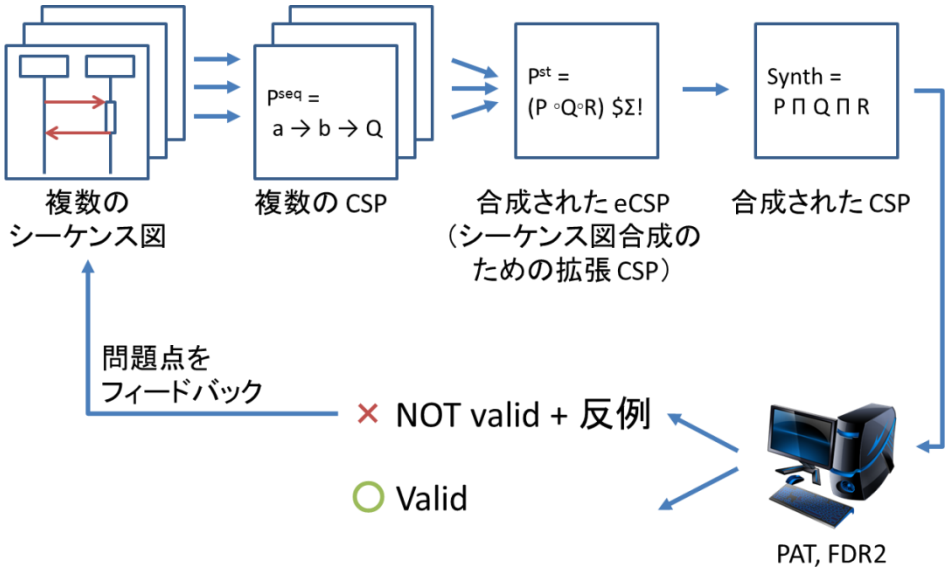
プロセス代数に基づく
非決定的なシナリオ合成による
シーケンス図の検証

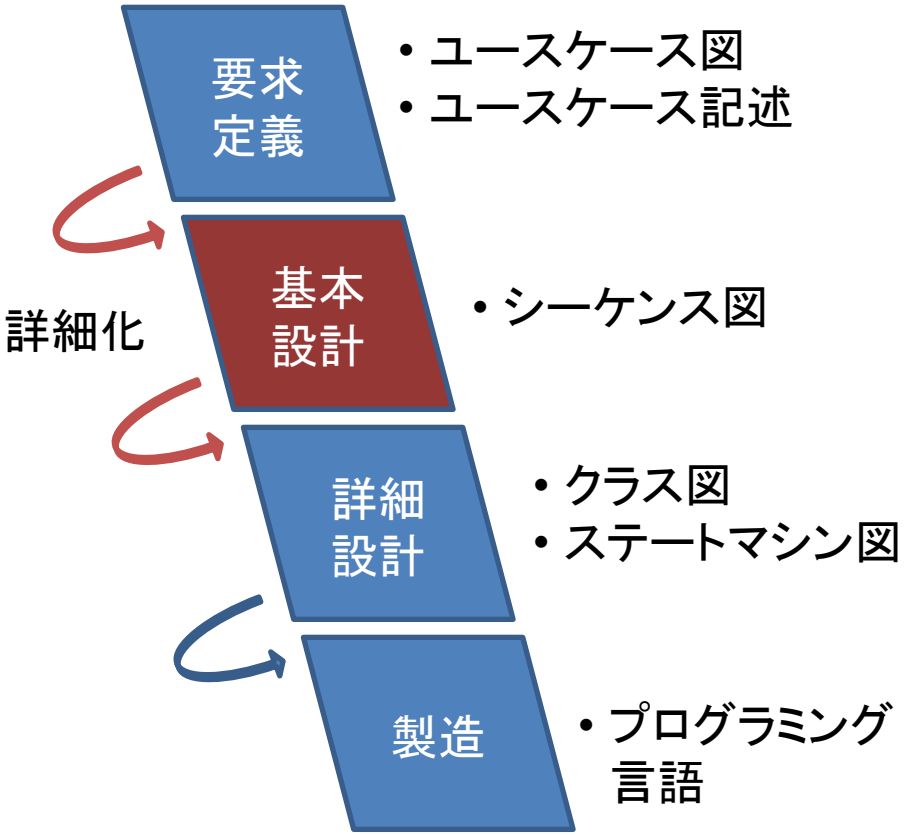
海津 智宏

目次

- はじめに
  - 背景、目的
  - プロセス代数
  - 研究の全体の概要
- CSPによるシーケンス図の検証
  - シーケンス図のCSP記述
  - eCSP（拡張CSP）によるプロセスの合成方法
  - eCSPからCSPへの変換方法
  - 変換ツールSDVerifier
  - 検証と反例解析
- より複雑なシーケンス図への対応
  - オブジェクト生成と破棄
  - 複数オブジェクト
- ケーススタディ
- 関連研究
- おわりに
  - まとめ
  - 成果

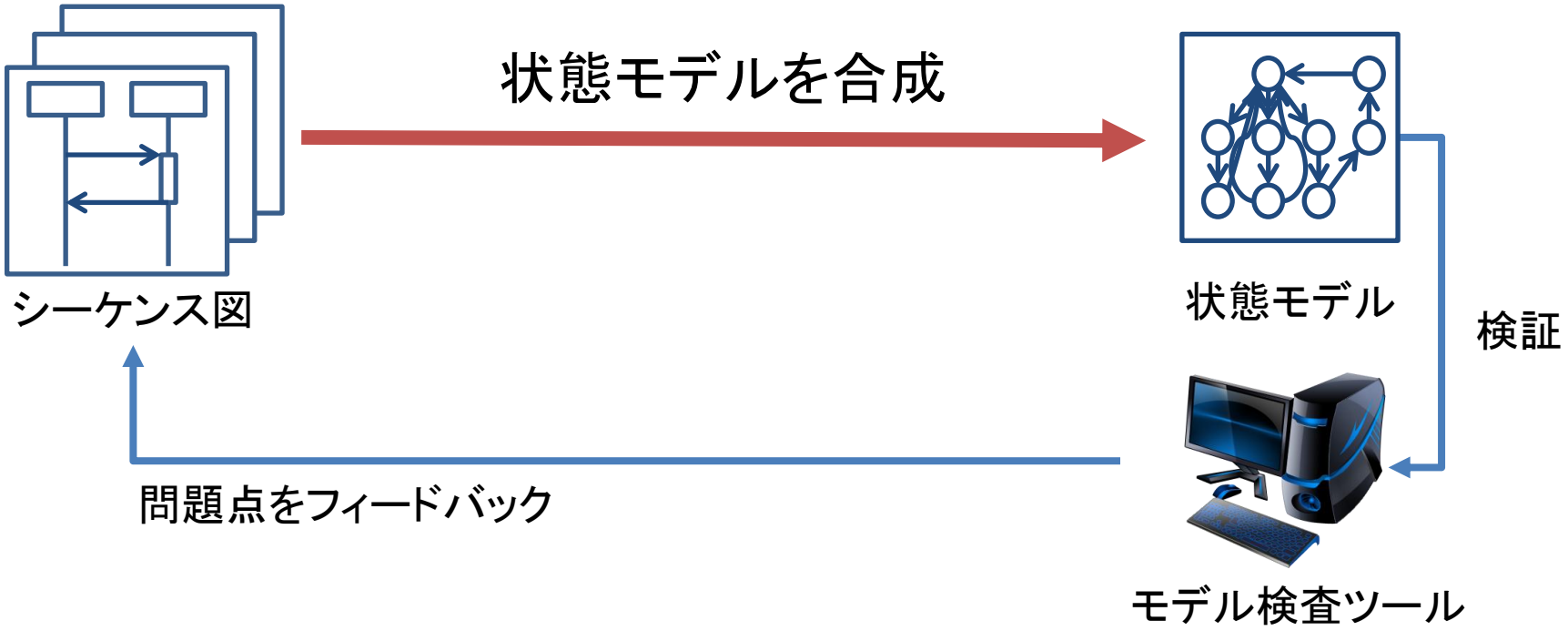


背景



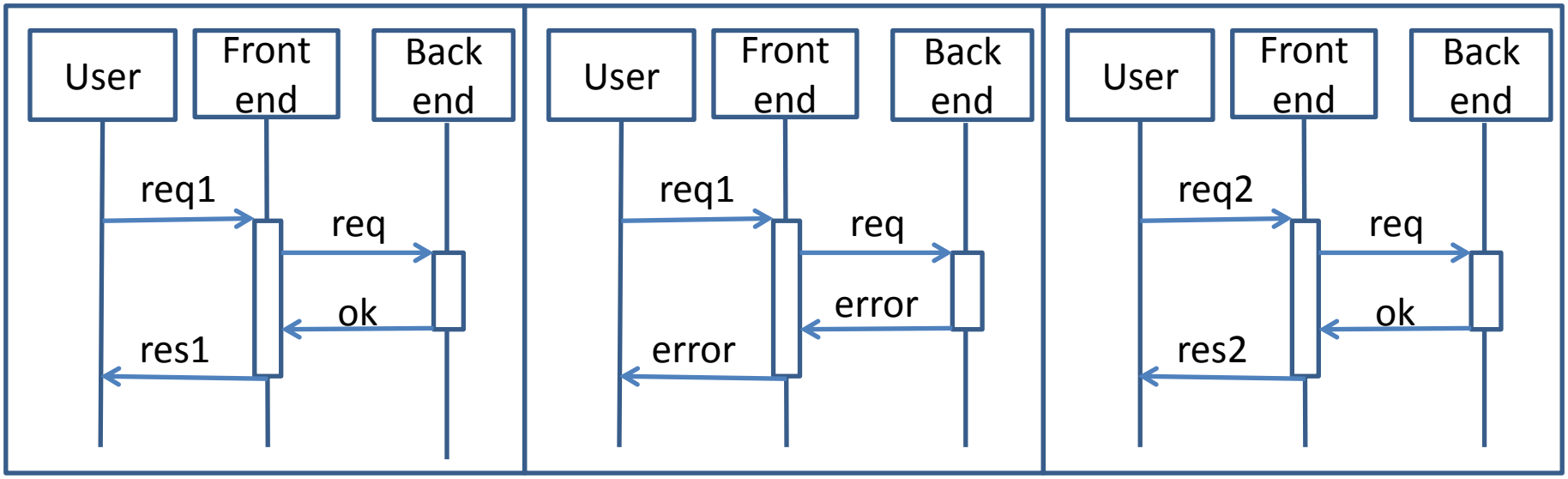
- ソフトウェア開発の上流工程で設計の整合性を検証することで、手戻りコストを削減したい
- 上流工程ではシーケンス図が幅広く利用されている
- 従来手法ではシーケンス図で記述されたふるまいの詳細化を検証することは困難だった

研究の目的



- シーケンス図から状態モデルへの合成手法を提案し、
  シーケンス図で記述された上流工程の設計を検証可能とする

例：不整合のあるシーケンス図



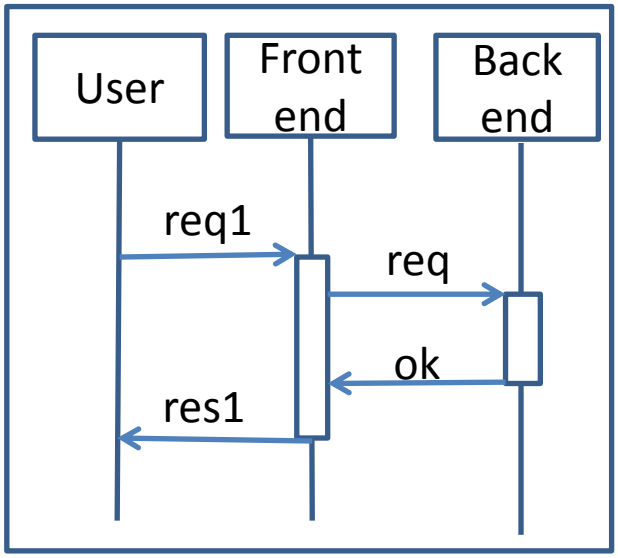
- Backend オブジェクトは req リクエストに対して ok もしくは error を返している
- User が req2 を送信した場合に Backend が error を返すと何が起こるのか？
  - Frontend は User に error を返すべき？
  - Frontend が error を適切に処理して res2 を返すべき？
  - そもそもその状況下で Backend が error を返すことは可能か？
- この設計は不十分であり、シナリオの追加・改善をすべきである

シーケンス図とそのふるまいとの対応

- 「Backend オブジェクトは ok もしくは error を返す」という考え方は通常のシーケンス図の意味論とは異なる
  - 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、各オブジェクトのふるまいは定義しない

- しかし、多くの場合、開発者は暗黙に各オブジェクトのふるまい（状態モデル）を考えており、期待される状態モデルを明確にすることで早い段階で設計を検証することが可能となる

本論文で前提とする シーケンス図と状態モデルとの対応



- 各オブジェクトはそれぞれ独立に動作する
  - User は req1 を送信し res1 を受信する
  - Frontend は req1 を受信し req を送信し ok を受信し res1 を送信する
  - Backend は req を受信し ok を送信する
- 活性区間内の一連のメッセージを送受信している間は、他のメッセージ送受信は行わない
- 活性区間完了時は元の状態にもどり、繰り返し処理を実行できる
  - Backend は req→ok を繰り返し実行できる

プロセス代数 CSP

- 詳細化関係・同値関係が定義されており、設計の各段階の整合性を検証できる
- 非決定性という概念があり、上流工程のモデルを適切に記述できる
  - 複数の選択肢のうち 1 つが非決定的に選択される、ということを記述可能
- PAT, FDR2 といったモデル検査ツールがあり、機械的な検証が可能

eCSP

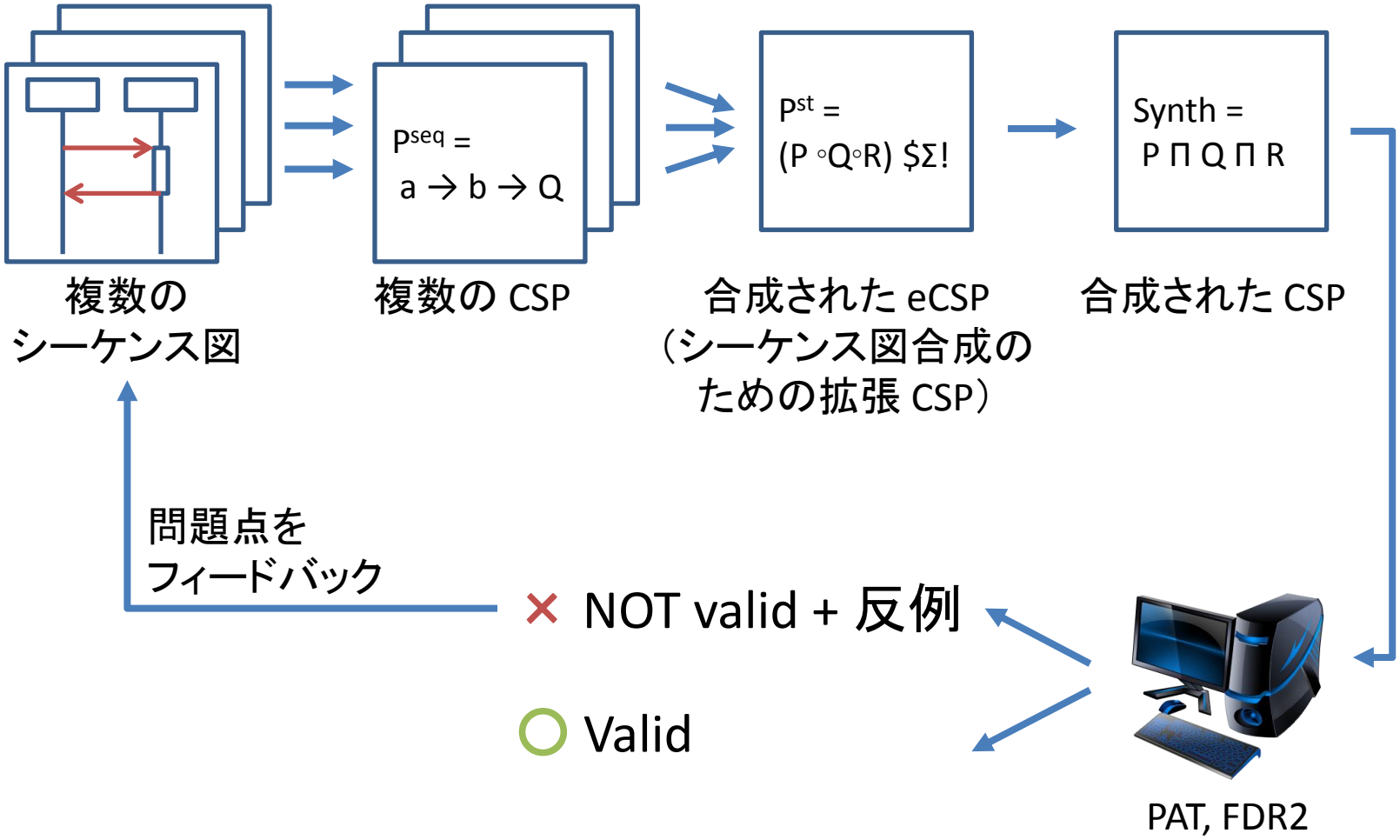
- シーケンス図の合成のために拡張した CSP
  - 標準の CSP に 2 つの演算子を追加
    - シーケンス図マージ演算子。
    - 内部選択化演算子 $

$$C ::= C_X\|_Y C \mid C \setminus X \mid P$$

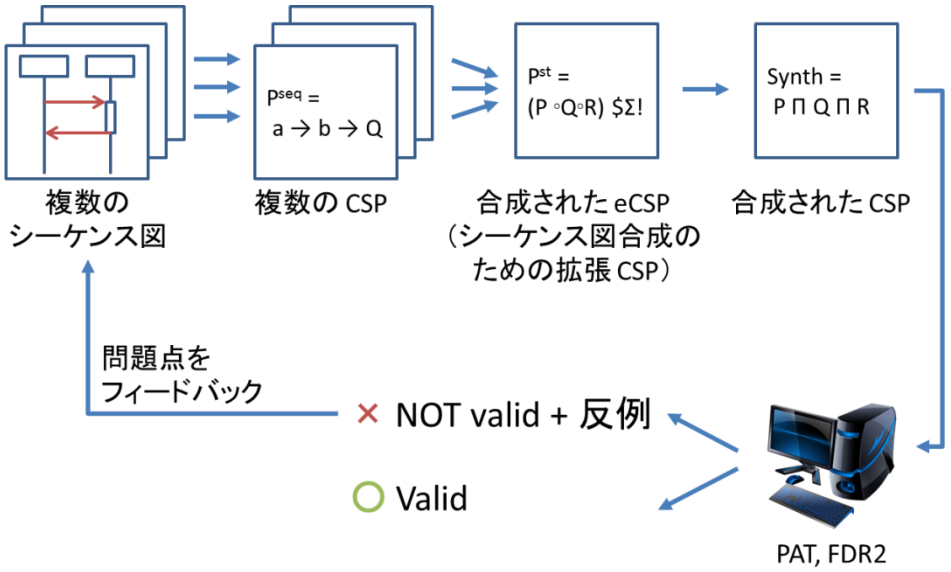
$$P ::= a \rightarrow P \mid P \,\square\, P \mid P \sqcap P \mid P \circ P \mid P\$X \mid PN$$

