社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

プロセス代数CSPによるシーケンス図設計の詳細化と検証

海津　智宏† 　磯部　祥尚†† 　鈴木　正人†

† 北陸先端科学技術大学院大学
†† 産業技術総合研究所
E-mail: †{tkaizu,suzuki}@jaist.ac.jp, ††y-isobe@aist.go.jp

あらまし　ソフトウェアのモジュール構造の設計には，シーケンス図が用いられる場合が多い．本論文では，プロセス代数 CSP を利用して，複数のシーケンス図から並行システムを合成し，詳細化関係の正しさや設計が要求を満たすかを検証する手法を提案する．本手法では，シーケンス図の合成に適した新しい演算子を CSP に追加する．また，この手法を実現するツールを開発し，効果を確認したことを報告する．
キーワード　シーケンス図，プロセス代数，CSP，プロセス合成

Refinement and Verification of Sequence Diagrams Using the Process Algebra CSP

Tomohiro KAIZU†, Yoshinao ISOBE††, and Masato SUZUKI†

† Japan Advanced Institute of Science and Technology
†† National Institute of Advanced Industrial Science and Technology
E-mail: †{tkaizu,suzuki}@jaist.ac.jp, ††y-isobe@aist.go.jp

Abstract　Sequence diagrams are often used in the modular design of softwares. In this paper, we propose a method to verify correctness of sequence diagrams. With this method, using the process algebra CSP, you can synthesize concurrent systems from multiple sequence diagrams. We define a new CSP operator for the synthesis of sequence diagrams. We also developed a tool to achieve this technique, and confirmed effectiveness of this approach.
Key words　Sequence Diagram, Process Algebra, CSP, Process Synthesis

1. はじめに

近年，ソフトウェアシステムに求められる機能はますます複雑化・多様化している．機能の複雑化に伴ってソフトウェアシステムの設計が複雑化すると，追跡容易性・再利用性・変更容易性・拡張性などが低くなってしまう．このような問題に対処するために，コンポーネントベースの開発が広まっている．

コンポーネントの設計には，OMG によって開発・標準化された UML を利用することが多い．特に，開発の上流工程では，コンポーネント挙動の理解と検証を目的としてシーケンス図がよく利用されている．

しかし，シーケンス図は多くの場合厳密には書かれておらず，曖昧性を含んだまま記述されているため，シーケンス図で記述されたふるまいを機械的に扱うことは難しかった．そのため，シーケンス図間の整合性や詳細化の正しさなど，問題点の発見は人手によるレビューに頼っていた．レビューで見落とされた問題については後工程で問題が顕在化するため，手戻りが発生し，開発期間やコストに大きな影響を与えてしまう．

そこで，本論文では，シーケンス図に記述された各コンポーネントのふるまいをプロセス代数 CSP (Communicating Sequential Processes) [1], [2] による形式的記述に変換することで，システムの整合性や詳細化の正しさを形式的に検証可能とする手法を提案する．

本手法では，シーケンス図を元に各シナリオにおけるコンポーネントのふるまいを CSP の記述に変換する．さらに，複数のシーケンス図から得られる CSP 記述を合成し，各コンポーネントの総合的なふるまいを得る．このふるまいは CSP を用いて形式的に記述されたものなので，FDR などの検査器を用いてその性質を形式的に検証できる．

複数の CSP 記述の合成にあたっては，CSP の持つ内部選択という仕組みを活用することで，シーケンス図の持つ曖昧性を適切に記述できる．本手法では，CSP の合成方法を論理的に定義するため，シーケンス図合成演算子 $\circ$ を新たに定義する．演算子を定義してその性質を確認することで，この合成手法がシーケンス図の合成に適した性質と検証に適した性質とを併せ持つことを確認できる．

