博 士 論 文

プロセス代数に基づく非決定的なシナリオ合成による シーケンス図の検証

海津 智宏

主指導教員 鈴木 正人

北陸先端科学技術大学院大学

情報科学研究科

平成 26 年 6 月

目 次

1　はじめに

近年，ソフトウェア開発ではより短い納期で多品種化・マルチプラットフォーム化への対応が求められるようになっている．そのような要求にこたえ効率的にシステムを開発するために，コンポーネントを利用したソフトウェア開発が広まっている．

ソフトウェアコンポーネントの構造やふるまいを定義する際には，OMG によって標準化されたモデリング言語である UML [26] がしばしば利用される．ソフトウェアの上流設計では，コンポーネントの相互作用を理解し検証するために，UML のうち特にシーケンス図が利用されることが多い．

しかし，シーケンス図は特定の条件下でのシナリオを記述するものであり，システムのふるまいを網羅的に定義するものではない．そのため，シーケンス図で記述された設計を元に，各コンポーネントが独立に動作する際にシステム全体が意図しない状態にならないことを検証することは難しい．そのため，設計の不整合や不完全な詳細化などのシーケンス図の誤りを発見するのは人手での確認にたよっている場合が多い．そのような誤りが開発の上流工程で発見されず下流工程で発見された場合，修正には多くの時間とコストが必要となる．

設計の不整合や不完全な詳細化などの誤りを発見するためには，シーケンス図で記述されたシナリオからシステムのふるまいを合成する必要がある．シナリオベースのモデルから状態モデルを合成する手法にはいくつかの先行研究が存在するが，シーケンス図の仕様が複雑かつ柔軟であること，設計の上流工程で主に利用されるために詳細なふるまいが確定していない場合が多いことが問題となり，実用は難しかった．例えば「ユーザー名とパスワードが正しければログイン成功メッセージを返し，ユーザー名とパスワードが不正であればログイン失敗メッセージを返す」という設計において，パスワードの保存方法やチェック方法がまだ設計されていなければ，どちらのメッセージを返すべきかを決定的に記述できない．このような設計に対しては,「ログイン成功メッセージもしくはログイン失敗メッセージを非決定的に返す」のように非決定性を残したまま検証できることが重要となる．従来手法によるシーケンス図の検証では，このような非決定性を扱うことは難しかった．

本論文では，形式的な意味論を定義したシーケンス図のサブセットを定義し，そのシーケンス図で記述された設計の正しさを検証する手法を提案する．本手法を用いることで，開発者は形式的な記述により仕様を明確にすることができ，検証ツールにより早期に誤りを発見することが可能となる．

本手法では，シーケンス図を検証するために，複数のシーケンス図を合成してシステムのふるまいを導出し，プロセス代数 CSP (Communicating Sequential Processes) [17, 27] で記述されたプロセスを生成する．この合成手法は次のように実現される．まず，シーケンス図から各コンポーネントのメッセージ送受信と状態変化の情報を抽出する．この情報は CSP のプロセスとして形式的に記述することができる．次

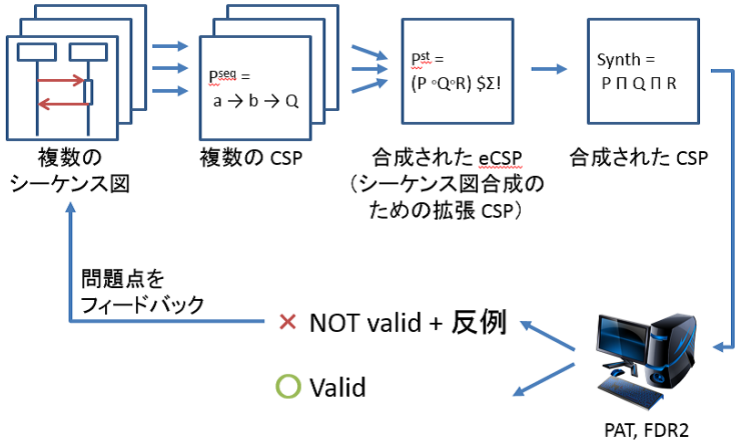


図 1: 提案手法の概要

に，同一のコンポーネントの同一の状態を起点とした CSP のプロセスを集め，提案するシーケンス図合成演算子を用いて合成する．最後に，合成されたプロセスに対してシーケンス図合成演算子の展開規則を適用し，通常の CSP プロセスへと変換する．シーケンス図合成演算子を含まない CSP プロセスは PAT [32] や FDR2 [10] などの既存のモデル検査器により検証が可能である．また，モデル検査器が出力する反例を逆変換して元のシーケンス図に戻すことによって，問題が発見された場合に効率的に原因を調査できる．図 1 は本手法の概要を図示したものである．

既存の関連研究と比較した本手法の利点は，非決定的な設計を検証できることである．シーケンス図は開発の上流工程で利用されるため，シーケンス図で記述された設計は非決定性を含んでいる場合が多い．本手法はそのような抽象的なシーケンス図にも適用できる．

本論文は以下のように構成される．まず，第 2 章では UML で定義されたシーケンス図について説明し，本論文で扱うシーケンス図のサブセットを定義する．次に，第 3 章では，プロセス代数 CSP の概要について説明する．第 4 章では，シーケンス図を合成して検証するための新たな CSP の演算子を定義し，その性質と検証手法について述べる．第 5 章では，提案手法をもとに開発したツールである SDVerifier について説明し，第 6 章で SDVerifier を用いて行ったケーススタディの結果を示す．最後に，第 7 章と第 8 章では，関連研究とこの論文の結論について述べる．

2　シーケンス図

2.1　シーケンス図の概要

シーケンス図は UML で定義された図の一種であり，オブジェクトやコンポーネントを表すライフライン間のメッセージ送受信による相互作用を表現する．図 2 はシーケンス図の例である．UML で定義されたシーケンス図の主な構成要素を以下に示す．

- ライフライン：相互作用の個々の参加者を表す．ライフラインは「ヘッド」と呼ばれる長方形とその下に続く縦の直線で表現される．ヘッドの中にはライフラインを特定するための情報が記述される．ソフトウェア部品が多重度をもつ場合があるが，複数の実体が存在する場合でも 1 つのライフラインは 1 つの実体を表す．
- メッセージ : ライフライン間の特定のコミュニケーションを表す．非同期メッセージは先端が開いた矢印，同期メッセージは先端が塗りつぶされた矢印，返信メッセージは破線の矢印で表現される．
- 活性区間 : ライフライン内の 1 つのふるまいやアクションを表す．ライフライン上の長方形で表現される．
- 状態不変式 : 実行時の制約として，属性・変数の値や状態を指定する．波括弧で囲んだテキスト，もしくは角丸の長方形で表現される．
- オブジェクト破棄：オブジェクトの破棄を示す．ライフライン終端の「×」記号で表現される．
- 結合フラグメント : 相互作用中の式を定義する．結合フラグメントにより，複数のシナリオをまとめて簡潔に表現することが可能となる．結合フラグメントは長方形で表現され，左上の区画にオペレータが記述される．オペレータの種類としては「alt/else」「opt」「break」「par」「seq」「strict」「neg」「critical」「consider/ignore」「assert」「loop」が定義されている．例えば，図 2 では alt オペレータが使われており，[*rooms* > *reservations*] が成り立つ場合には上の区画が，[*rooms* = *reservations*] が成り立つ場合には下の区画が実行される．つまり，この図は 2 つの異なる条件下のシナリオを 1 枚の図にまとめて記述したものである．

この他，シーケンス図全体を囲む枠である「フレーム」，他の相互作用を参照する「相互作用使用」，複数の結合フラグメントを接続するための「継続」，並行実行を表す「並置領域」，時間間隔やイベントの実行時間の制約を表す「時間制約」と「持続時間制約」，送信イベントや受信イベントが記述されないメッセージであ

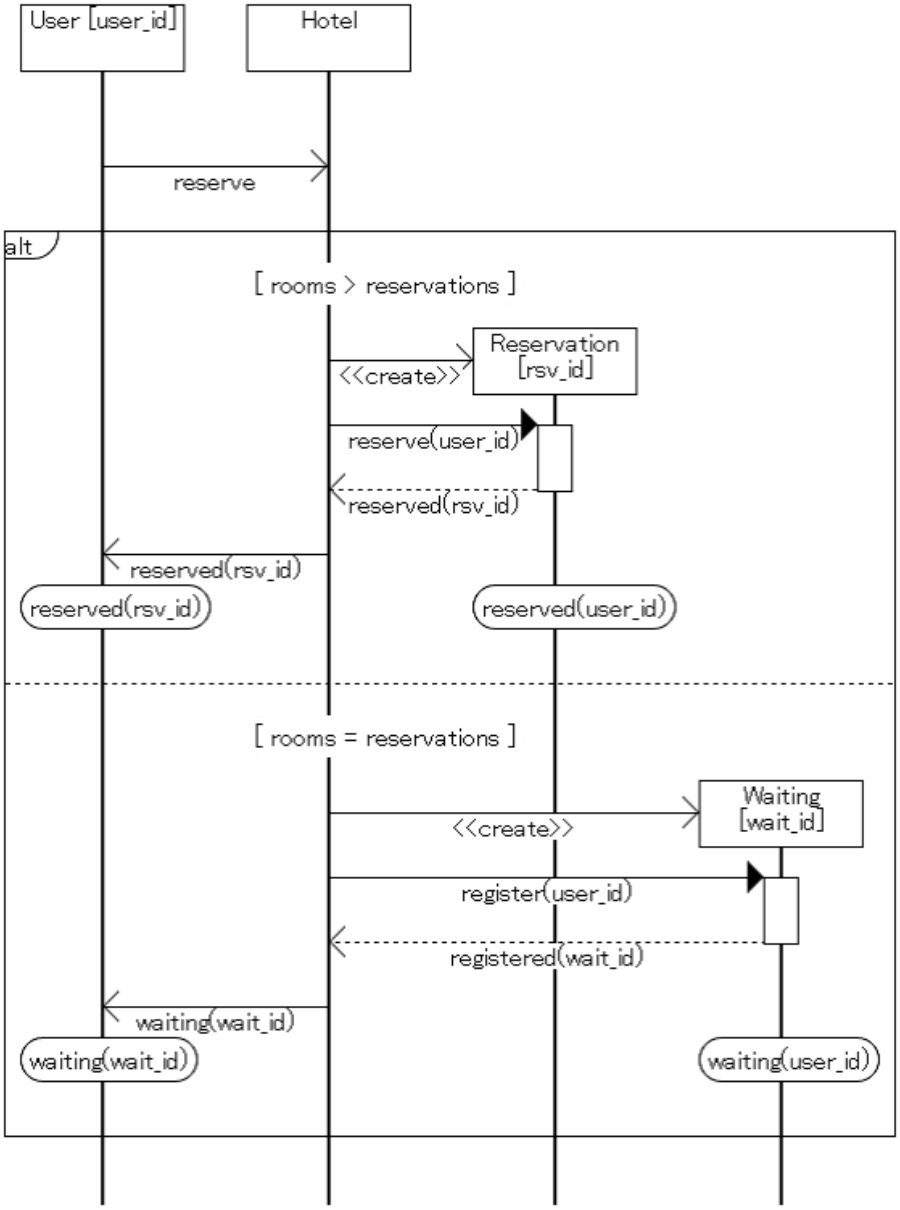

User [user_id]
Hotel
reserve
alt
[rooms > reservations]
Reservation
[rsv_id]
<<create>>
reserve(user_id)
reserved(rsv_id)
reserved(rsv_id)
reserved(rsv_id)
reserved(user_id)
[rooms = reservations]
Waiting
[wait_id]
<<create>>
register(user_id)
registered(wait_id)
waiting(wait_id)
waiting(wait_id)
waiting(user_id)

図 2: シーケンス図の例

る「拾得メッセージ」と「消失メッセージ」，2 点間の順序を指定する「汎用順序」が UML で定義されている．本論文ではこれらは利用しない．

2.2 設計検証用シーケンス図

シーケンス図は記述力が高く柔軟である反面，自動的な検証が困難であるという問題がある．本研究では，シーケンス図を自動的に検証するために，設計検証用にシーケンス図のサブセットを定義する．このサブセットに対し明確な意味論を与えることで，自動的な検証が可能となる．シーケンス図のサブセットを定義するにあたっては，現場で頻繁に使われる構成要素を採用した．図 3 は図 2 と同様の設計を設計検証用シーケンス図で記述した例である．

設計検証用シーケンス図の構成要素を以下に示す．

- ライフライン：相互作用の個々の参加者を表す．多重度が 1 の場合，ヘッドにはそのオブジェクト名を記述する．多重度が 1 より大きい場合，ヘッドにはクラス名および角括弧で囲んだ ID を記述する．この ID はメッセージおよび状態不変式のパラメータとして参照される．

- メッセージ：ライフライン間の特定のコミュニケーションを表す．矢印およびメッセージ名を記述する．

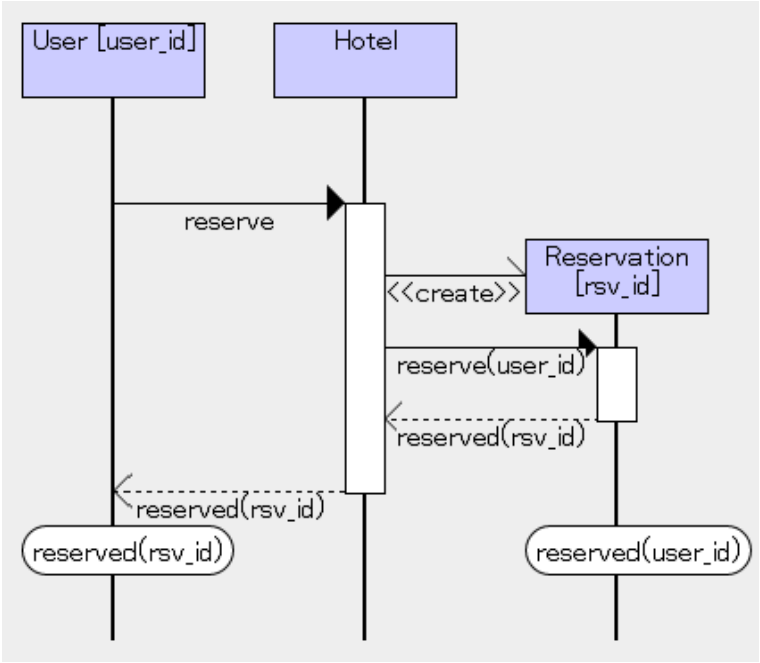
  引数を記述したい場合，パラメータとしてメッセージ名のあとに括弧をつけて値を記述する．同じオブジェクト間で送受信される同名のメッセージでは常に同じ数のパラメータを記述する．括弧内の位置が同じものは同一の引数を表すものとみなされる．値としてはライフラインの ID，メッセージで受信した値，もしくは以前の状態不変式のパラメータとなっていた値を指定できる．

  本論文では主に非同期メッセージを扱うが，同期メッセージを受信したオブジェクトは送信オブジェクトに返信メッセージを返すこと，送信オブジェクトは返信メッセージの受信まで他のふるまいをしないことの 2 点の制約が満たされれば，同期メッセージも非同期メッセージと同様に扱うことができる．

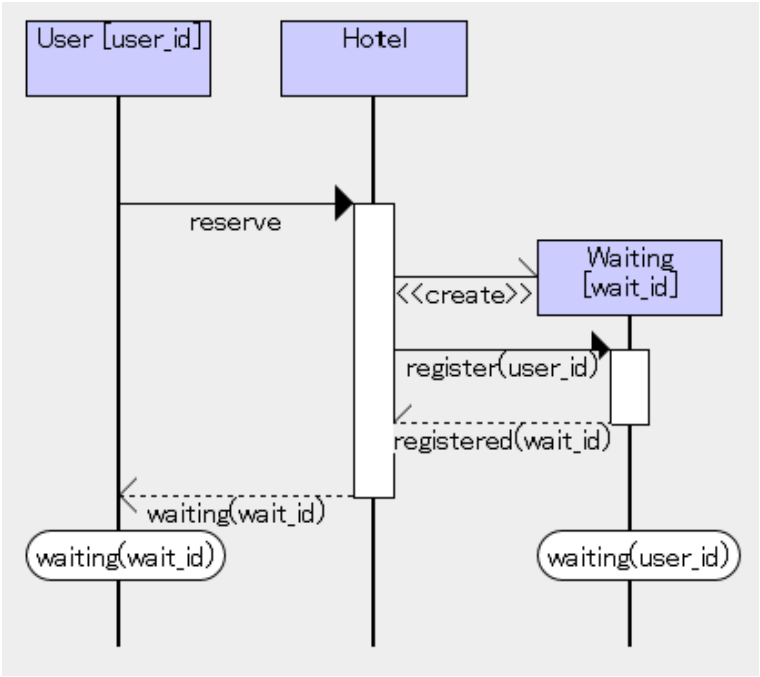
  メッセージ名の前に ⟨⟨*create*⟩⟩ と記述されたメッセージはオブジェクト生成メッセージを表し，そのメッセージによって新しいオブジェクトが生成されることを示す．生成されるオブジェクトはライフラインのヘッドをオブジェクト生成メッセージの先に置く．

- 活性区間：ライフライン内の 1 つのふるまいやアクションを表す．ライフライン上の長方形で表現される．

- 状態不変式：実行時の制約として，オブジェクトの状態を指定する．状態不変式は角丸の長方形で記述し，その中に状態名を記述する．属性・変数の値



(a)



(b)

図 3: 設計検証用シーケンス図で記述した図 2

を記述したい場合，パラメータとして状態名のあとに括弧をつけて値を記述する．同じ状態名の状態不変式では常に同じ数のパラメータを記述する．括弧内の位置が同じものは同じ属性・変数を表すものとみなされる．値としてはライフラインの ID，メッセージで受信した値，もしくは以前の状態不変式のパラメータとなっていた値を指定できる．

- オブジェクト破棄：オブジェクトの破棄を示す．ライフライン終端の「×」記号で表現される．

図 4 は簡単な設計検証用シーケンス図の例である．以降ではこの例を用いて提案手法を説明する．

通常，シーケンス図は特定のシナリオのみを表すものであり，シーケンス図に書かれたものと異なるシナリオが許可されるかどうかは不定である．複数のシナリオを合成して状態遷移モデルを得る場合,「出力される状態遷移モデルが入力のシナリオをすべて実行できる」ことが制約となるが，そのような状態遷移モデルは一意には定まらない．例えば,「すべてのオブジェクトがすべての状態ですべてのメッセージを送受信できる」という状態遷移モデルを考えれば，これは任意のシナリオを実行できるため，どのようなシーケンス図の入力に対しても正当な合成結果となってしまう．これは，シーケンス図の検証という目的にはそぐわない．

そのため，設計検証用シーケンス図では，各シーケンス図がどのように状態モデルと対応するかも定義する．例えば，図 4b における *User* オブジェクトは，通常のシーケンス図としては「*addToCart* を送信し，*ok* を受信し，*buy* を送信し，*ok* を受信する」という 1 シナリオのみしか表さないが，設計検証用シーケンス図としては「*addToCart* もしくは *buy* を送信し，*ok* を受信することを任意回数繰り返す」という意味を持つ．そのように 1 つのシーケンス図が複数のシナリオを表すことで，実用的な状態遷移モデルを合成して検証することが可能となる．

設計検証用シーケンス図におけるふるまいは次のように定義される．

- 各オブジェクトは，ライフラインにそって上から下の順にメッセージを送受信できる．
- オブジェクトが状態不変式に到達した場合，同じ状態名が記述された他の状態不変式の箇所から続きのメッセージを送受信できる．
- ライフラインの先頭および活性区間の完了直後で状態名の明記されていない状態を標準状態と呼ぶ．状態不変式が書かれている場合と同様に，オブジェクトが標準状態に到達した場合には他の標準状態の箇所から続きのメッセージを送受信できる．

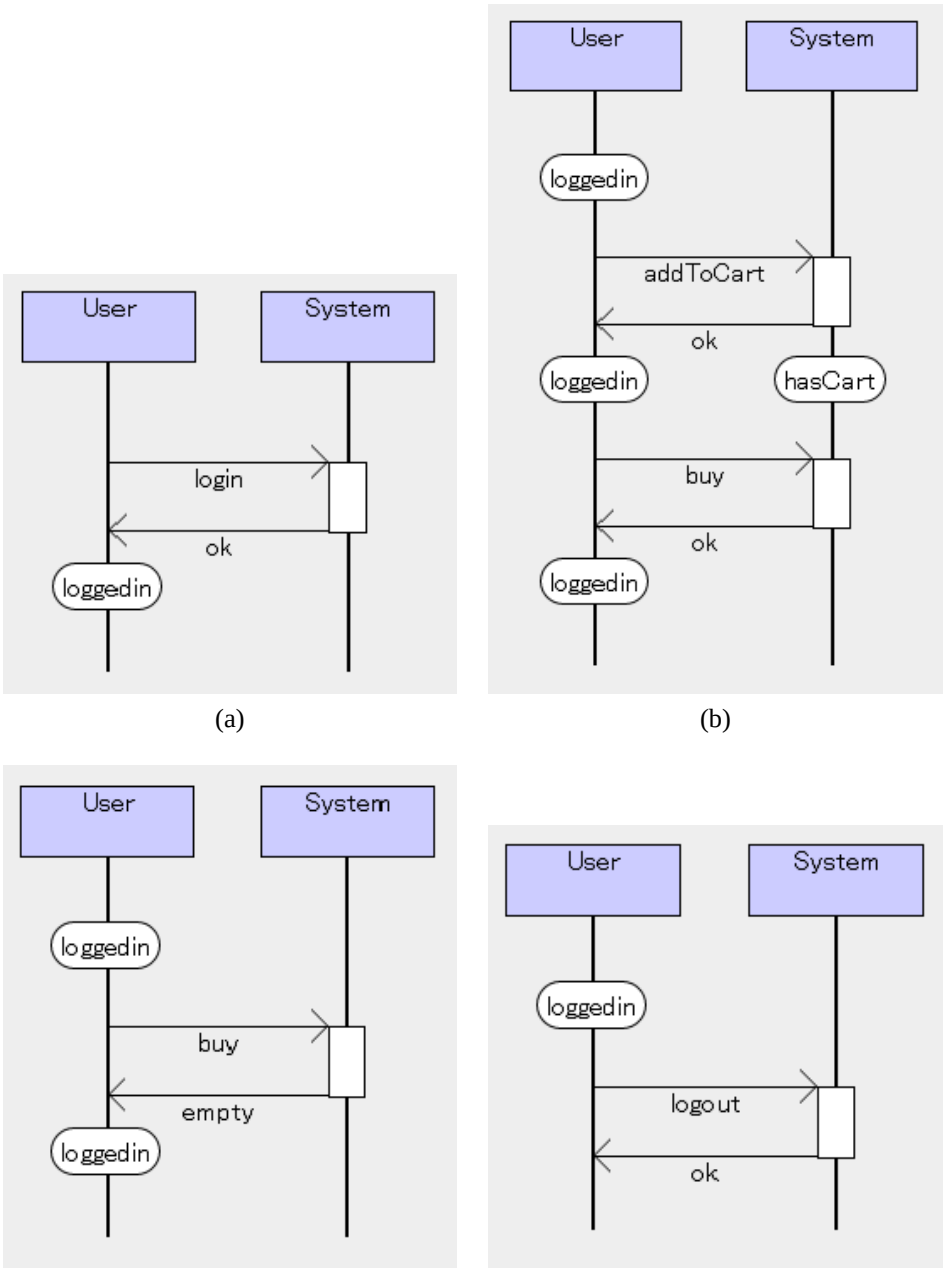


図 4: 簡単な設計検証用シーケンス図の例

設計検証用シーケンス図の形式的な定義を以下に示す.

$$
\begin{aligned}
Designs &= 2^{SDs} \\
SDs &= 2^{Transitions} \\
Transitions &= Objects \times States \times Messages \times States \\
Objects &= ClassNames \times Ids \\
Messages &= MessageNames \times Objects \times Objects \times 2^{Ids} \\
States &= StateNames \times 2^{Ids}
\end{aligned}
$$

ここで，$ClassNames$ はクラス名の集合，Ids はライフラインの ID の集合，$MessageNames$ はメッセージ名の集合，$StateNames$ は状態名の集合である．遷移 $(O, S1, M, S2) \in Transitions$ はオブジェクト O がメッセージ M を送信もしくは受信し，状態を $S1$ から $S2$ に変化させることを示す．メッセージ $(MN, OS, OR, P) \in Messages$ はメッセージ名が MN で，オブジェクト OS が送信しオブジェクト OR が受信するメッセージを表す．P はこのメッセージのパラメータである．オブジェクトはクラス名と ID のペアで表される．ライフラインに ID が記述されていない場合は，特別な ID (0) がそのオブジェクトの ID であると考えることができる．状態は状態名とパラメータの集合のペアで表される．

状態名は，シーケンス図上の状態不変式で明示されたもののほか，下記の特殊な状態を定義する．

- 標準状態 : ライフラインの先頭および活性区間の完了直後で，状態名の明記されていない状態を標準状態と呼ぶ.
- 準備状態 : ライフラインのオブジェクト破棄（×）以降およびオブジェクト生成メッセージの受信前の状態を準備状態と呼ぶ．この状態はオブジェクトが存在していないことを示す特別な状態である．
- 中間状態 : それ以外のメッセージ間の状態を中間状態と呼ぶ．それぞれの中間状態はすべて異なる状態として扱われる．

これらの状態を定義することで，すべてのメッセージ送受信におけるオブジェクトの状態変更を記述できる，設計検証用シーケンス図の定義では複数のメッセージの順序を明示的には記述できないが，遷移前後の中間状態をつなぐことで活性区間内のメッセージの順序が決定される．

例えば，図 4a の $User$ オブジェクトは，$login$ メッセージを送信して中間状態に遷移し，その中間状態から ok メッセージを受信して $loggedIn$ 状態に遷移するので，次の 2 つの遷移 T_1, T_2 で記述できる．

$$
\begin{aligned}
T_1 &= (User, d, login, i_1) \\
T_2 &= (User, i_1, ok, loggedin)
\end{aligned}
$$

ここで，d は標準状態を，i_1 は中間状態を表す．T_1 の遷移後の状態と T_2 の遷移前の状態がともに i_1 であることから，T_2 が T_1 の後に実行可能であることがわかる．

同様に，図 4 の 4 つのシーケンス図は次のように記述できる．

$$D = \{SD_a, SD_b, SD_c, SD_d\} \in Designs$$

$$\begin{aligned}
\{SD_a, SD_b, SD_c, SD_d\} &\subseteq SDs \\
SD_a &= \{T_1, T_2, T_3, T_4\} \\
SD_b &= \{T_5, T_6, T_7, T_8, T_9, T_{10}, T_{11}, T_{12}\} \\
SD_c &= \{T_{13}, T_{14}, T_{15}, T_{16}\} \\
SD_c &= \{T_{17}, T_{18}, T_{19}, T_{20}\}
\end{aligned}$$

$$\begin{aligned}
\{T_1, T_2, \ldots, T_{20}\} &\subseteq Transitions \\
T_1 &= (User, d, login, i_1) \\
T_2 &= (User, i_1, ok, loggedin) \\
T_3 &= (System, d, login, i_2) \\
T_4 &= (System, i_2, ok, d) \\
T_5 &= (User, loggedin, addToCart, i_3) \\
T_6 &= (User, i_3, ok, loggedin) \\
T_7 &= (User, loggedin, buy, i_4) \\
T_8 &= (User, i_4, ok, loggedin) \\
T_9 &= (System, d, addToCart, i_5) \\
T_{10} &= (System, i_5, ok, hasCart) \\
T_{11} &= (System, hasCart, buy, i_6) \\
T_{12} &= (System, i_6, ok, d) \\
T_{13} &= (User, loggedin, buy, i_7) \\
T_{14} &= (User, i_7, empty, loggedin) \\
T_{15} &= (System, d, buy, i_8) \\
T_{16} &= (System, i_8, empty, d) \\
T_{17} &= (User, loggedin, logout, i_9) \\
T_{18} &= (User, i_9, ok, d) \\
T_{19} &= (System, d, logout, i_{10}) \\
T_{20} &= (System, i_{10}, ok, d)
\end{aligned}$$

$$\{login, ok, addToCart, buy, empty, logout\} \subseteq Messages$$

$$
\begin{aligned}
login &= (\texttt{login}, User, System, \{\}) \\
ok &= (\texttt{ok}, System, User, \{\}) \\
addToCart &= (\texttt{addToCart}, User, System, \{\}) \\
buy &= (\texttt{buy}, User, System, \{\}) \\
empty &= (\texttt{empty}, System, User, \{\}) \\
logout &= (\texttt{logout}, User, System, \{\})
\end{aligned}
$$

$$
\begin{aligned}
\{User, System\} &\subseteq Objects \\
\{loggedin, hasCart, d, i_1, \ldots i_{10}\} &\subseteq States
\end{aligned}
$$

