データベースシステム 113－52
（1997. 7. 16）

バージョンフロー・モデルを用いた設計プロセス管理について

松尾 博基　　大久保 将也　　岩井原瑞穂
九州大学大学院システム情報科学研究科
〒816 福岡県春日市春日公園 6-1
E-mail: {hmatsuo,ohkubo,iwaihara}@c.csce.kyushu-u.ac.jp

アブストラクト　近年の分散協調設計環境において，プロセス管理は重要なものとなってきている．その中では集中型データベースの管理に代わって，プロセス間の設計バージョンの流れに着目した管理がなされている．本稿ではバージョンフロー(設計プロセス間を流れる様々なバージョン)に着目し，プロセス管理とデータ管理を統合した環境としてバージョンフロー・モデルを検討する．メッセージにプロセスの状態，バージョンへのリンクを付加することにより，利用者間のコミュニケーションの支援とバージョンフローの管理を可能にする．プロセス制御ルールにより各プロセス特有の制御セマンティクスを記述する．またメッセージリンク上のイベント伝播ルールを用いた一貫性の管理手法を述べる．

Version Flow Model in Collaborative Design Process Management

Hiroki MATSUO　　Masaya OHKUBO　　Mizuho IWAIHARA
Department of Computer Science and Communication Engineering, Kyushu University,
6-1 Kasuga-Koen, Kasuga-Shi, Fukuoka 816, JAPAN
E-mail: {hmatsuo,ohkubo,iwaihara}@c.csce.kyushu-u.ac.jp

abstract　Design process management is gaining importance in recent distributed engineering environments, in which flows of design versions among processes ,instead of centralized design databases, should be carefully managed. In this paper, we discuss version flow management issues, and propose a process management scheme, called Version Flow Model. Messages enhanced with process state tags and links to versioned objects are used for user communication and version flow management. Process control rules describe domain-specific process control semantics, and consistency checking is done through event propagation on message links.

1　はじめに

近年の計算機ネットワークの普及に伴い，遠隔地を高速ネットワークで結び，様々な協調作業を計算機で支援することが可能になりつつある．ビジネス分野におけるワークフローを用いた定型的処理の効率化などは研究が盛んな分野の一つである [6]．

CAD/CASE などの設計分野においてもプロセス管理に計算機支援を取り入れ，プロセス管理を組み込んだ CASE ツールにおける並行処理制御のための様々なモデルが提案されている [7]．

CAD/CASE 環境では拡張トランザクションモデル [3] が研究されており，長時間トランザクションやバージョン管理 [4] といった問題が検討されている．設計プロセス管理のような制御構造が与えられるようになっている現在，これらの技術を，トランザクションモデルで議論されてきた一貫性管理とどのように統合するかが主要な課題となるであろう．

1.1　バージョンフロー管理

プロセス間で受渡される種々の設計オブジェクトについてそのデータの授受関係や生成/消滅過程を依存関係として管理する必要がある．このような設計プロセス間での設計オブジェクトの依存関係をバージョンフローと呼ぶことにする [12]

ビジネスプロセスと異なり設計プロセスでは多くのバージョンを扱う必要があるが，今までワークフローモデルにおいてバージョンを扱った問題の詳細は議論されていない．

1.2　プロセス内の一貫性管理

設計プロセスのなかにはシミュレーションやレイアウトのように数日から数週間といった長期に渡るものもある．これらのプロセスの結果は，論理設計や改良のためのレイアウト設計などの上流プロセスで利用され，デザインループは期待した性能が得られるまで繰り返される．

実際の状況では急な仕様変更やバグレポートなどの予期しえない事象によりしばしばプロセスは中断される．稼働中のプロセスのアボートには多大なコストがかかり，他のプロセスへの一貫性が保てない場合でも古い入力のまま作業を続けることもある．このような状況に対しバージョンフローによる一貫性モデルを与えることが考えられる．

SAGAS[3] やトランザクション的ワークフローモデルのなどの拡張トランザクションモデルでは，失敗に対して前向き復旧 (forward recovery) を行なう．しかし複雑な CAD トランザクションに対して前向き復旧を定義することは困難な場合も多い．我々のアプローチではメッセージを利用することにより一貫性の保たれているメッセージリンクの範囲を制御する手法をとる．

