# 双方向変換網における操作変換を用いたデータ更新競合解決手法

仲野, 祐希

(出版者 / Publisher)
法政大学大学院情報科学研究科
(雑誌名 / Journal or Publication Title)
法政大学大学院紀要. 情報科学研究科編
(巻 / Volume)
19
(開始ページ / Start Page)
1
(終了ページ / End Page)
9
(発行年 / Year)
2024-03-24
(URL)
https://doi.org/10.15002/00030619

# 双方向変換網における操作変換を用いたデータ更新競合解決手法

# Operational Transformation-based Conflict Resolution for Data Updates in Bidirectional Transformation Network

仲野 祐希 *

法政大学 情報科学研究科
yuki.nakano.3i@stu.hosei.ac.jp

## Abstract

Bidirectional transformation enables data synchronization between multiple sources, but handling multiple conflicting updates poses a challenge. Habu et al.[12] solve this problem using an operational transformation, but it is based on the convergence property $C_1$, which assumes that only two updates conflict. When a network of bidirectional transformations is used to synchronize more than two sources, more than two updates may conflict. In such cases, the convergence property $C_2$ is also required. However, the normal API which is based on inserts and deletes, cannot be used since it necessitates an additional parameter to the update API. To ensure confluence with more than two updates for the normal API (and thus without $C_2$), appropriate control is needed when operational transformations are composed. This study proposes a solution for handling multiple conflicting updates in a network of bidirectional transformations using controlled operational transformations. Our extension to the duplicate primitive "Dup" enables a 4-way merge in its backward transformation. We use Orderer, a mediator of the distributed ledger platform Hyperledger Fabric, to achieve the aforementioned control. Our implementation assumes a collaborative editing for XML via three views and we confirmed through experiments the well-behavedness of the bidirectional transformation including duplicates and that update intentions were preserved in the final result.

## 1 はじめに

双方向変換とは，2 つもしくはそれ以上の情報源の間で変換を介して一貫性を維持する仕組みのことである [8, 15, 5, 3]. 従来は 2 つの情報源の間での双方向変換が多く研究されてきたが，近年では，Asano et al.[6] や Diskin et al.[9] の研究のように，複数の双方向変換を組み合わせて 3 つ以上の情報源の間で一貫性を維持する多方向変換について研究されている．多数の情報源の間で一貫性を維持する場合，自然な双方向変換の結合方法として，図 1 のようにパターン①:*get* および *put* が直列に並ぶ場合，パターン②:*put* が向き合う場合，パターン③:*get* が向き合う場合の 3 通りの場合が考えられる．何れにおいても，接続点で競合が生じる可能性がある．ただし，パターン①では，直列合成した 1 つの双方向変換として扱われるため，両側からの同時更新を考える必要がなく，現実的には競合は問題とならない．これら 3 つのパターンのうち，パターン②について本研究の対象とする．パターン③においても，本研究の提案手法を用いて接続点での競合解決を行うことが可能ではあるが，本研究では対象としない．

複数の更新を合成する競合解決の手段としては，何らかの優先順位に基づき選ばれた更新のみを受け入れる方法も考えられるが，本研究では，公平性を期して，複数の更新をすべて合成できる操作変換 [10] を用いて競合解決を行う．土生ら [12] は，操作変換を応用したアルゴリズムを適用することで 2 つのビューの更新内容をどちらも考慮したデータ更新ができる双方向変換を実現しているが，利用する操作変換 [20] が 2 者間の競合のみを仮定した合流性 $C_1$ に基づいている．3 つ以上の更新には，追加の合流性 $C_2$ が必要であるが，通常用いられる挿入，削除に基づく API では満たすことが出来ないことが知られている [18]．その後 $C_2$ を満たす操作変換が提案されているが，更新 API に追加のパラメータが必要となる [19] ため，従来の API を用いる場合は $C_2$ に頼ることができない．本研究では 3 つ以上のビューの更新内容を考慮した双方向変換を実現することを目的としつつ，従来の API と $C_1$ のみの合流性を仮定し，操作変換の合成を適切に制御することでこの問題に対処する．本研究で扱う多方向変換の例として，図 2 のような，双方向変換が辺となるネットワークを想定している．ソースやビューは

* 指導教員：日高 宗一郎 教授

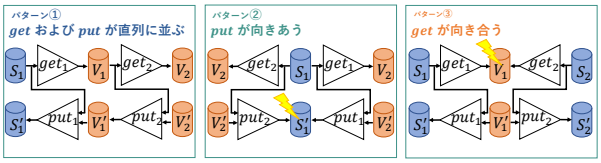



**図 1:** 自然な双方向変換の結合パターン

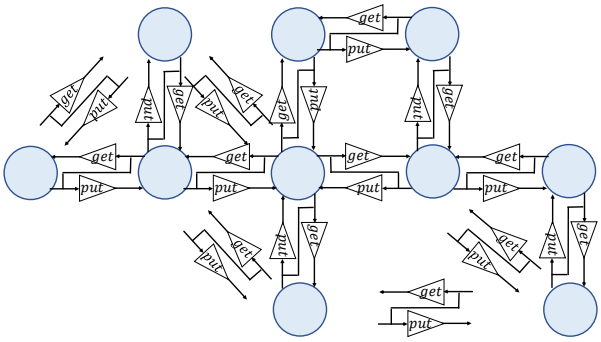



**図 2:** 本研究で想定する双方向変換網

ネットワークを介した多数のサーバーであると考えられる．

操作変換の合成の制御には分散台帳プラットフォーム Hyperledger Fabric[4] の調停機能である Orderer を活用する．競合解決機能は，複製機能を持つ双方向変換言語 X[13, 14] における複製結合子 Dup において逆方向変換で 4-way merge（3 つの更新された複製と複製元で計 4 つ）を行うことで実現し，XML に対する 3 つのビューを介した共同編集を想定した実装を行い，複製を含む双方向変換の well-behavedness と，競合した更新が最終結果に保持されていることを実験により確認した．以下，2 節で基本的な双方向変換，3 節で基本的な多方向変換，4 節で基本的な操作変換，5 節で実装の基盤として用いる Hyperledger Fabric の構成要素，6 節で設計，7 節で実装，8 節で評価実験，9 節で関連研究，10 節でまとめと今後の課題について述べる．

