2007年度　修士論文

ハイブリッド並行制約プログラミングにおける制約の階層化

提出日　：　2008年2月6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　：　3606U079-7

西村 光弘

概 要

ハイブリッド並行制約プログラミング言語 (以下 HCC) はハイブリッドシステム（離散的変化と連続的変化からなるシステム，例えば物体の移動と衝突のモデル）の表現に適した言語である．現実世界を表現する際に用いる常微分方程式を制約として扱う．その制約を処理系内部で高精度な求解アルゴリズムを用いているため，複雑な計算手法は不要となり解きたい問題の記述のみでサンプリングデータを生成することができる．

しかし，現状の HCC 処理系では制約に強さ（優先度）の概念は導入されていない．その問題点は，どの制約も必ず満たされなければならず，ユーザは１つの制約間の矛盾でさえ許されない．並行言語でもあるがゆえに処理の流れを追うことが困難であり，また，ユーザによる制約間のチェックは制約の数の組み合わせの個数でほぼ増大する．プログラミング言語でのバグを現実的な時間内に全ては解消しきれないとすると，HCC にもある程度の柔軟性は必要である．その解決策として制約階層が有効であると考える．

本研究の目的は，コーディングにおいて制約過多，制約不足といった状況の解消支援をするため，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．関連研究に比較子と呼ばれる同階層上で最適解を導き出す研究や，制約階層で制約の強さを適切に利用することで、UI においてユーザ支援の研究が行われている．従来の対象は幾何制約であったものに対し本研究は時間概念のある非線形制約も含み，目的も異なるため適用可能性，有効性を示すことは意義があると考える．

本研究では，記述性の多様化・デバグへの有効性・フールプルーフ設計・可読性の向上を評価基準に多くの例題をもとに検証を行い有効性を示した．また，３通りのアルゴリズムに従い，階層化設計について各々実験を行った．その実験結果をもとに現状最良と思われる設計を考案した．

目 次

図 目 次

表 目 次

第1章　研究の背景と目的

1.1　研究の背景

ハイブリッド並行制約プログラミング (hybrid cc) は連続および離散からなるハイブリッドシステムを簡潔かつ直感的な記述で表現できる高水準モデリング言語である．微分方程式を制約という形で容易に表現でき，現実世界の物理現象を制約を用いて記述しシミュレーションを行うことが可能である．

しかし，HCC での制約には強さ（優先度）の概念は導入されていない．つまり，全ての制約が，必ず満たされなければならない強い制約と呼ばれる制約であり，ユーザは１つの制約間の矛盾でさえ許されない．また並行言語でもあるがゆえにユーザによる制約間のチェックは制約の数の組み合わせの個数でほぼ増大する．プログラミング言語でのバグを現実的な時間内に全ては解消しきれないとすると，HCC にもある程度の柔軟性は必要であると考えた．

1.2　研究の目的とその意義

本研究の目的は，制約過多の状況下でコーディング，デバグ両面においてユーザが扱いやすいものにするため，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．この制約階層により HCC 処理系の冗長性が増し，コーディングの際はユーザの思いがけないミスを吸収し実行できる．デバグの際には同一階層上の制約のみに注視したらよくなり，エラー原因を特定する範囲を狭めることが可能となる．

関連研究に比較子と呼ばれる同階層上で解を導き出す手法を用い，大域的に最適解を導きだすといった制約階層の研究が行われている．従来の対象は幾何制約であったものに対し本研究は時間概念のある非線形制約も含み，目的も異なるため適用可能性，有効性を示すことは意義があると考える．

1.3 本論文の構成

- 第 2 章
  本研究に関連する研究について触れる
- 第 3 章
  ハイブリッド並行制約プログラミングに関する説明を行う
- 第 4 章
  制約と制約階層に関する説明を行う
- 第 5 章
  HCC へ適用した制約階層の概念の定義とそれに基づく有効性実験
- 第 6 章
  階層化を導入した HCC の処理アルゴリズムの設計
- 第 7 章
  実装と問題点について触れる
- 第 8 章
  まとめと今後の課題について述べる