ここで，d は各オブジェクトの標準状態を，$i_1, \ldots i_{10}$ は中間状態を表す．

なお，状態名のスコープはそれぞれのオブジェクト内のみである．*User* オブジェクトの標準状態と *System* オブジェクトの標準状態は異なる状態として認識される．シーケンス図中の特定のオブジェクトの特定の状態を表す場合には，状態名だけではなくオブジェクト名と状態名のペアを用いる．

2.3 設計検証用シーケンス図の記述能力

設計検証用シーケンス図は UML 2 で導入された結合フラグメントには対応していない．しかし，状態不変式を利用することで分岐や繰り返しを表現できる．例えば，図 4 中で *loggedin* 状態は複数回出現している．これは，結合フラグメント (alt) を用いて表現した図 5 と同様の意味である．オブジェクトは，その状態から始まる遷移のいずれでも実行できる．また，元の状態に戻る一連のメッセージは，繰り返し実行できるため，結合フラグメントの loop と同様の記述力を持つ．シーケンス図は開発の初期段階で利用されるため，結合フラグメントを使用して分岐や繰り返しを記述するよりも，さまざまなシナリオを個々の図で定義し，それらを組み合わせることでシステム全体のふるまいを表現するほうが便利な場合が多い．

UML で定義された構成要素のうち設計検証用シーケンス図に含まれないものについて，設計検証用シーケンス図での対応方法は以下のようになる．

- alt/else 結合フラグメント：それぞれの条件におけるシナリオを別シーケンス図として記述する．
- opt 結合フラグメント：結合フラグメント内が実行される場合と実行されない場合を別シーケンス図として記述する．
- break 結合フラグメント：処理を中断する場合と中断しない場合を別シーケンス図として記述する．

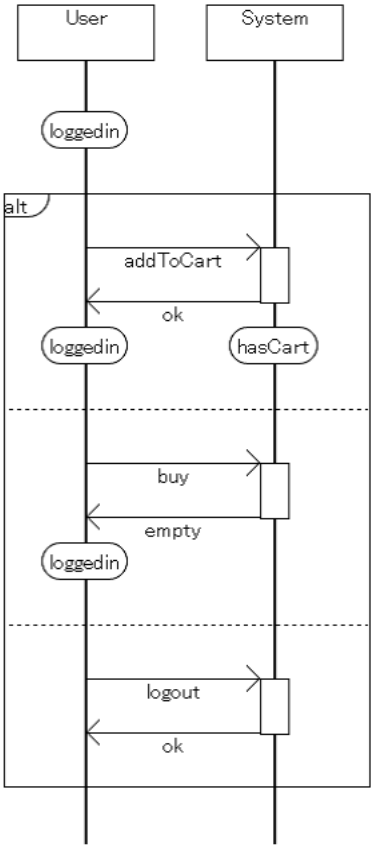


図 5: 結合フラグメントを用いて記述した *loggedin* 状態からの遷移

- par 結合フラグメント：各結合フラグメントの実行順序について，すべてのパターンのシーケンス図を記述する．
- seq 結合フラグメント：設計検証用シーケンス図では暗黙に各ライフラインの順序関係は保持される．標準状態への遷移を期待しない場合には，他の場所で使用されない状態を記述することでシーケンス図に記述されたとおりにメッセージが送受信されることを表せる．
- strict 結合フラグメント：設計検証用シーケンス図では各ライフラインは独立に動作することを前提とするため，ライフラインをまたがった順序関係は指定できない．
- neg 結合フラグメント：設計検証用シーケンス図では，どのシーケンス図にも記述されないシナリオは不正なシナリオである．
- critical 結合フラグメント：活性区間，もしくは他の場所で使用されない状態を使用した場合，他のメッセージは実行されない．
- consider/ignore 結合フラグメント：起こりうるメッセージをすべて記述する．
- assert 結合フラグメント：期待するメッセージ以外を含むシーケンス図が存在しなければ，そのメッセージの存在は必須となる．
- loop 結合フラグメント：結合フラグメントの開始箇所と終了箇所に同一の状態不変式を置く．
- フレーム：フレームはオブジェクトのふるまいを表すものではないので，検証においては必要ない．
- 相互作用使用：参照するシーケンス図の最初と最後に状態を付加しておけば，参照先の最初の状態へ遷移するシーケンス図と参照先の最後の状態から始まるシーケンス図としてシナリオを記述する．
- 継続：それぞれの条件におけるシナリオを別シーケンス図として記述する．
- 並置領域：複数のメッセージの実行順序について，すべてのパターンのシーケンス図を記述する．
- 時間制約，持続時間制約：提案手法では時間制約は検証しないため，記述できない．
- 拾得メッセージ，消失メッセージ：実際にメッセージを送信・受信する可能性があるすべてのオブジェクトに関してシーケンス図を記述する．

- 汎用順序：設計検証用シーケンス図では各ライフラインは独立に動作することを前提とするため，ライフラインをまたがった順序関係は指定できない．

以上のように，時間制約およびライフラインをまたがる順序関係を除き，多くの要素は設計検証用シーケンス図で書き換えが可能である．

3 プロセス代数 CSP

3.1 CSP の概要

プロセス代数は並行システムを記述・解析するための理論である．CSP はプロセス代数の一種であり，プロセッサの設計やセキュリティ・プロトコルの設計など，さまざまな場面に広く使われている [2, 28].

CSP で記述された並行プロセスは，PAT[32] や FDR[10] などの解析ツールによってデッドロック・フリー，ライブロック・フリー，詳細化関係などを検証できる．

CSP では 10 種類以上の演算子が用いられるが，本論文ではシーケンス図の検証に必要となる演算子のみを定義する．本論文で用いる CSP は次のように定義される．

$$P \quad ::= \quad a \rightarrow P \mid P \,\square\, P \mid P \sqcap P \mid P_X \|_Y P \mid P \setminus X \mid PN$$

ここで，a はイベント名，X および Y はイベントの集合，PN はプロセス名である．プロセス名は $PN = P$ の形の式で定義される．

それぞれの演算子の意味は以下の通りである．[1]

- プレフィックス：$a \rightarrow P$ (PAT では `a -> P`) はイベント a を実行し，その後プロセス P のようにふるまうプロセスである．
- 外部選択：$P \,\square\, Q$ (PAT では `P [*] Q`) はプロセス P もしくは Q のようにふるまうプロセスである．P と Q のどちらが選択されるかは次に実行されるイベントによって定まる．そのため，この選択は他のプロセスや外部環境など，プロセスの外部からコントロールできる．
- 内部選択：$P \sqcap Q$ (PAT では `P <> Q`) はプロセス P もしくは Q のようにふるまうプロセスである．P もしくは Q はプロセス内部で非決定的に選択され，プロセスの外部からはコントロールできない．
- 並行合成：$P_X \|_Y Q$ (PAT では `P || Q`) はプロセス P と Q が並行に動作するプロセスを表す．プロセス P はイベント集合 $(X - Y)$ を，プロセス Q はイベント集合 $(Y - X)$ を，それぞれ独立に実行できる．プロセス P および Q はイベ

[1] 詳細な定義は付録 A を参照

$$
\begin{aligned}
P1 &= in \rightarrow sync \rightarrow sync \rightarrow P1 \\
P2 &= sync \rightarrow out \rightarrow sync \rightarrow P2 \\
SYS &= P1_{\{in,sync\}} ||_{\{sync,out\}} P2
\end{aligned}
$$

図 6: CSP の並行プロセスの例

ント集合 $(X \cap Y)$ によって同期される．つまり，$(X \cap Y)$ に含まれるイベントは P と Q で同時に実行されなければならない．PAT ではイベント集合 X，Y は明示的には記述されず，プロセス P と Q に含まれるイベントが暗黙的に利用される．

- 隠蔽: $P \setminus X$ (PAT では `P \ X`) はプロセス P のうち，X に含まれるイベントが隠蔽されたプロセスである．X に含まれるイベントはプロセス外部からは観測できない．

図 6 は CSP における並行プロセスの例である．$P1$ はイベント *in*，*sync*，*sync* を繰り返し実行するプロセス，$P2$ はイベント *sync*，*out*，*sync* を繰り返し実行するプロセス，*SYS* は $P1$ と $P2$ を並行合成したプロセスである．*SYS* で同期されるのはイベント *sync* のみであり，合成されたプロセスは次のようにふるまう．まず，$P1$ が *in* を独立して実行し，次に $P1$ と $P2$ が同期して *sync* を実行し，次に $P2$ が *out* を独立して実行し，次にまた $P1$ と $P2$ が同期して *sync* を実行し，そしてこのふるまいを繰り返す．したがって，*SYS* は次の逐次的なプロセス *SYS′* と同じようにふるまうといえる．

$$SYS' = in \rightarrow sync \rightarrow out \rightarrow sync \rightarrow SYS'$$

イベント名は「.」記号を用いて構造化できる．例えば,「$c.x_1.x_2$」は 1 つのイベント名である．ここで，c をチャネル名と呼ぶ．x_1 および x_2 はチャネルを通じて受け渡されるデータである．さらに，データの受け渡しのための記号として,「?」および「!」を定義する.「?」はデータの入力を,「!」はデータの出力を表す．例えば，以下のプロセス P はチャネル c_1 からデータ x を受信し，そのデータをチャネル c_2 に送信する．

$$P = c_1?x \rightarrow c_2!x \rightarrow P$$

このプロセスは，x の取りうる値が $\{a_1, a_2, a_3\}$ であれば，次のプロセスと等価である．

$$
\begin{aligned}
P &= c_1.a_1 \rightarrow c_2.a_1 \rightarrow P \\
&\Box\ c_1.a_2 \rightarrow c_2.a_2 \rightarrow P \\
&\Box\ c_1.a_3 \rightarrow c_2.a_3 \rightarrow P
\end{aligned}
$$

また，本論文では，データ部分に定数値を記述した場合や変数にすでに値が紐付けられている場合には，その値のみを送受信できるプロセスを表す．例えば，a_1 が変数名ではなく定数値であれば，次の等式が成り立つ．

$$c_1?a_1 \rightarrow c_2!a_2 \rightarrow P \quad = \quad c_1.a_1 \rightarrow c_2.a_2 \rightarrow P$$

PAT ツールにおいても「.」「?」「!」はここでの定義と同様に定義されており，これらを含むプロセスを検証することができる．

3.2 CSP の等価関係と詳細化関係

CSP の等価関係と詳細化関係に関してはいくつかのモデルが知られている．ここでは，そのうちトレースモデルと失敗モデルについて説明する．

トレースモデルでは，等価関係と詳細化関係はトレース集合 $traces(P)$ で定義される．トレース集合とは，対象のプロセスが実行できるイベント列の集合である．演算子 $\rightarrow$，$\sqcap$，$\Box$ に対し，トレース集合は次のように定義される．

$$\begin{aligned} traces(a \rightarrow P) &= \{\langle\rangle\} \cup \{\langle a\rangle\hat{}s | s \in traces(P)\} \\ traces(P \sqcap Q) &= traces(P) \cup traces(Q) \\ traces(P \Box Q) &= traces(P) \cup traces(Q) \end{aligned}$$

トレース等価およびトレース詳細化は次のように定義される．

$$\begin{aligned} P =_T Q &\Leftrightarrow traces(P) = traces(Q) \\ P \sqsubseteq_T Q &\Leftrightarrow traces(P) \supseteq traces(Q) \end{aligned}$$

ここで，$P \sqsubseteq_T Q$ はプロセス Q がプロセス P を詳細化したプロセスであることを表す．

失敗モデルでは，等価関係と詳細化関係はトレース集合 $traces(P)$ および失敗集合 $failures(P)$ で定義される．失敗集合はペア (s, X) の集合であり，プロセス P がイベント列 s を実行した後にイベント集合 X を実行できない可能性があることを表す．演算子 $\rightarrow$，$\sqcap$，$\Box$ に対し，失敗集合は次のように定義される．

$$\begin{aligned} &failures(a \rightarrow P) = \\ &\quad \{(\langle\rangle, X) | a \notin X\} \cup \{(\langle a\rangle\hat{}s, X) | (s, X) \in failures(P)\} \\ &failures(P \sqcap Q) = failures(P) \cup failures(Q) \\ &failures(P \Box Q) = \\ &\quad \{(\langle\rangle, X) | (\langle\rangle, X) \in failures(P) \cap failures(Q)\} \\ &\quad \cup \{(s, X) | s \neq \langle\rangle, (s, X) \in failures(P) \cup failures(Q)\} \end{aligned}$$

トレース集合と異なり，失敗集合では内部選択と外部選択の定義が異なる点が重要である．これにより，失敗モデルでは決定的なプロセスと非決定的なプロセスを区別することができる．

失敗等価および失敗詳細化は次のように定義される．

$$
\begin{aligned}
P =_F Q \quad &\Leftrightarrow \quad traces(P) = traces(Q) \\
&\wedge \quad failures(P) = failures(Q) \\
P \sqsubseteq_F Q \quad &\Leftrightarrow \quad traces(P) \supseteq traces(Q) \\
&\wedge \quad failures(P) \supseteq failures(Q)
\end{aligned}
$$

ここで，$P \sqsubseteq_F Q$ はプロセス Q がプロセス P を詳細化したプロセスであることを表す．

失敗モデルではライブロックが正しく扱えないという問題が知られている．しかし，設計検証用シーケンス図においては状態の変化は必ずメッセージの送受信を伴うため，シーケンス図から生成される CSP プロセスはライブロックを起こさない．そこで，本論文ではシーケンス図の検証のために失敗モデルを利用する．

3.3 他のモデルとの比較

CSP 以外の状態遷移モデルとして，Calculus of Communicating Systems (CCS) [25] や IO オートマトンなどを利用することも考えられる．実際に，本論文の提案手法と同じ考え方でシーケンス図を CCS や IO オートマトンへ変換して検証することは可能である．しかし，CSP を用いることで，本論文の提案手法をより自然な形で議論できる．

1 点目の特徴として，CSP では内部選択と外部選択が基本的な演算子として用意されている．これを用いることで，オブジェクトが内部的にどちらかを選択する場合は $P \sqcap Q$，外部環境によって実行できるものを選択する場合は $P \,\square\, Q$ と簡便に記述できる．これらの違いを区別して検証できることは提案手法の大きな特徴である．CCS や IO オートマトンでは，内部選択は内部遷移として $(\tau \rightarrow P) \,\square\, (\tau \rightarrow Q)$ のように記述する必要があり，直観的に内部選択と外部選択の違いを議論するには不向きである．

2 点目の特徴として，CSP では等価関係・詳細化関係がトレース集合及び失敗集合で定義されている．CCS では，等価関係・詳細化関係を双模倣性で定義する場合が多い．本論文の提案手法では，$a \rightarrow P$ と $a \rightarrow Q$ との合成結果として $a \rightarrow (P \,\square\, Q)$ を得るというような変換を行う．このとき，合成結果における a イベント実行後の状態は，$a \rightarrow P$ における a イベント実行後の状態と $a \rightarrow Q$ における a イベント実行後の状態両方と対応する．これは，状態の対応関係が 1:1 とならないため，双模倣性では扱いづらい．トレース集合および失敗集合を用いることで，オブジェクトのふるまいに着目して等価関係・詳細化関係を議論できる．

4　シーケンス図合成のための CSP の拡張

4.1　シーケンス図から CSP への変換

本研究では，設計検証用シーケンス図で記述されたオブジェクトのふるまいを CSP を用いて定義することで，シーケンス図から CSP への変換および検証を可能とする．シーケンス図から CSP への変換においては，まずシーケンス図で記述された遷移をそれぞれ CSP で表し，次に複数のシーケンス図から得られた CSP を合成することでオブジェクトのふるまいが得られる．

1 つのシーケンス図と CSP との対応は直観的に定義できる．ライフラインと状態のペアが CSP のプロセスと，メッセージが CSP のイベントと対応する．例えば，図 4 は次の CSP に変換できる．

$$
\begin{aligned}
\mathcal{P}_D^{seq}(T_1) &= Login \rightarrow \mathcal{P}_D^{st}(User, i_1) \\
\mathcal{P}_D^{seq}(T_2) &= OK \rightarrow \mathcal{P}_D^{st}(User, loggedin) \\
\mathcal{P}_D^{seq}(T_3) &= Login \rightarrow \mathcal{P}_D^{st}(System, i_2) \\
\mathcal{P}_D^{seq}(T_4) &= OK \rightarrow \mathcal{P}_D^{st}(System, d) \\
\mathcal{P}_D^{seq}(T_5) &= AddToCart \rightarrow \mathcal{P}_D^{st}(User, i_3) \\
\mathcal{P}_D^{seq}(T_6) &= OK \rightarrow \mathcal{P}_D^{st}(User, loggedin) \\
\mathcal{P}_D^{seq}(T_7) &= Buy \rightarrow \mathcal{P}_D^{st}(User, i_4) \\
\mathcal{P}_D^{seq}(T_8) &= OK \rightarrow \mathcal{P}_D^{st}(User, loggedin) \\
\mathcal{P}_D^{seq}(T_9) &= AddToCart \rightarrow \mathcal{P}_D^{st}(System, i_5) \\
\mathcal{P}_D^{seq}(T_{10}) &= OK \rightarrow \mathcal{P}_D^{st}(System, hasCart) \\
\mathcal{P}_D^{seq}(T_{11}) &= Buy \rightarrow \mathcal{P}_D^{st}(System, i_6) \\
\mathcal{P}_D^{seq}(T_{12}) &= OK \rightarrow \mathcal{P}_D^{st}(System, d) \\
\mathcal{P}_D^{seq}(T_{13}) &= Buy \rightarrow \mathcal{P}_D^{st}(User, i_7) \\
\mathcal{P}_D^{seq}(T_{14}) &= Empty \rightarrow \mathcal{P}_D^{st}(User, loggedin) \\
\mathcal{P}_D^{seq}(T_{15}) &= Buy \rightarrow \mathcal{P}_D^{st}(System, i_8) \\
\mathcal{P}_D^{seq}(T_{16}) &= Empty \rightarrow \mathcal{P}_D^{st}(System, d) \\
\mathcal{P}_D^{seq}(T_{17}) &= Logout \rightarrow \mathcal{P}_D^{st}(User, i_9) \\
\mathcal{P}_D^{seq}(T_{18}) &= OK \rightarrow \mathcal{P}_D^{st}(User, d) \\
\mathcal{P}_D^{seq}(T_{19}) &= Logout \rightarrow \mathcal{P}_D^{st}(System, i_{10}) \\
\mathcal{P}_D^{seq}(T_{20}) &= OK \rightarrow \mathcal{P}_D^{st}(System, d)
\end{aligned}
$$

$$
\begin{aligned}
Login &= call\_.login.User.System \\
OK &= call\_.ok.System.User \\
AddToCart &= call\_.addToCart.User.System \\
Buy &= call\_.buy.User.System \\
Empty &= call\_.empty.System.User \\
Logout &= call\_.logout.User.System
\end{aligned}
$$

ここで，$\mathcal{P}_D^{seq}(T)$ はシーケンス図の遷移 T と対応する CSP のプロセスである．また，$\mathcal{P}_D^{st}(O,S)$ は状態 S にあるオブジェクト O と対応する CSP プロセスである．例えば $\mathcal{P}_D^{seq}(T_1)$ は *User* オブジェクトが標準状態から *login* メッセージを受信するシナリオに対応するプロセスであるが，シーケンス図は実行可能なひとつのシナリオを表すものであり，*User* が標準状態から実行可能なイベントは *Login* のみとは限らない．そのため，単一のシナリオと対応する $\mathcal{P}_D^{seq}(T_1)$ とオブジェクトの特定の状態と対応する $\mathcal{P}_D^{st}(User,d)$ は異なるプロセスとして変換されている．検証対象となるシーケンス図中にオブジェクト O の状態 S が複数回出現する場合，$\mathcal{P}_D^{st}(O,S)$ は状態 S から始まるすべての遷移を合成したものとなる．$\mathcal{P}_D^{st}(O,S)$ の定義については 4.2 節で述べる．

プロセス $\mathcal{P}_D^{seq}(T)$ は，メッセージのパラメータが空集合の場合次のように定義できる．

$$
\mathcal{P}_D^{seq}(T) = \mathcal{M}(M) \rightarrow \mathcal{P}_D^{st}(O,S2)
$$

ただし，

$$
\begin{aligned}
\mathcal{M}(M) &= call\_.OS.OR.MN \\
T &= (O,S1,M,S2) \\
M &= (MN,OS,OR,\{\})
\end{aligned}
$$

まず，4.2 節から 4.3 節ではメッセージのパラメータが空集合の場合のみを説明する．メッセージがパラメータを持つ場合については 4.4 節で述べる．

4.2 シーケンス図合成演算子を用いた CSP の合成

複数のシーケンス図から得られた CSP の合成に関して，直観的に以下のことが期待される．

1. 合成後のオブジェクトは，合成前のシーケンス図に含まれる遷移をすべて実行できる．

2. 同一のふるまいを表す遷移を複数のシーケンス図に記述することができる．同一の状態から始まる同一のメッセージが複数のシーケンス図に記述されていた場合，それらは同一のふるまいを表すものであり，実行時にどちらのシーケンス図が選択されたかは区別できない．

3. ある状態から送受信可能なメッセージが複数記述されていた場合，送信オブジェクトがそのうちの 1 つを非決定的に選択して送信する．ソフトウェアの実装ではオブジェクトの内部状態によって決定的にメッセージが選択されるが，シーケンス図が使われる上流設計では，条件判断に必要な情報が記述されない場合が多い．そのような場合，非決定的な選択として CSP に変換することで，詳細化後のモデルとの詳細化関係の検証が可能となる．

例えば，図 4 中で *User* の *loggedin* 状態から送信可能なメッセージは *addToCart*, *buy*，*logout* の 3 種類がある．この選択は送信オブジェクトである *User* によって外部の要因に依存せずになされるため，非決定的な選択である．一方，*System* オブジェクトはこの 3 つのいずれのメッセージも受信可能でなければならない．これは，外部の要因によって決定的に定まる選択である．また，*buy* メッセージは 図 4b と図 4c の 2 ヶ所に記述されているが，これらはどちらも *User* オブジェクトの *loggedin* 状態から送信されるものであり，*User* オブジェクトのふるまいとしてはこの 2 つのメッセージは同一である．図 4b が選択されたか図 4c が選択されたかは *buy* メッセージ送信時には区別できず，次の受信メッセージである *ok* もしくは *empty* を受信した時に初めて区別される．

これらの期待を満たす合成手法を定義するため，**シーケンス図マージ演算子** ∘ と **内部選択化演算子** $ という 2 つの新しい演算子を導入する．∘ 演算子と $ 演算子をあわせてシーケンス図合成演算子と呼ぶ．これらの演算子を用いると，プロセス *P*, *Q*, *R*... を合成した結果は $(P \circ Q \circ R...)\$\Sigma!$ と表現できる．ここで，$\Sigma!$ は *P*, *Q*, *R*... に含まれるイベントのうち，送信イベントの集合である．演算子 ∘ は複数のシーケンス図に記述された同一の遷移を統合するために，演算子 $ は送信イベントを非決定的な選択とするために利用される．

シーケンス図合成演算子を含む CSP を eCSP と呼ぶ．eCSP は次のように定義できる．

定義 1

$$
\begin{aligned}
C &::= C_X\|_Y C \mid C \setminus X \mid P \\
P &::= a \rightarrow P \mid P \,\Box\, P \mid P \sqcap P \mid P \circ P \mid P\$X \mid PN
\end{aligned}
$$

ここで，*a* はイベント名，*X* および *Y* はイベントの集合，*PN* はプロセス名である．プロセス名は $PN = P$ の形の式で定義される． ■

この定義では，並行合成および隠蔽が他の演算子から分離されている．これは，オブジェクト間の接続関係が動的には変化しないことを表している．システムの構造が動的に変化する場合への対応は今後の課題の 1 つである．

シーケンス図合成演算子は，以下の性質を持つことが期待される．

(1) 合成後のオブジェクトは，合成前のシーケンス図に含まれる遷移をすべて実行できる．これは，外部選択 □ がトレース集合を保存することをふまえて，次の式で表現できる．なお，この性質はトレース集合のみを考えるため，失敗等価になるとは限らない．

$$(P \,\square\, Q \,\square\, R \ldots) =_T (P \circ Q \circ R \ldots)\$\Sigma!$$

(2) 同一の状態から始まる同一のメッセージが複数のシーケンス図に記述されていた場合，それらは同一のふるまいを表すものであり，実行時にどちらのシーケンス図が選択されたかは区別できない．例えば，あるオブジェクトのふるまいが $a \to b \to P$ と $a \to c \to Q$ という 2 つのプロセスで表わされる場合，このオブジェクトはイベント a の実行時には選択を行わず，その後に b と c のどちらのイベントも実行できなければならない．

$$\begin{aligned}&((a \to b \to P) \circ (a \to c \to Q))\$\Sigma! \\ =_F\quad &(a \to ((b \to P) \circ (c \to Q)))\$\Sigma!\end{aligned}$$

一般的には次の性質が期待される．

$$((a \to P) \circ (a \to Q))\$\Sigma! \quad =_F \quad a \to (P \circ Q)\$\Sigma!$$

この等式では左辺を展開した結果である右辺にも ∘ 演算子と $ 演算子が残るため，簡単な糖衣構文として定義することはできない．

(3) ある状態から送受信可能なメッセージが複数記述されていた場合，送信オブジェクトがそのうちの 1 つを非決定的に選択して送信する．CSP では，非決定的な選択には内部選択 ⊓ を利用できる．また，受信オブジェクトは送信オブジェクトが選択したメッセージをすべて処理できなければならないが，このふるまいは CSP における外部選択 □ のふるまいと一致する．送信イベントの集合は Σ! で与えられるため，以下の式で期待される性質を表現できる．

$$((a \to P) \circ (b \to Q))\$\Sigma! \quad =_F \quad (a \to P\$\Sigma!) \sqcap (b \to Q\$\Sigma!)$$

ただし $a \in \Sigma!, b \in \Sigma!$

$$((a \to P) \circ (b \to Q))\$\Sigma! \quad =_F \quad (a \to P\$\Sigma!) \,\square\, (b \to Q\$\Sigma!)$$

ただし $a \notin \Sigma!, b \notin \Sigma!$

演算子 ∘ および $ はこれらの期待されるふるまいを実現するように定義される．期待 (2) を満たすため，∘ 演算子は複数のシーケンス図中の同一のイベントを 1 つ