## 2 基本的な双方向変換の性質

### 2.1 双方向変換

双方向変換の基本的な性質 [11] について紹介する．双方向変換は 2 つの集合間の変換であり，一方をソース $S$，他方をビュー $V$ とする．$s \in S$ から関心あるデータを抜き出し $v \in V$ を作成する変換を順方向変換 $get : S \to V$ という．$v$ に何らかの更新を行い，更新による変更を $s$ に伝播する変換を逆方向変換 $put : V \times S \to S$ という．双方向変換は以下の 2 つの特性を満たしている．

$$put\,(get\,(s)\,,\,s) \;=\; s \qquad \text{(GETPUT)}$$

$$get\,(put\,(v,\,s)) \;=\; v \qquad \text{(PUTGET)}$$

GETPUT は抽出したビューが更新されないとき，逆方向変換後のソースも更新されないことを示す．PUTGET はビューのすべての情報をソースに伝播できることを示す．GETPUT, PUTGET を同時に満たす双方向変換を well-behavedness を満たす双方向変換という．

### 2.2 X 言語

Hu et al.[13] は，双方向変換言語である X 言語を提案している．X 言語では，2.3 節で紹介する Dup とよばれるコンビネータが定義されており，木に対する双方向変換のうち，複製を含むビューを扱うことが可能である．双方向変換のフレームワークの一つである Lens[11] においても，copy とよばれるコンビネータにより，複製を含むビューを扱うことが可能である．しかしながら，Lens では well-behavedness を満たす必要があり，copy の逆方向変換によりマージする際には，全ての複製に対して全く同じ更新を行わなければならない．本研究で扱う共同編集のシナリオでは，各ユーザーが複製されたビューを保持し，ビューに対して各々が別の更新を行うとみなすことができる．したがって，Dup のように複製に対して同じ更新を仮定しないコンビネータが必要となるため，本研究では X 言語を採用した．本論文で利用する Dup について 2.3 節で述べる．X 言語の詳しい構文や意味論については文献 [13, 14] を参照されたい．X 言語で定義されたコンビネータを組み合わせた変換は，以下の性質を満たす．

$$get\,(put\,(get\,(s)\,,\,s)) \;=\; v \qquad \text{(GETPUTGET)}$$

$$put\,(get\,(put\,(v,\,s))\,,\,s') \;=\; s' \qquad \text{(PUTGETPUT)}$$

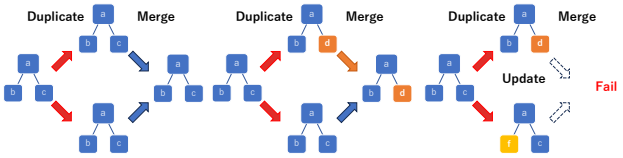
$$s' \;=\; put\,(v,\,s)$$

### 2.3 Dup

X 言語では，2.2 節で紹介する，Dup というコンビネータを用いて，ビューの複製を扱うことが可能である．図 3 に示すような Dup を用いた更新について考える．複製を扱う副作用として，図 4 に示すように，2 つの集合間での双方向変換で考えられる性質である，強い well-behavedness (GETPUT, PUTGET) を満たすことは不可能である．したがって，X 言語が満たしている，弱い well-behavedness (GETPUTGET, PUTGETPUT) について考える．2.1 節で紹介した，GETPUT, PUTGET をソースとビュー間で同期がとれるまでに必要な変換の回数の点で，制約を緩めた性質である．

### 2.4 Inv 言語と X 言語の埋め込み

Inv 言語 [17] とは，構造化文書を双方向に更新できる言語である．X 言語のコンビネータを Inv で表現することにより，順




**(a)** Dup を用いた更新（更新なし） **(b)** Dup を用いた更新（片方更新） **(c)** Dup を用いた更新（両方更新）

**図 3:** Dup を用いた更新

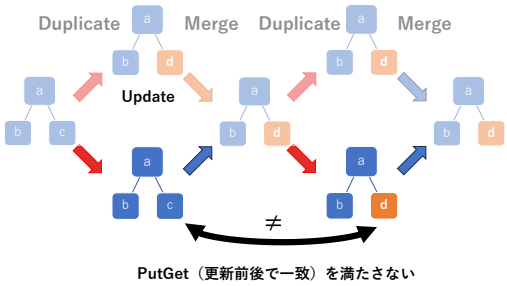



**図 4:** Dup が PUTGET を満たさない反例

方向・逆方向の両方の変換を行うことができる．ただし，Inv では複製を扱うため，単射でない変換も許容している．Hu et al.[14] では Inv へ X のプリミティブを埋め込むという形で実現されており，先行研究 [12] でも Inv を用いているため，本研究においても Inv を用いて実装する．本研究では，Inv 言語に新たな操作を追加する．Inv 言語は次のような構文を持つ．コンストラクタの詳しい説明は [14] を参照されたい．

$$\begin{aligned}\mathrm{Inv} ::= {} & \mathrm{Inv}^{\smile} \mid \mathit{nil} \mid \mathit{cons} \mid \mathit{node} \mid P? \mid V \mid \delta \mid \mathit{dupNil} \\ & \mid \mathit{dupStr}\ \mathrm{String} \mid \mathrm{Inv} \,;\, \mathrm{Inv} \mid \mathit{id} \mid \mathrm{Inv} \cup \mathrm{Inv} \mid \mathrm{Inv} \times \mathrm{Inv} \\ & \mid \mathit{assocr} \mid \mathit{assocl} \mid \mathit{swap} \mid \mu(V : \mathrm{Inv}) \mid \mathit{prim}(f, g) \\ & \mid \mathit{resC} \mid \mathit{extendResC}\end{aligned}$$