第2章　関連研究

この章では，制約の階層化に関連する研究とプログラムミングにおける生産性を向上させる研究を紹介する．

2.1　制約に基づく描画ツールの階層化の研究

制約を階層化した研究の中で特に UI の簡易化に成功した階層化の研究を紹介する．論文参照 [6],[5]

2.1.1　Grifon における制約の階層化

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と幾何制約によってアニメーションを表現する．

この研究では、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している．ハイブリッド並行制約データに優先度を付随させ，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間の優先度付けを実現した．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している．また、Grifon において制約の優先度を付随させることで，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ている．

2.1.2　ユーザインタフェースのための線形等式・不等式制約解消系

ユーザインタフェースを応用対象とし，線形等式および不等式制約からなる制約階層を処理するための制約解消系を構築している．その制約解消の実現は，HiRise 制約解消系における等式制約処理法に対して，新たに考案した不等式制約処理法を組み合わせることで行っている．そして実験により，この制約解消系が，制約が 1,000 個を超える状況でも UI を実現するのに十分に高速な制約解消ができることを示している．

2.2 プログラムミングにおける生産性の研究

プログラムの可読性とデバグに関する研究を紹介する．論文参照 [4],[7]

2.2.1 可読性を重視したプログラムの生成に関する研究

プログラムを人間とコンピュータとの間のインターフェイス (双方向の意志伝達手段) と位置付け, その際に重要となってくることの一つとして,”可読性” という概念を取り上げている. そして, 物理分野の数値シミュレーションコード (有限要素法解析) 生成支援システムを作成．プログラム生成支援システムが可読性を重視したプログラムを生成することによって, 信頼性の高いプログラムを構築することを可能としている．

2.2.2 ハイブリッド並行制約プログラミングにおける制約エラー説明機能の設計と実装

この研究ではプログラム実行中に制約エラーが発生した場合，その原因の解明に有用とされる制約の遷移，詳細をユーザーに提供する制約エラー説明機能の設計と実装を行っている．以前の処理系では制約の矛盾が発生した時点で計算は中断され，その矛盾の原因となる制約や変数の詳細を知ることはできなかった．実装は制約エラーを検出したときにエラーが生じた point phase 内で Tell された制約の記録，同じ phase で上書きされた変数名と値の出力，そして制約ストアの最終状態を出力する機能を備えた制約説明モードを追加している．以前の処理系はエラーの原因に結びつく情報が得られなかったのに対し，実装後は上書きされた変数や今まで Tell された制約の遷移を追うことでエラーの原因をみつけることが以前よりも容易にしている．

2.3 本研究の新規性

従来の制約階層の研究としては，ユーザによる記述に多少の誤りを含んでいたとしても最適解を導き出すものが主であったのに対し，本研究ではプログラミング言語の記法を制約階層に基づいて変換し明示的に階層構造を記述可能にすることで，コードの可読性を高め，予期せぬエラーの発生する場合を減少させる．そのために制約を階層化することの有効性を示したものである．これはユーザの記述の誤りに対し，期待しない結果を返す誤りであるならば，解を補正するのではなくエラーとして出力し，記述の誤りの原因を限定させるという点が大きく異なり，新規性がある．

第3章　ハイブリッド並行制約プログラミング

3.1　HCCの概要

HCCとはHybrid Concurrent Constraint programing(ハイブリッド並行制約プログラミング) のことで，時間軸に沿った表現ができるようデフォルト並行制約プログラミングを拡張した枠組みである．

HCCの基となるデフォルト並行制約プログラミング (default cc) は， 並行制約プログラミングに否定情報を扱うための機能を追加した枠組みである．

つまり，HCCは並行制約プログラミングと否定情報を扱う機能と時間軸を扱う機能を併せ持つ．これにより，各時点で実行されるような離散的変化とある時点とその次の時点までの間で実行される連続的変化を同時に記述できるシステムをもつ言語となっている．

具体的にはx'のような構文を用いることにより，変数の時間による微分係数を記述することができる．これの良いところは，c言語で同様のことを表現しようとしたら時間のための変数tなどを宣言し，それを何度も組み込まなければならない手間を省いてくれている．つまり，HCCでは直感的かつ少ない作業工程で記述し，それによってアニメーションデータサンプルを取れる．HCCと他言語のコードの行数比較した研究がなされている．

