段階的詳細化をサポートするUML開発環境

海津　智宏†

アブストラクト
複雑なソフトウェアシステムを設計する手法として，コンポーネントの分割による段階的詳細化が有効である．しかし，詳細化に間違いがあるとシステムにバグが混入してしまう．詳細化の各段階のシーケンス図を元に整合性を検査することで，問題点を早期に発見して修復することが可能となる．

A Development Environment for Stepwise Refinement Using UML

Tomohiro Kaizu†

Abstract
Stepwise refinement of components is effective for designing a complex software system. However, a bug occurs when the developer mistakes refinement. Checking the consistency of the sequence diagram of each refinement step, you can find and repair the mistake at the early stage.

1. はじめに

近年，ソフトウェアシステムに求められる機能はますます複雑化・多様化している．機能の複雑化に伴ってソフトウェアシステムの設計が複雑化すると，追跡容易性・再利用性・変更容易性・拡張性などが低くなってしまう．このような問題に対処するために，コンポーネントベースの開発が広まっている[1].

ソフトウェアシステムをコンポーネントに分割することで，設計の複雑さが解消され，システムの保守性が向上する．コンポーネントを用いた開発のための技術としてはJ2EEやStrutsなど多くの実装技術が普及している．また，コンポーネントの分割の指針としていくつかの開発方法論が提案されている．例えば，KobrA 開発方法論[2]では，コンポーネントを機能ごとに分割していくことで段階的に詳細化する手法が定められている．

しかし，このような開発方法論に従っていても，詳細化に間違いがあると正しい設計を得ることはできない．そのため，段階的詳細化の各段階で，詳細化したシステムが元のシステムの性質を引き継いでいることを検証することは重要である．

そこで，本研究では，詳細化の各段階の設計情報を元に整合性を検査することで，詳細化の誤りを発見するツールを提案する．

ソフトウェアの設計には，OMG によって開発・標準化されたUML[3]を利用することが多い．UMLでは，開発工程や目的に応じて，ユースケース図やクラス図などさまざまな図を定義している．

本ツールでは，UML のうち特にシーケンス図に着目し，詳細化を検証する．これは，シーケンス図が設計の初期段階から多数作成され，ふるまいの情報を含むためである．シーケンス図の詳細化関係を検証することによって誤りを早い段階で検出できる．

詳細化関係の判定には，プロセス代数 CSP[4][5]を利用する．CSP では2つのシステム間の詳細化関係を検証する方法が確立しており，詳細化関係や等価性を自動検証するためのモデル検査器 FDR[6]も開発されている．仕様として時相論理を用いる SPIN[7]，SMV[8]，LTSA[9]等のモデル検査器はこのような検証には不向きである．

本研究では，シーケンス図の設計から FDR 記述を得る変換ツールを開発し，詳細化が正しいことを確認しながら設計を進めることのできるソフトウェア開発環境を提供する．このツールの利用により誤りを早期に発見できるため，品質を向上し，修正による手戻り工数を削減することができる．さらに，常に詳細化前後の設計が整合していることを確認できるため，開発途中での仕様変更に対しても，仕様と実装とのずれをなくし，ソフトウェアの保守性を向上することができる．

†日本電気株式会社
NEC Corporation

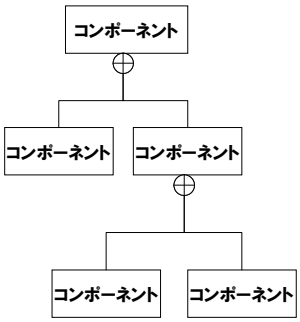

図 1 コンテインメントツリー

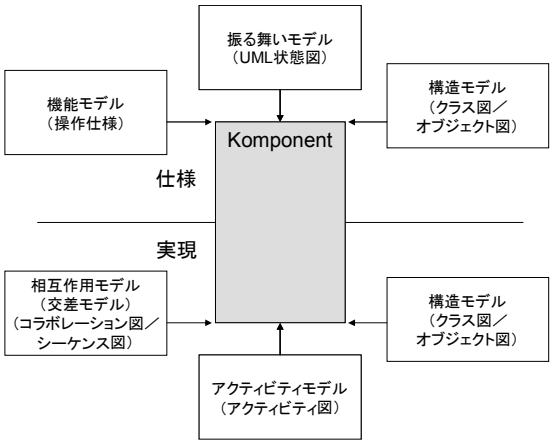

図 2 KobrA のコンポーネントモデル

2. KobrA 開発方法論とプロセス代数 CSP

2.1. KobrA 開発方法論

KobrA 開発方法論は，機能の分割の視点に立って，コンポーネントの仕様作成と実現設計を交互に繰り返すことによってシステムを実現する方法論である．ドイツ・フラウンホーファ協会実践的ソフトウェア工学研究所（Fraunhofer IESE）において開発された．

KobrA における開発のプロセスには，コンテキスト実現，コンポーネント仕様，コンポーネント実現，コンポーネント再利用，品質保証の各工程がある．このうち，上位のコンポーネント実現によって得られるモデルが下位のコンポーネント仕様に対応し，コンポーネント実現とコンポーネント仕様の工程が上位から下位のコンポーネントへと繰り返し実施される．