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

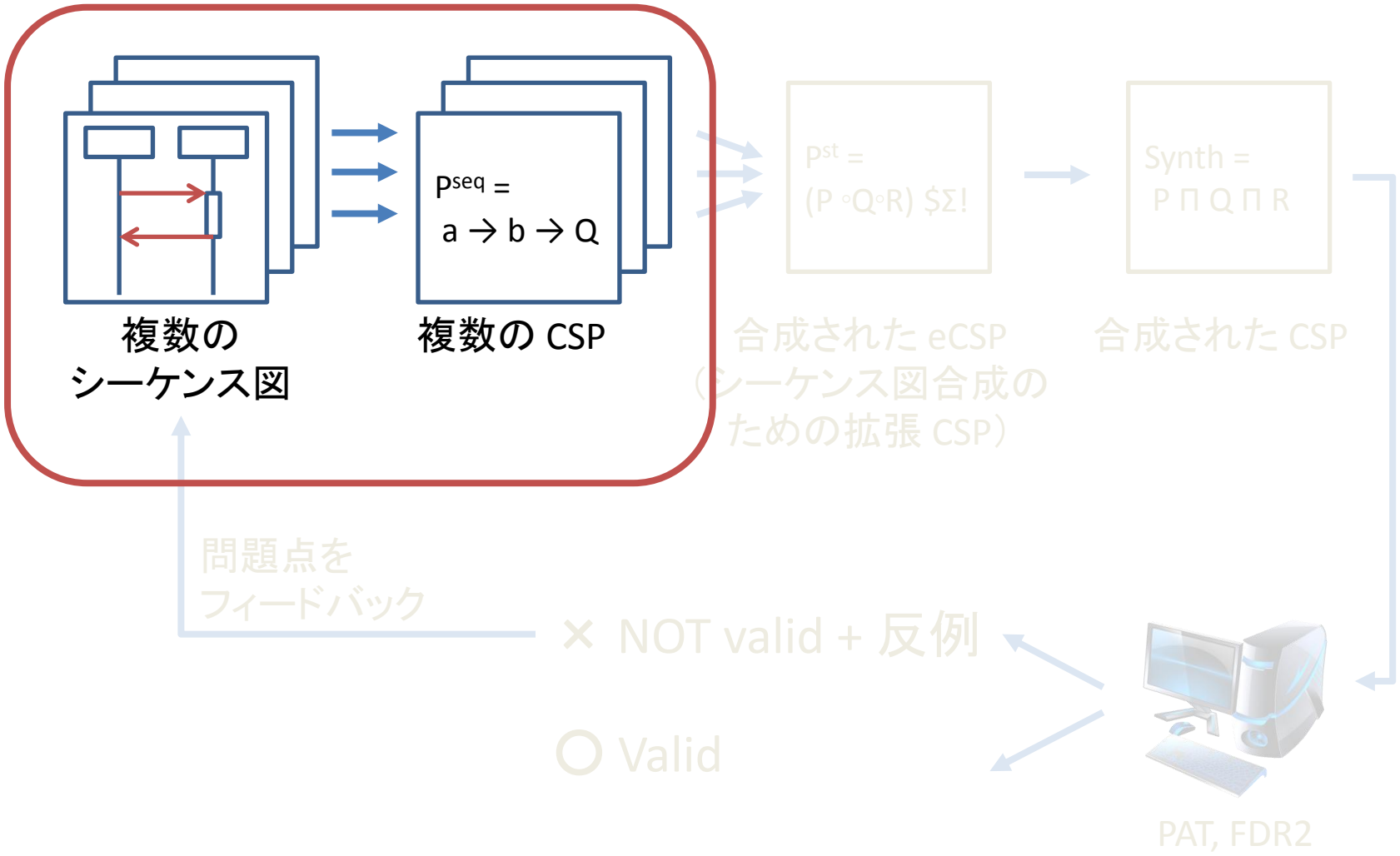
シーケンス図から CSP への変換



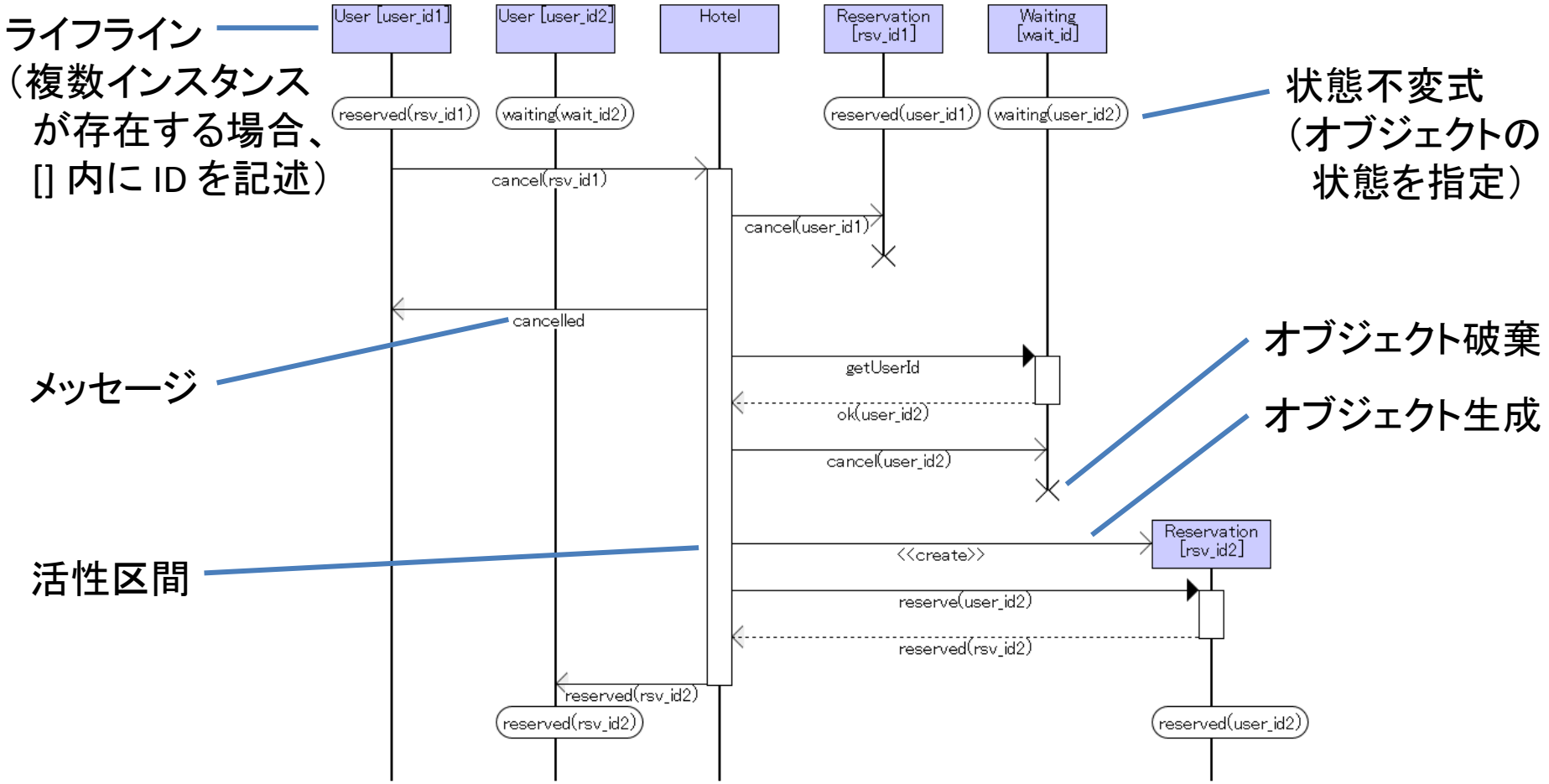
- はじめに
  - 背景、目的
  - プロセス代数
  - 研究の全体の概要
- CSPによるシーケンス図の検証
  - シーケンス図のCSP記述
  - eCSP（拡張CSP）によるプロセスの合成方法
  - eCSPからCSPへの変換方法
  - 変換ツールSDVerifier
  - 検証と反例解析
- より複雑なシーケンス図への対応
  - オブジェクト生成と破棄
  - 複数オブジェクト
- ケーススタディ
- 関連研究
- おわりに
  - まとめ
  - 成果



シーケンス図から CSP への変換

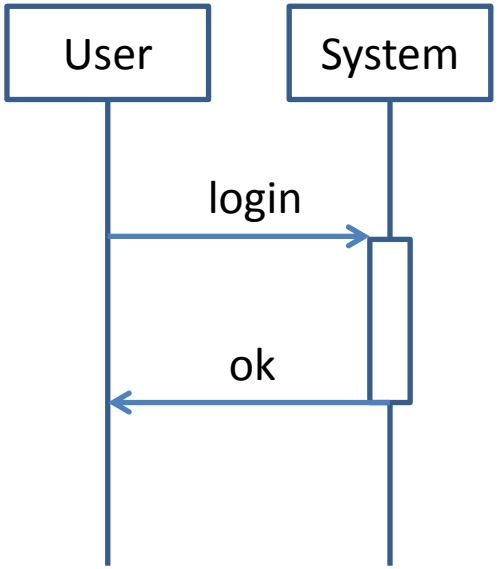


例：シーケンス図


ライフライン
（複数インスタンス
が存在する場合、
[] 内に ID を記述）
User [user_id1]
User [user_id2]
Hotel
Reservation [rsv_id1]
Waiting [wait_id]
状態不変式
（オブジェクトの
状態を指定）
reserved(rsv_id1)
waiting(wait_id2)
reserved(user_id1)
waiting(user_id2)
cancel(rsv_id1)
cancel(user_id1)
cancelled
メッセージ
getUserId
ok(user_id2)
cancel(user_id2)
オブジェクト破棄
オブジェクト生成
Reservation [rsv_id2]
<<create>>
活性区間
reserve(user_id2)
reserved(rsv_id2)
reserved(rsv_id2)
reserved(rsv_id2)
reserved(user_id2)

変換 (1/3):ライフライン、メッセージ

- ライフラインは CSP のプロセス、メッセージは CSP のイベントと対応させる



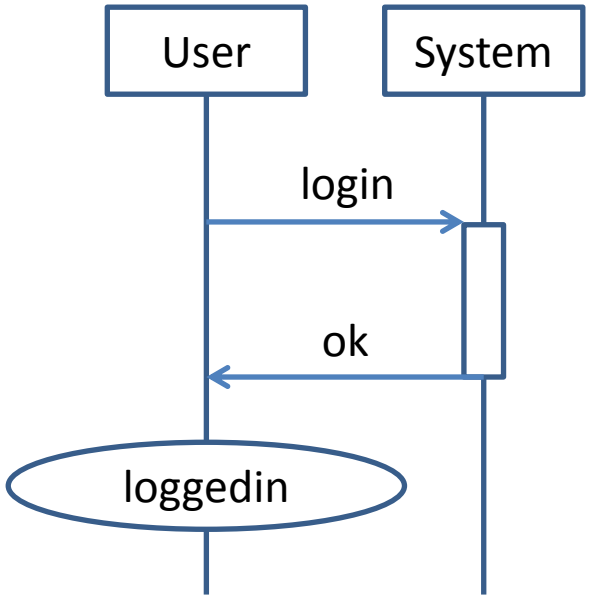
User = login → ok → User

System = login → ok → System

「a → P」はイベント a を実行した後プロセス P のようにふるまうプロセス

変換 (2/3) : 状態不変式

- 状態 S にあるオブジェクト O のふるまいを CSP のプロセス P(O, S) と対応させる



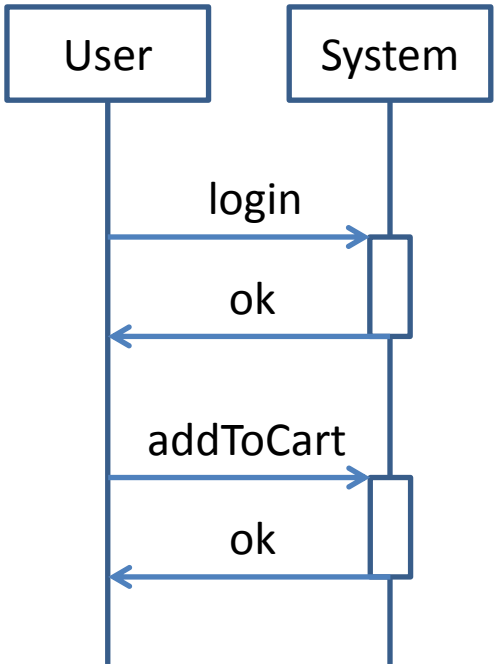
P(User, d) = login → ok → P(User, loggedin)

P(System, d) = login → ok → P(System, d)

※ 状態不変式が記述されていない箇所を標準状態（d）と呼ぶ

変換 (3/3): 活性区間

- 活性区間終了後の状態は、状態不変式が記述されていなければ標準状態とする



P(User, d) = login → ok →
addToCart → ok → P(User, d)

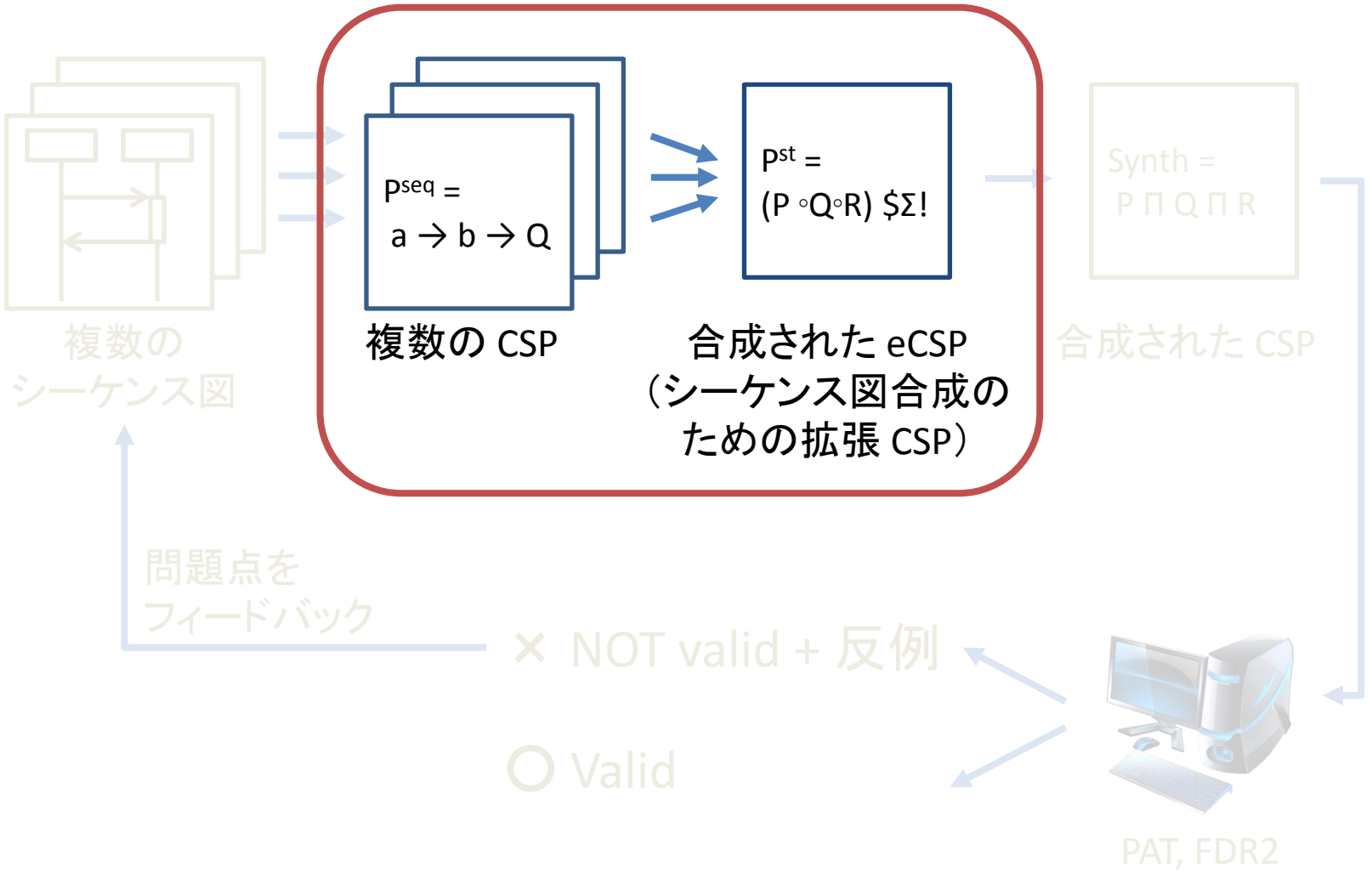
P(System, d_1) = login → ok → P(System, d)
P(System, d_2) = addToCart → ok → P(System, d)

この例では System の標準状態から始まるシナリオが 2 つあるため、「d_1」「d_2」として区別している。これらのシナリオは後述の合成手法で合成する

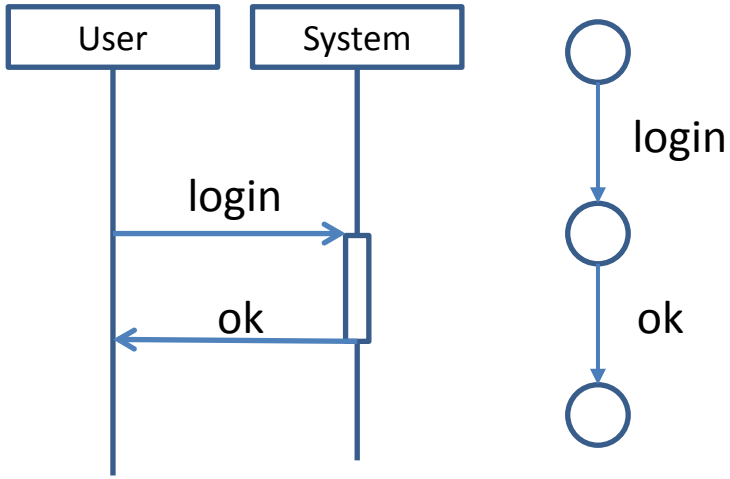
より複雑なシーケンス図のCSP への変換

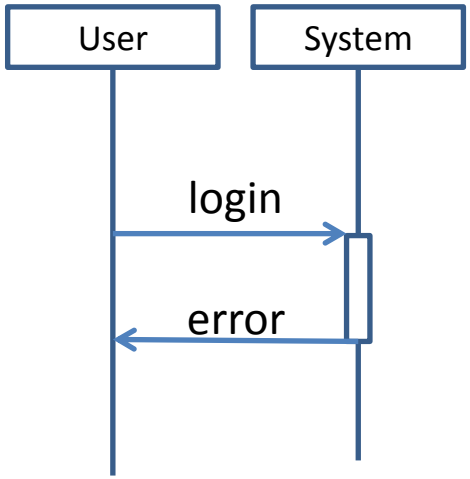
- 下記は後述
  - オブジェクトの生成と破棄
  - 複数オブジェクトを持つクラス
  - パラメータの受け渡し

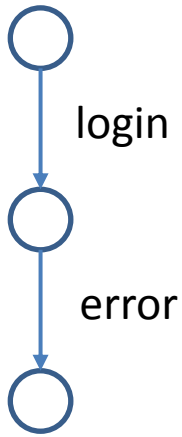
シーケンス図の合成


複数の
シーケンス図
Pseq =
a → b → Q
複数の CSP
Pst =
(P ◦Q◦R) $Σ!
合成された eCSP
（シーケンス図合成の
ための拡張 CSP）
Synth =
P Π Q Π R
合成された CSP
問題点を
フィードバック
× NOT valid + 反例
○ Valid
PAT, FDR2

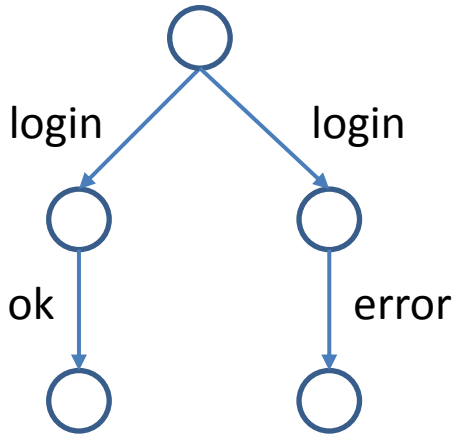
合成例:シーケンス図 (1/3)





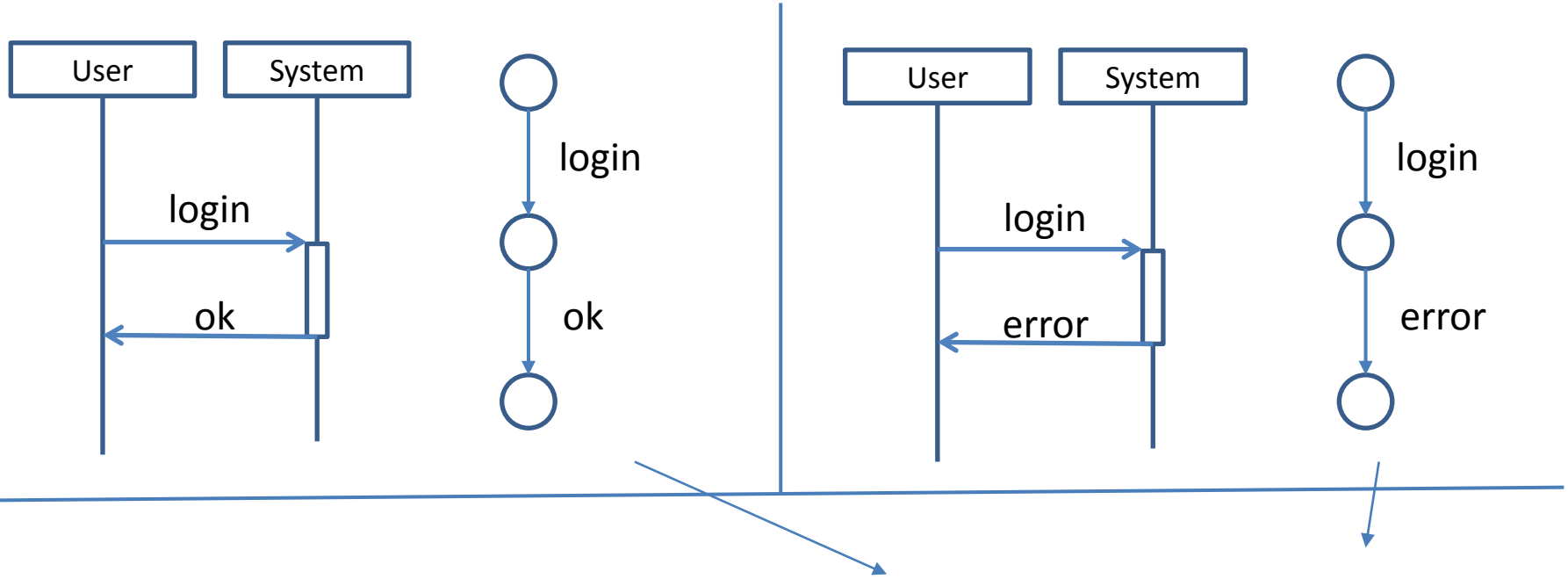


User のふるまいを CSP の外部選択 □ で合成した場合、login 送信時にイベントが選択されるため、ok や error を受信できない可能性がある

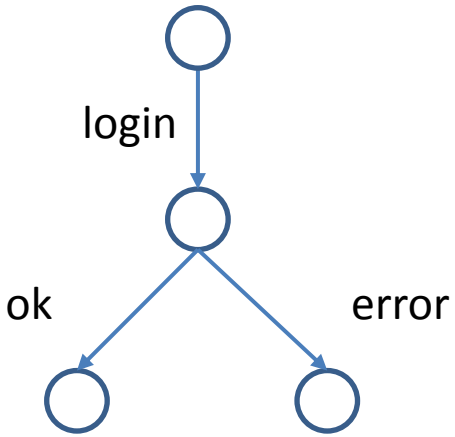


✕ (login → ok → …) □ (login → error → …)

合成例:シーケンス図 (2/3)



User が ok を受信するか error を受信するかは
login 送信後に選択される

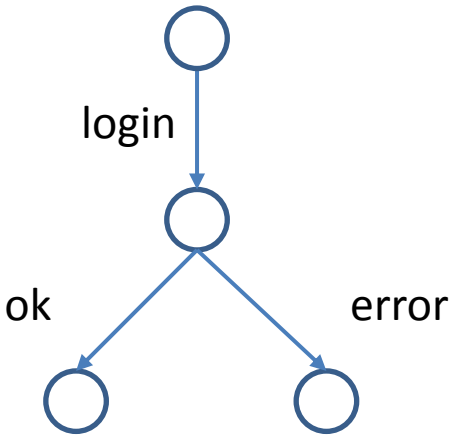


○ login → (ok → … □ error → …)