の CSP のイベントに集約する．期待 (3) を満たすため，$ 演算子は送信イベントに対応する選択を外部選択ではなく内部選択とする．

シーケンス図マージ演算子 $P \circ Q$ は外部選択 $P \,\square\, Q$ と似ているが，同一のイベントを 1 つの CSP のイベントに集約する点が異なる．もし P と Q が同一のトレース s を実行でき，その後にイベント a が失敗しないのであれば，$P \circ Q$ も a を失敗せずに実行できる．$P \circ Q$ はトレース集合と失敗集合を用いて次のように定義できる．

定義 2

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

$$\begin{aligned} failures(P \circ Q) = \{(s, X) | \\ & (s, X) \in failures(P) \cup failures(Q), \\ & g(s, P) \Rightarrow (s, X) \in failures(P), \\ & g(s, Q) \Rightarrow (s, X) \in failures(Q)\} \end{aligned}$$

ここで，$g(s, P)$ は次のように定義され，プロセス P がトレース s の実行に失敗しないことを表す．

$$\begin{aligned} g(\langle\rangle, P) &= true \\ g(s^\frown\langle a\rangle, P) &= g(s, P) \wedge ((s, \{a\}) \notin failures(P)) \end{aligned}$$

■

$\circ$ 演算子は結合則と交換則を満たす．

$$\begin{aligned} (P1 \circ P2) \circ P3 &=_F P1 \circ (P2 \circ P3), \\ P1 \circ P2 &=_F P2 \circ P1 \end{aligned}$$

$\circ$，$\square$，$\sqcap$ の各演算子は結合則と交換則を満たすため，3 個以上のプロセスを合成する場合に合成の順序を考慮する必要はない．そのため，n 個のプロセスを合成する場合には以下の省略記法を用いることができる．

$$\begin{aligned} \bigcirc_{i \in \{0 \ldots n\}} @P_i &= P_0 \circ P_1 \circ \cdots \circ P_n \\ \square_{i \in \{0 \ldots n\}} @P_i &= P_0 \,\square\, P_1 \,\square\, \ldots \,\square\, P_n \\ \sqcap_{i \in \{0 \ldots n\}} @P_i &= P_0 \sqcap P_1 \sqcap \ldots \sqcap P_n \end{aligned}$$

この記法を用いると，次の定理が成り立つ．

定理 1

(1) $\bigcirc_{i \in I} @(a \rightarrow P_i) =_F a \rightarrow (\bigcirc_{i \in I} @P_i)$

(2) $(i \neq j \Rightarrow a_i \neq a_j) \Rightarrow$
$\bigcirc_{i \in I} @(a_i \rightarrow P_i) =_F \square_{i \in I} @(a_i \rightarrow P_i)$

ここで，I は有限かつ空でないインデックス集合である． ■

この定理は次の補題を用いて証明することができる．[2]

$$(1) \quad g(s, P) \Leftrightarrow g(\langle a \rangle \hat{} s, a \to P)$$
$$(2) \quad a \neq b \Rightarrow g(\langle a \rangle \hat{} s, b \to P) = false$$

定理 1(1) は合成するすべてのイベントが等しい場合に適用できる．定理 1(2) は合成するすべてのイベントが異なる場合に適用できる．それ以外の場合は，次の系が適用できる．

系 1

$$\bigcirc_{i \in I} @(a_i \to P_i) \quad =_F \quad \Box_{a \in A} @(a \to P'_a)$$

ここで，

$$A = \{a \mid \exists i.\ i \in I, a_i = a\}$$
$$P'_a = \bigcirc_{\{i \in I \mid a_i = a\}} @P_i$$

■

P'_a の定義は ∘ 演算子を含んでいるため，系 1 は ∘ を定義するには不十分である．∘ を系 1 の等式の不動点として定義することもできるが，定義 2 のようにトレース集合と失敗集合を用いて定義するほうが扱いやすい．

内部選択化演算子 $ は送信イベントに対応する選択を外部選択ではなく内部選択とする．この演算子は次のように定義される．

定義 3

$$traces(P\$Z) = traces(P)$$

$$failures(P\$Z) = failures(P) \cup \{(s, X) \mid$$
$$\exists Y.((s, Y) \in failures(P) \wedge X \subseteq Y \cup Z),$$
$$\exists a.(s \hat{} \langle a \rangle \in traces(P) \wedge a \notin X))\}$$

■

$(X \subseteq Y \cup Z)$ という条件により，Z に含まれるイベントが $P\$Z$ の失敗集合に含まれる．ただし，$(a \notin X)$ という条件により，そのイベントが唯一の実行可能なイベントである場合は失敗集合に含まない．

$ 演算子に関しては以下の定理が成り立つ．[3]

[2]詳細な証明は付録 B を参照.
[3]証明は付録 C を参照.

定理 2

(1) $(A_! = \phi) \Rightarrow$

$(\Box_{a \in A} @(a \rightarrow P_a))\$\Sigma! =_F \Box_{a \in A_?} @(a \rightarrow P_a \$\Sigma!)$

(2) $(A_! \neq \phi \wedge A_? = \phi) \Rightarrow$

$(\Box_{a \in A} @(a \rightarrow P_a))\$\Sigma! =_F \sqcap_{a \in A_!} @(a \rightarrow P_a \$\Sigma!)$

(3) $(A_! \neq \phi \wedge A_? \neq \phi) \Rightarrow$

$(\Box_{a \in A} @(a \rightarrow P_a))\$\Sigma! =_F \sqcap_{a \in A_!} @(a \rightarrow P_a \$\Sigma!) \triangleright \Box_{b \in A_?} @(b \rightarrow P_b \$\Sigma!)$

ここで，A はイベント集合であり，$A_! = A \cap \Sigma!$，$A_? = A - \Sigma!$ である．「$-$」は差集合を表す．[4] ■

定理 2(3) で，$\triangleright$ はタイムアウト演算子と呼ばれ，次のように定義される．

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

直観的には，$P \triangleright Q$ は最初は P のようにふるまい，しばらくすると Q のようにふるまうプロセスとみなすことができる．$ 演算子をトレース集合と失敗集合で定義して定理 2(3) を導くまでは，メッセージ送信と受信が同時に実行可能な場合に合成結果をどうするべきかは明確でなかったが，定理 2(3) により，一貫性があり直観的にも正しい合成方法を導き出すことができた．

さらに，$\circ$ 演算子と $ 演算子は失敗詳細化関係を保存する．これは CSP の演算子として重要な性質である．演算子が詳細化関係を保存することにより，$\circ$ や $ を含む式に対して部分式ごとに詳細化を証明・検証できる．

定理 3

$$P1 \sqsubseteq_F Q1, P2 \sqsubseteq_F Q2 \quad \Rightarrow \quad P1 \circ P2 \sqsubseteq_F Q1 \circ Q2,$$
$$P \sqsubseteq_F Q \quad \Rightarrow \quad P\$Z \sqsubseteq_F Q\$Z$$

■

特に，$\circ$ 演算子の定義中の $g(s, P)$ は $s \in traces(P)$ と似た意味を表すが，$g(s, P)$ の代わりに $s \in traces(P)$ を用いた演算子は詳細化関係を保存しない．$\circ$ は失敗詳細化関係を保存するように注意深く定義されている．

以上の定義および定理を用いると，シーケンス図合成に期待されるふるまいが満たされていることは次のように確認できる．

(1) $(P \Box Q \Box R \ldots) =_T (P \circ Q \circ R \ldots)\$\Sigma!$

証明：定義より $traces(P \Box Q) = traces(P \circ Q)$，$traces(P\$Z) = traces(P)$. ■

[4]定理 2 中の $\Sigma!$ は任意の集合 Z でも成り立つが，意図を明確化するために具体的な集合である $\Sigma!$ を用いている．

(2) $((a \to P) \circ (a \to Q))\$\Sigma! =_F a \to (P \circ Q)\$\Sigma!$

証明：定理 1(1) より $(a \to P) \circ (a \to Q) =_F a \to (P \circ Q)$. ■

(3.1) $((a \to P) \circ (b \to Q))\$\Sigma! =_F (a \to P\$\Sigma!) \sqcap (b \to Q\$\Sigma!)$ ただし $a \in \Sigma!, b \in \Sigma!$.

証明：

$$\begin{aligned} & ((a \to P) \circ (b \to Q))\$\Sigma! \\ =_F \quad & ((a \to P) \square (b \to Q))\$\Sigma! \\ =_F \quad & (a \to P\$\Sigma!) \sqcap (b \to Q\$\Sigma!) \end{aligned}$$

2 行目は定理 1(2)，3 行目は定理 2(2) より. ■

(3.2) $((a \to P) \circ (b \to Q))\$\Sigma! =_F (a \to P\$\Sigma!) \square (b \to Q\$\Sigma!)$ ただし $a \notin \Sigma!, b \notin \Sigma!$.

証明：

$$\begin{aligned} & ((a \to P) \circ (b \to Q))\$\Sigma! \\ =_F \quad & ((a \to P) \square (b \to Q))\$\Sigma! \\ =_F \quad & (a \to P\$\Sigma!) \square (b \to Q\$\Sigma!) \end{aligned}$$

2 行目は定理 1(2)，3 行目は定理 2(1) より. ■

なお，$\circ$ 演算子と \$ 演算子が失敗詳細化関係を保存するため，証明中で部分式に対して定理を適用することが可能となっている.

$\circ$ 演算子を用いると，$\mathcal{P}^{st}_D(O, S)$ は次のように計算できる.

$$\mathcal{P}^{st}_D(O, S) \quad = \quad \bigcirc_{T \in \mathcal{T}_D(O,S)} @ \mathcal{P}^{seq}_D(T)$$

ここで,

$$\mathcal{T}_D(O, S) \quad = \quad \{T \in tr(D) \mid O = obj(T), S = prev(T)\}$$

$tr(D)$ は D 内のすべての遷移の集合，$obj(T)$ は遷移 T を実行するオブジェクト，$prev(T)$ は遷移 T の直前の状態である.

$$\begin{aligned} tr(D) \quad &= \quad \{T \mid \exists S\, D.\ T \in S\, D, S\, D \in D\} \\ obj(T) \quad &\in \quad \{O \mid \exists S\, 1. \exists M. \exists S\, 2.\ T = (O, S\, 1, M, S\, 2)\} \\ prev(T) \quad &\in \quad \{S\, 1 \mid \exists C. \exists M. \exists S\, 2.\ T = (O, S\, 1, M, S\, 2)\} \end{aligned}$$

例えば，図 4 の $User$ および $System$ は次のようにふるまう．

$$
\begin{aligned}
\mathcal{P}^{st}_D(User, d) &= Login \rightarrow \mathcal{P}^{st}_D(User, i_1) \\
\mathcal{P}^{st}_D(User, i_1) &= OK \rightarrow \mathcal{P}^{st}_D(User, loggedin) \\
\mathcal{P}^{st}_D(User, loggedin) &= (AddToCart \rightarrow \mathcal{P}^{st}_D(User, i_3) \\
&\quad \circ\ Buy \rightarrow \mathcal{P}^{st}_D(User, i_4) \\
&\quad \circ\ Buy \rightarrow \mathcal{P}^{st}_D(User, i_7) \\
&\quad \circ\ Logout \rightarrow \mathcal{P}^{st}_D(User, i_9)) \\
\mathcal{P}^{st}_D(User, i_3) &= OK \rightarrow \mathcal{P}^{st}_D(User, loggedin) \\
\mathcal{P}^{st}_D(User, i_4) &= OK \rightarrow \mathcal{P}^{st}_D(User, loggedin) \\
\mathcal{P}^{st}_D(User, i_7) &= Empty \rightarrow \mathcal{P}^{st}_D(User, loggedin) \\
\mathcal{P}^{st}_D(User, i_9) &= OK \rightarrow \mathcal{P}^{st}_D(User, d) \\
\mathcal{P}^{st}_D(System, d) &= (Login \rightarrow \mathcal{P}^{st}_D(System, i_2) \\
&\quad \circ\ AddToCart \rightarrow \mathcal{P}^{st}_D(System, i_5) \\
&\quad \circ\ Buy \rightarrow \mathcal{P}^{st}_D(System, i_8) \\
&\quad \circ\ Logout \rightarrow \mathcal{P}^{st}_D(System, i_{10})) \\
\mathcal{P}^{st}_D(System, i_2) &= OK \rightarrow \mathcal{P}^{st}_D(System, d) \\
\mathcal{P}^{st}_D(System, i_5) &= OK \rightarrow \mathcal{P}^{st}_D(System, hasCart) \\
\mathcal{P}^{st}_D(System, i_8) &= Empty \rightarrow \mathcal{P}^{st}_D(System, d) \\
\mathcal{P}^{st}_D(System, i_{10}) &= OK \rightarrow \mathcal{P}^{st}_D(System, d) \\
\mathcal{P}^{st}_D(System, hasCart) &= Buy \rightarrow \mathcal{P}^{st}_D(System, i_6) \\
\mathcal{P}^{st}_D(System, i_6) &= OK \rightarrow \mathcal{P}^{st}_D(System, d)
\end{aligned}
$$

シーケンス図中の各オブジェクトの初期状態 $Init$ が与えられれば，シーケンス図で定義された設計全体を表す CSP のプロセスは次の式で表現できる．

$$\mathcal{P}^{design}_D(Init) = \|_{O \in \mathcal{O}_D} @[\Sigma_D(O)]\ \mathcal{P}^{st}_D(O, Init(O))\$\Sigma!_D(O)$$

ここで，

$$
\begin{aligned}
\mathcal{O}_D &= \{O \mid \exists T \in tr(D).\ O = obj(T)\} \\
\Sigma_D(O) &= \{\mathcal{M}(M) \mid \exists T \in tr(D).\ M = mes(T),\ O = obj(T)\} \\
\Sigma!_D(O) &= \{\mathcal{M}(M) \mid \exists T \in tr(D).\ M = mes(T),\ O = sender(M)\}
\end{aligned}
$$

また，$mes(T)$ は遷移 T に対応するメッセージ，$sender(M)$ はメッセージ M の送信オブジェクトである．

$$
\begin{aligned}
mes(T) &\in \{M \mid \exists C.\exists S1.\exists S2.\ T = (O, S1, M, S2)\} \\
sender(M) &\in \{OS \mid \exists MN.\exists OR.\ M = (MN, OS, OR)\}
\end{aligned}
$$

$\|_i@[X_i]\,P_i$ はプロセス P_i をイベント集合 X_i で同期させて並行合成したプロセスである．

$$\|_{i\leq n}@[Xi]\,Pi \quad = \quad ((P_0{}_{X_0}\|_{X_1}P_1)_{X_0\cup X_1}\|_{X_2}P_2)\ldots{}_{(X_0\cup\cdots\cup X_{n-1})}\|_{X_n}P_n$$

例えば，図 4 の $User$ および $System$ オブジェクトは次のようにふるまう．

$$\mathcal{P}_D^{st}(User, d)\$\{Login, AddToCart, Buy, Logout\}$$
$$_{\Sigma_D}\|_{\Sigma_D}\mathcal{P}_D^{st}(System, d)\$\{OK, Empty\}$$

ここで，$\Sigma_D = \{Login, AddToCart, Buy, Logout, OK, Empty\}$ である．この式は，$User$ の初期状態が $default$ 状態であり，その送信メッセージ集合が $\{Login, AddToCart, Buy, Logout\}$ であること，および $System$ の初期状態が $default$ 状態であり，その送信メッセージ集合は $\{OK, Empty\}$ であることを示している．

入力したシーケンス図を検証するには，この CSP のプロセスがデッドロックしないことや詳細化関係が正しいことを検証すればよい．

なお，与えられたシーケンス図中の特定の 1 枚に着目した場合，各オブジェクトの初期状態 $Init$ をシーケンス図の先頭の状態と一致させれば $\mathcal{P}_D^{design}(Init)$ はシーケンス図に書かれたメッセージをその順序で実行できる．これは，合成されたプロセスが合成前のすべてのトレースを含むことから導ける．ただし，システム全体のふるまい $\mathcal{P}_D^{design}(Init)$ では，オブジェクト間の想定外の相互作用によってシーケンス図に明記されていないふるまいを生じることもある．システム全体が期待通りにふるまうことの検証にはモデル検査が有効である．4.3 節で説明する手法により，シーケンス図合成演算子を含むプロセスである $\mathcal{P}_D^{design}(Init)$ をシーケンス図合成演算子を含まない標準的な CSP のプロセスに変換し，既存のモデル検査器を活用して検証することが可能となる．

4.3 シーケンス図合成演算子を含まない CSP への変換

本節では，シーケンス図合成演算子を含む任意の eCSP のプロセス $\mathcal{P}_D^{st}(O, S)\$\Sigma!$ をシーケンス図合成演算子を含まない標準的な CSP のプロセス $Synth_D(O, \mathcal{S})$ に変換する手法について述べる．検証対象のプロセスを標準的な CSP のプロセスで表すことにより，既存のモデル検査器での検証が可能となる．

$$Synth_D(O, \mathcal{S}) \quad =_F \quad (\bigcirc_{S\in\mathcal{S}}@\mathcal{P}_D^{st}(O, S))\$\Sigma!_D(O)$$

ここで，$\Sigma!_D(O)$ は 4.2 節で定義した送信イベントの集合である．

$Synth_D$ は次のように定義される．

$$
\begin{aligned}
&(1)\quad (A!_D(O,\mathcal{S}) = \phi) \Rightarrow \\
&\qquad Synth_D(O,\mathcal{S}) = \ \Box_{a \in A?_D(O,\mathcal{S})} @a \rightarrow Synth_D(O, N_D(O,\mathcal{S},a)) \\
&(2)\quad (A!_D(O,\mathcal{S}) \neq \phi \wedge A?_D(O,\mathcal{S}) = \phi) \Rightarrow \\
&\qquad Synth_D(O,\mathcal{S}) = \sqcap_{a \in A!_D(O,\mathcal{S})} @a \rightarrow Synth_D(O, N_D(O,\mathcal{S},a)) \\
&(3)\quad (A!_D(O,\mathcal{S}) \neq \phi \wedge A?_D(O,\mathcal{S}) \neq \phi) \Rightarrow \\
&\qquad Synth_D(O,\mathcal{S}) = \sqcap_{a \in A!_D(O,\mathcal{S})} @a \rightarrow Synth_D(O, N_D(O,\mathcal{S},a)) \\
&\qquad\qquad \triangleright\ \Box_{a \in A?_D(O,\mathcal{S})} @a \rightarrow Synth_D(O, N_D(O,\mathcal{S},a))
\end{aligned}
$$

ここで，$N_D(O,\mathcal{S},a)$ は遷移先の状態の集合，$A!_D(O,\mathcal{S})$ はその状態における送信イベントの集合，$A?_D(O,\mathcal{S})$ はその状態における受信イベントの集合である．

$$
\begin{aligned}
N_D(O,\mathcal{S},a) &= \{next(T) \mid T \in tr(D), \\
&\quad O = obj(T), prev(T) \in \mathcal{S}, a = \mathcal{M}(mes(T))\} \\
A!_D(O,\mathcal{S}) &= \{\mathcal{M}(M) \mid \exists T \in tr(D). \\
&\quad M = mes(T), O = sender(M), prev(T) \in \mathcal{S}\} \\
A?_D(O,\mathcal{S}) &= \{\mathcal{M}(M) \mid \exists T \in tr(D). \\
&\quad M = mes(T), O = receiver(M), prev(T) \in \mathcal{S}\}
\end{aligned}
$$

また，$receiver(M)$ はメッセージ M の受信オブジェクトである．

$$
receiver(M) \ \in \ \{OR \mid \exists MN.\exists OS.\ M = (MN, OS, OR)\}
$$

等式 $Synth_D(O,\mathcal{S}) =_F (\bigcirc_{S \in \mathcal{S}} @\mathcal{P}^{st}_D(O,S))\$\Sigma!_D(O)$ は 4.2 節の系 1 および定理 2 より構造的帰納法で証明できる．

例えば，図 4 の $User$ オブジェクトのふるまいを表すプロセス $\mathcal{P}^{st}_D(User,d)$ \$ $\{Login, AddToCart, Buy, Logout\}$ と $System$ オブジェクトのふるまいを表すプロセス $\mathcal{P}^{st}_D(System,d)\$\{OK, Empty\}$ は次のプロセス $Synth_D(User, \{d\})$ および $Synth_D$

$(System, \{d\})$ に変換できる．

$$
\begin{aligned}
Synth_D(User, \{d\}) &= Login \rightarrow Synth_D(User, \{i_1\}) \\
Synth_D(User, \{i_1\}) &= OK \rightarrow Synth_D(User, \{loggedin\}) \\
Synth_D(User, \{loggedin\}) &= (AddToCart \rightarrow Synth_D(User, \{i_3\}) \\
&\sqcap\ Buy \rightarrow Synth_D(User, \{i_4, i_7\}) \\
&\sqcap\ Logout \rightarrow Synth_D(User, \{i_9\})) \\
Synth_D(User, \{i_3\}) &= OK \rightarrow Synth_D(User, \{loggedin\}) \\
Synth_D(User, \{i_4, i_7\}) &= (OK \rightarrow Synth_D(User, \{loggedin\}) \\
&\Box\ Empty \rightarrow Synth_D(User, \{loggedin\})) \\
Synth_D(User, \{i_9\}) &= OK \rightarrow Synth_D(User, \{d\})
\end{aligned}
$$

$$
\begin{aligned}
Synth_D(System, \{d\}) &= (Login \rightarrow Synth_D(System, \{i_2\}) \\
&\Box\ AddToCart \rightarrow Synth_D(System, \{i_5\}) \\
&\Box\ Buy \rightarrow Synth_D(System, \{i_8\}) \\
&\Box\ Logout \rightarrow Synth_D(System, \{i_{10}\})) \\
Synth_D(System, \{i_2\}) &= OK \rightarrow Synth_D(System, \{d\}) \\
Synth_D(System, \{i_5\}) &= OK \rightarrow Synth_D(System, \{hasCart\}) \\
Synth_D(System, \{i_8\}) &= Empty \rightarrow Synth_D(System, \{d\}) \\
Synth_D(System, \{i_{10}\}) &= OK \rightarrow Synth_D(System, \{d\}) \\
Synth_D(System, \{hasCart\}) &= Buy \rightarrow Synth_D(System, \{i_6\}) \\
Synth_D(System, \{i_6\}) &= OK \rightarrow Synth_D(System, \{d\})
\end{aligned}
$$

$\mathcal{S}$ はオブジェクトの状態の部分集合であるため，最悪の場合でも状態数 N に対して $O(2^N)$ 回の変換を行えば任意のプロセスを標準的な CSP のプロセスに変換できる．しかし，実際のシーケンス図においては，初期状態から遷移可能な $\mathcal{S}$ の数は十分に小さい．例えば，図 4 の *User* オブジェクトには 7 つの状態がある（$2^N = 128$）が，そのうちの 6 つの部分集合のみが初期状態から到達可能である．必要となる $\mathcal{S}$ に対してのみ $Synth_D(O, \mathcal{S})$ を展開することで，十分に短い時間で変換を完了できる．

変換されたプロセス $Synth_D(User, \{d\})_{\Sigma_D} \| _{\Sigma_D} Synth_D(System, \{d\})$ を既存のモデル検査器で検査することで，この設計がデッドロックする可能性があることが検出できる．*login* → *ok* → *addToCart* → *ok* というトレースの後，*User* は *loggedin* 状態にあり，*buy* もしくは *logout* もしくは *addToCart* メッセージを送信できる．しかし，*System* は *hasCart* 状態にあり，*buy* メッセージしか受信できない．もし *User*

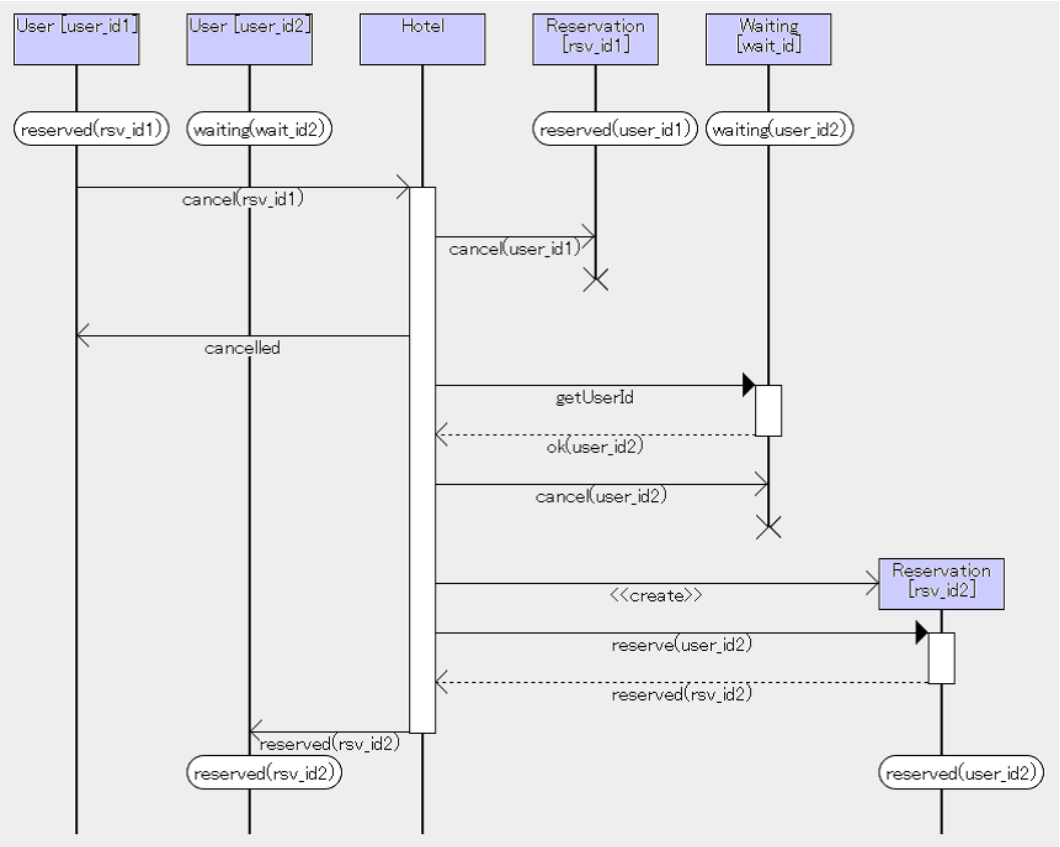


図 7: ホテル予約システムのシーケンス図

が *logout* もしくは *addToCart* を選択した場合，このモデルはデッドロックする．開発者はこのような状況を考慮し，*logout* や *addToCart* が送信された場合のシーケンス図を作成する必要がある．

4.4 パラメータを持つシーケンス図の変換

ここまでの説明で用いた図 4 では，オブジェクトやメッセージ，状態にパラメータを用いていなかった．本節では，パラメータを持つシーケンス図の意味論および変換方法について説明する．

図 7 はホテル予約システムのシーケンス図である．このシーケンス図には以下のような特徴がある．

- *User*[*user_id*1] と *User*[*user_id*2] は同一クラスの異なるインスタンスである．
- メッセージはパラメータを持つ．
- 状態はパラメータを持つ．
- *Reservation*[*rsv_id*2] はシナリオの途中で生成される．

- *Reservation*[*rsv_id*1] と *Waiting*[*wait_id*] はシナリオの途中で破棄される．

複雑なシステムを設計する際にはこれらの特徴をもつシーケンス図は有用である．

しかし，シーケンス図において，同一クラスの複数インスタンスが存在する場合のふるまいは必ずしも明確ではない．本論文では，実際のシーケンス図の調査の結果として，下記のふるまいを仮定する．

- 返信メッセージは対応する同期メッセージの送信元オブジェクトに返される．
- もし送信オブジェクトが受信オブジェクトのプロセス ID を知っている場合，その ID を持つオブジェクトがメッセージを受信する．
- もし送信オブジェクトが受信オブジェクトのプロセス ID を知らない場合，受信オブジェクトは非決定的に選択される．
- 明示的に示されない限り，プロセス ID は単一の活性区間の間のみ保持される．

