Vol.2015-SE-187 No.28
2015/3/13

# データが動的に連動するソフトウェアの設計指針導出手法

小林　謙太郎[1,a)]　新田 直也[1,b)]

**概要**：データが動的に連動するソフトウェアの開発では, データをどのように管理し, いつどのように更新すればよいか等の設計指針を決定する必要がある. しかしながら決定した設計指針が不適切であると, 期待される振る舞いを実現できなくなる可能性がある. そこで本研究では, システムに対するドメイン固有の制約と, システムに期待される振る舞いを共にラベル付き遷移システムでモデル化し, それらを満たす設計指針を導出する手法を提案する. 本稿では 2 つの事例に対して本手法を適用し, 適切な設計指針を導出できることを確認したので, その報告を行う.

## 1. はじめに

データが動的に連動するソフトウェアの設計では, あるデータの更新をどのように整合性を保ちながら別のデータに反映させるかが問題となる. このようなソフトウェアはグラフィックエディタソフトやゲームソフトなど, 様々な分野で見受けられる. このようなソフトウェアの設計は一般的に困難であり, 特にデータ間の依存関係が複雑で, 考慮すべきシステムの操作の組み合わせが膨大になる場合, 適切な設計を得ることは難しい. しかしながら決定した設計指針が不適切であると, 期待される振る舞いの一部を実現できなくなる可能性がある. 特に実現できない振る舞いが存在することが実装工程の終盤で発覚した場合, 大きな手戻りを生じることになり, プロジェクト全体に深刻な影響をおよぼす. そこでこのような設計上の問題を解決するために様々なソフトウェア設計に関する研究が行われている [1][2][3][4][5]. 善明ら [1] はアプリケーションフレームワーク [6] の設計がもたらす制約のもと, システムに期待される振る舞いを実現できるかどうかを判定する手法を考案した. しかし, 善明らの手法ではアプリケーションフレームワークという限定した対象領域のみに着目しており, 適用範囲が限られていた.

そこで本研究では, システムに対するドメイン固有の制約だけを前提とし, システムに期待される振る舞いを満たす設計指針を導出する手法を提案する. 具体的には, データ間の依存関係をシステムに対するドメイン固有の制約から抽出し, システムに期待される振る舞いを満たすように依存関係を改良しながら設計を進めていく. 本手法はシステムに期待される振る舞いを満たすまで改良を終えた時点でそれらを満たすシステムの設計指針を出力する. 本稿ではこの手法の一部を自動化しツールとして実装を行った. さらに, 2 つの事例に対して本ツールを適用し, 適切な設計指針を導出できることを確認した.

[1] 甲南大学大学院 自然科学研究科
Graduate School of Natural Science, Konan University
[a)] m1324005@center.konan-u.ac.jp
[b)] n-nitta@konan-u.ac.jp

## 2. データが動的に連動するシステム

**データが動的に連動するシステム**とは, システムの動作中に行われたあるデータの変更に伴って, そのデータに依存している別のデータが即座に連動することを要求されるシステムのことを言う. 例としてグラフィックエディタを挙げる. 図面上で 2 つのノードが 1 つのコネクタによって接続されている状態のときに, ノードの位置を動かすとコネクタもノードの位置に応じて即座に状態を変える (図 1 参照). 以下, この例をノードとコネクタの例と呼ぶ. データが動的に連動するシステムでは, データの更新をどのように整合性を保ちながら他のデータに反映させるかが問題となる. 特に以下のような場合, 適切な設計を見つけ出すことは難しい.

(1) システムが対象とするデータの種類が膨大で, それらの間に要求される依存関係が複雑である.

(2) システムに対して要求される仕様を, システムに対する操作とそれへの応答として考えたときに, それらの対応関係が複雑で, 考慮すべき操作の組み合わせが膨大になる場合.

システムが対象とする問題領域を**ドメイン**と呼ぶ. 一般に特定のドメインの中で開発されうるシステムは複数存在する. 上記 (1) はドメインそのものが各システムに対して要

求するドメイン固有の制約を含むと考えることができる. 一方, (2) は個々のシステムに求められる個別の要求仕様と考えることができる. データが動的に連動するシステムの設計ではこのようなドメイン固有の制約とシステムに求められる振る舞いの両立が必要である. これらの両立を図るため本研究では以下のアプローチをとる.

(1) 対象領域に存在するデータ間の依存関係を形式的に表現するモデルを考える.

(2) 典型的な操作手順と, それに対するシステムの応答を実行シナリオで表す.

これによって設計が形式的に評価できるようになり, さらにその評価を満たすような設計指針を系統的に導出する手法の構築も検討することができる.