*resC* は土生の先行研究 [12] で新たに追加されたコンストラクタであり，2 つのビューの競合解決を行う際に利用される関数である．*extendResC* は本研究で新たに追加するコンストラクタであり，*resC* を拡張し，3 つのビューの競合解決を行う際に利用される関数である．*resC*，*extendResC* はともに，Inv における 2 複製・3 複製の逆方向変換，すなわち，複製されたビューをマージする際に動作することを想定している．

### 2.5 ドキュメントのモデル

2.2 節で紹介した X 言語では XML に代表される，構造化文書に対する変換が定義されている．X 言語で対象とする構造化文書のデータ型は以下の通りである．

$$\begin{array}{lcl} \mathit{Val} & ::= & \mathit{Atom} \\ & \mid & *\mathit{Atom} \mid \mathit{Val}^{+} \mid \mathit{Val}^{-} \\ & \mid & (\mathit{Val} \times \mathit{Val}) \mid [\mathit{Val}] \mid \mathit{Tree}\ \ \mathit{Val} \\ \mathit{Atom} & ::= & \mathrm{String} \mid () \\ [a] & ::= & [\,] \mid a : [a] \\ \mathit{Tree}\ a & ::= & \mathrm{N}\ a\ [\mathit{Tree}\ a] \end{array}$$

X 言語では，構造化文書に対する操作として，以下の 4 つのコマンドが想定されている．

$$\begin{aligned} \mathit{Command}\ a ::= {} & \mathrm{Insert}\ \mathit{Path}\ a \\ \mid {} & \mathrm{Delete}\ \mathit{Path}\ a \\ \mid {} & \mathrm{EditLabel}\ \mathit{Path}\ a \\ \mid {} & \mathrm{Stay} \\ \mathit{Path} ::= {} & [\mathrm{Int}] \end{aligned}$$

実装では *Val* が $a$ に対応している．*Path* は非負整数のリスト $[i_1, i_2, \ldots, i_n]$ で表される．木のルートの $i_1$ 番目の子，さらにその子の $i_2$ 番目の子といったように $n$ 回，木の子をたどって到達する際の部分木を指している．*Command* のそれぞれの編集操作は次のように動作する．Insert は *Path* の位置に $a$ を挿入する．Delete は *Path* の位置の部分木を削除する．Delete において，引数 $a$ は動作に影響しないが，任意の操作にこれを取り消す操作が定義されるという性質 $IP_1$ を満たす操作変換の実装に合わせるために引数 $a$ は存在している．EditLabel は *Path* の位置のラベルを $a$ に更新する．Stay は何もしない．

## 3 基本的な多方向変換の性質

Asano et al.[6] などで，多数の双方向変換を組み合わせた多方向変換 [7] について研究されている．一般的に，個々の双方向変換は同期対象間に計算が介在したとしても，その計算により定義された一貫性を双方向変換で接続された対象間で保証している．多方向変換では，ソースやビューがネットワークを介した，分散環境が想定される．Asano et al.[6] は，分散環境において更新伝播を考える際に，Local Privacy，Global Consistency の 2 点について考えている．

**Local Privacy** サイトに保存されているデータの所有者が，どのデータを公開し他のシステムにおいてデータがどのように利用・更新されるか決定できること

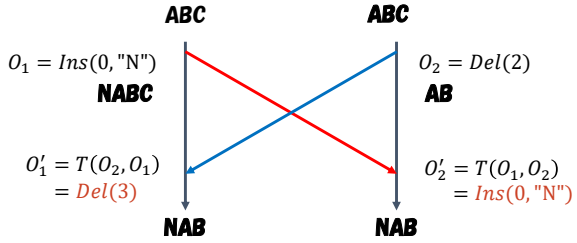
**Global Consistency** すべてのデータについて，システムがグローバルに一貫したビューを持つこと

Local Privacy を実現するためには，サイトをソース，共有スキーマをビューとする双方向変換を組む際の工夫が必要となる．本研究においては，文書の共同編集において文書全体をそのまま共有する応用を想定しており，Local Privacy は考慮していないため双方向変換部分は恒等変換を用いているが，恒等変換でない双方向変換を用いることにより本研究においても同様に Local Privacy を実現することができる．その他，競合解決の制御を行うコンポーネントに追加のデータアクセスの制御を行わせることでも原理的には Local Privacy を実現することが可能となるが，本研究では対象としない．

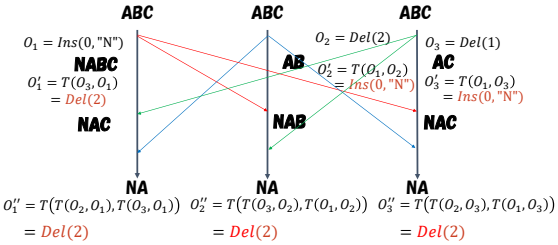
また，分散システムにおいてグローバルなデータ共有を考える際，グローバルで共有された 1 つのスキーマか，Peer ごとに異なるスキーマのどちらかが仮定される．どちらのスキーマを仮定した場合においても，合流する接点が存在することは避けられない．そのため，競合解決を考える必要がある．競合解決を行う手法の中で，合流する更新の反映の公平性を重視し，本研究では，複数の更新を平等に併合できる操作変換を利用する．

## 4 操作変換

文字列を対象とした操作変換 [20] では，挿入，削除といった操作を引数とする変換関数 $T$ により，互いの文字列が一致するような編集操作を算出することで，共同編集文書の一貫性を保つことができる．操作変換の主な特性として，データを更新する主体の数に応じて $C_1$ と $C_2$ がある．$C_1$ では 2 者によ