それに加えて，計算精度の高さがある．数値計算はデフォルトでは適応刻み幅４次５次ルンゲクッタ法が取り入れられており，それについてHCCプログラムに何か特別に記述することなしに，ルンゲクッタ法の精度が得られる．また，デフォルトの適応刻み幅４次５次ルンゲクッタ法以外にオプションによってオイラー法，４次ルンゲクッタ法，リチャードソンの補外法を使ったベアストウ法などの数値計算法を適宜選択もできるようになっている．このようなアルゴリズムの性能評価では参考文献 [8] を参照させてもらった．

HCCが高精度数値計算可能でかつ微分方程式を計算するアニメーションに対して直感的記述でかけることの有用性を実験した一例を紹介する．

初期条件の微小な誤差が時間と共に指数関数的に成長し続けることを，初期値に関する鋭敏な依存性 (SEDIC : Sensitive Dependence on Initial Conditions) という． SEDICを示す系の振る舞いをカオス（Chaos）という．つまり，数値計算が高精度でなくてはならない．そして，常微分方程式を用いる．

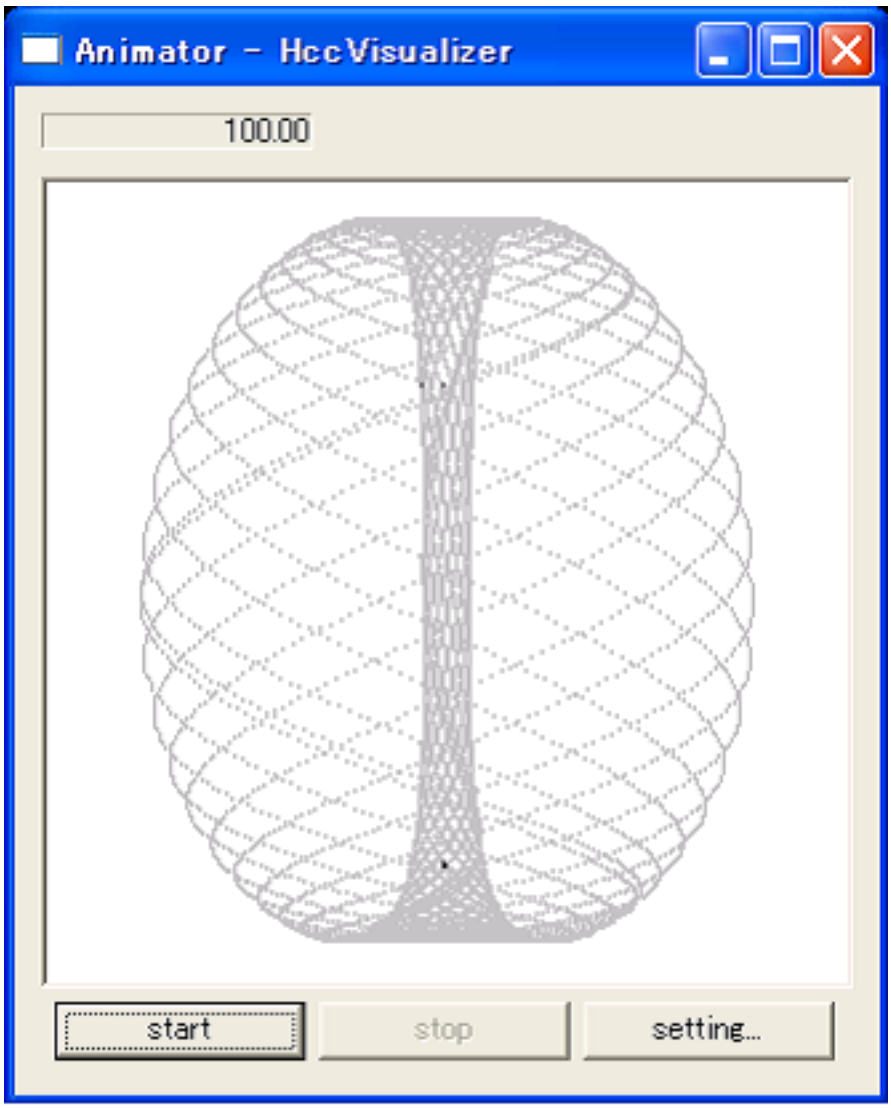


図 3.1: ラングフォード方程式

HCC では上図 2.1 のようにサンプルできる．(描画は HccVisualizer によるもの)
ラングフォード方程式はカオスの一種で，以下の 3 つの常微分方程式で表される．

$$x' = (z - b) * x - d * y$$
$$y' = d * x + (z - b) * y$$
$$z' = c + a * z - z^3/3 - (x^2 + y^2) * (1 + e * z) + f * z * x^3$$

