2004年度 修士論文

制約に基づくアニメーションツール Grifonにおける制約の階層化

提出日: 2005年2月2日
指導: 上田和紀 教授

早稲田大学大学院
理工学研究科 情報ネットワーク専攻
学籍番号: 3603U102-3
中村 好一

目 次

図 目 次

表 目 次

概 要

Grifon とは、論理的な概念や関係を表すようなアニメーションを柔軟に表現することを目的とした制約に基づくアニメーション作成システムである。Grifon は、従来のアニメーションツールとは違い、ハイブリッド並行制約 (Hybrid Concurrent Constraint Programming) と幾何制約によってアニメーションを表現する。

ハイブリッド並行制約は時間的な処理を、時間軸上の連続的・離散的変化を表す。また、幾何制約は図形同士の相互関係やグラフィックオブジェクトの空間的な関係を幾何的関係として表現するものである。つまり、ユーザは自分の描いた図に制約をかけることで図の幾何学的関係を保持したまま編集を続けられることができるのである。

既存のシステムでは、幾何制約とハイブリッド並行制約において幾何制約が優先されていた。そのため、ユーザの意向と異なるアニメーションが作成されてしまうなど、アニメーション作成システムとして表現の幅が狭かった。

本研究の目的は、Grifon における幾何制約とハイブリッド並行制約が矛盾し合う制約過多な状況でも適切な最適解を得られるようにするため、強さと呼ばれる制約の優先度を可能にする制約階層を実現することである。本研究では、ハイブリッド並行制約データに優先度を付随し、制約処理系の改良を行うことでハイブリッド並行制約と幾何制約の優先度を実現した。制約の階層化によって、ユーザーが制約の矛盾性を気にすることなく制約を扱えるようになった。

また、Grifon において制約の優先度を付随することで、幾何制約優先、ハイブリッド並行制約優先のアニメーションを表現し、アニメーションの拡張性を実現した。

第1章　関連研究

この章では、制約に基づくグラフィックスアプリケーションを紹介する。

1.0.1　Juno-2

インターフェイスは、マウス操作とテキストによる描画言語の表現が用意されている。どちらの操作を行っても、すぐにもう片方の表現反映される。どちらかというと言語表現の方に重点があり、専用の描画言語によって図形を表現する。描画言語は、Algol に近い制御構造を持ち、Lisp に似たデータ構造を扱う。UI を対象とした制約解消アルゴリズムにおいて、非線形制約の系を解くために $Newton$ 法を用いている。

1.0.2　Pegasus

ペンで描画するときに、コンピュータが綺麗な図を描くための支援をするシステム。ある線分の隣に、傾きがほぼ等しい線分を描画すると、それを発見して平行な線分に修正してくれる。また次にユーザが描く図形を予測し提示する機能も備わっているが、これは複雑な図形を描画すると、たちまち予測空間が莫大になるという問題もある。また、ラフスケッチのような作図作業を対象としているため、一度描画した図形に対する編集機能はほとんど無い。

1.0.3　Udraw

1996 年に吉田法茂氏と内藤頼光氏により作成され、それを基にして 1998 年に柳吾郎氏と坂東雄人氏によって、改良が加えられた幾何学的制約処理に基づく描画ツールである。一度図形同士に制約をかけることによって、その後の処理は、ユーザは図形の幾何学的な関係を気にすることなく、自動的に図形をその制約にしたがって、変形させることが可能である。

1.0.4 Constrator

Constrator とは、２００２年に石井大輔氏が Udraw を基に、一から作成した幾何学的制約処理に基づく描画ツールである。つまり、ユーザが一度、図形に制約をかけることによって、その後、図形を移動したり、ドラッグしたりしてもその制約によって、図形同士の相互関係は保つことができるという描画ツールである。

1.0.5 幾何制約一覧

制約の名前	制約の性質
Pin	点を固定する
FixX	X 座標の相対座標を固定する
FixY	Y 座標の相対座標を固定する
FixR	
UnifyX	X 座標を統一する
UnifyY	Y 座標を統一する
UnifyXY	XY 座標を固定する
MidPoint height Pararell	中点に固定する 平行にする
Orthogonal	垂直にする
LengthX	X 座標の長さを固定する
LengthY	Y 座標の長さを固定する

表 1.1: 幾何制約一覧

1.0.6 Constrator の実装環境

以下に Constrator の実装環境について示す。

パート	実装環境	使用言語
GUI	JDK 1.3.1	JAVA
データ管理	JDK 1.3.1	JAVA
JAVA-Prolog インタフェース	JDK 1.3.1 + Jasper	JAVA
制約処理エンジン	SICStus Prolog 3.9.1	Prolog
プラットホーム	Solaris 2.6 / Linux	

表 1.2: Constrator の実装環境

第2章　Hybrid並行制約プログラミング

2.1　ハイブリッド並行制約プログラミング(concurrent constraint programming)

Grifon において、ハイブリッド並行制約プログラミングに基づきアニメーション中の時間的変化を表現している。本章では、ハイブリッド並行制約プログラミングのベースとなっている並行制約プログラミング、デフォルト並行プログラミングについて述べ、ハイブリッド並行制約について説明する。

2.2　並行制約プログラミング

並行制約プログラミング (concurrent constraint programming) は、宣言的で並行なモデルを持ったプログラミングの枠組みである。並行制約プログラミング言語は論理的関係についての記述からなる。
並行制約プログラミングは、制約ストア (Constraint Store) とエージェント (Agent) という概念を用いた計算モデルを持つ。制約ストアに制約の集合が格納されており、制約ストア中の制約をエージェントが操作することで計算が行われるような枠組みである。制約ストアは、変数の値を制限する情報群の集合体で、制約の集合を表し計算中にエージェントによって加えられる。エージェントは制約ストアとやりとりを行う計算主体である。エージェントは、tell エージェントと ask エージェントの２種類に分類できる。tell エージェントは制約ストアに情報を加える。ask エージェントは、制約ストアにおいて制約が充たされているかどうかを調べる。
並行制約プログラミング言語によるエージェントの構文を以下に示す。

A::=a | if a then A | new X in A | A,A
a は制約を表している。

上記の構文について説明する。

- if a then A
  制約 a が充たされたならば、A を実行する。

- new X in A
  A の中に変数 X を生成する（X は A のローカル変数である）。

並行制約プログラミング言語によるプログラム例を示す。

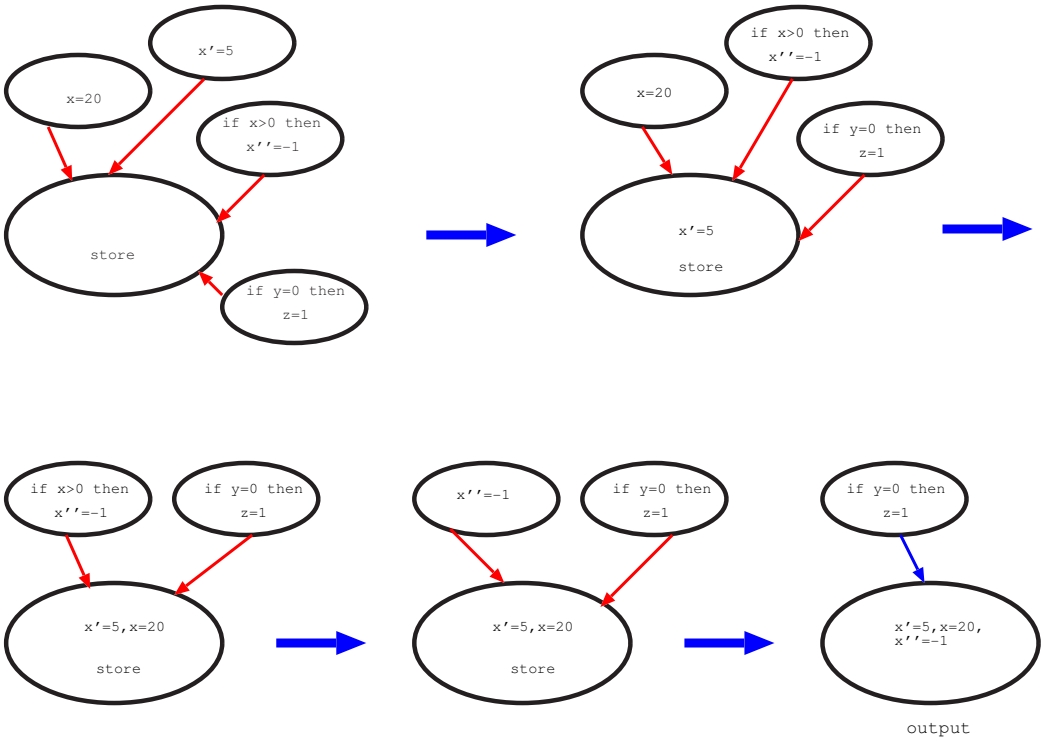

図 2.1: 並行制約プログラムの処理

並行制約プログラミング言語は、制約情報が欠けている状態 (否定情報) の認識ができないという欠点がある。

2.3 デフォルト並行制約プログラミング (default concurrent constraint programming)

上記の通り、並行制約プログラミング言語は否定情報の認識ができないという欠点があった。デフォルト並行制約プログラミングは、並行制約プログラミング

で扱えなかった否定情報を扱うための機能を追加した枠組みである。以下にエージェントの構文を示す。

A::=a | if a then A | new X in A | A,A | if a else A
a は制約を表している。

if a then A、new X in A は、上記と同様である。

- if a else A
  制約 a が充たされない場合、A を実行する。
  ∗if a else A と if ¬ a then A は異なる。前者は、制約ストアの最終状態が制約 a を含まない場合、A に遷移する。後者は、制約ストアの最終状態が ¬ a を含む場合、A に遷移する。

否定情報は if a else A という構文からなる $negativeask$ エージェントが行う。$negativeask$ エージェントが行う操作は、計算時に有効であるかどうか決めることができない。$Defaultconcurrentconstraintprogramming$ では、$negativeask$ の結果を推測し、実行する。推測が正しければ、実際に出力が得られる。推測が異なる場合は、実行を止め、バックトラックなどを行う。

2.4 ハイブリッド並行制約プログラミング (hybrid concurrent constraint programming)

2.4.1 ハイブリッド並行制約言語における計算手法

ハイブリッド並行制約プログラミングは、ハイブリッドシステムのモデリングと検証のための言語として考案された。ハイブリッドシステムは、離散的・連続的な動作からなるシステムである。物体の離散的・連続的な動作や変化を数値計算 (微分方程式) で求めるために下記の 3 つの計算手法を取り入れている。