以上のようなふるまいを実現するため，各オブジェクトでどのようにデータが保持されるかを次のように定める．

- オブジェクトがメッセージを受信した場合，送信オブジェクトの ID およびメッセージのパラメータがメモリに加えられる．例えば，図 7 において，*Hotel* オブジェクトは *cancel*(*rsv_id*1) メッセージを受信した後，その送信オブジェクトである *User* の ID をメモリ上に保持する．これにより，*Hotel* オブジェクトは *cancelled* メッセージを同一の *User* オブジェクトに返却することができる．
- オブジェクトがメッセージを送信する場合，もし受信オブジェクトの ID がメモリに存在するならば，その ID のオブジェクトのみがメッセージを受信できる．
- オブジェクトがメッセージを送信する場合，もし受信オブジェクトの ID がメモリに存在しないならば，受信オブジェクトはそのメッセージを受信可能なオブジェクトのうち 1 つが自動的に選択される．選択された ID は送信オブジェクトのメモリに加えられる．例えば，図 7 中で，*getUserId* メッセージは任意の *Waiting* オブジェクトが受信できる．それに続く *cancel*(*user_id*2) メッセージは *getUserId* メッセージで選択されたオブジェクトのみが受信できる．
- 活性区間の完了時に，その活性区間中で記憶されたメモリは破棄される．ただし，活性区間の直後にパラメータを持つ状態不変式がある場合，状態不変式のパラメータに指定された ID は次の活性区間の完了まで保持される．例えば，図 7 中で，*Hotel* のメモリは *reserved*(*rsv_id*2) メッセージの送信後に削除される．また，*Reservation*[*rsv_id*2] のメモリは *reserved*(*rsv_id*2) メッセージの後で削除され，*user_id*2 のみ保持される．

ここで，メモリに ID 番号が記憶されているかどうかは，中間状態のパラメータとして形式化できる．例えば，図 7 の *Hotel* オブジェクトは次のように形式的に記述できる．

$$
\begin{aligned}
\{T_1, T_2, \ldots, T_{10}\} &\subseteq Transitions \\
T_1 &= (Hotel, d, Cancel_1, i_1) \\
T_2 &= (Hotel, i_1, Cancel_2, i_2) \\
T_3 &= (Hotel, i_2, Cancelled, i_3) \\
T_4 &= (Hotel, i_3, GetUserId, i_4) \\
T_5 &= (Hotel, i_4, OK, i_5) \\
T_6 &= (Hotel, i_5, Cancel_3, i_6) \\
T_7 &= (Hotel, i_6, Create, i_7) \\
T_8 &= (Hotel, i_7, Reserve, i_8) \\
T_9 &= (Hotel, i_8, Reserved_1, i_9) \\
T_{10} &= (Hotel, i_9, Reserved_2, d)
\end{aligned}
$$

$$
\begin{aligned}
&\{Cancel_1, Cancel_2, Cancelled, GetUserId, OK, \\
&Cancel_3, Create, Reserve, Reserved_1, Reserved_2\} \subseteq Messages
\end{aligned}
$$

$$
\begin{aligned}
Cancel_1 &= (\texttt{cancel}, User_1, Hotel, \{\texttt{rsv\_id1}\}) \\
Cancel_2 &= (\texttt{cancel}, Hotel, Reservation_1, \{\texttt{user\_id1}\}) \\
Cancelled &= (\texttt{cancel}, Hotel, User_1, \{\}) \\
GetUserId &= (\texttt{getUserId}, Hotel, Waiting, \{\}) \\
OK &= (\texttt{ok}, Waiting, Hotel, \{\texttt{user\_id2}\}) \\
Cancel_3 &= (\texttt{cancel}, Hotel, Waiting, \{\texttt{user\_id2}\}) \\
Create &= (\texttt{<<create>>}, Hotel, Reservation_2, \{\}) \\
Reserve &= (\texttt{reserve}, Hotel, Reservation_2, \{\texttt{user\_id2}\}) \\
Reserved_1 &= (\texttt{reserved}, Reservation_2, Hotel, \{\texttt{user\_id2}\}) \\
Reserved_2 &= (\texttt{reserved}, Hotel, User_2, \{\texttt{rsv\_id2}\})
\end{aligned}
$$

$$
\{Hotel, User_1, Reservation_1, Waiting, Reservation_2, User_2, \} \subseteq Objects
$$

$$
\begin{aligned}
\mathit{Hotel} &= (\texttt{Hotel}, \emptyset) \\
\mathit{User}_1 &= (\texttt{User}, \texttt{user\_id1}) \\
\mathit{Reservation}_1 &= (\texttt{Reservation}, \texttt{rsv\_id1}) \\
\mathit{Waiting} &= (\texttt{Waiting}, \texttt{wait\_id}) \\
\mathit{Reservation}_2 &= (\texttt{Reservation}, \texttt{rsv\_id2}) \\
\mathit{User}_2 &= (\texttt{User}, \texttt{user\_id2})
\end{aligned}
$$

$$
\begin{aligned}
\{d, i_1, \ldots, i_9\} &\subseteq \mathit{States} \\
d &= (\mathrm{d}, \{\}) \\
i_1 &= (\mathrm{i1}, \{\texttt{user\_id1}, \texttt{rsv\_id1}\}) \\
i_2 &= (\mathrm{i2}, \{\texttt{user\_id1}, \texttt{rsv\_id1}\}) \\
i_3 &= (\mathrm{i3}, \{\texttt{user\_id1}, \texttt{rsv\_id1}\}) \\
i_4 &= (\mathrm{i4}, \{\texttt{user\_id1}, \texttt{rsv\_id1}, \texttt{wait\_id}\}) \\
i_5 &= (\mathrm{i5}, \{\texttt{user\_id1}, \texttt{rsv\_id1}, \texttt{wait\_id}, \texttt{user\_id2}\}) \\
i_6 &= (\mathrm{i6}, \{\texttt{user\_id1}, \texttt{rsv\_id1}, \texttt{wait\_id}, \texttt{user\_id2}\}) \\
i_7 &= (\mathrm{i7}, \{\texttt{user\_id1}, \texttt{rsv\_id1}, \texttt{wait\_id}, \texttt{user\_id2}, \texttt{rsv\_id2}\}) \\
i_8 &= (\mathrm{i8}, \{\texttt{user\_id1}, \texttt{rsv\_id1}, \texttt{wait\_id}, \texttt{user\_id2}, \texttt{rsv\_id2}\}) \\
i_9 &= (\mathrm{i9}, \{\texttt{user\_id1}, \texttt{rsv\_id1}, \texttt{wait\_id}, \texttt{user\_id2}, \texttt{rsv\_id2}\})
\end{aligned}
$$

i_1 は $(\texttt{cancel}, \mathit{User}_1, \mathit{Hotel}, \{\texttt{rsv\_id1}\})$ 受信後の状態であり，送信オブジェクトの ID である user_id1 とメッセージのパラメータである rsv_id1 が状態のパラメータに加えられる．i_4 は $(\texttt{getUserId}, \mathit{Hotel}, \mathit{Waiting}, \{\})$ 送信後の状態であり，受信オブジェクトの ID である wait_id が状態のパラメータに加えられる．i_5 は $(\mathrm{ok}, \mathit{Waiting}, \mathit{Hotel}, \{\texttt{user\_id2}\})$ 受信後の状態であり，メッセージのパラメータである user_id2 が状態のパラメータに加えられる．i_7 は $(\texttt{<<create>>}, \mathit{Hotel}, \mathit{Reservation}_2, \{\})$ 送信後の状態であり，受信オブジェクトの ID である rsv_id2 が状態のパラメータに加えられる．最後の $(\texttt{reserved}, \mathit{Hotel}, \mathit{User}_2, \{\texttt{rsv\_id2}\})$ 送信後は標準状態 d に戻るため，状態のパラメータも空集合に戻る．

本手法で用いる変数はオブジェクト ID のみであり，それ以外のデータは検証に用いない．入力モデル内で算術演算子や制御構造を記述可能とすればオブジェクト ID 以外の変数を検証対象とすることも可能ではあるが，シーケンス図はそのような詳細なふるまいを記述するには適しておらず，シーケンス図を検証対象とする本手法ではそのような需要は少ないと考えられる．

CSP のプロセス $\mathcal{P}_D^{seq}(T)$ は次のように定義できる．

$$
\begin{aligned}
&O = S2 \Rightarrow \\
&\quad \mathcal{P}_D^{seq}(T) = call\_?MN.OS.OR.P_1 \ldots P_N \rightarrow \mathcal{P}_D^{st}(O, S2) \\
&O = S1 \wedge id(OR) \in param(S1) \Rightarrow \\
&\quad \mathcal{P}_D^{seq}(T) = call\_!MN.OS.OR.P_1 \ldots P_N \rightarrow \mathcal{P}_D^{st}(O, S2) \\
&O = S1 \wedge id(OR) \notin params(S1) \Rightarrow \\
&\quad \mathcal{P}_D^{seq}(T) = \Box_{id(OR) \in Ids(OR)} @ (call\_!MN.OS.OR.P_1 \ldots P_N \rightarrow \mathcal{P}_D^{st}(O, S2)
\end{aligned}
$$

ここで，$id(O)$ はオブジェクトの ID，$param(S)$ は状態のパラメータ，Ids はオブジェクト ID として使われる可能性のある値の集合を表す．

$$
\begin{aligned}
T &= (O, S1, M, S2) \\
M &= (MN, OS, OR, \{P_1 \ldots P_N\}) \\
id(O) &\in \{I \mid \exists C.\ O = (C, I)\} \\
params(S) &\in \{P \mid \exists SN.\ S = (SN, P)\} \cup \{0\}
\end{aligned}
$$

$P_1 \ldots P_N$，および $id(OS)$，$id(OR)$ は CSP の変数名と対応する．パラメータが空集合かつ OR がオブジェクト ID を持たない場合には，MN, OS, OR は定数値であり，4.1 節で定義した $\mathcal{P}_D^{seq}(T)$ と等しい．この変換により，メッセージを受信する際にはチャネルから $P_1 \ldots P_N$，$id(OS)$ を受け取ることができる．また，メッセージ送信の際に受信オブジェクトの ID がメモリに存在するならばその ID が使われ，メモリに存在しない場合は外部選択によりそのメッセージを受信可能なオブジェクトのうち 1 つが選択される．

図 7 の *Hotel* オブジェクトは次のような CSP のプロセスに変換できる．

$$
\begin{aligned}
\mathcal{P}_D^{seq}(T_1) &= call\_?cancel.User.user\_id1.Hotel.0.rsv\_id1 \rightarrow \mathcal{P}_D^{st}(Hotel, i_1) \\
\mathcal{P}_D^{seq}(T_2) &= call\_!cancel.Hotel.0.Reservation.rsv\_id1.user\_id1 \\
&\qquad \rightarrow \mathcal{P}_D^{st}(Hotel, i_2) \\
\mathcal{P}_D^{seq}(T_3) &= call\_!cancel.Hotel.0.User.user\_id1 \rightarrow \mathcal{P}_D^{st}(Hotel, i_3) \\
\mathcal{P}_D^{seq}(T_4) &= \Box_{wait\_id \in Ids(Waiting)} @ \\
&\qquad call\_!getUserId.Hotel.0.Waiting.wait\_id \rightarrow \mathcal{P}_D^{st}(Hotel, i_4) \\
\mathcal{P}_D^{seq}(T_5) &= call\_?ok.Waiting.wait\_id.Hotel.0.user\_id2 \rightarrow \mathcal{P}_D^{st}(Hotel, i_5) \\
\mathcal{P}_D^{seq}(T_6) &= call\_!cancel.Hotel.0, Waiting.user\_id2 \rightarrow \mathcal{P}_D^{st}(Hotel, i_6) \\
\mathcal{P}_D^{seq}(T_7) &= \Box_{rsv\_id2 \in Ids(Reservation_2)} @ \\
&\qquad call\_!create.Hotel.0.Reservation.rsv\_id2 \rightarrow \mathcal{P}_D^{st}(Hotel, i_7) \\
\mathcal{P}_D^{seq}(T_8) &= call\_!reserve.Hotel.0.Reservation.rsv\_id2.user\_id2 \\
&\qquad \rightarrow \mathcal{P}_D^{st}(Hotel, i_8)
\end{aligned}
$$

$$\mathcal{P}_D^{seq}(T_9) = call\_?reserved.Reservation.rsv\_id2.Hotel.0.user\_id2 \rightarrow \mathcal{P}_D^{st}(Hotel, i_9)$$
$$\mathcal{P}_D^{seq}(T_{10}) = call\_!reserved.Hotel.0.User.user\_id2.rsv\_id2 \rightarrow \mathcal{P}_D^{st}(Hotel, d)$$

$\mathcal{P}_D^{seq}(T_4)$ および $\mathcal{P}_D^{seq}(T_7)$ では受信オブジェクトの ID がメモリに存在しないため，外部選択により *getUserId* および *create* の送信先を選択している．

$\mathcal{P}_D^{st}(O, S)$ はパラメータを含まない場合と同様の考え方で定義できるが，オブジェクトの ID や状態のパラメータは変数であり，名前が異なっていても合成対象となることに注意が必要である．例えば，図 7 では $Reservation[rsv\_id1]$ オブジェクトの $reserved(user\_id1)$ 状態と $Reservation[rsv\_id2]$ オブジェクトの $reserved(user\_id2)$ 状態が存在するが，この 2 つの状態は変数名を置換すれば同じ状態である．$\mathcal{P}_D^{st}(O, S)$ は次のように定義できる．

$$\mathcal{P}_D^{st}(O, S) = \bigcirc_{T \in \mathcal{T}'_D(O,S)} @ \mathcal{P}_D^{seq}(T) \, [\![{}^{id(O), params(S)} / _{id(obj(T)), params(prev(T))}]\!]$$
$$\mathcal{T}'_D(O, S) = \{T \in tr(D) \mid class(O) = class(obj(T)), \; name(S) = name(prev(T))\}$$
$$class(O) \in \{C \mid \exists I.\ O = (C, I)\}$$
$$name(S) \in \{SN \mid \exists P.\ S = (SN, P)\}$$

ここで，$P[\![{}^{a} / _{b}]\!]$ はプロセス P 中の b を a に置換したプロセスを表す．

1 つのクラスが複数のインスタンスを持つ場合，設計全体を表す $\mathcal{P}_D^{design}$ は複数のインスタンスすべてを並行合成したプロセスとなる．オブジェクトの ID が Ids で与えられれば，$\mathcal{P}_D^{design}$ は次のように定義できる．

$$\mathcal{P}_D^{design}(Init) = \|_{O \in \mathcal{O}'_D} @ [\Sigma'_D(O)] \; \mathcal{P}_D^{st}(O, Init(O)) \$ \Sigma!'_D(O)$$

$$\mathcal{O}'_D = \{(C, I) \mid \exists T \in tr(D).\ C = class(obj(T)),\ I \in Ids(obj(T))\}$$
$$\Sigma'_D(O) = \{\mathcal{M}(M) \mid \exists T \in tr(D).\ M = mes(T),\ class(O) = class(obj(T))\}$$
$$\Sigma!'_D(O) = \{\mathcal{M}(M) \mid \exists T \in tr(D).\ M = mes(T),\ class(O) = class(sender(M))\}$$

例えば，図 7 から得られる $\mathcal{O}'_D$ は，Ids がすべてのオブジェクトに対して $\{0, 1, 2\}$ を返すとすると次のようになる．

$$\mathcal{O}'_D = \{(User, 0), (User, 1), (User, 2), (Hotel, 0), (Reservation, 0), (Reservation, 1), (Reservation, 2), (Waiting, 0), (Waiting, 1), (Waiting, 2)\}$$

$\mathcal{P}_D^{design}$ はこれら 10 個のオブジェクトに対応するプロセスを並行合成したプロセスである．

4.5 オブジェクトの生成および破棄

図 7 の *Reservation* オブジェクトや *Waiting* オブジェクトのように，オブジェクトはシナリオの途中で生成および破棄することができる．本手法では，生成前および破棄後のオブジェクトは準備状態という特別な状態にあるとみなす．準備状態のプロセスは ⟨⟨*create*⟩⟩ メッセージのみを受信できる．⟨⟨*create*⟩⟩ メッセージを受信した後は通常のオブジェクトのようにふるまう．シーケンス図中の × は準備状態への状態遷移に変換される．

例えば，図 7 の *Reservation* オブジェクトは次のように記述できる．

$$
\begin{aligned}
\{T_{11}, T_{12}, T_{13}, T_{14}\} &\subseteq Transitions \\
T_{11} &= (Reservation_1, r_1, Cancel_1, s) \\
T_{12} &= (Reservation_2, s, Create, i_{10}) \\
T_{13} &= (Reservation_2, i_{10}, Reserve, i_{11}) \\
T_{14} &= (Reservation_2, i_{11}, Reserved_1, r_2)
\end{aligned}
$$

ここで，s は準備状態，i_{10} と i_{11} は中間状態，r_1 と r_2 はそれぞれ *reserved* (*user_id*1) と *reserved*(*user_id*2) を表す状態である．変換された CSP は次のようになる．

$$
\begin{aligned}
\mathcal{P}_D^{st}(Reservation_2, s) &= call\_?create.Hotel.0.Reservation.rsv\_id2 \\
&\quad \rightarrow \mathcal{P}_D^{st}(Reservation_2, i_{10}) \\
\mathcal{P}_D^{st}(Reservation_2, i_{10}) &= call\_?reserve.Hotel.0.Reservation.rsv\_id2.user\_id2 \\
&\quad \rightarrow \mathcal{P}_D^{st}(Reservation_2, i_{11}) \\
\mathcal{P}_D^{st}(Reservation_2, i_{11}) &= call\_!reserved.Reservation.rsv\_id2.Hotel.0.user\_id2 \\
&\quad \rightarrow \mathcal{P}_D^{st}(Reservation_2, r_2) \\
\mathcal{P}_D^{st}(Reservation_2, r_2) &= call\_!cancel.Hotel.0.Reservation.rsv\_id2.user\_id2 \\
&\quad \rightarrow \mathcal{P}_D^{st}(Reservation_2, s)
\end{aligned}
$$

準備状態から *create* イベントを受けて通常の状態に遷移し，*cancel* イベントを受けると準備状態に戻るように変換されていることがわかる．

このようにあらかじめすべてのオブジェクトを準備状態として並行合成する手法では，オブジェクトの最大個数が有限である必要がある．本手法では無限にオブジェクトが生成されるようなモデルを記述することができない．しかし，オブジェクトの数が増えるとモデル検査に必要となる時間が爆発的に増加すること，多くのバグはオブジェクト数が 2 や 3 でも検出できることから，オブジェクトの最大個数を有限に制限しても十分に有益な検証ができると考えられる．

5 検証ツール SDVerifier

5.1 SDVerifier の概要

SDVerifier はモデル検査器 PAT [32] を用いてシーケンス図を検証するためのツールである．このツールは，シーケンス図を PAT ツールに入力可能な CSP モデルに変換する機能と，PAT による解析結果を逆変換してシーケンス図上に表示する機能を持つ．

SDVerifier は Dart 言語で実装され，JavaScript にコンパイルされて Web で公開されている．そのため，HTML5 と CSS3 をサポートする任意のブラウザで実行することができる．SDVerifier と図 4，図 7 に対応するサンプルデータは下記の URL でアクセスできる．

- SDVerifier – `http://dr.asukaze.net/sdverifier/editor.html`
- 図 4 – `http://dr.asukaze.net/sdverifier/demo.cgi?model=cart`
- 図 7 – `http://dr.asukaze.net/sdverifier/demo.cgi?model=hotel`

図 8 は SDVerifier のスクリーンショットである．画面右側にはシーケンス図が表示される．画面左側には 3 つのテキストエリアが表示される．これらは順にシーケンス図のテキスト表現の入力欄，変換された CSP の出力欄，PAT ツールによる解析結果の入力欄である．また，PAT ツールによる解析結果が入力されている際には，画面左下のセレクションボックスにイベントトレースが表示される．開発者がこの中のイベントを選択すると，SDVerifier はそのイベントと対応するメッセージをハイライト表示する．また，右下の表には，それぞれのオブジェクトの状態と送受信可能なメッセージの一覧が表示される．

図 9 は SDVerifier のデータフロー図である．開発者は入力となるシーケンス図設計をテキスト表現 (text input) で入力する．TextParser モジュールはテキスト表現を解析し，シーケンス図のモデル (diagram) をメモリ上に構築する．さらに，得られたシーケンス図のモデルに対して 4.3 節で定義した $Synth_D$ 関数に相当する変換を行い，各オブジェクトのふるまいのモデル (state machine) を構築する．HtmlConverter モジュールは diagram モデルを受け取り，HTML と CSS で記述された図へと変換する．SDVerifier Editor の PatConverter モジュールは diagram モデルおよび state machine モデルを受け取り，PAT ツールに入力可能な CSP を出力する．開発者は出力された CSP を PAT ツールで検査することができ，得られた解析結果 (trace log) は diagram, statemachine と対応させることで HTML 上に図示できる．

5.2 シーケンス図編集機能

検証用のシーケンス図を簡単に作成できるように，SDVerifier はシーケンス図編集機能をツールに統合した．このシーケンス図編集機能では，独自の文法を用い

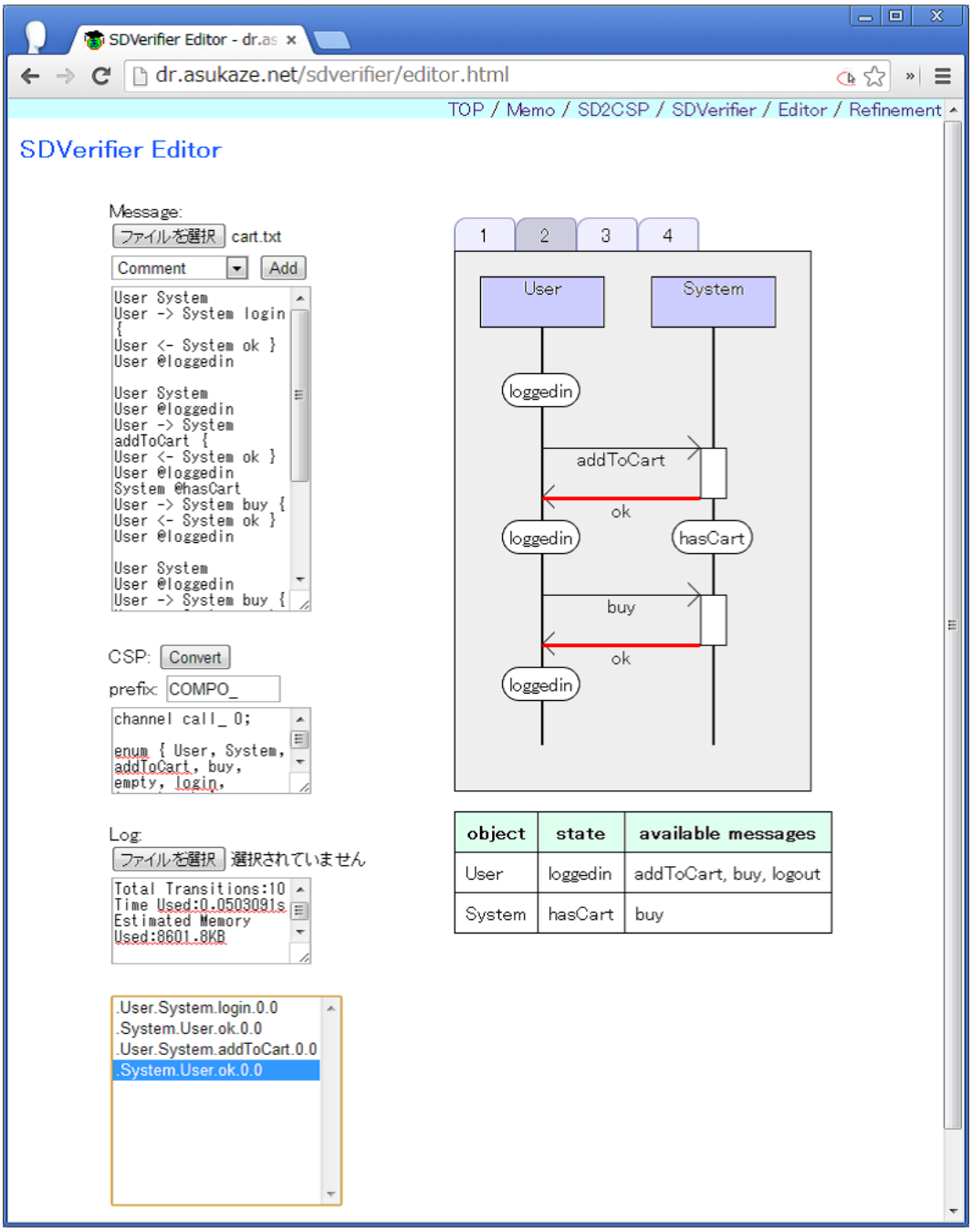


図 8: SDVerifier ツールのスクリーンショット

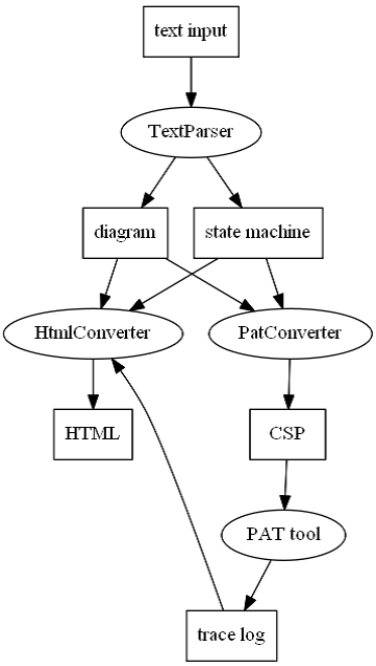


図 9: SDVerifier ツールのデータフロー図

```
User System
User @loggedin
User -> System addToCart {
User <- System ok }
User @loggedin System @hasCart
User -> System buy {
User <- System ok }
User @loggedin
```

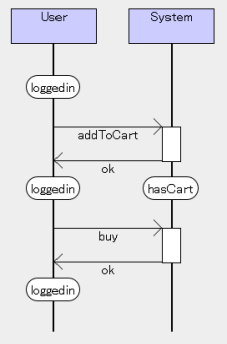


図 10: SDVerifier の記法で記述した図 4b

てシーケンス図を記述する．開発者は画面の左上のテキストエリアにシーケンス図を記述できる．テキストエリアの内容が変更されると，右側のシーケンス図が動的に更新される．

なお，UML のモデルを機械処理可能な形式で記述する場合には XMI 形式が用いられる場合が多い．SDVerifier の記法も XMI 形式もメッセージや状態不変式などの要素を列挙する点は同一であり，これらを相互変換することは容易である．

SDVerifier の記法の例として，図 4b （図 10 右に再掲）は図 10 左のように記述できる．最初の行はこの図に登場するオブジェクトを列挙したものである．残りの行にメッセージと状態不変式を記述する．「`User @loggedin`」は *User* のライフライン上の *loggedin* 状態不変式を表す．これは *User* オブジェクトが *loggedin* 状態にあることを示している．「`User -> System addToCart`」は *User* から *System* に送

信される *addToCart* という名前のメッセージを表す．また,「{」により，受信オブジェクトでこのメッセージから活性区間が開始されることを表している.「User <- System ok」は *System* から *User* に送信される *ok* という名前のメッセージを表す．また,「}」により，送信オブジェクトの活性区間がこのメッセージで終了することを表している.

SDVerifier におけるシーケンス図の詳細な文法を以下に示す．図 11 は 図 7 を SDVerifier の記法で記述したものである.