1.3　コミュニケーションとオブジェクト管理の統合

設計オブジェクトの受け渡しに伴ない，メイルや電話，ファクスなどを用いて多量の通信が行なわれる．その内容は，ツールの使い方などの初歩的な質問から，データ受け渡し方法の確認，設計上の問題点の指摘，仕様変更の解説，設計制約違反の報告など設計オブジェクトに関するものや，さらに署名

の伴なう設計移行承認書など，設計責任の所在を明確にするための書類も存在する．このように設計者間のコミュニケーションを図るためのメッセージは，設計の目的であるオブジェクト (バージョン) の受け渡しよりも頻度の面からは，はるかに多い．これらのメッセージを管理し，バージョンと関連づけることにより，より効率的な設計支援が行なえる．

1.4 バージョンフロー・モデル

我々は協調設計環境のためのバージョンフロー・モデルを提案している [12]．バージョンフロー・モデルは以下の構成要素からなる．

- プロセス，メッセージ，ハンドル
- メッセージリンク
- プロセス制御ルール，イベント伝播ルール

メッセージは利用者間のコミュニケーションに利用される，プロセスの状態，オブジェクトのバージョンはメッセージと共に記録され，バージョンフローを表現する．ハンドルはメッセージから設計オブジェクトへのリンクでインターフェースとして利用される．

プロセス制御ルールはプロセスの状態遷移を制御し，プロセスの実行によりメッセージのネットワークを生成する．新しいメッセージを生成する場合，それは状態の更新を意味する．イベント伝播ルールはイベント伝播関数により更新の影響を解析する．イベント伝播関数とは衝突の程度を示すイベントランクを返す関数であり，メッセージリンク型ごとに定義される．

本稿の手法は，バージョンを持つオブジェクトへのリンクを持つメッセージを用いた協調作業全般に適用することができる．

まず2節で複合オブジェクトやバージョンモデルなどの基本的な概念を述べる．次に3節でメッセージ，プロセス，ハンドルを定式化する．4節ではプロセス制御ルール，イベント伝播ルールを定義し，一貫性を維持する方法を示す．最後に5節でまとめを述べ，関係する研究との比較を行なう．

2 基本的概念

2.1 複合オブジェクトのモデル

定義 1 オブジェクトは以下の形式を持つ組である．

$$o = [OID : o, A_1 : v_1, A_2 : v_2 ..., A_k : v_k]$$

各オブジェクトは唯一に定まるオブジェクト識別子 (OID) o を持つ．A_iは属性と呼ばれ，定義域と呼ばれる値の集合が割り当てられている．v_i は値と呼ばれ，属性 A_iの定義域の要素である．

意味的にまとまりのあるオブジェクトの集合を表わすために，名前(クラス名) を与えられたオブジェクト集合をオブジェクト・クラスと呼ぶ．本稿では，オブジェクト識別子には小文字 o_1, o_2等を用い，オブジェクト・クラスには大文字 O_1, O_2 等を用いる．オブジェクト・クラスの全体集合を $\mathcal{O}$で表わす．

定義 2 リンクは $ln\langle o_1, o_2\rangle$ の形式を持つオブジェクト識別子上の関係である．ここで ln はリンク名と呼ぶ．リンク名の全体集合を $\mathcal{L}$ とする．

2.2 バージョンモデル

バージョンモデルの基本的な点は [4] を参考にしている．

部品関連 $part\langle o_i, o_j\rangle$ というリンク型を用いて，ひとつのオブジェクトに o_i(親オブジェクト) に0個以上のオブジェクト o_j(子オブジェクト) を関連づける．ひとつのオブジェクトは複数の親オブジェクトを持つことが可能である．

ひとつのオブジェクト o_jから部品関連を推移的に適用して得られるオブジェクトの集合を o_jを根とする階層オブジェクトクラスと呼び，$hier(o_j)$ で表わす．履歴関連 $hist\langle o_i, o_j\rangle$ というリンク名によりオブジェクト識別子の二項関係を保持し，これによりオブジェクト o_iを更新して o_jが得られたことを表わす．履歴関連 $hist\langle o_i, o_j\rangle$ もひとつの履歴関連を有向枝とし，オブジェクト識別子を節点とする有向非巡回グラフとして表現される．その有向非巡回グラフにおいて o_iから o_k への有向パスがある場合，$o_i \leq_h o_k$ と表記し，2 つの識別子の間の推移的な履歴関連を表わす．履歴関連の有向グラフに対し，その枝の向きを無視して得られる無向グラフにおいて，ある識別子 o_iと連結な識別子の集合を o_iの等価オブジェクトクラスといい $equiv(o_i)$ と表わす．