### 2.1 リソース

ドメイン固有の制約と要求仕様の両方を構成する要素として, 本研究ではリソースに着目する. **リソース**とは対象領域に存在し, その状態が動的に変化しうるデータのことを言う. 例えばノードとコネクタの例では, ノードやコネクタ端点の位置がそれぞれリソースとして挙げられる.

### 2.2 システムに対するドメイン固有の制約

リソース間には個々のシステムに依存しないドメイン固有の制約が存在しうる. 具体的にはあるリソースがある状態を取るときに, 別のリソースがその状態に依存した特定の状態を取らなければならないという形で制約が存在する. 例えば, ノードの位置を変えるとそれに接続されているコネクタ端点の位置が変わらなければならない. 4.2 節でこのようなリソース間の依存関係のモデル化について説明する.

### 2.3 実行シナリオ

システムに対する要求仕様をシステムに対する操作と, それへの応答として考え, 複数の実行シナリオで表すものとする. **実行シナリオ**とはシステムの典型的な操作手順と, それに対するシステムの応答を表すものであり, システムの応答はリソースの状態変化によって表すものとする.

## 3. 諸定義

本節では本稿全体を通じて必要な定義を述べる. **ラベル付き遷移システム**（Labelled Transition System: LTS）は, 有限の状態間の遷移関係に基づいてシステムの挙動を表す計算モデルであり, 本研究ではリソースまたはリソース群の状態遷移の表現に用いる.

**定義 3.1 (ラベル付き遷移システム)** ラベル集合 $\mathcal{L}$ 上のラベル付き遷移システムは, $\Gamma = (\mathcal{Q}, \mathcal{T})$ の二項組で定義される. ここで, $\mathcal{Q}$ は状態の有限集合, $\mathcal{T} \subseteq \mathcal{Q} \times \mathcal{L} \times \mathcal{Q}$ は遷移関係を示す. 任意の状態 $q \in \mathrm{Q}$, ラベル $l \in \mathcal{L}$ について, $\langle q, l, q' \rangle \in \mathcal{T}$ のとき状態 $q$ において $l$ による遷移が発生したとき, 状態が $q'$ に変わることを示す. 以下ではこれを $q \xrightarrow{l} q'$ と書く. また LTS $\Gamma = (\mathcal{Q}, \mathcal{T})$ について, 関数 $\mathrm{Q}, \mathrm{T}$ を $\mathrm{Q}(\Gamma) = \mathcal{Q}, \mathrm{T}(\Gamma) = \mathcal{T}$ で定義する. LTS は初期状態を持つことができ, LTS $\Gamma$ の初期状態を $q_0(\Gamma)$ で表す. □

**定義 3.2 (LTS の並列合成)** LTS $P_1, P_2$ に対して, $P_1|P_2$ を $P_1, P_2$ の並列合成と呼ぶ. $P_1|P_2$ は以下の条件式を満たす LTS である.

- $Q(P_1|P_2) = Q(P_1) \times Q(P_2)$.
- $T(P_1|P_2) = \{\langle p_1, p_2 \rangle \xrightarrow{l} \langle p_1', p_2' \rangle \mid p_1 \xrightarrow{l} p_1' \text{かつ } p_2 \xrightarrow{l} p_2'\}$.

以下では簡単のため, $P_1|(P_2|(P_3|\cdots))$ $= (\cdots((P_1|P_2)|P_3)|\cdots) = P_1|P_2|P_3|\cdots$ と書く. また, $q = \langle q_1, q_2, \ldots, q_n \rangle \in Q(P_1|P_2|\cdots|P_n)$ について, $\mathrm{set}(q) = \{q_1\} \cup \{q_2\} \cup \cdots \cup \{q_n\}$ とおく. □

**定義 3.3 (双模倣等価 [7])** LTS $P_1, P_2$ が与えられたとき, 以下の条件を満たす関係 $R \subseteq \mathrm{Q}(P_1) \times \mathrm{Q}(P_2)$ が存在するとき $P_1$ と $P_2$ は双模倣等価であると言い, $P_1 \sim P_2$ で表す.

- すべての $\langle q_1, q_2 \rangle \in R, l \in \mathcal{L}$ に対し, $q_1 \xrightarrow{l} q_1'$ に対して, $q_2'$ が存在して, $q_2 \xrightarrow{l} q_2'$ かつ $\langle q_1', q_2' \rangle \in R$. □

## 4. 提案モデル

2 節で説明したようにデータが動的に連動するシステムの設計では, データの更新をどのように整合性を保ちながら反映させるかが問題となる. 本節ではそれらを LTS を用いてモデル化する方法について説明する.

### 4.1 リソースのモデル化

リソースはドメイン固有の制約と実行シナリオの両方を構成する要素であり, 対象領域に存在する状態が変化するデータのことを言う. リソース全体からなる集合を $\mathcal{R}$ とする.