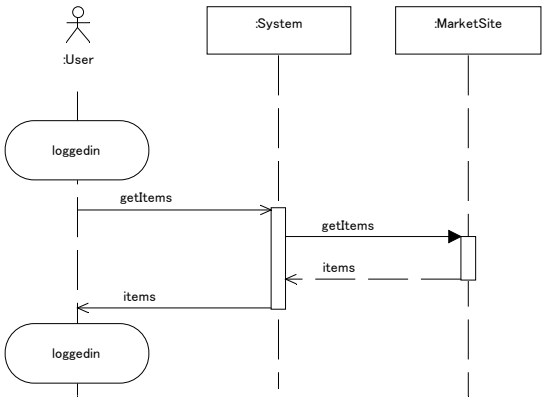

図 3 シーケンス図の例

下位のコンポーネントでは，上位のコンポーネントを機能ごとに分割していく．繰り返しコンポーネントを分割していくことで，図 1 のようなコンポーネントの階層構造が得られる．このツリーはコンテインメントツリーと呼ばれる．

KobrA のコンポーネントモデルを図 2 に示す．それぞれのコンポーネントごとに，構造・振る舞い・機能の仕様と実現が記述される．コンポーネント実現の相互作用モデルでは，コラボレーション図もしくはシーケンス図で相互作用が記述される．

2.2. シーケンス図

シーケンス図は，UML で定義された図の 1 つで，オブジェクト間のメッセージの流れを時系列的に表現する．UML は OMG によって開発・標準化されたオブジェクトモデリングのための仕様記述言語である．図 3 にシーケンス図の例を挙げる．

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

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

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

であれば点線の矢印で記述される.

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

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

2.3. プロセス代数 CSP

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

CSP では, 発生するイベントを単位として並行プロセスの動作を記述することができる. 記述された並行プロセスはモデル検査器 FDR を利用することにより, デッドロックがないこと, ライブロックがないこと, システム間に詳細化関係が成り立つことを検査できる. さらに, JCSP[10]という実装ライブラリがあり, CSP で記述されたプロセスの構造を保ったままシステムを実装することができる.

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

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

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

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

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

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

```
P1 = in → sync → sync → P1
P2 = sync → out → sync → P2
SYS = (P1 [|{sync}|] P2)＼{sync}
```

図 4 CSP の例

隠蔽は「P ＼ X」(FDR では「P ¥ X」)という形で記述され, プロセス P のうちイベントの集合 X に含まれるイベントが内部で自動的に実行されるプロセスを表す. X に含まれるイベントは外部から制御できない.

図 4は CSP の例である. P1 は in, sync, sync を繰り返すプロセス, P2 は sync, out, sync を繰り返すプロセスである. 並行プロセス SYS では P1 と P2 が sync で同期して並行に動作し, P1 が in を独立に実行, P1 と P2 が sync で同期, P2 が out を独立に実行, P1 と P2 が sync で同期, という動作を繰り返す. このとき, sync は隠蔽されて外部からは制御できないため, 並行プロセス SYS は in と out を交互に実行するプロセスとして観測される.

2.4. モデル検査器 FDR

FDR は CSP の詳細化関係や等価性を検査できるモデル検査器である. FDR の入力である FDR 記述は CSP 記述とほぼ同等だが, いくつかの点で違いがある. FDR 記述と CSP 記述の違いとしては,「→」などの記号の代わりに「->」などのアスキー表現を使用すること, 基本データ型以外のデータを用いる場合には型を定義する必要があること, 表明を記述できることなどが挙げられる.

データ型は datatype 宣言で定義できる. 例えば, Color 型が Red, Green, Blue のいずれかの値をとる場合, 次のように宣言する.

```
datatype Color = Red | Green | Blue
```

他のデータ型の値の部分集合を別の型として定義するには, subtype として宣言する. 例えば, 次の RedOrBlue 型は Color 型の subtype となる.

```
subtype RedOrBlue = Red | Blue
```

FDR での検証では, トレース等価, 失敗等価, 失敗発散等価という3種類の等価性がサポートされ, それぞれの意味論で詳細化関係を検証できる. トレースに基づく詳細化関係([T=])では安全性を, 失敗に基づく詳細化関係([F=])ではデッドロックフリー性と活性を, 失

敗発散に基づく詳細化関係([FD=]ではライブロックフリー性を検証できる.

2 つのシステムが等価であることを検証するには, 両方向の詳細化関係を検証する. たとえば, 次の 2 つの表明がともに成り立つ場合, プロセス P とプロセス Q は失敗発散等価である.

assert P [FD= Q
assert Q [FD= P

3. 段階的詳細化の検証

3.1. 提案ツールの概要

詳細化の各段階の設計情報を元に整合性を検査することで, 詳細化の誤りを発見するツールを提案する. このツールの利用により誤りを早期に発見できるため, 品質を向上し, 修正による手戻り工数を削減することができる. さらに, 常に詳細化前後の設計が整合していることを確認できるため, 開発途中での仕様変更に対しても, 仕様と実装とのずれをなくし, ソフトウェアの保守性を向上することができる.