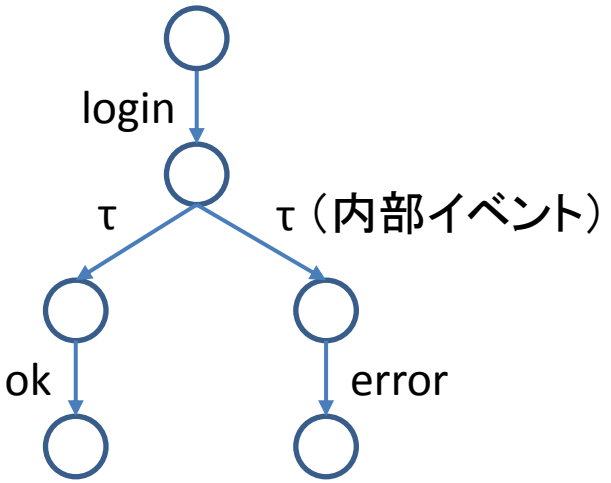
合成例:シーケンス図 (3/3)

- System オブジェクトが ok を送信するか error を送信するかを決定する。

P(User, d) =
login → (ok → … □ error → …)



P(System, d) =
login → (ok → … Π error → …)



- System オブジェクトは ok もしくは error を内部的に選択する（内部選択 **Π**）
  - 外部の環境に依存しない**非決定的**な選択
- User オブジェクトはどちらのイベントも受信できる（外部選択 **□**）
  - System オブジェクトの選択によって**決定的**に定まる

シーケンス図合成演算子

- シーケンス図合成のための演算子◦と$を定義する。
  - 合成したいプロセスをP, Q, Rとすれば、合成後のプロセスは(P◦Q◦R)$Σ!
    - ただしΣ!は送信イベントの集合
- シーケンス図マージ演算子◦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。
- 内部選択化演算子$は送信イベントを非決定的な選択に変換する

traces と failures

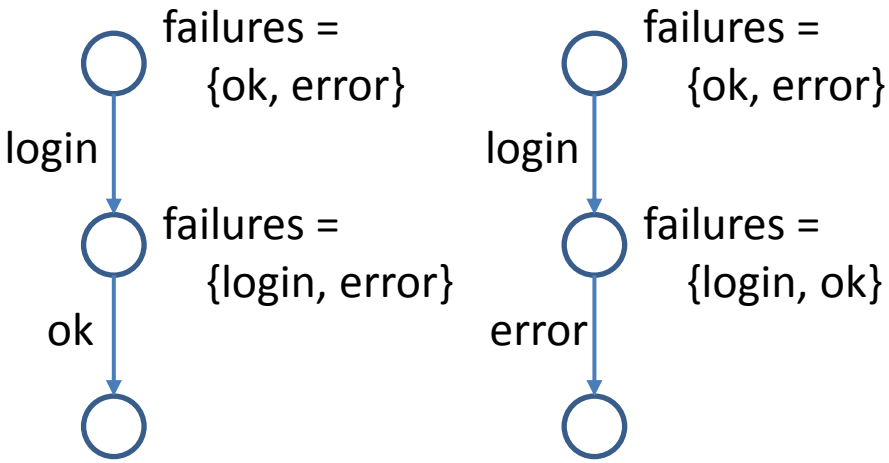
- CSP (失敗モデル) では、演算子の性質は traces 集合と failures 集合で表される
  - $s \in traces(P)$: プロセス P はトレース s を実行可能
  - $(s, X) \in failures(P)$: トレース s の実行後に X に含まれるイベントの実行は失敗する可能性がある

- トレース等価 $P =_T Q \Leftrightarrow traces(P) = traces(Q)$
- 失敗等価 $P =_F Q \Leftrightarrow traces(P) = traces(Q) \wedge failures(P) = failures(Q)$

定義：◦演算子

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

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



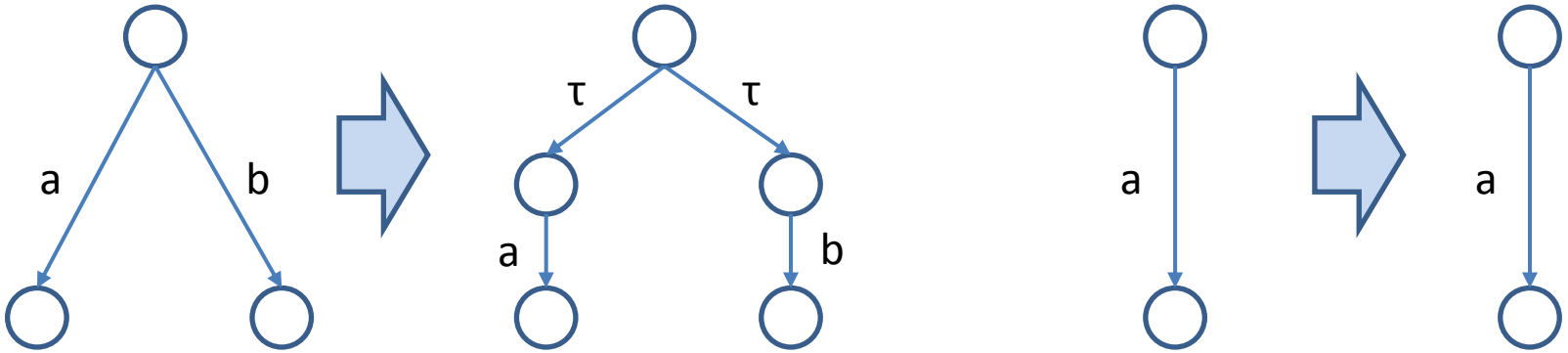
$$\begin{aligned} g(\langle\rangle, P) &= true \\ g(s^\frown\langle a\rangle, P) &= g(s,P) \wedge ((s,\{a\}) \notin failures(P)) \end{aligned}$$

- g(s, P) はプロセス P がトレース s を失敗せずに実行できることを表す
  - s が失敗しないならば、その後 P が失敗せずに実行できるイベントは P◦Q でも失敗せずに実行できる
- g(s, P) は s∈traces(P) に似ているが、もし g(s, P) を s∈traces(P) と定義すると詳細化関係・同値関係を保存しないため、CSP の演算子としては不十分（定理 3）

定義：$ 演算子

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

$$failures(P\$Z) = failures(P) \cup \{(s, X) |$$
$$\exists Y.((s, Y) \in failures(P) \wedge X \subseteq Y \cup Z),$$ — イベント集合 Z を失敗集合に加える
$$\exists a.(s^\frown\langle a\rangle \in traces(P) \wedge a \notin X))\}$$ —— 他に実行できるイベントが存在する場合のみ



- $ 演算子は送信イベントを失敗集合に加える。ただし、実行できるイベントが 1 つのみの場合は失敗しない
  - 実行できるイベントが 1 つのみかどうかは設計全体が与えられなければ定まらないため、◦演算子とは別に $ 演算子が必要

性質 : ∘, $ 演算子 (1/2)

- 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

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

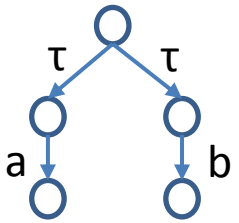
- 同一のメッセージが複数のシーケンス図に記述されていた場合、選択はそのメッセージの後で行われる（定理1）

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

性質 : ∘, $ 演算子 (2/2)

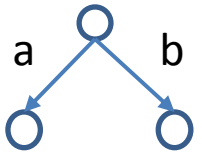
- 送受信可能なメッセージが複数存在する場合、送信オブジェクトがそのうちの 1 つを非決定的に選択する（定理2）
  - a. b がともに送信メッセージの場合

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



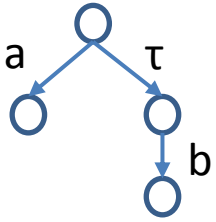
  - a, b がともに受信メッセージの場合

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



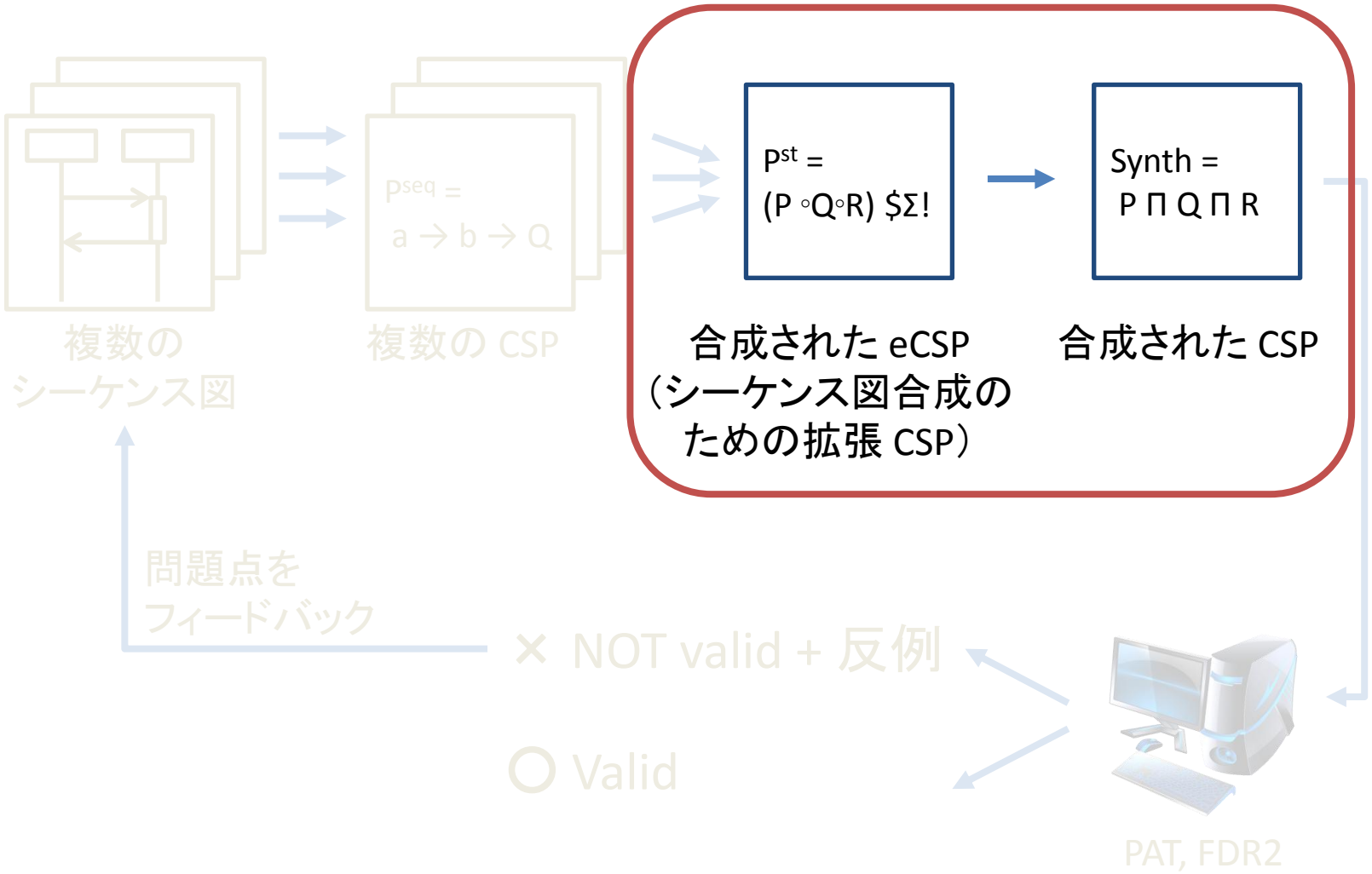
  - a が送信メッセージ、b が受信メッセージの場合

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



▷ はタイムアウト演算子。送信は失敗の可能性があるが受信は失敗しない

標準CSP への変換


複数の
シーケンス図
Pseq =
a → b → Q
複数の CSP
Pst =
(P ◦Q◦R) $Σ!
合成された eCSP
（シーケンス図合成の
ための拡張 CSP）
Synth =
P Π Q Π R
合成された CSP
問題点を
フィードバック
× NOT valid + 反例
○ Valid
PAT, FDR2

定義：CSP への変換関数

- 以下の等式を満たす Synth を計算する
  - Synth (O, $\boldsymbol{S}$) $=_F$ ($\bigcirc_{S \in \boldsymbol{S}}$@P(O, S))$Σ!
- Synth は次のように定義できる

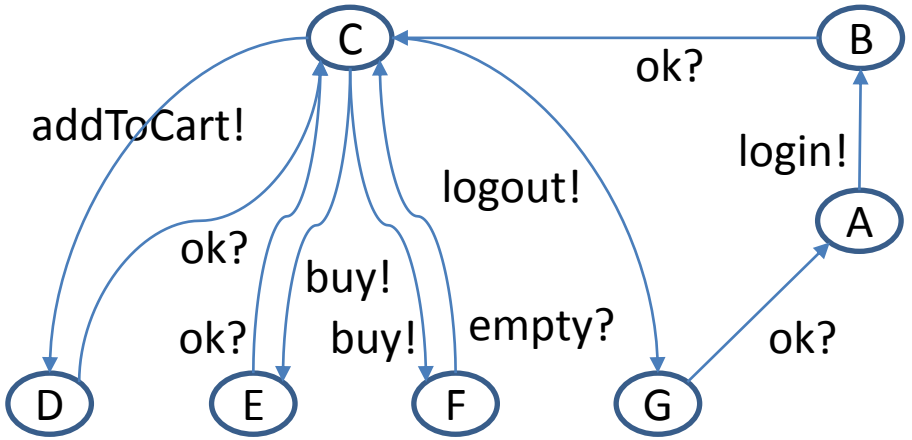
$$
\begin{array}{ll}
(1) & (A!_D(O,\mathcal{S}) = \phi) \Rightarrow \\
 & Synth_D(O,\mathcal{S}) = \ \Box_{a \in A?_D(O,\mathcal{S})} @a \rightarrow Synth_D(O, N_D(O,\mathcal{S},a)) \\
(2) & (A!_D(O,\mathcal{S}) \neq \phi \wedge A?_D(O,\mathcal{S}) = \phi) \Rightarrow \\
 & Synth_D(O,\mathcal{S}) = \sqcap_{a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

  - A! はその状態における送信イベント集合、A? はその状態における受信イベント集合、N は a イベント発生後の遷移先となりうる状態の集合
- モデル検査ツールは Synth の定義を直接検証できる

変換例 (1/6)

Synth(O, {A}) = A $ {login!, addToCart!, buy!, logout!}

A = login! → B
B = ok? → C
C = (addToCart! → D) ◦ (buy! → E) ◦ (buy! → F) ◦ (logout! → G)
D = ok? → C
E = ok? → C
F = empty? → C
G = ok? → A

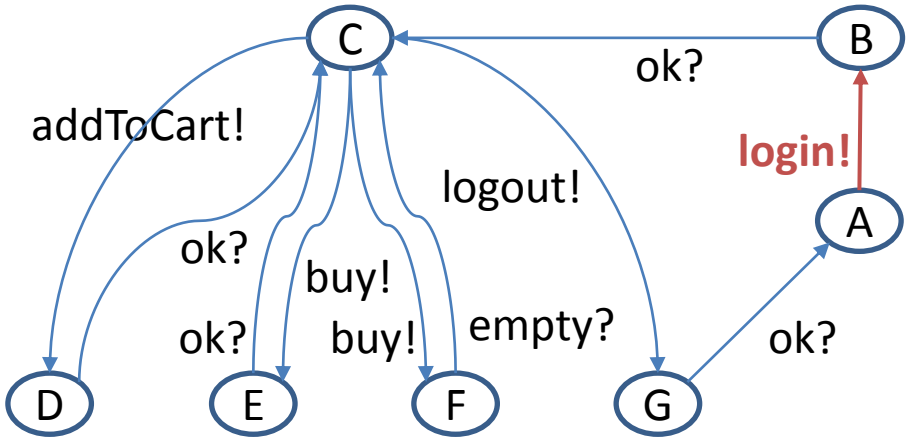


- この例では、送受信を明確にするため、送信イベントに「!」、受信イベントに「?」をつける

変換例 (2/6)

Synth(O, {A}) = login! → Synth(O, {B})

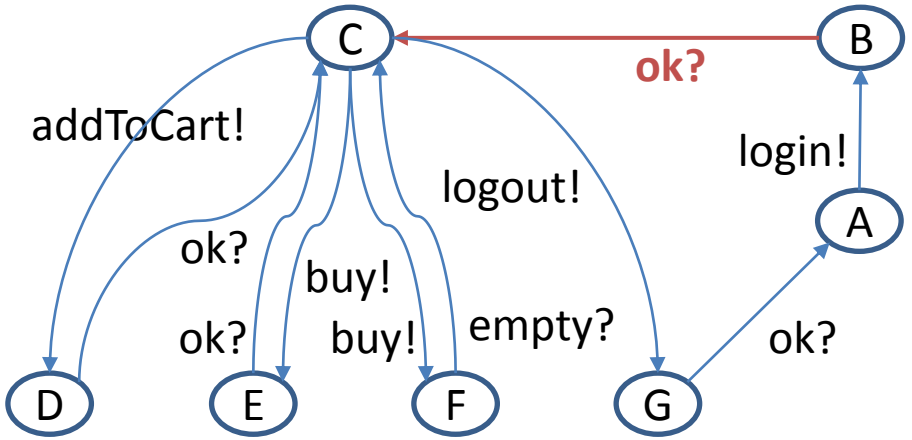
A = login! → B
B = ok? → C
C = (addToCart! → D) ◦ (buy! → E) ◦ (buy! → F) ◦ (logout! → G)
D = ok? → C
E = ok? → C
F = empty? → C
G = ok? → A



変換例 (3/6)

Synth(O, {A}) = login! → Synth(O, {B})
Synth(O, {B}) = ok? → Synth(O, {C})

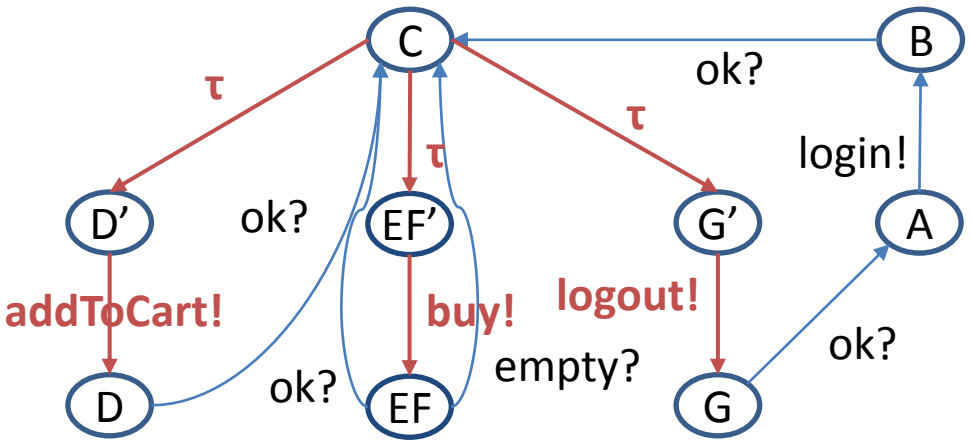
B = ok? → C
C = (addToCart! → D) ◦ (buy! → E) ◦ (buy! → F) ◦ (logout! → G)
D = ok? → C
E = ok? → C
F = empty? → C
G = ok? → A



変換例 (4/6)

Synth(O, {A}) = login! → Synth(O, {B})
Synth(O, {B}) = ok? → Synth(O, {C})
Synth(O, {C}) = addToCart! → Synth(O, {D})
Π buy! → Synth(O, {E, F})
Π logout! → Synth(O, {G})

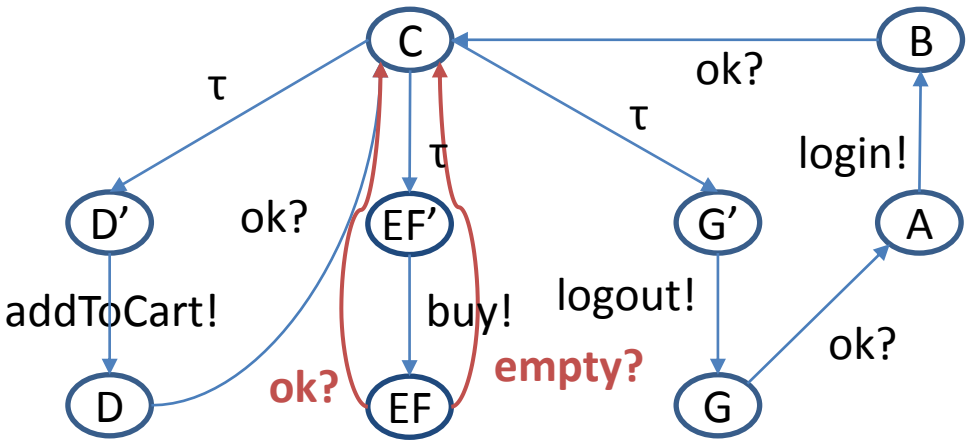
C = (addToCart! → D) ◦ (buy! → E) ◦ (buy! → F) ◦ (logout! → G)
D = ok? → C
E = ok? → C
F = empty? → C
G = ok? → A



変換例 (5/6)