**(a)** $C_1$ に対応する例




**(b)** $C_2$ に対応する例

**図 5:** 操作変換の特性 $C_1$，$C_2$ に対応する例

る同時更新の場合に，$C_2$ では 3 者以上による同時更新の場合も，適用順によらず編集結果が等しくなるという性質であり，以下のように定義される．ここで，共同編集文書の初期状態を $S$ とし，各編集者の操作は $S$ に対するものであることを前提する．また，セミコロン（;）は操作の逐次合成を表す．

$$(O_1;\ T(O_2, O_1))\, S \ =\ (O_2;\ T(O_1, O_2))\, S \qquad (C_1)$$

$$T(T(O_1, O_2), T(O_3, O_2)) = T(T(O_1, O_3), T(O_2, O_3)) \qquad (C_2)$$

また，2.5 節で登場した性質 $IP_1$ は，共同編集文書の状態を $S$ として，ある操作 $O$ に対して，これを打ち消す操作 $\overline{O}$ が存在するという性質である．共同編集ソフトウェアにおいて Undo 機能を考える際に満たす必要がある．

$$(O\ ;\ \overline{O})\, S \ =\ S \qquad (IP_1)$$

図 5 は，共同編集ソフトウェアで起こる操作変換の過程を例示したものであり，先に述べた操作変換の特性 $C_1$，$C_2$ に対応する．操作変換の特性 $C_1$ のみが満たされている場合に，3 者以上がデータを更新すると，編集結果は操作の適用順に依存してしまう．この問題に対して，(1) 特性 $C_2$ を満たす (2) 何らかの方法で適用順を決定するという 2 つのアプローチが考えられる．Randolph et al.[18] は，文字列に対して挿入・削除を行う API について，特性 $C_1$，$C_2$ をともに満たす変換関数を実装することは不可能であることを示している．その後，$C_2$ を満たす操作変換を提案しているが，更新 API に追加のパラメータが必要となる [19] ため，従来の API を用いる場合は $C_2$ に頼ることができない．したがって，従来の API と $C_1$ のみの合流性を仮定し，操作変換の合成を適切に制御することでこの問題に対処する．

## 5 Hyperledger Fabric の構成要素

本節では，実装基盤として用いる Hyperledger Fabric[4] の構成要素について紹介する．

**Application** Peer 上の Chaincode を呼び出し，Ledger をクエリまたは更新する．

**Peer** Blockchain Network 上のノードを表す．Ledger と Chaincode を保持する．

**Chaincode** Blockchain Network 外部にある Application によって呼び出され，トランザクションを通してステートデータベースへのアクセスや更新を管理するプログラム．

**Ledger** 一般的なブロックチェーンと同様な構造を持つ Blockchain と，ステートデータベースの 2 つからなる．ステートデータベースはトランザクションを実行した結果得られる最新の状態を格納する．

**Orderer** トランザクションの順序を決定して Block を作成し，接続している Peer に Block を配布する．

**Organization** Blockchain Network に参加する組織を表し，Peer や Orderer は Organization に所属する．

**Channel** Blockchain Network を論理的に分割したネットワークである．2 つ以上の特定のネットワークメンバー間でプライベートで機密のトランザクションを実行できる．1 つの Blockchain Network 内に複数の Channel が存在することが可能であり，各 Channel 内で固有の Ledger を保持する．

## 6 設計

本節では，提案手法の基本的な考え方について述べる．本研究では，3 者による XML の共同編集を題材とする．各ユーザーが編集する文書をビューとして扱い，3 ユーザー分のビューを保持する．ユーザーを Peer として扱い，分散している状況を想定する．ユーザーが発行する Operation に対して，分散配置可能なコーディネータが操作変換の合成を調整する．分散配置可能なコーディネータを持つ Hyperledger Fabric[4] を実装基盤として利用する．

2.3 節で述べた，Dup というコンビネータを用いて，共通のソースから構築されたビューを扱うことが可能な双方向変換を実現する．3 つのビューに対し，双方向変換を行う関数を 6.1 節，3 つのビューの同時更新による競合解決を行う際に考慮する必要がある，操作変換の適用順について 6.2 節で紹介する．ユーザーが発行する編集操作に対して，操作変換を適用する順序を調整する役割を担うコーディネーターについて 6.3 節で述べる．コーディネーターによって決定された順序を保持する手法について 6.4 節で述べる．

### 6.1 双方向変換関数 triget, triput

双方向変換関数 *triget*, *triput* はそれぞれ，入力 $a$ を 3 つに複製する TriDup コンストラクタ，その逆方向変換を行う TriDup$^\smile$ コンストラクタにより，実現している．X 言語のコン

ビネータの Inv への埋め込みは，⌈ _ ⌉ で囲むことで表現する．TriDup の埋め込みは以下の通りである．

$$\lceil \mathrm{TriDup} \rceil = dup_a ; (id \times (dup_a ; (id \times dup_a) ; mittsu ; mkRoot))$$
$$\text{where } mittsu = (id \times (id \times (dupNil\,;\, cons)\,;\, cons))\,;\, cons$$
$$mkRoot = dupStr\text{ }"Dup"\,;\, swap\,;\, node$$

$dup_a$ が 3 度呼ばれる理由は，1 度目に入力をキャッシュし，2 度目以降で実際に複製を行うためである．$mittsu$ 関数の入力を $(x,(x,x))$ とすると，$x : x : x : []$ が出力となるように関数を設計した．リストの各要素は，ビューの状態を表すことを想定している．また，TriDup$^\smile$ の埋め込みは以下の通りである．

$$\lceil \mathrm{TriDup}^\smile \rceil = (id \times rmRoot)\,;\, extendResC$$
$$\text{where } rmRoot = node^\smile ;\, swap ;\, (dupStr^\smile\text{ }"\_dup")$$

$rmRoot$ 関数では，ビューの状態を表すリストを抽出している．$extendResC$ 関数では，競合解決を行う．ビューに対し，変更がなかった場合はそのまま，変更があった場合はそれを優先して返すのが基本方針である．ただし，複数の変更があった場合は，操作変換を利用し，競合解決を行った上で返す．3 つのビューのうち，2 つのビュー変更があった場合は，変更があるビュー同士で操作変換を行い，競合解決を行う．3 つのビューすべてで変更があった場合は，6.2 節で紹介する何れかの操作変換の並びで操作変換を適用し，競合解決を行う．