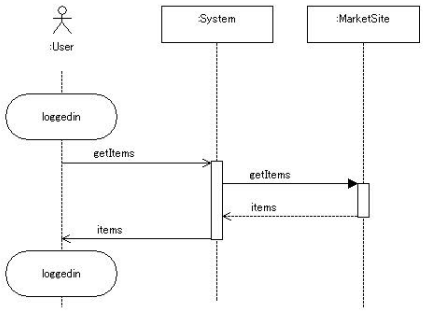

図 1　シーケンス図の例

本論文では，演算子 ∘ による合成を行うツールを実際に開発し，例題を用いて効果を確認したことを報告する．

2. 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の流れを時系列的に表現する．UML は OMG によって開発・標準化されたオブジェクトモデリングのための仕様記述言語である．

シーケンス図は，ライフライン，メッセージ，活性区間，状態不変式などを組み合わせてメッセージ送受信を表現する．図 1 はシーケンス図の例である．

ライフラインは，縦に引かれた点線で記述され，オブジェクトの存在を示す．縦軸は時間の流れを表し，シーケンス図の上から下に向かって時間が流れていくことを示す．ライフラインの先頭には矩形もしくはスティックマン（棒人間）形状でそのライフラインがどのオブジェクトを表すかを記述する．矩形のオブジェクトはシステム内のコンポーネントを表し，スティックマン形状のオブジェクトはシステム外部のアクターを表す．

送受信されるメッセージはライフライン間の矢印で記述される．メッセージの種類により，同期メッセージであれば三角形が塗りつぶされた矢印，非同期メッセージであれば塗りつぶされない矢印，同期メッセージの戻りであれば点線の矢印で記述される．

活性区間はライフライン上に置かれた縦長の矩形であり，そのオブジェクトが動作している期間を表す．手続き型のプログラミング言語におけるメソッドや関数の実行に対応する．

状態不変式はライフライン上に置かれた楕円形で記述され，その時点でのオブジェクトの状態を表す．状態不変式には，状態名もしくはその時点で成立する制約の式が記述される．

3. プロセス代数 CSP

3.1 CSP の概要

プロセス代数とは並行プロセスを記述し解析するための理論である．プロセス代数の 1 つに CSP がある [1], [2].

CSP では，発生するイベントを単位として並行プロセスの動作を記述することができる．記述された並行プロセスはモデル検査器 FDR を利用することにより，デッドロックがないこと，

$$
\begin{aligned}
P1 &= in \rightarrow sync \rightarrow sync \rightarrow P1 \\
P2 &= sync \rightarrow out \rightarrow sync \rightarrow P2 \\
SYS &= P1 [\![\{sync\}]\!] P2
\end{aligned}
$$

図 2　CSP による並行プロセスの記述例

ライブロックがないこと，システム間に詳細化関係が成り立つことなどを検証できる [3]．さらに，JCSP という実装ライブラリがあり，CSP で記述されたプロセスの構造を保ったままシステムを実装することができる [4].

CSP のプロセスは，プロセス名，プレフィックス，外部選択，内部選択，並行合成などの組み合わせで記述される．

プロセス名は，プロセスを表す記号である．CSP のプロセスの多くは「プロセス名 = 動作式」という等式で定義される．

プレフィックスは，プロセスが実行するイベントを記述する．「$a \rightarrow P$」（FDR では「`a -> P`」）という形で記述され，イベント a を実行することができ，実行後は P のように動作するプロセスを表す．

外部選択は「$P \Box Q$」（FDR では「`P [] Q`」）という形で記述され，P または Q のように動作するプロセスを表す．次に実行するイベントによって P と Q のうち実行可能なものが選択される．この選択は外部から制御できる．

内部選択は「$P \sqcap Q$」（FDR では「`P |~| Q`」）という形で記述され，P または Q のように動作するプロセスを表す．この選択は内部的な状態によって定まり，外部からは制御できない．

並行合成は「$P [\![X]\!] Q$」（FDR では「`P [|X|] Q`」）という形で記述され，イベントの集合 X で同期しつつ，P と Q が並行に動作するプロセスを表す．P と Q との間で受け渡されるメッセージをイベントとして同期することで，メッセージ送受信を CSP で表現できる．