(a,b,c,d,e,f は定数である)

そして，どのくらい直感的に記述できているか次頁にそのソースコードを示す．

```
#SAMPLE_INTERVAL_MAX 0.01
/*クラス指定*/
Ball =(initx, inity, initz, a, b, c, d, e, f)[x, y, z]{
   x=initx, y=inity, z=initz,//初期値
   always{
      cont(x), cont(y), cont(z),//連続量宣言
/*メインとなる常微分方程式*/
      x'=(z-b)*x-d*y,
      y'=d*x+(z-b)*y,
      z'=c+a*z-z^3/3-(x^2+y^2)*(1+e*z)+f*z*x^3
   }
},
/*クラスのインスタンス*/
Ball(B, 0.9, 0.1, 1.7, 1, 0.7, 0.6, 3.5, 0.25, 0),
sample(B.x, B.y, B.z)//サンプルを取るための構文
```

図 3.2: ラングフォード方程式を計算する HCC ソースコード

常微分方程式に初期値を設定しただけで計算が可能であることがわかる．これを java で同様の記述をすると，計算精度が悪いため解が計算できず途中で実行が止まってしまう．

3.2 計算モデル

3.2.1 並行制約プログラミング

ハイブリッド並行制約プログラミングの大本となっている並行制約プログラミングについて先に説明する．
並行制約プログラミングは，制約ストアとエージェント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．

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 種類ある．tell エージェントは制約ストアに制約を加える操作を行う．ask エージェントは制約ストアにおいて制約が満たされるかどうかを調べる操作を行う．

具体的に $x = 1$，$x > 0$ の 2 つの制約だけがあったとすると，
1．tell エージェントにより任意に制約がストアに加えられる．
2．tell エージェントによりもうの片方の制約がストアに加えられる．

この時点でストアに $x = 1$，$x > 0$ が加えられたことになる．
３．ask エージェントにより，２つの制約が互いに満たすことが検査される．

3.2.2 HCC の計算モデル

HCC の計算モデルは前節の並行制約プログラミングに否定情報 (デフォルトルール) を適用したデフォルト並行制約プログラミングに，時間の概念を加えるため，各時点においてデフォルト制約プログラミングを実行できるようになっている．

(１) 点計算 (point phase)
デフォルト制約プログラミングで実行される内容は，制約を制約ストアに加える (tell エージェント)，制約ストアにある制約の検査 (ask エージェント)，否定情報の実行である．これが満たされた状態になると区間計算（２）を開始する．
(２) 区間計算 (interval phase)
ここで制約ストアに連続的な計算をするプログラムが記述されていなければ，Queue empty となって計算が終了する．
逆に連続的な計算をするプログラムの記述がある場合は，その記述された制約を実行することで区間計算はなされる．連続的な制約を実行する場合に，微分方程式の積分が用いられる．そして，制約の状態が変わる，つまり連続的な制約を満たす条件を満たさなくなると積分が終了する．その後，新たに各時点での計算（１）に戻る．

この計算アルゴリズムを１ step ごとに見ていく．
point phase と interval phase のアルゴリズムはほぼ同じであるが，interval phase の積分計算部分は除く
Γ　:HCC プログラムの各部分の集合
σ　: 制約ストア
$next$: 次の phase で実行されるプログラムの集合
$else$: 否定情報の集合
とし，次の１から６の step に向かって順に進んでいくことを繰り返す．