- Euler 法
  微分方程式を数値計算で解く方法として最も単純であるが、最も精度の悪い方法である。オイラー法の式を以下に示す。

  $$x_{i+1} = x_i + f(t_i, x_i)\ \Delta\ t$$

  この方法は打切り誤差が大きいが、シンプルで荒い近似を得るのに使われることもある。精度を上げようとして、$\Delta\ t$ を小さくしすぎると丸め誤差の影響を受けて誤差がかえって増大することに注意が必要である。時刻 t_i での微分係数 $x'(t_i = f(t_i, x_i))$ が微小時間 $t_i < t < t_{i+1}$ の間は一定であるとした折れ線近似になっている。

- Runge-Kutta 法
  4 次ルンゲクッタ法はオイラー法より精度よく解を得ることができる。すなわち、時間刻み幅 $\Delta\ t$ をオイラー法より大きく取ることができ、精度の良い結果が得られる。以下に 4 次ルンゲクッタ法を示す。
  連立一階常微分方程式

$$x(t) = f(x(t))$$

  に対する解 $x(t)$ を

$$x(t + \Delta\ t) = x(t) + \frac{(d_1 + 2d_2 + 2d_3 + d_4)}{6}$$

  ただし、

$$\begin{aligned} \mathrm{d}_1 &= f(x(t))\ \Delta\ t \\ \mathrm{d}_2 &= f(x(t) + \tfrac{d_1}{2})\ \Delta\ t \\ \mathrm{d}_3 &= f(x(t) + \tfrac{d_2}{2})\ \Delta\ t \\ \mathrm{d}_4 &= f(x(t) + (d_3)\ \Delta\ t \end{aligned}$$

  により求める方法である。以下はルンゲクッタ法の概念図である。
  以下の図において、$\Delta\ t = t'$ とする。

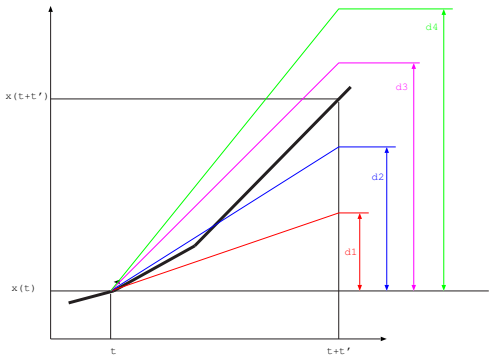

図 2.2: ルンゲクッタ法の概念図

- Bulirsch-Stoer 法
  実数係数の n 次方程式、

$$f(x) = a_0x^n + a_1x^{n-1} + ... + a_{n-1}x + a_n = 0$$

のすべての根を求める方法としてベアストウ法がある。ベアストウ法は、対象とする式を 2 次式、

$$x^2 + px + q$$

と $(n-2)$ 次式の積の形に変換することを繰り返して解く方法である。p と q の値を決めるために、その初期値 p_0 と q_0 を与え、$f(x)$ を $x^2 + p_0x + q_0$ で割った余りが 0 になるように p_0 と q_0 を徐々に修正していく。具体的なアルゴリズムは以下に示す。

1. p_0 と q_0 を与え、$k = 1$ とおく。
2. $b_i = a_i - p_{k-1}b_{i-1} - q_{k-1}b_{i-2}$、ただし、$i = 0, 1, ..., n$
3. $c_i = b_i - p_{k-1}c_{i-1} - q_{k-1}c_{i-2}$、ただし、$i = 0, 1, ..., n$
4. $D = c_{n-2}^2 - c_{n-3}(c_{n-1} - b_{n-1})$
   $\Delta\ p = (b_{n-1}c_{n-2} - b_nc_{n-3})/D$
   $\Delta\ q = (b_nc_{n-2} - b_{n-1}(c_{n-1} - b_{n-1}))/D$
5. もし、$\Delta\ p <\ \varepsilon$、かつ $\Delta\ q <\ \varepsilon$ であれば、
   A. 2 次方程式 $x^2 + p_{k-1} + q_{k-1} = 0$ を解く。
   B. $b_0, ..., b_{n-2}$ を新たに、$a_0, ..., a_{n-2}$ とおき、$n = n-2$ とし、1 へ戻る。
   ただし、$n = 1$ または $n - 2$ の場合は、その方程式を解く。
   そうでなければ、
   A. $p_k = p_{k-1} + \Delta\ p$
   $q_k = q_{k-1} + \Delta\ q$
   B. $k = k + 1$ とし、2 へ戻る。

2.4.2 ハイブリッド並行制約言語の構文

ハイブリッド並行制約プログラミングは時間軸に沿った表現ができるようにデフォルト並行制約プログラミング言語を拡張したものである。プログラムはある時点での処理についての記述とある時点から次の時点までの間の処理についての記述から構成される。

ハイブリッド並行制約プログラミング言語の構文を以下に示す。ハイブリッド並行制約プログラミングでは、hence 構文、always 構文と微分方程式の記述が重要である。

$*$ $hence\ A$ と $always\ A$ について

$$always\ A = A,\ hence\ A$$

Agents	Propositions
a	a holds now
if a then A	if a holds now,then A holds now
if a else A	if a will not hold now,then A holds now
new X in A	there is an instance $A[T/X]$ that holds now
A, B	both A and B hold now
hence A	A holds at every instant after now
always A	A holds at every instant from now

表 2.1: ハイブリッド並行制約言語の構文

hence A と *always* A は同じような役割を果たすが、厳密な定義は異なっている。*hence* A は現時点を含まない。*always* A は現時点を含む。

hence 構文はある時点以降に連続して行われる処理を記述するために用いる。ハイブリッド並行制約言語は、x' のような微分係数や微分方程式を記述できる。*hence* $x' = 1$ は、数学的に $dx/dt = 1$ を表し、x の値を現在の時間に保つ。

以下に、ハイブリッド並行制約言語のプログラム例 (ボール落下の例) を示す。

```
y=10,y'=0,
hence{
   cont(y),
   if y > 0 ||(y=0 && prev(y') > 0) then
            y'' = -1.0
   else if prev(y') < 0 then
           y' = -0.5 * prev(y')
   else always y'=0
},
sample(y)
```

ハイブリッド並行制約言語のプログラム例

上記のプログラムを 30s 実行したデータ (ボールの軌跡) を以下に示す。

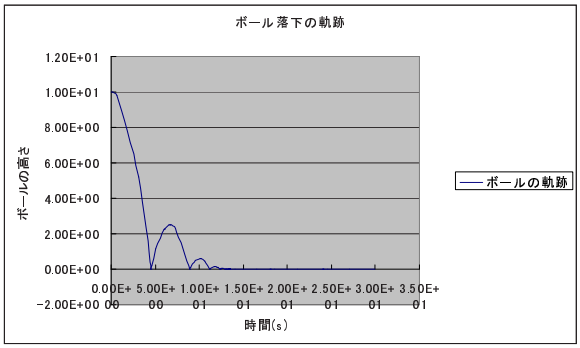

図 2.3: ハイブリッド並行制約言語におけるボール落下の例

第3章　制約に基づくアニメーション作成システムGrifon

3.1　制約に基づくアニメーションツールGrifon

Grifonは、ハイブリッド並行制約と幾何制約によってアニメーションを表現できるアニメーション作成システムである。ハイブリッド並行制約により、アニメーション中の動作について柔軟かつ簡潔に表現する。ハイブリッド並行制約は時間的な処理を、時間軸上の連続的・離散的な処理の記述により構成する強固な枠組みである。また、幾何制約は座標やサイズなどの実数属性値変数とする制約である。
図4.2は、Grifonのスクリーンショットである。

パート	実装環境	使用言語
GUI	JDK 1.4.1	JAVA
データ管理	JDK 1.4.1	JAVA
JAVA-C インタフェース	JDK 1.4.1 + JNI	JAVA
ハイブリッド並行制約処理エンジン		C
幾何制約処理エンジン height プラットホーム	JDK1.4.1 Windows	JAVA

表 3.1: Grifonの実装環境

3.2　hybrid concurrent constraint programming(ハイブリッド並行制約言語)

Grifonでは、ハイブリッド並行制約によってアニメーション中の時間的変化を表現する。ハイブリッド並行制約は特に離散的・連続的動作についての表現に優れている。例えば、現システムにおいてキーフレームなどを用いてユーザが直接指定していた離散的動作を、Grifonではハイブリッド並行制約によって制約充足系に計算させることができる。

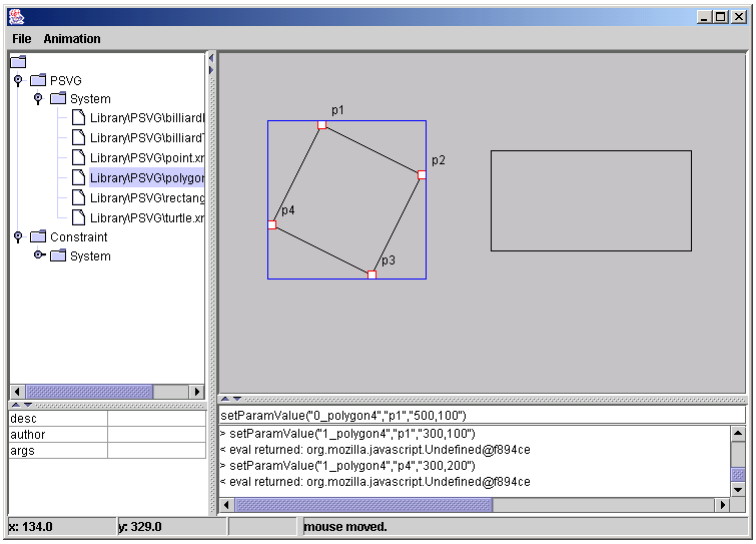

図 3.1: Grifon のスクリーンショット

3.3 Cassowary(幾何制約処理系)

Cassowary は、線形方程式や線形不等式を解くための制約ソルバーである。制約充足処理は ClSimplexSolver に実装されている。Cassowary のユーザは、制約を表すオブジェクトを生成し、ClSimplexSolver に登録することで制約をできる。制約の設定時に与える変数は ClLinearExpression を作成し、ClLinearExpression を基に制約を表す ClLinearEquation や ClLinearIn equality を作成する。方程式や不等式を解く方法として、*Simplex* 法を適用している。