図 2 は CSP による並行プロセスの記述例である．P1 は in, sync，sync を繰り返すプロセス，P2 は sync，out，sync を繰り返すプロセスである．並行プロセス SYS では P1 と P2 が sync で同期して並行に動作し，P1 が in を独立に実行，P1 と P2 が sync で同期，P2 が out を独立に実行，P1 と P2 が sync で同期，という動作を繰り返す．

3.2 CSP の等価性と詳細化関係

CSP では，プロセス間の等価性・詳細化関係を定義する方法として，トレースモデル・失敗モデル・失敗発散モデルの 3 種類のモデルが定義されている．

トレースモデルでは，そのプロセスが実行できるイベント列（トレース）を用いて等価性・詳細化関係を定義する．トレースの集合が等しいプロセスはトレース等価であり，トレースの集合が部分集合になっていればプロセス間には詳細化関係がある．

プロセス P のトレース集合 $traces(P)$ は P の構造に関して定義される．例えば，$\sqcap$ と $\Box$ では次のように定義される．

$$
\begin{aligned}
traces(P \sqcap Q) &= traces(P) \cup traces(Q) \\
traces(P \Box Q) &= traces(P) \cup traces(Q)
\end{aligned}
$$

失敗モデルでは，トレース集合に加えて失敗集合を用いて等価性・詳細化関係を定義する．失敗集合は，特定のイベント列に対して，失敗する可能性があるイベントの集合である．トレース集合と失敗集合がともに部分集合になっていればプロセス間には詳細化関係がある．

プロセス P の失敗集合は P の構造に関して定義される．例えば，⊓ と □ では次のように定義される．

$$failures(P \sqcap Q) = failures(P) \cup failures(Q)$$

$$failures(P \,\Box\, Q) =$$
$$\{(\langle\rangle, X) | (\langle\rangle, X) \in failures(P) \cap failures(Q)\}$$
$$\cup \{(s, X) | s \neq \langle\rangle, (s, X) \in failures(P) \cup failures(Q)\}$$

ここで，成功終了イベントは簡単のために省略している．

失敗発散モデルでは，失敗集合と発散トレース集合とを用いて等価性・詳細化関係を定義する．発散トレース集合は実行後にライブロックするイベント列の集合である．失敗モデルではライブロックを正しく扱うことができないが，失敗発散モデルを使うことでライブロックを保存できるようになる．

今回扱うシーケンス図の合成においてはライブロックは生じないので，今後，特に断りのない限り失敗モデルで等価性や詳細化関係を議論する．

3.3 シーケンス図合成演算子 ∘

シーケンス図で記述されるコンポーネントのふるまいを複数合成して総合的なふるまいを得るために，シーケンス図の合成を表す新しい演算子 ∘ を定義する．

シーケンス図合成演算子 ∘ は CSP の演算子として期待される一般的な性質を満たした上で，シーケンス図の合成を実現する．CSP の演算子としての性質としては，任意のプロセスに対する適用結果が実現可能となることや，詳細化関係を保存することが挙げられる．演算子がこれらの性質を満たすことで，他の CSP ツールとの連携や部分的な詳細化の検証が可能となる．

シーケンス図の合成を実現するための性質について述べる．

コンポーネントは複数のふるまいを持つが，一般的なシステムでは，同じ状態から同じイベントが発生した場合にはイベント発生後も同じ状態へと遷移する．CSP のプロセスとしては，次の式が成り立つことが期待される．

$$(\alpha \to P) \circ (\alpha \to Q) =_{\mathcal{F}} (\alpha \to (P \circ Q)) \quad (1)$$

また，シーケンス図に現れるメッセージの内容は送信側で決定されるため，メッセージの内容に複数の可能性がある場合，送信側にとっては内部選択，受信側にとっては外部選択となると考えられる．

$$(a! \to P) \circ (b! \to Q) =_{\mathcal{F}} (a! \to P) \sqcap (b! \to Q) \quad (2)$$
$$(a? \to P) \circ (b? \to Q) =_{\mathcal{F}} (a? \to P) \,\Box\, (b? \to Q) \quad (3)$$