検証の対象とする設計情報としてはシーケンス図を用いる. シーケンス図は設計の初期段階から多数作成され, ふるまいの情報を含むため, これらの詳細化関係を検証することによって, 誤りを早い段階で検出できる.

各シーケンス図はシステムの一つの動作例を表すものであるが, 複数のシーケンス図からシステム全体の動作を類推することができる. 検証の際には, 各詳細化段階の複数のシーケンス図を合成して FDR 記述に変換し, FDR で段階的詳細化を検証する.

3.2. 検証の内容

提案するツールでは, コンポーネントの分割による段階的詳細化が行われていることを前提とする. 詳細化前のシステムのコンポーネントを複数のコンポーネントに分割することで, 詳細化後のシステムが得られる. この時, 分割したコンポーネントの 1 つは細化前と同名のコンポーネントとし, 外部との通信の窓口として動作する. 外部からのメッセージはこのコンポーネントが受け取り, 内容に応じて分割したコンポーネントに処理を振り分ける(図 5).

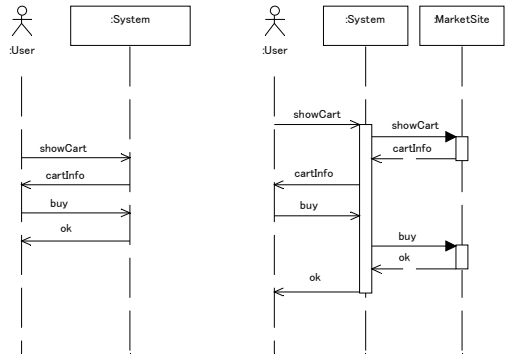

図 5 詳細化前後のシーケンス図の例

シーケンス図	CSP
オブジェクト	システムに並行合成されるプロセス
メッセージ	イベント
状態不変式	プロセスの状態
活性区間	活性区間どうしの順序関係を無視するように変換する.

表 1 シーケンス図と CSP との対応

このような詳細化において, 詳細化前のシステムに含まれるメッセージ送受信は全て詳細化後のシステムにも含まれている. 提案するツールは, 詳細化前後のシステムのうち同名のコンポーネントのみを抽出し, メッセージ送受信の内容・順序が変更されていないことを検証する.

3.3. シーケンス図と CSP との対応

提案するツールでは, シーケンス図を CSP に変換する. シーケンス図と CSP との対応関係を 表 1 に示す. シーケンス図中のオブジェクトが CPS のプロセスとなり, メッセージ送受信をイベントとして変換する.

シーケンス図中の結合フラグメント, 相互作用使用, 消失・拾得メッセージ, 汎用順序には未対応である. また, メッセージの種類は考慮されない. 状態不変式は状態名のみ記述可能で, 制約式が記述された状態不変式には対応しない. 異なる状態名が記述された各状態は全て排他的とみなされ, 状態不変式のない状態とも排他的なものとして扱われる. これらの制限は今後改善していく予定である.

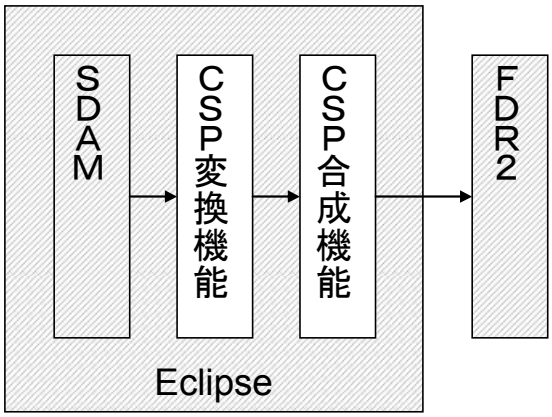

図 6 提案ツールの全体構成

メッセージ送受信の内容に複数の可能性がある場合，送信側のオブジェクトの内部状態によってメッセージを選択する．受信側のオブジェクトは，任意のメッセージを受信可能とする．このような選択方式の違いはCSPの外部選択・内部選択の記法により表現可能であり，メッセージ送受信を適切に表現できる．例外として，スティックマン形状で表現されるアクターの動作は検証対象のシステムには決定できないため，アクターの動作に複数の可能性がある場合は一律外部選択に変換する．

4. 実装

4.1. 提案ツールの全体設計

提案ツールは，開発環境として広く普及しているEclipse 上で動作する．シーケンス図の記述にはEclipse 上で動作する UML エディタであるSystemDirector Application Modeler UML Editor（以下SDAM）を用いる．詳細化の検証にはモデル検査器FDR を利用する．提案ツールは，SDAM の出力するUML ファイルを読み込んで FDR 記述を出力する CSP変換機能と，複数の詳細化段階の FDR 記述を合成して表明を含むFDR記述を出力するCSP合成機能を持つ．

提案ツールの全体構成を図 6に示す．網掛け部分は既存のツールを利用し，白抜きの「CSP 変換機能」および「CSP 合成機能」は新規に開発した．

4.2. CSP 変換機能

CSP 変換機能は，UML のシーケンス図をもとにFDR 記述を出力する．変換の入力は XMI 2.1 形式のXMLファイルとする．これはOMGで定められた標準形式なので，対応する UML エディタであれば SDAM 以外の UML エディタも利用可能である．SDAM では，モデルのリポジトリが XMI 2.1 形式で出力される．あらかじめ全てのシーケンス図をリポジトリに保存しておく．