- ページ：空行で区切られた 1 つのブロックが 1 つのページを表す.
- ページ名:「###」に続けてページのタイトルを記述できる．ページ名は HTML に表示されるが，検証には利用されない．省略した場合は自動的に連番がふられる.
- オブジェクト一覧：ページ名を除いたブロックの最初の行に，このページに登場するオブジェクトの一覧を列挙する．ここで記述された順番にライフラインが描画される．オブジェクトが ID を持つ場合，ID は [] で囲んで記述する.
- メッセージ :「オブジェクト名」「メッセージ種別」「オブジェクト名」「メッセージ名」「活性区間」をスペース区切りで記述する．オブジェクト名の代わりに番号でオブジェクトを指定することもできる．この番号はオブジェクト一覧で記述した順に $0, 1, 2, \ldots$ となる．メッセージ名が「<<creare>>」で始まる場合，メッセージはオブジェクト生成メッセージとなる．メッセージがパラメータを持つ場合，パラメータは () で囲んでカンマ区切りで記述する.
- メッセージ種別 :「=>」「<=」「->」「<-」「-->」「<--」の 6 種類のうち 1 つを指定する.「=>」および「<=」は同期メッセージを,「->」および「<-」は非同期メッセージを,「-->」および「<--」は返信メッセージを表す．矢印の方向により送信オブジェクトと受信オブジェクトが決定される.
- 活性区間：メッセージが活性区間の開始・終了と関連しない場合には省略する．記述する場合は,「{」「}」「}{」「}}」「|}」の 5 種類のうち 1 つを指定する.「{」は受信オブジェクトで活性区間が開始することを表す.「}」は送信オブジェクトで活性区間が終了することを表す.「}{」は送信オブジェクトで活性区間が終了し，受信オブジェクトで活性区間が開始することを表す.「}}」は送信オブジェクト・受信オブジェクトの双方で活性区間が終了することを表す.「|}」は受信オブジェクトで活性区間が終了することを表す.
- 状態不変式 :「オブジェクト名」「@状態名」をスペース区切りで記述する．状態名の後にさらにスペースで区切ってオブジェクト名と状態名を記述することで，複数の状態不変式を 1 行で記述できる．オブジェクト名の代わりに番

```
### CancelNotify
User[user_id1] User[user_id2] Hotel Reservation[rsv_id1]
    Waiting[wait_id] Reservation[rsv_id2]
User @reserved(rsv_id1) 1 @waiting(wait_id2)
    Reservation @reserved(user_id1) Waiting @waiting(user_id2)
User -> Hotel cancel(rsv_id1) {
Hotel -> Reservation cancel(user_id1)
Reservation @/X
Hotel -> User cancelled
Hotel => Waiting getUserId {
Waiting --> Hotel ok(user_id2) }
Hotel -> Waiting cancel(user_id2)
Waiting @/X
Hotel -> 5 <<create>>
Hotel => 5 reserve(user_id2) {
5 --> Hotel reserved(rsv_id2) }
Hotel -> 1 reserved(rsv_id2) }
1 @reserved(rsv_id2) 5 @reserved(user_id2)
```

図 11: SDVerifier の記法で記述した図 7

号でオブジェクトを指定することもできる．状態がパラメータを持つ場合，パラメータは () で囲んでカンマ区切りで記述する．

- オブジェクト破棄 :「オブジェクト名」「@/X」をスペース区切りで記述する．オブジェクト破棄と状態不変式を組み合わせて 1 行で記述することもできる．
- オブジェクトの最大個数 :「#count」「オブジェクト名」「オブジェクトの最大個数」をスペース区切りで指定する．オブジェクトの最大個数を N と指定すると，オブジェクト ID 0 から $N-1$ までの N 個のプロセスを並行合成した CSP が出力される．
- 停止可能状態 :「#end_states」の後に状態名をスペース区切りで列挙する．詳細は 5.3 節で述べる．
- プロセス名プレフィックス :「#prefix」の後にプレフィックスを指定する．ここで指定したプレフィックスは出力される CSP のプロセス名に付加される．
- コメント : 前述したもの以外で「#」から始まる行はコメントとして扱われる．

5.3 CSP 出力機能

SDVerifier は 4 章で定義した $\mathcal{P}_D^{design}$ を出力する．図 12 は図 4 から変換された CSP プロセスである．1 行目の channel call_ 0 はチャネル名である call_ の宣言である．2 行目では，この CSP 内で使われるオブジェクト名およびメッセージ名

```
channel call_ 0;
enum { User, System, addToCart, buy, empty, login, logout, ok };

User_default_() = (
     call_!User.System.login.0.0 -> User_i2_()
);
User_i2_() = (
      call_?System.User.ok.0.0 -> User_loggedin_()
);
User_loggedin_() = (
     call_!User.System.addToCart.0.0 -> User_i4_()
  <> call_!User.System.buy.0.0 -> User_i6_i8_()
  <> call_!User.System.logout.0.0 -> User_i10_()
);
User_i4_() = (
      call_?System.User.ok.0.0 -> User_loggedin_()
);
User_i6_i8_() = (
      call_?System.User.ok.0.0 -> User_loggedin_()
  [*] call_?System.User.empty.0.0 -> User_loggedin_()
);
User_i10_() = (
      call_?System.User.ok.0.0 -> User_default_()
);
System_default_() = (
      call_?User.System.login.0.0 -> System_i1_()
  [*] call_?User.System.addToCart.0.0 -> System_i3_()
  [*] call_?User.System.buy.0.0 -> System_i7_()
  [*] call_?User.System.logout.0.0 -> System_i9_()
);
System_i1_() = (
     call_!System.User.ok.0.0 -> System_default_()
);
System_i3_() = (
     call_!System.User.ok.0.0 -> System_hasCart_()
);
System_i7_() = (
     call_!System.User.empty.0.0 -> System_default_()
);
System_i9_() = (
     call_!System.User.ok.0.0 -> System_default_()
);
System_hasCart_() = (
      call_?User.System.buy.0.0 -> System_i5_()
);
System_i5_() = (
     call_!System.User.ok.0.0 -> System_default_()
);
System_() = User_default_() || System_default_();
#assert System_() deadlockfree;
```

図 12: 図 4 から変換された CSP プロセス

を列挙している．4 行目から 48 行目までがオブジェクトのふるまいを記述した部分である．4 章で定義した $Synth_D(Object, States)$ に対して，Object_States_() という名前のプロセスが生成されている．ここで，default は標準状態を，i_1...i_10 は中間状態を表す．49 行目ではすべてのプロセスを並行合成した $\mathcal{P}_D^{design}(Init)$ を出力している．各オブジェクトの初期状態は標準状態（default）となっている．50 行目では，生成されたプロセスがデッドロックしないことを検証するためのアサーションが生成されている．開発者は，この CSP をそのまま PAT ツールで検証できる．デッドロック以外を検証したい場合には，出力された CSP に編集を加えて検証項目を追加する．メッセージの変換規則に変更が加えられていなければ，SDVerifier は PAT ツールの出力を解析できる．

また，4.5 節でも述べたように，SDVerifier はシナリオ中でのオブジェクトの生成と破棄に対応している．⟨⟨*create*⟩⟩ メッセージやオブジェクト破棄（×）を含むシーケンス図を変換すると，SDVerifier は準備状態のオブジェクトとして CSP のプロセスを生成する．準備状態のプロセスは ⟨⟨*create*⟩⟩ メッセージのみを受信できる．⟨⟨*create*⟩⟩ メッセージを受信した後は通常のオブジェクトのようにふるまう．シーケンス図中の × は準備状態への状態遷移に変換される．

ここで，動的なオブジェクトを検証するためには，オブジェクトの個数の最大値を有限個に制限する必要がある．制限されたモデルでは，実際のシステムでは起こりえないデッドロックが発生する場合がある．例えば，ホテル予約システムは，ユーザの予約リクエストをキャンセル待ちとする場合，*Waiting* オブジェクトを作成する．しかし，もし *Waiting* オブジェクトがすでに最大個数作成されていれば，それ以上 *Waiting* オブジェクトを作成することができず，処理が進められなくなる．PAT ツールはこれをデッドロックとして報告するが，実際にはこれは設計の誤りではない．

SDVerifier では，このような誤判定を抑えるため，「End states」という変換オプションを設けている．開発者は，シーケンス図中の停止可能な状態を「End states」として指定することができる．SDVerifier は指定された状態を変換する際，次のようなプロセスを出力する．

$$P \,\square\, end\_ \rightarrow SKIP$$

ここで，P はシーケンス図から変換された通常のプロセスである．例えば，図 13 は図 7 を含むシーケンス図設計から変換された CSP のプロセスである．Waiting__ が準備状態，Waiting_default_ が標準状態，Waiting_waiting_ が *waiting* 状態，Waiting_i13_, Waiting_i14_, Waiting_i25_が中間状態と対応する．ここでは，*Waiting* オブジェクトの *waiting* 状態および標準状態が「End states」として指定されており，18 行目と 26 行目に「[*] end_ -> Skip」が出力されている．また，準備状態は暗黙的に「End states」として扱われる．そのため，3 行目にも「[*] end_ -> Skip」が出力されている．

並行合成したすべてのプロセスの状態が「End states」に含まれる場合，*end*_ イベントが実行可能となる．*end*_ イベントが実行されるとすべてのプロセスが成功

```
Waiting__(id_) = ((
      call_?Hotel.Waiting.create.0.id_ -> Waiting_i13_(id_)
) [*] end_ -> Skip);

Waiting_i13_(id_) = (
  call_?Hotel.Waiting.register.b1_.0.id_ -> Waiting_i14_(id_, b1_)
);

Waiting_i14_(id_, a1_) = (
  call_!Waiting.Hotel.registered.id_.0
      -> Waiting_waiting_(id_, a1_)
);

Waiting_waiting_(id_, a1_) = ((
      call_?Hotel.Waiting.cancel.a1_.0.id_ -> Waiting__(id_)
  [*] call_?Hotel.Waiting.getUserId.0.id_
          -> Waiting_i25_(id_, a1_)
) [*] end_ -> Skip);

Waiting_i25_(id_, a1_) = (
  call_!Waiting.Hotel.ok.a1_.id_.0 -> Waiting_default_(id_)
);

Waiting_default_(id_) = ((
      call_?Hotel.Waiting.cancel.user_id2.0.id_ -> Waiting__(id_)
) [*] end_ -> Skip);

System_() = User_default_(0) || User_default_(1)
    || User_default_(2) || Hotel_default_() || Reservation__(0)
    || Reservation__(1) || Reservation__(2) || Waiting__(0)
    || Waiting__(1) || Waiting__(2);
```

図 13: 変換された Waiting の CSP プロセス（プロセス数の最大値は 3）

終了を表す *SKIP* 状態となるため，PAT ツールはこれをデッドロックとして報告しない．これによって誤判定を回避し，本来の設計ミスを発見することが可能となる．

5.4 反例解析機能

検証が失敗した場合，PAT ツールは期待しない状態になるまでのイベントトレースを反例として出力する．SDVerifier はこのイベントトレースを逆変換し，元のシーケンス図上で図示できる．この反例解析機能を使うと，イベントトレースのそれぞれのステップで，CSP のイベントに対応するシーケンス図のメッセージをハイライト表示できる．また，それぞれのオブジェクトの状態とその状態から送受信できるメッセージの一覧を表示できる．これらの情報を用いることで，期待しない動作の原因を効率的に調べることができる．

図 8 のスクリーンショットでは，図 12 の CSP をデッドロック検証した結果の反例を表示している．左下のセレクションボックスにはイベントトレースが表示されている．開発者がこの中のイベントを選択すると，SDVerifier はそのイベントと対応するメッセージをハイライト表示する．また，右下の表には，それぞれのオブジェクトの状態と送受信可能なメッセージの一覧が表示される．このスクリーンショットでは，最後の *ok* メッセージが選択されている．*User* は *loggedIn* 状態であり，*buy*，*logout*，*addToCart* のメッセージを送信できる．*System* は *hasCart* 状態であり，*buy* メッセージを受信できる．

この反例解析機能はイベントトレースの実行をエミュレートし，それぞれのオブジェクトの状態を判断している．SDVerifier はシーケンス図を CSP に変換する際，送信オブジェクトのクラス名とプロセス ID，受信オブジェクトのクラス名とプロセス ID，メッセージ名とパラメータを出力するため，隠蔽されたイベントがなければオブジェクトのふるまいを完全にエミュレートでき，オブジェクトの状態を正確に出力できる．隠蔽されたイベントがある場合，「state」と「available messages」は観測できるイベントのみから推測されたものとなる．

6 ケーススタディ

SDVerifier ツールの有効性を確認するため，いくつかのケーススタディを行った．6.1 節では EC サイトのカスタマーサポートシステムの検証について，6.2 節ではエレベータコントロールシステムの検証について，6.3 節では銀行システムの検証について述べる．EC サイトは Web システムであり，ユーザのリクエストをトリガとしてサーバで処理が行われる．エレベータコントロールシステムは組み込みシステムであり，ユーザのリクエストおよび複数のセンサのイベントをトリガと

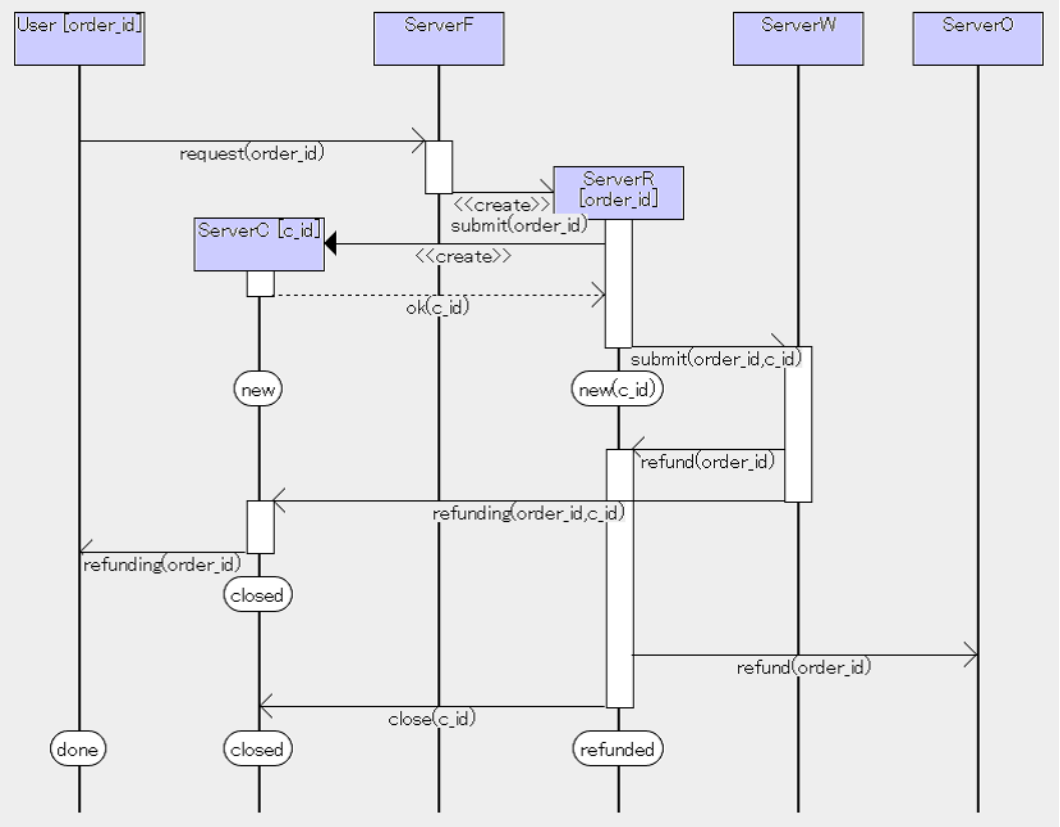


図 14: ケーススタディで用いたシーケンス図の一部

して処理が行われる．また，銀行システムは 2 つのサブシステムから構成されるシステムである．

さらに，SDVerifier のスケーラビリティを調べるために，より大きな入力をランダムに生成して与えてシーケンス図から CSP への変換時間を計測した．この実験については 6.4 節で述べる．

6.1 カスタマーサポートシステム

1 つめのケーススタディとして，EC サイトのカスタマーサポートシステムの設計を検証した．このシステムはある企業で実際に運用されているものである．検証対象のシーケンス図は抽象モデル 4 枚，詳細モデル 8 枚の合計 12 枚を用意した．図 14 は詳細モデルのシーケンス図の中のひとつである．このシーケンス図を検証することで，モデルにデッドロックがないことおよびユーザから見て抽象モデルと詳細モデルが同じふるまいをすることを確認できた．検証には SDVerifier と PAT ツールのバージョン 3.2.1 を用いた．

表 1 は検証対象のモデルのサイズと検証にかかった時間をまとめたものである．中央 4 列は検証対象のシーケンス図のページ数，定義されたクラス数，1 クラス

表 1: ケーススタディの結果（実験環境 : 2.8 GHz Intel Core i7-2640M，8 GB RAM）

	シーケンス図				CSP	
	ページ数	クラス数	オブジェクト数	状態数	状態数	検証時間
抽象モデル デッドロック検証	4	3	5	12	263	0.026s
詳細モデル デッドロック検証	8	7	10	44	6481	0.337s
抽象・詳細モデル 詳細化検証	-	-	-	-	20691	0.575s
不正モデル デッドロック検証	8	7	10	44	-	0.006s

に対して生成されるオブジェクトの最大個数を 3 とした場合の最大オブジェクト数，状態不変式で定義された状態の数を表している．右側 2 列は PAT ツールの出力であり，全オブジェクトの状態を組み合わせたシステム全体の状態数および検証にかかった時間を表している．

検証対象の抽象モデルはユーザおよびユーザと直接メッセージを交換するフロントエンドのオブジェクトのみを記述したシーケンス図であり，詳細モデルはフロントエンドから処理を移譲されるバックエンドのオブジェクトを含めて記述したシーケンス図である．2 種類のモデルでそれぞれデッドロックが発生しないことと，2 つのモデルがユーザの視点で同じふるまいをすることを検証し，すべて 1 秒以下で検証を完了することができた．なお，2 種類のモデルが同じふるまいをすることは，次のような検証項目で検証した．

$$(AbstractModel \setminus X) \quad =_F \quad (ConcreteModel \setminus X)$$

ここで，*AbstractModel* は抽象モデルから変換された CSP モデル，*Concrete Model* は詳細モデルから変換された CSP モデル，X は 2 つのモデルで共有されていないイベントの集合である．このケーススタディではユーザが直接入出力をするメッセージは抽象モデルと詳細モデルで共通していたため，この検証項目によりそれらのメッセージのふるまいが変化していないことを検証できた．ここで，この検証項目では隠蔽が使われているため，ライブロックが起こる可能性がある．ライブロックが発生するモデルは CSP の失敗モデルでは正しく検証することができないため，$(AbstractModel \setminus X)$ および $(ConcreteModel \setminus X)$ がライブロックを含まないことも別途確認した．

さらに，SDVerifier でエラーが発見できることを確認するため，図 14 で $ServerW$ が送信する $refunding$ と $refund$ の 2 つのメッセージの順序を入れ替えて意図的に不正なシーケンス図を作成した．この不正なシーケンス図に対しても，1 秒以内で

この問題を検出することができた．これにより，シーケンス図の問題点が期待通りに検出されており，反例解析機能を用いて効率的に原因を調査できることが確認できた．

このケーススタディにおける詳細化の検証は，CSP の詳細化関係ではなく，2 つのモデルの等価関係を検証した．これは，今回検証対象としたシーケンス図の詳細化ではバックエンドのオブジェクトの数は増加したが非決定性の情報が変化しなかったからである．今後この設計がステートマシン図やプログラミング言語のソースコードとして詳細化されていく過程では，シーケンス図において非決定的であった選択が決定的な選択に置き換えられていくことが想定できる．そのようなモデルが与えられれば，CSP の詳細化関係を検証することで，詳細化の妥当性が確認できる．

このケーススタディの検証対象であるシーケンス図では，同一クラスの複数インスタンス，メッセージおよび状態のパラメータ，オブジェクトの生成が使用されている．ケーススタディを通して，SDVerifier がこれらを適切に表現し，実用的な時間で解析できることを確認できた．

6.2 エレベータコントロールシステム

2 つめのケーススタディとして，エレベータコントロールシステムのふるまいを検証した．ここでは，参考文献 [12] の書籍中で紹介されているケーススタディを例題として用いた．これは組み込みソフトウェアのシステムであり，イベントの発火点がユーザだけではなく複数のセンサを含んでいる点がカスタマーサポートシステムのケーススタディと異なる．なお，この書籍ではシーケンス図ではなくコミュニケーション図が使用されているが，UML の仕様上シーケンス図とコミュニケーション図は同じ概念を表す図であり，相互に変換が可能である．図 15，図 16，図 17，図 18 は入力としたシーケンス図である．シーケンス図は 4 枚であり，20 個のオブジェクトが記述されている．

このケーススタディのデータは下記の URL でアクセスできる．

- `http://dr.asukaze.net/sdverifier/demo.cgi?model=elevator`

このモデルを検証すると，ElevatorManager オブジェクトと Scheduler オブジェクト間のコミュニケーションでデッドロックが検出された．図 15 では ElevatorButtonRequest をトリガとして，ElevatorManager から Scheduler に ElevatorCommitment メッセージが送信される．一方，図 16 では FloorButtonRequest をトリガとして，Scheduler から ElevatorManager に SchedulerRequest メッセージが送信される．もしこの 2 つのイベントがほぼ同時に起こった場合，ElevatorManager と Scheduler がお互いに相手がメッセージを受信するのを待ち続け，デッドロックとなる．この問題は，例えば ElevatorManager や Scheduler におけるメッセージ受信に十分なサイズのメッセージキューを用いるといった方法で回避する必要がある．

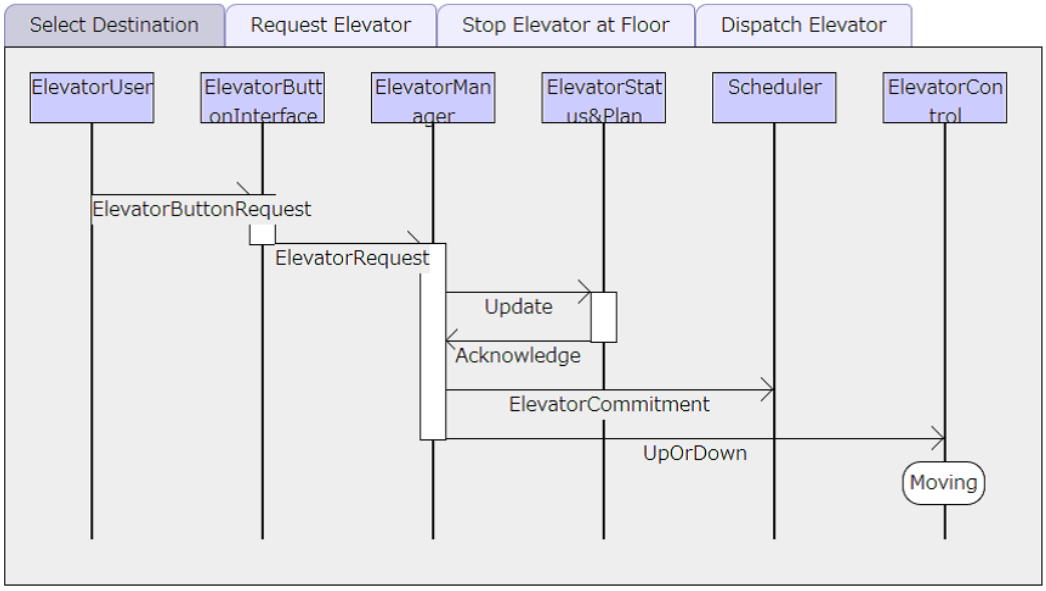


図 15: エレベータコントロールシステムのシーケンス図 (Select Destination)

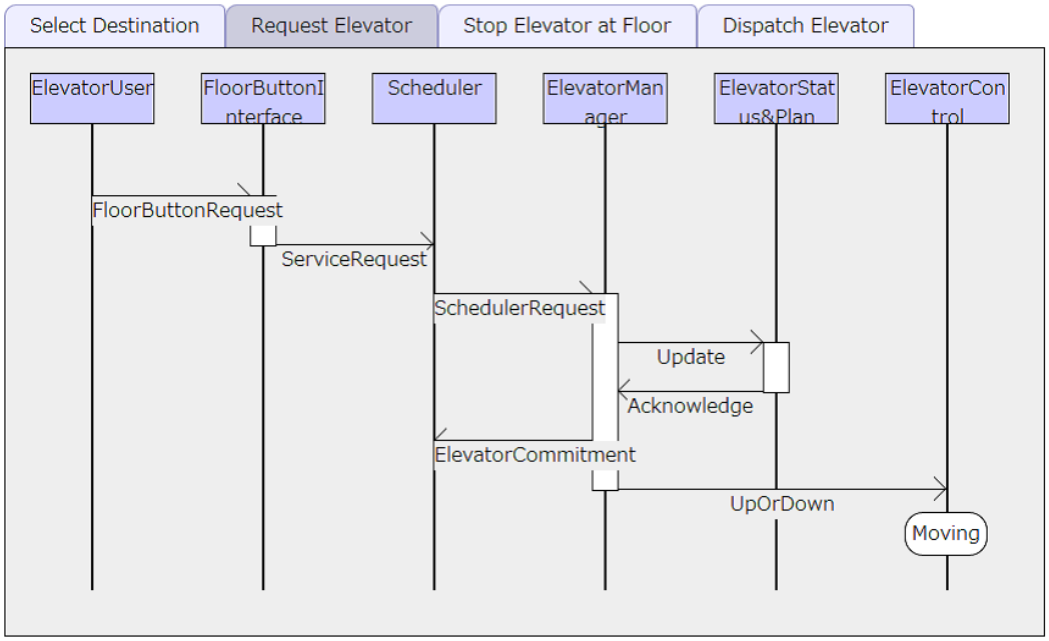


図 16: エレベータコントロールシステムのシーケンス図 (Request Elevator)

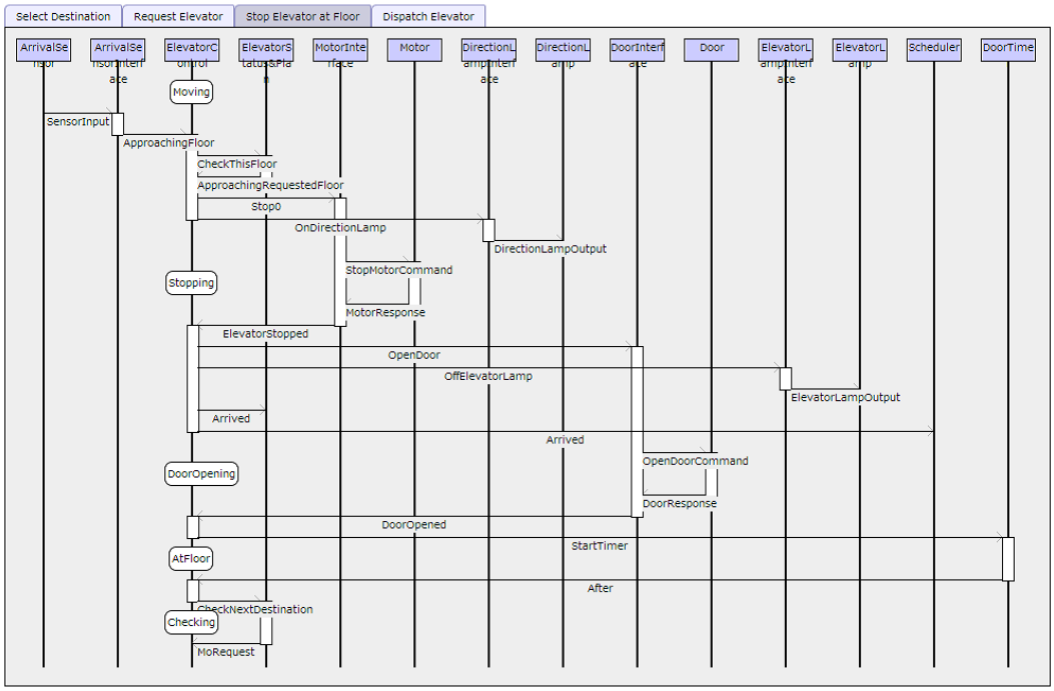


図 17: エレベータコントロールシステムのシーケンス図 (Stop Elevator at Floor)