ここで，$a!$ は送信イベント，$a?$ は受信イベントを表す．データ送受信を伴わない場合，CSP では一般に送信と受信は区別されないが，本研究ではシーケンス図合成のために送信と受信を明に区別している．

これらの条件を満たすように，失敗集合とトレース集合によって $P \circ Q$ の意味を定義することを考える．

基本的には $P \circ Q$ は内部選択や外部選択と同様に P もしくは Q としてふるまうプロセスである．しかし，シーケンス図の合成では，P と Q で同じイベント列 s が発生した後に異なる受信イベント a?, b? を待つ場合，P 由来の a? と Q 由来の b? がともに失敗せずに受信可能となる．つまり，受信イベントに関しては，内部選択や外部選択よりも失敗集合が小さくなり，P と Q でともに失敗するイベントのみが失敗集合に含まれる．

イベント列 s の実行に失敗の可能性があればイベント発生後に異なる状態となる場合があることを考慮し，詳細化関係が保存されるように注意しながら，$P \circ Q$ の意味を次のように定義する．

$$traces(P \circ Q) = traces(P) \cup traces(Q)$$

$$failures(P \circ Q) = \{(s, X) |$$
$$(s, X) \in failures(P) \cup failures(Q),$$
$$g(s, P) \Rightarrow (s, X \cap \mathcal{A}_?) \in failures(P),$$
$$g(s, Q) \Rightarrow (s, X \cap \mathcal{A}_?) \in failures(Q)\}$$

ここで，$\mathcal{A}_?$ は受信イベントの集合を表す．また，$g(s, P)$ は P の実行で失敗しないトレース s を表し，次のように定義される．

$$g(\langle\rangle, P) = true$$
$$g(s\hat{}\langle\alpha\rangle, P) = g(s, P) \wedge (s, \{\alpha\}) \notin failures(P)$$

実際，次の命題が示すように $P \circ Q$ は失敗詳細化関係を保存する．

命題：$P \sqsubseteq_{\mathcal{F}} P', Q \sqsubseteq_{\mathcal{F}} Q' \Rightarrow P \circ Q \sqsubseteq_{\mathcal{F}} P' \circ Q'$

また，以下の補題を証明することにより，期待通り (1), (2), (3) の CSP 規則を導ける．

補題 1：$g(s, P) \Leftrightarrow g(\langle\alpha\rangle\hat{}s, \alpha \to P)$

補題 2：$\alpha \neq \beta \Rightarrow g(\langle\alpha\rangle\hat{}s, \beta \to P) = false$

さらに，送受信が混在するときは次のような CSP 規則が得られる．

$$(a! \to P) \circ (b? \to Q) =_{\mathcal{F}} (a! \to P) \triangleright (b? \to Q) \quad (4)$$

ここで，▷ は次のように定義される糖衣構文である．

$$P \triangleright Q = (P \,\Box\, Q) \sqcap Q$$

この規則については直観的にも一致しており，$P \circ Q$ はシーケンス図の合成に有効であると考えられる．

4. 変換ツール SD2CSP

演算子 ∘ による合成を用いてシーケンス図を CSP に変換するツールとして，SD2CSP を開発した．

SD2CSP は，シーケンス図の情報が記述された XMI 形式の

ファイルを読み込んで FDR 記述を出力する機能を持つ．XMI 形式は UML モデルをやりとりするために OMG 標準として定められた XML フォーマットである．SD2CSP の出力として得られた記述を FDR で読み込むことで，シーケンス図の正しさを検証できる．

変換規則を以下に示す．

まずライフラインのヘッドに記述された型名からコンポーネント名集合を得る．例えば，クラス User のライフラインとクラス System のライフラインが存在する場合，次のような FDR 記述を出力する．

```
datatype CompoName_ = User | System
```

