message Sequence Chart) と呼ぶ．図 3 に hMSC の例を示す．hMSC はノードとエッジから構成される．ノードは開始点および終了点と接続点，そして bMSC および他の hMSC への参照を表す．エッ

ジはノードの処理順序を表す．一つのノードから出た複数のエッジは制御の分岐を表し，制御の合流は接続点によって表す．

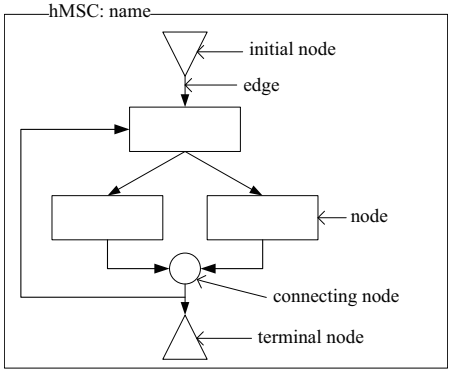



**図 3** hMSC

### 2.4 詳 細 化

文献[3] の定義に従い，本論文ではシーケンス図の詳細化関係を以下のように定義する．

**定義** 2 つのシーケンス図 $S$ および $S'$ に対して，任意の実行 $t^\lambda$ に対して $t^\lambda \models S' \Rightarrow t^\lambda \uparrow (L' \setminus L) \models S$ であるとき，かつそのときのみ，$S'$ は $S$ の詳細化であるといい，$S \rightsquigarrow S'$ と書く．ここで，$t^\lambda \uparrow L$ は $t^\lambda$ から $L$ に含まれるラベルを除いて得られる実行を表す．この定義は，シーケンス図 $S'$ を満足するような実行は必ず $S$ も満足することを表しており，詳細化を行うことによって新たな実行が追加されることがないことを表している．

シーケンス図の詳細化の例を図**??**に示す．図 4(a) のシーケンス図 ($S$ とよぶ) は，ユーザが ID およびパスワードをインタフェースへと送信するシステムを表している．これに対して図 4(b) および図 4(c)($S'$,$S''$ とよぶ) のシーケンス図では，ユーザが ID を送信した後，インタフェースがサーバに ID を送信し，その後サーバがインタフェースに認証を返す，というように処理が細分化されている．パスワードについても同様である．$S'$ では，インタフェースが認証を受け取った後にユーザにも認証を送るが，$S''$ ではユーザへの認証は送られない．

$S$ は $t = id!id?pw!pw?$ という実行のみをもつ．そして，$S'$ の実行は $t' = id!id?snd\_id!snd\_id?\cdots$ と，各メッセージを順に処理するもののみである．$t$ は $t'$ から $S$ に含まれないメッセージのラベルのみを除くことによって得ることができるため，詳細化関係 $S \rightsquigarrow S'$ は満たされる．

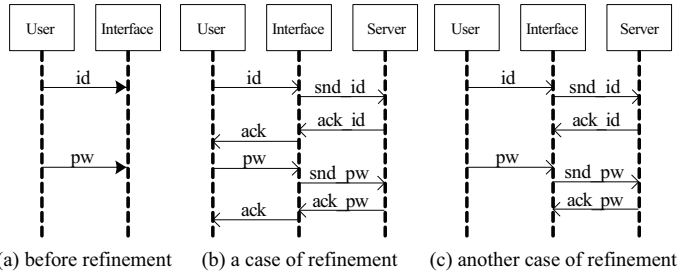



**図 4** シーケンス図の詳細化

一方で，$S''$ ではインタフェースがユーザに認証を送らないため，ユーザは $id$ を受けとる前に $pw$ を送ることができる．従って，$S''$ は $t'' = id!pw!id?\cdots$ という実行をもつが，$t''$ から $t$ は得られないため，$S''$ は $S$ の詳細化とはならない．

## 3. LTSA

LTSA (Labeled Transition System Analyser) は，有限オートマトンの集合としてモデル化されたシステムを対象とするモデル検査ツールである．LTSA では，システムがデッドロックを含んでいないことや，その他プロセスとして与えられた性質を満たすか否かを検証することが可能である．システムの記述には，有限状態プロセス (FSP) と呼ばれるプロセス代数形式の入力記述を用いる．LTSA では検証対象となるシステムの動作や検証結果をオートマトンで出力し，視覚的に確認することができる．また反例が存在する場合には，全ての反例を出力することが可能である．