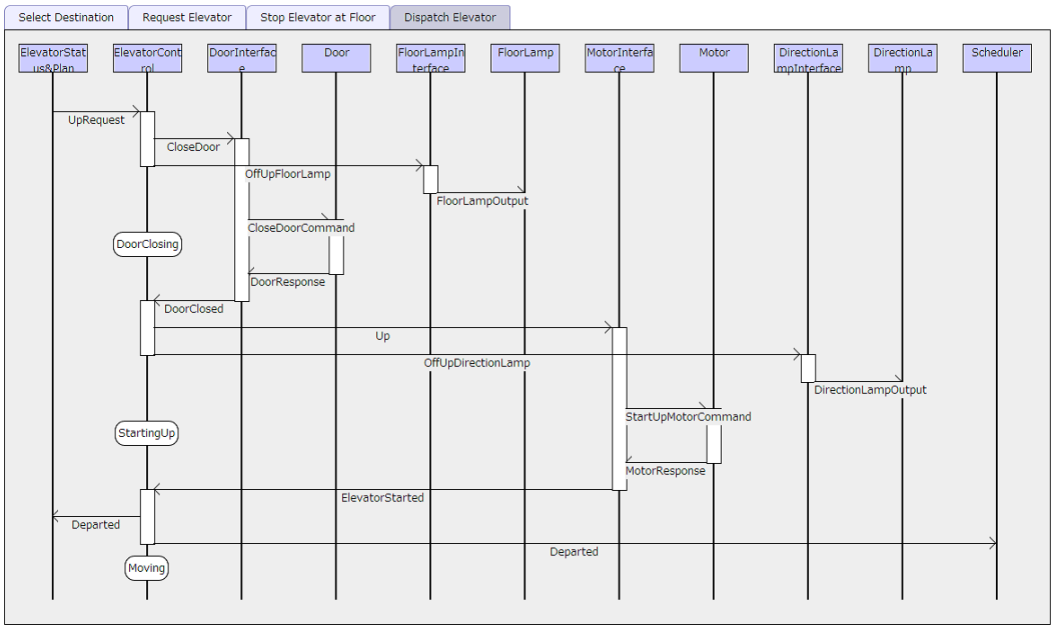


図 18: エレベータコントロールシステムのシーケンス図 (Dispatch Elevator)

このケーススタディによって，SDVerifier がエレベータコントロールシステムのモデルを適切に表現し，問題点を発見できることを確認できた．

6.3 銀行システム

3 つめのケーススタディとして，同じく参考文献 [12] の書籍中で紹介されているケーススタディから，銀行システムのふるまいを検証した．このシステムは BankServer と ATMClient という 2 つのサブシステムから構成されている．設計中で BankServer と ATMClient との接続関係は自然言語のみで書かれているため，2 つのサブシステムを組み合わせた全体の検証は対象外とした．図 19，図 20，図 21，図 22 は入力としたシーケンス図である．シーケンス図は 4 枚であり，16 個のオブジェクトが記述されている．

SDVerifier でこのモデルのデッドロックを検証したところ，BankServer と ATM-Client それぞれのサブシステムで不整合がないことを確認できた．

このケーススタディのデータは下記の URL でアクセスできる．

- `http://dr.asukaze.net/sdverifier/demo.cgi?model=banking`

この入力としたモデルでは，図 19 および図 20 中に BankServer というライフラインが，図 21 および図 22 中に ATMClient というライフラインが存在している．これらは特定のオブジェクトではなくサブシステムを表すライフラインである．今回検証の対象外とした BankServer と ATMClient との接続関係については、シーケンス図を書き換えてこれらのライフラインをサブシステム内のオブジェクトを特定するような記述に変更すれば検証可能であると考えられる．また，サブシステムの情報を別途入力として与えることでこの書き換えを自動的に行えるようにすることは今後の課題の 1 つであるといえる．

6.4 スケーラビリティ

最後に，SDVerifier のスケーラビリティを調べるため，より大きな入力を与えてシーケンス図から CSP への変換時間を計測した．図 23 は計測結果をグラフ化したものである．

図 23a はオブジェクト数を変化させ，各オブジェクトが他のオブジェクトに 1 回ずつメッセージを送信するシーケンス図をランダムに生成したものである．各オブジェクトはメッセージ送受信後に状態が変わるものとして，状態不変式もランダムに生成した．ここでは，状態不変式内に記述する状態名を 100 種類に固定している．また，図 23b はオブジェクト数を 100 に固定し，シーケンス図に記述されるメッセージ数を変化させてランダムにシーケンス図を生成したものである．これらのグラフを見ると，変換時間はおおよそオブジェクト数・メッセージ数に

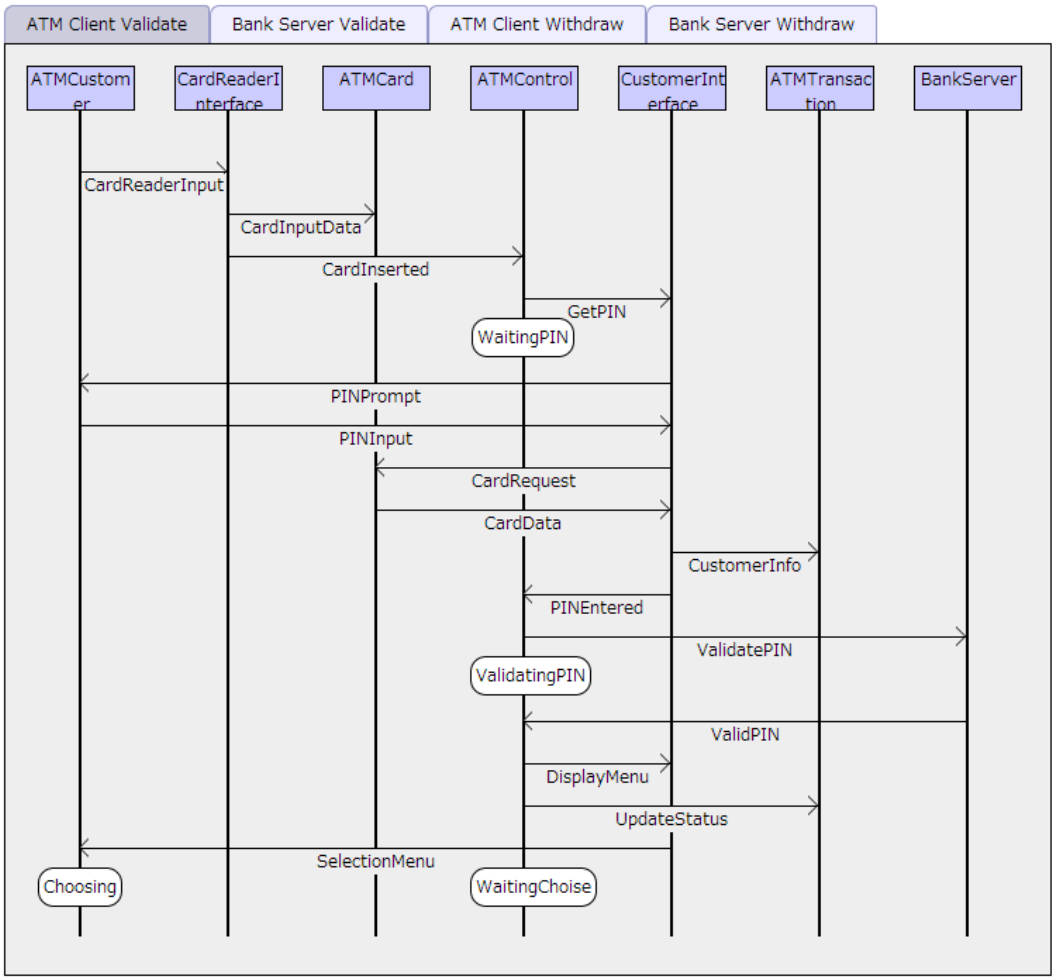


図 19: 銀行システムのシーケンス図 (ATM Client Validate)

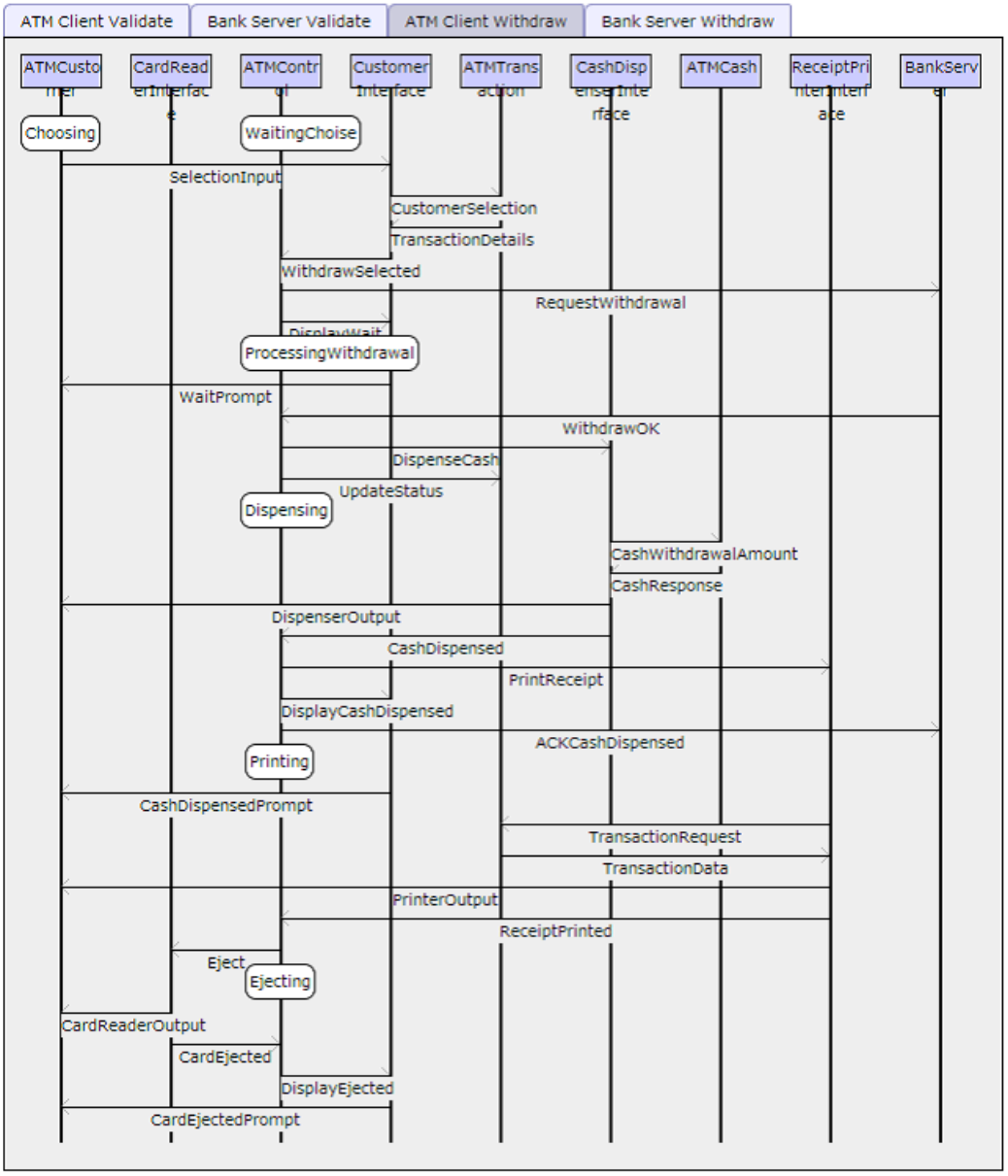


図 20: 銀行システムのシーケンス図 (ATM Client Withdraw)

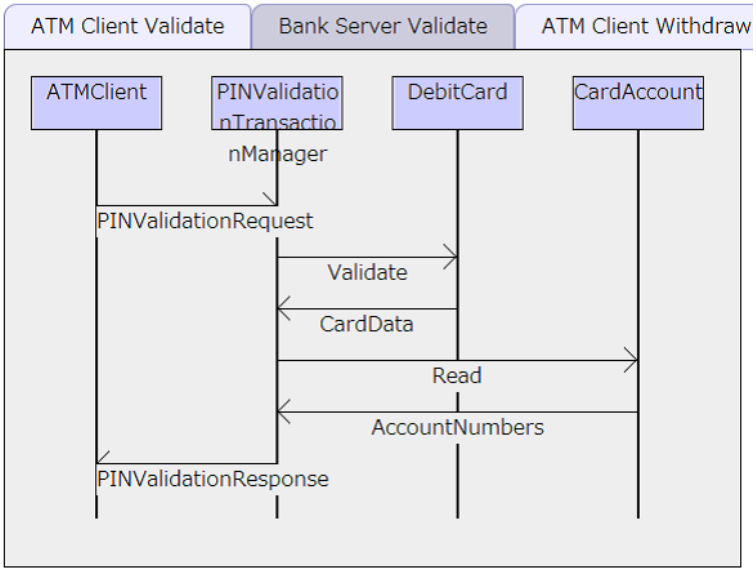


図 21: 銀行システムのシーケンス図 (Bank Server Validate)

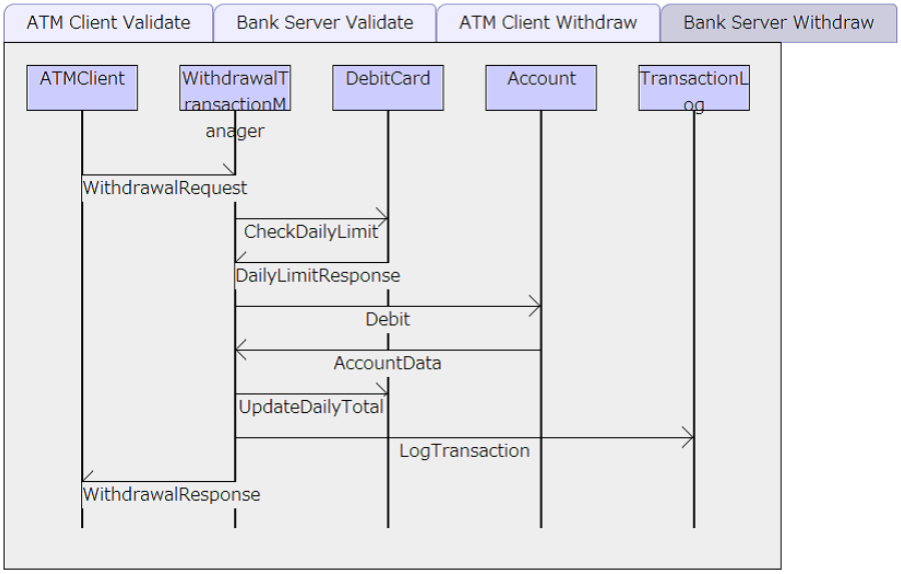
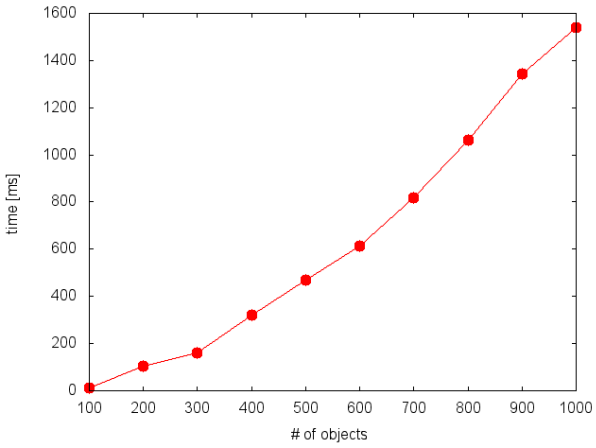
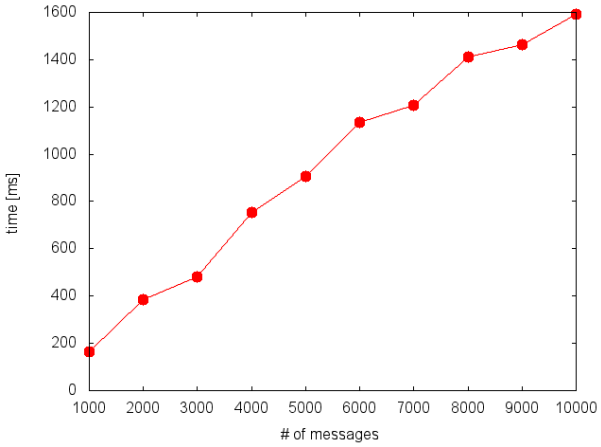


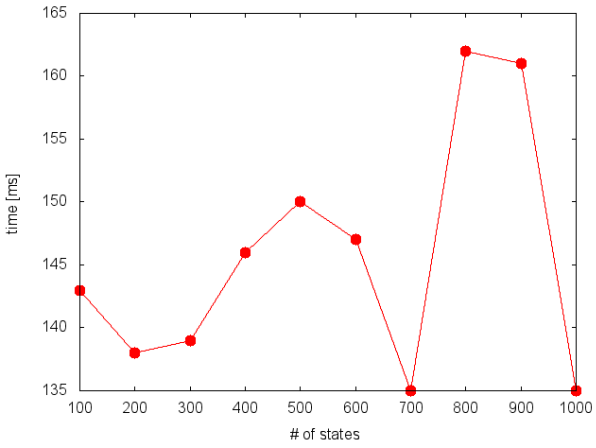
図 22: 銀行システムのシーケンス図 (Bank Server Withdraw)



(a)



(b)



(c)

図 23: シーケンス図の変換時間（実験環境：1.8 GHz Intel Core i5-3337U，4 GB RAM）

比例していることがわかる．オブジェクト数を変化させた実験では 700 オブジェクトまで，メッセージ数を変化させた実験では 5000 メッセージまでのシーケンス図を 1 秒以内で変換できており，実用上十分なサイズのシーケンス図を高速に変換可能であるといえる．

一方，図 23c はオブジェクト数を 100，メッセージ数を 1000 に固定し，状態不変式として記述する状態の数を変化させたものである．この実験結果では，すべての場合で 200 ミリ秒以内で変換が完了しており，状態数と変換時間の間に相関はみられない．これは，SDVerifier はシーケンス図に記述された状態を逐次変換するのではなく，初期状態から遷移可能な状態のみを変換する仕様となっているためである．4.3 節で述べたように最悪計算量は $O(2^N)$ となるが，実際には初期状態から遷移可能な状態の集合は小さく，メッセージごとの処理時間と比べて無視できるものとなっている．

7 関連研究

UML やシナリオベースのモデルの検証に関してはすでにいくつかの研究が存在する．

I. Abdelhalim らは UML のステートマシン図と fUML のアクティビティ図を対象として CSP を用いて整合性を検証する手法を提案している [1]．ステートマシン図やアクティビティ図ではシステムのふるまいが状態ベースのモデルとして記述されるため，対象となる設計が十分に詳細に記述されていれば，その図の整合性を検証したり動作可能なプログラムを自動生成したりすることが可能となる．本論文の提案手法は，シナリオベースの設計を合成することでそのような状態ベースのモデルを得ることを目的としている．

L. Lu らは UML のシーケンス図の意味を明確にし，詳細化の検証をする手法を提案している [23]．L. Dan らは UML のシーケンス図を CSP に変換する手法を，T. Ziadi らは UML のシーケンス図を状態遷移図に変換する手法を提案している [9, 35]．これらの手法では，UML 2 で導入された結合フラグメントを用い，「opt」「alt」「loop」といった演算子をもとに実行可能なトレースを定め，検証を可能としている．複数のシナリオを合成したい場合には，シーケンス図の相互作用使用を用いて合成方法を手作業で指定し，1 つの大きなシーケンス図にまとめたものを入力とする必要がある．本論文の提案手法は，複数のシナリオを入力として与え，それらを合成して状態モデルを得ることを目的としていることがこれらの手法とは異なる．結合フラグメントを用いたシーケンス図が複数枚ある場合，L. Dan らの手法を用いて個々のシーケンス図を CSP に変換し，提案手法を用いてそれらを合成してシステム全体のふるまいを得ることが可能であると考えられる．

シナリオベースの設計から状態ベースのモデルを合成する手法としては，古くは 1970 年代から研究が始まっている．A. W. Biermann らはサンプルの実行ステップを元にプログラムを自動生成する手法を提案している [5]. この手法では，各ス

テップで実行された操作と満たされるべき条件を元に状態モデルを生成する．入力の実行ステップのうち，同じ操作をする複数のステップは 1 つの状態にまとめられる可能性があるが，それらの状態から同じ条件下で異なる状態に遷移する場合には異なる状態として出力される．この手法では，元の実行ステップを実行可能な決定的な状態モデルのうち状態数が最小のものを出力できる．この変換は生成される状態モデルが決定的であることを前提としているため，非決定的にいずれかのメッセージが選択されるようなモデルは記述できない．

シーケンス図やメッセージシーケンスチャートを対象とした状態モデルの合成に関しても，1990 年代後半から多くの研究が存在する [22]．具体的な研究の 1 つとして，E. Mäkinen らは「MAS」という名前のツールを提案している [24]．この手法では，メッセージ送信を状態に，メッセージ受信を状態間の遷移に対応させることでシーケンス図から状態ベースのモデルを得る．この手法では状態不変式などの情報を付加することなく状態ベースのモデルを生成できる点が優れている．この手法では A. W. Biermann らの手法と同様に，生成されるモデルが決定的であることを前提として最小の状態数のモデルを出力する．常に決定的なモデルが生成されるため，非決定的にいずれかのメッセージが選択されるようなモデルを記述できない．シーケンス図は開発の初期段階で用いられる図であるため，シーケンス図の範囲では記述されない要因による非決定性は重要である．本論文の提案手法ではメッセージ送信に非決定性を持たせることで，この問題を解決している．

また，R. Alur らはメッセージシーケンスチャートから有限状態の IO オートマトンを生成する手法を提案している [3]．この手法では，メッセージシーケンスチャートで記述された複数のシナリオから不完全な IO オートマトンを生成し，そこからメッセージシーケンスチャート中のラベルを用いて同一の状態をまとめ，最後に欠けている遷移を加えて完全な IO オートマトンを得る．ここで，欠けている遷移を加える方法として，その制約を外部環境のオートマトンとして与えることで，合成結果が決定的かつデッドロックしない遷移を探すという手法をとっている．これにより，システムの制約が明確である場合には，すべての遷移をメッセージシーケンスチャートで与えるよりも簡潔に記述できる．一方，この手法も合成後のモデルが決定的であることを前提としているため，本論文の提案手法のように非決定性を扱うことはできない．

メッセージが送信される場合と送信されない場合が確定していないモデルを扱う方法として，S. Uchitel らはシナリオを Modal Transition System (MTS) [21] へ変換する手法を提案している [33]．MTS は maybe 遷移をもつ状態遷移モデルである．彼らの手法では，FLTL [11] を 3 値に拡張した Three-Valued FLTL を用いる．Three-Valued FLTL では，t (true) と f (false) の他に $\perp$ (maybe, unknown) という値が使われる．この手法では,「rep」や「alt」などを利用したシナリオと FLTL による制約のそれぞれを状態モデルに変換し，その共通部分を合成結果として出力する．I. Krka らは MTS の詳細化関係を検証する手法を提案している [18].

シーケンス図に付加的な情報を加えることで検証やコード生成を可能とする手法

としては,「Live Sequence Chart」や「HMSC」という図を用いる手法がある．Live Sequence Chart を用いる手法は D. Harel らによって提案されている [8, 13, 14, 15, 16]. Live Sequence Chart はメッセージシーケンスチャートを拡張した図であり，特定の条件でトリガされるシナリオを記述できる．また，必須のメッセージ，許可されるメッセージ，許可されないメッセージを区別して記述することで柔軟な設計を可能としている．明示的にこのような情報を記述することで，各シナリオの意味が明確化され，状態ベースのモデルへの変換が可能となる．Y. Bontemps らや H. Kugler ら，G. Sibay らも Live Sequence Chart を状態モデルに変換する手法を提案している [6, 7, 19, 29]．また，J. Sun らは Live Sequence Chart を CSP に変換することで検証する手法を提案している [31]．R. Kumar らは Live Sequence Chart をオートマトンに変換することでコミュニケーション・プロトコルを自動的に検証する手法を提案している [20]．G. Sibay らは Live Sequence Chart を改善した記述言語である Triggered Scenario Specification Language を提案している [30]．HMSC を用いる手法は R. Alur らによって提案されている [4]．HMSC はシーケンス図間の実行順序を示す図である．LTSA モデル検査器のプラグインである Message Sequence Chart plugin (LTSA-MSC) は HMSC を用いてシーケンス図を検証する [34].

Live Sequence Chart や HMSC を用いる手法では,「ログイン成功メッセージもしくはログイン失敗メッセージを返す」のように実際に送受信されるメッセージが確定してない設計を表すことができる．しかし，これらの手法では送信オブジェクトと受信オブジェクトの違いが考慮されていなかった．そのため，送信オブジェクトがログイン成功メッセージもしくはログイン失敗メッセージを返すにもかかわらず受信オブジェクトがログイン成功メッセージしか考慮していなかった場合，ログイン失敗メッセージは暗黙的に無視され，常にログイン成功メッセージを返すシステムと同一視されてしまっていた．本論文の提案手法では，送信オブジェクトの選択に CSP の内部選択を用いることで，非決定的な選択が送信オブジェクトによってなされることを明確にモデル化し，検証できるようになった．

8 まとめ

本論文では，上流設計におけるシーケンス図を検証するため、複数のシーケンス図から状態モデルを合成する手法を提案した．提案手法は設計上流における非決定性を考慮しており、非決定的な選択の主体を明確化したモデルを合成できる．合成方法を厳密に議論するため、シーケンス図の形式的な定義からプロセスの形式記述まで CSP を使用し、合成に適した CSP 演算子 ∘ と $ を定義し，その性質を数学的に明らかにした上で合成アルゴリズムを構築した．また，∘ や $ を用いて表現されたプロセスをそれらを含まない通常の CSP のプロセスに変換する手法を説明し，既存の CSP ツールを活用した検証を可能とした．

提案手法は SDVerifier ツールとして実装され，ケーススタディにより有効性が確認された．このツールはシーケンス図を CSP に変換することで，モデル検査器

PAT で検証可能とする．また，このツールは同一クラスに複数のインスタンスが存在でき，オブジェクトをシナリオ中で動的に生成・破棄できるようなシーケンス図を検証できる．さらに，PAT の出力する反例をシーケンス図上に逆変換することで効率的に反例を解析できる．EC サイト、エレベータ、銀行システムの設計に対してケーススタディを行い，複雑なシステムの設計において提案手法およびツールが有用であることを確認できた．

今後の課題としては，現状ではシーケンス図を検証可能とするために自明なシナリオも含めて網羅的にシーケンス図を記述しなければならないため，これを軽減するような改善が必要であると考えている．具体的には，以下のような改善が考えられる．

- 異常系のシナリオはテンプレートから自動生成可能とする．
- オブジェクトのおおまかな状態遷移設計などを別途与えることで，状態不変式の記述を省略可能とする．

参考文献

[1] Islam Abdelhalim, Steve Schneider, and Helen Treharne. Towards a practical approach to check uml/fuml models consistency using csp. *Proceeding ICFEM'11 Proceedings of the 13th international conference on Formal methods and software engineering*, pages 33–48, 2011.