3 プロセスとメッセージのモデル化

3.1 概要

意味的なまとまりのある作業をプロセスと呼ぶ．すべての作業はメッセージによって記録され，メッセージはそのメッセージ列に加えられる．

プロセスクラスとは同一機能を持つプロセスの集合であり，唯一のプロセスクラス名を持つ．プロセスクラス名は例えば，「提案」(proposal),「評価結果」(evaluation result),「設計変更」(designchange),「バグ追跡」(bug chase),「改善要求」(improverequest), といったものからなる．

メッセージは本体 (テキスト/音声/画像) や日付をといった電子メールと同様の情報を持ち，参加者によって生成される．各メッセージはハンドル集合を持つ，ハンドル集合とは設計データ，ドキュメントへのリンク集合である．

それぞれのメッセージは状態を持ち，それらはプロセスに属する．各プロセスクラスにはそのプロセスで起こり得るメッセージ状態集合が割り当てられている．

メッセージの生成はメッセージ管理ルールで制御される．参加者がメッセージや既存のメッセージに関連づけるリンクを生成する要求を出すと，メッセージ管理ルールは生成できるメッセージ状態を与える．

コメントのように拘束力のないメッセージは，自由に生成され，他のメッセージにリンクされる．しかし設計移行承認，エラー報告のような拘束力のある公式のメッセージはルールにより制御され，監視される．

プロセスには，議論の経過を示すために「提案」,「意見」,「賛成」,「結論」等の構造化議論モデル (gIBIS)[1] と同様の状態集合を用意する．

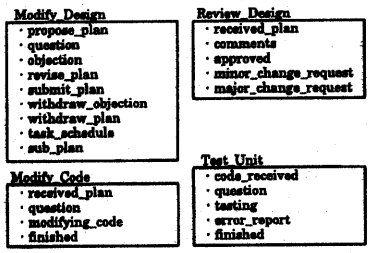

図 1: プロセスクラスとメッセージ状態

図1はプロセスクラスとメッセージ状態の例である．これらのプロセスクラスにはソフトウェアプロセスのベンチマー

ク例題を利用してある [9]．その例にはソフトウェアプロセス記述言語が使われており，これに必要なメッセージ状態を付加した．

3.2 プロセス

プロセスは以下の属性からなるオブジェクトである．

$$[pID : p, Pclass : P, Members : M]$$

ここで pID はプロセスのオブジェクト識別子である．各プロセスはそれぞれ一つのプロセスクラス (process class) に属する．プロセスクラスの全体集合を $\mathcal{P}$で表す．

$Menbers(p)$ は参加者集合の ID でありプロセス p に参加している人を表わす．

3.3 メッセージ

メッセージは以下の属性からなるオブジェクトである．

$$[mID : m, Sender : d, Process : p, State : s, \\ Handles : H, Body : b]$$

ここでメッセージのオブジェクト識別子を特にメッセージ識別子 (mID) と呼ぶ．$Sender(m)$ は m を作成した参加者の名前であり，$Menber(p)$ の要素である．$Body(m)$ はメッセージ本体である．

$Handle(m)$ はメッセージ m が使用するハンドル (3.6 節参照) の集合である．

メッセージは生成の時刻順にメッセージ列 $\mathcal{M} = [m_1; m_2; ...]$ に追加される．メッセージ間の生成の時刻順を表わす全順序 $\leq_t$ もメッセージ識別子間に定義される．つまり m_1 が m_2 の前に生成されたならば $m_1 <_t m_2$ が成り立つ．

3.4 メッセージ状態

各メッセージ m はメッセージ状態集合 $\sum$ の要素をメッセージ状態 (m 状態あるいは単に状態と略) $State(m)$ として持つ．また各プロセスクラス P について，それに属するメッセージの状態集合 $\sum^P \subset \sum$ が定められており，P の各メッセージ m について $\sum^P$ の状態が成り立つ．