FDR 記述への変換規則を以下に示す．

コンポーネント名集合

ライフラインのヘッドに記述された型名の集合をコンポーネント名集合とする．FDR 記述での集合名はCompoName_とする．

例えば，クラス User のライフラインとクラス System のライフラインが存在する場合，次のような FDR 記述を出力する．

datatype CompoName_ = User | System

状態名集合

シーケンス図中に存在する状態不変式の名前の集合に，特殊な状態「default_」を加えた集合を状態名集合とする．FDR 記述での集合名は StateName_とする．

例えば，シーケンス図中に loggedin という状態不変式と haveCart という状態不変式が存在する場合，次のような FDR 記述を出力する．

datatype StateName_ = default_ | loggedin | haveCart

メッセージ名集合

シーケンス図中に存在するメッセージ名の集合をメッセージ名集合とする．FDR 記述での集合名はCallName_ とする．

例えば，次のような FDR 記述を出力する．

datatype CallName_ = getRecipe | recipe | buy | back | cartInfo | logout | getItems | items | showCart | clearCart | empty | ng | login | ok | addToCart

アクター名集合

コンポーネント名集合のうち，型が Actor であるものの集合をアクター名集合とする．シーケンス図中ではヘッドが矩形ではなくスティックマン形状で表示される．FDR 記述での集合名は ActorName_ とする．AcorName_ は CompName_ の部分型となるので，datatype ではなく subtype で定義する．

例えば，User の型が Actor である場合，次のようなFDR 記述を出力する．

subtype ActorName_ = User

メッセージ

シーケンス図中の各メッセージは，メッセージ名・送信元・送信先の組み合わせに変換する．シーケンス図中のメッセージを表す call_チャネルを用意し，メッセージ名・送信元・送信先を順にドット(.)で結ぶ．

例えば，User から System へ getItems メッセージが送信される場合，このメッセージは次のように変換される．

call_.getItems.User.System

インターフェース関数

特定のコンポーネントの入力もしくは出力となるメッセージの集合を返す関数を定義する．常に次の FDR 記述を出力する．

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

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

プロセス

各コンポーネントのふるまいは全て COMPO_というプロセスとして定義する．COMPO_は引数としてコンポーネント名と状態名をドット(.)で結んだ値をとる．状態名のつけられた状態から次の状態名のつけられた状態までのメッセージの送受信を記述する．この変換の詳細については4.3節で詳しく述べる．

システム

全てのコンポーネントを並行合成したプロセスを SYSTEM_とする．初期状態のコンポーネントは全て default 状態にあると仮定する．常に次の FDR 記述を出力する．

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

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

表明

システムがデッドロックしないことを検証する．常に次の FDR 記述を出力する．

```
assert SYSTEM_ :[deadlock free [F]]
```

4.3. プロセスの変換規則

前述したように，各コンポーネントのふるまいは全て COMPO_というプロセスとして定義し，状態名のつけられた状態から次の状態名のつけられた状態までのメッセージの送受信を記述する．状態名のつけられた状態とは，シーケンス図中で状態不変式の置かれた位置，各シーケンス図の先頭・末尾，活性区間の切れ目のいずれかである．

COMPO_プロセスの内容は以下のアルゴリズムで出力する．

1. 全てのライフラインについて，状態名のつけられた状態で分割する．
2. 同じコンポーネントのライフラインについて，同じ状態から始まり同一のメッセージ送受信が存在するものをまとめ，ツリーを作る．
3. ツリーの各ノードを CSP のプレフィックスとして出力する．ツリーが分岐する箇所では，外部選択もしくは内部選択を出力する．コンポーネントがアクターでなく，分岐の次のメッセージ送受信が全てメッセージ送信の場合のみ内部選択を，それ以外の場合は外部選択を出力する．

例として 図 7，図 8 のようなシーケンス図を考える．このシーケンス図によると，ユーザーは request1，request2 をシステムに送信する．図 7 はそれぞれに対してシステムから ok が返されている．図 8 の左のシーケンス図は request1 に対して ng が返され，ユーザーはそこで処理を中断している．図 8 の右のシーケンス図では request1 に対しては ok が返され，request2 に対して ng が返されている．

このシーケンス図には状態不変式が存在しないので，状態名のつけられた状態は各シーケンス図の先頭・末尾，活性区間の切れ目である．これらは全て特殊な状態「default_」として扱われる．