次に，シーケンス図中に存在する状態不変式から状態名集合を得る．状態不変式が記述されていない場合，シーケンス図の最初と最後は「default_」という特殊な状態であるとみなし，これを状態名の集合に加える．例えば，シーケンス図中に loggedin という状態不変式と haveCart という状態不変式が存在する場合，次のような FDR 記述を出力する．

```
datatype StateName_ =
  default_ | loggedin | haveCart
```

各コンポーネントのふるまいは全て COMPO_というプロセスとして定義する．COMPO_は引数としてコンポーネント名と状態名をドット (.) で結んだ値をとる．状態名のつけられた状態から次の状態名のつけられた状態までのメッセージの送受信を記述する．状態名のつけられた状態とは，シーケンス図中で状態不変式の置かれた位置，各シーケンス図の先頭・末尾，活性区間の切れ目のいずれかである．

COMPO_プロセスの内容は以下のアルゴリズムで出力する．

（1） 全てのライフラインについて，状態名のつけられた状態で分割する．

（2） 同じコンポーネントのライフラインについて，同じ状態から始まり同一のメッセージ送受信が存在するものをまとめ，ツリーを作る．

（3） ツリーの各ノードを CSP のプレフィックスとして出力する．ツリーが分岐する箇所では，分岐の次のメッセージ送受信が全てメッセージ送信の場合内部選択（規則 2）を，分岐の次のメッセージ送受信が全てメッセージ受信の場合外部選択（規則 3）を出力する．

これは，シーケンス図の各ライフラインを状態不変式の位置で分割して CSP に変換し，同じコンポーネントの同じ状態からの CSP を ○ 演算子で合成したものである．ツリーが分岐する箇所でメッセージ送信とメッセージ受信が混在する場合も規則 4 に従って出力すべきであるが，現時点では未実装となっている．

各メッセージは，メッセージ名・送信元・送信先の組み合わせに変換する．シーケンス図中のメッセージを表す call_チャネルを用意し，メッセージ名・送信元・送信先を順にドット (.) で結ぶ．これにより，同名の異なるメッセージが同期することを防ぐ．例えば，User から System へ getItems メッセージが送信される場合，このメッセージは次のように変換される．

```
call_.getItems.User.System
```

```
COMPO_(System.default_) =
  (call_.request1.User.System ->
    (call_.ok.System.User ->
    COMPO_(System.default_)
    |~|
    call_.ng.System.User ->
    COMPO_(System.default_))
  []
  call_.request2.User.System ->
    (call_.ok.System.User ->
    COMPO_(System.default_)
    |~|
    call_.ng.System.User ->
    COMPO_(System.default_)))
```

図 3　生成される FDR 記述の例

以上により，例えば図 3 のような FDR 記述が得られる．

システム全体のふるまいを検証するために，全てのコンポーネントを並行合成したプロセス SYSTEM_ を生成する．初期状態のコンポーネントは全て default 状態にあると仮定する．常に次の FDR 記述を出力する．

```
SYSTEM_ = || cn_:CompoName_ @
  [interface_(cn_)] COMPO_(cn_.default_)
```

この記述は，集合に含まれる要素をもとにプロセスを並行合成する糖衣構文を用いている．STSTEM_プロセスは，CompoName_に含まれる各コンポーネントを，他のコンポーネントと共有するメッセージで同期して並行合成したプロセスである．

interface_ 関数は特定のコンポーネントの入力もしくは出力となるメッセージの集合を返す関数であり，常に次のように定義される．

```
interface_(cn_) =
  union({call_.c_.cn_.to_ |
      c_ <- CallName_, to_ <- CompoName_},
    {call_.c_.from_.cn_ |
      c_ <- CallName_, from_ <- CompoName_})
```

ここで，x <- S は $x \in S$ の FDR 表記である．引数として渡されたコンポーネントが送信元か送信先に入っており，メッセージ名が CallName_に含まれ，送信元・送信先が CompoName_に含まれる集合を返している．