### 4.2 システムに対するドメイン固有の制約のモデル化

全リソース間に成り立つ依存関係を $\mathcal{A} = R_{\mathcal{A}}^1|R_{\mathcal{A}}^2|\cdots|R_{\mathcal{A}}^{|\mathcal{R}|}$ で表す. リソースへの操作の集合をラベル集合 $L_{\mathcal{A}}$ で表す. ただし $R_{\mathcal{A}}^j (1 \leq j \leq |\mathcal{R}|)$ は各リソースの状態遷移を表す $L_{\mathcal{A}}$ 上の LTS であり, $\sigma(R_{\mathcal{A}}^j) = r \in \mathcal{R}$ は $R_{\mathcal{A}}^j$ に対応するリソースを表す. なお, 個々のリソースをモデル化する際, LTS が状態爆発を引き起こしたり, 状態数が無限になることを防ぐため, 必要に応じて取りうる状態の集合を適切な範囲で限定する. 例えばノードとコネクタ例では, ノードが取りうる座標を厳密に状態に反映すると, 状態爆発を引き起こす. そのためノードが取りうる座標を P0~P8 の 9 つの座標に限定した (図 2 参照).

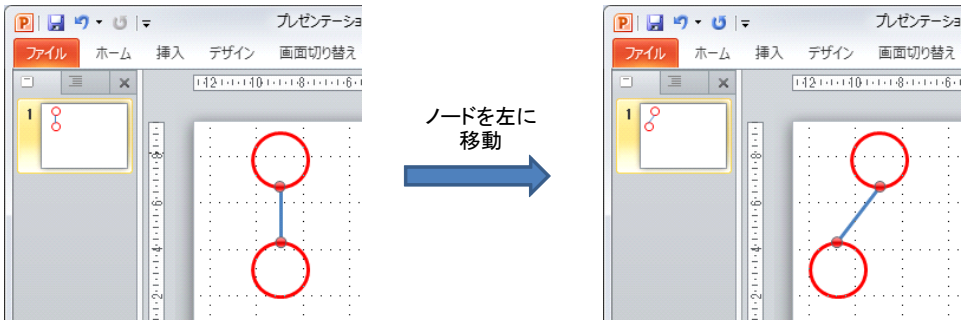


**図 1** ノードとコネクタの例

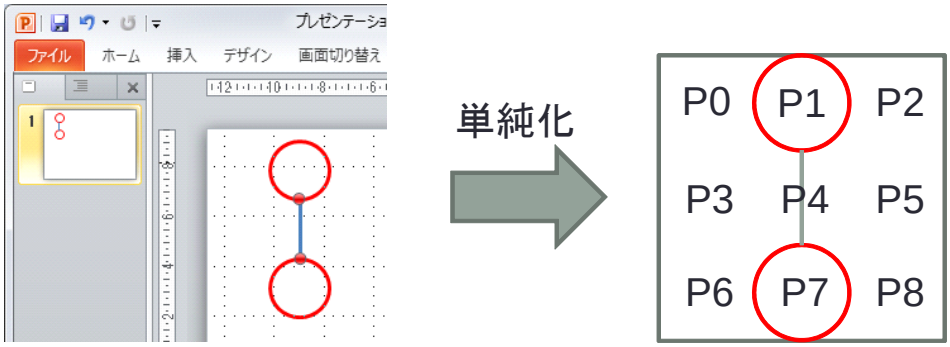


**図 2** ノードとコネクタ例の簡単化

### 4.3 実行シナリオのモデル化

実行シナリオ全体からなる集合を $\mathcal{S}$ とする．イベント全体からなる集合を $L_{\mathcal{S}}$ で表す．各実行シナリオ $S_i \in \mathcal{S}$ を $S_i = R^1_{S_i}|R^2_{S_i}|\cdots|R^{N(S_i)}_{S_i}$ で表す．ここで $N(S_i) \geq 1$ は $S_i$ によって定まる適当な整数とする．ただし，$R^j_{S_i}(1 \leq j \leq N(S_i))$ は実行シナリオ $S_i$ における各リソースの状態遷移を表す $L_{\mathcal{S}}$ 上の LTS である．$\sigma(R^j_{S_i}) = r \in \mathcal{R}$ は $R^j_{S_i}$ に対応するリソースを表す．ただし，実行シナリオはシステムが満たすべき特定の動作を表すため，$\forall_{q \in Q(S_i)}.(\langle q, l_1, q_1\rangle \in T(S_i)$ かつ $\langle q, l_2, q_2\rangle \in T(S_i)) \Rightarrow (l_1 = l_2$ かつ $q_1 = q_2)$ を満たすものとする．ここで $\langle q, l, q'\rangle \in T(S_i)$ のとき，$\mu_{S_i}(q) = l, \tau_{S_i}(q) = q'$ とおく．また，LTS $R$ の遷移系列全体からなる集合を $\mathrm{W}(R)$ とおく．イベントとリソース操作の対応 $\kappa : L_{\mathcal{S}} \to L_{\mathcal{A}}$ に対して，$\kappa(\mathrm{W}(R))$ は $\mathrm{W}(R)$ の各遷移系列中のイベント $e$ を，リソース操作 $\kappa(e)$ に置き換えて得られる集合とする．また，$\sigma(R^j_{\mathcal{A}}) = \sigma(R^k_{\mathcal{S}})$ のとき $\mathrm{Q}(R^j_{\mathcal{A}}) = \mathrm{Q}(R^k_{\mathcal{S}})$ を満たすものとする．