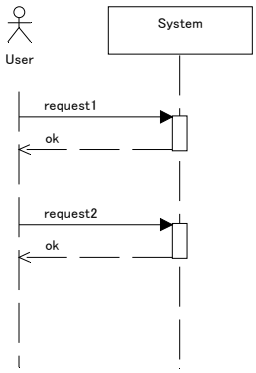

図 7 シーケンス図の例（正常系）

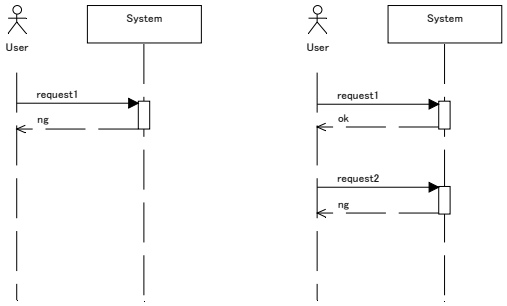

図 8 シーケンス図の例（異常系）

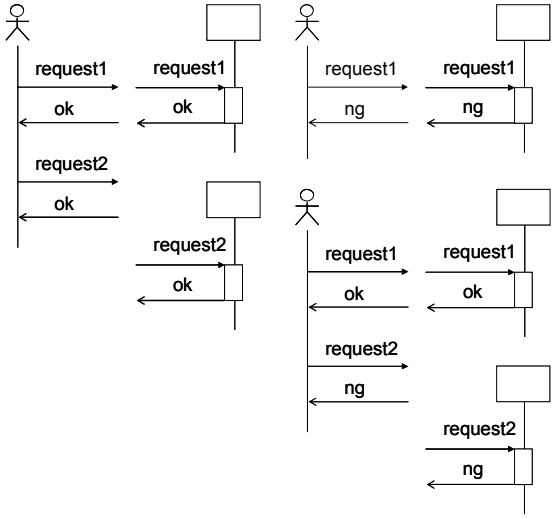

図 9 分割したシーケンス図

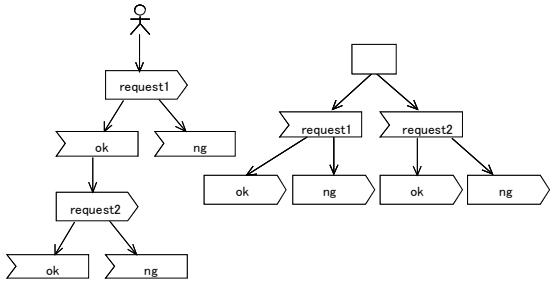

図 10 メッセージ送受信のツリー

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

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

図 11 出力される FDR 記述

活性区間の切れ目でシーケンス図を分割すると，図 9 のようになる．このシーケンス図から，request1, request2 等について同一のメッセージ送受信をまとめていくと，図 10のようになる.

このツリーをたどって FDR 記述を出力する. スティックマン形状の User コンポーネントはアクターなので全て外部選択を出力する. 矩形の System コンポーネントは request1, request2 とも次の ok, ng がともにメッセージ送信なので，内部選択を出力する.

この例では，出力は図 11のような FDR 記述となる.

4.4. CSP 合成機能

CSP 合成機能は，2 つの FDR 記述を合成し，詳細化の検証が可能な FDR 記述を出力する. 合成の規則を以下に説明する.

コンポーネント名集合

2 つの FDR 記述のコンポーネント名集合の和集合を合成後のコンポーネント名集合とする. もとの FDR 記述のコンポーネント名集合は CompoName1_と CompoName2_ に名前を変更し，CompoName_ の subtype とする.

例えば，一方のコンポーネント名集合が User | System であり，もう一方が User | System | MarketSite | RecipeSite | UserManager の場合，次のような FDR 記述を出力する.

datatype CompoName_ = User | System | MarketSite | RecipeSite | UserManager
subtype CompoName1_ = User | System
subtype CompoName2_ = User | System | MarketSite | RecipeSite | UserManager

状態名集合

2 つの FDR 記述の状態名集合の和集合を合成後の状態名集合とする. もとの FDR 記述の状態名集合は StateName1 と StateName2 に名前を変更し，StateName_の subtype とする.

メッセージ名集合

2 つの FDR 記述のメッセージ名集合の和集合を合成後のメッセージ名集合とする. もとの FDR 記述のコンポーネント名集合は CallName1_と CallName2_に名前を変更し，CallName_の subtype とする.

プロセス

それぞれの FDR 記述内のプロセス記述を併記する. プロセス名を COMPO_から COMPO1_と COMPO2_に変更することで名前の衝突を避ける.

システム

全てのコンポーネントを並行合成したプロセスを SYSTEM1_と SYSTEM2 とする. 常に次の FDR 記述を出力する.

SYSTEM1_ = || cn_:CompoName1_ @ [interface_(cn_)] COMPO1_(cn_.default_)
SYSTEM2_ = || cn_:CompoName2_ @ [interface_(cn_)] COMPO2_(cn_.default_)

内部遷移集合

CompoName1_と CompoName2_のいずれか一方にしか含まれないコンポーネントを抽出し，そのコンポーネントの入出力となるメッセージの集合を internalEvents_として出力する.