[2] Allon Adir, Eli Almog, Laurent Fournier, Eitan Marcus, Michal Rimon, Michael Vinov, and Avi Ziv. Genesys-pro: Innovations in test program generation for functional processor verification. *Design & Test of Computers, IEEE*, 21(2):84–93, 2004.

[3] Rajeev Alur, Milo Martin, Mukund Raghothaman, Christos Stergiou, Stavros Tripakis, and Abhishek Udupa. Synthesizing finite-state protocols from scenarios and requirements. *arXiv preprint arXiv:1402.7150*, 2014.

[4] Rajeev Alur and Mihalis Yannakakis. Model checking of message sequence charts. *Proceedings of the 10th International Conference on Concurrency Theory*, 1999.

[5] Alan W Biermann and Ramachandran Krishnaswamy. Constructing programs from example computations. *IEEE Trans. Softw. Eng.*, vol. SE-2(no. 3):141–153, 1976.

[6] Yves Bontemps, Patrick Heymans, and P Schobbens. From live sequence charts to state machines and back: A guided tour. *Software Engineering, IEEE Transactions on*, 31(12):999–1014, 2005.

[7] Yves Bontemps, Pierre-Yves Schobbens, and Christof Löding. Synthesis of open reactive systems from scenario-based specifications. *Fundamenta Informaticae*, 62(2):139–169, 2004.

[8] Werner Damm and David Harel. Lscs: Breathing life into message sequence charts. *Formal methods in system design*, 19(1):45–80, 2001.

[9] Li Dan and Li Danning. An approach to formalize uml sequence diagrams in csp. *3rd International Conference on Computer and Electrical Engineering*, 2010.

[10] Formal Systems. FDR2, 2010. http://www.fsel.com/software.html.

[11] Dimitra Giannakopoulou and Jeff Magee. Fluent model checking for event-based systems. In *ACM SIGSOFT Software Engineering Notes*, volume 28, pages 257–266. ACM, 2003.

[12] Hassan Gomaa. *Designing concurrent, distributed, and real-time applications with UML*. Addison-Wesley Professional, 2000.

[13] David Harel and Hillel Kugler. Synthesizing state-based object systems from lsc specifications. *International Journal of Foundations of Computer Science*, 2002.

[14] David Harel and Hillel Kugler. Synthesizing state-based object systems from lsc specifications. *International Journal of Foundations of Computer Science*, 13(01):5–51, 2002.

[15] David Harel, Assaf Marron, and Gera Weiss. Behavioral programming. *Communications of the ACM*, 55(7):90–100, 2012.

[16] David Harel and Itai Segall. Synthesis from scenario-based specifications. *Journal of Computer and System Sciences*, 78(3):970–980, 2012.

[17] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[18] Ivo Krka and Nenad Medvidovic. Distributing refinements of a system-level partial behavior model. In *Requirements Engineering Conference (RE), 2013 21st IEEE International*, pages 72–81. IEEE, 2013.

[19] Hillel Kugler and Itai Segall. Compositional synthesis of reactive systems from live sequence chart specifications. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 77–91. Springer, 2009.

[20] Rahul Kumar and Eric G Mercer. Verifying communication protocols using live sequence chart specifications. *Proceedings of the 8th International Workshop on Automated Verification of Critical Systems*, 2008.

[21] Kim G Larsen and Bent Thomsen. A modal process logic. In *Logic in Computer Science, 1988. LICS'88., Proceedings of the Third Annual Symposium on*, pages 203–210. IEEE, 1988.

[22] Hongzhi Liang, Juergen Dingel, and Zinovy Diskin. A comparative survey of scenario-based to state-based model synthesis approaches. In *Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools*, pages 5–12. ACM, 2006.

[23] Lunjin Lu and Dae-Kyoo Kim. Required behavior of sequence diagrams: Semantics and refinement. In *Engineering of Complex Computer Systems (ICECCS), 2011 16th IEEE International Conference on*, pages 127–136. IEEE, 2011.

[24] Erkki Mäkinen and Tarja Systä. Mas — an interactive synthesizer to support behavioral modelling in uml. *Proceedings of the 23rd International Conference on Software Engineering*, 2001.

[25] Robin Milner. *A calculus of communicating systems*. Springer-Verlag New York, Inc., 1982.

[26] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure Version 2.4.1, 2011. http://www.omg.org/spec/UML/2.4.1/.

[27] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

[28] Steve Schneider. Verifying authentication protocols in csp. *Software Engineering, IEEE Transactions on*, 24(9):741–758, 1998.

[29] German Sibay, Sebastian Uchitel, and Victor Braberman. Existential live sequence charts revisited. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 41–50. IEEE, 2008.

[30] German Emir Sibay, Victor Braberman, Sebastian Uchitel, and Jeff Kramer. Synthesizing modal transition systems from triggered scenarios. *Software Engineering, IEEE Transactions on*, 39(7):975–1001, 2013.

[31] Jun Sun and Jin Song Dong. Model checking live sequence charts. In *Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings. 10th IEEE International Conference on*, pages 529–538. IEEE, 2005.

[32] Jun Sun, Yang Liu, and Jin Song Dong. Model checking csp revisited: Introducing a process analysis toolkit. pages 307–322, 2009.

[33] Sebastian Uchitel, Greg Brunet, and Marsha Chechik. Synthesis of partial behavior models from properties and scenarios. *Software Engineering, IEEE Transactions on*, 35(3):384–406, 2009.

[34] Sebastian Uchitel, Robert Chatley, Jeff Kramer, and Jeff 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.

[35] Tewfic Ziadi, Loic Helouet, and Jean-Marc Jezequel. Revisiting statechart synthesis with an algebraic approach. In *Proceedings of the 26th International Conference on Software Engineering*, pages 242–251. IEEE Computer Society, 2004.

A CSP の演算子の定義

3.1 節で述べたように，本論文では CSP の演算子として「→」「□」「⊓」「||」「\」を用いる．これらの演算子は，3.2 節で説明したトレース集合 $traces(P)$ および失敗集合 $failures(P)$ で定義することができる．ここでは，それぞれの演算子の定義を説明する．

- プレフィックス：$a \rightarrow P$ はイベント a を実行し，その後プロセス P のようにふるまうプロセスである．

$$\begin{aligned} traces(a \rightarrow P) &= \{\langle\rangle\} \cup \{\langle a\rangle \hat{} s \mid s \in traces(P)\} \\ failures(a \rightarrow P) &= \{(\langle\rangle, X) | a \notin X\} \cup \{(\langle a\rangle \hat{} s, X) \mid (s, X) \in failures(P)\} \end{aligned}$$

  トレース集合は，$traces(P)$ に含まれる各トレースの前にイベント a を実行するものとなっている．失敗集合は，トレースが空集合の場合には a 以外のすべてのイベントが拒否され，イベント a 実行後は $failures(P)$ と同様にふるまうものとなっている．

- 内部選択：$P \sqcap Q$ はプロセス P もしくは Q のようにふるまうプロセスである．P もしくは Q はプロセス内部で非決定的に選択され，プロセスの外部からはコントロールできない．

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

  内部選択では，トレース集合・失敗集合とも，P および Q のトレース集合・失敗集合の和集合となる．P もしくは Q のどちらかで拒否されるイベントは $P \sqcap Q$ でも拒否される可能性がある．

- 外部選択：$P \,\square\, Q$ はプロセス P もしくは Q のようにふるまうプロセスである．P と Q のどちらが選択されるかは次に実行されるイベントによって定まる．そのため，この選択は他のプロセスや外部環境など，プロセスの外部からコントロールできる．

$$\begin{aligned} traces(P \,\square\, Q) &= traces(P) \cup traces(Q) \\ failures(P \,\square\, Q) &= \{(\langle\rangle, X) \mid (\langle\rangle, X) \in failures(P) \cap failures(Q)\} \\ &\quad \cup \{(s, X) \mid s \neq \langle\rangle, (s, X) \in failures(P) \cup failures(Q)\} \end{aligned}$$

  外部選択では，トレースが空集合の場合とそれ以外の場合で失敗集合の式が異なる．トレースが空集合の場合，つまり最初のイベントでは，P および Q でともに拒否されるイベントのみが拒否される．例えば，最初のイベントが P では拒否されるが Q では拒否されない場合，そのイベントによって Q が

選択される．この場合，プロセスは Q のようにふるまうので，そのイベントは拒否されない．P もしくは Q は最初のイベントで選択されるため，トレースが空集合でない場合，失敗集合は内部選択と同様に P の失敗集合と Q の失敗集合の和集合となる．

- 並行合成：$P_X||_Y Q$ はプロセス P と Q が並行に動作するプロセスを表す．プロセス P はイベント集合 $(X-Y)$ を，プロセス Q はイベント集合 $(Y-X)$ を，それぞれ独立に実行できる．プロセス P および Q はイベント集合 $(X \cap Y)$ によって同期される．つまり，$(X \cap Y)$ に含まれるイベントは P と Q で同時に実行されなければならない．

$$
\begin{aligned}
traces(P_X||_Y Q) &= \{s \in (X \cup Y)^* \mid s \upharpoonright X \in traces(P) \wedge s \upharpoonright Y \in traces(Q)\} \\
failures(P_X||_Y Q) &= \{(u, W \cup Z) \mid W - Y = Z - X \wedge \\
&\quad \exists s, t.(s, W) \in failures(P) \wedge (t, Z) \in failures(Q) \wedge \\
&\quad u \in s \underset{X \cap Y}{\|} t\}
\end{aligned}
$$

ここで，$s \upharpoonright X$ はトレース s の中で X に含まれるイベントのみを抽出したトレースである．また，$s \underset{X \cap Y}{\|} t$ は s と t とをインターリーブすることで生成できるトレースの集合である．$P_X||_Y Q$ が実行できるトレースは，X に含まれるイベントのみを抽出すれば P で実行でき，Y に含まれるイベントのみを抽出すれば Q で実行できる．失敗集合はイベント集合 $(X \cap Y)$ については P の失敗集合と Q の失敗集合の和集合となる．一方，X のみに含まれるイベントは P のみによって決定される．X のみに含まれるイベントは Q では常に失敗の可能性があるため，$W - Y = Z - X$ という条件によってそれらを含まない集合のみが抽出される．Y のみに含まれるイベントも同様である．$s \upharpoonright X$ と $s \underset{X}{||} t$ はそれぞれ次のように定義できる．

$$
\begin{aligned}
\langle\rangle \upharpoonright A &= \langle\rangle \\
(s^\frown\langle a\rangle) \upharpoonright A &= (s \upharpoonright A)^\frown\langle a\rangle \\
(s^\frown\langle b\rangle) \upharpoonright A &= (s \upharpoonright A)
\end{aligned}
$$

ただし，$a \in A$，$b \notin A$.

$$
\begin{aligned}
s \underset{X}{\|} t &= t \underset{X}{\|} s \\
\langle\rangle \underset{X}{\|} \langle\rangle &= \{\langle\rangle\} \\
\langle\rangle \underset{X}{\|} \langle x\rangle &= \{\langle\rangle\} \\
\langle\rangle \underset{X}{\|} \langle y\rangle &= \{\langle y\rangle\} \\
\langle\rangle \underset{X}{\|} \langle x\rangle \hat{} t &= \{\} \\
\langle\rangle \underset{X}{\|} \langle y\rangle \hat{} t &= \{\langle y\rangle \hat{} u \mid u \in \langle\rangle \underset{X}{\|} t\} \\
\langle x\rangle \hat{} s \underset{X}{\|} \langle y\rangle \hat{} t &= \{\langle y\rangle \hat{} u \mid u \in \langle x\rangle \hat{} s \underset{X}{\|} t\} \\
\langle x\rangle \hat{} s \underset{X}{\|} \langle x\rangle \hat{} t &= \{\langle x\rangle \hat{} u \mid u \in s \underset{X}{\|} t\} \\
\langle x\rangle \hat{} s \underset{X}{\|} \langle x'\rangle \hat{} t &= \{\} \\
\langle y\rangle \hat{} s \underset{X}{\|} \langle y'\rangle \hat{} t &= \{\langle y\rangle \hat{} u \mid u \in s \underset{X}{\|} \langle y'\rangle \hat{} t\} \\
&\quad \cup \{\langle y'\rangle \hat{} u \mid u \in \langle y\rangle \hat{} s \underset{X}{\|} t\}
\end{aligned}
$$

ただし $x \in X$，$x' \in X$，$y \notin X$，$y' \notin X$，$x \neq x'$，$y \neq y'$．

- 隠蔽: $P \setminus X$ はプロセス P のうち，X に含まれるイベントが隠蔽されたプロセスである．X に含まれるイベントはプロセス外部からは観測できない．

$$
\begin{aligned}
traces(P \setminus X) &= \{s \setminus X \mid s \in traces(P)\} \\
failures(P \setminus X) &= \{(s \setminus X, Y) \mid (s, Y \cup X) \in failures(P)\}
\end{aligned}
$$

ここで，$s \setminus X$ はトレース s から X に含まれるイベントを除去したトレースである．すべてのイベントの集合を Σ とすれば，前述の $\restriction$ 演算子を用いて $s \setminus X = s \restriction (\Sigma - X)$ と定義できる．この定義では，トレース集合，失敗集合ともトレースから X に含まれるイベントを除去したものとなっている．また，失敗集合では，$failures(P)$ に X がすべて含まれるもののみが抽出されている．これは，もし $failures(P)$ に X がすべて含まれていなければ，外部からのイベントを拒否する前に隠蔽されたイベントをプロセス内部で実行できるということを表している．

B　定理 1 の証明

B.1　補題 (1)

次の補題を s の長さについての構造的帰納法で証明する．

$$g(s, P) \Leftrightarrow g(\langle a \rangle \hat{} s, a \rightarrow P)$$

(1) $s = \langle\rangle$ の場合：
左辺について，$g(\langle\rangle, P) = true$.
右辺について，

$$\begin{aligned}
& g(\langle a \rangle, a \rightarrow P) \\
&= g(\langle\rangle \hat{} \langle a \rangle, a \rightarrow P) \\
&= g(\langle\rangle, a \rightarrow P) \wedge ((\langle\rangle, \{a\}) \notin failures(a \rightarrow P)) \\
&= true \wedge (\langle\rangle, \{a\}) \notin (\{(\langle\rangle, X) \mid a \notin X\} \cup \{(\langle a \rangle \hat{} t, X) \mid (t, X) \in failures(P)\}) \\
&= true
\end{aligned}$$

(2) $s = s' \hat{} \langle a' \rangle$ の場合：
s' の長さは s よりも 1 短いので，帰納法の仮定として以下を仮定する．

$$g(s', P) \Leftrightarrow g(\langle a \rangle \hat{} s', a \rightarrow P)$$

(2-1) 左辺 $\Rightarrow$ 右辺を証明する．
$g(s, P) = true$ を仮定すると，$g(s' \hat{} \langle a' \rangle, P) = true$．よって，$g(s', P) \wedge ((s', \{a'\}) \notin failures(P)) = true$．つまり，

$$\begin{aligned}
g(s', P) &= true \\
(s', \{a'\}) &\notin failures(P)
\end{aligned}$$

帰納法の仮定より，$g(\langle a \rangle \hat{} s', a \rightarrow P) = true$ が成立する．
右辺について，

$$\begin{aligned}
& g(\langle a \rangle \hat{} s, a \rightarrow P) \\
&= g(\langle a \rangle \hat{} s' \hat{} \langle a' \rangle, a \rightarrow P) \\
&= g(\langle a \rangle \hat{} s', a \rightarrow P) \wedge ((\langle a \rangle \hat{} s', \{a'\}) \notin failures(a \rightarrow P)) \\
&= true \wedge ((s', \{a'\}) \notin failures(P)) \\
&= true
\end{aligned}$$

(2-2) 右辺 $\Rightarrow$ 左辺を証明する．

$g(\langle a\rangle\hat{}s, a \to P) = true$ を仮定すると，$g(\langle a\rangle\hat{}s'\hat{}\langle a'\rangle, a \to P) = true$．よって，$g(\langle a\rangle\hat{}s', a \to P) \wedge ((\langle a\rangle\hat{}s', \{a'\}) \notin failures(a \to P)) = true$．つまり，

$$\begin{aligned} g(\langle a\rangle\hat{}s', a \to P) &= true \\ (s', \{a'\}) &\notin failures(P) \end{aligned}$$

帰納法の仮定より，$g(s', P) = true$ が成立する．

左辺について，

$$\begin{aligned} g(s, P) &= g(s'\hat{}\langle a'\rangle, P) \\ &= g(s', P) \wedge ((s', \{a'\}) \notin failures(P)) \\ &= true \end{aligned}$$

以上より証明された．

B.2 補題 (2)

次の補題を s の長さについての構造的帰納法で証明する．

$$a \neq b \Rightarrow g(\langle a\rangle\hat{}s, b \to P) = false$$

$s = \langle\rangle$ の場合，

$$\begin{aligned} g(\langle a\rangle, b \to P) &= g(\langle\rangle, b \to P) \wedge (\langle\rangle, \{a\}) \notin failures(b \to P) \\ &= (\langle\rangle, \{a\}) \notin failures(b \to P) \end{aligned}$$

ここで，$\{b\} \notin \{a\}$ より，$(\langle\rangle, \{a\}) \in failures(b \to P)$．したがって，$g(\langle a\rangle, b \to P) = false$．

$s = s'\hat{}\langle a'\rangle$ の場合，

$$g(\langle a\rangle\hat{}s'\hat{}\langle a'\rangle, b \to P) = g(\langle a\rangle\hat{}s', b \to P) \wedge (\langle a\rangle\hat{}s', \{a\}) \notin failures(b \to P)$$

帰納法の仮定より，$g(\langle a\rangle\hat{}s', b \to P) = false$．したがって，$g(\langle a\rangle, b \to P) = false$．

以上より証明された．

B.3 定理 1(1) イベントが等しい場合

$$\bigcirc_{i\in I}@(a \to P_i) =_F a \to (\bigcirc_{i\in I}@P_i)$$

traces

左辺について，

$$\begin{aligned} traces(\bigcirc_{i\in I}@(a \to P_i)) &= \cup_{i\in I} traces(a \to P_i) \\ &= \cup_{i\in I}(\langle\rangle \cup \{(\langle a\rangle\hat{}\, s) \mid s \in traces(P_i)\}) \\ &= \{\langle\rangle\} \cup \cup_{i\in I}\{(\langle a\rangle\hat{}\, s) \mid s \in traces(P_i)\} \end{aligned}$$

右辺について，

$$\begin{aligned} traces(a \to (\bigcirc_{i\in I}@P_i)) &= \{\langle\rangle\} \cup \{(\langle a\rangle\hat{}\, s) \mid s \in traces(\bigcirc_{i\in I}@P_i)\} \\ &= \{\langle\rangle\} \cup \{(\langle a\rangle\hat{}\, s) \mid s \in \cup_{i\in I} traces(P_i)\} \\ &= \{\langle\rangle\} \cup \cup_{i\in I}\{(\langle a\rangle\hat{}\, s) \mid s \in traces(P_i)\} \end{aligned}$$

以上より，左辺と右辺の traces は等しい．

failures

左辺について，

$$\begin{aligned} &failures(\bigcirc_{i\in I}@(a \to P_i)) \\ &= \{(s, X) \mid (s, X) \in \cup_{i\in I} failures(a \to P_i), \\ &\quad \forall i \in I.\ (g(s, a \to P_i) \Rightarrow (s, X) \in failures(a \to P_i))\} \\ &= \{(\langle\rangle, X) \mid a \notin X, \\ &\quad \forall i \in I.\ (g(\langle\rangle, a \to P_i) \Rightarrow (\langle\rangle, X) \in failures(a \to P_i))\} \\ &\quad \cup\{(\langle a\rangle\hat{}\, s', X) \mid (s', X) \in \cup_{i\in I} failures(P_i), \\ &\quad \forall i \in I.\ (g(s, a \to P_i) \Rightarrow (s, X) \in failures(a \to P_i))\} \\ &= \{(\langle\rangle, X) \mid a \notin X\} \\ &\quad \cup\{(\langle a\rangle\hat{}\, s', X) \mid (s', X) \in \cup_{i\in I} failures(P_i), \\ &\quad \forall i \in I.\ (g(\langle a\rangle\hat{}\, s', a \to P_i) \Rightarrow (s', X) \in failures(P_i))\} \end{aligned}$$

右辺について，

$$\begin{aligned} &failures(a \to (\bigcirc_{i\in I}@P_i)) \\ &= \{(\langle\rangle, X) \mid a \notin X\} \cup \{(\langle a\rangle\hat{}\, s', X) \mid (s', X) \in failures(\bigcirc_{i\in I}@P_i)\} \\ &= \{(\langle\rangle, X) \mid a \notin X\} \\ &\quad \cup\{(\langle a\rangle\hat{}\, s', X) \mid (s', X) \in \cup_{i\in I} failures(P_i), \\ &\quad \forall i \in I.\ (g(s', P_i) \Rightarrow (s', X) \in failures(P_i))\} \end{aligned}$$

補題より $g(\langle a\rangle\hat{}\, s', a \to P_i) = g(s', P_i)$ なので，左辺と右辺の failures は等しい．

B.4　定理 1(2) イベントが異なる場合

$$(i \neq j \Rightarrow a_i \neq a_j) \quad \Rightarrow \quad \bigcirc_{i\in I}@(a_i \rightarrow P_i) =_F \ \Box_{i\in I}@(a_i \rightarrow P_i)$$

traces

$$\begin{aligned} traces(\bigcirc_{i\in I}@(a_i \rightarrow P_i)) &= \cup_{i\in I} traces(a \rightarrow P_i) \\ &= traces(\Box_{i\in I}@(a_i \rightarrow P_i)) \end{aligned}$$

より，左辺と右辺の traces は等しい．

failures

右辺を変形すると．

$$\begin{aligned} & failures(\Box_{i\in I}@(a_i \rightarrow P_i)) \\ &= \{(\langle\rangle, X) \mid X \cap A = \phi\} \cup \{(\langle a_i\rangle\hat{}s', X) \mid i \in I, (s', X) \in failures(P_i)\} \end{aligned}$$

ただし $A = \{a_i \mid i \in I\}$.

(1) 左辺 $\subseteq$ 右辺を証明する．

$(s, X) \in failures(\bigcirc_{i\in I}@(a_i \rightarrow P_i))$ を仮定すると，

$$\begin{aligned} (s, X) &\in \cup_{i\in I} failures(a_i \rightarrow P_i) \\ \forall i \in I.\ (g(s, a_i \rightarrow P_i) &\Rightarrow (s, X) \in failures(a_i \rightarrow P_i)) \end{aligned}$$

$s = \langle\rangle$ の場合，$g(\langle\rangle, a_i \rightarrow P_i) = true$ より，$\forall i \in I.\ (\langle\rangle, X) \in failures(a_i \rightarrow P_i)$. よって，$X \cap A = \phi$. したがって，$(s, X) \in failures(\Box_{i\in I}@(a_i \rightarrow P_i))$.

$s = \langle a\rangle\hat{}s'$ の場合，$(s, X) \in \cup_{i\in I} failures(a_i \rightarrow P_i)$ より，$(\langle a\rangle\hat{}s', X) \in \{(\langle a_i\rangle\hat{}s', X) \mid i \in I, (s', X) \in failures(P_i)\}$. したがって，$(s, X) \in failures(\Box_{i\in I}@(a_i \rightarrow P_i))$.

(2) 右辺 $\subseteq$ 左辺を証明する．

$(s, X) \in failures(\Box_{i\in I}@(a_i \rightarrow P_i))$ を仮定すると，

$$(s, X) \in \{(\langle\rangle, X) \mid X \cap A = \phi\} \cup \{(\langle a_i\rangle\hat{}s', X) \mid i \in I, (s', X) \in failures(P_i)\}$$

$s = \langle\rangle$ の場合，$X \cap A = \phi$．任意の $i \in I$ について $a_i \notin X$ なので，$(s, X) \in \cup_{i \in I} failures(a_i \to P_i)$ かつ $\forall j \in I.\ (s, X) \in failures(a_i \to P_i)$．したがって，$(s, X) \in failures(\bigcirc_{i \in I} @(a_i \to P_i))$．

$s = \langle a_i \rangle \hat{} s'$ の場合，$i \in I, (s', X) \in failures(P_i)$．よって $(s, X) \in \cup_{i \in I} failures(a_i \to P_i)$ は成り立つ．したがって，

$$\forall j \in I.\ (g(s, a_j \to P_j) \ \Rightarrow\ (s, X) \in failures(a_j \to P_j))$$

が成り立てばよい．

$$\forall j \in I.\ (g(\langle a_i \rangle \hat{} s', a_j \to P_j) \ \Rightarrow\ (\langle a_i \rangle \hat{} s', X) \in failures(a_j \to P_j))$$

ここで，$i \neq j$ の時，$a_i \neq a_j$ であるから，補題より $g(\langle a_j \rangle \hat{} s', a_i \to P_i) = false$ となり条件は成り立つ．$i = j$ とすると，

$$g(\langle a_i \rangle \hat{} s', a_i \to P_i) \ \Rightarrow\ (\langle a_i \rangle \hat{} s', X) \in failures(a_i \to P_i)$$

$(\langle a_i \rangle \hat{} s', X) \in failures(a_i \to P_i)$ は $\to$ 演算子の定義より成り立つ．

したがって，$(s, X) \in failures(\bigcirc_{i \in I} @(a_i \to P_i))$．

以上より，左辺と右辺の failures は等しい．

C　定理 2 の証明

この証明では，4.2 節の定義 1, 2, 3 の他，以下の定理を用いる．これらは，3.2 節および 4.2 節で挙げた各演算子の定義から容易に導ける．

定理 4

$$traces(\Box_{a\in A}@a \to P_a) = \{\langle\rangle\} \cup \{\langle a\rangle\hat{}s \mid a \in A, s \in traces(P_a)\}$$

$$failures(\Box_{a\in A}@a \to P_a) = \{(\langle\rangle, X) \mid X \cap A = \phi\}$$
$$\cup\{(\langle a\rangle\hat{}s, X) \mid a \in A, (s, X) \in failures(P_a)\}$$

定理 5

$$traces(\sqcap_{a\in A}@a \to P_a) = \{\langle\rangle\} \cup \{\langle a\rangle\hat{}s \mid a \in A, s \in traces(P_a)\}$$

$$failures(\sqcap_{a\in A}@a \to P_a) = \{(\langle\rangle, X) \mid \exists a \in A.\ a \notin X\}$$
$$\cup\{(\langle a\rangle\hat{}s, X) \mid a \in A, (s, X) \in failures(P_a)\}$$

定理 6

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

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

C.1　定理 2(1) 受信イベントのみの場合

$$(A_! = \phi) \Rightarrow (\Box_{a\in A}@(a \to P_a))\$\Sigma! =_F \Box_{a\in A_?}@(a \to P_a\$\Sigma!)$$

$A_! = \phi$ を仮定すると $A = A_?$ なので，

$$(\Box_{a\in A_?}@(a \to P_a))\$\Sigma! =_F \Box_{a\in A_?}@(a \to P_a\$\Sigma!)$$

が言えればよい．$\Box_{a\in A_?}@(a \to P_a)$ を P とおく．

traces

左辺について，定義 3 と定理 4 より，

$$\begin{aligned} traces((\Box_{a\in A_?}@(a \to P_a))\$\Sigma!) &= traces(\Box_{a\in A_?}@(a \to P_a)) \\ &= \{\langle\rangle\} \cup \{\langle a\rangle\hat{}s \mid a \in A_?, s \in traces(P_a)\} \end{aligned}$$

右辺について，定義 3 と定理 4 より，

$$\begin{aligned} traces(\Box_{a\in A_?}@(a \to P_a\$\Sigma!)) &= \{\langle\rangle\} \cup \{\langle a\rangle\hat{}s \mid a \in A_?, s \in traces(P_a\$\Sigma!)\} \\ &= \{\langle\rangle\} \cup \{\langle a\rangle\hat{}s \mid a \in A_?, s \in traces(P_a)\} \end{aligned}$$