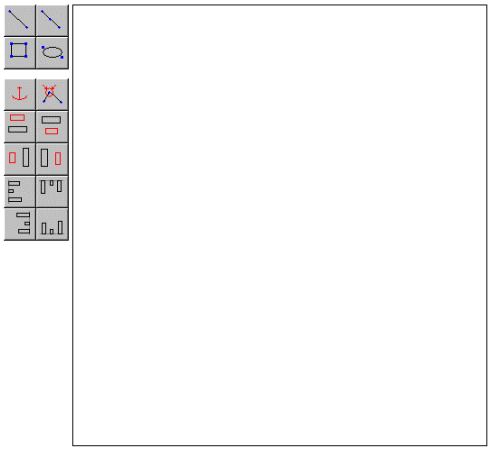

図 3.2: cassowary のスクリーンショット

3.4 *Simplex* 法

3.4.1 基底解と最適解

例として次の線形計画問題を考える。
問題 1

$$\text{目的関数:} -x_1 - x_2 \rightarrow \quad \text{最小}$$
$$\text{制約条件:} 3x_1 + 2x_2 \leqq 12$$
$$x_1 + 2x_2 \leqq 8$$
$$x_1 \geqq 0, x_2 \geqq 0,$$

図はこの問題の実行可能領域と目的関数の等高線を示している。この図からわかるように、2 変数の線形計画問題においては一般に実行可能領域は平面状の凸多角形となり、目的関数の等高線は平行な直線となるので、最適解は実行可能領域である凸多角形の境界上に存在する。さらに、その凸多角形の頂点のうち少なくとも一つが必ず最適解になっている。さらに、これと同様の性質は一般の n 変数の問題に対しても成り立つ。すなわち、線形計画問題の実行可能領域は一般的に凸多面体であり、最適解はその凸多面体の頂点の中に必ず図存在する。頂点の個数は有限であるから、原理的には、全ての頂点をしらみつぶしに調べていけば、必ず最適解を見つけることができる。しかし、頂点の個数は問題の変数や制約条件式が増えるにつれて急激に大きくなるので、全頂点を調べるのではなく、ごく一部の頂点だけを探索して最適解を見つけ出すような効率的なアルゴリズムが必要である。その代表的な手法が 1947 年に G.B.Dantzig によって提案された *Simplex* 法である。

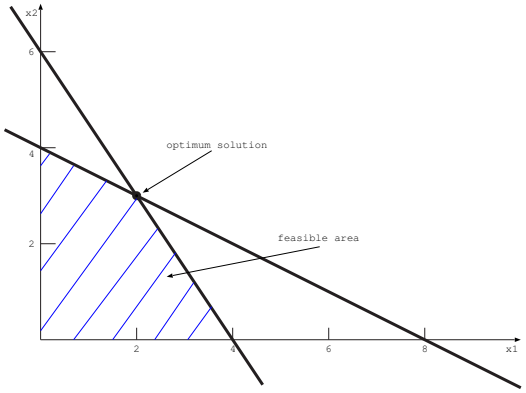

図 3.3: 線形計画問題の例

Simplex 法の基本的な考え方は、実行可能領域の一つの頂点から始めて、目的関数の値が必ず減少するように次々と隣接する頂点に移動していき、最終的に最

適解に到達しようというものである。まず、先にあげた例を基にスラックス変数を用いて問題を標準形に変換する。
問題 2

$$\text{目的関数: } -x_1 - x_2 \rightarrow \text{最小}$$
$$\text{制約条件: } 3x_1 + 2x_2 + x_3 = 12$$
$$x_1 + 2x_2 + x_4 = 8$$
$$x_1 \geqq 0, x_2 \geqq 0, x_3 \geqq 0, x_4 \geqq 0$$

この問題の二つの等式制約条件を満たす $x = (x_1, x_2, x_3, x_4)^T$ は無数に存在するが、二つの変数を適当に選んでそれらを 0 とおけば、残りの二つの変数の値は一意的に定まる。そのようにして定まる x は等式制約条件を満たす特殊解であり、基底解と呼ばれる。さらに、基底解のうち $x \geqq 0$ を満たすものは問題の実行可能解であるから、実行可能基底解という。また、基底解定める際に 0 とおいた変数を非基底変数と呼び、それらを要素とするベクトルを x_N で表す。また、それ以外の変数を基底変数と呼び、それらを要素とするベクトルを x_B で表す。

最適解は実行可能基底解の中に存在するが、ある実行可能基底解が実際に最適解かどうかを調べる方法を述べる。基底変数 x_B は非基底変数 x_N を用いて、

$$xx_B = B^{-1}b - B^{-1}Nx_N \ (1)$$

と表されるので、これを目的関数に代入すると、

$$c^T x = c_B^T x_B + c_N^T x_N$$
$$= c_B^T B^{-1} b + (c_N^T - c_B^T B^{-1} N)\, x_N \ (2)$$

となる。ただし、c_B と c_N はそれぞれ x_B と x_N に対応するベクトル c の要素からなるベクトルである。ここで、ベクトル

$$\pi = (B^T)^{-1} c_B \ (3)$$

を定義する。π はシンプレックス乗数と呼ばれる。いま、

$$c_N^T - \pi^T N = c_N^T - c_B^T B^{-1} N \geqq 0 \ (4)$$

が成り立つとする。そのとき、(2) 式より、全ての実行可能解 $x_N \geqq 0$ のなかで目的関数は $x_N = 0$ のとき最小値 $c^T x = c_B^T B^{-1} b (= \pi^T b)$ をとるので、今考えている実行可能基底解 $(x_B, x_N) = (B^{-1}b, 0)$ は最適解であることがわかる。すなわち、(4) 式を満たす実行可能基底解を最適基底解と呼び、そのときの基底行列を最適基底行列あるいは最適基底と呼ぶ。

3.4.2 *Simplex* 法

実行可能基底解において条件 (4) が成り立たないとき、基底変数の入れ替え、すなわちピボット操作を行うことにより、新しい実行可能基底解に移る。その際、目的関数値が減少するように移動するのがシンプレックス法の基本的な考え方である。
まず、条件 (4) が成立していないので、(2) 式における非基底変数 x_N の係数ベクトル $c_N^T - c_B\ TB^{-1}N$ すなわち相対コスト係数 $c_N^T - \pi^T N$ のなかに負の要素が少なくとも一つ存在する。そこで、そのような負の係数をもつ非基底変数 x_k を一つ選んで、その値を現在の値 0 から増加させれば、(2) 式より、目的関数の値を減少できる。また、その際、他の非基底変数の値は全て 0 に固定しておき、基底変数 x_B を (1) 式に従って、

$$x_B = B^{-1}b - B^{-1}a_k x_k \ (5)$$

と変化させれば、制約条件 $Ax = b$ は満たされる。ただし、a_k は非基底変数 x_k に対応する行列 A の列を表す。さらに、

$$b' = B^{-1}b, y = B^{-1}a_k \ (6)$$

とおけば、非基底変数 x_k を最大

$$\Delta\ = min\ [b'_i/y_i \mid y_i > 0 (i = 1, ..., m)] \ (7)$$

まで増加させても、各変数に対する非負条件 $x \geqq 0$ は保たれる。また、(5) 式より、x_k の値を Δ 間で増やしたとき、$\Delta = b'_i/y_i$ を満たす i に対応する基底変数の値は 0 になっているので、そのような基底変数と非基底変数 x_k を入れ替えるピボット操作を行う。なお、$y_i > 0$ であるような i が存在しないときは、非基底変数 x_k をどんどん大きくしていくと、制約条件を破ることなく、目的関数値をいくらでも小さくできるので、この問題は有限な最小値をもたないことがわかる。そのような問題は有界でないという。シンプレックス法の計算手順を以下にまとめる。

シンプレックス法
(0) 初期実行可能基底解 $(x_B, x_N) = (B^{-1}b, 0)$ を選ぶ。$b' = B^{-1}b$ とおく。
(1) シンプレックス乗数 $\pi\ = (B^T)^{-1}c_B$ を計算する。
(2) 非基底変数の相対コスト係数 c_j - $\pi^T a_j$ がすべて 0 以上なら、最適基底解が得られているので計算終了。
(3) ベクトル $y = B^{-1}a_k$ を計算する。
(4) ベクトル y に正の要素がなければ、問題は有界でないので計算終了。
そうでなければ、(7) 式の Δ および $\Delta = b'_i/y_i$ なる i を求める。
(5) 非基底変数 x_k の値を Δ、それ以外の非基底変数の値を 0 とおく。

基底変数 x_B の値を $x_B = b'$ - $\Delta\ y$ とおく。
非基底変数 x_k を基底変数とし、ステップ (4) で求めた i に対応する基底変数を非基底変数として基底を更新する。
ステップ (1) に戻る。
シンプレックス法の計算が終了したとき、問題が有界でないと判定されるか、または最適解が得られている。

3.5 アニメータ

アニメータは制約ソルバで処理された制約の解を元にして作られたサンプリングデータを受け取ってアニメーションを実行する。メソッド *startAnimation*() が呼ばれて、サンプリングデータに基づき行う。

3.6 サンプリングデータ

アニメーションの単位時間 (フレーム) ごとの値を保持するデータ構造である。アニメーション全体のサンプリングデータを *SamplingData* が管理する。

3.7 Grifon の制約充足系

制約充足系は、図形間に付加されたハイブリッド並行制約と幾何制約 (cassowary) を処理し、それらの制約を充たすアニメーションの実行のために必要なサンプリングデータを生成する。

3.8 Grifon における制約充足系の構成

クラス GrifonSolver が制約充足系の中心となる。ハイブリッド並行制約の処理には Gupta によるインタプリタを、幾何制約の処理には Cassowary を適用した。制約を表すデータ構造はインターフェース GrifonConstraint を実装した、クラス HccConstraint と GeometricConstraint として表される。
上記の図について説明する。クラス HccConstraint とクラス GeomConstraint によって、キャンバス上にかけられている並行制約と幾何制約を取得する。それらの制約をインターフェースである GrifonConstraint を通して、クラス GrifonSolver に送られる。GrifonSolver において、制約の管理を行う。例えば、ハイブリッド並行制約の場合は hcc インタプリタである hcc.dll、幾何制約の場合は Clsolver を GrifonSolver が呼び出して制約の処理を行う。