3.5 メッセージ・リンク

メッセージを始点または終点とするリンクをメッセージリンク (m リンクと略) と呼ぶ．

メッセージ列 $\mathcal{M}$ は節点のラベルが mID で枝のラベルが m リンク名からなる有向グラフで記述される．これをメッセージネットワークと呼ぶ．

リンクごとに取り得るメッセージ状態を定め m リンク型として導入する．$\mathcal{L} \times \sum \times \sum$ 上の関係 $mlTypes$ を (m リンク型関係) と呼び，これにより組 $[ln, s1, s2]$ が $mlTypes$ に含まれるならば，m 状態 s_1 を始点とし，m 状態 s_2 を終点とする m リンク名 ln の m リンクが存在できることを表わす．

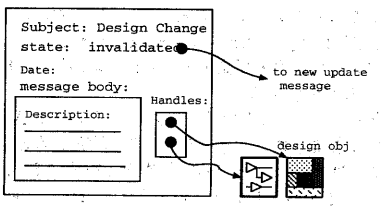

図 2: メッセージ形式

3.6 ハンドル

ハンドルをメッセージとオブジェクトのインターフェースとして定義する．利用者はオブジェクトを関連づけるためにメッセージ中にハンドルをおく．各ハンドルは一つのプロセスに属しそのプロセス中のメッセージもそれを所有する．ハンドルは次の目的で用いられる．

- メッセージが生成されたときに参照していたオブジェクトのバージョンを記録する．
- プロセス内からのデータベースへのビュー．
- 合意事項や To-Do リストのような拘束力のあるメッセージを，ハンドルから参照するオブジェクトとし，更新される場合の影響を監視する．
- 構成 (configuration) として，一貫性を保持する単位としてオブジェクト集合をひとつにまとめる．

定義 3 ハンドル型 とは以下のオブジェクトクラス名 $O_i (i = 1; ...; k)$ を要素とする組である．

$$H = [O_1, ..., O_k]$$

ハンドルは以下の形式のオブジェクトである．

$$[OID : h, Message : m, Object : [O_1 : o_1, ..., O_k : o_k]]$$

ここで h はオブジェクト識別子，m はメッセージ ID で必須である．属性の $Object$ は O_i に対応するオブジェクト識別子 o_i の組である．

$o_i \in Object(h)$ を省略して $O_i \in h$ と表す．

3.7 ハンドル操作

プロセス/メッセージ間をハンドルが移動することによりバージョンフローが表現される．

まずハンドル間の集合論的関係の定義を行なう．以下では h, h_1, h_2 はハンドルを o, o_1, o_2 はオブジェクトを表すものとする．

定義 4 1. $h_1 \subseteq h_2$ iff $\forall o_2 \in h_2, \exists o_1 \in h_1; o_2 \in hier(o_1)$.

2. $h_1 =_h h_2$ iff $h_1 \subseteq_h h_2$ and $h_2 \subseteq_h h_1$.

3. $h_1 \leq_d h_2$ iff $\forall o_1 \in h_1, \exists o_2 \in h_2, o_1 \leq_d o_2$ or $o_1 = o_2$.

ハンドルリンク型 $trans_\theta\langle h_1, h_2\rangle$ は $\theta \in \{\subseteq_h, =_h, <_d\}$ に対して $h_1\ \theta\ h_2$ もしくは $h_2\ \theta\ h_1$ であることを示す．

またハンドルの分割，統合を表す $split$ と $join$ を定義する．

1. $h_2 \subseteq h_1$ である場合に $split\langle h_1, h_2\rangle$ に対して $split\langle h_1, h_2'\rangle$ ならば h_2 と h_2' は共通するオブジェクトを持たない．
2. $h_1 \subseteq h_2$ である場合に $join\langle h_1, h_2\rangle$ に対して $split\langle h_1', h_2\rangle$ ならば h_1 と h_1' は共通するオブジェクトを持たない．

例 1 図 3はハンドルリンクの例を示す．ここではメッセージ m_1 のハンドル h_1 がハンドル h_2 と h_3 に分割されハンドルリンクの $split\langle h_1, h_2\rangle$ $split\langle h_1, h_3\rangle$ が作成されている．

4 メッセージ管理ルール

メッセージの生成とプロセスの実行を管理するためにメッセージ管理ルール $\mathcal{R}$ を定義する．$\mathcal{R}$ はプロセス制御ルール $\mathcal{R}_p$ とイベント伝播ルール $\mathcal{R}_e$ の 2 つからなる．