### 3.1 FSP

FSP におけるプロセスは順序制約をもったアクションの集合であり，演算子->を用いて

$P$ = ($label_1$->$label_2$->$\cdots$->$label_m$->$P$)．

のように記述する．例として，アクション a および b の処理を繰り返すプロセス P は，以下のように定義される．

P = (a->b->P)．

また，FSP ではアクションの処理を行わないプロセス END が予め定義されており，プロセ

スの終了を表すために用いられる．以下のプロセス Q はイベント a，b を処理した後，終了する．

```
Q = (a->b->END).
```

上記のプロセス P は，以下のように定義することもできる．

```
P = (a->R),
R = (b->P).
```

このとき R を P のローカルプロセスとよび，カンマ (,) で区切って記述する．R のスコープは P の定義の中のみに限定される．

プロセスの並列合成は演算子||を用いて記述し，合成後のプロセスでは合成前の各プロセスのアクションが交互実行される．そして，合成前のプロセス間に共通するアクションが存在するならば，合成後のプロセスの動作はそのアクションで同期するよう制限される．例として，二つのプロセス Q と R の並列合成は以下のように記述される．

```
Q = (in->sync->sync->Q).
R = (sync->out->sync->R).
||P = (Q||R).
```

Q は in，sync，sync を，R は sync，out，sync をそれぞれ順に繰り返すプロセスであり，P は Q および R の並列合成プロセスである．Q および R は共通するアクション sync で同期するため，P は，in，sync，out，sync を繰り返すプロセスとなる．

アクションの選択による処理の分岐は演算子|で記述する．以下の例は，二つのアクション x と y のいずれかを最初に処理するプロセス P を記述している．

```
P = (x->Q|y->R).
```

もし x が処理された場合はその後 Q が実行され，y が処理されたならば R が実行される．共通のアクションを持つ場合，その後に実行されるプロセスは非決定的に選ばれる．

プロセスの逐次実行は演算子; で記述する．演算子; の直前のプロセスは，END で終了するよう定義する必要がある．以下に例を示す．

```
Q = (x->END).
R = (y->END).
P = Q;R;P.
```

Q はアクション x の処理後に終了し，R はアクション y の処理後に終了するプロセスである．P はプロセス Q および R の処理を順次繰り返すプロセス，すなわち x，y の処理を繰り返すプロセスである．

演算子 \ を用いることで，アクションを他のプロセスから隠蔽することが可能である．隠蔽されたアクションは tau で表される．以下の Q，R は前述の例と同様の動作を行うプロセスである．

```
Q = (in->sync->sync->Q)\{sync}.
R = (sync->out->sync->R).
||P = (Q||R).
```

Q と R に共通するアクション sync を持つが，Q において sync は隠蔽されるため，P はこれらの 2 つのプロセスが独立して並行動作するプロセスとなる．

### 3.2 安全性検証

安全性は，システムにおいて受理されるような振る舞いの集合として表現される．LTSA では，この安全性をプロセスによって与えることができる．安全性を表すプロセスは，プロセスの定義の際にキーワード property をつけることにより記述する．例として，以下のプロセス SPC は，x および y が必ず交互に処理されるという性質を表している．

```
property SPC = (x->y->SPC).
```

この性質は図 5 のオートマトンで表される．図に示すとおり，与えられた性質を満たさないアクションの系列に対して，SPC は −1 でラベル付けされた拒否状態へと遷移する．

LTSA では，このように定義されたプロセスを検証の対象となるプロセスと並列合成することで，安全性の検証を行う．もし，検証対象プロセスにおけるアクションの系列が全て受理されるならば，プロセスは制約を受けることなく実行可能であり，拒否状態に遷移することはない．しかし安全性が満たされない場合，安全性を表すプロセスが拒否状態へと遷移するため合成後のプロセスはデッドロック状態に陥り，誤りが検出される．このとき LTSA は，拒否状態へと遷移する処理系列を表すオートマトンを反例として出力する．

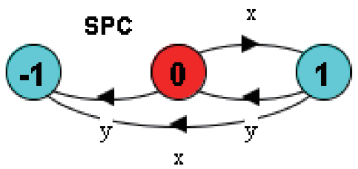



**図 5** プロセス SPC に対するオートマトン

例として，以下のプロセス P1 および P2 が，前述の安全性 SPC を満たすか否かの検証を

行う．プロセス P1 は x，a，y の処理を，プロセス P2 は x，y，x の処理を繰り返すプロセスである．

```
P1 = (x->a->y->P1).
P2 = (x->y->x->P2).
||REF1 = (SPC||P1).
||REF2 = (SPC||P2).
```