## 5. 設計指針導出問題

設計指針導出問題とは，システムに対して課されるドメイン固有の制約と，実行シナリオを同時に満たすような設計指針を導出する問題である (図 3 参照)．具体的には設計指針導出問題は全リソース間に成り立つ依存関係 $\mathcal{A}$ と実行シナリオ全体からなる集合 $\mathcal{S}$ に対して定義される．解となる設計指針は以下の 3 点で構成される．

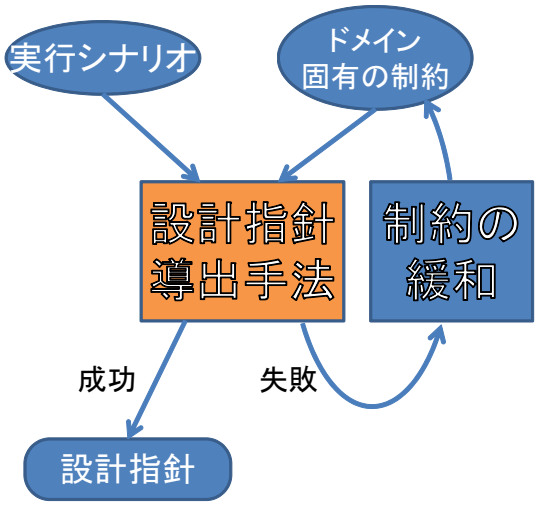


**図 3** 設計指針導出手法

(1) **リソース記憶の必要性**: 各リソースの状態を記憶する必要があるか否かを示す．リソースは更新処理の煩雑さや不整合を避けるため，必要がなければ，できる限り記憶させない方が望ましい．$R^j_{\mathcal{A}}$ が $R^k_{\mathcal{A}}$ に依存するとき $R^j_{\mathcal{A}}$ の状態が $R^k_{\mathcal{A}}$ の状態に対して一意に定まるなら $R^j_{\mathcal{A}}$ の状態を記憶させる必要がない．形式的に，リソース $R^j_{\mathcal{A}}$ の状態がリソース $R^k_{\mathcal{A}}$ の状態に対して一意に定まるとは，$R^j_{\mathcal{A}} \sim R^k_{\mathcal{A}}$ で定義される．一方，$R^j_{\mathcal{A}}$ に対するある操作に対して遷移先が一意に定まる場合も $R^j_{\mathcal{A}}$ の状態を記憶させる必要がない．形式的に，$R^j_{\mathcal{A}}$ に対する操作 $l(\in \mathcal{L}_{\mathcal{A}})$ に対して遷移先が一意に定まるとは，$\forall_{q \in Q(R^j_{\mathcal{A}})}.(q \xrightarrow{l} q_1$ かつ $q \xrightarrow{l} q_2) \Rightarrow q_1 = q_2$.

(2) **リソースへのアクセス API の隠蔽の許容性**: 各リソースに対する操作はリソースへのアクセス API に相当し, 各 API に対して外部への公開非公開を決定する必要がある. API を非公開にすると, 実装できない実行シナリオが生じる可能性がある. 一方, API を公開すると互いに依存するリソース間で不整合を生じる実装を導く可能性がある.

(3) **イベントとリソース操作の対応**: 実行シナリオに出現する各イベントをどのリソース操作に対応させるかを示す. どのリソース操作へも対応させることができないイベントが 1 つでも存在すると, その実行シナリオは実装できないと判断する. 具体的には, 次の条件を満たす $\kappa : L_{\mathcal{S}} \rightarrow L_{\mathcal{A}}$ をイベントとリソース操作の対応という. $\forall_{S_i \in \mathcal{S}}.\mathrm{W}(\mathcal{A}) \supseteq \kappa(\mathrm{W}(S_i))$.