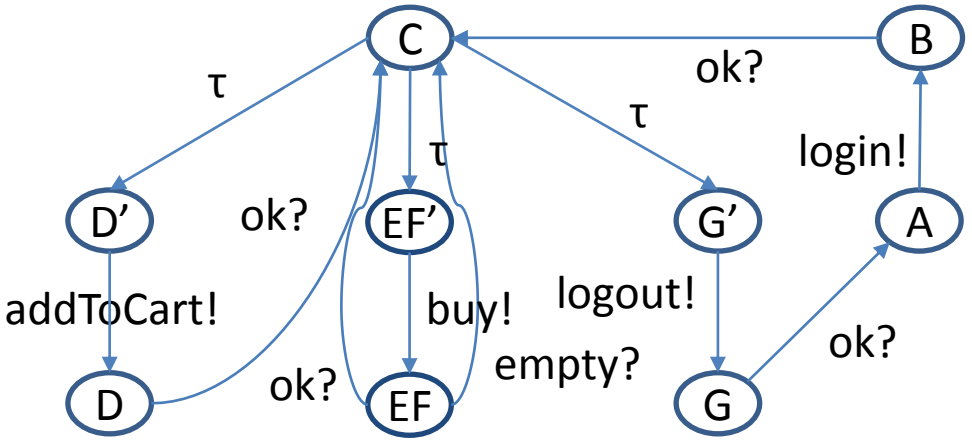
Synth(O, {A}) = login! → Synth(O, {B})
Synth(O, {B}) = ok? → Synth(O, {C})
Synth(O, {C}) = addToCart! → Synth(O, {D})
Π buy! → Synth(O, {E, F})
Π logout! → Synth(O, {G})
Synth(O, {D}) = ok? → Synth(O, {C})
Synth(O, {E, F}) = (ok? → Synth(O, {C})) □ (empty? → Synth(O, {C}))

E = ok? → C
F = empty? → C
G = ok? → A



変換例 (6/6)

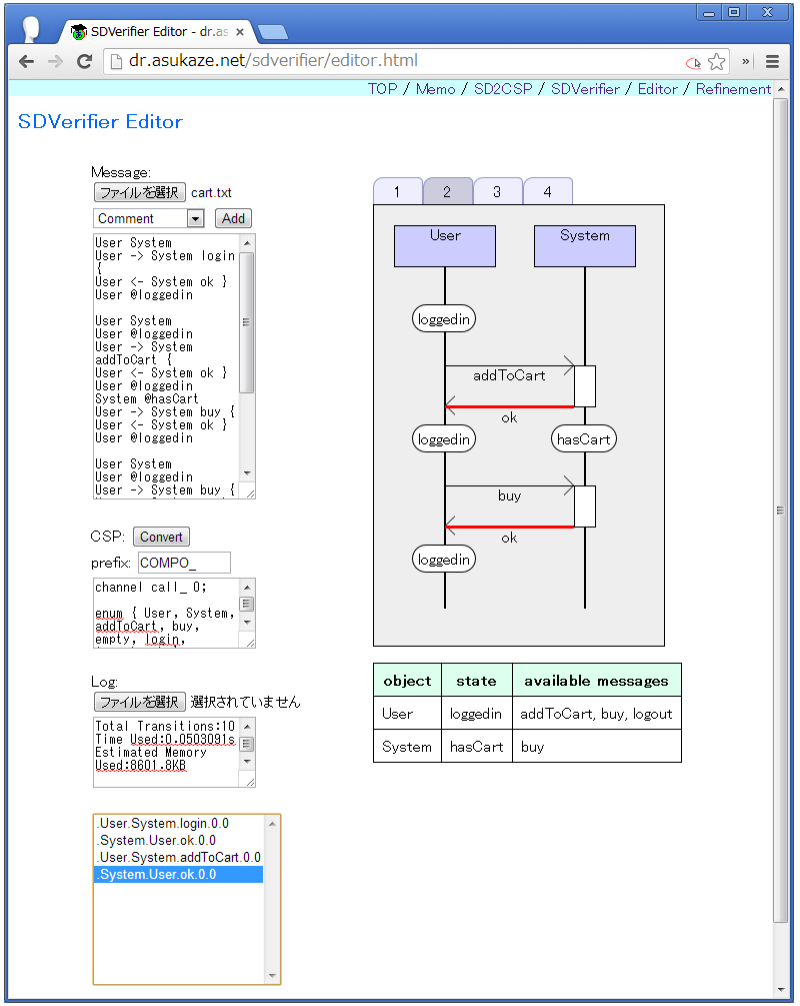
Synth(O, {A}) = login! → Synth(O, {B})
Synth(O, {B}) = ok? → Synth(O, {C})
Synth(O, {C}) = addToCart! → Synth(O, {D})
Π buy! → Synth(O, {E, F})
Π logout! → Synth(O, {G})
Synth(O, {D}) = ok? → Synth(O, {C})
Synth(O, {E, F}) = (ok? → Synth(O, {C})) □ (empty? → Synth(O, {C}))
Synth(O, {G}) = ok? → Synth(O, {A})



- 変換後の状態は変換前の状態の部分集合と対応
- 有限時間で変換できる

SDVerifier ツール

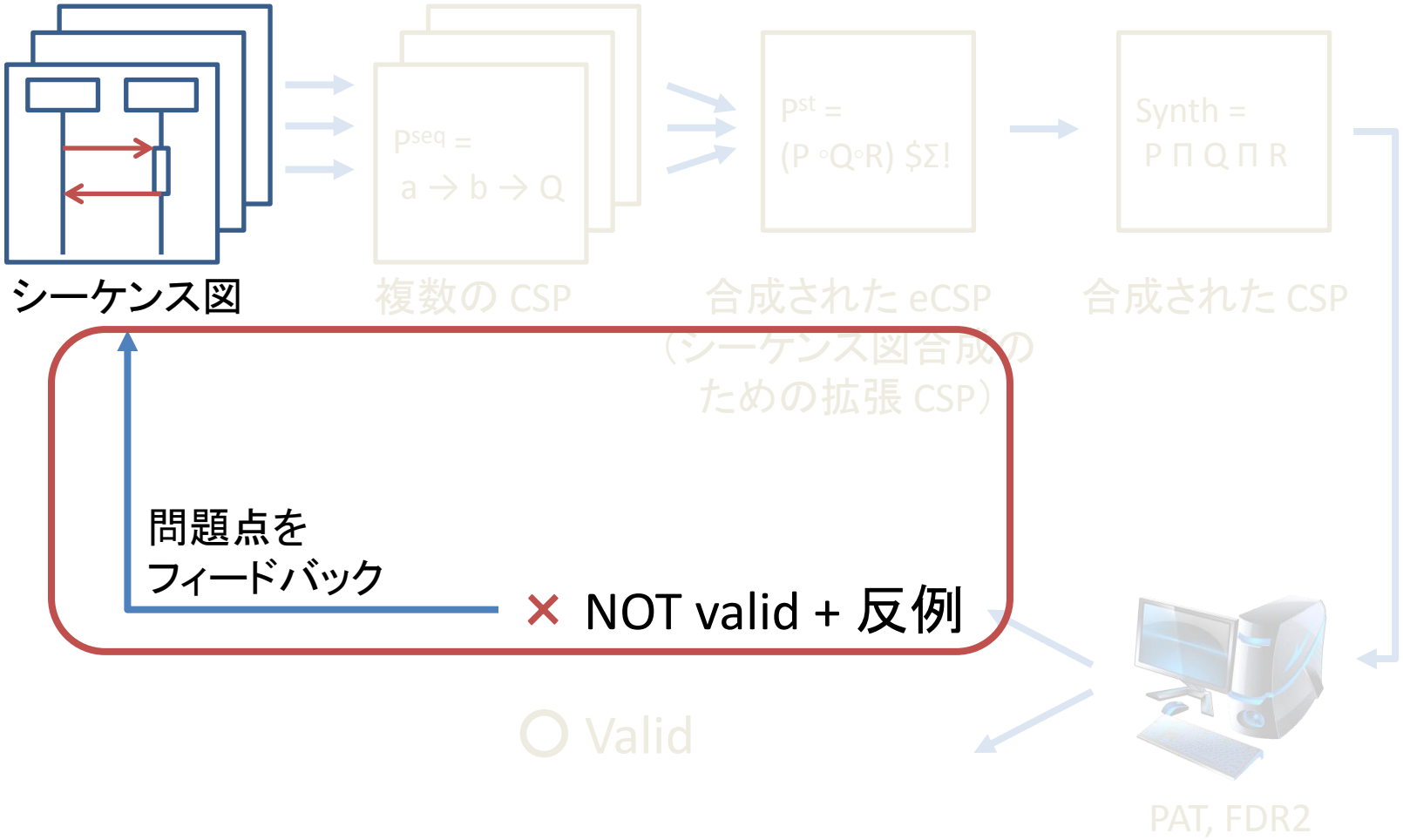
- シーケンス図編集
- CSP 出力
  - 提案手法を用いてシーケンス図をPAT ツールに入力可能な CSP に変換
- 反例解析
  - PAT ツールの生成する反例を逆変換して図示



PAT を用いた検証

- デッドロック検証
  - 変換された CSP がデッドロックしないことを確認
    - 到達しうるシステムの状態がシーケンス図で網羅されていない場合に変換された CSP はデッドロックする
- 詳細化検証
  - 抽象モデルと詳細モデルが与えられた時に、ふるまいが変わっていないことを確認

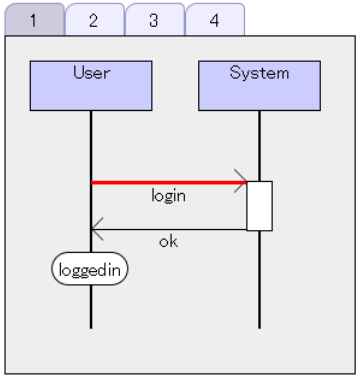
反例解析


シーケンス図
Pseq =
a → b → Q
複数の CSP
Pst =
(P ◦Q◦R) $Σ!
合成された eCSP
（シーケンス図合成の
ための拡張 CSP）
Synth =
P Π Q Π R
合成された CSP
問題点を
フィードバック
× NOT valid + 反例
○ Valid
PAT, FDR2

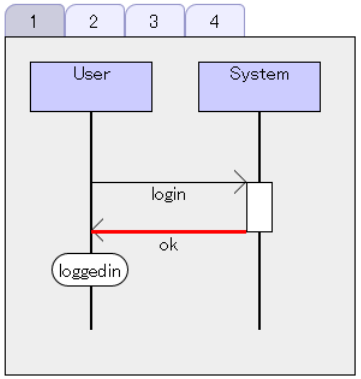
反例解析

PAT ツールの生成する反例を逆変換し、元のシーケンス図上で図示できる

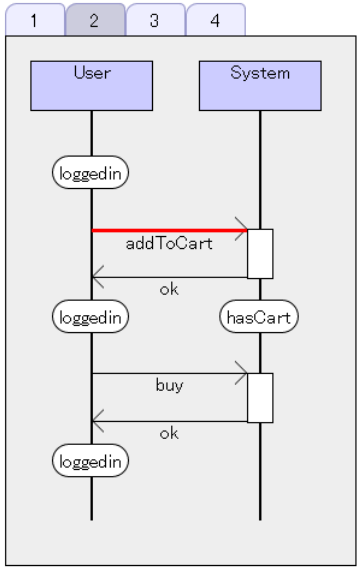
<init -> call_.User.System.**login**.0.0 -> call_.System.User.**ok**.0.0 -> [int_choice] ->
call_.User.System.**addToCart**.0.0 -> call_.System.User.**ok**.0.0 -> [int_choice]>



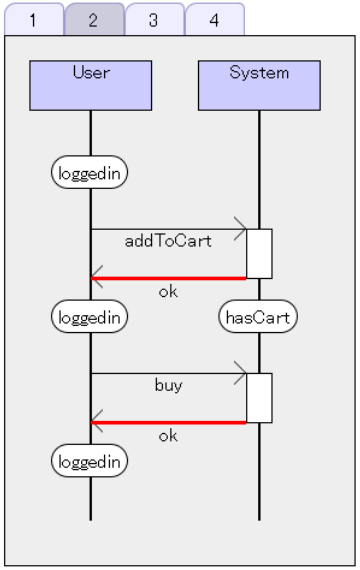
object	state	available messages
User	i2	ok
System	i1	ok



object	state	available messages
User	loggedin	logout, addToCart, buy
System	default	logout, login, addToCart, buy

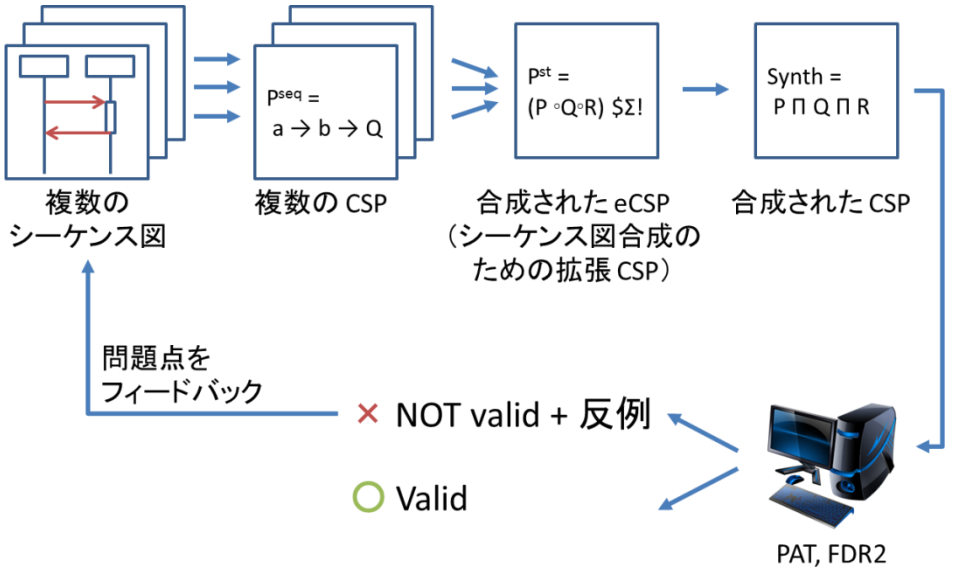


object	state	available messages
User	i4	ok
System	i3	ok

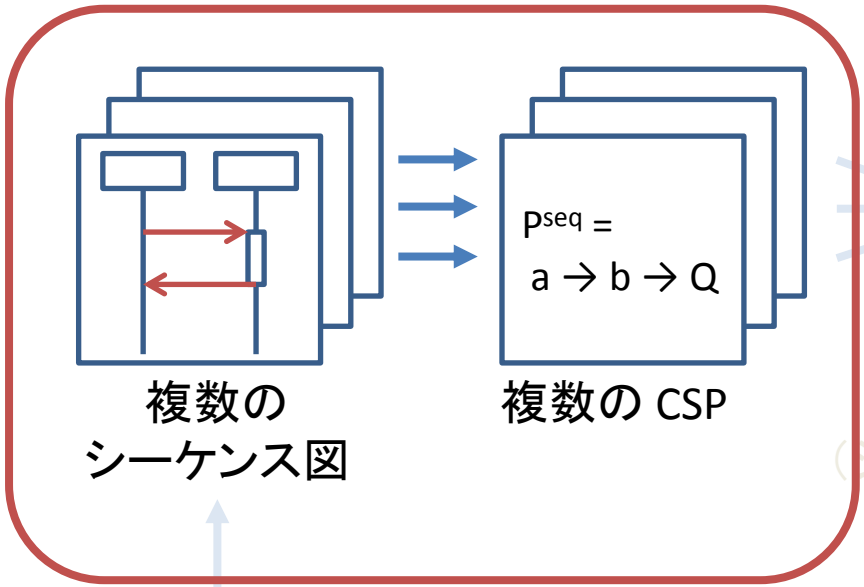


object	state	available messages
User	loggedin	logout, addToCart, buy
System	hasCart	buy

- はじめに
  - 背景、目的
  - プロセス代数
  - 研究の全体の概要
- CSPによるシーケンス図の検証
  - シーケンス図のCSP記述
  - eCSP（拡張CSP）によるプロセスの合成方法
  - eCSPからCSPへの変換方法
  - 変換ツールSDVerifier
  - 検証と反例解析
- **より複雑なシーケンス図への対応**
  - オブジェクト生成と破棄
  - 複数オブジェクト
- ケーススタディ
- 関連研究
- おわりに
  - まとめ
  - 成果



より複雑なシーケンス図の CSP への変換



Pst =
(P ◦Q◦R) $Σ!

合成された eCSP
（シーケンス図合成の
ための拡張 CSP）

Synth =
P Π Q Π R

合成された CSP

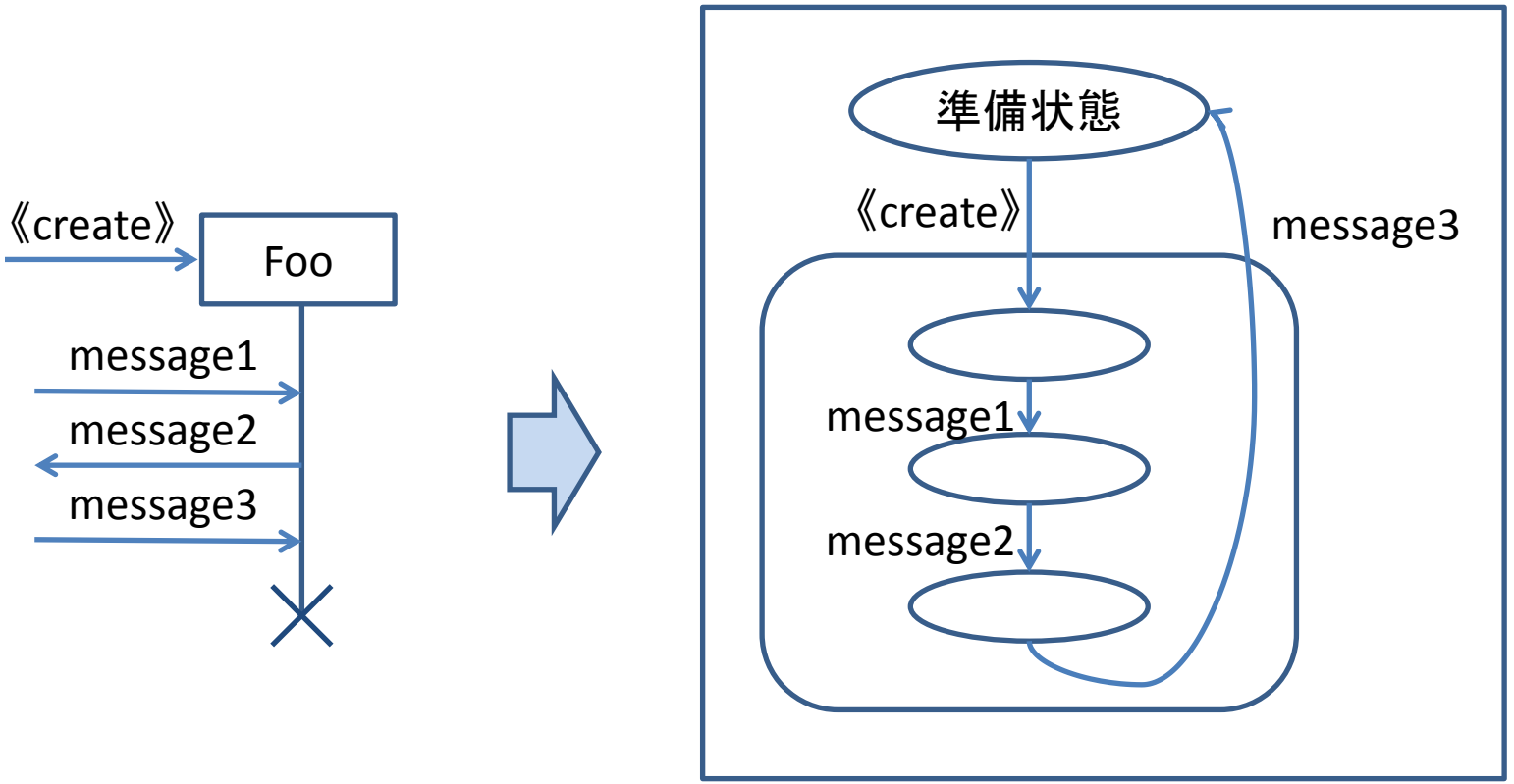
問題点を
フィードバック

× NOT valid + 反例

○ Valid

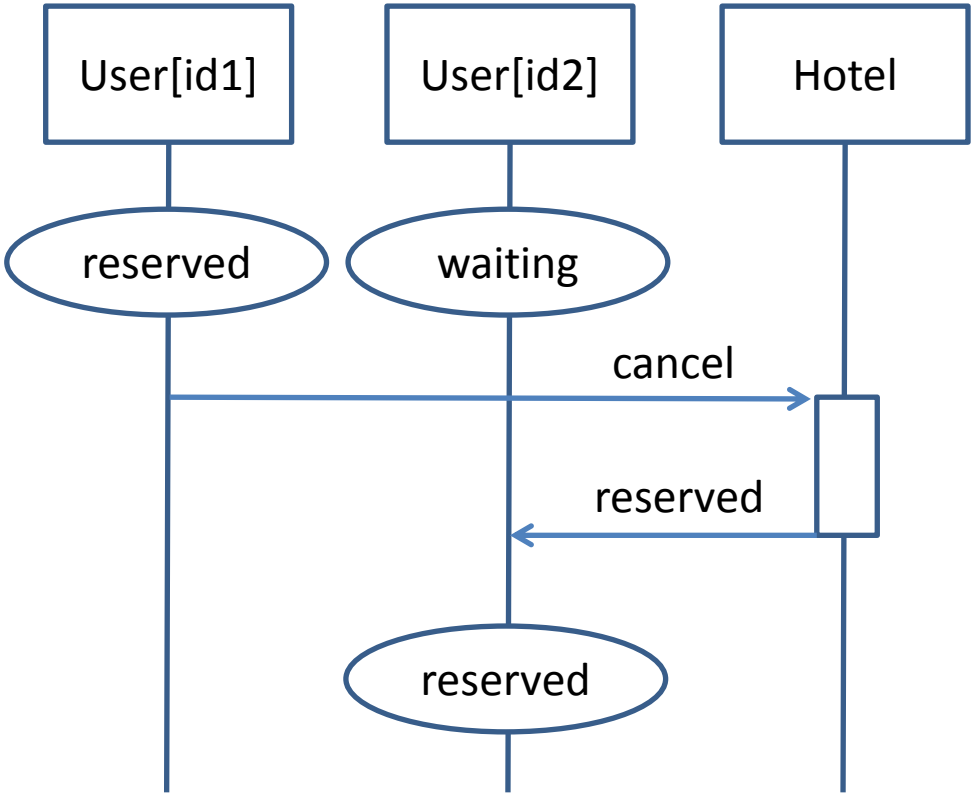
PAT, FDR2

オブジェクトの生成と破棄



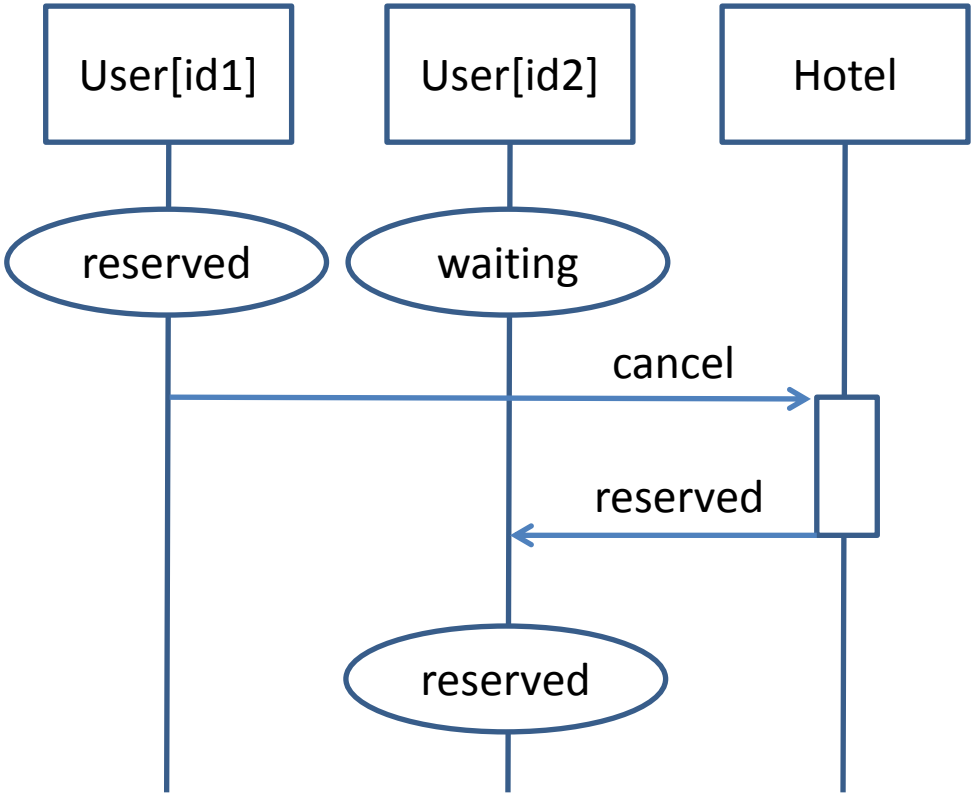
- 生成前・破棄後のオブジェクトは「準備状態」という特別な状態にあるとみなし、あらかじめ準備状態にある CSP のプロセスを用意しておくことで検証を可能とする
- 準備状態のオブジェクトは《create》メッセージを受け取ると準備状態でない状態に遷移し、オブジェクト破棄に到達すると準備状態に戻る