例えば，MarketSite，RecipeSite，UserManager が一方の FDR 記述にしか含まれない場合，次のような FDR 記述を出力する.

internalEvents_ = Union ({interface_(MarketSite), interface_(RecipeSite), interface_(UserManager)})

表明

それぞれのシステムがデッドロックしないこと，内部遷移集合を無視したときに 2 つのシステムのふるまいが等しくなることを検証する. 常に次の FDR 記述を出力する.

assert SYSTEM1_ :[deadlock free [F]]
assert SYSTEM2_ :[deadlock free [F]]

assert SYSTEM1_ ¥ internalEvents_ [FD= SYSTEM2_¥ internalEvents_
assert SYSTEM2_ ¥ internalEvents_ [FD= SYSTEM1_¥ internalEvents_

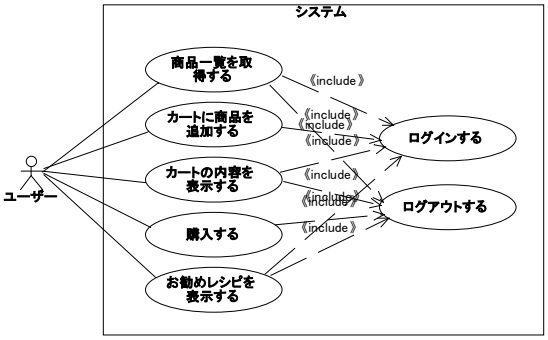

図 12 ユースケース図

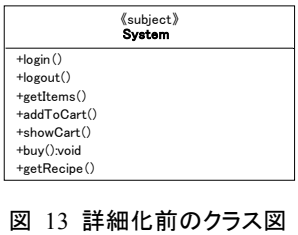

図 13 詳細化前のクラス図

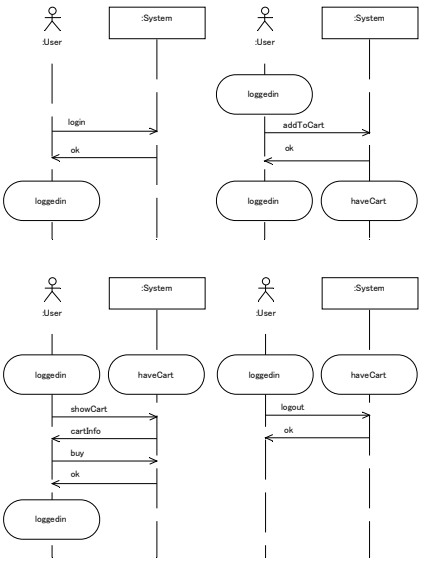

図 14 詳細化前のシーケンス図（一部）

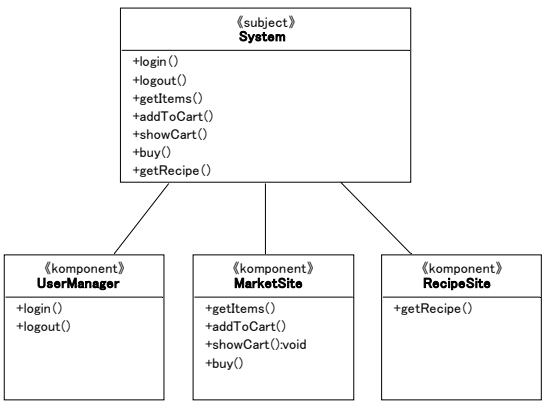

図 15 詳細化後のクラス図

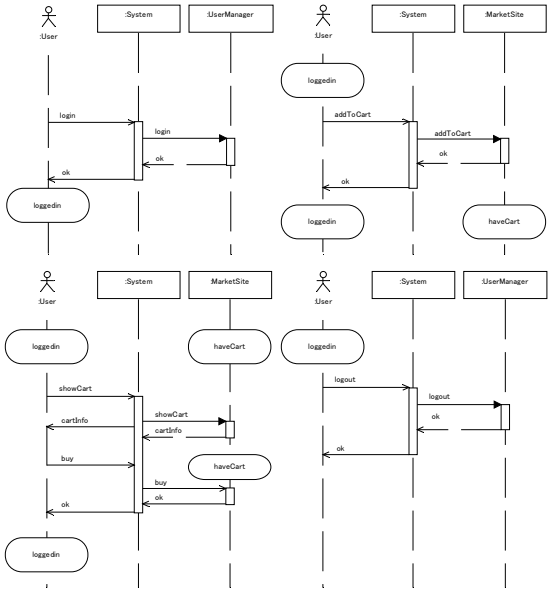

図 16 詳細化後のシーケンス図（一部）

5. 検証の例

このツールを使用した検証の例として，ショッピングサイトの例を考える．このショッピングサイトには，商品一覧の表示・商品のカートへの追加・購入・お勧めレシピの表示といった機能が存在する．また，これらの機能を使用するためにはログインとログアウトが必要である．ユースケース図は 図 12 のようになる．

最初の段階では，全ての機能を単一の System コンポーネントで実現する．図 13 がクラス図である．

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

シーケンス図は，異常系のフローも記述し，事前条件が異なる場合の挙動もそれぞれ記述する．この例では 14 種類のシーケンス図を記述した．シーケンス図の例を 図 14 に示す．これらのシーケンス図から，提案ツールにより FDR 記述を得ることができる．

次に，機能ごとにコンポーネントを分割する．ここでは，ショッピングサイトの機能を「ユーザーの管理」「商品の販売」「レシピの提供」の 3 種類に分類し，機能ごとのコンポーネントに分割する．図 15 がクラス図である．

このクラス図に従い，機能ごとに詳細化したコンポーネントに処理を委譲するようにシーケンス図を詳細化する．シーケンス図の例を 図 16 に示す．これらのシーケンス図からも FDR 記述が得られる．