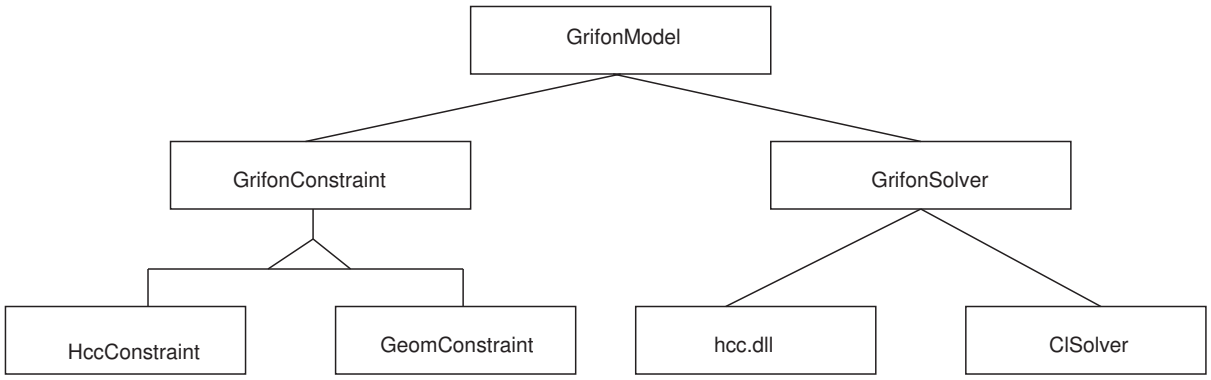

図 3.4: 制約充足系の構成

3.9 ハイブリッド並行制約の処理

ハイブリッド並行制約はクラス HccConstraint として実装している。ハイブ

名前	説明
constants	定数の表
agentCode	エージェント定義の文字列
procedureCalls()	ProcedureCall の表
getConstants()	定数宣言の文字列
getAgentScript()	エージェント定義の文字列を得る
getGoal()	ゴールの文字列を得る

表 3.2: HccConstraint のインスタンス変数とメソッド

リッド並行制約の処理は V.Gupta 氏によるインタプリタ実装によって実行される。Grifon は、ハイブリッド並行制約インタプリタと JNI(Java Native Interface) を介する。インタプリタは、変数値のサンプリング間隔の調整や、Grifon へのサンプリングデータ登録処理の追加などを修正する。
以下に、Grifon におけるハイブリッド並行制約の処理について例を用いて説明する。

ハイブリッド並行制約の XML データ

3.10 幾何制約の処理

幾何制約はクラス GeomConstraint として実装している。まずハイブリッド並行制約が上記と同様の処理によって実行される。ハイブリッド並行制約の処理の後、サンプリングデータを基にアニメータがアニメーションを表示させる。アニメータは幾何制約充足系の処理を適宜呼び出し、幾何制約を充足させる。

3.11 Grifon における制約処理アルゴリズム

以下に、Grifon におけるハイブリッド並行制約と幾何制約を組み合わせた場合の処理アルゴリズムを述べる。

- ステップ 1.
  GrifonSolver はハイブリッド並行制約を処理し、アニメーションの各フレームにおけるパラメタ値のサンプリングデータを生成する。キャンバスからの情報を元にハイブリッド並行制約スクリプトを生成し、ハイブリッド並行制約インタプリタへ渡す。インタプリタは数値計算により離散変化が起こる時点を求める。離散変化時点における処理と、次の時点までの時間の処理を交互に繰り返し、サンプリングデータを生成する。

- ステップ 2.
  GrifonSolver はサンプリングデータに基づき、アニメーションを試行する。

- ステップ 3.
  途中で幾何制約充たされない状態になった (エラーが生じた) 場合、ハイブリッド並行制約インタプリタへ知らせる。インタプリタは知らされた時点から処理を実行する。

- ステップ 4.
  ハイブリッド並行制約、幾何制約が全て充たされたならば処理を終了し、アニメーションを実行する。

第4章　制約階層

前回の章では、Grifonがどのようなアニメーションツールであるか、設計・実装などについて説明した。この章では、一般的な制約階層の研究とGrifonにおける制約の階層化について述べる。

4.1　制約階層

まず、一般的な階層化とは何かについて説明する。階層とは、段階的に層をなすものの各層のことをいう。また階層化することにより全体の見通しが良くなる。例えば、ある一つの会社があったとする。その会社の中には、営業部門、開発部門、広報部門、などの各部門が存在する。そして、その各部門には、それぞれの係りがある。この階層の例では、一番上の階層に会社、その下（二番目の階層）に営業部門、開発部門、広報部門、またその下（三番目の階層）に係りの階層が存在するという階層構造をなしている。制約階層 (constraint hierarchy) は、硬い制約と軟らかい制約という概念を強さ (strength) と呼ばれる、有限段階の優先度へ拡張したものである。いわゆる硬い制約が最も強く、必須 (required) 制約と呼ばれ、それ以外の軟らかい制約は選考 (preferential) 制約と呼ばれる。通常、必須制約の強さはrequiredとして表され、選考制約の強さは強いものから順にstrong、medium、weakとして表現される。
制約階層で制約の強さを適切に利用することで、UIの構築が容易になる．特に、強さは、グラフィカルオブジェクトの振舞いの表現に有効である．典型的には、デフォルトの振舞いを弱い制約で指定しておき、特定の振舞いが必要な状況で、より強い制約を課すという方法が採られる。例えば図4(a)では、オブジェクトa、b、c、dが、4つの強さrequiredの制約で長方形に配置され、その縦・横の間隔が2つのweakの制約で指定されている。この場合、weakの制約は，長方形の大きさの保持というデフォルトの振舞いを指定しており、mediumの制約によってdがドラッグされると、長方形の大きさを保ったまま、他のオブジェクトも移動する。一方、図4(b)のように、新たにstrongの制約でaの位置を固定すると、weakの制約が無視され、dがドラッグされたときの振舞いは、長方形の大きさの変更になる。

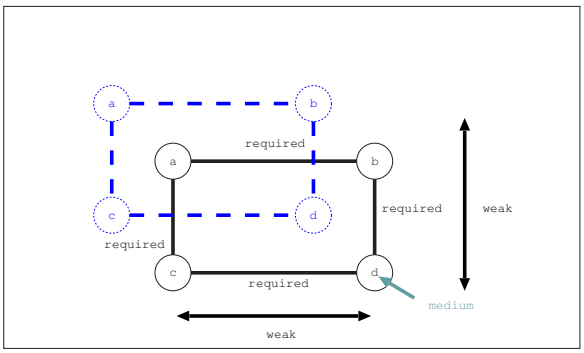

図 4.1: 制約階層の利用例 (a)

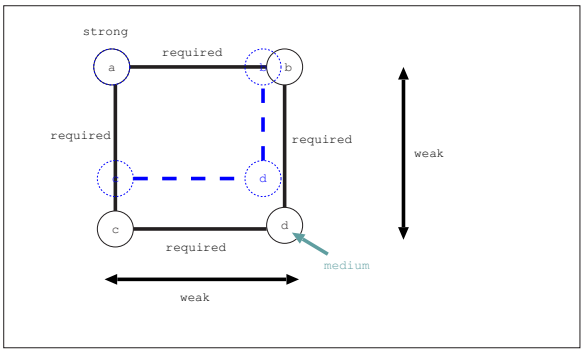

図 4.2: 制約階層の利用例 (b)

4.1.1 制約階層の方式

制約階層とその解

制約階層 Z は、レベル 0 からレベル n までの $(n+1)$ 個のレベルからなるベクトル

$$Z = (Z_0, Z_1, ..., Z_n)$$

であり、その各レベル Z は、強さ k の制約を m_k 個含むベクトル

$$Z_k = (c_{k,1}, c_{k,2}, ..., c_{k,m_k})$$

である。強さは、$0, 1, ..., n$ の順で優先度が高くなり、強さ n の制約は必須制約で、それ以外は選考制約である。強さが記号的に、required,strong, medium,weak と表現される場合、それぞれ強さ 3,2,1,0 を表す。
制約階層の解は、*better* という比較子 (comparator) を用いて定義される。

$better$ は、制約階層 Z に従って 2 つの変数値ベクトル v, v' の解としての「良さ」を比べる述語であり、$better(v, v', Z)$ が真であることは、Z を充足する度合いにおいて v が v' よりも良いことを意味する。この比較の際、強い制約をより良く充たす変数値ベクトルを高く評価するように、比較子は定義される。

比較子

同一の制約階層でも、比較子 の具体的な定義に応じて、その解集合は異なってくる。比較子にはいくつかの具体例が提案されており、ほとんどの比較子は、その定義の方法に応じて、大域的 (global) 比較子または局所的 (local) 比較子のいずれかに分類される。
大域的比較子の具体例として、$least - squares - better(LSB)$ がある。この比較子は、レベル内の矛盾する制約に対して最小 2 乗法を行う。具体的には，各レベルで制約の誤差の 2 乗の総和を計算し、より強いレベルにおいて総和が小さいほど、変数値ベクトルを良く評価する。形式的には，次のように定義される。

$$
\begin{aligned}
&least - squares - better(v, v', Z) \\
&= \exists\ k\ \forall\ k'(k' < k \\
\Rightarrow &\Sigma_i(e(c_{k',i}, v))^2 = \Sigma_i(e(c_{k',i}, v'))^2 \\
\wedge &\Sigma_i(e(c_{k',i}, v))^2 < \Sigma_i(e(c_{k',i}, v'))^2
\end{aligned}
$$

ただし、$e(c, v)$ は、変数値ベクトル v のもとで制約 c の誤差を非負実数として返す、距離的誤差関数である。これは、通常の算術制約については、等式ならば、右辺と左辺の差の絶対値を取ることで定められる。大域的比較子には、LSB 以外にも、レベル内の制約の距離的誤差関数の結果の和を用いる $weighted - sum - better(WSB)$ や，レベル内の制約で最大の距離的誤差関数の結果を用いる $worst - case - better(WCB)$ などがある。
局所的比較子は、レベル内において任意の順序で 1 個ずつ制約の誤差を最小化していったときに、解が得られるように定義されている。局所的比較子には、$locally - error - better(LEB)$ と $locally - predicate - better(LPB)$ がある。LEB は，$locally - metric - better$ とも呼ばれ、距離的誤差関数を用いる。一方、LPB は誤差関数として、制約が正しく充足される場合にを、それ以外の場合はを返す、述語的誤差関数 $(predicate\ error\ function)$ を用いる。実用上、LEB は不等式制約を含む系に、LPB は等式制約のみの系に適しているとされる。
大域的比較子と局所的比較子の大きな相異点として、常に変数値ベクトルを比較可能かどうかという点がある。大域的比較子では、各レベルにおけ