複数オブジェクトをもつクラス (1/2)



- ライフラインに [] と記述することでそのクラスのインスタンスの 1 つであることを表す
- User[id1] と User[id2] は同じふるまいをする
  （User[id2] は reserved メッセージの受信後 cancel メッセージを送信できる）
- User の最大数はこのシーケンス図からだけではわからない

複数オブジェクトをもつクラス (2/2)

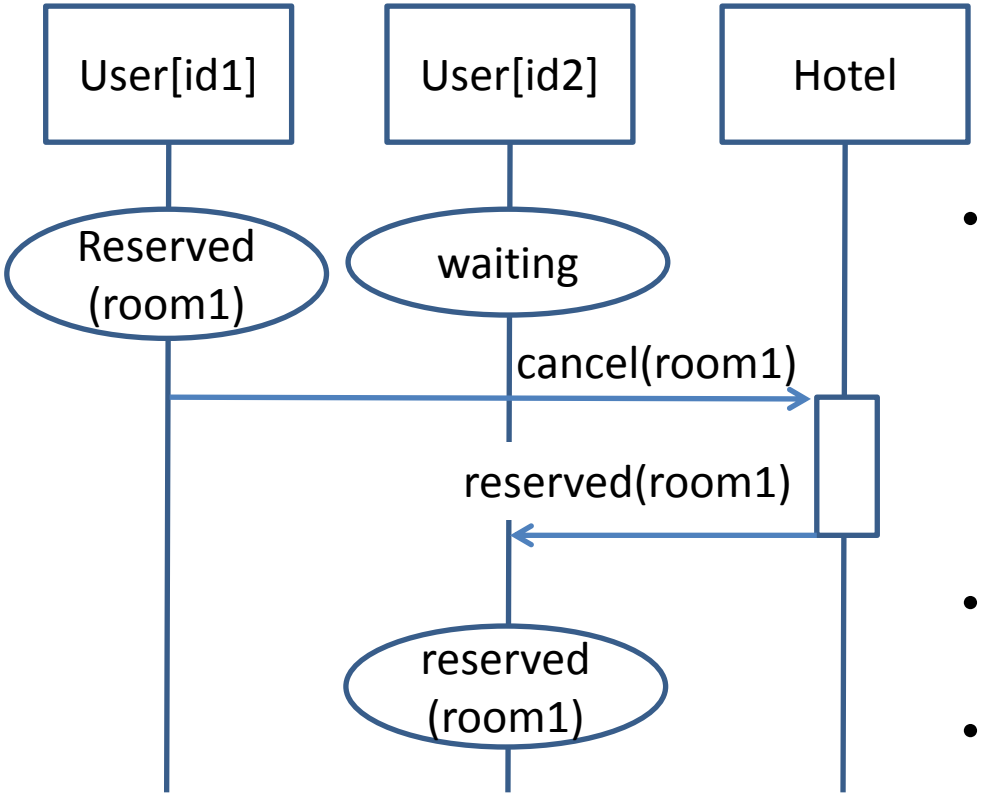


#count User 3

検証用の設定として
オブジェクトの最大数を指定

「User(0) || User(1) || User(2) || Hotel」というプロセスが生成される。
ただし、User(x) は User[id1] と User[id2] を合成して得られる CSP のプロセス。
（|| はプロセスの並行合成）

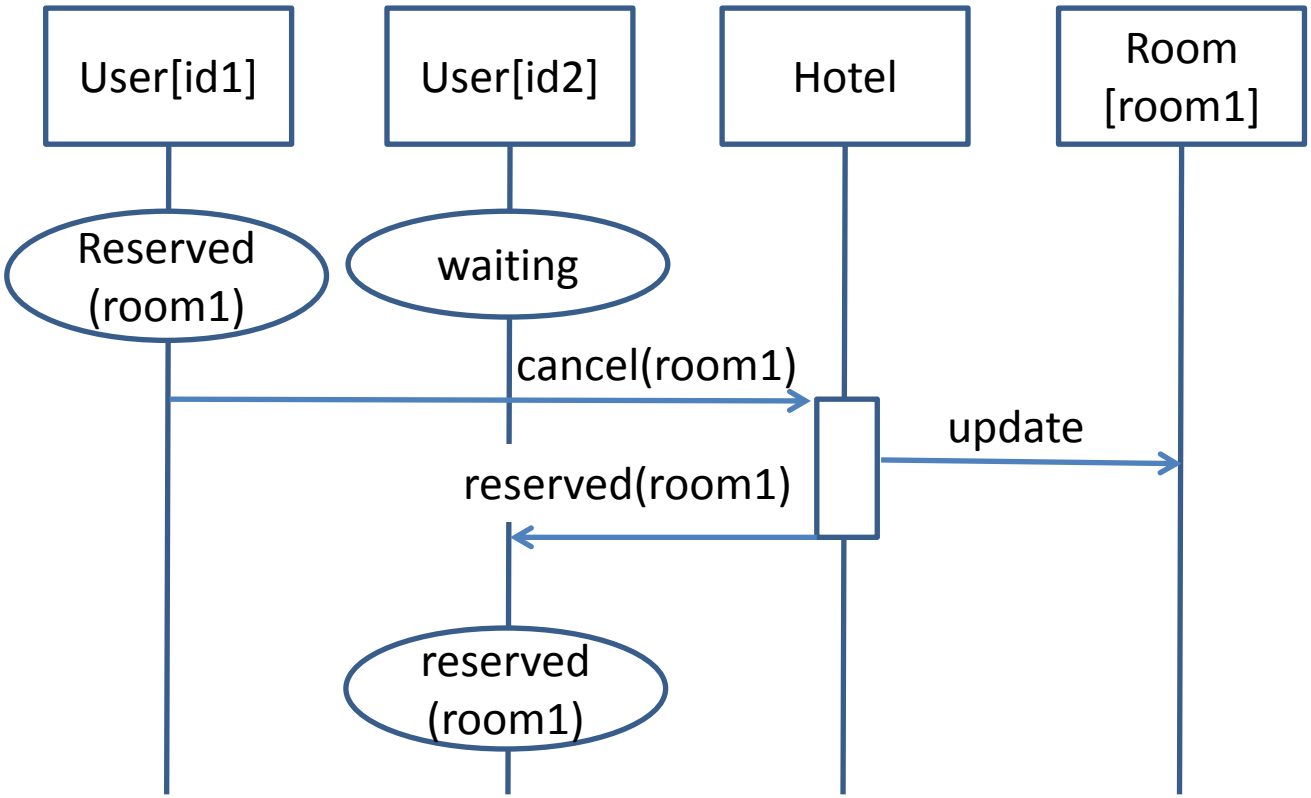
パラメータの受け渡し (1/2)



- メッセージのパラメータによるデータの受け渡しに対応
（CSP では「cancel.room1」のようなドット区切りのイベントに変換できる）

- 受信したデータは活性区間の間のみ保持される
- 状態不変式のパラメータに指定されたデータは次の活性区間の完了まで保持される

パラメータの受け渡し (2/2)

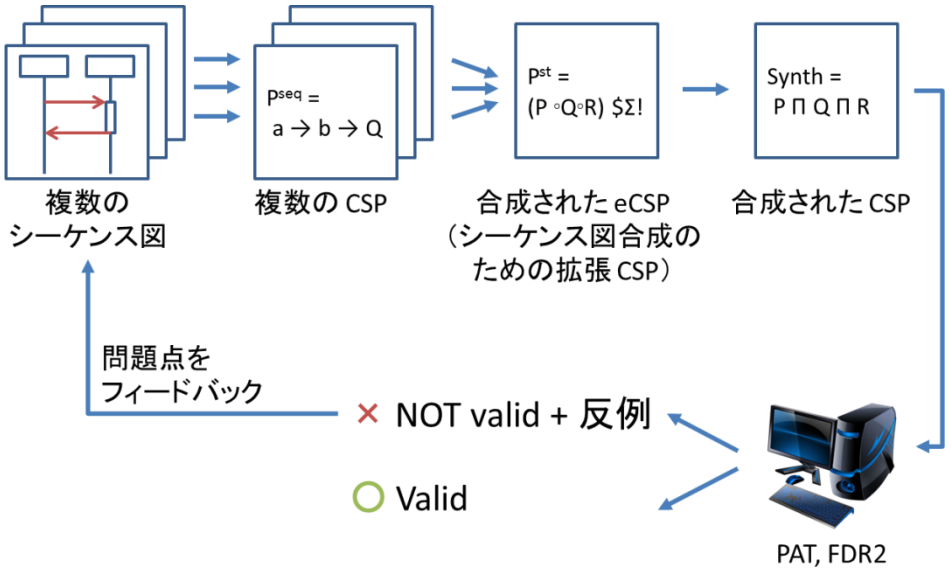


- Hotel は update 送信時に room1 の値を知っているので、ID の一致する Room のみがメッセージを受信できる
- Hotel は reserved 送信時に id2 の値を知らないので、任意の User が reserved を受信できる

定義：パラメータによるふるまいの決定

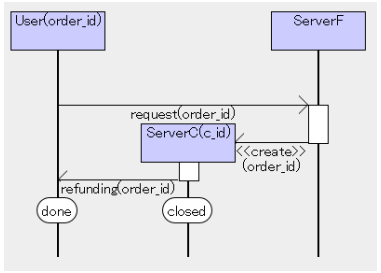
- オブジェクトがメッセージを受信した場合：
  - 送信オブジェクトの ID およびメッセージのパラメータがメモリに加えられる
- オブジェクトがメッセージを送信する場合：
  - もし受信オブジェクトの ID がメモリに存在するならば、そのIDのオブジェクトのみがメッセージを受信できる
  - もし受信オブジェクトの ID がメモリに存在しないならば、受信オブジェクトはそのメッセージを受信可能なオブジェクトのうち1 つが自動的に選択される
- 活性区間の完了時に、その活性区間中で記憶されたメモリは破棄される
  - ただし、活性区間の直後にパラメータを持つ状態不変式がある場合、状態不変式の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

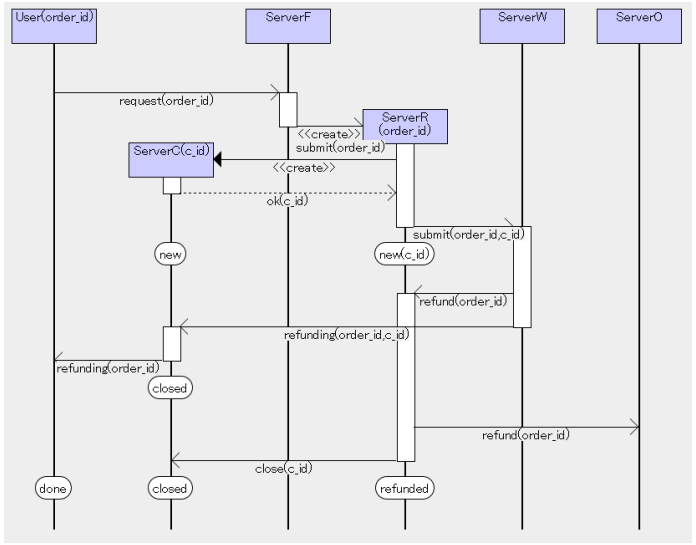
- はじめに
  - 背景、目的
  - プロセス代数
  - 研究の全体の概要
- CSPによるシーケンス図の検証
  - シーケンス図のCSP記述
  - eCSP（拡張CSP）によるプロセスの合成方法
  - eCSPからCSPへの変換方法
  - 変換ツールSDVerifier
  - 検証と反例解析
- より複雑なシーケンス図への対応
  - オブジェクト生成と破棄
  - 複数オブジェクト
- ケーススタディ
- 関連研究
- おわりに
  - まとめ
  - 成果



ケーススタディ１

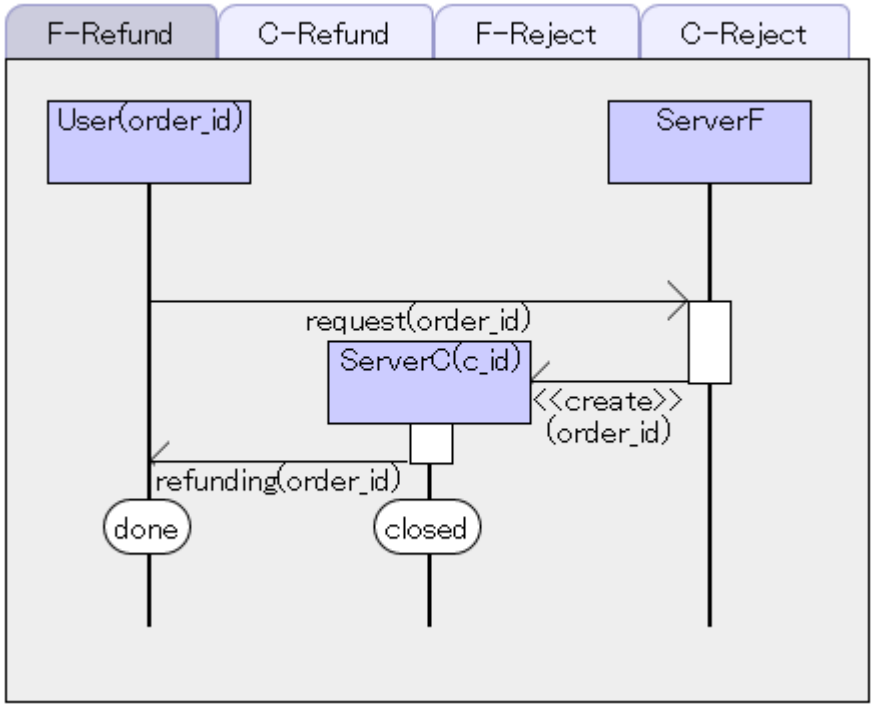
- 検証対象：
  - ECサイトのカスタマーサポートシステム（実システム）
- 検証項目：
  - デッドロックしない
  - 抽象モデルと詳細モデルはユーザーから見えるイベント送受信の順序が等しい



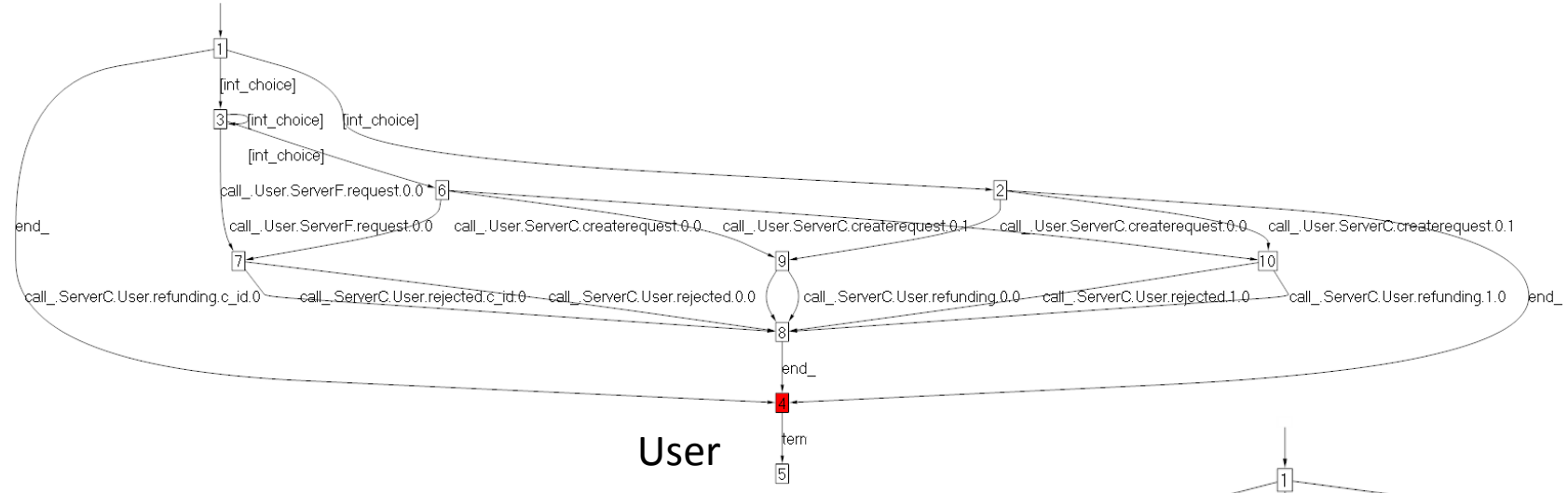


抽象モデル

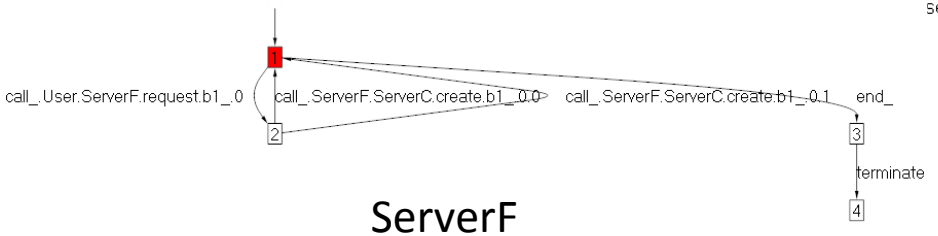
抽象モデルでは直接ユーザーとメッセージ送受信をするサーバのみをモデル化し、ユーザーから見えるふるまいを記述した