### 6.2　操作変換の適用順序

ここでは 4 節で述べた，特性 $C_1$ のみを満たされている場合に，3 者以上がデータを更新すると，編集結果は操作の適用順に依存してしまうという問題について詳しく見ていく．図 6 では，頂点が状態，辺は操作を表しており，辺をたどる経路は適用される操作の順序列を示している．図中では操作変換の特性は以下のように表現される．特性 $C_1$ のみが満たされている場合，2 つの操作が合流し，面となる．さらに特性 $C_2$ が満たされている場合，3 つの操作が合流し，直方体となる．本研究では特性 $C_1$ のみが満たされている場合を考えており，特性 $C_2$ が成立していないことは，例えば辺 $HG_1$ と $HG_3$ が一致していないことと対応している．この図から 3 者の場合，操作変換適用後の操作を実行した結果は高々 3 パターンであること，そのうちのどれかに導くために必要な操作変換の並びがわかる．これらをまとめると，次のようになる．

- 操作の実行結果として $G_1$ を選んだ場合
  1. $O_1\,;\, T(O_2, O_1)\,;\, T(T(O_3, O_1), T(O_2, O_1))$
  2. $O_1\,;\, T(O_3, O_1)\,;\, T(T(O_2, O_1), T(O_3, O_1))$
  3. $O_2\,;\, T(O_1, O_2)\,;\, T(T(O_3, O_1), T(O_2, O_1))$
  4. $O_3\,;\, T(O_1, O_3)\,;\, T(T(O_2, O_1), T(O_3, O_1))$
- 操作の実行結果として $G_2$ を選んだ場合
  1. $O_1\,;\, T(O_2, O_1)\,;\, T(T(O_3, O_2), T(O_1, O_2))$
  2. $O_2\,;\, T(O_1, O_2)\,;\, T(T(O_3, O_2), T(O_1, O_2))$
  3. $O_2\,;\, T(O_3, O_2)\,;\, T(T(O_1, O_2), T(O_3, O_2))$
  4. $O_3\,;\, T(O_2, O_3)\,;\, T(T(O_1, O_2), T(O_3, O_2))$
- 操作の実行結果として $G_3$ を選んだ場合
  1. $O_1\,;\, T(O_3, O_1)\,;\, T(T(O_2, O_3), T(O_1, O_3))$
  2. $O_2\,;\, T(O_3, O_2)\,;\, T(T(O_1, O_3), T(O_2, O_3))$
  3. $O_3\,;\, T(O_2, O_3)\,;\, T(T(O_1, O_3), T(O_2, O_3))$
  4. $O_3\,;\, T(O_1, O_3)\,;\, T(T(O_2, O_3), T(O_1, O_3))$

例えば，操作の実行結果として $G_1$ を選んだ場合では，頂点 A から D に進んだのち，C ではなく，必ず H に進む必要があることを示している．

双方向変換として扱うことを考えると，頂点 A の状態をソース，ビューの初期状態, D,B,E を 3 つの異なる更新後のビュー，それぞれ A との差分を $O_1, O_2, O_3$ として，$G_1, G_2, G_3$ のいずれかの頂点に向かわせることとなる．操作変換の適用順を決定することは，$G_1, G_2, G_3$ のいずれかの頂点に向かう際の経路を決定する意味を持つ．

### 6.3　コーディネーター

本節では，ユーザーが発行する編集操作に対して，操作変換の合成の制御を調整する役割を担うコーディネーターについて述べる．コーディネーターは各 Peer からの編集操作を受け取り，6.2 節で定義された操作変換の列を返す．本研究では，Hyperledger Fabric の台帳に対するトランザクションの調停機構である Fabric の Orderer を用いて調停を行う．その際，台帳は編集操作を蓄積するストレージとして用いる．Orderer は指定された時間範囲に到着した，前述の編集操作情報の挿入トランザクションを並べ替えて台帳の Block に挿入する．そこで，整順されたトランザクションの順列（3 者の場合 $3! = 6$ 通りの可能性がある）を，6.2 節で定義された操作変換の適用指示にマップする．この際，完全な経路を指示すると本来 $C_2$ で享受していた自由度が完全に損なわれる．例えば，図 6 における状態 $G_1$ へ向かう場合，Peer$_1$ は D に達した後 C を経由しても H を経由しても良い．そのため，完全な経路ではなく，このような自由度を許容した指示にマップする．

### 6.4　順序情報の埋め込み

Orderer が並び替えたトランザクションの順序をもとにして操作変換を適用するが，Orderer は並び替えたトランザク

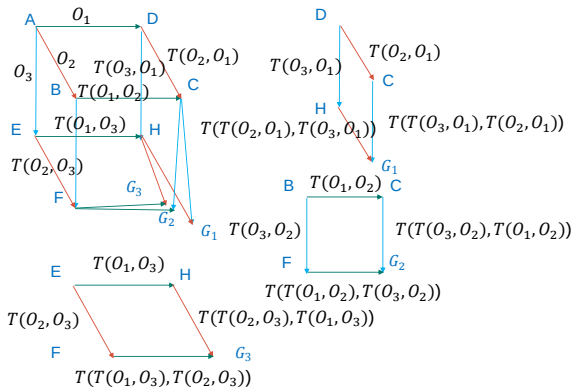



**図 6:** $C_1$ のみを満たした場合の 3 者での操作変換

ションの順序を直接出力しない．並び替えたトランザクション順序情報は Block に格納されることから，Block を保存する Ledger から読み出すことを試みたが，既存ライブラリの競合のため，Ledger からトランザクションの順序を取得することができなかった．Channel 上を流れる Block を読み出す listener が存在するが，読み出した結果を順序なしデータストア（CouchDB）に保存することから，順序情報が失われる．この問題に対して，(1)listener に順序情報を注入させる (2)CouchDB の変更通知機能を利用し，listener が読み出したトランザクションをデータストアに保存する前に（listener からの出力順に）捕捉する，2 つの手法を検討するが，いずれも実装に至っていない．