る制約の誤差が、常に比較可能な単一の非負実数に帰着されるため、制約階層全体においても変数値ベクトルは常に比較可能となる。一方、局所的比較子では、レベル内における制約の誤差の最小化の順序の違いによって、変数値ベクトルを比較できない場合が生じることがある。このような性質から、一般に、大域的比較子よりも局所的比較子の方が、解の条件が緩いために解の個数が多くなり、高速な制約解消アルゴリズムを設計しやすくなる傾向がある。
以下に,、具体的な比較子を用いて制約階層を解消する例として、次の制約階層 Z を LSB で解く場合を説明する。

$$\begin{aligned} required &\quad x_1 = x_2 \\ strong &\quad x_2 + 1 = x_3 \\ weak &\quad x_1 = 0 \\ weak &\quad x_3 = 3 \end{aligned}$$

最初に、$required$ レベルの制約を充たすよう、$S_0(Z) = (v_1, v_2, v_3)|v_1 = v_2$ 次に、$S_0(Z)$ の中で, より良い変数値ベクトルが存在しないものの集合として、解集合 $S(Z)$ が決定される。変数値ベクトルの比較は、LSB 等の大域的比較子では、各レベルの誤差を表す非負実数を、強いレベルから順に比べることで可能である。比較の例を示すため、表に，$S_0(Z)$ の要素の内、$(0,0,1)$,$(1,1,2)$,$(2,2,3)$,$(0.0.3)$ に対応するレベルの誤差を列挙する (例えば、$(0,0,1)$ に対する weak レベルの誤差は、$(v_1 - 0)^2 + (v_3 - 3)^2 = 4$ として計算される)。まず、(0,0,3) など、$strong$ レベルの誤差が 0 でない変数値ベクトルは，他により良いものが存在するため、解になり得ない。また、$strong$ レベルの誤差がになる変数値ベクトルの中でも、$(0,0,1)$ や $(2,2,3)$ などは、$(1,1,2)$ より良くないため、解ではない。一方、$S_0(Z)$ のいかなる要素も $(1,1,2)$ より良くはない (実際に、$v_1 = v_2$ かつ $v_2 + 1 = v_3$ という条件のもとで、$v_1^2 + (v_3 - 3)^2$ を最小化することで確認できる) ことから、最終的に、唯一の解 $(1,1,2)$ からなる解集合 $S(Z)$ が求められる。

$$S(Z) = (1,1,2)$$

4.1.2 制約階層のアルゴリズム

以下では、制約階層のための制約解消アルゴリズムを、精製法、局所伝播法、最適化アプローチに分類し、解説する。

(v_1, v_2, v_3)	$strong$	$weak$
$(0, 0, 1)$	0	4
$(1, 1, 2)$	0	2
$(2, 2, 3)$	0	4
$(0, 0, 3)$	1	0

表 4.1: 変数値ベクトルに対するレベルの誤差

精製法

精製法は、各レベルを強い順に充足することで、制約階層を解消する。精製法の具体例としては、以下のようなものがある。Orange アルゴリズムでは、各レベルにシンプレックス法を適用することで、1 次等式と 1 次不等式からなる制約階層を WSB または WCB で充足している。DeltaStar は、Orange を一般化したアルゴリズムで、大域的比較子だけでなく局所的比較子にも対応している。

局所伝播法

局所伝播法を採用した制約解消法は、通常、強さを与えられた多方向制約の系を局所的比較子で解く。単調なデータフロー方式の場合と同様、制約解消は，プランニングと実行の 2 段階からなる。プランニング段階では、制約の強さを考慮し、制約階層の解の定義に従うように、充足すべき制約のメソッドを選択する。一方、実行段階では、単方向制約の場合と同様の局所伝播法を実行する。
Blue は、制約階層のための最初の局所伝播法アルゴリズムである。既知状態伝播法を実行しながら、各ステップで選択すべきメソッドを検索する際に、制約を強い順に調べるようにしている。この方法で、循環を必要としない場合には、LPB 解を得ることが可能である。
DeltaBlue は、プランニングをインクリメンタルに行うことで、LPB 解を効率的に求めるアルゴリズムである。具体的には、新規の制約が追加された場合、または既存の制約が削除された場合に、現在のメソッド選択を部分的に修正することで、新しいメソッド選択を得るようにしている。その効率化の鍵となっているのは、$walkabout\ strength$ と呼ばれる手法である。これによって、制約追加の際に、その制約を充足すべきかどうか素早く判断した上で、充足すべき場合には、それまで充足されていた制約の中からメソッド選択を解除すべきものを高速に発見することを可能にしている。
DeltaBlue 以降、様々なインクリメンタルアルゴリズムが提案されている。

SkyBlue は、DeltaBlue をより一般化し、制約の循環的関係と、多出力のメソッドを持つ制約に対応したものである。QuickPlan は、自由度伝播法を基礎としたアルゴリズムで、循環的関係を生じずに解く方法がある場合に、それを必ず発見することを保証している。DETAIL は、局所伝播法の枠組を拡張して、LSB などの比較子を可能にしたアルゴリズムで、制約の循環的関係にも対応している。

最適化アプローチ

最適化アプローチは、強さを重みに対応させ、制約階層を線形計画法などの 1 つの最適化問題に変換して解く。例えば，上記で与えた制約階層は、次のような最適化問題に変換される。

$$\begin{aligned} \text{minimize} \quad & w_{strong} f(\varepsilon_1) + w_{weak} f(\varepsilon_2) + w_{weak} f(\varepsilon_3) \\ \text{subject to} \quad & x_1 = x_2 \\ & x_2 + 1 = x_3 + \varepsilon_1 \\ & x_1 = 0 + \varepsilon_2 \\ & x_3 = 3 + \varepsilon_3 \end{aligned}$$

ただし、w_{strong}、w_{weak} は、それぞれ強さ $strong$、$weak$ に対応する重みである。重みは、対応する強さが強いほど、重くなるように決められる (制約階層に従うよう実現されることもあれば、近似的に実現されることもある)。最適化アプローチを採用した制約解消系には、シンプレックス法を応用して LEB を求める Cassowary と、アクティブセット法を用いて近似的な LSB 解を求める QOCA がある。

第5章　Grifonにおける制約の階層化の設計・実装

この章では前章で述べた制約の階層化を応用し、Grifon における制約の階層化の設計と実装について述べる。

5.1　現状のGrifonの制約処理アルゴリズム

前述の現状の Grifon の制約処理アルゴリズムについて記す。

- ステップ 1.
  GrifonSolver はハイブリッド並行制約を処理し、アニメーションの各フレームにおけるパラメタ値のサンプリングデータを生成する。キャンバスからの情報を元にハイブリッド並行制約スクリプトを生成し、ハイブリッド並行制約インタプリタへ渡す。インタプリタは数値計算により離散変化が起こる時点を求める。離散変化時点における処理と、次の時点までの時間の処理を交互に繰り返し、サンプリングデータを生成する。
- ステップ 2.
  GrifonSolver はサンプリングデータに基づき、アニメーションを試行する。
- ステップ 3.
  途中で幾何制約充たされない状態になった (エラーが生じた) 場合、ハイブリッド並行制約インタプリタへ知らせる。インタプリタは知らされた時点から処理を実行する。
- ステップ 4.
  ハイブリッド並行制約、幾何制約が全て充たされたならば処理を終了し、アニメーションを実行する。

5.2 現状の Grifon における制約データフォーマット

```
<?xml version="1.0" encoding="Shift_JIS" ?>
<grifon xmlns="http://www.ueda.info.waseda.ac.jp/grifon">
 <constraint name="制約の名前">
  <script type="text/hcc">
  <![CDATA[
  Hybrid CC のスクリプト
  /*
  ]]>
  </script>
 </constraint>
</grifon>
```

表 5.1: 現状の制約データ

5.3 XML

拡張可能なマーク付け言語 XML(eXtensible Markup Language) は，XML 文書というデータオブジェクトのクラスを規定し，XML 文書を処理するプログラムの動作の一部を規定する。XML は，SGML(Standard Generalized Markup Language) の制限したサブセットとする。XML 文書は，必ずSGML 規格に適合する。

XML 文書は実体という記憶単位から成り，実体は構文解析されるデータ又は構文解析されないデータから成る。構文解析されるデータは，文字から成り，その一部は文字データを構成し，一部はマーク付けを構成する。マーク付けは，文書の記憶レイアウト及び論理構造を記述する符号とする。XML は，記憶レイアウト及び論理構造についての制約条件を記述する機構を提供する。文書の内容が要素（element）という単位からなっている。マーク付け <> をタグ（tag）と呼ぶ。タグの名前は、自由にきめて良い。

5.4 Grifonのタグセット

以下にGrifonで使われているデータのタグセットを挙げる。

5.4.1 PSVG

- psvg要素
  PSVGデータのルート要素となる。一つのpsvg要素が一つのPSVGを表す。PSVGのIDはid属性によって設定される。
- description要素
  description要素には、PSVGの説明文を記述する。
- defaultShape要素
  図形情報を表す。defaultShapeにはSVGデータを記述する。パラメタによる操作対象となる要素には、PSVGドキュメント中で一意となるようなid属性を設定する必要がある。
- parameter要素、parameters要素
  一つのparameter要素が一つのパラメタを表し、複数のparameter要素をparameters要素として表現する。以下にparameter要素が持つ属性を示す。

5.4.2 Grifonにおけるハイブリッド並行制約

以下にハイブリッド並行制約のデータ例 (ビリヤードのテーブルの例) を示す。

- constraint要素
  制約を表す。一つのconstraint要素が一つの制約を表す。constraint要素の内容を以下に示す。
  ・description要素
  ・vars要素
  ・constants要素
  ・agnet要素
  ・goal要素
- description要素
  制約の説明文 (コメント文) を記述する。

```
  <?xml version="1.0" encoding="Shift_JIS" ?>
<grifon xmlns="http://www.ueda.info.waseda.ac.jp/ucae">
<constraint name="billiardTable2">
  <desc xml:lang="ja">ビリヤード台。</desc>
  <constants>
    <constant name="tx" value="100" />
         *(以下省略)
  </constants>
  <procedure name="Edges">
    <script type="text/hcc">
      <![CDATA[
Edges = (xMax, yMax)[]{
           *(以下省略)
},
  ]]>
   </script>
   </procedure>
   <procedure name="Pockets">
     <script type="text/hcc">
       <![CDATA[
Pockets = (pocket, xMax, yMax){
           *(以下省略)
},
  ]]>
   </script>
   </procedure>
   <goal>
     <procedureCall name="Edges" />
     <procedureCall name="Pockets" />
   </goal>
</constraint>
</grifon>
```