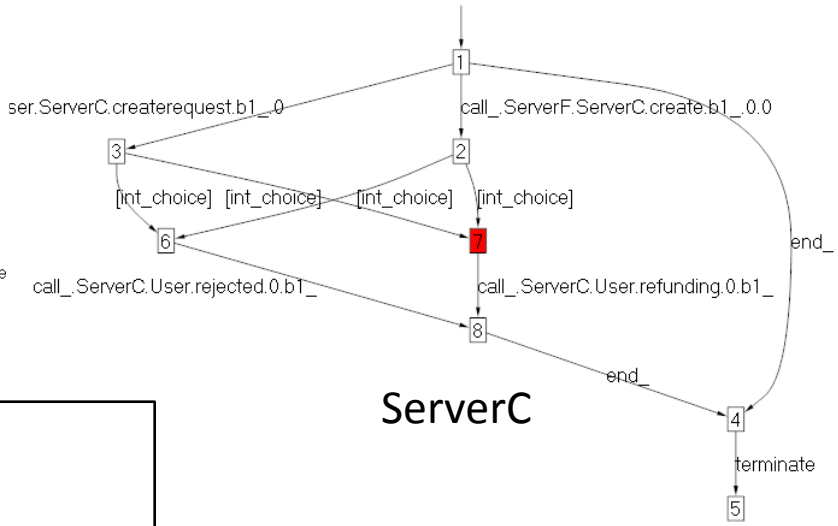
合成された抽象モデルのCSP



User



ServerF

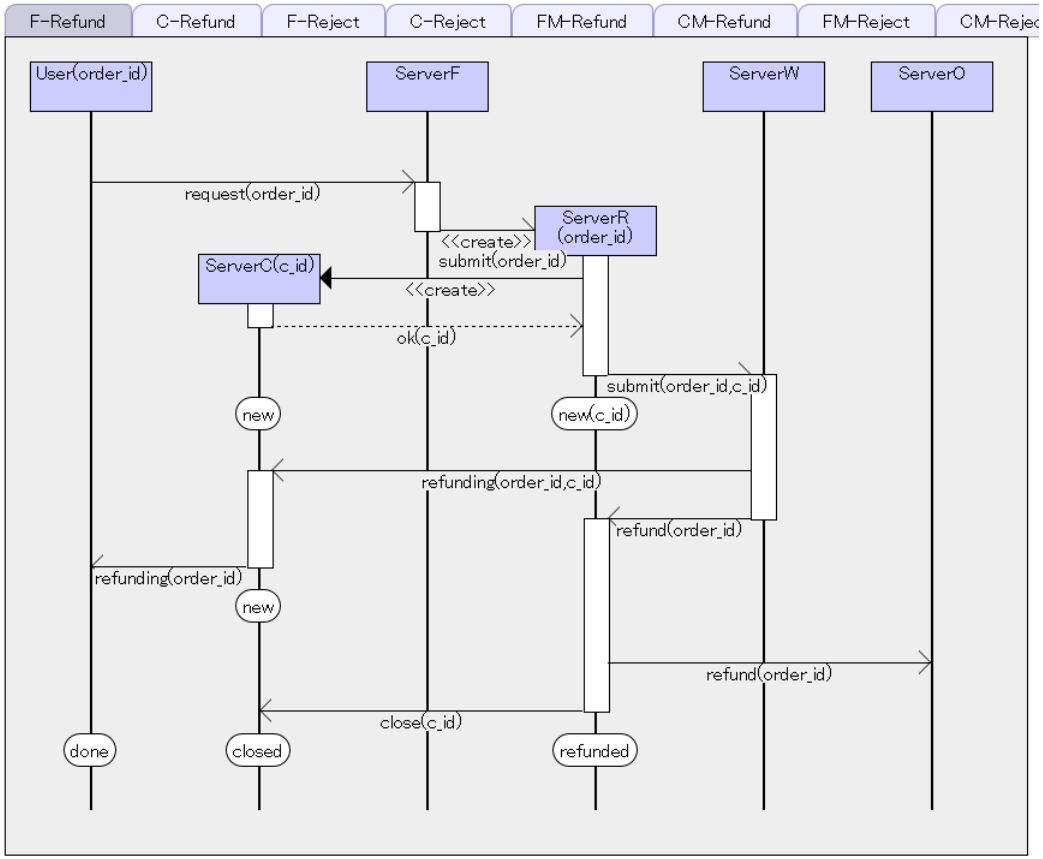


ServerC

********Verification Result********
The Assertion (System_C_() deadlockfree) is VALID.

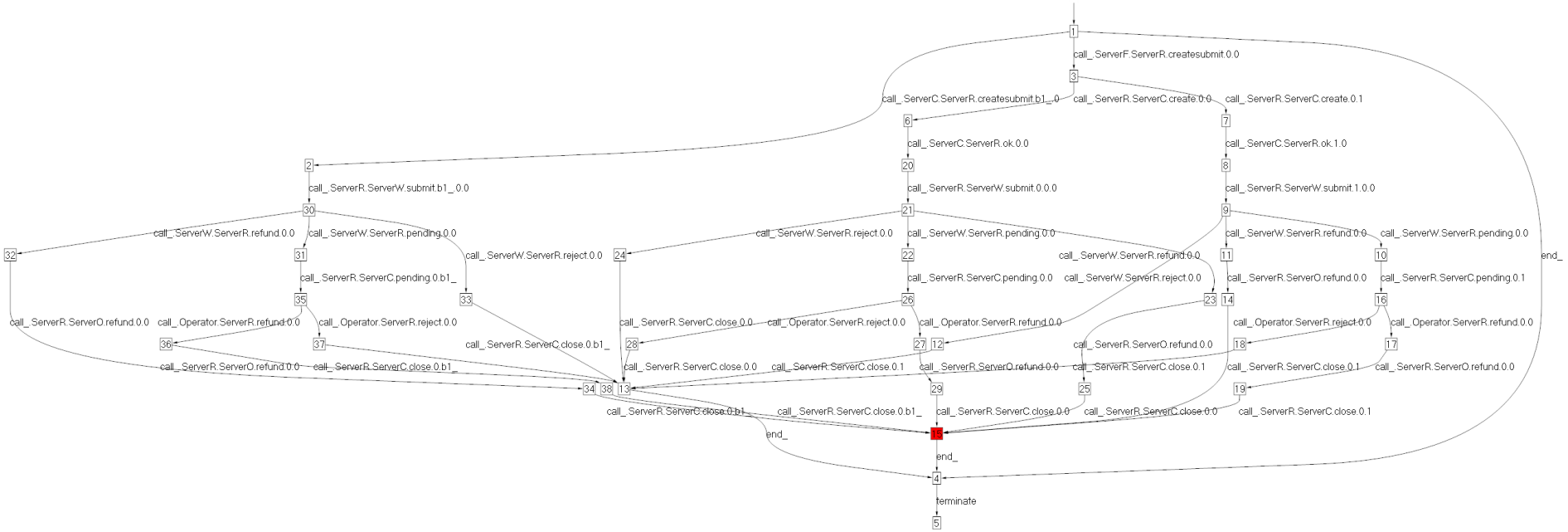
詳細モデル

詳細モデルではバックエンドサーバーのふるまいを追加した



左側の User, ServerC, ServerF オブジェクトは抽象モデルと共通

合成された詳細モデルのCSP



ServerR

```
********Verification Result********
The Assertion (System_C_() deadlockfree) is VALID.
```

詳細化の検証

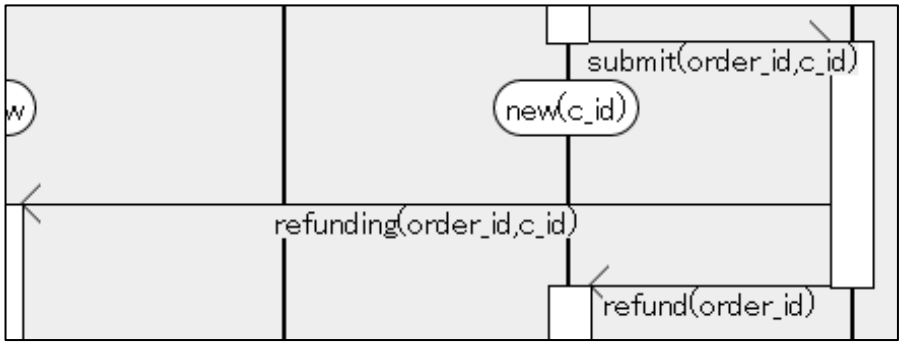
```
System_C_Hidden_() = System_C_() \ {
  call_.Operator.ServerC.refunding.0.0.0,
  call_.ServerW.ServerR.refund.0.0,
  call_.Operator.ServerR.reject.0.0,
  call_.ServerR.ServerW.submit.0.0.0,
  ...
};

#assert System_A_Hidden_() refines<F> System_C_Hidden_();
#assert System_C_Hidden_() refines<F> System_A_Hidden_();
```

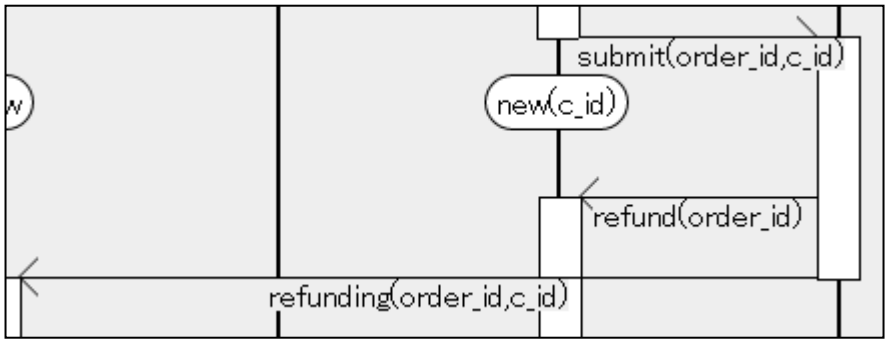
```
********Verification Result********
The Assertion (System_C_Hidden_() refines <F> System_A_Hidden_()) is VALID.

********Verification Result********
The Assertion (System_A_Hidden_() refines <F> System_C_Hidden_()) is VALID.
```

意図的にバグを混入したモデル



The Assertion (System_C_() deadlockfree) is VALID.



The Assertion (System_C_() deadlockfree) is NOT valid.

バックエンドサーバでのメッセージ送信の順番を変更するとエラーが発生

反例解析

Message:
Choose File rap-error-c.txt
Comment Add

```
#prefix COMPO_C_
#end_states done
closed refunded
rejected default
#count User 2
#count ServerC 2
#count ServerR 2

### F-Refund
User(order_id)
ServerC(c_id)
ServerF
ServerR(order_id)
ServerW ServerO
User -> ServerF
request(order_id) {
ServerF -> ServerR
<<create>>submit(ord
er_id) }{
ServerR => ServerC
```

CSP: Convert
prefix: COMPO_C_

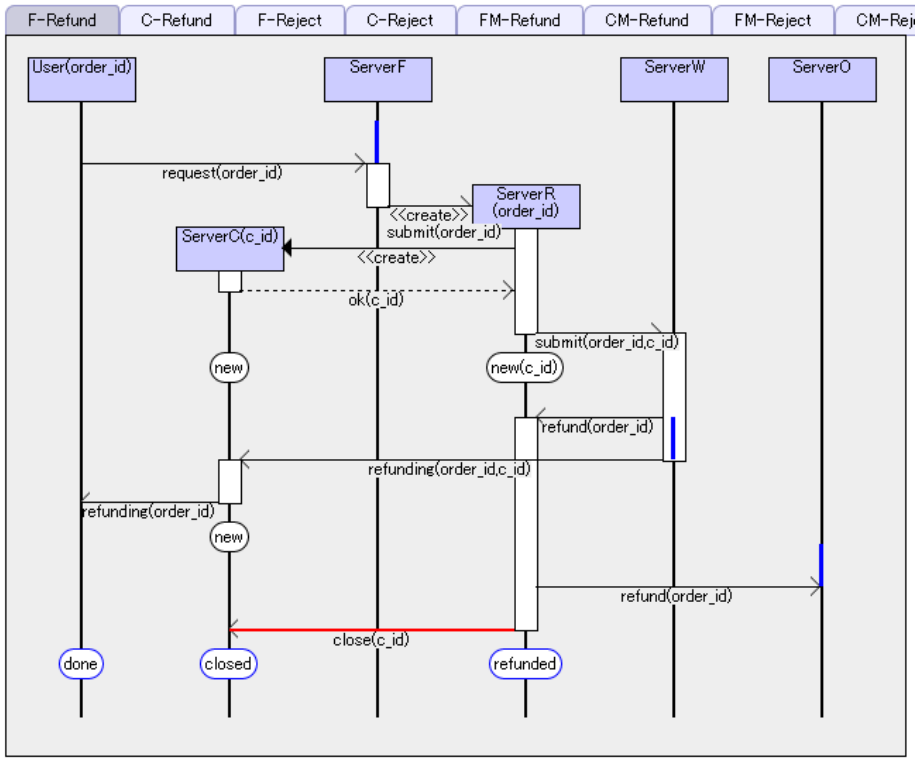
```
channel call_ 0;

enum { ServerO,
ServerC, Operator,
ServerF, ServerR,
```

Log:
Choose File No file chosen

```
call_.ServerR.Server
O.refund.1.0 ->
call_.ServerR.Server
C.close.1.1>
```

```
.ServerC.ServerR.createsubmit.1.1
.ServerW.ServerC.refunding.0.0.0
.ServerC.User.refunding.0.0
.ServerW.ServerR.refund.0.0
.ServerR.ServerO.refund.0.0
.ServerR.ServerC.close.0.0
.ServerR.ServerW.submit.1.1.0
.ServerW.ServerR.refund.0.1
.ServerR.ServerO.refund.1.0
.ServerR.ServerC.close.1.1
```



object	state	available messages
ServerO	default	refund
ServerC[0]	closed	
ServerC[1]	closed	
Operator	default	review
ServerF	default	request
ServerR[0]	refunded	
ServerR[1]	refunded	
User[0]	done	
User[1]	i17_i35_i58_i82	refunding, rejected
ServerW	i7	refunding

ServerW は ServerC にメッセージを送信しようとしているが、ServerC はすでに「closed」状態となっていてメッセージを受信できない

ケーススタディの結果

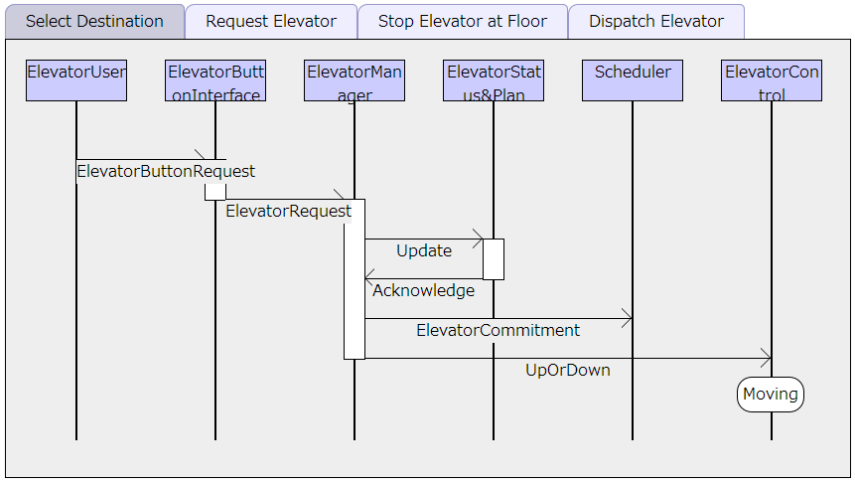
	シーケンス図のページ数	シーケンス図のクラス数	シーケンス図のインスタンス数	シーケンス図の状態数	CSP の状態数	検証時間
抽象モデルデッドロック	4	3	5	12	263	0.026s
詳細モデルデッドロック	8	7	10	44	6481	0.337s
抽象・詳詳細化検証	-	-	-	-	20691	0.575s
不正モデルデッドロック	8	7	10	44	-	0.006s

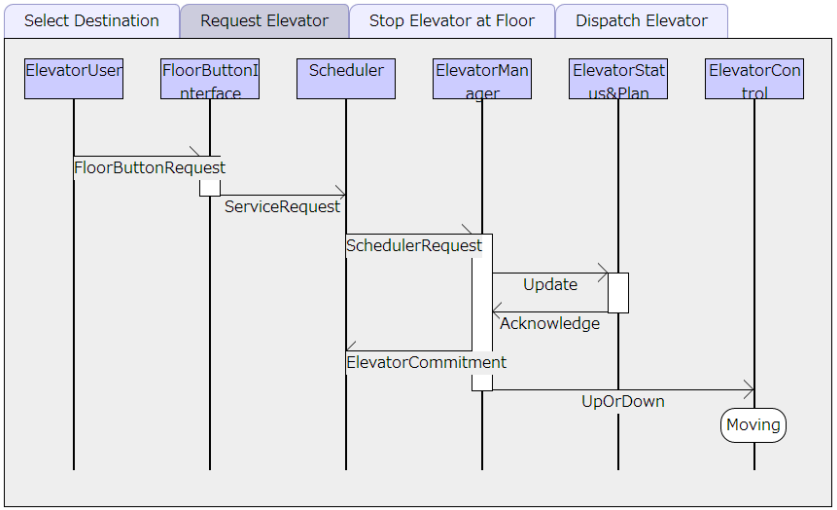
実験環境：2.8 GHz Intel Core i7-2640M, 8 GB RAM

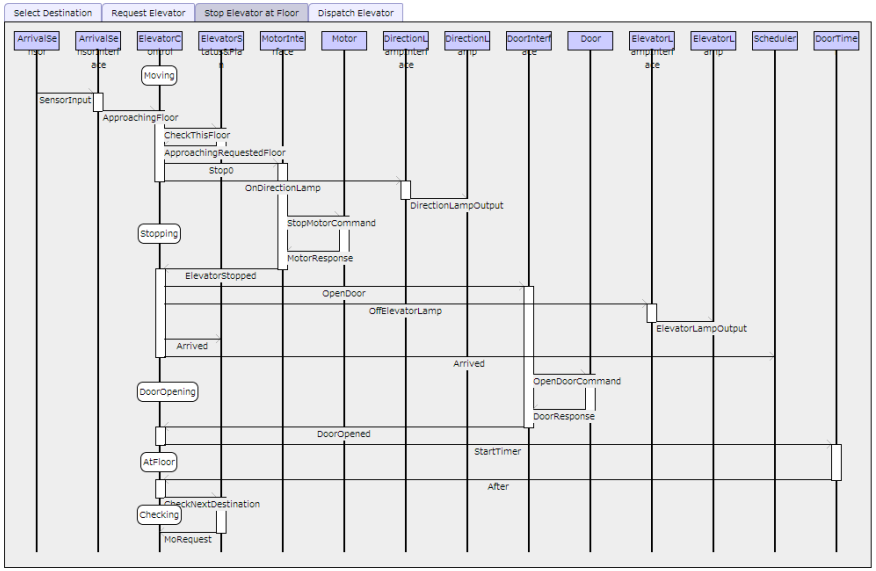
ケーススタディ 2

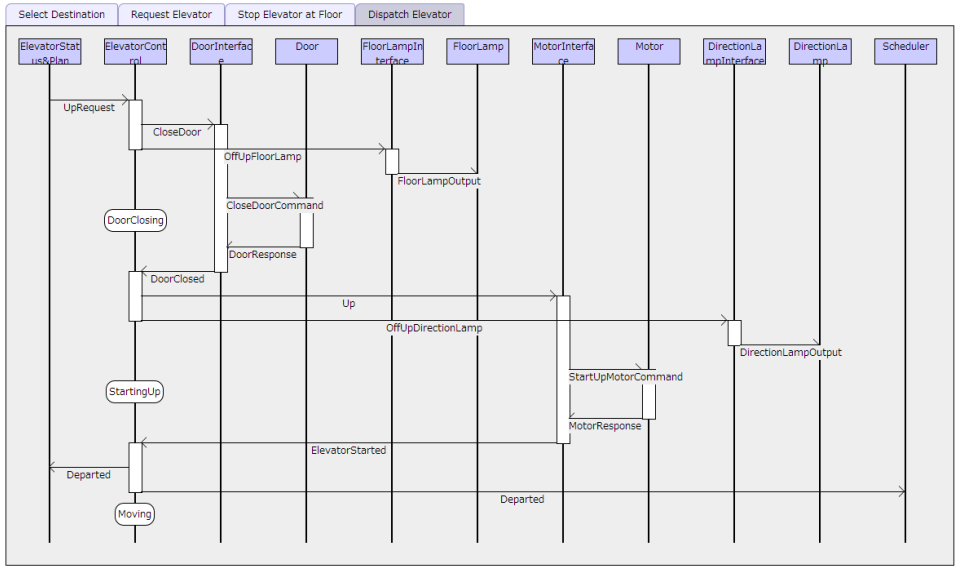
- 検証対象
  - 書籍「Designing Concurrent, Distributed, and Real-Time Applications with UML」で紹介されているエレベータコントロールシステム
    - イベントの発火点がユーザだけではなく、複数のセンサを含む
    - 書籍中でコミュニケーション図で書かれているものをシーケンス図に書きなおして検証
- 検証項目
  - デッドロックしない

入力したシーケンス図


Select Destination
Request Elevator
Stop Elevator at Floor
Dispatch Elevator
ElevatorUser
ElevatorButtonInterface
ElevatorManager
ElevatorStatus&Plan
Scheduler
ElevatorControl
ElevatorButtonRequest
ElevatorRequest
Update
Acknowledge
ElevatorCommitment
UpOrDown
Moving

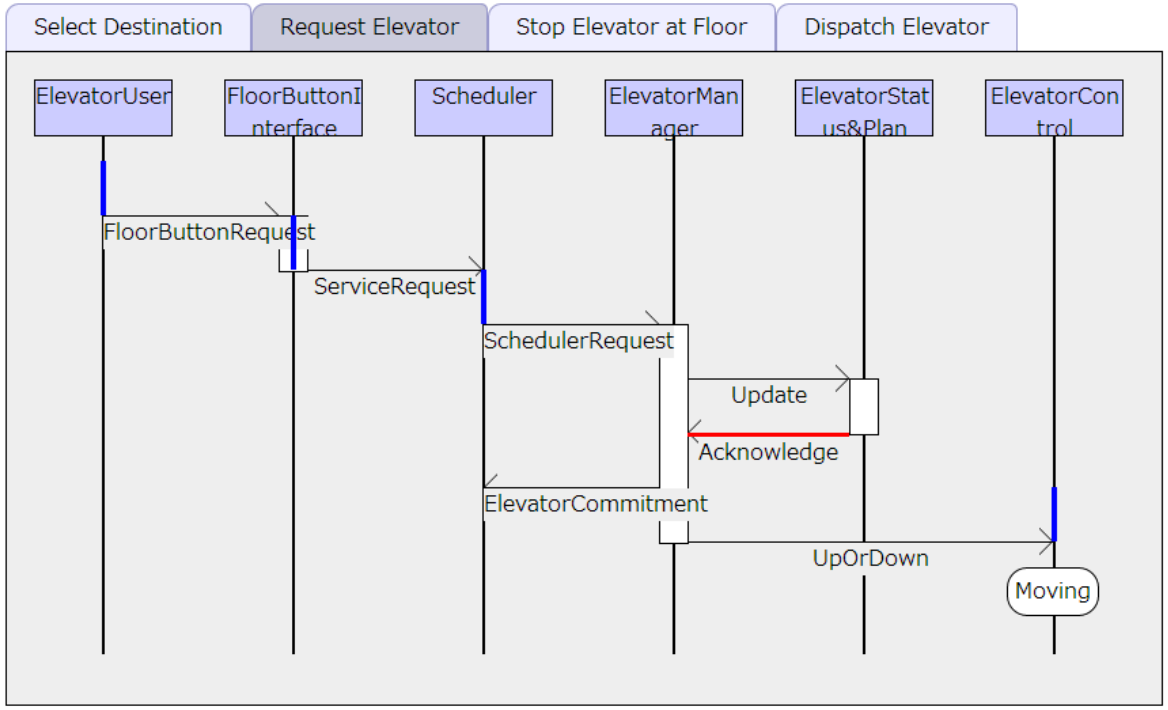

Select Destination
Request Elevator
Stop Elevator at Floor
Dispatch Elevator
ElevatorUser
FloorButtonInterface
Scheduler
ElevatorManager
ElevatorStatus&Plan
ElevatorControl
FloorButtonRequest
ServiceRequest
SchedulerRequest
Update
Acknowledge
ElevatorCommitment
UpOrDown
Moving


Select Destination
Request Elevator
Stop Elevator at Floor
Dispatch Elevator
Moving
SensorInput
ApproachingFloor
CheckThisFloor
ApproachingRequestedFloor
Stop0
OnDirectionLamp
DirectionLampOutput
StopMotorCommand
Stopping
MotorResponse
ElevatorStopped
OpenDoor
OffElevatorLamp
ElevatorLampOutput
Arrived
Arrived
OpenDoorCommand
DoorResponse
DoorOpening
DoorOpened
StartTimer
AtFloor
After
CheckNextDestination
Checking
MoRequest
Scheduler
DoorTime


Select Destination
Request Elevator
Stop Elevator at Floor
Dispatch Elevator
ElevatorStatus&Plan
ElevatorControl
DoorInterface
Door
FloorLampInterface
FloorLamp
MotorInterface
Motor
DirectionLampInterface
DirectionLamp
Scheduler
UpRequest
CloseDoor
OffUpFloorLamp
FloorLampOutput
DoorClosing
CloseDoorCommand
DoorResponse
DoorClosed
Up
OffUpDirectionLamp
DirectionLampOutput
StartingUp
StartUpMotorCommand
MotorResponse
ElevatorStarted
Departed
Departed
Moving

検証結果

The Assertion (deadlockfree) is NOT valid.