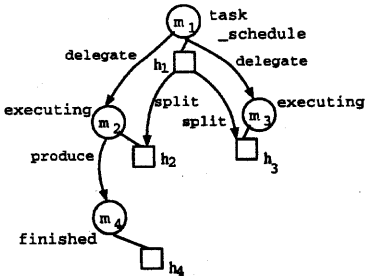

図 3: ハンドルリンク

4.1 プロセス制御ルール

まず $\mathcal{R}_p$ について述べる．$\mathcal{R}_p$ は $\mathcal{M}, \sum, \mathcal{L}$ を定義域とする制約集合である．前述したように $\mathcal{R}_p$ は $\leq_t$ を含む．

$\mathcal{R}_p$ は新しいメッセージ m の状態が生成済みのメッセージ列 $\mathcal{M}$ に対して生成可能であるかどうかを決定し，メッセージ状態の正しい遷移を定義する．

以下にプロセス制御ルールの例を示す．まずルール記述のための述語を示す．ここで m リンク型の集合を L，プロセスクラスを P，メッセージ状態集合を S とする．

- $current(P, x)$: (最新のメッセージ)
  プロセス P 中のメッセージ x のうち x を始点とする m リンクのないものについて真となる．
- $mlpath(L, x, y)$: (パスが存在)
  メッセージ x と y との間に，リンク名が L に含まれる m リンクで構成されたパスが存在するとき，または $x = y$ のとき真となる．

以下は制約の例である．

- $linear(P, L)$: (P 中で L は線形)
  プロセスクラスが P であるメッセージ x について，$ln \in L$ となる m リンク $ln\langle x, y\rangle$ は高々一つしか存在しない．
- $connected(P, L)$: (P 中で L は連結)
  プロセスクラスが P である任意のメッセージ x, y について，$mlpath(L, x, y)$，または $mlpath(L, y, x)$ が成立する．

例 2 L_1 がある分岐を持たないリンク型の集合であるとする．$linear$(Modify_code,L_1)，$connected$(Modify_code,L_1) というルールを付加することによりプロセスクラス Modify_Code の状態遷移は線形に制約される．

以下は m リンクの分岐を制御する制約の例である．

- $merge(L, s, t)$:(合流)
  $State(x) = s$ である任意のメッセージ x について $State(y_1) = State(y_2) = t$ かつ $mlpath(L, x, y_1)$ かつ $mlpath(L, x, y_2)$ であるメッセージ y_1, y_2 が存在するならば $y_1 = y_2$ である．つまり状態 s のメッセージから分岐したすべてのパスは状態 t のメッセージで合流する．
- $selective(L, s, t)$:(選択性)
  状態 s のメッセージから状態 t のメッセージへに到達できるパスは一つだけである．

例 3 図 4の例ではプロセス p2 において，状態 submit_plan のメッセージに到達するためには，すべての objection:(反対) の状態が withdraw: (撤回, 状態 withdraw_plan) されなければならない．分岐している意見はすべて submit_plan または withdraw_plan に合流 (ルール $merge$) しなければならない，という同期に関する制約が充足されている．

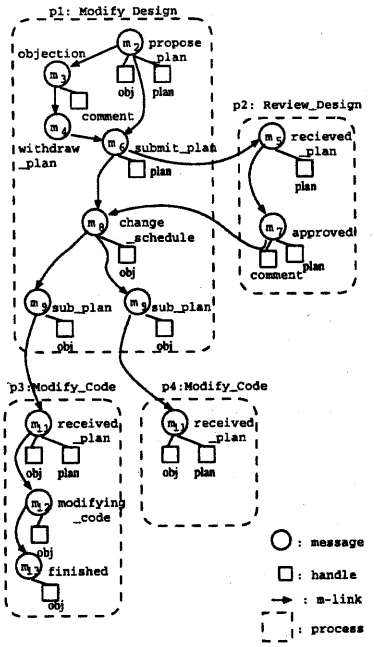

図 4: メッセージ ネットワーク

4.2 イベント伝播ルール

4.1節でのプロセス制御ルールはメッセージ生成での次のメッセージ状態を決定するものである．一方イベント伝播ルール $\mathcal{R}_e$ はオブジェクトの更新やそれに伴うアクセスの競合を検出する．$\mathcal{R}_e$ は各リンク型それぞれに対しイベント伝播関数を定義する．メッセージ生成による影響をメッセージネットワークの解析により計算する．解析の結果は，例えばメッセージ生成の拒否，生成済みメッセージのアボート，生成済みリンクの依存関係の変更といったトランザクションに利用できる．