## 7　実装

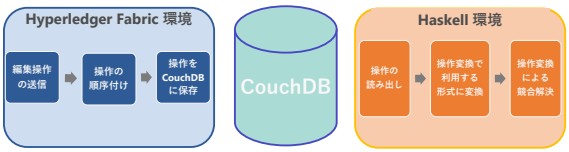



**図 7:** 実装ステップ

図 7 に示す通り，実装は大きく分けて Hyperledger Fabric 環境と Haskell 環境の 2 つからなる．各環境で行われる処理の概要を示す．

### 7.1　Hyperledger Fabric 環境

Hyperledger Fabric 環境では，図 8 のような 3 つの Organization からなる，Hyperledger Fabric Network を addOrg3 サンプルに基づいて構築する [1]．サンプルでは，Org1・Org2 からなる Test-Network に Org3 を追加するという構成になっており，本研究においても同様の構成をとる．本研究の実装では，同一文書を 3 人で共同編集する状況を想定している．各 Peer が編集者に対応する．編集者がおこなった操作は，Chaincode にしたがって Orderer に送信される．Orderer は Peer から受け取ったトランザクションの順序を決定してブロックを作成し，接続している Peer にブロックを配布する．そして，ブロックは各 Peer が持つ Ledger に保存される．本研究では，Orderer によって決定された操作の順序に基づいて操作変換を適用する方針をとる．そのためには，ブロックに格納されたトランザクションを読み出す必要がある．6.4 節で述べたように，Channel 上を流れる Block を読み出し (listener)，順序なしデータストア (CouchDB) に保存する方針をとる．これを実装している Offchain サンプル [2] に従った実装を行う．CouchDB を介して，Hyperledger Fabric 環境と Haskell 環境の 2 つを接続する．

### 7.2　Haskell 環境

Haskell 環境では，JSON 形式で CouchDB に格納された順序付けされた操作を読み出し，2.5 節で紹介した *Command* 形式に変換する．そして，Orderer により決定された順序に基づいて操作変換を適用し，競合解決を行う．

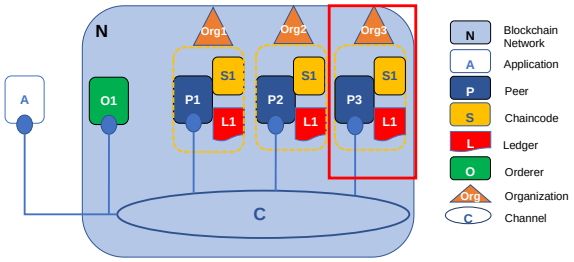



**図 8:** 3 つの Org を持つ Hyperledger Fabric Network の構築例
公式ドキュメント addOrg3[1] をもとに構成

### 7.3　トランザクション履歴の取得

本研究における Hyperledger Fabric を用いる際の関心事は，複数の編集者による編集操作の順序を Ordering Service により決定することである．すなわち，Ordering Service から各 Peer に送信される Block 内に保持される，トランザクション履歴を取得する必要がある．

## 8　評価実験について

Orderer が並び替えたトランザクションを listener で捕捉することは可能となったが，順序情報を注入する実装には至っていない．したがって，Orderer に順序決定させるために発行したトランザクションに，順序情報が埋め込まれていると仮定した API を設計し，決め打ちデータで実験を行う．現状の決め打ちデータは，トランザクション中のフィールド PeerID と Orderer が並び替えたトランザクションの順序情報を示す OrderID であり，これらのデータが CouchDB から読み出した JSON に含まれると仮定して，6.2 節で紹介した操作の適用順（図 6 における経路）を決定する．

双方向変換の性質を満たすこと，操作変換により競合解決を行うことができているか確認する．XML 形式のファイルを用意する．3 者以上により，同時刻に同じ XML 要素に対し更新を行った場合，検討した操作変換が期待する結果と本研究の提案手法により得られた結果が一致しているかどうかを評価対象とする．また，双方向変換の特性を満たしているか，PUTGETPUT，GETPUTGET に対応する動作を与えることによって調査を行う．以下の 3 つのテストを行った．

1. *triputtest* : $triget \rightarrow triput \rightarrow triget \rightarrow triput$ の順番で実行する．ビューに対し，必ず新たな編集操作を行った上で *triput* を実行する．この時，3 つのビューに対する異なる編集操作が競合解決された上でソースに反映されていることを確認する．また，2 回目の *triput* では前の操作の効果が保存された上で変換できているか確認する．
2. PUTGETPUT: $triput \rightarrow triget \rightarrow triput$ の順番で実行し，変換を通して情報の欠損がないことを確認する．
3. GETPUTGET: $triget \rightarrow triput \rightarrow triget$ の順番で実行し，変換を通して情報の欠損がないことを確認する．

実行の結果，PUTGETPUT，GETPUTGET を満たしている

**Listing 1:** ソース $s$

```
<a>
  <b>
    <c>d</c>
  </b>
</a>
```

**Listing 2:** *triget* 後のビュー $v$

```
<_dup>
  <a>
    <b>
      <c>d</c>
    </b>
  </a>
  <a>
    <b>
      <c>d</c>
    </b>
  </a>
  <a>
    <b>
      <c>d</c>
    </b>
  </a>
</_dup>
```

**Listing 3:** 更新後のビュー $v'$

```
<_dup>
  <a>
    <b>
      <c>d</c>
    </b>
    <xxx>yyy</xxx>
  </a>
  <a>
    <b>
      <c>d</c>
      <xx>yy</xx>
    </b>
  </a>
  <a>
    <b>
      <z>x</z>
      <c>d</c>
    </b>
  </a>
</_dup>
```

**Listing 4:** *triput* による更新後のソース $s'$ ($s_2$)

```
<a>
  <b>
    <z>x</z>
    <c>d</c>
    <xx>yy</xx>
  </b>
  <xxx>yyy</xxx>
</a>
```