REF1 は SPC と P1 の並列合成であり，REF2 は SPC と P2 の並列合成である．検証結果として得られたオートマトンを図 6 および図 7 に示す．REF1 は拒否状態に遷移せず，x，a，y を繰り返し処理するという P1 と同じ処理を表しており，P1 が SPC の性質を満たすことがわかる．一方で，REF2 のオートマトンは x，y，x と処理を行った後に，x が処理されることで拒否状態へと遷移している．この結果から，P2 は SPC で表される性質を満たさないことがわかり，さらに SPC に違反する処理系列を図 7 のオートマトンとして得ることができる．

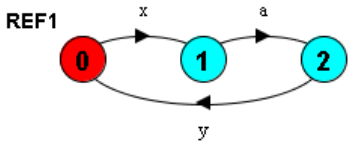



**図 6** プロセス REF1 に対するオートマトン

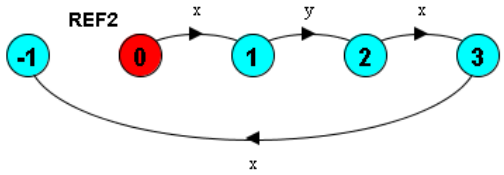



**図 7** プロセス REF2 に対するオートマトン

## 4. LTSA による詳細化検証

### 4.1 FSP によるシーケンス図の表現

本手法では，シーケンス図のイベントオカレンス $e_{i_j}$ をそれぞれ FSP のアクションとして表現する．ここで $e_{i_j}$ は，ライフライン $Obj_i$ の $j$ 番目のイベントオカレンスを表すものとする．ライフラインは，ライフライン上のイベントオカレンスに対応したアクションを処理するプロセスとして表現される．ここで，アクションの順序は，イベントオカレンス間の順序関係を表している．さらに，メッセージの送受信に関する順序関係を同様にプロセスとして定義する．最後に，このように定義されたプロセス全ての並行動作プロセスとして，シーケンス図で実行可能なトレースの集合を表現することができる．

$Obj_i$ 上に同期メッセージを送信するイベントオカレンスが存在しない場合，$Obj_i$ を表すプロセス Obj_i は以下のように定義できる．

```
Obj_i = (e_i_1->e_i_2->···->e_i_n->END).
```

ここで，e_i_j は $e_{i_j}$ に対応するアクションである．

$Obj_i$ 上のイベントオカレンス $e_{i_j}$ が同期メッセージの送信であった場合，それ以降のイベントオカレンスは同期メッセージの受信 ($e_{k_l}$ とする) が完了するまで処理できない．従って，このようなプロセス Obj_i は以下のように定義できる．

```
Obj_i = (e_i_1->e_i_2->···->e_i_j->e_k_l->e_i_j+1->···->e_i_n->END).
```

また，メッセージの送受信に関する順序関係も同様に FSP のプロセスとして表現する．$m!$ または $m?$ とラベル付けされたイベントオカレンス $e_{i_j}$ は，それぞれアクション m_snd および m_rcv として表現する．メッセージ $m_k$ の送受信に関する順序関係を表すプロセス M_k は以下のように定義できる．

```
M_k = (m_k_snd->m_k_rcv->M_k).
```

シーケンス図において可能な実行の集合は，イベントオカレンスの順序関係を表すプロセスの並列合成として求めることができる．従って，シーケンス図 $S$ は以下のプロセス S として定義できる．

```
S = (Obj_1|···||Obj_N||M_1||···||M_L).
```

### 4.2 FSP による結合フラグメントの表現

本手法では，フレームの処理の開始と終了を表すアクションを定義し，そのアクションによってフレーム内外の処理を同期することで，結合フラグメントをもつシーケンス図をプロセスとして表現する．

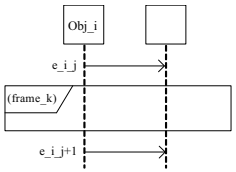



**図 8** 結合フラグメントを含むシーケンス図

図 8 に示すように，$Obj_i$ において，フレーム $frame_k$ が $e_{i_j}$ と $e_{i_{j+1}}$ の間にあるとする．本手法では，フレーム $frame_k$ の処理の開始と終了をアクション `f_k_bgn` および `f_k_end` でそれぞれ表し，その上で $Obj_i$ を表すプロセス `Obj_i` を，`f_k_bgn` および `f_k_end` を用いて以下のように定義する．

`Obj_i = (e_i_1->e_i_2->`$\cdots$`->e_i_j->f_k_bgn`
`->f_k_end->e_i_j+1->`$\cdots$`->e_i_n->END)`.