以上により，FDR で検証可能な記述が得られる．

5. ツールによる検証例

SD2CSP を使用した検証の例として，ショッピングサイトの例を考える．このショッピングサイトには，商品一覧の表示・商品のカートへの追加・購入・お勧めレシピの表示といった機能が存在する．また，これらの機能を使用するためにはログインとログアウトが必要である．

最初の段階として，全ての機能を単一の System コンポーネ

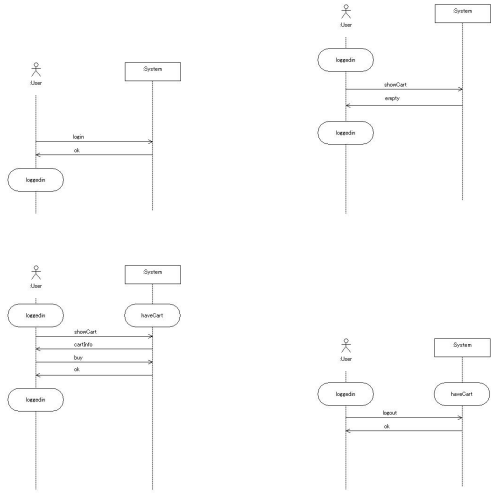

図 4　詳細化前のシーケンス図

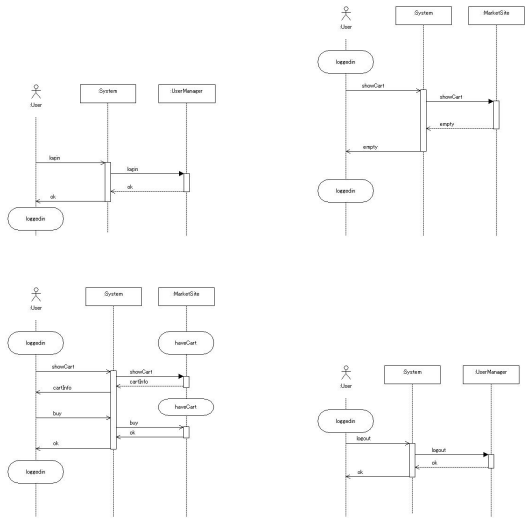

図 5　詳細化後のシーケンス図

ントで実現したモデルを考える．ユースケースごとに，ユーザーとシステムとのメッセージ送受信を記述する．一般的な開発プロセスではユースケース記述として記述される内容を，ここではツールによる検証のためシーケンス図で記述する．ユーザーとシステムとの間のメッセージ送受信の他，ユースケースごとの事前条件・事後条件を記述する．ここでは，ログイン済みかどうかをユーザーの状態として記述し，該当ユーザーのカートが空かどうかをシステムの状態として記述する．

シーケンス図は，異常系のフローも記述し，事前条件が異なる場合の挙動もそれぞれ記述する．この例では 14 種類のシーケンス図を記述した．シーケンス図の例を 図 4 に示す．これらのシーケンス図から，FDR 記述が得られる．

次に，機能ごとにコンポーネントを分割する．ここでは，ショッピングサイトの機能を「ユーザーの管理」「商品の販売」「レシピの提供」の 3 種類に分類し，機能ごとのコンポーネントに分割する．機能ごとに分割したコンポーネントに処理を委譲するようにシーケンス図を詳細化する．シーケンス図の例を 図 5 に示す．これらのシーケンス図からも FDR 記述が得られる．

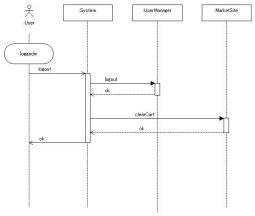

図 6　修正したシーケンス図

この詳細化が正しいことを FDR で検証する．詳細化前のシーケンス図から得られるシステムの FDR 記述と詳細化後のシーケンス図から得られるシステムの FDR 記述をあわせ，詳細化前後で共通するユーザーとシステム間のメッセージ送受信が保存されていることを検証する．実際に得られた FDR 記述を FDR で検証することで，エラーを発見できる．FDR のデバッガで問題点を確認すると，一度ログアウトして再度ログインした時に問題が発生することがわかる．showCart メッセージに対する返信として，詳細化前では empty メッセージが返されていた状況で，詳細化後では cartInfo メッセージを返そうとする．