以上より，右辺と左辺の traces は等しい．

failures

(1) $failures(P\$\Sigma!) \subseteq failures(\Box_{a \in A_?} @(a \rightarrow P_a\$\Sigma!)$ を証明する．

$$(s, X) \in failures(P\$\Sigma!)$$

とする．定義 3 より，$(s, X) \in failures(P)$ または，$\exists Y.((s, Y) \in failures(P) \wedge X \subseteq Y \cup \Sigma!), \exists a.(s^\frown\langle a\rangle \in traces(P) \wedge a \notin X))$.
(i) $(s, X) \in failures(P)$ の場合：
定理 4 より，

$$(s, X) \in \{(\langle\rangle, X) \mid X \cap A_? = \phi\} \cup \{(\langle a\rangle^\frown s', X) \mid a \in A_?, (s', X) \in failures(P_a)\}$$

(i-1) $(s, X) \in \{(\langle\rangle, X) \mid X \cap A_? = \phi\}$ の場合：
すなわち，$s = \langle\rangle, X \cap A_? = \phi$．よって，定理 4 より，

$$(\langle\rangle, X) \in failures(\Box_{a \in A_?} @(a \rightarrow P_a\$\Sigma!))$$

(i-2) $(s, X) \in \{(\langle a\rangle^\frown s', X) \mid a \in A_?, (s', X) \in failures(P_a)\}$ の場合：
すなわち，$s = \langle a\rangle^\frown s', a \in A_?, (s', X) \in failures(P_a)$. 定義 3 より，

$$(s', X) \in failures(P_a\$\Sigma!).$$

よって，定理 4 より，

$$(s, X) \in failures(\Box_{a \in A_?} @(a \rightarrow P_a\$\Sigma!))$$

(ii) ある Y と a で $(s, Y) \in failures(P), X \subseteq Y \cup \Sigma!, s^\frown\langle a\rangle \in traces(P), a \notin X$ の場合：
定理 4 より，

$$(s, Y) \in \{(\langle\rangle, X) \mid X \cap A_? = \phi\} \cup \{(\langle a'\rangle^\frown s', X) \mid a' \in A_?, (s', X) \in failures(P_{a'})\}$$

(ii-1) $(s, Y) \in \{(\langle\rangle, X) \mid X \cap A_? = \phi\}$ の場合：
すなわち，$s = \langle\rangle, Y \cap A_? = \phi$．ここで，$A_? \subseteq \Sigma?$ かつ $X \subseteq Y \cup \Sigma!$ であるので，$X \cap A_? = \phi$．よって，定理 4 より，

$$(\langle\rangle, X) \in failures(\Box_{a \in A_?} @(a \rightarrow P_a\$\Sigma!))$$

(ii-2) $(s, Y) \in \{(\langle a'\rangle^\frown s', X) \mid a' \in A_?, (s', X) \in failures(P_{a'})\}$ の場合：
すなわち，$s = \langle a'\rangle^\frown s', a' \in A_?, (s', X) \in failures(P_{a'})$. また，$\langle a'\rangle^\frown s'^\frown\langle a\rangle \in traces(P)$ であるので，定理 4 より，$s'^\frown\langle a\rangle \in traces(P_{a'})$．以上をまとめると，

$$(s', X) \in failures(P_{a'}), X \subseteq Y \cup \Sigma!, s'^\frown\langle a\rangle \in traces(P_{a'}), a \notin X$$

であるので，定義 3 より，

$$(s', X) \in failures(P_{a'}\$\Sigma!)$$

よって，定理 4 より，

$$(s, X) \in failures(\Box_{a \in A_?} @(a \to P_a\$\Sigma!))$$

(2) $failures(\Box_{a \in A_?} @(a \to P_a\$\Sigma!)) \subseteq failures(P\$\Sigma!)$ を証明する．
$(s, X) \in failures(\Box_{a \in A_?} @(a \to P_a\$\Sigma!))$ とする．定理 4 より，

$$(s, X) \in \{(\langle\rangle, X) \mid X \cap A_? = \phi\} \cup \{(\langle a\rangle \hat{} s', X) \mid a \in A_?, (s', X) \in failures(P_a\$\Sigma!)\}$$

(i) $(s, X) \in \{(\langle\rangle, X) \mid X \cap A_? = \phi\}$ の場合：
すなわち，$s = \langle\rangle, X \cap A_? = \phi$. よって，定義 3 と定理 4 より，

$$(\langle\rangle, X) \in failures((\Box_{a \in A_?} @x \to P_a)\$\Sigma!) = failures(P\$\Sigma!)$$

(ii) $(s, X) \in \{(\langle a\rangle \hat{} s', X) \mid a \in A_?, (s', X) \in failures(P_a\$\Sigma!)\}$ の場合：
すなわち，$s = \langle a\rangle \hat{} s', a \in A_?, (s', X) \in failures(P_a\$\Sigma!)$. 定義 3 より，ある Y と a で，

$$(s', Y) \in failures(P_a), X \subseteq Y \cup \Sigma!, s' \hat{} \langle a\rangle \in traces(P_a), a \notin X$$

よって，定理 4 より，

$$(s, X) \in failures(\Box_{a \in A_?} @a \to P_a) \subseteq failures(P\$\Sigma!)$$

(1) と (2) より，

$$failures((\Box_{a \in A_?} @(a \to P_a))\$\Sigma!) = failures(\Box_{a \in A_?} @(a \to P_a\$\Sigma!))$$

C.2　定理 2(2) 送信イベントのみの場合

$$(A_! \neq \phi \wedge A_? = \phi) \Rightarrow (\Box_{a \in A} @(a \to P_a))\$\Sigma! =_F \sqcap_{a \in A_!} @(a \to P_a\$\Sigma!)$$

$A_! \neq \phi \wedge A_? = \phi$ を仮定すると $A = A_!$ なので，

$$(\Box_{a \in A_!} @(a \to P_a))\$\Sigma! =_F \sqcap_{a \in A_!} @(a \to P_a\$\Sigma!)$$

が言えればよい．$\Box_{a \in A_!} @(a \to P_a)$ を P とおく．

traces

左辺について，定義 3 と定理 4 より，

$$\begin{aligned} traces((\Box_{a\in A_!}@(a \to P_a))\$\Sigma!) &= traces(\Box_{a\in A_!}@(a \to P_a)) \\ &= \{\langle\rangle\} \cup \{\langle a\rangle\hat{}s \mid a \in A_!, s \in traces(P_a)\} \end{aligned}$$

右辺について，定義 3 と定理 5 より，

$$\begin{aligned} traces(\Box_{a\in A_!}@(a \to P_a\$\Sigma!)) &= \{\langle\rangle\} \cup \{\langle a\rangle\hat{}s \mid a \in A_!, s \in traces(P_a\$\Sigma!)\} \\ &= \{\langle\rangle\} \cup \{\langle a\rangle\hat{}s \mid a \in A_!, s \in traces(P_a)\} \end{aligned}$$

以上より，右辺と左辺の traces は等しい．

failures

(1) $failures(P\$\Sigma!) \subseteq failures(\sqcap_{a\in A_!}@(a \to P_a\$\Sigma!)$ を証明する．
$(s, X) \in failures(P\$\Sigma!)$ とする．定義 3 より，$(s, X) \in failures(P)$ または，$\exists Y.((s, Y) \in failures(P) \wedge X \subseteq Y \cup \Sigma!), \exists a.(s\hat{}\langle a\rangle \in traces(P) \wedge a \notin X))$.
(i) $(s, X) \in failures(P)$ の場合：
定理 4 より，

$$(s, X) \in \{(\langle\rangle, X) \mid X \cap A_! = \phi\} \cup \{(\langle a\rangle\hat{}s', X) \mid a \in A_!, (s', X) \in failures(P_a)\}$$

(i-1) $(s, X) \in \{(\langle\rangle, X) \mid X \cap A_! = \phi\}$ の場合：
すなわち，$s = \langle\rangle, X \cap A_! = \phi$．$X \cap A_! = \phi$ と $A_! \neq \phi$ より，ある $a \in A_!$ で $a \notin X$. よって，定理 5 より，

$$(\langle\rangle, X) \in failures(\sqcap_{a\in A_!}@(a \to P_a\$\Sigma!))$$

(i-2) $(s, X) \in \{(\langle a\rangle\hat{}s', X) \mid a \in A_!, (s', X) \in failures(P_a)\}$ の場合：
すなわち，$s = \langle a\rangle\hat{}s', a \in A_!, (s', X) \in failures(P_a)$. 定義 3 より，

$$(s', X) \in failures(P_a\$\Sigma!).$$

よって，定理 5 より，

$$(s, X) \in failures(\sqcap_{a\in A_?}@(a \to P_a\$\Sigma!))$$

(ii) ある Y と a で $(s, Y) \in failures(P), X \subseteq Y \cup \Sigma!, s\hat{}\langle a\rangle \in traces(P), a \notin X$ の場合：
定理 4 より，

$$(s, Y) \in \{(\langle\rangle, X) \mid X \cap A_! = \phi\} \cup \{(\langle a'\rangle\hat{}s', X) \mid a' \in A_!, (s', X) \in failures(P_{a'})\}$$

(ii-1) $(s, Y) \in \{(\langle\rangle, X) \mid X \cap A_! = \phi\}$ の場合：

すなわち，$s = \langle\rangle, Y \cap A_! = \phi$. ここで，$\langle\rangle\hat{}\langle a\rangle \in traces(P)$ であるので，定理 4 より，$a \in A_!$. すなわち，$a \in A_!, a \notin X$ よって，定理 5 より，

$$(s, X) \in failures(\sqcap_{a \in A_!} @(a \rightarrow P_a \$ \Sigma!))$$

(ii-2) $(s, Y) \in \{(\langle a'\rangle\hat{}s', X) \mid a' \in A_!, (s', X) \in failures(P_{a'})\}$ の場合：

すなわち，$s = \langle a'\rangle\hat{}s', a' \in A_!, (s', X) \in failures(P_{a'})$. また，$\langle a'\rangle\hat{}s'\hat{}\langle a\rangle \in traces(P)$ であるので，定理 4 より，$s'\hat{}\langle a\rangle \in traces(P_{a'})$. 以上をまとめると，

$$(s', X) \in failures(P_{a'}), X \subseteq Y \cup \Sigma!, s'\hat{}\langle a\rangle \in traces(P_{a'}), a \notin X$$

であるので，定義 3 より，

$$(s', X) \in failures(P_{a'} \$ \Sigma!)$$

よって，定理 5 より，

$$(s, X) \in failures(\sqcap_{a \in A_!} @(a \rightarrow P_a \$ \Sigma!))$$

(2) $failures(\sqcap_{a \in A_!} @(a \rightarrow P_a \$ \Sigma!)) \subseteq failures(P \$ \Sigma!)$ を証明する.

$(s, X) \in failures(\sqcap_{a \in A_!} @(a \rightarrow P_a \$ \Sigma!)$ とする．定理 5 より，$(s, X) \in \{(\langle\rangle, X) \mid \exists a \in A_!.\ a \notin X\} \cup \{(\langle a'\rangle\hat{}s', X) \mid a' \in A_!, (s', X) \in failures(P_{a'} \$ \Sigma!)\}$.

(i) $(s, X) \in \{(\langle\rangle, X) \mid \exists a \in A_!.\ a \notin X\}$ の場合：

$Y = \Sigma? \cup (\Sigma! - A_!)$ とおくと，定理 4 より，

$$(\langle\rangle, Y) \in failures(\Box_{a' \in A_!} @x \rightarrow P_{a'}) = failures(P)$$

また，仮定より $a \in A_!$ なので，定理 4 より，

$$\langle a\rangle \in traces(\Box_{a' \in A_!} @a \rightarrow P_{a'}) = traces(P)$$

ここで，

$$X \subseteq (\Sigma? \cup \Sigma!) = \Sigma? \cup (\Sigma! - A_!) \cup \Sigma! = Y \cup \Sigma!$$

以上をまとめると，

$$\begin{aligned} s &= \langle\rangle, \\ (\langle\rangle, Y) &\in failures(P), \\ X &\subseteq Y \cup \Sigma!, \\ \langle a\rangle &\in traces(P), \\ a &\notin X \end{aligned}$$

すなわち，

$$(s, X) \in failures(P\$\Sigma!).$$

(ii) $(s, X) \in \{(\langle a' \rangle \hat{} s', X) \mid a' \in A_!, (s', X) \in failures(P_{a'}\$\Sigma!)\}$ の場合：

すなわち，$s = \langle a' \rangle \hat{} s'$, $a' \in A_!$, $(s', X) \in failures(P_{a'}\$\Sigma!)$. 定義 3 より，ある Y と a で，$(s', Y) \in failures(P_{a'})$，$X \subseteq Y \cup \Sigma_!$，$s' \hat{} \langle a \rangle \in traces(P_{a'})$，$a \notin X$．よって，定理 4 と定義 3 より，

$$(s, X) \in failures(\Box_{a \in A_!} @x \rightarrow P_a) \subseteq failures(P\$\Sigma!)$$

(1) と (2) より，

$$failures(P\$\Sigma!) = failures(\sqcap_{a \in A_!} @(a \rightarrow P_a\$\Sigma!))$$

C.3　定理 2(3) 送信イベントと受信イベントが混在する場合

$$(A_! \neq \phi \wedge A_? \neq \phi) \Rightarrow$$
$$(\Box_{a \in A} @(a \rightarrow P_a))\$\Sigma! =_F \sqcap_{a \in A_!} @(a \rightarrow P_a\$\Sigma!) \triangleright \Box_{b \in A_?} @(b \rightarrow P_b\$\Sigma!)$$

$A_! = \phi$ かつ $A_? = \phi$ を仮定する．

$$(\Box_{a \in A} @(a \rightarrow P_a))\$\Sigma! =_F \sqcap_{a \in A_!} @(a \rightarrow P_a\$\Sigma!) \triangleright \Box_{b \in A_?} @(b \rightarrow P_b\$\Sigma!)$$

が言えればよい．

$$\begin{aligned} P &= \Box_{a \in A} @(a \rightarrow P_a) \\ R &= \sqcap_{a \in A_!} @(a \rightarrow P_a\$\Sigma!) \triangleright \Box_{b \in A_?} @(b \rightarrow P_b\$\Sigma!) \end{aligned}$$

とおく．

traces

左辺について，定義 3 と定理 4 より，

$$\begin{aligned} traces(P\$\Sigma!) &= traces(\Box_{a \in A} @(a \rightarrow P_a)) \\ &= \{\langle\rangle\} \cup \{\langle a \rangle \hat{} s \mid a \in A, s \in traces(P_a)\} \end{aligned}$$

右辺について，定義 3, 4, 5, 6 より，

$$\begin{aligned} traces(R) &= traces(\sqcap_{a\in A_!}@(a \to P_a\$\Sigma!) \triangleright \Box_{b\in A_?}@(b \to P_b\$\Sigma!)) \\ &= traces(\sqcap_{a\in A_!}@(a \to P_a\$\Sigma!)) \cup traces(\Box_{b\in A_?}@(b \to P_b\$\Sigma!)) \\ &= \{\langle\rangle\} \cup \{\langle a\rangle\hat{}s \mid a \in A_!, s \in traces(P_a\$\Sigma!)\} \\ &\quad \cup \{\langle b\rangle\hat{}s' \mid b \in A_?, s' \in traces(P_b\$\Sigma!)\} \\ &= \{\langle\rangle\} \cup \{\langle a\rangle\hat{}s \mid a \in A, s \in traces(P_a\$\Sigma!)\} \\ &= \{\langle\rangle\} \cup \{\langle a\rangle\hat{}s \mid a \in A, s \in traces(P_a)\} \end{aligned}$$

以上より，右辺と左辺の traces は等しい．

failures

(1) $failures(P\$\Sigma!) \subseteq failures(R)$ を証明する．

$(s, X) \in failures(P\$\Sigma!)$ とする．定義 3 より，

$(s, X) \in failures(P)$ または，$\exists Y.((s, Y) \in failures(P) \wedge X \subseteq Y \cup \Sigma!), \exists a.(s\hat{}\langle a\rangle \in traces(P) \wedge a \notin X)$.

(i) $(s, X) \in failures(P)$ の場合：

定理 4 より，

$$(s, X) \in \{(\langle\rangle, X) \mid X \cap A = \phi\} \cup \{(\langle a\rangle\hat{}s', X) \mid a \in A, (s', X) \in failures(P_a)\}$$

(i-1) $(s, X) \in \{(\langle\rangle, X) \mid X \cap A = \phi\}$ の場合：

すなわち，$s = \langle\rangle, X \cap A = \phi$．ここで，$A = A_! \cup A_?$ なので，$X \cap A_? = \phi$．よって，定理 4 より，

$$(\langle\rangle, X) \in failures(\Box_{a\in A_?}@(a \to P_a\$\Sigma!))$$

さらに，定理 6 より，

$$(\langle\rangle, X) \in failures(\sqcap_{a\in A_!}@(a \to P_a\$\Sigma!) \triangleright \Box_{b\in A_?}@(b \to P_b\$\Sigma!))$$

(i-2) $(s, X) \in \{(\langle a\rangle\hat{}s', X) \mid a \in A, (s', X) \in failures(P_a)\}$ の場合：

すなわち，

$$s = \langle a\rangle\hat{}s', a \in A_? \cup A_!, (s', X) \in failures(P_a)$$

(i-2-1) $a \in A_?$ の場合：

定理 4 より，

$$(\langle a\rangle\hat{}s', X) \in failures(\Box_{a\in A_?}@(a \to P_a\$\Sigma!))$$

よって，定理 6 より，

$$\begin{aligned} (\langle a\rangle\hat{}s', X) &\in failures(\sqcap_{a\in A_!}@(a \to P_a\$\Sigma!) \triangleright \Box_{a\in A_?}@(a \to P_a\$\Sigma!)) \\ &\subseteq failures(R) \end{aligned}$$

(i-2-2) $a \in A_!$ の場合：
定理 5 より，

$$(\langle a \rangle \hat{} s', X) \in failures(\sqcap_{a \in A_!} @(a \to P_a \$ \Sigma !))$$

よって，$\langle a \rangle \hat{} s' \neq \langle \rangle$ であるので，定理 6 より，

$$\begin{aligned} (\langle a \rangle \hat{} s', X) &\in failures(\sqcap_{a \in A_!} @(a \to P_a \$ \Sigma !) \triangleright \Box_{a \in A_?} @(a \to P_a \$ \Sigma !)) \\ &\subseteq failures(R) \end{aligned}$$

(ii) ある Y と a で $(s, Y) \in failures(P), X \subseteq Y \cup \Sigma !, s \hat{} \langle a \rangle \in traces(P), a \notin X$ の場合：
定理 4 より，

$$(s, Y) \in \{(\langle \rangle, X) \mid X \cap A = \phi\} \cup \{(\langle a' \rangle \hat{} s', X) \mid a' \in A, (s', X) \in failures(P_{a'})\}$$

(ii-1) $(s, Y) \in \{(\langle \rangle, X) \mid X \cap A = \phi\}$ の場合：
すなわち，$s = \langle \rangle, Y \cap A = \phi.$
$Y \cap A = \phi, A_? \subseteq A, X \subseteq Y \cup \Sigma !$ より，$X \cap A_? = \phi$ を導ける．よって，定理 4 より，

$$(\langle \rangle, X) \in failures(\Box_{a \in A_?} @(a \to P_a \$ \Sigma !))$$

さらに，定理 6 より，

$$\begin{aligned} (\langle \rangle, X) &\in failures(\sqcap_{a \in A_!} @(a \to P_a \$ \Sigma !) \triangleright \Box_{a \in A_?} @(a \to P_a \$ \Sigma !)) \\ &\subseteq failures(R) \end{aligned}$$

(ii-2) $(s, Y) \in \{(\langle a' \rangle \hat{} s', X) \mid a' \in A, (s', X) \in failures(P_{a'})\}$ の場合：
すなわち，$s = \langle a' \rangle \hat{} s', a' \in A, (s', X) \in failures(P_{a'})$. また，$\langle a' \rangle \hat{} s' \hat{} \langle a \rangle \in traces(P)$ であるので，定理 4 より，

$$s' \hat{} \langle a \rangle \in traces(P_a)$$

以上をまとめると，

$$\begin{aligned} (s', Y) &\in failures(P_a), \\ X &\subseteq Y \cup \Sigma !, \\ s' \hat{} \langle a \rangle &\in traces(P_a), \\ a &\notin X \end{aligned}$$

であるので，定義 3 より，

$$(s', X) \in failures(P_a \$ \Sigma !).$$

よって，定理 4 および 定理 5 より，

$$\begin{aligned} (\langle a \rangle \hat{} s', X) &\in failures(\sqcap_{a \in A_!} @(a \to P_a \$ \Sigma !)) \\ (\langle a \rangle \hat{} s', X) &\in failures(\Box_{a \in A_?} @(a \to P_a \$ \Sigma !)) \end{aligned}$$

すなわち，定理 6 より，

$$\begin{aligned}(s, X) &\in failures(\sqcap_{a \in A_!} @(a \to P_a \$\Sigma!) \triangleright \Box_{a \in A_?} @(a \to P_a \$\Sigma!)) \\ &\subseteq failures(R)\end{aligned}$$

(2) $failures(R) \subseteq failures(P\$\Sigma!)$ を証明する．
$(s, X) \in failures(R)$ とする．定理 6 より，

$$(s, X) \in failures(\Box_{a \in A_?} @(a \to P_a \$\Sigma!))$$

または，

$$(s, X) \in \{(s, X) \mid s \neq \langle\rangle, (s, X) \in failures(\sqcap_{a \in A_!} @(a \to P_a \$\Sigma!))\}$$

(i) $(s, X) \in failures(\Box_{a \in A_?} @(a \to P_a \$\Sigma!))$ の場合：
すなわち，

$$(s, X) \in \{(\langle\rangle, X) \mid X \cap A_? = \phi\} \cup \{(\langle a \rangle \hat{} s', X) \mid a \in A_?, (s', X) \in failures(P_a \$\Sigma!)\}$$

(i-1) $(s, X) \in \{(\langle\rangle, X) \mid X \cap A_? = \phi\}$ の場合：
すなわち，$s = \langle\rangle, X \cap A_? = \phi$. よって，定義 3 と定理 4 より，

$$(\langle\rangle, X) \in failures(\Box_{a \in A_?} @(a \to P_a))$$

$A_? \neq \phi$ なので，$\exists a' \in A_?$. $X \cap A_? = \phi$ なので，$a' \notin X, \langle a' \rangle \in traces(\Box_{a \in A_?} @ (a \to P_a))$. $Y = X - \Sigma!$ とおくと，$Y \cap (A_! \cup A_?) = \phi, X \subseteq Y \cup \Sigma!$.
以上より，

$$\begin{aligned}(\langle\rangle, X) \in \{(s, X) \mid\ & \exists Y. (s, Y) \in failures(\Box_{a \in A} @ a \to P_a), \\ & X \subseteq Y \cup \Sigma!, \\ & \exists a'.\ s \hat{} \langle a' \rangle \in traces(\Box_{a \in A} @ a \to P_a), \\ & a' \notin X\}\end{aligned}$$

よって，

$$(s, X) \in failures(P\$\Sigma!)$$

(i-2) $(s, X) \in \{(\langle a \rangle \hat{} s', X) \mid a \in A_?, (s', X) \in failures(P_a \$\Sigma!)\}$ の場合：
すなわち，$s = \langle a \rangle \hat{} s', a \in A_?, (s', X) \in failures(P_a \$\Sigma!)$. 定義 3 より

$$(s', X) \in failures(P_a)$$

または，ある Y と a' で，

$$(s', Y) \in failures(P_a), X \subseteq Y \cup \Sigma!, s' \hat{} \langle a' \rangle \in traces(P_a), a' \notin X$$

(i-2-1) $(s', X) \in failures(P_a)$ の場合：

$s = \langle a \rangle \hat{} s'$, $a \in A_? \subseteq A$, $(s', X) \in failures(P_a)$ よって，定理 4 より，

$$\begin{aligned}(\langle a \rangle \hat{} s', X) &\in failures(\Box_{a \in A} @a \to P_a) \\ &\subseteq failures((\Box_{a \in A} @a \to P_a)\$\Sigma!) \\ &= failures(P\$\Sigma!)\end{aligned}$$

(i-2-2) ある Y と a' で $(s', Y) \in failures(P_a), X \subseteq Y \cup \Sigma!, s' \hat{} \langle a' \rangle \in traces(P_a), a' \notin X$ の場合：

$s = \langle a \rangle \hat{} s'$, $a \in A_? \subseteq A$, $(s', Y) \in failures(P_a)$, $X \subseteq Y \cup \Sigma!$, $s' \hat{} \langle a' \rangle \in traces(P_a)$, $a' \notin X$ よって，定理 4 より，

$$(\langle a \rangle \hat{} s', Y) \in failures(\Box_{a \in A} @a \to P_a),$$

ここで，

$$\langle a \rangle \hat{} s' \hat{} \langle a' \rangle \in traces(\Box_{a \in A} @a \to P_a)$$

より，

$$\begin{aligned}(\langle a \rangle \hat{} s', X) &\in failures((\Box_{a \in A} @a \to P_a)\$\Sigma!) \\ &= failures(P\$\Sigma!)\end{aligned}$$

(ii) $(s, X) \in \{(s, X) | s \neq \langle\rangle, (s, X) \in failures(\sqcap_{a \in A_!} @(a \to P_a\$\Sigma!))\}$ の場合：

$$\begin{aligned}s &\neq \langle\rangle \\ (s, X) &\in \{(\langle\rangle, X) \mid \exists a \in A_!.a \notin X\} \\ &\quad \cup \{(\langle a \rangle \hat{} s', X) \mid a \in A_!, (s', X) \in failures(P_a\$\Sigma!)\}\end{aligned}$$

すなわち，

$$s = \langle a \rangle \hat{} s', a \in A_!, (s', X) \in failures(P_a\$\Sigma!)$$

定義 3 より

$$(s', X) \in failures(P_a)$$

または，ある Y と a' で，

$$(s', Y) \in failures(P_a), X \subseteq Y \cup \Sigma!, s' \hat{} \langle a' \rangle \in traces(P_a), a' \notin X$$

(ii-1) $(s', X) \in failures(P_a)$ の場合：

$$s = \langle a \rangle \hat{} s', a \in A_! \subseteq A, (s', X) \in failures(P_a)$$

よって，定理 4 より，

$$\begin{aligned}(\langle a \rangle \hat{} s', X) &\in failures(\Box_{a \in A} @a \to P_a) \\ &\subseteq failures((\Box_{a \in A} @a \to P_a)\$\Sigma!) \\ &= failures(P\$\Sigma!)\end{aligned}$$

(ii-2) ある Y と a' で $(s', Y) \in failures(P_a), X \subseteq Y \cup \Sigma!, s'\hat{}\langle a' \rangle \in traces(P_a), a' \notin X$ の場合：

すなわち $s = \langle a \rangle\hat{}s', a \in A_! \subseteq A, (s', Y) \in failures(P_a), X \subseteq Y \cup \Sigma!, s'\hat{}\langle a' \rangle \in traces(P_a), a' \notin X$

よって，定理 4 より，

$$(\langle a \rangle\hat{}s', Y) \in failures(\Box_{a \in A} @a \rightarrow P_a)$$

ここで，

$$\langle a \rangle\hat{}s'\hat{}\langle a' \rangle \in traces(\Box_{a \in A} @a \rightarrow P_a)$$

より，

$$\begin{aligned} (\langle a \rangle\hat{}s', X) &\in failures((\Box_{a \in A} @a \rightarrow P_a)\$\Sigma!) \\ &= failures(P\$\Sigma!) \end{aligned}$$

(1) と (2) より，

$$failures(P\$\Sigma!) = failures(R)$$