次に，フレーム $frame_k$ 内部の処理を表すプロセス `F_k` を，結合フラグメントの種類に応じて定義する．並列 ($par$) では，フレーム内の各領域のメッセージは並行して処理される．従って，フレーム $frame_k$ の各領域内の処理を通常のシーケンス図と同様にプロセスとして表現し，その上で領域の開始および終了をアクションによって同期することで，$par$ フレームの動作を表現できる．領域の開始および終了の同期は，各プロセスの先頭と末尾に共通のアクション `f_k_bgn`，`f_k_end` の処理をそれぞれ付け加えることで実現できる．

この $par$ フレーム $frame_k$ が $n$ 個の領域 $r_{k_1}, r_{k_2}, \cdots r_{k_n}$ をもつとする．領域 $r_{k_l}$ 内での $Obj_i$ の動作が $prefix_{k_l \cdot i}$`->END` というプロセスで定義されていたとするならば，領域における各ライフラインの振る舞いは以下のように定義される．

`R_k_l_i = (f_k_bgn->`$prefix_{k_l \cdot i}$`->f_k_end->END)`.

$par$ フレーム $frame_k$ 内の処理は，全ての領域 $r_{k_l}$ における各 $Obj_i$ に対するプロセス `R_k_l_i` の並列合成として表される．従って，フレーム $frame_k$ 内部の処理を表すプロセス `F_k` は以下のように定義できる．

`||F_k = (R_k_1_1||R_k_1_2||`$\cdots$
$\vdots$
`R_k_n_1||R_k_n_2||`$\cdots$`)`.

弱シーケンス ($seq$) のフレーム内では，通常のシーケンス図と同様の順序でイベントオカレンスが処理される．従って，フレーム $frame_k$ 内を通常のシーケンス図と同様にプロセスとして表現し，各プロセスの先頭と末尾にアクション `f_k_bgn`，`f_k_end` の処理をそれぞれ付け加える．$seq$ フレーム $frame_k$ 内での $Obj_i$ の動作が $prefix_{k \cdot i}$`->END` というプロセスで定義されていたとするならば，$frame_k$ の振る舞いは以下のように定義される．

`F_k_i = (f_k_bgn->`$prefix_{k \cdot i}$ `->f_k_end->END)`.

フレーム $frame_k$ 内の処理を表すプロセス `F_k` は，全ての $Obj_i$ に対して求められたプロセス `F_k_i` の並列合成として以下のように定義できる．

`||F_k = (F_k_1||F_k_2||`$\cdots$`)`.

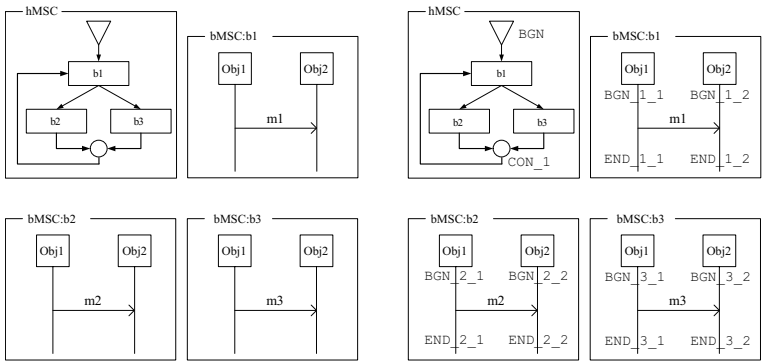
最後に，強シーケンス ($strict$) のフレーム内では，全てのメッセージ通信はあらかじめ定められた線形順序に従って処理される．従って，フレーム $frame_k$ 内の処理は対応するアクションを定められた線形順序に従って処理するプロセスとして表現できる．その上で，上記の二つと同様に，プロセスの先頭と末尾にアクション `f_k_bgn`，`f_k_end` の処理をそれぞれ付け加える．$strict$ フレーム $frame_k$ 内のメッセージが $m1, m2, \cdots$ の順序であったとすると，フレーム $frame_k$ 内の処理を表すプロセス `F_k` は以下のように定義できる．

`F_k = (f_k_bgn->m1_snd->m1_rcv->m2_snd->m2_rcv`$\cdots$ `->f_k_end->END)`.