シーケンス図を確認すると，カートが存在する時のログアウト処理の内容が詳細化前後で一致していないことがわかる．詳細化前では，カートが存在する状態でログアウトメッセージが送られると，事後条件ではカートはクリアされている．詳細化後では，カートの有無は System コンポーネントではなく MarketSite コンポーネントで管理するように変更されているが，ログアウト時には UserManager のみに処理が委譲され，MarketSite の情報が更新されない．もし詳細化前の挙動が正しいのであれば，ログアウト時にも MarketSite の状態を更新するように詳細化後のシーケンス図を直すことで検証が成功する．図 6 は図 5 の右下のシーケンス図を修正したものである．逆に，詳細化後のシーケンス図が正しい場合には詳細化前の仕様を直すことでも検証が成功する．

このように，SD2CSP および FDR を用いて段階的詳細化を検証することで，詳細化前後での仕様の違いを早期に発見できる．もし詳細化に間違いがあった場合，検出された箇所を修正することで，品質を向上し手戻り工数を削減することができる．詳細化後のシーケンス図が正しい場合には，詳細化前の仕様を修正することで，仕様と実装のずれをなくすことができる．

6. 関 連 研 究

シナリオベースのモデルから状態遷移モデルを合成する手法としては，状態遷移を決定的に保ったまま状態数を削減する手法，Live Sequence Chart と呼ばれる図を用いてシナリオの実行条件を明確化する手法，シナリオ間の実行順序を明示的に入力する手法などが提案されている．

状態遷移を決定的に保ったまま状態数を削減する手法として，A. W. Biermann は計算の実行ステップをもとにプログラムを自動合成する手法を提案している [5]．この手法では，実行する

命令と，各ステップで成り立つ条件とをもとに状態遷移モデルを構築する．このとき，同じ命令を実行する箇所を同じ状態の候補とし，最小の状態数でプログラムが実現可能であるように状態をまとめる．ただし，同じ状態から同じ条件の遷移が複数あって異なる状態に遷移する場合，決定的なプログラムとしては実現できなくなってしまうため，そのような非決定性が起こる場合は状態を分離し，異なる状態として扱うようにする．

これと同様の考え方をシーケンス図に適用した手法として，E. Mäkinen は MAS というツールを提案している [6]．この手法では，送信イベントと受信イベントのペアを考え，送信イベントを状態に，受信イベントを遷移の条件に対応させる．Biermann の手法と同様に決定性を保ったまま最小の状態数となるように状態遷移を構成するが，期待と異なる状態遷移が生成された場合には反例を入力することで状態遷移を修正できる．

Biermann や Mäkinen の手法により，入力したシナリオを実行可能なシンプルな状態遷移モデルを得ることができる．しかし，これらの手法では，決定的なモデルとなるように状態遷移を構築するため，曖昧性の残ったシーケンス図に対しては適用できない．本論文の提案手法では，複数の送信イベントを内部選択のように扱うため，コンポーネント内部の詳細な動作を決定していない段階でも合成と検証が可能である．

Live Sequence Chart を用いる手法は D. Harel らが提案している [7]．Live Sequence Chart はシーケンス図を拡張したもので，条件に一致する場合に必ず特定のシナリオが発生することを記述できる．このような記法を用いることで，シナリオの意味を明確化し，その条件をみたすような状態遷移を導出することが可能となる．

しかし，実際のソフトウェア開発においては，シナリオの条件を正確に定義することは簡単ではない．同じイベントを受け取った場合でも内部的な状態の違いにより異なるふるまいが発生する場合があるが，その前提をすべて Live Sequence Chart の条件部に記述する必要がある．本論文の提案手法では，条件を明示しないシーケンス図を元に状態遷移モデルを構築できる．