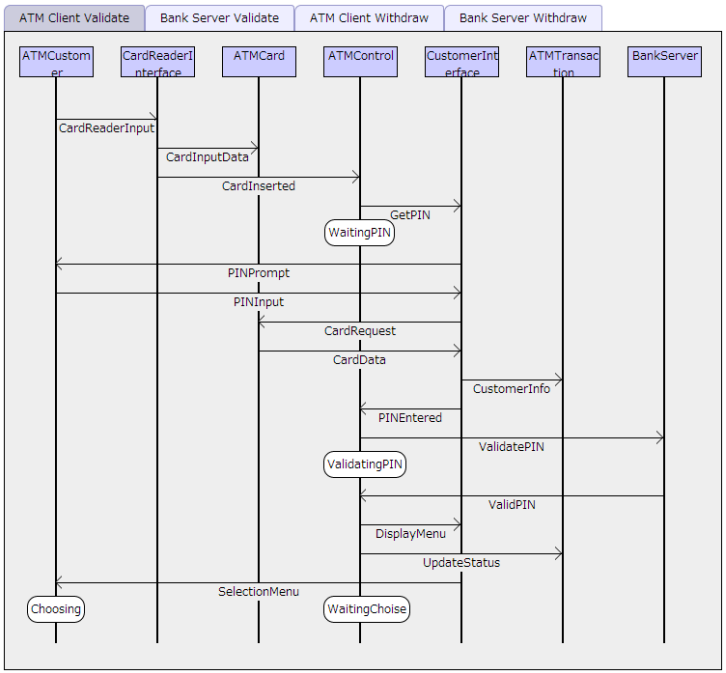


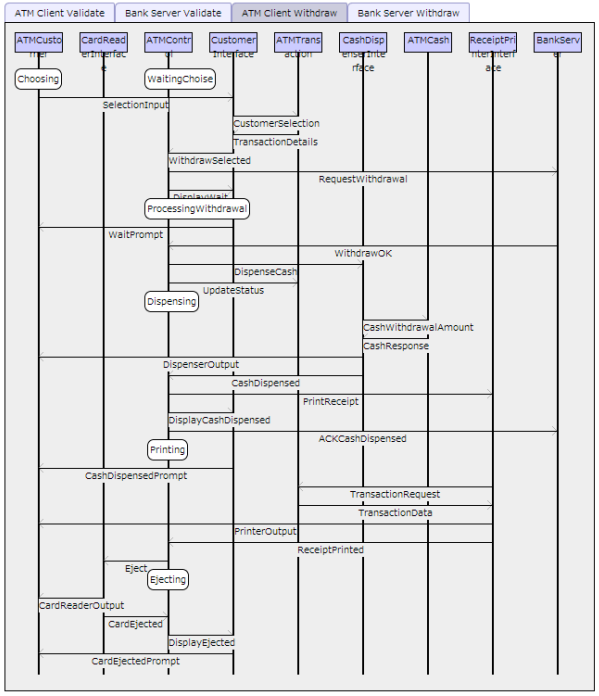
- Scheduler が EventManager にリクエストを送るシナリオと EventManager が Scheduler にリクエストを送るシナリオがあり、2 つのイベントがほぼ同時におこるとデッドロックする
- 十分なサイズのメッセージキューを用意するなどの対応が必要であることを発見できた

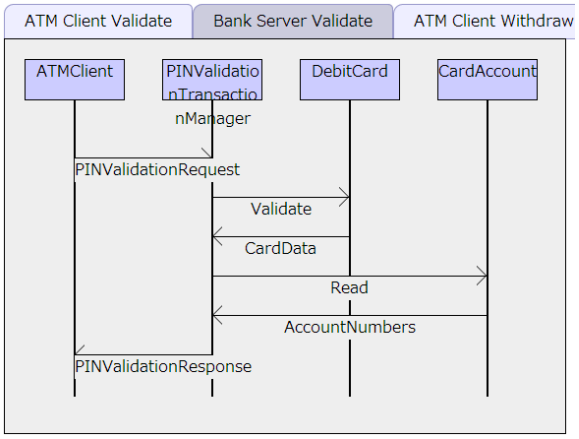
ケーススタディ 3

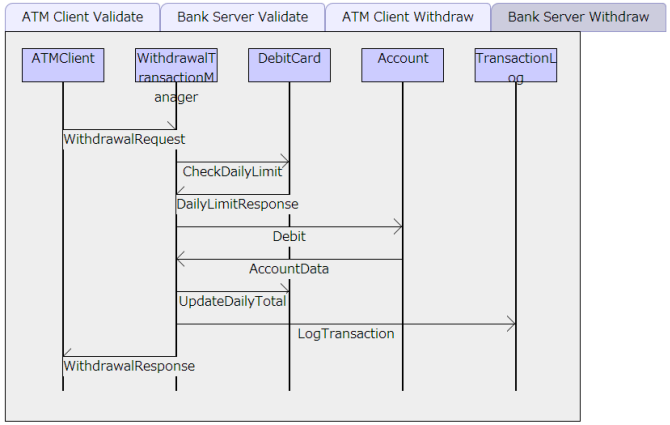
- 検証対象
  - 書籍「Designing Concurrent, Distributed, and Real-Time Applications with UML」で紹介されている銀行コントロールシステム
    - 書籍中でコミュニケーション図で書かれているものをシーケンス図に書きなおして検証
    - BankServer と ATMClient との接続関係が自然言語で書かれているため、全体の検証は対象外とする
- 検証項目
  - デッドロックしない

入力したシーケンス図


ATM Client Validate
Bank Server Validate
ATM Client Withdraw
Bank Server Withdraw
ATMCustomer
CardReaderInterface
ATMCard
ATMControl
CustomerInterface
ATMTransaction
BankServer
CardReaderInput
CardInputData
CardInserted
GetPIN
WaitingPIN
PINPrompt
PINInput
CardRequest
CardData
CustomerInfo
PINEntered
ValidatePIN
ValidatingPIN
ValidPIN
DisplayMenu
UpdateStatus
SelectionMenu
Choosing
WaitingChoise


ATM Client Validate
Bank Server Validate
ATM Client Withdraw
Bank Server Withdraw
Choosing
WaitingChoise
SelectionInput
CustomerSelection
TransactionDetails
WithdrawSelected
RequestWithdrawal
ProcessingWithdrawal
WaitPrompt
WithdrawOK
DispenseCash
UpdateStatus
Dispensing
CashWithdrawalAmount
CashResponse
DispenserOutput
CashDispensed
PrintReceipt
DisplayCashDispensed
ACKCashDispensed
Printing
CashDispensedPrompt
TransactionRequest
TransactionData
PrinterOutput
ReceiptPrinted
Eject
Ejecting
CardReaderOutput
CardEjected
DisplayEjected
CardEjectedPrompt


ATM Client Validate
Bank Server Validate
ATM Client Withdraw
ATMClient
PINValidationTransactionManager
DebitCard
CardAccount
PINValidationRequest
Validate
CardData
Read
AccountNumbers
PINValidationResponse


ATM Client Validate
Bank Server Validate
ATM Client Withdraw
Bank Server Withdraw
ATMClient
WithdrawalTransactionManager
DebitCard
Account
TransactionLog
WithdrawalRequest
CheckDailyLimit
DailyLimitResponse
Debit
AccountData
UpdateDailyTotal
LogTransaction
WithdrawalResponse

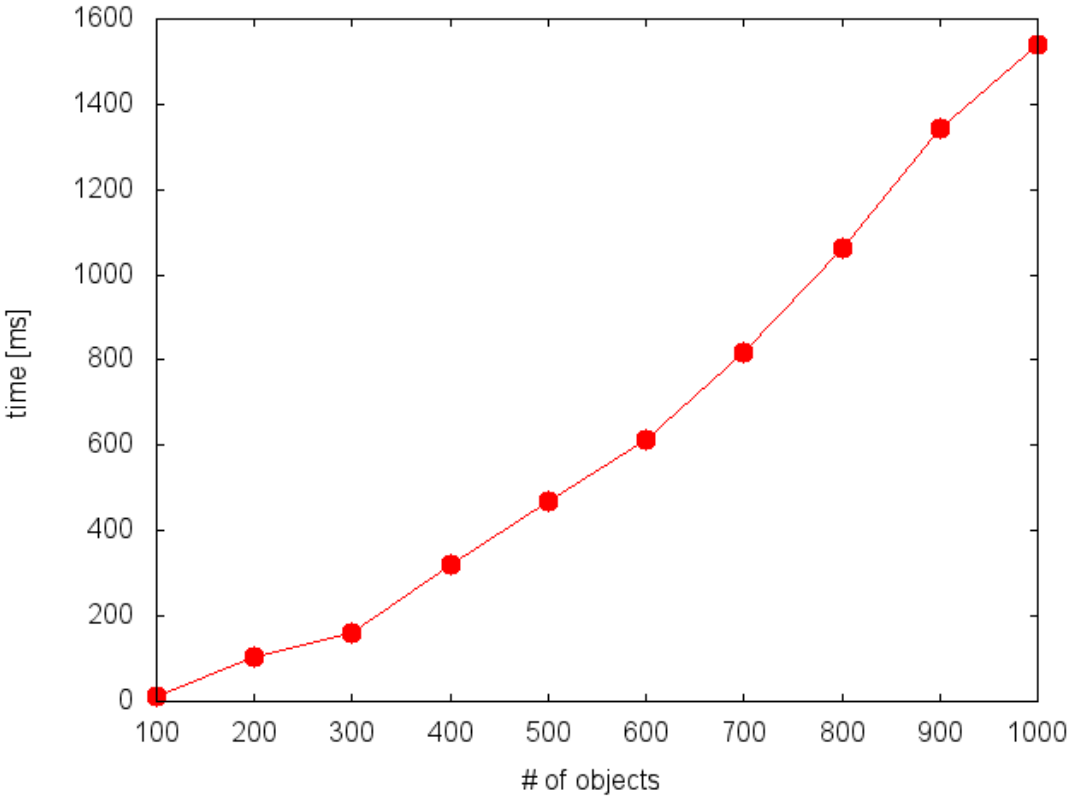
検証結果

- BankServerとATMClientそれぞれのサブシステムで不整合がないことを確認
- 今回対象外としたBankServerとATMClientとの接続関係については、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シーケンス図を書き換えれば検証可能であると考えられる

スケーラビリティ

- ランダムなシーケンス図を生成して変換に必要な時間を計測
  - オブジェクト数を変化させた場合
    - オブジェクト数=N、メッセージ数=N、状態数=100
  - メッセージ数を変化させた場合
    - オブジェクト数=100、メッセージ数=N、状態数=100
  - 状態不変式として記述する状態の数を変化させた場合
    - オブジェクト数=100、メッセージ数=1000、状態数=N

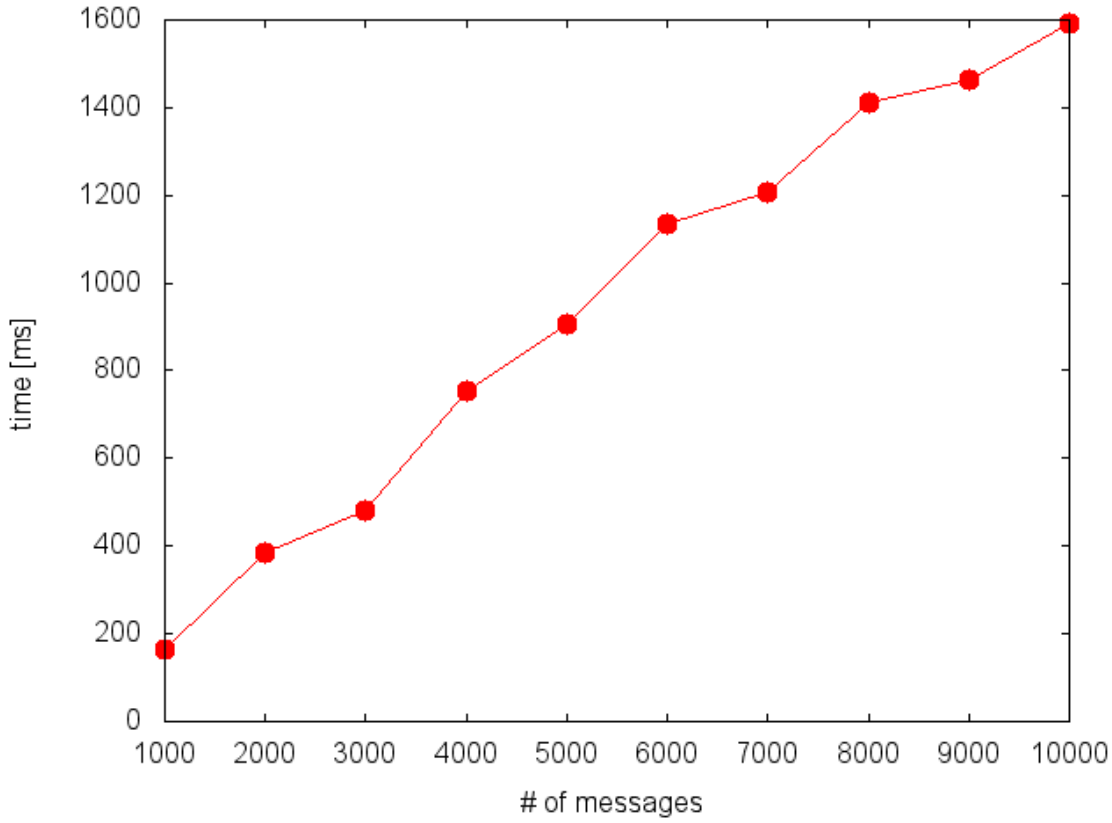
オブジェクト数



- 変換時間はおおよそオブジェクト数に比例
- 700 オブジェクトまでを 1 秒以内に変換可能

実験環境 : 1.8 GHz Intel Core i5-3337U, 4 GB RAM

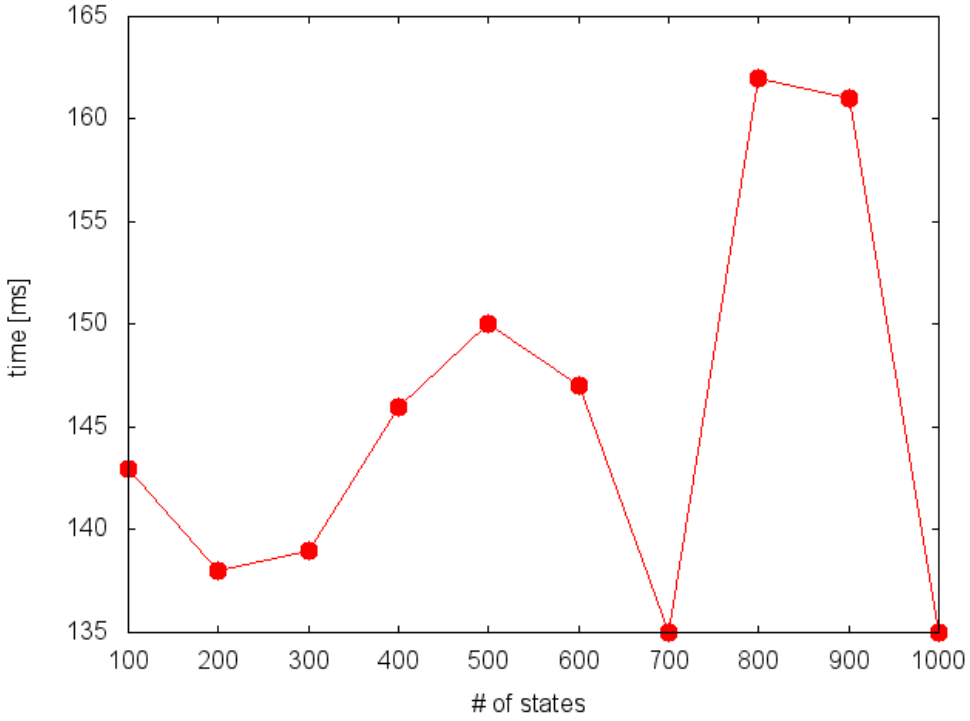
メッセージ数



- 変換時間はおおよそメッセージ数に比例
- 5000 メッセージまでを 1 秒以内に変換可能

実験環境 : 1.8 GHz Intel Core i5-3337U, 4 GB RAM

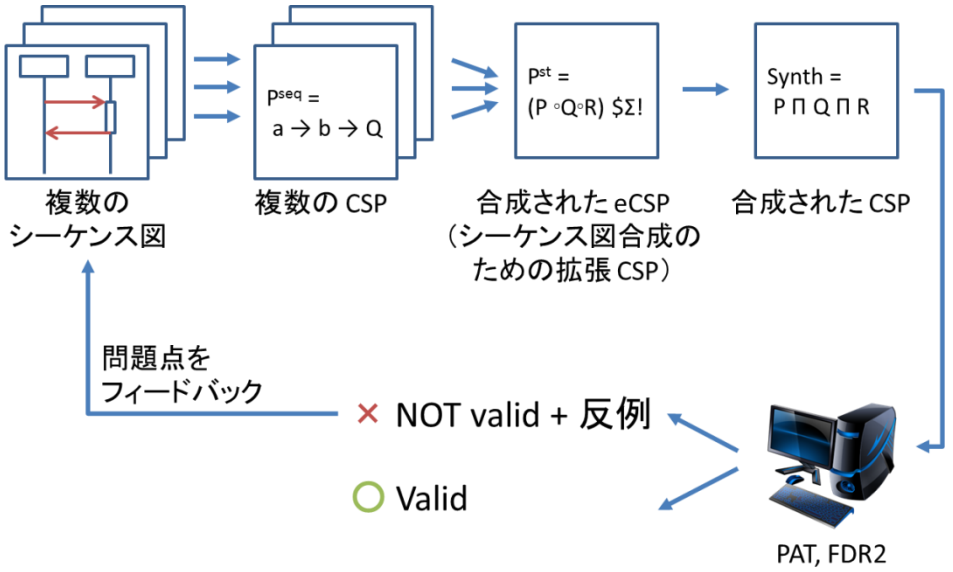
状態数



- オブジェクト数を 100 、メッセージ数を 1000 に固定
- すべての場合で 200 ミリ秒以内に変換可能
  - 遷移可能な「状態の組み合わせ」は小さく、オブジェクト数やメッセージ数と比べて処理時間は無視できる。

実験環境 : 1.8 GHz Intel Core i5-3337U, 4 GB RAM

- はじめに
  - 背景、目的
  - プロセス代数
  - 研究の全体の概要
- CSPによるシーケンス図の検証
  - シーケンス図のCSP記述
  - eCSP（拡張CSP）によるプロセスの合成方法
  - eCSPからCSPへの変換方法
  - 変換ツールSDVerifier
  - 検証と反例解析
- より複雑なシーケンス図への対応
  - オブジェクト生成と破棄
  - 複数オブジェクト
- ケーススタディ
- 関連研究
- おわりに
  - まとめ
  - 成果



関連研究 (1/3)

- ステートマシン図・アクティビティ図を検証する手法
  - Islam Abdelhalim [2011]
  - ○ モデル検査ツールの入力である状態モデルに自然にマッピングできる
  - △ 上流で使われるシナリオベースの設計は検証できない

- 1 枚のシーケンス図を CSP へ変換する手法
  - Li Dan [2010]
  - ○ 結合フラグメントを用いてふるまいを正確に定義できる
  - △ 複数のシナリオが与えられた場合の合成方法には言及していない

関連研究 (2/3)

- 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
  - A. W. Biermann [1976]
- シーケンス図のメッセージ送受信から決定的な状態モデルを生成する手法
  - E. Mäkinen [2001]