### 4.3 hMSC に基づいたシーケンス図の合成

本節では，hMSC から参照されている bMSC のプロセスを hMSC に従って合成し，1 つのプロセスを求める手法について示す．まず，hMSC の開始記号に `BGN`，終了記号に `END`，そして全ての接続点に `CON_1`, $\cdots$ ,`CON_M` とラベルを付ける．さらに，hMSC で参照された bMSC$b_k$ の各ライフライン $Obj_i$ の先頭と末尾に，それぞれ `BGN_k_i` と `END_k_i` とラベルを付ける．図 9 は，hMSC および bMSC へのラベル付けの例である．

各ライフライン毎に，hMSC のエッジの始点と終点に対応するラベルの組を抽出し，連結することでローカルプロセスとして表現する．bMSC$b_k$ における $Obj_i$ の処理は，プロセス `BGN_k_i` およびローカルプロセス `END_k_i` として表現する．例として，図 9 のライフライン $Obj_1$ は，以下のプロセス `Obj_1` として定義される．




(a) Examples of hMSC and bMSCs (b) Labeled hMSC and bMSCs

**図 9** hMSC と bMSC へのラベル付加

```
  Obj_1 = BGN,
    BGN = BGN_1_1,
BGN_1_1 = (m1_snd->END_1_1),
END_1_1 = (BGN_2_1|BGN_3_1),
BGN_2_1 = (m2_snd->END_2_1),
END_2_1 = CON_1,
BGN_3_1 = (m3_snd->END_3_1),
END_3_1 = CON_1,
  CON_1 = BGN_1_1.
```

アクションを含まないローカルプロセスは，以下のようにプロセスを置き換えることによって記述を省略する．例として，ローカルプロセス BGN および BGN_1_1 はアクションを持たないため，以下のように省略できる．

```
Obj_1 = (m1_snd->END_1_1),
```

同様に簡略化を繰り返すことにより，最終的にプロセス Obj_1 は以下のように求められる．

```
  Obj_1 = BGN_1_1,
BGN_1_1 = (m1_snd->END_1_1),
END_1_1 = (BGN_2_1|BGN_3_1),
BGN_2_1 = (m2_snd->BGN_1_1),
BGN_3_1 = (m3_snd->BGN_1_1).
```

統合されたシステム全体の動作を表すプロセス SYS は，以下のように並行動作プロセスとして定義できる．

```
||SYS = (Obj_1||Obj_2||M_1||M_2||M_3).
```

### 4.4 詳細化検証手法

本手法では，$S'$ の全ての実行が $S$ の実行となるならば，$S \rightsquigarrow S'$ であると定義している．ここで LTSA は検証対象のプロセスが，安全性プロセスの振る舞いを全て満たすか否かを検証できるため，$S'$ を検証対象，$S$ を安全性を表すプロセスとして安全性検証を行うことにより，詳細化関係の検証が実現できる．検証対象プロセスを SYS，安全性プロセスを SPC とすると，それらの並列合成は以下のように定義される．

```
property ||SPC = (Obj_1||···).
         ||SYS = (Obj_1||···)\{m_snd,···}.
         ||REF = (SYS||SPC).
```

SYS に含まれており，かつ SPC に含まれないアクションは隠蔽される．これにより，もし SYS の振る舞いの中に SPC を満たさないものがあるならば，プロセス REF のオートマトンは拒否状態へと遷移し，誤りが検出される．すなわち，詳細化関係が満たされないことが検出できる．また，このとき，拒否状態へと遷移するアクションの示すイベントオカレンスが誤り箇所となる．

## 5. 検 証 例

図 10 は，あるシーケンス図とその二つの詳細化例を示している．図 10(a) のシーケンス図は strict フレームを持つ．図 10(b) のシーケンス図は $Obj\_2$ が新たなオブジェクト $Obj\_3$ に対して二つのメッセージ通信 $m4$ および $m5$ を行うよう詳細化したものである．図 10(c) のシーケンス図も図 10(b) と同様のメッセージ通信を行うものであるが，strict フレームが seq フレームへと置き換えられている．従って，図 10(c) のシーケンス図は，元のシーケンス図には存在しない実行 $\cdots m2!m3!m2?m3?\cdots$ が可能となっている．これにより，図 10(a) と (c) は詳細化関係を満たさない．以下，本手法を用いることで，LTSA によりこの詳細化の誤りが正しく検出できることを示す．