１．現時点での $< \Gamma, \sigma, next, else >$ に対して，Γ にあるプログラムの各部分を解釈して $\sigma, next, else$ などに入れる作業を可能なところまで実行して２へ
２．σ に矛盾が生じたら中断する．(constraint error)
矛盾がない場合は３へ
３．$else$ が空なら１に戻る．空でないなら４へ
４．$else$ が空でないなら，$else$ の中から１つの else 文 (if c else A) をと

ってくる.
σ から c が真であると言えるなら 3 に戻る.
σ から c が真ではないなら 5 へ

5．Γ に (if c else A) の A を入れ，その状態でインタプリタを実行.
その結果が計算された，かつまだ σ から c が真ではないことが
言えるなら 1 へ戻る.
そうではない場合は 6 へ

6．バックトラックして Γ に A を入れる前の状態で，インタプリタを実行.
その結果が計算された，かつ σ から c が真であることが言えるなら
1 へ戻る.
そうではない場合は中断する．(default error)

interval phase における積分計算

HCC では導関数を積分する際，デフォルトでは適応刻み幅 (adaptive step-size) を用いた 4 次 5 次ルンゲクッタ法を用いている.
積分の初期制約は，最も直前の point phase での制約ストアにある制約が与えられる.
次ページの擬似コードは簡略化した 4 次ルンゲクッタ法を示す.
擬似というのは，ソースを言葉で説明している部分があり，積分と伝播との間の相互作用を説明する error checking の部分が省いてある.
詳しくは参考文献 [2] を参照.

```
integrate(diff_eqs,check_list) {
   let h be the initial step size(0.1);
   let V be the dependent variables of diff_eqs;
   allocate a choicepoint;
   set the initial values of V by the store from the point phase
   propagate;
next:
   integrate V;
   if a constraint in check_list is overshot, bracktrack and shrink h;
   if a constraint in check_list changes state, stop;
   go to next;
}

integrate V{ //4th order Runge-Kutta with no error-control
   compute k1 from current x' (for each x in V);
   for i in [2,4] {
  undo trail;
  update x (for each x in V) to compute ki;
  propagate;
  compute ki for current x' (for each x in V);
 }
 undo trail;
 set new x = old x + 1/6*k1 + 1/3*k2 + 1/3*k3 + 1/6*k4;
 propagate;
}
```

図 3.3: 擬似コードによる積分計算手順の概観

HCC は基本的なルンゲクッタ法から 3 点変更している.

1．完全に区間計算を用いる

これにより，特定の初期値が必要なくなり，システムがより柔軟に対応できる．しかし，値が発散して正確性を欠くという欠点を補う改良はまだ考えられてない.

2．積分の各刻み幅で伝播を利用

どんな形の連立方程式がきても解けるようにするため.

3．積分の始まる初期状態から一つでも制約が変化したと検出されたら，すぐに積分を中断する.

積分の終点となる point phase を break point とし，その積分の終点となるべき break point を見つけるためにある．初期状態から一つでも制約が変化した時を一つの刻み幅 (step-size) とし，それをすべて記憶する．各 step-size を記憶するのは，もし break point を飛び越えて step-size を取ってしまった時に，今の実行を無効にして最も新しい記憶にバックトラックして，今実行した step-size よりも小さくして極限を求めるように break point を定めていく.

その際，誤差 e^{-4} 内にはいるとそこを終わりの point phase，つまり break point にする．

3.2.3 HCC で主に使用される構文

４章の説明で実際に HCC プログラム用いるため，その時用いる主な HCC 構文と制約を簡単に紹介する．詳しくは参考文献 [3],[10] を参照してもらいたい．

表 3.1: HCC 制御構文

制御構文	簡単な説明
c	制約 c を制約ストアに加える
A,B	”,”で分けられた文の集合は並行して動作するものとして認識される．
if c then A	制約 c が真なら A を実行
unless c then A	制約 c が真でない (c が偽または unknown) なら A を実行
hence A	初期状態以外で A を常に実行
always A	A を常に実行 (A,hence A と同じ意味)
sample($x1, \cdots, xn$)	$x1, \cdots, xn$ のサンプルを取るよう設定する．