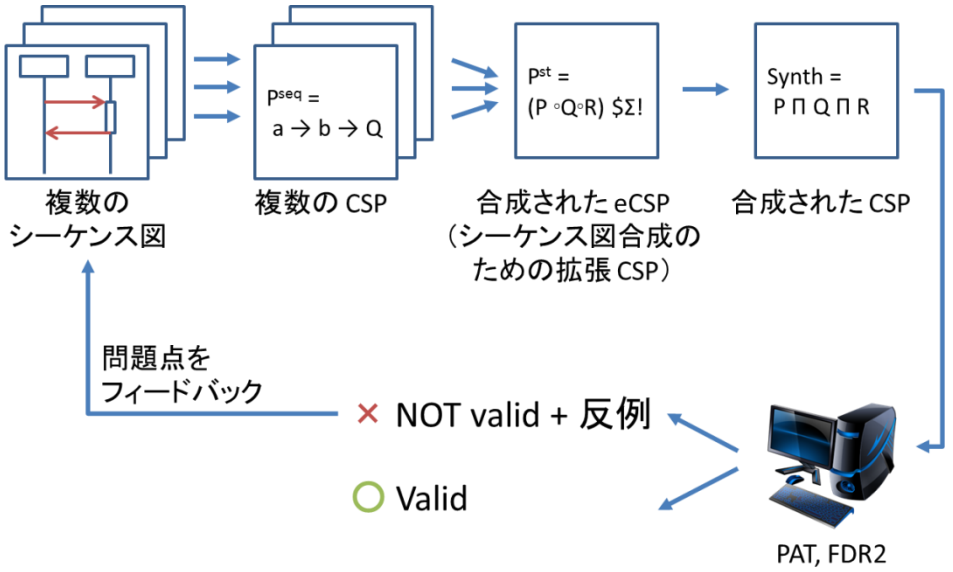
- ○ 付加的な情報を必要とせず、シンプルな状態モデルが合成できる
- △ 決定的なモデルが前提なので非決定性が扱えない

関連研究 (3/3)

- MTS (Modal Transition System) を用いる手法
  - S. Uchitel [2009], I. Krka [2013]
- Live Sequence Chart（シナリオのトリガされる条件を記述したシーケンス図）を用いる手法
  - D. Harel [2002], Y. Bontemps [2005], J. Sun [2005], G. Sibay [2008], R. Kumar [2008], H.Kugler [2009]
- HMSC (シーケンス図間の順序関係を与える図) を用いる手法
  - R. Alur [1999], S. Uchitel [2003]

- ○ 発生するかどうかが未確定なイベントを合成できる
- △ メッセージの送信オブジェクトが未確定なメッセージを選択するということを明示できない

- はじめに
  - 背景、目的
  - プロセス代数
  - 研究の全体の概要
- CSPによるシーケンス図の検証
  - シーケンス図のCSP記述
  - eCSP（拡張CSP）によるプロセスの合成方法
  - eCSPからCSPへの変換方法
  - 変換ツールSDVerifier
  - 検証と反例解析
- より複雑なシーケンス図への対応
  - オブジェクト生成と破棄
  - 複数オブジェクト
- ケーススタディ
- 関連研究
- おわりに
  - まとめ
  - 成果



まとめ

- 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、提案手法に基づいて SDVerifier ツールを開発した
  - 設計上流における非決定性を考慮し、非決定的な選択の主体をモデル化する手法を提案した
  - 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、検証には既存のCSPツールを活用した
  - 合成方法を厳密に議論するため、合成に適したCSP演算子を定義し、その性質を数学的に明らかにして、合成アルゴリズムを構築した
  - シーケンス図の検証と設計ミスの原因解析に SDVerifier が有効であることを、ECサイト、エレベータ、銀行システムの例で実証した
- 今後の課題として、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

成果

- Refinement and Verification of Sequence Diagrams Using the Process Algebra CSP
  - IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences Vol.E96-A No.2 pp.495-504
- Verifying sequence diagrams using the process algebra CSP
  - AVOCS 2013 Automated Verification of Critical Systems Pre-proceedings pp.203-206
- SDVerifier: プロセス代数CSPを用いたシーケンス図検証ツール
  - 日本ソフトウェア科学会 コンピュータソフトウェア（査読結果「照会後判定」により、修正版の判定待ち）

図 2

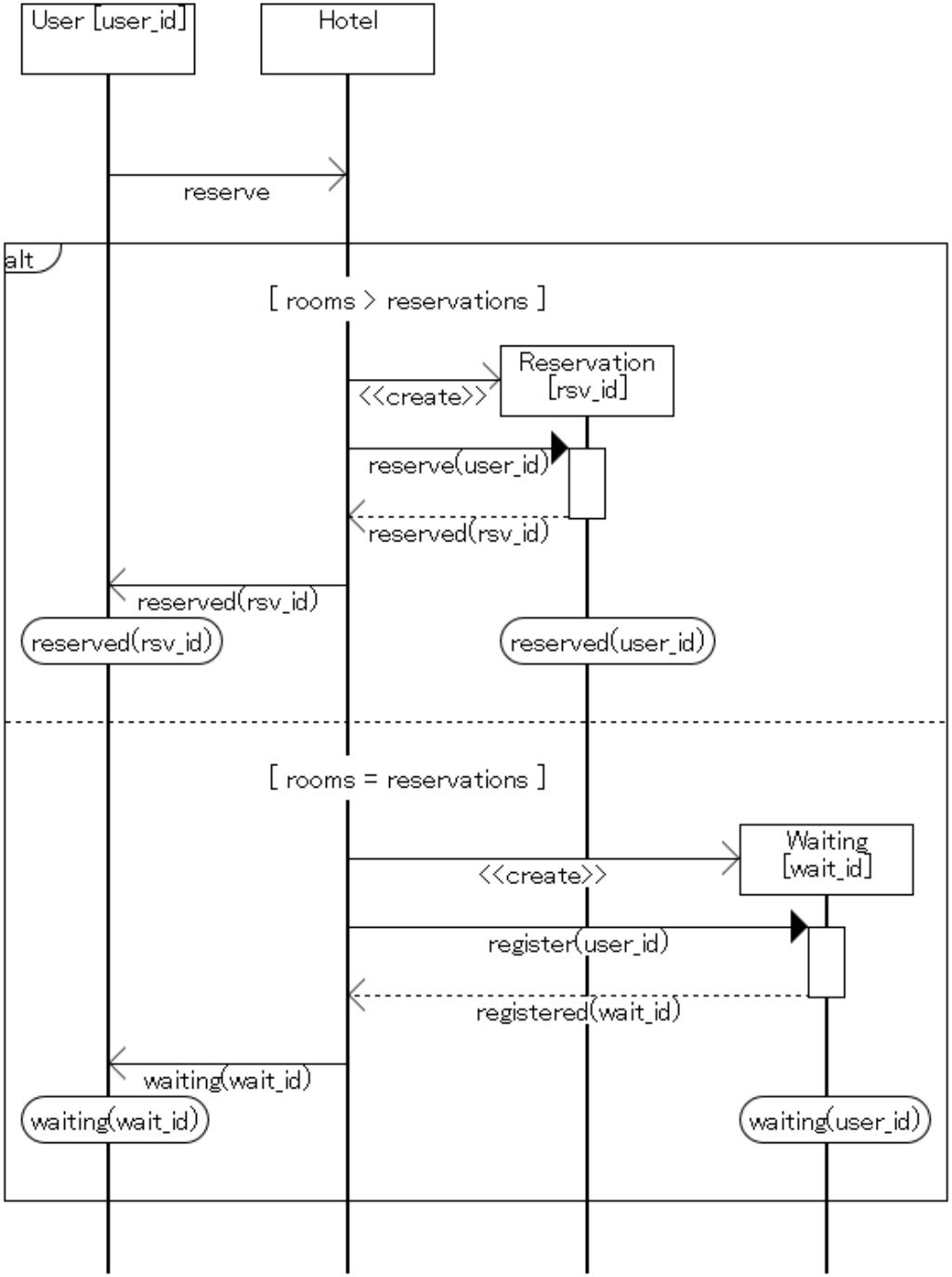
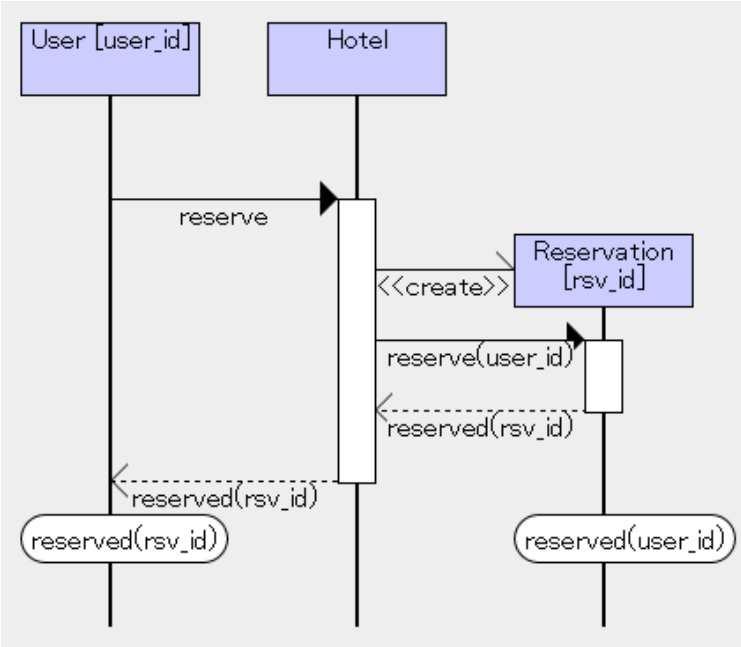

User [user_id]
Hotel
reserve
alt
[rooms > reservations]
Reservation [rsv_id]
<<create>>
reserve(user_id)
reserved(rsv_id)
reserved(rsv_id)
reserved(rsv_id)
reserved(user_id)
[rooms = reservations]
Waiting [wait_id]
<<create>>
register(user_id)
registered(wait_id)
waiting(wait_id)
waiting(wait_id)
waiting(user_id)

図 3


User [user_id]
Hotel
reserve
Reservation
[rsv_id]
<<create>>
reserve(user_id)
reserved(rsv_id)
reserved(rsv_id)
reserved(rsv_id)
reserved(user_id)

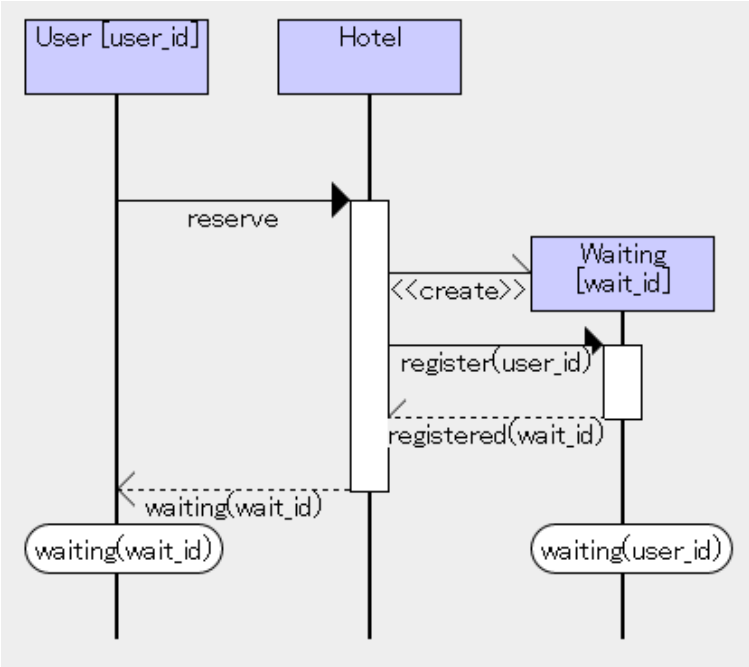
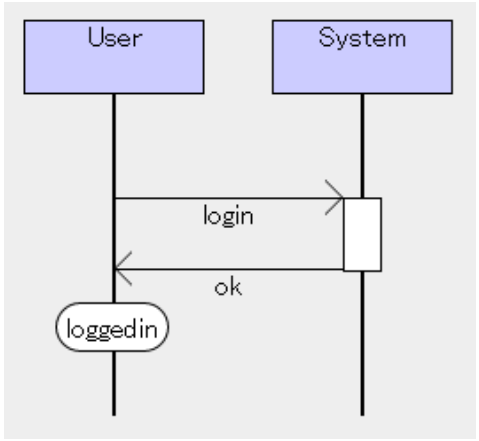
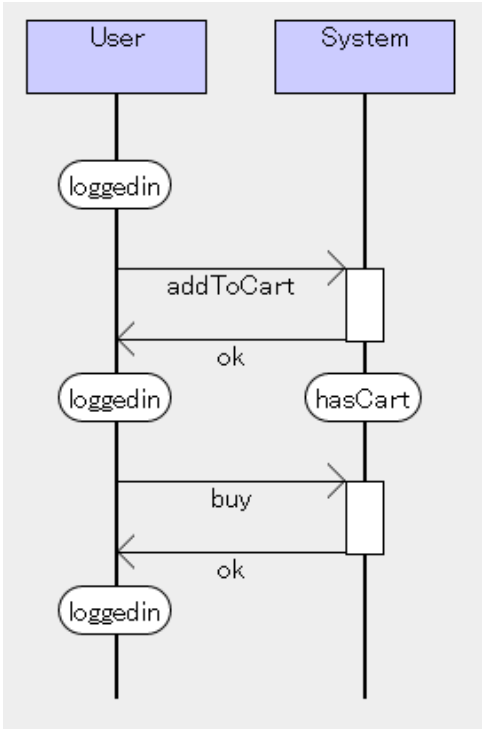
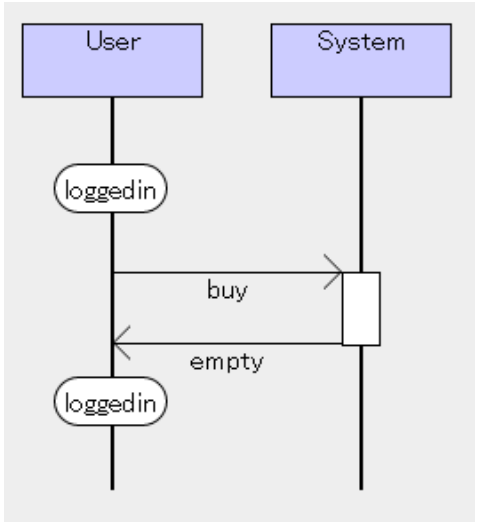

User [user_id]
Hotel
reserve
Waiting
[wait_id]
<<create>>
register(user_id)
registered(wait_id)
waiting(wait_id)
waiting(wait_id)
waiting(user_id)

図 4


User
System
login
ok
loggedin


User
System
loggedin
addToCart
ok
loggedin
hasCart
buy
ok
loggedin


User
System
loggedin
buy
empty
loggedin

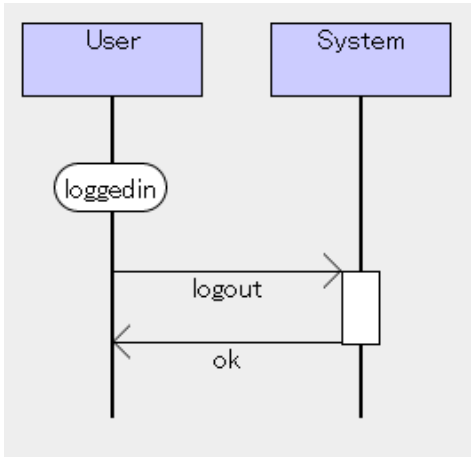

User
System
loggedin
logout
ok

図 5

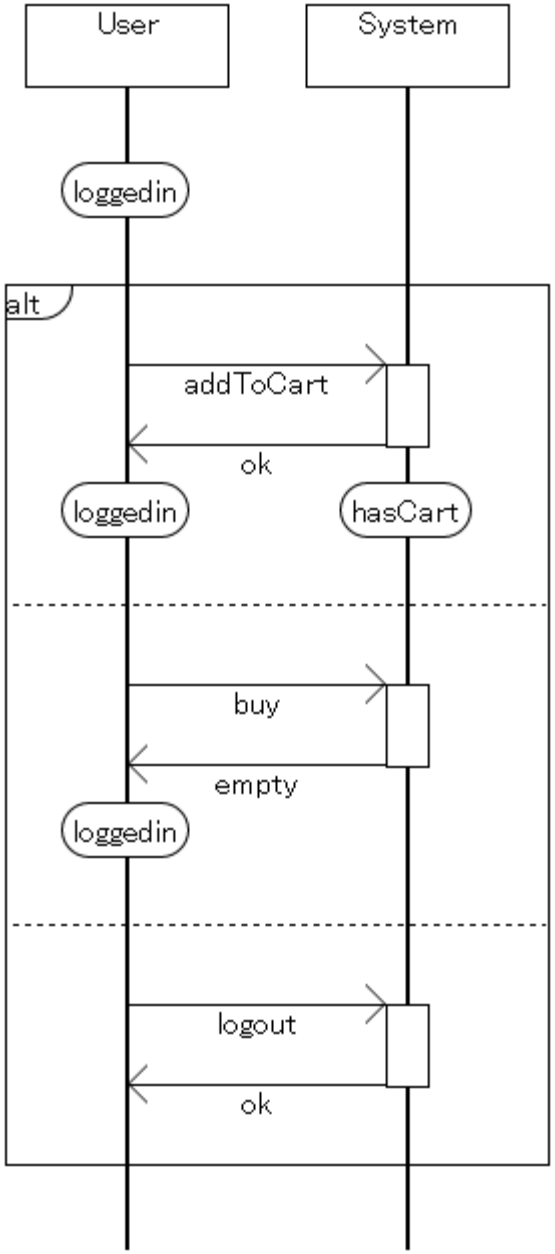
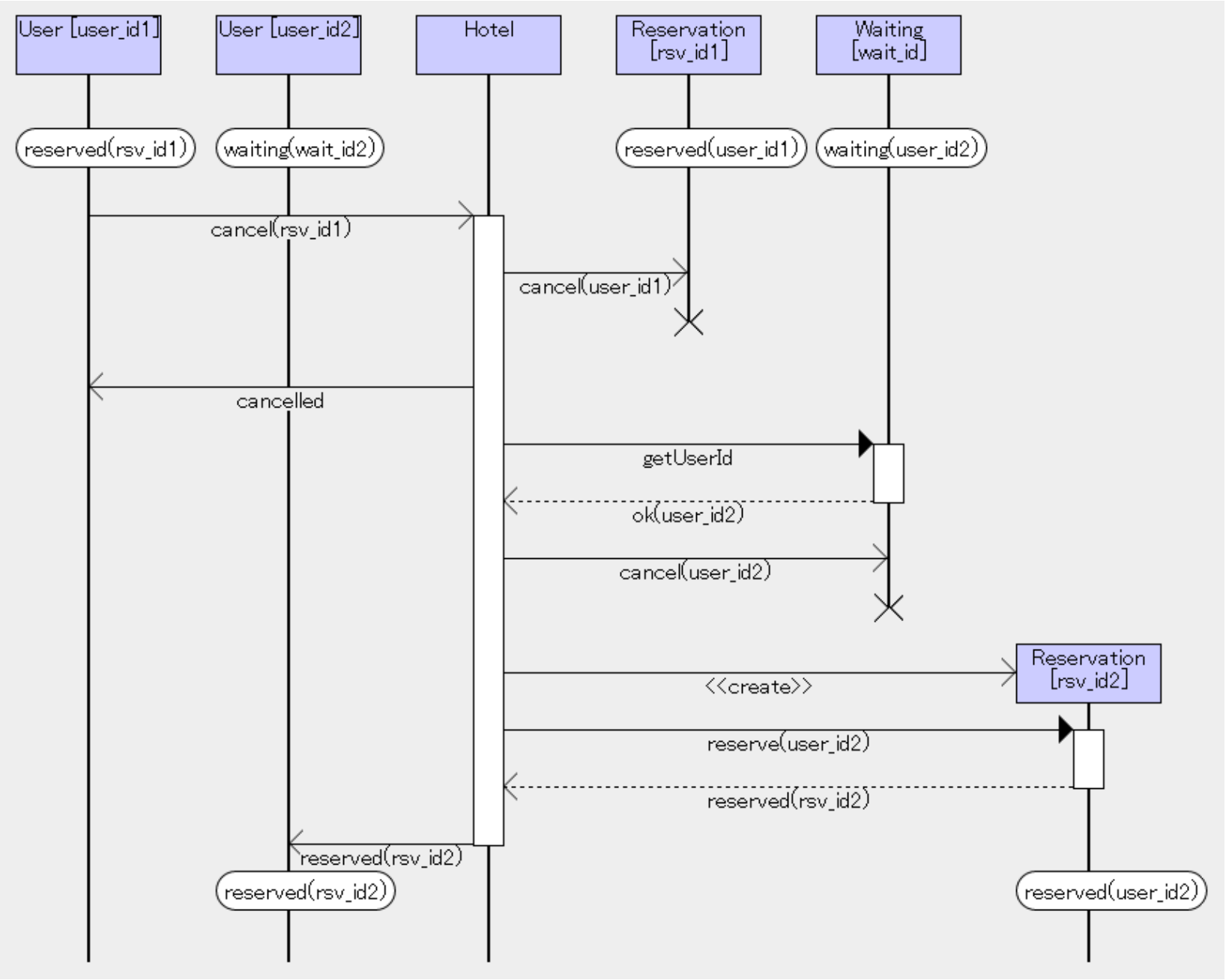

User
System
loggedin
alt
addToCart
ok
loggedin
hasCart
buy
empty
loggedin
logout
ok

図 7


User [user_id1]
User [user_id2]
Hotel
Reservation [rsv_id1]
Waiting [wait_id]
reserved(rsv_id1)
waiting(wait_id2)
reserved(user_id1)
waiting(user_id2)
cancel(rsv_id1)
cancel(user_id1)
cancelled
getUserId
ok(user_id2)
cancel(user_id2)
Reservation [rsv_id2]
<<create>>
reserve(user_id2)
reserved(rsv_id2)
reserved(rsv_id2)
reserved(rsv_id2)
reserved(user_id2)