**Listing 5:** $s_2$ に対し *triget* 後のビュー $v_2$

```
<_dup>
  <a>
    <b>
      <z>x</z>
      <c>d</c>
      <xx>yy</xx>
    </b>
    <xxx>yyy</xxx>
  </a>
  <a>
    <b>
      <z>x</z>
      <c>d</c>
      <xx>yy</xx>
    </b>
    <xxx>yyy</xxx>
  </a>
  <a>
    <b>
      <z>x</z>
      <c>d</c>
      <xx>yy</xx>
    </b>
    <xxx>yyy</xxx>
  </a>
</_dup>
```

**Listing 6:** 更新後のビュー $v'_2$

```
<_dup>
  <a>
    <wwww>
      <z>x</z>
      <c>d</c>
      <xx>yy</xx>
    </wwww>
    <xxx>yyy</xxx>
  </a>
  <a>
    <b>
      <z>x</z>
      <xx>yy</xx>
    </b>
    <xxx>yyy</xxx>
  </a>
  <a>
    <b>
      <iii>jjj</iii>
      <z>x</z>
      <c>d</c>
      <xx>yy</xx>
    </b>
    <xxx>yyy</xxx>
  </a>
</_dup>
```

**Listing 7:** *triput* による更新後のソース $s'_2$

```
<a>
  <wwww>
    <iii>jjj</iii>
    <z>x</z>
    <xx>yy</xx>
  </wwww>
  <xxx>yyy</xxx>
</a>
```

こと，検討した操作変換が期待する結果と本研究の提案手法により得られた結果が一致していることを確認した．ここでは *triputtest* のテストについて説明する．はじめに，テストにおける初期状態のソース $s$ と *triget* を行い生成したビュー $v$ は（Listing 1，Listing 2）の通りである．*triget* により，初期状態のソースに基づいて複製が生成される．ここで，それぞれのビュー上で，次の操作を行う．

- Insert [1] (N "xxx" ["yyy"])
- Insert [0, 1] (N "xx" ["yy"])
- Insert [0, 0] (N "z" ["x"])

更新後のビュー $v'$ と *triput* により更新をマージしたソース $s'$ ($s_2$) は（Listing 3，Listing 4）の通りである．また，同じ状態に行き着く経路が複数存在することも確認した．更新後のソース $s_2$ ($s'$) に対し *triget* を行い生成したビュー $v_2$ の XML は Listing 5 の通りである．ここで，ビュー上で，次の操作を行う．Delete の引数について，2.5 節で述べたように，$IP_1$ を満たす操作変換の実装に合わせるため，ダミーデータとして"___"を与えている．本研究で想定している共同編集ソフトウェアでは，Undo 機能を実現することは目指していないため，Delete の第 2 引数の値は使われることはない．しかしながら，内部仕様上，何らかの Val 型のデータを入れる必要があるために，ダミーデータを与えている．

- EditLabel [0] "wwww"
- Delete [0, 1] "___"
- Insert [0, 0] (N "iii" ["jjj"])

更新後のビュー $v'_2$ と *triput* による更新をマージしたソース $s'_2$ は（Listing 6，Listing 7）の通りである．*triget* → *triput* → *triget* → *triput* の順番で実行した際，ビュー上で行った更新の意図がすべて反映されており，検討した操作変換が期待する結果が出力されていることがわかる．

## 9 関連研究

Sinchuk et al. はサーバ・クライアントモデルを想定しているため，$C_2$ については考慮していない．Randolph et al.[19] は，$C_1$ と $C_2$ を満たす変換関数について提案しているが，特殊なパラメータ追加が必要となる．本研究では，分散環境を想定しており，本来は $C_2$ を満たす変換関数を用いる必要があるが，パラメータ追加を行わず，$C_1$ のみを満たす変換関数を提案する．

本研究では，木構造について $C_1$ が満たされていることが Coq[16] を用いて数学的に証明済である Sinchuk et al. による研究 [20] を操作変換の実装として利用する．$C_2$ に関して，

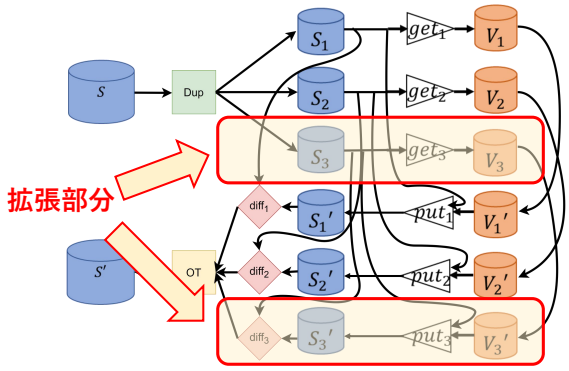



**図 9:** 3 つのビューの競合解決を行う双方向変換

Sinchuk et al. は，極めて特殊な状況を対象とすることから満たす必要はないとして，証明は省略している．本研究では，$C_1$ を満たす Sinchuk et al. の実装を基に，操作変換の適用順の制御を行うことで，3 者による同時更新に対応した操作変換を実現した．本研究の先行研究として，土生らによる研究 [12] が挙げられる．一般的な双方向変換では，ソースとビューは 1 対 1 の関係であるが，ソースとビューが 1 対 2 の場合に対応するため，複製されたデータをマージする能力を備えた原子的なコンビネータ Dup を持つ双方向変換言語 X を応用している．Dup は順方向変換ではデータの複製を，逆方向変換ではマージを行う．土生らはこの Dup を拡張することにより，順方向に対応した *mget*，逆方向に対応した *mput* という新しい関数を定義・実装している．図 9 の上半分は *mget*，下半分は *mput* のデータ伝播の流れを示している．土生らの *mget* はソース $S$ を複製してビューを 2 つ生成する．土生らの *mput* はソースと 2 つのビューを入力とし，3-way merge を行う．*mput* の特徴として，入力される複製のビューに更新が発生した場合には，必ずビュー間の競合解決を行う必要がある．操作変換を利用することにより，2 つのビューの更新内容が異なる場合においても，どちらの更新も考慮したデータ伝播が可能となる．実装の実行結果から，X 言語の特性 GETPUTGET，PUTGETPUT，操作変換の特性 $C_1$ を満たすことが確認されている．本研究では土生らの研究を拡張し，3 つのビューのに競合解決を行う双方向変換を実現し，土生らの研究では対応できない網構成への対応の第一歩となった．公平性の維持を司る操作変換と一貫性の維持を司る Orderer を分けて考え，一貫性維持を司る部分が Pluggable である（現状は Orderer に移譲）アーキテクチャを新たに採用した．また，競合解決機能において，複製機能を持つ双方向変換言語 X[13, 14] を拡張した土生の実装を複製結合子 Dup において，逆方向変換で 4-way merge（3 つの更新された複製と複製元で計 4 つ）を行うようにさらに拡張を行った．XML に対する 3 つのビューを介した共同編集を想定した実装を行い，複製を含む双方向変換の well-behavedness と，競合した更新が最終結果に保持されていることを実験により確認した．