新しいメッセージ m により引き起こされる影響をイベントと呼び，m リンクでのイベントの伝播によるメッセージへの影響を計算する．各メッセージにイベントクラスという記号を割り当て，イベントの重要度を表現させる．イベント伝播関数はイベントランクに基づき記述する．

定義 5 集合 $\mathcal{E} = \{E_1, ..., E_k\}$ をイベントクラスと呼び，各要素 E_i をイベントランクと呼ぶ．$\mathcal{E}$ には全順序 $\leq_\mathcal{E}$ が与えれれているものとする．

以下ではイベントクラスとして $\{Null, Notice, Urgent\}$ を用い，$Null <_\mathcal{E} Notice <_\mathcal{E} Urgent$ とする．

イベントに対する反応 (リアクション) はメッセージとしてシステムが生成する．

例としては，$Null$ では“メッセージなし”，$Notice$ では“関連する参加者にメッセージを送り，状態更新を制御する”，$Urgent$ では“メッセージの状態更新を拒否する”といったものがあり，各イベントランクの重要度を表現している．

定義 6 リンク $ln\langle m_1, m_2\rangle$ 上のイベントは形式

$$e = [E, O, ln\langle m_1, m_2\rangle],$$

の組で表される，ここで E はイベントランク，$ln\langle m_1, m_2\rangle$ は

mリンクである．O はオブジェクトの集合であり，影響を受けたオブジェクトと呼ぶ．

定義 7　イベント伝播関数 $\langle E, O\rangle = Prop(m, e)$ とは，入力であるイベント e とメッセージ m からイベントランク E と影響を受けたオブジェクトの集合 O を計算する関数である．

$Prop(m, e)$ はイベント e がメッセージ m に到達した際にイベントランク E を計算するために使われる．$Prop(m, e)$ により利用者はプロセス特有の実行規則を記述する．

これらの実行規則を記述するうえにおいて，メッセージ間の依存関係の重要度をイベントランクに対応させるようにする．例えば，ドキュメントの参照などの弱い依存関係には低いイベントランクを，シミュレーションの入力などの強い依存関係には高いイベントランクを与えるものとする．

例 4　例1のリンクに対するイベント伝播関数の例を示す．図5はリンク型 *split* に対する関数の定義を示す．ここでは入力 *Urgent* に対してハンドル h_2 が影響を受けたオブジェクト，またはそれから派生したオブジェクトを含むときのみ *Urgent* を返す．

```
Input: m2, e = [E, O, ln(m1/h1, m2/h2)], where ln = split
Output: E', O'
begin
    O' ← O;
    case E is
    Urgent:
            begin
            if (∃o2 ∈ h2, ∃o1 ∈ O, o1 ≤d o2) then E' ← Urgent
            else E' ← Notice;
            end;
    Notice:
            E' ← Notice;
    Null:   E' ← Null;
    end case
end
```

図 5: リンク型 *split* のイベント伝播関数

図6はリンク型 *produce* に対する関数の定義を示す．ここでは関与しているプロセスの現在の状態によってランクが決定される．プロセスが実行中のときに入力が変更されると，関数はは *Urgent* を返す．一方プロセスがすでに完了している場合は，完了したプロセスはコミットされているので，関数は *Notice* を返す．

```
Input: m2, e = [E, O, ln(m1, m2)], where ln = produce
Output: E', O'
begin
    O' ← O ∪ (∪h∈Handle(m) Object(h));
    case E is
    Urgent:
            begin
            if (∃x(current(Process(m2), x)), State(x) ∈
                {executing, finished) then E' ← Urgent
            else E' ← Notice;
            end;
    Notice:
            begin
            if (∃x(current(Process(m2), x)),
                State(x) = executing then E' ← Notice
            else E' ← Null;
            end;
    Null:   E' ← Null;
    end case
end
```

図 6: リンク型 *produce* のイベント伝播関数

並行するリンクを含むメッセージネットワークにおいてイベントランクの決定のためにいくつかの機構が必要である．イベント伝播関数を定義するうえで以下の制約を与えることにする．