本手法の入力は複数の実行シナリオと 1 つのモデル化したドメイン固有の制約である. これらに手法を適応し成功した場合, 設計指針を出力する. しかし失敗した場合はドメイン固有の制約を緩和して再度手法を適応する. ドメイン固有の制約の緩和とは, 実行シナリオが求める振る舞いが, 現状のドメイン固有の制約では許されない例外的な振る舞いを示すとき, リソースの分割や遷移の追加を組み合わせて, 例外的な動作も許容できるようにすることである. 例えばノードとコネクタ例で, 実行シナリオが図 4 のように 2 つのノードがコネクタと接続されている状態で, 片方のノードが削除されたとき, コネクタが現状のまま存在するという振る舞いを求めているとする. この場合ドメイン固有の制約を純粋なグラフ理論の枠組みでのみ捉えてリソースをグラフのノードとエッジ, リソース間の依存関係をグラフにおけるノードとエッジの間に成り立つ関係とみなすと (図 5 参照), このドメイン固有の制約では, 片方のノードを消すとコネクタも同時に消えることになり, 実行シナリオが要求する振る舞いを許容できない. そこでドメイン固有の制約を緩和し, コネクタの端点をノードとは別のリソースとして分割する (図 6 参照).

## 6. 設計指針導出アルゴリズム

本節では前節で定義した設計指針を導出するアルゴリズムを説明する. 出力される設計指針は 5 節で説明した以下の 3 つである.

(1) リソース記憶の必要性

(2) リソースへのアクセス API の隠蔽の許容性

(3) イベントとリソース操作の対応

これらの設計指針を導出するアルゴリズムについて説明する. アルゴリズムは以下の 3 つである.

```
isOneDestination:
 入力 : R_A
 出力 : 遷移先が同じかどうか

 foreach l ∈ L_A
  foreach q ∈ Q(R_A)
   Q_cur = {q' | q -1-> q'}
   q_tmp := null
   foreach q_cur ∈ Q_cur
    if q_tmp = null
     q_tmp := q_cur
    else if q_tmp ≠ q_cur
     return false
    endif
   repeat
  repeat
 repeat
 return true
```

**図 7** ラベルによる遷移先が一意に定まるか否かを判定するアルゴリズム

a) **リソースの全状態に対して与えられたラベルによる遷移先が一意に定まるかどうかの判定**

b) **リソースが依存している他のリソースによって模倣可能かどうかの判定**

c) **イベントとリソース操作の対応の導出**

アルゴリズム a) と b) はリソース記憶の必要性を判定するアルゴリズムである. 入力されるリソースはドメイン固有の制約 $\mathcal{A}$ に含まれる各リソース $R_{\mathcal{A}}$ である. この 2 つのアルゴリズムのどちらかが記憶不要と判定したとき, 入力されたリソースは記憶が不必要と判定する. またアルゴリズム c) はリソースへのアクセス API の隠蔽許容性と, イベントとリソース操作の対応を同時に導出するアルゴリズムである. これはリソースの状態間の不整合を未然に防ぐため, イベントと対応関係があるリソース操作 API 以外の API を隠蔽するからである. もしこのアルゴリズム c) によって適切なイベントとリソース操作の対応が得られなかった場合は, 設計指針の導出は失敗と判定する. リソースの全状態に対してラベルによる遷移先が一意に定まるかどうかを判定するアルゴリズムを図 7 に示す. リソースが依存しているリソースによって模倣可能かどうかを判定するアルゴリズムは紙面の都合上省略する. イベントとリソース操作の対応を導出するアルゴリズムを図 8 に示す.

## 7. 事例研究

提案アルゴリズムを Java で実装し, アルゴリズムの評価を行った. ツールは LTS でモデル化された実行シナリオとドメイン固有の制約を入力とし, 設計指針を出力する. LTS は XML 形式で入力され, 設計指針はコンソールに出力さ