表 5.2: ハイブリッド並行制約の例 (ビリヤードテーブル)

- var、vars 要素
  制約の変数を表す。一つの var 要素が一つの変数を表し、複数の要素を vars 要素でまとめる。vars 要素には type 属性と name 属性がある。type 属性で変数の型を決める。また、name 属性で変数の名前を決定する。
- constant 要素、constants 要素
  制約中で用いられる定数である。一つの constant 要素が一つの定数を表し、複数の要素を constants 要素でまとめる。constant 要素には name 属性と value 属性があり、name 属性は定数の名前を、value 属性は定数の値を設定する。
- agent 要素
  エージェント情報を表す。in 要素、out 要素、script 要素を持つ。in 要素によりエージェントのプロシージャ呼び出しの引数を、out 要素にエージェント変数を設定する。この変数を使って、サンプリングデータを読み取る。script 要素がエージェント情報の本体を表す。ハイブリッド並行制約を定義するプログラムを記述し、script 要素はその文字列情報を持つ。script 要素において、CDATA セクションとして記述するか、外部ファイルに記述するという 2 つの方法がある。
- goal 要素
  制約のゴール情報を表す。goal 要素は procedureCall 要素を子に持つ。procedureCall 要素はプロシージャの呼び出しを行う。name 属性に呼び出すべきプロシージャの名前を設定する。varLink 要素を子に持つ。varLink 要素は var 要素と agentVar 要素を結びつける役割を果たす。name1 属性に agentVar 要素の名前を、name2 属性に var 要素の名前を設定し、これらを結びつける。

5.5 制約データの扱い

先に示した XML での部品データを XML-Parser を通して解析し、DomTree を作る。DomTree とは、読み込んだ XML 文書を木（ツリー）の形に変換されたデータのことである。Grifon では、作成された DomTree は Java 側で管理され、内部構造に変換される。

5.6 現状の Grifon における問題点

以下に現状の Grifon における問題点を示す。

– 問題点 1.
Grifon において、同物体に矛盾し合うハイブリッド並行制約をかけた場合、制約充足せずにアニメーションを実行できない (*Constraint Error* が生じてしまう)。

– 問題点 2.
Grifon は、ハイブリッド並行制約より幾何制約優先のため、アニメーションを表現する幅が狭い。

5.6.1 制約過多な状況 (ハイブリッド並行制約のみの場合)

例えば、あるボールに falling、spring 、inelastic 、penetrating という４つの制約をかける。
falling : 物体の自由落下制約
spring : 物体が弾む制約
inelastic : 物体が弾まない制約
penetrating:物体を通り抜ける制約

しかし、このボールが地面に到達した場合、Spring、Inelastic、Penetrating の３通りの動きが考えられる。 現在の Grifon では、このような矛盾し合う

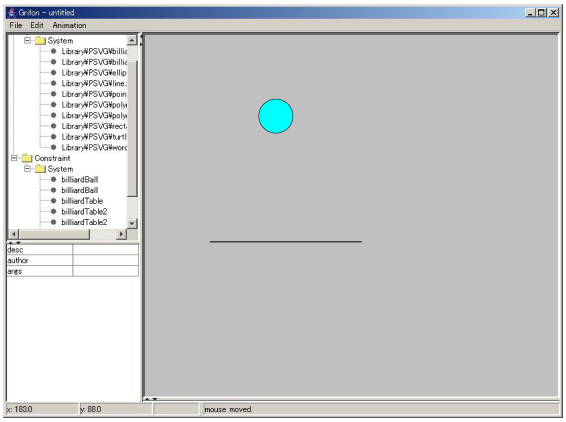

図 5.1: 複数の制約が矛盾し合う制約過多な例

制約過多な状況を処理できない (Constraint Error が発生する)。

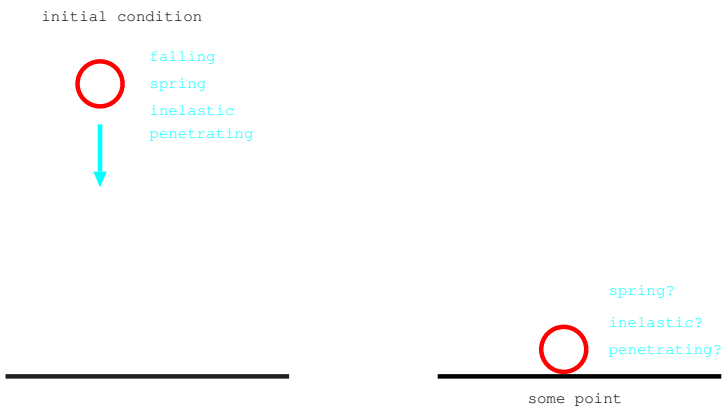

図 5.2: 複数の制約が矛盾し合う制約過多な例

5.6.2 制約過多な状況 (ハイブリッド並行制約と幾何制約の場合)

下図は、階段上にドミノが等間隔で置かれていて、このドミノが倒れるアニメーションである。下図における幾何制約とハイブリッド並行制約について説明する。ドミノ同士の間隔を等しくする幾何制約 (LengthX)、ドミノが倒れるハイブリッド並行制約がそれぞれ図形にかけられている。 上図

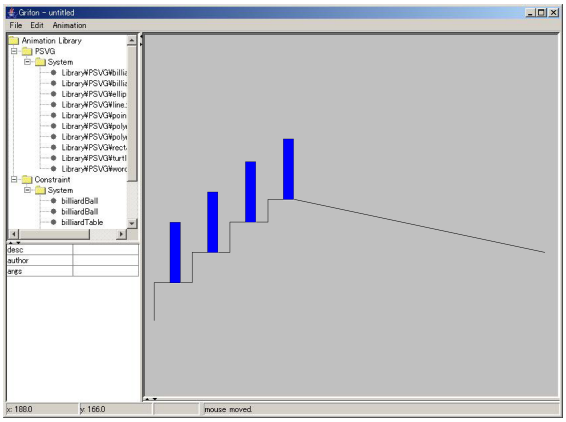

図 5.3: ドミノ倒しの例 (Grifon のスクリーンショット)

の場合、一般ユーザはドミノが坂道を滑り落ちるアニメーションにしたい。しかし、現在の Grifon で実行すると、階段一番上のドミノが倒れないアニメーションになる。これは、ドミノが倒れるというハイブリッド並行制約よりドミノを等間隔に保つという幾何制約の方が優先されるからである。

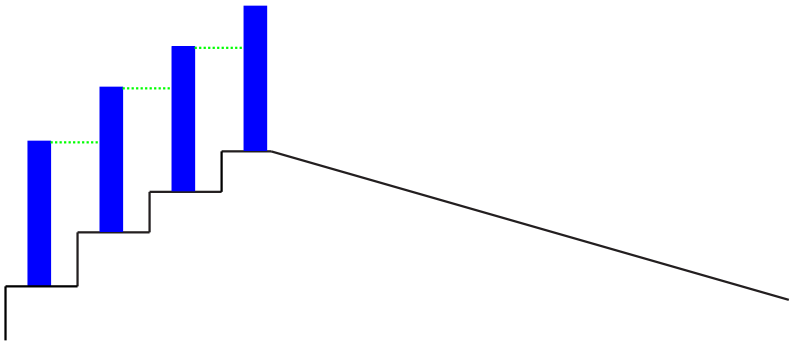

図 5.4: ドミノ倒しの例

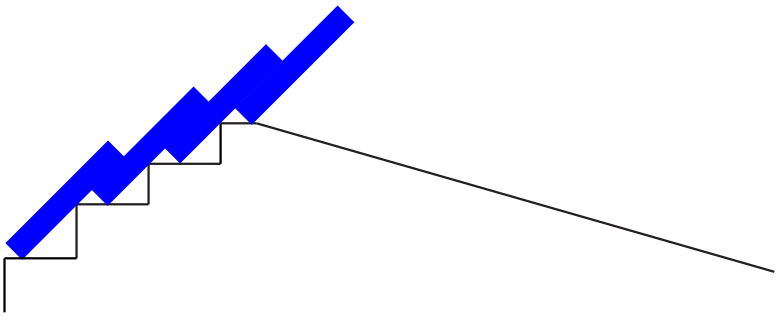

図 5.5: ドミノ倒しの例 (幾何制約優先の場合)

5.7 制約の優先度の設計

先に示した並行制約データに制約の優先度を加える。優先度は $required$、$strong$、$medium$、$weak$ と定義付ける。それぞれの優先度の重み付けは、

$$required = 4$$
$$strong = 3$$
$$medium = 2$$
$$weak = 1$$

という実数値で設定する。また、ハイブリッド並行制約と幾何制約の優先度を切り替えるための優先度 (重み付け) も付随する。ハイブリッド並行制約と幾何制約における重み付けは、$hcc = 0$ が幾何制約優先、$hcc = 1$ がハイブリッド並行制約優先とする。以下に優先度を付随した制約データフォーマットを示す。

```
<?xml version="1.0" encoding="Shift_JIS" ?>
<grifon xmlns="http://www.ueda.info.waseda.ac.jp/grifon">
 <constraint name="制約の名前"
             strength="制約の強さ"
              weight="優先度の重み付け"
               hcc="重み付け">
  <script type="text/hcc">
   <![CDATA[
  Hybrid CC のスクリプト
 /*
  ]]>
  </script>
 </constraint>
</grifon>
```

表 5.3: 優先度を付随した制約データフォーマット

5.8 制約処理のアルゴリズム (ハイブリッド並行制約優先の場合)

以下に、ハイブリッド並行制約優先の場合の Grifon における制約処理アルゴリズムについて述べる。

- ステップ 1.
  GrifonSolver において、ハイブリッド並行制約、幾何制約のどちらを優先するかを比較する。また、キャンバス上でかけられた全てのハイブリッド並行制約のスクリプトを生成し、ハイブリッド並行制約インタプリタへ渡す。インタプリタは数値計算により離散変化が起こる時点を求める。離散変化時点における処理と、次の時点までの時間の処理を交互に繰り返し、サンプリングデータを生成する。
- ステップ 2.
  GrifonSolver はサンプリングデータを元にアニメーションを試行する。
- ステップ 3.
  ステップ 1 で「Constraint Error」が生じた場合、ハイブリッド並行制約の優先度を比較して、優先度の一番低い制約以外の制約をハイブリッド並行制約インタプリタへ渡す。
- ステップ 4.
  制約充足後、サンプリングデータを生成し、アニメーションを実行する。