1. イベント伝播関数は引数のイベントよりも高い優先順位のイベントクラスを与えることはない．
2. 1つのメッセージにイベントを伝播する経路が複数ある場合は，もっとも優先度の高いイベントランクを与える経路からの影響のみを考慮する．
3. 複数の経路からのイベントが同じイベントランクを与える場合は，そのイベントランクを採用し，伝播する複数のハンドルは和集合を取り1つのハンドルとする．
4. ハンドルの伝播方向はつねにメッセージの生成順と同じ方向であり，逆方向に伝播することはない．

制約1はイベント伝播の起点となるメッセージからは，イベントランクの優先順位が単調非減少で変化することを表わす．制約2と3は複数の経路がある場合の対応であり，制約4はデータの依存関係は時間的な順序と矛盾することはないという自然な制約を表現している．

以上に基づいたイベント伝播アルゴリズムを図7に示す．結果として各メッセージ m ごとにイベントランクが $E(m)$ に計算される．ここでは，例4のように制約(1)は関数 $Prop(m, e)$ の定義中で実現されていることを前提とする．制約(2),(3),(4)は図7のそれぞれ(A),(B),(C)で実現されている．

性質 1　イベント伝播アルゴリズムは各メッセージに一意のイベントランクを計算する．

例 5　図3のメッセージへのイベントランクを計算してみる．ここではランク *Urgent* のイベントが m_1 に与えられていることにする．m_1 は影響を受けたオブジェクト $O = \{o_1\}$ のオブジェクトを持ち，$o_1 \in h_1, o_1 \in h_2, o_1 \notin h_3$ であるとする．ここではリンク *delegate* でのイベント伝播関数は省略し *delegate* を通しての伝播は無視する．

Urgent を持つ m_1 からはじめる．まず $E(m_2) = Urgent$ と $E(m_3) = Notice$ を得る．次にリンク $produce(m_2, m_4)$ 上の伝播を計算すると，$E(m_4) = Urgent$ を得る．

4.3　一貫性モデル

新しいメッセージ m を生成する要求がなされると，プロセス制御ルール $\mathcal{R}_p$ を用いて正当な状態更新であるかの検査を行なう．そしてイベント伝播ルール $\mathcal{R}_e$ を適用し，既存のメッセージに対してイベントランクを計算する．

ここで次の一貫性モデルを導入する．

定義 8　E をイベントランクとする．$E(m)$ は入力 m, $ln\langle m_1, m\rangle, E$ に対する評価の結果であるとする．そのとき，もし各 $m' \in \mathcal{M}$ に対して $m' \neq m, E(m') <_\varepsilon E$ であるならば $\mathcal{M}$ に対して生成された m と $ln\langle m_1, m\rangle$ は *E-consistent* であるという．

例えばもしすべての m' に対して $E = Urgent$ かつ $E(m') = Notice$ ならば *Urgent-consistent* であるという．

本稿の一貫性モデルはイベント伝播ルールに基づいている．またイベントランクは様々な影響の重要度を比較するための基準として用いられる．

以下の性質はイベント伝播アルゴリズムの構成から得られる．

性質 2　メッセージ m とランク E からのイベント伝播に対し，各 $m' \in \mathcal{M}$ について $E(m_i) \geq E(m_{i+1})(i = 1, ..., k-1)$ となるメッセージ $m_1 = m, m_2, ..., m_k = m'$ 上のmリンクが存在する．

利用者は上述のパスをランクが割り当てられた理由付けとみなす事ができる.

```
Input: A message m_0 and e_0 = [E, O, ln(m'_0, m_0)]
Output: An event rank to each message in M
begin
    foreach m ∈ M
    begin
        E(m) ← Null;   O(m) ← ∅;
    end;
    W ← {[e_0, m_0]};
    while W ≠ ∅ do
    begin
        W' ← ∅;
        foreach [e, m] ∈ W do
        begin
            assume e = [E, O, ln(m_1, m)];
            W ← W − {[e, m]};
            (E', O') ← Prop(m, e);
(A)         if E(m) ≤_ε E' then
            begin
                E' ← E(m);
                if E' =_ε E(m) then
(B)                 O' ← O' ∪ O(m);
                O(m) ← O';
                foreach link ln'(m', m) or ln'(m, m') which
                    is adjacent to m and ln' ≠ ln
                    and E(m') ≤_ε E
                begin
                    if m' <_t m then
(C)                     O'' ← ∅;
                    else
                        O'' ← O';
                    e' ← [E', O'', ln'];
                    W' ← W' ∪ {e'};
                end
            end
        end;
        W ← W';
    end
end.
```

図 7: イベント伝播アルゴリズム

5 おわりに

最後にまとめと関連研究を述べる.