詳細化前のシーケンス図は以下のプロセス SYS として定義する．

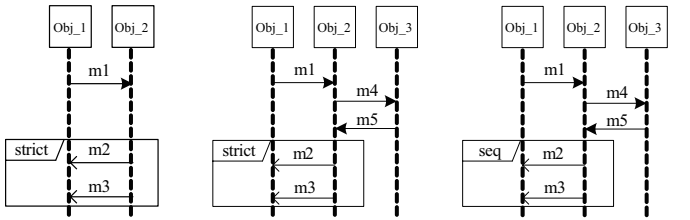



**図 10** 詳細化例

```
Obj_1 = (m1_snd->m1_rcv->f_1_bgn->f_1_end->END).
Obj_2 = (m1_rcv->f_1_bgn->f_1_end->END).
  F_1 = (f_1_bgn->m2_snd->m2_rcv->m3_snd->m3_rcv->f_1_end->END).
  M_1 = (m1_snd->m1_rcv->M_1).
  M_2 = (m2_snd->m2_rcv->M_2).
  M_3 = (m3_snd->m3_rcv->M_3).
property||SPC = (Obj_1||Obj_2||M_1||M_2||M_3||F_1).
```

SPC は安全性プロセスとして定義する．

次に，図 10(b) のシーケンス図は，以下のプロセス SYS1 として定義する．

```
R1_Obj_1 = (m1_snd->m1_rcv->f_1_bgn->f_1_end->END).
R1_Obj_2 = (m1_rcv->m4_snd->m4_rcv->m5_rcv->f_1_bgn->f_1_end->END).
R1_Obj_3 = (m4_rcv->m5_snd->END).
  R1_F_1 = (f_1_bgn->m2_snd->m2_rcv->m3_snd->m3_rcv->f_1_end->END).
     M_4 = (m4_snd->m4_rcv->M_4).
     M_5 = (m5_snd->m5_rcv->M_5).
||SYS1 = (R1_Obj_1||R1_Obj_2||R1_Obj_3||M_1||M_2||M_3||M_4||M_5||R1_F_1)
         \{m4_snd,m4_rcv,m5_snd,m5_rcv}.
```

図 10(c) のシーケンス図も同様に，以下のプロセス SYS2 として定義する．

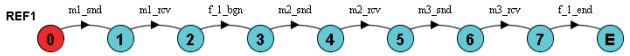



**図 11** プロセス REF1 に対するオートマトン

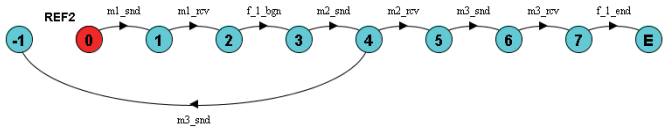



**図 12** プロセス REF2 に対するオートマトン

```
R2_Obj_1 = (m1_snd->m1_rcv->f_1_bgn->f_1_end->END).
R2_Obj_2 = (m1_rcv->m4_snd->m4_rcv->m5_rcv->f_1_bgn->f_1_end->END).
R2_Obj_3 = (m4_rcv->m5_snd->END).
R2_F_1_1 = (f_1_bgn->m2_rcv->m3_rcv->f_1_end->END).
R2_F_1_2 = (f_1_bgn->m2_snd->m3_snd->f_1_end->END).
||SYS2 = (R2_Obj_1||R2_Obj_2||R2_Obj_3||M_1||M_2||M_3||M_4||M_5
        ||R2_F_1_1||R2_F_1_2)\{m4_snd,m4_rcv,m5_snd,m5_rcv}.
```

そして，詳細化関係は，検証対象プロセス SYS1 および SYS2 と安全性プロセス SPC を以下のように並列合成することにより検証できる．

```
||REF1 = (SPC||SYS1).
||REF2 = (SPC||SYS2).
```