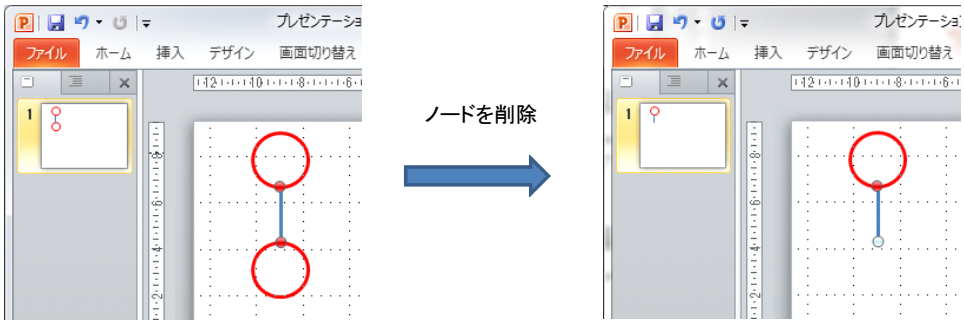


**図 4** ノードを削除する実行シナリオ

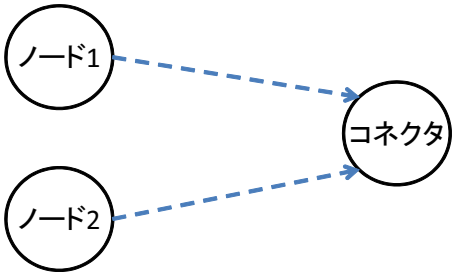


**図 5** ノードとコネクタのドメイン固有の制約におけるリソース間の依存関係

```
calcCorrespondency:
 入力：S_i, A
 出力：イベントと操作の対応関係
 waitingStates := ⋃_{j=1}^{N(S_i)} Q(R^j_{S_i})
 a_cur := q_0(A)
 s_cur := q_0(S_i)
 while waitingStates ≠ φ
  waitingStates := waitingStates \ set(s_cur)
  s'_cur = τ_{S_i}(s_cur)
  foreach a ∈ L_A
   a'_cur ∈ {q' ∈ Q(A) | a_cur -a-> q'}
   if set(s'_cur) ⊆ set(a'_cur)
   correspondency := correspondency ∩ {⟨μ_{S_i}(s_cur), a⟩}
    s_cur = s'_cur
    a_cur = a'_cur
    break
   endif
  repeat
 repeat
 return correspondency
```

**図 8** イベントとリソース操作の対応を導出するアルゴリズム

れる.

### 7.1 評価事例

#### 7.1.1 ノードとコネクタ

前述したノードとコネクタの例を提案アルゴリズムで評価する. 本事例で評価した実行シナリオは以下の 2 つである.

S1: 2 つのノードが 1 つのコネクタで接続されているとき, 片方のノードを移動するとコネクタも接続されたまま移動する (図 1 参照).

S2: 2 つのノードが 1 つのコネクタで接続されており, 片方のノードが削除されたとき, コネクタは現状のまま存在する (図 4 参照).

ただし, 単純化のためノードが取りうる位置は P0∼ P8 の 9 点に限定する (図 2 参照). これらの実行シナリオを LTS でモデル化したものが図 9, 図 10 になる. またこのシステムのドメイン固有の制約におけるリソース間の依存関係を示したものが図 5 になり, これを実行シナリオを実装するためにリソースを分割して緩和したものが, 図 6 である. この緩和されたドメイン固有の制約を LTS でモデル化し, 入力とした (図 11 参照).

#### 7.1.2 RadishFight

RadishFight[8] とは著者らの研究室で開発された 3D 格闘ゲームである. 本事例で評価した実行シナリオは以下の 3 つである.

S1: プレイヤーが水平な地面の上にいるとき, 右移動 (左移動) ボタンを押すと, プレイヤーは右 (左) を向いて地面の上を移動する.

S2: プレイヤーが左または右に移動した際に, 足元に地面がなくなると, プレイヤーは自由落下を開始する.

S3: プレイヤーが水平な地面の上または空中にいるとき, 上ボタンを押すと, プレイヤーはジャンプし始める.

ただし, 単純化のためプレイヤーの取りうる位置を初期位置の P0, 地面の上を移動した後の位置 P1, ジャンプした直後の位置 P2, 自由落下が始まった後の位置 P3 と限定する.

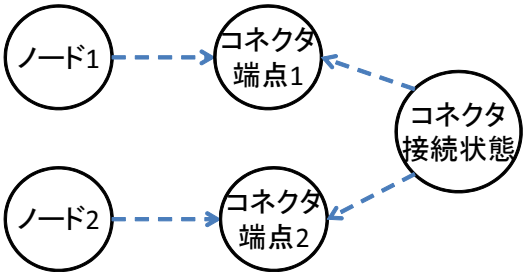


**図 6** 緩和されたノードとコネクタのドメイン固有の制約におけるリソース間の依存関係

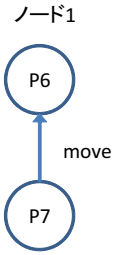


**図 9** ノードとコネクタ S1

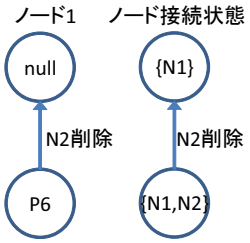


**図 10** ノードとコネクタ S2

**表 1** リソース記憶の必要性評価結果

| アルゴリズム / 事例 | isSameTransition | isBisimulation |
|---|---|---|
| ノードとコネクタ | なし | なし |
| RadishFight | 加速度 | なし |

これを LTS でモデル化した図が図 12, 図 13 になる．またこのシステムのドメイン固有の制約においてリソースは物体の物理量, リソース間の依存関係は物理量の間に成り立つ力係に相当する (図 14 参照)．これを LTS でモデル化しシステムの入力とした (図 15 参照)．ただし, ジャンプ動作を実現する際, 無限の加速度をとることになり, 力学的な制約から一時的に逸脱してしまう．このような場合, 加速度リソースにおいて非決定的に遷移し, 任意の状態となるように遷移を追加して, 制約を緩和する (図 15 中のラベル b による遷移).