この詳細化が正しいことを FDR で検証する．2 つの FDR 記述を提案ツールの CSP 合成機能で合成することにより，表明を含む FDR 記述が得られる．この例では，詳細化前後で共通するユーザーとシステム間のメッセージ送受信が保存されていることが検証される．実際に得られた FDR 記述を FDR で検証すると，図 17 のようにエラーが発見され，「×」マークが表示される．FDR のデバッガ（図 18）で問題点を確認すると，一度ログアウトして再度ログインした時に問題が発生することがわかる．showCart メッセージに対する返信として，詳細化前では empty メッセージが返されていた状況で，詳細化後では cartInfo メッセージを返そうとしている．

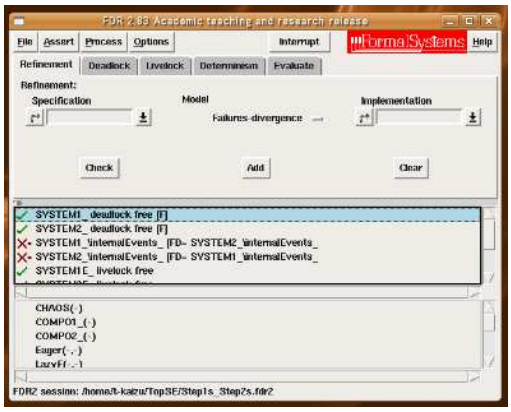

図 17 FDR での実行結果

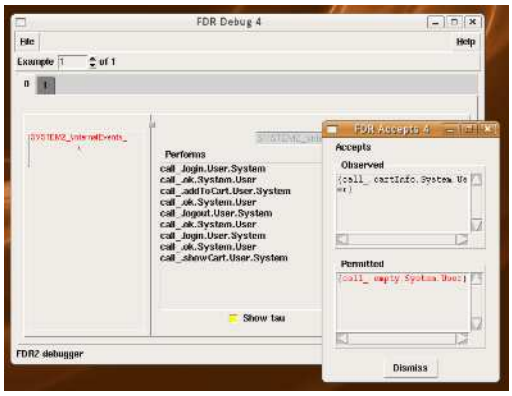

図 18 FDR のデバッガ

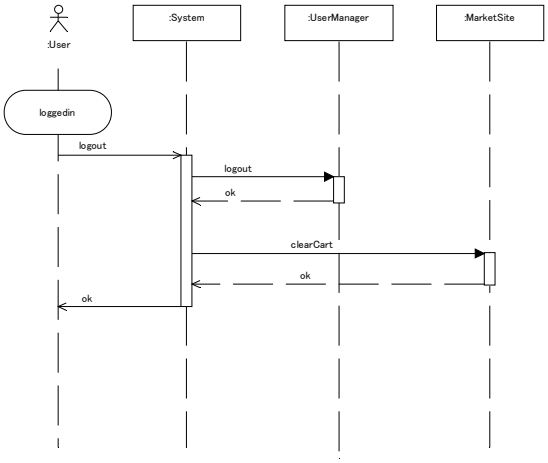

図 19 修正したシーケンス図

シーケンス図を確認すると，カートが存在する時のログアウト処理の内容が詳細化前後で一致していないことがわかる．詳細化前では，カートが存在する状態でログアウトメッセージが送られると，事後条件ではカートはクリアされている．詳細化後では，カートの有無はSystemコンポーネントではなくMarketSiteコンポーネントで管理するように変更されているが，ログアウト時にはUserManagerのみに処理が委譲され，MarketSiteの情報が更新されない．

もし詳細化前の挙動が正しいのであれば，ログアウト時にもMarketSiteの状態を更新するように詳細化後のシーケンス図を直すことで検証が成功する（図 19）．

逆に，詳細化後のシーケンス図が正しい場合には詳細化前の仕様を直すことでも検証が成功する．

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

6. 関連研究

メッセージシーケンスチャートのモデル検査については，複数のメッセージシーケンスチャートおよびそれらの間の順序関係を記述したHMSCというモデルを状態遷移モデルに変換して検証する手法がR.Alurなどにより提案されている[11]．また，この手法に基づき，モデル検査器LTSA上でメッセージシーケンスチャートを検証するためのMessage Sequence Chart plugin[12]（以下LTSA-MSC）というツールが開発されている．

LTSA-MSCはメッセージシーケンスを検証するという点で本研究と同様の問題を対象にしている．しかし，LTSA-MSCでは，モデルの変換は行われるが，そのモデルに対して何を検証するかについては言及していない．そのため，各開発者が別途検証したい仕様を定義する必要がある．本研究では，段階的詳細化を検証対象とするため，表明まで自動的に生成することができる．

また，本研究では入力としてUMLのシーケンス図を使っているため，メッセージシーケンスチャートよりも多くの情報をモデルから得ることができる．例えば，活性区間や状態不変式はメッセージシーケンスチャートには存在しない．さらに，LTSA-MSCではメッセージ送信とメッセージ受信の違いを考慮しないが，本研究では検査用の記述にCSPを用いるため，送信と受信をより正確に変換することができる．このように，本研究では，より正確な変換を行うことで検証の精度を高めている．