我々の分散設計環境はメッセージリンクの管理に基づく. これはプロセス制御ルールとイベント伝播ルールの管理からなる.

メッセージネットワークは現在のプロセスの状態と過去の履歴を表現し, 利用者のより良い理解と制御のしやすさを実現する.

オブジェクトのバージョンがハンドルによりメッセージから関連づけられ, 利用者間で異なったバージョンを見ることにより生じる誤解を避けることができる.

矛盾の生じる更新はイベント伝播関数が割り当てられたmリンクを解析することにより検出される.

ConTract モデル [11] ではデータベース動作がループや条件付の分岐などの多くの制御スクリプトの集合でモデル化されている. しかし例外の多い設計プロセス管理の記述にそのような静的なスクリプトは適していない. それに対しバージョンフローモデルではプロセス制御ルールにより比較的自由にメッセージネットワークを生成できるようにしている. そして生成されたメッセージネットワークを依存関係の集合ととらえ, そのうえで矛盾を検出する方法を取っている.

Cooperative activity モデル [5] では compatibility ルールを用いて, 並行的な作業の結果をマージする. このモデルは一つの共有データベースと他数の独立した個人用ワークスペースを利用する. しかしこのモデルでは実際の設計プロセスで現れるマルチバージョンを用いた作業を詳しく議論しておらず, またプロセス間の一貫性の定義法も考慮されていない.

今後, バージョンフロー・モデルの理論的な性質を明らかにし, 実際の利用者の意図にあったイベント伝播ルールの記述を行なうことが予定である.

参考文献

[1]J. Conklin and M. L. Begeman, "gIBIS: A Hypertext Tool for Exploratory Policy Discussion," *ACM Trans. Office Information Systems*, Vol.6, No.4, pp.303-331, Oct 1988.

[2]ECMA and NIST, "Reference Model for Frameworks of Software Engineering Environments," Draft Edition 3 of Technical Report ECMA TR/55 and NIST Special Publication 500-201, 1993.

[3]A. K. Elamagarmid (Ed.), "*Database Transaction Models for Advanced Applications*, " Morgan Kaufman,1992.

[4]R. H. Katz, "Toward a Unified Framework for Version Modeling in Engineering Databases," *ACM Computing Surveys*, Vol.22, No.4, pp.375-408, Dec. 1990.

[5]M. Rusinkiewicz, W.Klas, T.Tesh, J.Waesch nad P.Muth, "Towards a Cooperative Transaction Model – The Cooperative Activity Model–," *Proc. 21st VLDB Conf.*, Zurich, 1995.

[6]C. Mohan, G. Alonso, R. Guenthoer, M. Kamath, B. Reinwald, "An Overview of the Exotica Research Project on Workflow Management Systems," *proc. 6th International Workshop on High Performance Transaction Systems*, Asilomar,Sept. 1995.

[7]C. Montangero (ED.), "Software Process Technology, Proc. 5th European Workshop, EWSPT'96," *Lecture Note in Computer Science*, Springer, 1996.

[8]橋本 孝幸, "LSI 開発環境における遠隔協調設計環境の構築に関する検討," 九州大学大学院総合理工学研究科 修士論文, 1996 年 2 月

[9]M. I. kellner, et al., "Software Process Modeling Example Problem," *Proc. 6th Int. Software Process Workshop, pp.19–29, 1990.*

[10]"平成 6 年度 次世代技術教育・研究環境高度化システムの構築に関する研究調査報告書,"(財) 新機能素子研究開発協会, 1995 年.

[11]H. Wächter and A. Reuter, "The ConTract Model," in [3].

[12]岩井原 瑞穂, "分散協調設計環境におけるバージョンフロー制御に必要な機能," 重点領域研究「高度データベース」, 第 2 回研究集会論文集, Vol.2, pp.441–448, 1996 年 9 月.