### 7.2 評価結果

本評価事例の結果を本節で示す．提案アルゴリズムを適用して, 状態を記憶する必要がないと判断されたリソースを表 1, イベントとリソース操作の対応関係を表 2, 表 3, 隠蔽すべきでないと判断されたリソースアクセス API を表 4, 表 5 に示す.

**表 2** RadishFight のイベントとリソース操作の対応

| 実行シナリオ / 事例 | S1 | S2 | S3 |
|---|---|---|---|
| RadishFight | (move,m) | (fall,a1) | (jump,b) |

**表 3** ノードとコネクタ例のイベントとリソース操作の対応

| 実行シナリオ / 事例 | S1 | S2 |
|---|---|---|
| ノードとコネクタ | (move,N2 移動 6) | (delete,N2 削除) |

**表 4** ノードとコネクタのリソースへのアクセス API の隠蔽の許容性

| 実行シナリオ / リソース | S1 | S2 |
|---|---|---|
| ノード 1 | なし | なし |
| ノード 2 | N2 移動 6 | N2 削除 |
| コネクタ端点 1 | なし | なし |
| コネクタ端点 2 | なし | なし |
| コネクタ接続状態 | なし | なし |

**表 5** RadishFight のリソースへのアクセス API の隠蔽の許容性

| 実行シナリオ / リソース | S1 | S2 | S3 |
|---|---|---|---|
| 加速度 | なし | なし | なし |
| 速度 | m | a1 b | b |
| 位置 | なし | なし | なし |

## 8. 考察

前節で述べた評価結果から, ノードとコネクタと RadishFight の両方の事例において, 実行シナリオが求める振る舞いを再現できる設計指針を出力することができ, 本手法の有効性が確認された．また両例とも緩和しなかったドメイン固有の制約では実行シナリオを実装できなかったが, 緩和を行うことによって実行シナリオを実装できたため, 緩和するプロセスの有効性が確認できた．先行研究ではフレームワークの制約を対象として評価を行っていたが, 本