HCC の実行時操作　SAMPLE_INTERVAL_MAX

sample($x1, \cdots, xn$) でサンプルをとる場合に，サンプル間の時間差の最大値を決めるためのパラメータである．サンプル間隔の大きな飛びがなくなり，アニメーションのデータを取る時に効果的である．具体的に説明すると，RKQC の刻み幅は break point を定める時でなければ，１ step 前の刻み幅よりも大きくしていくので長い時間積分を計算する場合に，サンプル間隔が広がりすぎてアニメーションのデータとしては不十分になるということである．

連続制約 cont()

cont(x) は x が連続的であることを意味する．構文 always や hence など共に用いることが多く，always cont(x) で x がいつも連続であることを宣言している．

非算術制約 prev()

非算術制約は連続的には値が変わらない．prev(x) は point phase で用いられ，その point phase に入る直前の interval phase の x の値を与える．

第4章　制約階層

本章では，制約階層の概念といくつかのアルゴリズムをあげる．以下は参考文献 [1],[5] に従う．

4.1　制約の種類

Hard Constraint
硬い制約は必ず満たす制約であり，必須制約とも呼ばる．通常，必須制約の強さは required として表される．これらの制約間に矛盾があると解なしとなる．必須制約のみで制約過多な状況になるモデルをプログラミングする際，矛盾を避けることが困難である．

Soft Constraint
軟らかい制約はできるだけ満たすべき制約であり，選好制約とも呼ばる．選好制約の強さは，強いものから順に strong, medium, weak として表される．

4.2　制約階層

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い必須制約であり，それ以外の軟らかい制約は選好制約である．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ことが知られている．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．

以下は制約階層の考え方の簡単な一例である．

```
例：required x+y=z
    strong x=2
    medium x=4, y=3
    weak y=1, z=1
解は x=2, y=3, z=5
```

必須制約である $x + y = z$ は必ず満たされる必要があるので常に考慮されるが，上記の例では必須制約だけではどの変数も値を定める（値の範囲を狭める）ことはできない．次に選好制約の中で最も強い strong の $x = 2$ を考える．同階層の制約に矛盾となる制約がないことと必須制約にも矛盾しないため $x = 2$ は確定する．同様に選好制約の弱いほうの制約を考える．medium の強さの選好制約では $x = 4$ があるが strong の選好制約によって，この $x = 4$ という選好制約は破棄される．$y = 3$ についてはこの時点で確定する．もちろん必須制約に矛盾はない．また，必須制約により変数 z の値が定まる $(z = 2 + 3 = 5)$. 同様に選好制約の weak を考える．変数は y と z しかなく，どちらもより強い制約によって値を確定されているので破棄される．最終的に解は $x = 2, y = 3, z = 5$ となる．

4.3 制約階層の方式

ここでは，Borning らが提案した制約階層の定式化の概要を紹介する．
制約階層 H 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 $H = (H_0, H_1, ..., H_n)$ であり，その各レベルの H_k は，強さ k の制約を m_k 個含むベクトル $H_k = (c_{k,1}, c_{k,2}, ..., c_{k,m_k})$ である．強さは 0,1,...,n の順で優先度が低くなり，強さ 0 の制約は必須制約で，それ以外は選好制約である．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 0, 1, 2, 3 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，制約階層に従って 2 つの変数値ベクトル , の解としての「良さ」を比べる述語であり， $better(v, v', H)$ が真であることは，制約階層 H を充足する度合において 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満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．

制約階層 H の解としては，必須制約 H_0 を満たす変数値ベクトルの中から，比較子 *better* の意味で，より良いものが存在しないベクトルが選ばれる．形式的には, H の解集合 $S(H)$ は次のように定められる．

$$S(H) \equiv \{v' \in S_0(H) \vdash \exists v(v \in S_0(H) \vee better(v, v', H))\}$$

ただし，$S_0(H)$ は，全ての必須制約 H_0 を満たす変数値ベクトルの集合である．

$$S_0(H) \equiv \{v | \forall i holds(c_{0,i}, v)\}$$

上の解の定義では,「最も良い」変数値ベクトルではなく,「より良いものが存在しない」変数値ベクトルを選ぶようにしている．このような定義の理由は，比較子の種類によっては，変数値ベクトルを比較できない ($better(v, v', H)$ でも $better(v', v, H)$ でもない) 場合があるために，最も良い変数値ベクトルが存在しないことがあるからである．