5.9 制約処理アルゴリズム (幾何制約優先の場合)

- ステップ 1.
  GrifonSolver において、ハイブリッド並行制約、幾何制約のどちらを優先するかを比較する。また、キャンバス上でかけられた全てのハイブリッド並行制約のスクリプトを生成し、ハイブリッド並行制約インタプリタへ渡す。インタプリタは数値計算により離散変化が起こる時点を求める。離散変化時点における処理と、次の時点までの時間の処理を繰り返し、サンプリングデータを生成する。
- ステップ 2.
  GrifonSolver はサンプリングデータを元にアニメーションを試行する。
- ステップ 3.
  処理の途中で幾何制約が充たされない状態が生じた場合、ハイブリッド並行制約インタプリタへ知らせる。ハイブリッド並行制約インタプリタは知らされた時点から処理をやり直す。

– ステップ 4.
ハイブリッド並行制約、幾何制約を充たしたサンプリングデータが生成されたならば、処理を終了する。

5.10 Grifon における制約の階層化の実装

本研究において、主にクラス DomConstraintFactory とクラス GrifonSolver の実装を行った。以下に、実装モデルを示す。
実装モデルについて説明する。Constraint Data は、XML で記述されたハイブリッド並行制約データである。このハイブリッド並行制約データをクラス DomConstraintFactory が取得し、保持する。クラス GrifonSolver は制約充足系の根幹であり、ハイブリッド並行制約充足系と幾何制約充足系を一括管理する。そして、その保持されたハイブリッド並行制約データを GrifonSolver で扱う。

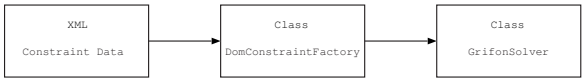

図 5.6: 実装モデル

5.10.1 DomConstraintFactory

クラス DomConstraintFactory は、キャンバス上に存在するハイブリッド並行制約データ (XML) を取得する役割を担う。以下にクラス DomConstraint-Factory の一部を示す。以下の部分は、ハイブリッド並行制約データにおける、「制約の名前、強さ、優先度の重み付け、重み付け」を取得するプログラムである。

5.10.2 GrifonSolver

クラス GrifonSolver は、Grifon において幾何制約とハイブリッド並行制約を扱う制約処理系である。本研究では GrifonSolver において、以下の処理を実装した。

– DomConstraintFactory が取得した制約データ (XML) を扱う

– 幾何制約、ハイブリッド並行制約の優先順位を扱う

```
Element constraintElement =
DomUtils.getFirstChildElement(document.
                              getDocumentElement());
 // 制約の名前、強さ、優先度の重み付け、重み付けを取得
 String name = constraintElement.getAttribute(ATTR_NAME);
 if (name == null || name == "") {
   throw new RuntimeException("constraint name is null");
 }
  * 以下省略 (制約の強さ、重み付けも同様)
```

表 5.4: ソースコード (DomConstraintFactory) の一部

第6章　結果・考察

Grifon における制約の階層化に基づき、幾何制約優先、ハイブリッド並行制約優先のアニメーションをそれぞれ考察する。

6.1　結果と考察

6.2　例:囲壁内を弾むボールと四角形

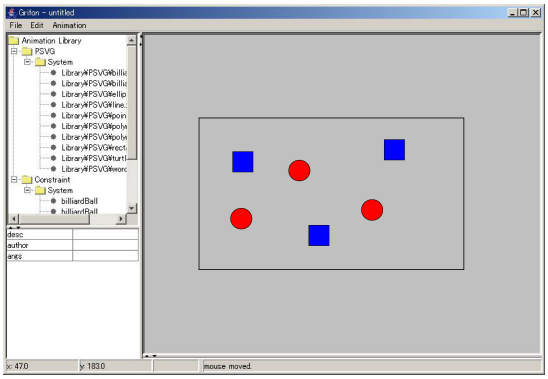

図 6.1: 囲壁内を弾むボールと四角形 (Grifon のスクリーンショット)

ボールに、物体が移動する move(ハイブリッド並行制約)、物体とぶつかって跳ね返る collision 制約 (ハイブリッド並行制約) をかける。また、囲壁の４点に座標を固定する Pin 制約 (幾何制約) と点同士を結ぶ Unify 制約 (幾何制約)、囲壁の辺 bc に move 制約 (ハイブリッド並行制約) をかける。この例は、ある特定のボールが辺 bc に３回当たると辺 bc が移動する例である。

6.2.1　囲壁内を弾むボールと四角形 (幾何制約優先の場合)

まず、この例を幾何制約優先の場合について考える。ある特定のボールが辺 bc に３回ぶつかったとする。この時に、辺 bc は move 制約がかけられて

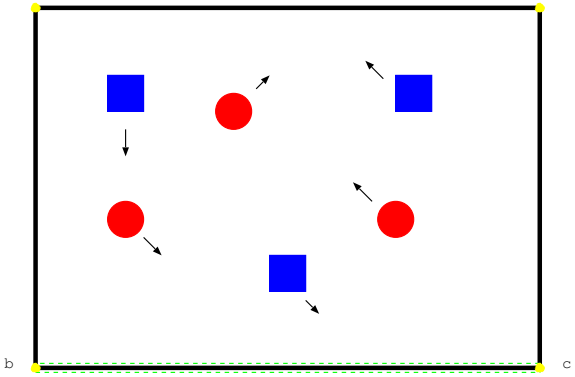

図 6.2: 囲壁内を弾むボールと四角形

いるため移動するはずである。しかし、幾何制約である Pin 制約と Unify 制約が優先であるので、辺 bc は移動しないアニメーションになる。

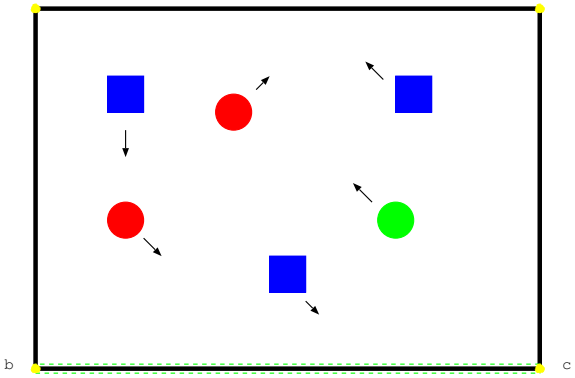

図 6.3: 囲壁内を弾むボールと四角形 (幾何制約優先の場合)

6.2.2 囲壁内を弾むボールと四角形 (ハイブリッド並行制約優先の場合)

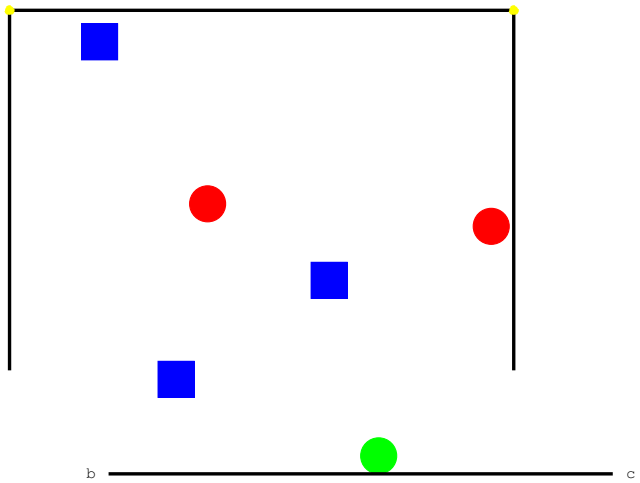

図 6.4: 囲壁内を弾むボールと四角形 (ハイブリッド並行制約優先の場合)

第7章　これまでのまとめと今後の課題

7.1　これまでのまとめ

7.2　今後の課題と目標

今後の課題は、

- 幾何制約とハイブリッド並行制約の優先度に互換性を持たせる。
- アニメーションライブラリの充実化

である。本章では、これらについて考察する。

7.3　幾何制約とハイブリッド並行制約の優先度に互換性を持たせる

幾何制約と並行制約の優先度に互換性を持たせる。前章で述べた囲壁内を弾むボールと四角形のアニメーション例について考察する。
ボールに、物体が移動する move(ハイブリッド並行制約)、物体とぶつかって跳ね返る collision 制約 (ハイブリッド並行制約) をかける。また、囲壁の４点に座標を固定する Pin 制約 (幾何制約) と点同士を結ぶ Unify 制約 (幾何制約)、囲壁の辺 bc に move 制約 (ハイブリッド並行制約) をかける。

7.3.1　$pin < move < unify$

この例は、ある特定のボールが辺 bc に当たると辺 bc が移動する例である。囲壁内を弾むボールと四角形のアニメーション例において辺 bc に以下のように制約をかける。

$$pin < move < unify$$

上記のように、幾何制約とハイブリッド並行制約の優先度に持たせることを考える。ある特定のボールが辺 bc に当たったときの辺 bc の動作を考察する。辺 bc に pin(幾何制約) $<$ $move$(ハイブリッド並行制約) $<$ $unify$(幾何制約) がかけられているので、辺 bc は四角形の形状を崩さずにある特定のボールが当たる度に y 軸方向に移動していくアニメーションになる。

図 7.1: 囲壁内を弾むボールと四角形 ($pin < move < unify$)

7.3.2 $move < pin < unify$

囲壁内を弾むボールと四角形のアニメーション例において辺 bc に以下のように制約をかける。

$$move < pin < unify$$

上記のように、幾何制約とハイブリッド並行制約の優先度を持たせることを考える。この場合、$move$ 制約 (ハイブリッド並行制約) の優先度が pin(幾何制約) 、$unify$(幾何制約) より弱い。よって、ある特定のボールが辺 bc に当たった瞬間においても辺 bc は移動しない ($move$ 制約は無視され、pin と $unify$ 制約が実行される)。これは本研究における幾何制約優先の例と同じである。