## 10 まとめと今後の課題

本稿では，3 つ以上のビューの更新内容を考慮した双方向変換を実現するにあたり，複数の操作が同時に行われた際，操作変換の合成を分散台帳プラットフォーム Hyperledger Fabric の調停機能である Orderer で適切に制御することを提案した．多数の情報源の間での双方向変換として共同編集ソフトウェアによる利用を想定し，評価実験を行った結果，PUTGETPUT，GETPUTGET を満たしていること，検討した操作変換が期待する結果と本研究の提案手法により得られた結果が一致していることを確認した．

双方向変換言語 Inv を拡張することにより，3 者による同時更新に対応した双方向変換を実現することができた．しかしながら，追加した演算 *extendResC* が Inv の性質（順方向変換を定義すれば，˘ をつけることにより，自動的に逆方向変換も定義できること）について未確認である．

8 節で行なった評価実験は，編集者が行う操作は最大 1 つという仮定のもと行なったものである．編集者が複数の操作を行う場合，操作変換を行う際，同じ編集者が行った操作を合成するかどうかで編集結果が異なる可能性がある．本研究の手法で対応すること考えると，他の編集者が行った操作と変換を行う前に，同じ編集者が行ったすべての操作を合成し，1 つの操作として扱う必要があると考えられる．

本研究では，図 1 のパターン②:*put* が向き合う場合について対象としており，文書全体を共同編集者に共有し，共同編集を行うシナリオを応用として想定していた．よって，ソースからビューを生成する際，恒等変換を利用していたが，共有したいデータのみを抽出するような変換に変更することで，Local Privacy に対応できると考えられる．また，図 1 のパターン③:*get* が向き合う場合についても実装することで，Local Privacy を実現できると考えられる．

分散環境的な側面では，単体の CouchDB と Orderer において，データが集中する点について，クライアントごとにインスタンスを立てる対応を今後の課題とする．また，地理的に分散した 3 つの Organization 内の 3 者が同時にトランザクションを送信した際も同様に，同時更新に対応できることを評価する必要がある．

## 参考文献

[1] Hyperledger fabric docs: Adding an org to a channel. `https://hyperledger-fabric.readthedocs.io/ja/latest/channel_update_tutorial.html`. 取得: 2024/02/01.

[2] Hyperledger fabric offchain sample. `https://github.com/hyperledger/fabric-samples/tree/main/off_chain_data`.

[3] F. Abou-Saleh et al. *Introduction to Bidirectional Transformations*, volume 9715 of *LNCS*, pages 1–28.

Springer International Publishing, Cham, 2018.

[4] E. Androulaki et al. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proc of EuroSys'18*, page 15, 2018.

[5] A. Anjorin et al. Bidirectional transformations, (NII shonan meeting 2016-13). *NII Shonan Meet. Rep.*, 2016.

[6] Y. Asano et al. Bidirectional collaborative frameworks for decentralized data management. In *Software Foundations for Data Interoperability*, pages 13–51. Springer International Publishing, 2022.

[7] A. Cleve et al. Multidirectional Transformations and Synchronisations (Dagstuhl Seminar 18491). *Dagstuhl Reports*, 8(12):1–48, 2019.

[8] K. Czarnecki et al. Bidirectional transformations: A cross-discipline perspective. In *ICMT*, pages 260–283, 2009.

[9] Z. Diskin et al. Multiple model synchronization with multiary delta lenses with amendment and k-putput. *Form. Asp. Comput.*, 31(5):611–640, nov 2019.

[10] C. A. Ellis et al. Concurrency control in groupware systems. *SIGMOD Rec.*, 18(2):399–407, jun 1989.

[11] J. N. Foster et al. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM TOPLAS*, 29(3):17, 2007.

[12] M. Habu et al. Conflict resolution for data updates by multiple bidirectional transformations. In *Software Foundations for Data Interoperability*, pages 62–75. Springer International Publishing, 2022.

[13] Z. Hu et al. A programmable editor for developing structured documents based on bidirectional transformations. In *PEPM'04*, page 178–189. ACM, 2004.

[14] Z. Hu et al. A programmable editor for developing structured documents based on bidirectional transformations. *Higher-Order and Symbolic Computation*, 21(1):89–118, 2008.

[15] Z. Hu et al. Bidirectional transformation "bx" (dagstuhl seminar 11031). *Dagstuhl Reports*, 1(1):42–67, 2011.

[16] INRIA. The coq proof assistant. `https://coq.inria.fr/`.

[17] S.-C. Mu et al. An algebraic approach to bi-directional updating. In *Programming Languages and Systems*, pages 2–20. Springer Berlin Heidelberg, 2004.

[18] A. Randolph et al. On consistency of operational transformation approach. *Electronic Proceedings in Theoretical Computer Science*, 107:45–59, Feb 2013.

[19] A. Randolph et al. On synthesizing a consistent operational transformation approach. *IEEE Trans. Comput.*, 64(4):1074–1089, apr 2015.

[20] S. Sinchuk et al. Verified operational transformation for trees. In *ITP'16*, pages 358–373. Springer, 2016.