シナリオ間の実行順序を明示的に入力する手法としては，複数のメッセージシーケンスチャートおよびそれらの間の順序関係を記述した HMSC というモデルを状態遷移モデルに変換して検証する手法が R. Alur らにより提案されている [8]．また，この手法に基づき，モデル検査器 LTSA 上でメッセージシーケンスチャートを検証するための Message Sequence Chart plugin（LTSA-MSC）というツールが開発されている [9]．

LTSA-MSC では，HMSC で複数の遷移を記述することにより，不確定性を持った仕様を記述できる．しかし，各シーケンス図内のふるまいは確定的であるため，不確定性がある場所ごとにシーケンス図を分割し，注意深く HMSC モデルを記述する必要がある．本論文の提案手法では，送信イベントと受信イベントの扱いを変えることで，シーケンス図自体に不確定性をもたせることが可能となっている．

7. おわりに

本手法により，各コンポーネントのふるまいをプロセス代数 CSP による形式的記述に変換し，システムの整合性や詳細化の正しさを形式的に検証することが可能となった．

本手法では，複数のシーケンス図を合成する手法として新たな演算子 ∘ を導入した．この演算子はシーケンス図の合成に適した性質を持ち，また，詳細化関係を保存する．

本手法を実装したツールである SD2CSP を用いて段階的詳細化を検証すれば，詳細化前後でのふるまいの不一致や仕様とのずれを早期に発見できる．もし詳細化に間違いがあった場合，検出された箇所を修正することで，品質を向上し手戻り工数を削減できる．実際にショッピングサイトの例を適用して効果を確認した．

今後の課題としては，シーケンス図合成演算子 ∘ の性質をさらに明らかにするとともに，合成アルゴリズムの改善やユーザのシーケンス図記述を省力化する仕組みの提案などが考えられる．合成アルゴリズムの改善としては，状態不変式が記述されていない場合でも，途中の状態からのシーケンス図を適切に合成できるようなアルゴリズムを考えたい．記述の省力化としては，異常系などで記述が省略されている場合に，似た箇所の記述をもとにシナリオを自動的に提案する機能などを導入することでより簡単に本手法が適用できるようになると考えられる．

謝辞

本研究の機会を与えて頂いる国立情報学研究所教授本位田真一先生と北陸先端科学技術大学院大学学教授落水浩一郎先生に深く感謝いたします．なお，本研究は一部科研費（20500023）の助成を受けたものです．

文　献

[1] C. A. R. Hoare: “Communicating Sequential Processes”, Prentice Hall (1985).

[2] A. W. Roscoe: “The Theory and Practice of Concurrency”, Prentice Hall (1998).

[3] Formal Systems (Europe) Limited: “FDR2”, http://www.fsel.com/software.html.

[4] P. Welch: “Communicating sequential processes for java (jcsp)”, http://www.cs.kent.ac.uk/projects/ofa/jcsp/.

[5] A. W. Biermann and R. Krishnaswamy: “Constructing programs from example computations”, IEEE Transactions on Software Engineering (1976).

[6] E. Mäkinen and T. Systä: “Mas — an interactive synthesizer to support behavioral modelling in uml”, Proceedings of the 23rd International Conference on Software Engineering (2001).

[7] D. Harel and H. Kugler: “Synthesizing state-based object systems from lsc specifications”, International Journal of Foundations of Computer Science (2002).

[8] R. Alur and M. Yannakakis: “Model checking of message sequence charts”, Proceedings of the 10th International Conference on Concurrency Theory (1999).

[9] S. Uchitel, R. Chatley, J. Kramer and J. Magee: “Ltsa-msc: Tool support for behaviour model elaboration using implied scenarios”, Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (2003).

[10] H. Liang, J. Dingel and Z. Diskin: “A comparative survey of scenario-based to state-based model synthesis approaches”, Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools (2006).