図 11 および 12 は，REF1 および REF2 に対して得られたオートマトンである．

図 11 に示す通り，REF1 に対するオートマトンは拒否状態に遷移することはない．従って，図 10(b) は図 10(a) を正しく詳細化したものであるといえる．これに対して図 12 に示す通り，REF2 のオートマトンは状態 4 でイベント m3_snd が処理されることにより拒否状態に遷移する．これにより図 10(c) と図 10(a) の詳細化関係には誤りがあることが正しく検出され，さらに $m3$ の送信処理が詳細化後のシーケンス図において想定されない処理であることが示された．

## 6. 関 連 研 究

シーケンス図の意味論の定義については種々のものが存在し，その振る舞いをモデル化す

るための手法についても数多く開発されている．

中でも，プロセス代数に基づいてシーケンス図の振る舞いをモデル化する手法は，数多く報告されている[8)–11)]．Korenblat ら[8)] は，シーケンス図および状態マシン図をπ-calculus を用いてモデル化する手法を提案している．これにより，シーケンス図と状態マシン図という異なる図の一貫性の検証を行うとともに，これらの図に形式的な意味論を与えることに成功している．

Tribastone ら[9)] は，シーケンス図の定量的な評価を行うための手法を提案している．この手法では，確率プロセス代数 PEPA[12)] を，UML プロファイルの一つである MARTE[13)] のパフォーマンス計測器として用いている．この中で，MARTE によって拡張されたシーケンス図を PEPA による記述へと変換する手法を示し，変換の手続きの自動化を行っている．

Uchitel ら[10)] は，シーケンス図の振る舞いを FSP へと合成し，LTSA による検証を行う手法を提案している．この手法では，hMSC を用いて複数のシーケンス図を統合することも可能である．また，既存のモデル統合手法を含めた開発環境を実装している．

これらの手法は，シーケンス図の振る舞いをメッセージ通信の半順序関係と考え，その上でプロセス代数により表現するものである．しかしながら，詳細化関係の検証を行うには，詳細化前後のシーケンス図の振る舞いをより詳細にモデル化する必要がある．本手法では既存の手法とは異なり，イベントオカレンスの半順序関係としてシーケンス図をモデル化することが可能である．

一方で，Mitchell[11)] は，MRA(Mauw and Reniers' Algebraic)[14)] を拡張したプロセス代数を新たに提案し，それを用いてシーケンス図を記述している．この手法では，シーケンス図の振る舞いをイベントオカレンスの半順序関係として表現されている．彼の提案するプロセス代数は，ライフライン上の位置関係ではなく，メッセージ処理の因果関係に基づいた意味論である，causal semantics[15)] を採用している．

本手法では，文献[3)] で提案されたシーケンス図の意味論に基づいて定義された詳細化関係の検証を目的としている．この定義では標準的なシーケンス図の意味論を採用しているため，Mitchell のプロセス代数を利用することはできない．

プロセス代数以外の方法によりシーケンス図をモデル化する試みも存在する[16),17)]．Zhao ら[16)] はモデル検査ツール SPIN を用いて，状態マシン図とシーケンス図の動作の整合性を検証する手法を提案している．ここで，シーケンス図の振る舞いはオートマトンを用いて表現されている．また，Bernardi ら[17)] は状態マシン図とシーケンス図の振る舞いを，一般確率ペトリネットによるモデルへと変換する手法を提案している．しかしながら，Zhao らの手法は一つのメッセージ通信をオートマトンの一つの遷移に対応させてモデル化を行っているため，シーケンス図の全ての実行をモデル化することができない．また，Bernardi らの手法も，ペトリネットへの変換を行う際に全てのイベントオカレンスに全順序を定める必要があるため，同様に全ての実行が表現できない．

これまでに，シーケンス図に関しては，時間に関する整合性[18),19)] や，実時間制約[20)]，さらには他の UML 図との間の整合性[8),16),17)] 等のように，様々な特性に着目した検証が行われてきたが，これらはシーケンス図間の詳細化関係の検証を目的としたものではなかった．これに対して本手法では，詳細化検証を目的としてシーケンス図のモデル化を行うものである．

## 7. ま と め

本稿では，UML のシーケンス図で記述され，hMSC に基づいて統合されたソフトウェアの設計記述を対象として，設計間の詳細化関係の検証を行う手法を提案した．まず，シーケンス図をモデル検査ツール LTSA の入力言語 FSP で表現するための手法を提案した．また 3 種の結合フラグメント記法を含むシーケンス図を FSP で表現するための手法についても示した．さらに，与えられた hMSC 記述に従ってシーケンス図を合成し，FSP 表現を得るための手法を示した．最後に，適用実験を通して本手法により LTSA による詳細化検証が正しく実現されることを示し，詳細化関係に誤りを持つ場合には，LTSA の反例として誤り箇所が検出されることを確認した．