7.3.3 $unify < pin < move$

囲壁内を弾むボールと四角形のアニメーション例において辺 bc に以下のように制約をかける。

$$unify < pin < move$$

上記のように、幾何制約とハイブリッド並行制約の優先度を持たせることを考える。この場合、$move$ 制約 (ハイブリッド並行制約) の優先度が pin(幾何制約) 、$unify$(幾何制約) より強い。よって、ある特定のボールが辺 bc に当たったらボールと辺 bc が共に移動するアニメーションになる。これは本研究におけるハイブリッド並行制約優先の例と同じである。

7.3.4 $unify < move < pin$

7.4 例:ドミノ倒し

図 7.2: ドミノ倒しの例 (Grifon のスクリーンショット)

下図は、階段上にドミノが等間隔で置かれていて、このドミノが倒れるアニメーションである。下図における幾何制約とハイブリッド並行制約について説明する。ドミノ同士の間隔を等しくする幾何制約 (LengthX)、ドミノが倒れるハイブリッド並行制約がそれぞれ図形にかけられている。

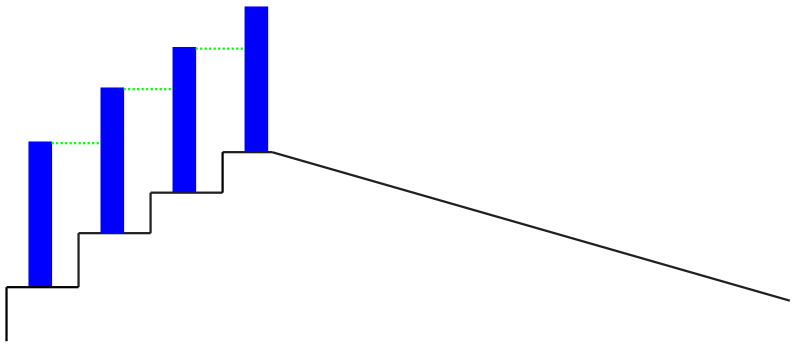

図 7.3: ドミノ倒しの例

7.4.1 ドミノ倒しの例 (幾何制約優先の場合)

上図の場合、一般ユーザはドミノが坂道を滑り落ちるアニメーションにしたい。しかし、幾何制約優先の場合、階段一番上のドミノが倒れないアニメーションになる。これは、ドミノが倒れるというハイブリッド並行制約よりドミノを等間隔に保つという幾何制約の方が優先されるからである。以下で幾何制約優先の場合における Grifon 内の処理について考察する。

- ステップ 1.
  GrifonSolver において、ハイブリッド並行制約、幾何制約のどちらを優先するかを比較する。今回の場合は、幾何制約である。キャンバス上でかけられたハイブリッド並行制約のスクリプトを生成し、ハイブリッド並行制約インタプリタへ渡す。離散変化時点における処理と、次の時点までの時間の処理を交互に繰り返し、ドミノのサンプリングデータを生成する。
- ステップ 2.
  GrifonSolver はサンプリングデータに基づき、アニメーションを試行する。
- ステップ 3.
  処理の途中で幾何制約が充たされない状態が生じた場合、ハイブリッド並行制約インタプリタへ知らせる。ハイブリッド並行制約インタプリタは知らされた時点から処理をやり直す。
- ステップ 4. ハイブリッド並行制約、幾何制約を充たしたサンプリングデータが生成されたならば、処理を終了する。

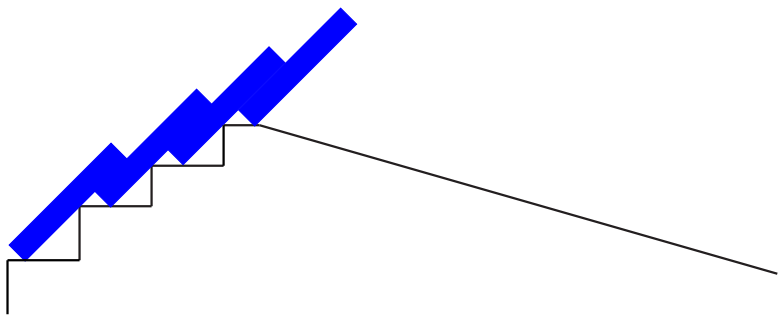

図 7.4: ドミノ倒しの例 (幾何制約優先の場合)

7.4.2 ドミノ倒しの例 (ハイブリッド並行制約優先の場合)

下図はハイブリッド並行制約を優先にしたアニメーションである。先に述べた階段一番上のドミノが倒れないというアニメーションではなく、坂道を滑り落ちるアニメーションである。これは、ドミノが倒れるというハイブリッド並行制約がドミノを等間隔に保つという幾何制約より優先されるからである。具体的には、階段上ドミノの下から順に３つは幾何制約優先、階段一番上のドミノはハイブリッド並行制約優先にする。
以下でハイブリッド並行制約優先の場合における Grifon 内の処理について考察する。

- ステップ 1.
  GrifonSolver において、ハイブリッド並行制約、幾何制約のどちらを優先するかを比較する。今回の場合は、ハイブリッド並行制約である。キャンバス上でかけられたハイブリッド並行制約のスクリプトを生成し、ハイブリッド並行制約インタプリタへ渡す。離散変化時点における処理と、次の時点までの時間の処理を交互に繰り返し、ドミノのサンプリングデータを生成する。
- ステップ 2.
  GrifonSolver はサンプリングデータに基づき、アニメーションを試行する。
- ステップ 3.
  互いに矛盾し合うハイブリッド並行制約がなく、「Constraint Error」が生じなかったので、サンプリングデータに基づきアニメーションを実行する。

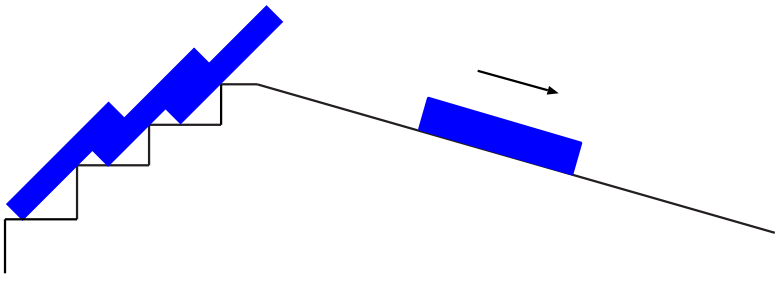

図 7.5: ドミノ倒しの例 (ハイブリッド並行制約優先の場合)

7.4.3　アニメーションライブラリの充実化

アニメーションライブラリを充実させていく必要がある。

謝辞

本研究を進めるに辺り、ご指導いただきました上田和紀教授に深く感謝致します。また、最後まで見捨てずにいてくださった石井大輔さん、若槻聡一郎君に感謝致します。描画班の飯盛幸太郎君、宇佐美智史君、飛田伸一君に感謝致します。またよく徹夜に付き合ってくれた大野太郎君に感謝致します。

参考文献

[1] 石井大輔:制約に基づくアニメーション作成システム Grifon の設計と実装,
早稲田大学大学院理工学研究科情報科学専攻修士論文 (2003).

[2] 若槻聡一郎:制約に基づくアニメーション作成システム Grifon におけるデータ構造の設計と実装,
早稲田大学理工学部情報学科卒業論文 (2003).

[3] Borning A., Marriott K., Stuckey P., and Xiao Y.: Solving Linear Arithmetic Constraints for User Interface Applications,
In *Proceedings of the 1997 ACM Symposium on UIST*, 1997, pp. 87–96.

[4] Borning A., Duiserg R., Freeman-Benson., Kramer A.,and Woolf M.: Constraint Hierarchies,
Proc. ACM OOPSLA, 1987, pp. 48–60.

[5] Borning A.,Freeman-Benson., : UltraViolet:A Constraint Satisfaction Algorithm for Interactive Graphics Constraints,1998.

[6] Bjorn Carlson., Vineet Gupta.: An implementation of Hybrid CC. Submitted for publication, April 1996

[7] Bjorn Carlson.,Vineet Gupta.:The hcc Programmer's Manual,1999

[8] Vineet Gupta.,Radha Jagadeesan.,Vijay Saraswat.,Daniel G Bobrow.: Programming in hybrid constraint languages,1997.

[9] Vineet Gupta.,Radha Jagadeesan.,Vijay Saraswat.,Daniel G Bobrow.: Computing with continuous change,
Technical report,Xerox Palo Alto Research Center,May 1995.

[10] 細部博史:モジュール機構を備えた幾何制約解消系,
WISS 2000.

[11] Scalable Vector Graphics1.1 Specification-Japanese translation.
`http://www.hcn.zaq.ne.jp/`$_{/REC-SVG11-20030114/}$$GuardedHornClauses(GHC)$, および並行論理プログラミ

[12] 細部博史:ユーザーインターフェースにおける制約解消法の研究動向,
`http://www.ueda.info.waseda.ac.jp/ ueda/ghc.html`

[13] SICStus Prolog Homepage,
http://www.sics.se/isl/sicstus.html.

[14] Hiroshi Hosobe:A Geometric Constraint Library for 3D Graphical Applications.

[15] Hiroshi Hosobe,Satoshi Matsuoka, and Akinori Yonezawa: Generalized Local Propagation:A Framework for Solving Constraint Hierarchies.

[16] 平川正人・安村通晃編:bit 別冊 ビジュアルインターフェース, 共立出版 (1996).

[17] M・オローク:3 次元コンピュータ・アニメーションの原理第 2 版, 近代科学社, 2000,pp.143-172.

[18] Juno-2 Homepage,
http://www.research.digital.com/SRC/juno-2/juno-2-home.html.

[19] Alexander Bockmayr,Arnaud Courtois: Modeling biological systems in hybrid concurrent constraint programming,LORIAUniversity Henri Poincare Nancy,France.

[20] 大石進一:非線形解析入門, 現代非線形科学シリーズ, コロナ社,1997.

[21] 矢部博, 八巻直一:非線形計画法, 応用数値計算ライブラリ, 朝倉書店,1999.

[22] 福島雅夫:数値計画入門, システム制御情報ライブラリー, システム制御情報学会編, 朝倉書店,1996. XML 中山幹敏, 奥井康弘, 吉田稔, 山本弘明, 太田純, 吉田晃伸, 村上泰介, 上戸鎖めぐみ: 改訂版標準 XML 完全解説上下, 技術評論社,2001.

[23] Erich Gamma : “Design Pattern”, Addison-Wesley Publishing Company (1995). オブジェクト指向における再利用のためのデザインパターン, 本位田 真一, 吉田 和樹 監訳 (ソフトバンク, 1995).