コンポーネントベースの開発の検証としては，松本充広氏が代数仕様言語CafeOBJを用いた手法を提案している[13]．松本氏の手法では，コンポーネントのふるまいをCafeOBJで記述することにより，コンポーネントの組み合わせが，作成するソフトウェアの満たすべき振舞いを満たしているかどうかを検証できる．また，検証後のCafeOBJ記述からJava言語への変換についても提案している．

松本氏の提案では，システムは原始オブジェクトを段階的に組み合わせたものとして記述される．原始オブジェクトのふるまいは，操作に対する状態変化として，等式を用いて記述される．本研究がトップダウン的な詳細化を検証するのに対し，松本氏の提案はボトムアップ的なコンポーネントの組み合わせを検証する枠組みになっており，相補的な関係にある．

なお，松本氏の提案では，コンポーネントの仕様がCafeOBJで記述されることを前提としているが，実際の開発者がCafeOBJを使いこなすことは難しい．この技術を一般に利用できるようにするためには，UMLのような広く普及した記述を入力とするよう，拡張が必要である．

7. 考察

このツールを用いて段階的詳細化を検証することで，詳細化前後での仕様の違いを早期に発見できるようになった．もし詳細化に間違いがあった場合，検出された箇所を修正することで，品質を向上し手戻り工数を削減することができる．また，詳細化後のUMLが正しい場合には，詳細化前のUMLに戻って仕様を修正することで，仕様と実装のずれをなくすことができる．

また，このツールでは，十分にシステムのふるまいを網羅したシーケンス図が与えられれば表明を含むFDR記述を自動的に生成するため，開発者がCSPやFDR記述などの新しい言語を学習する必要がなく，導入障壁が低い．

実際にショッピングサイトの例を本ツールに適用することで，開発者がFDR記述を記述することなく，早期に詳細化の間違いを検出できることを確認した．

今後の課題を 2 点挙げる.

1 点目は, 詳細化のパターンに合わせた検査規則の策定である. ここで提案した検査規則は, コンポーネントが増加した時に既存のコンポーネント内のメッセージ送受信が変化しないということを仮定している. しかし, 実際には, 詳細化段階でメッセージを追加したり, 機能を追加・削除したりすることが考えられる. このような詳細化のパターンに応じて複数の検査規則を用意し, 開発者が最適な検査規則を選択できるようにする必要がある.

2 点目は, シーケンス図の記述力の向上である. 現在のところ, シーケンス図の結合フラグメントや汎用順序, メッセージの種類などには対応していない. これらに対応することで, より正確なシーケンス図を記述できるようになる. また, 状態不変式についても, 状態名にしか対応しておらず, 各状態名で表される状態は全て排他的であることを仮定している. これらの制限は緩和することが望ましい. さらに, シーケンス図だけでなく, ユースケース記述やアクティビティ図からもシーケンス図相当の情報を抽出して変換を行うことで, より実際の開発に近い形で設計を検査できるようになる.

これらの課題は, UML の意味論を可能な限り FDR 記述に反映するよう改善し, 解決できると考えられる.

参考文献

[1] 本位田 真一, 鷲崎 弘宣, 丸山 勝久, 山本 理枝子 : サイエンスによる知的ものづくり 4 コンポーネントベース開発テキスト. 近代科学社, 2006

[2] C. Atkinson, J. Bayer, and D. Muthig: Component-Based Product Line Development. The KobrA Approach. Proc. of the First Software Product Lines Conference, 2000

[3] Object Management Group: UML 2.1.2. http://www.omg.org/spec/UML/2.1.2/

[4] C. A. R. Hoare: Communicating Sequential Processes. Prentice Hall, 1985.

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

[6] Formal Systems (Europe) Limited: Failures-divergence refinement: FDR. http://www.fsel.com/

[7] Gerard Holzmann: Spin - Formal Verification. http://spinroot.com/spin/whatispin.html

[8] CMU Model Checking Group: The SMV System. http://www.cs.cmu.edu/~modelcheck/smv.html

[9] Jeff Magee, Jeff Kramer, Robert Chatley, Sebastian Uchitel: LTSA - Labelled Transition System Analyser. http://www.doc.ic.ac.uk/ltsa/

[10] Peter Welch (University of Kent): Communicating sequential processes for java (jcsp). http://www.cs.kent.ac.uk/projects/ofa/jcsp/.

[11] R.Alur, M.Yannakakis: Model Checking of Message Sequence Charts. Proc. CONCUR'99, 1999

[12] Sebastian Uchitel, Robert Chatley, Jeff Magee, Jeff Kramer: LTSA - Message Sequence Chart Plugin. http://www.doc.ic.ac.uk/ltsa/msc/

[13] 松本充広: コンポーネント仕様に基づく高信頼 Java コード生成支援ツール. 高度情報化支援ソフトウェアシーズ育成事業 99 第 004 号, 1999.

謝辞

ご多忙の中本修了製作を丁寧にご指導いただいた産業技術総合研究所の磯部祥尚先生, 有益な助言をいただいた国立情報学研究所の鷲崎弘宣助手, ならびにトップエスイープロジェクトリーダーの本位田真一教授, トップエスイープロジェクト講師の皆様に深く感謝いたします.