本手法では，シーケンス図における正常な振る舞いのみを実行として考えていたが，結合フラグメントの neg および assert といった演算子を扱うには，異常な振る舞い[2)] という概念を導入する必要がある．また，与えられたシーケンス図から，FSP への変換の自動化等が今後の課題として挙げられる．

## 参 考 文 献

1) Object Management Group: *Unified Modeling Language Specification Version 2.0* (2004). http://www.uml.org.
2) Störrle, H.: Assert, Negate and Refinement in UML-2 Interactions, *2nd Intl. Workshop on Critical Systems Development with UML (CSDUML '03, Proceedings).*, Technische Universität München, pp.79–93 (2003).
3) Cengarle, M. V. and Knapp, A.: UML 2.0 Interactions: Semantics and Refinement., *3rd Intl. Workshop on Critical Systems Development with UML (CSDUML*

*'04, Proceedings)*. (Jürjens, J., Fernandez, E.B., France, R. and Rumpe, B., eds.), Technische Universität München, pp.85–99 (2004).
4) Hoare, C. A.R.: Communicating sequential processes, *Commun. ACM*, Vol.21, pp. 666–677 (online), DOI:http://doi.acm.org/10.1145/359576.359585 (1978).
5) Magee, J. and Kramer, J.: *Concurrency: state models & Java programs*, John Wiley & Sons, Inc., New York, NY, USA (1999).
6) Clarke, E., Grumberg, O. and Peled, D.: *Model Checking*, MIT Press (1999).
7) Alur, R. and Yannakakis, M.: Model Checking of Message Sequence Charts, *Proceedings of the 10th International Conference on Concurrency Theory*, CONCUR '99, pp.114–129 (1999).
8) Pokozy-Korenblat, K. and Priami, C.: Toward Extracting pi-calculus from UML Sequence and State Diagrams, *Electron. Notes Theor. Comput. Sci.*, Vol.101, pp. 51–72 (2004).
9) Tribastone, M. and Gilmore, S.: Automatic Translation of UML Sequence Diagrams into PEPA Models, *Proceedings of the 2008 Fifth International Conference on Quantitative Evaluation of Systems*, pp.205–214 (2008).
10) Uchitel, S. and Kramer, J.: A workbench for synthesising behaviour models from scenarios, *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pp.188–197 (2001).
11) Mitchell, B.: Characterizing Communication Channel Deadlocks in Sequence Diagrams, *IEEE Trans. Softw. Eng.*, Vol.34, pp.305–320 (2008).
12) Hillston, J.: *A Compositional Approach to Performance Modelling (Distinguished Dissertations in Computer Science)*, Cambridge University Press, New York, NY, USA (2005).
13) Object Management Group: UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE).
14) Mauw, S. and Reniers, M.: An Algebraic Semantics of Basic Message Sequence Charts, *The Computer Journal*, Vol.37, pp.269–277 (1994).
15) Sibertin-Blanc, C., Tahir, O. and Cardoso, J.: Interpretation of UML Sequence Diagrams as Causality Flows, *ISSADS* (Corchado, F. F.R., Larios-Rosillo, V. and Unger, H., eds.), Lecture Notes in Computer Science, Vol.3563, Springer, pp.126–140 (2005).
16) Zhao, X., Long, Q. and Qiu, Z.: Model Checking Dynamic UML Consistency, *ICFEM* (Liu, Z. and He, J., eds.), Lecture Notes in Computer Science, Vol.4260, Springer, pp.440–459 (2006).
17) Bernardi, S., Donatelli, S. and Merseguer, J.: From UML sequence diagrams and statecharts to analysable petrinet models, *Workshop on Software and Performance*, pp.35–45 (2002).
18) Li, X. and Lilius, J.: Timing Analysis of UML Sequence Diagrams, *UML* (France, R.B. and Rumpe, B., eds.), Lecture Notes in Computer Science, Vol.1723, Springer, pp.661–674 (1999).
19) Li, X. and Lilius, J.: Checking compositions of UML sequence diagrams for timing inconsistency, *APSEC*, IEEE Computer Society, pp.154–161 (2000).
20) Lano, K.: Formal Specification using Interaction Diagrams, *Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods*, pp. 293–304 (2007).