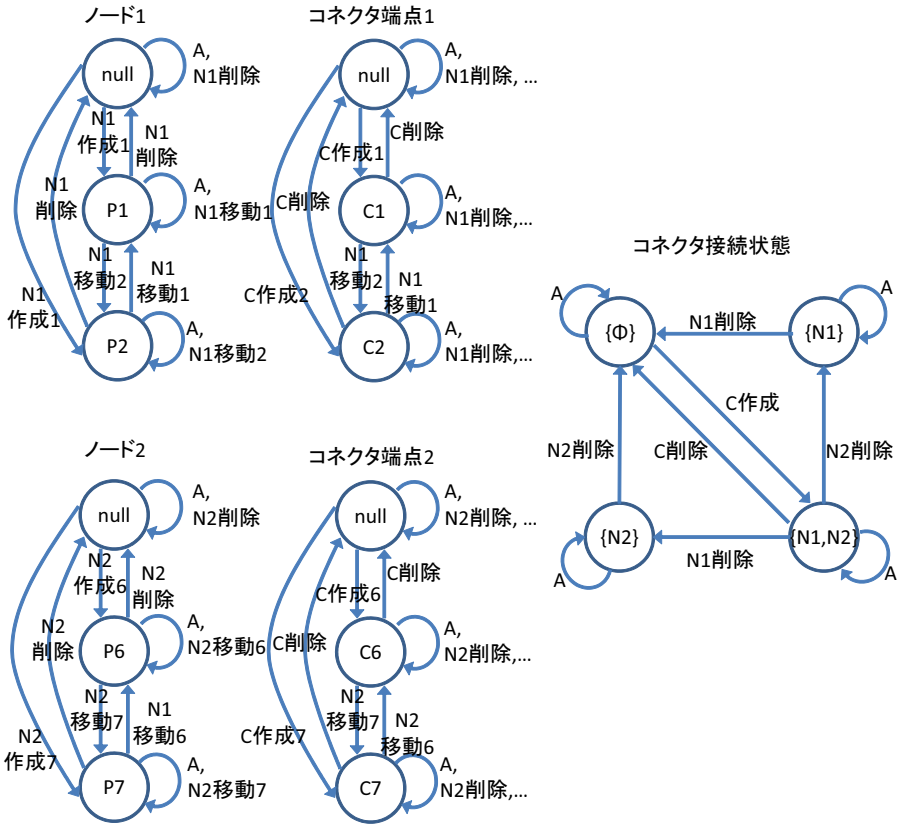


**図 11** ノードとコネクタのドメイン固有の制約

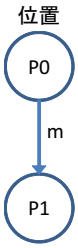


**図 12** RadishFight(S1)

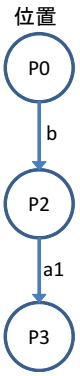


**図 13** RadishFight(S2,S3)

手法ではより一般的なドメイン固有の制約を対象として評価を行えたため, この点では先行研究より一般化されたと考えられる. ただし, 本研究で対象とするシステムはデータが動的に連動するものに限定されている. 今後どの程度まで対象範囲を広げて行けるかについては今後の課題である. 他の今後の課題として, よりツールの利便性を高めるため, 入出力を GUI で行えるツールの実装が挙げられる. また, 本手法が出力した設計指針から設計を導出するため, 設計指針と既存の設計手法との関係をより明らかにする必要がある.

## 9. おわりに

データが動的に連動するソフトウェアの設計指針を形式的に導出する手法の構築を行った. グラフィックエディタ上でのノードとコネクタの例や著者らの研究室で開発された 3D 格闘ゲームのドメイン固有の制約と実行シナリオから設計指針を導出したところ, 実行シナリオが求める振る舞いを再現できる設計指針を出力することができた.

今後, 本手法を他の事例に適用して実行シナリオが求める振る舞いを再現できる設計指針が出力することが可能であるか調査したい. また本ツールの GUI 化を行うことを検討したい.

## 参考文献

[1] Teruyoshi ZENMYO, Takashi KOBAYASHI, Motoshi SAEKI, "Supporting Application Framework Se-


**図 14** RadishFight のドメイン固有制約

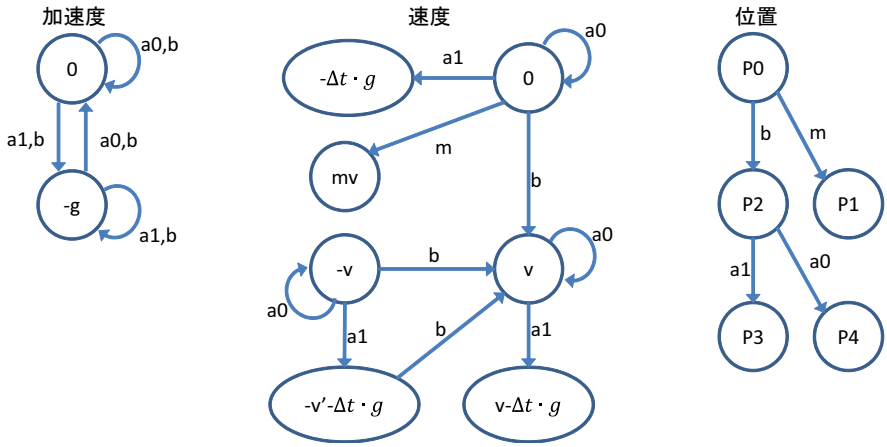


**図 15** LTS でモデル化した RadishFight

lection Based on Labeled Transition Systems," *IEICE TRANSACTIONS on Information and Systems* Vol.E89-D No.4 pp.1378-1389, 2006.

[2] N. Nitta, I. Kume and Y. Takemura, "A method for early detection of mismatches between framework architecture and execution scenarios," in *Asia-Pacific Software Engineering Conference (APSEC' 2013)*, 6 pages, in USB, 2013.

[3] Liliana Dobrica and Eila Niemelä. A survey on software architecture analysis methods, *IEEE Trans. on Softw. Eng.*, Vol. 28, No. 7, pp. 638–653, 2002.

[4] Chung-Horng Lung, Sonia Bot, Kalai Kalaichelvan and Rick Kazman. An approach to software architecture analysis for evolution and reusability, In *Proceedings of the Centre for Advanced Studies Conf. (CASCON)*, pp. 144–154, 1997.

[5] Eike Falk Anderson, Steffen Engel, Leigh McLoughlin and Peter Comninos. The case for research in game engine architecture, In *Proceedings of the ACM International Academic Games Conference on the Future of Game Design and Technology (FuturePlay)*, pp. 228–231, 2008.

[6] Mohamed E. Fayad, Ralph E. Johnson and Douglas C. Schmidt, *Building Application Frameworks: Object-Oriented Foundations of Framework Design.* John Wiley & Sons Inc, 1999.

[7] R. Milner, Communication and Concurrency, Prentice Hall, 1989.

[8] 新田 直也, 久野 剛司, 久米 出, 武村 泰宏, "3D ゲームエンジン Radish の開発とそのアーキテクチャ比較への応用," 日本デジタルゲーム学会, デジタルゲーム学研究, vol. 4, no. 1, pp. 1–12, 2010.