4.4 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．

```
required   x1 = x2
strong   x2 + 1 = x3
weak   x1 = 0
weak   x3 = 3
```

最初に、required レベルの制約を充たすよう，$S0(Z) = (v1, v2, v3)|v1 = v2$ 次に，$S0(Z)$ 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，$S0(Z)$ 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，$(v1 - 0)2 + (v3 - 3)2 = 4$ 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、S0(Z) のいかなる要素も (1, 1, 2) より良くはない (実際に，$v1 = v2$ かつ $v2 + 1 = v3$ という条件のもとで，$v2\ 1 + (v3 - 3)2$ 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．
$S(Z) = (1, 1, 2)$
同じ強さの制約間の矛盾を解消するための基準 Least-squares-better（誤差の 2 乗の和の最小化）Weighted-sum-better（誤差の総和の最小化）Locally-predicate-better（制約が任意順の誤差の最小化）

4.5 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．

4.5.1 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．24 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．

4.5.2 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．

4.5.3 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．

$$
\begin{gathered}
minimize wstrong f(\varepsilon\ 1) + wweak f(\varepsilon\ 2) + wweak f(\varepsilon\ 3) \\
subject to x1 = x2 \\
x2 + 1 = x3 + \varepsilon\ 1 \\
x1 = 0 + \varepsilon\ 2
\end{gathered}
$$

$$x3 = 3 + \varepsilon\ 3$$

ただし、wstrong、wweak 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　HCCの制約階層化

5.1　HCC処理系を制約階層化する利点

5.1.1　現状のHCC処理系の問題点

現状のHCC処理系では，ある変数に関わる制約はどの制約も必ず満たされなければならず，ユーザは１つの制約間の矛盾でさえ許されない．つまり，矛盾する制約間をユーザ自身が取り除ききらない限りプログラムは動かない．また並行言語でもあるがゆえにユーザによる制約間のチェックは制約の数の組み合わせの個数でほぼ増大する．プログラミング言語でのバグを現実的な時間内に全ては解消しきれないとすると，HCCにもある程度の柔軟性は必要である．

5.1.2　階層化の利点

記述性の多様化・デバグへの有効性

現状のプログラミング手法では，矛盾する制約をif文を用いて解消する方法が主流であったが，制約階層を用いるとより明示的に解消することが可能となる．これはデバグにおいても有効である．なぜなら，現状は時間とともに移り変わる変数で判定するif文がリダクションされているかを考慮にいれ，全制約間の矛盾のチェックをする必要があった．それを明示的に記述してある同階層間の制約のみをチェックするだけで良くなる．

フールプルーフ設計

制約の有無の観点からプログラミングを考えると，全時間において制約が無いという時間がなく，かつ矛盾する制約が２つ以上存在してはいけないというのが現状である．制約階層を用いると選好制約のweakで代表されるデフォルトとなる制約によって，全時間において制約が無いという時間は存在しなくなる．さらに，同階層の制約の整合性が保てるなら矛盾する制約がいくら存在してもかまわないことになる．

可読性の向上

また，異なるクラス間では，階層構造をなしているif文の中に用いられているifとelseを別々に記述することが難しい．そのため，片方のクラスでif文を用いると，もう片方のクラスでタグをつけunless文を用いるのが主流となっている．これも制約階層を用いることで、if文の階層構造を小さくできるので、if Aとif¬Aを使うことが容易になる．つまり，c言語でいうgoto文に近い意味合いのタグを多用することが減るのでコードの可読性が高まる．

5.2 制約階層の定義

矛盾する制約が 2 つ以上ある場合においてのみ適用される.
制約階層 H は，レベルからレベルまでの () 個のレベルからなるベクトル

$$H = (H_0, H_1, ..., H_l)$$

であり，その各レベル Hk は，強さ k の制約 C を n 個含むベクトル

$$H_k = \{c_{k,1}\ , c_{k,2}\ , ..., c_{k,n}\ \}$$

である. 強さはの順で優先度が低くなり，強さの制約は必須制約で，それ以外は選好制約である．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, , , を表す.

制約階層の解 $S(H)$ は，H_i の $\{c_{i,m}\}$ と H_j の $\{C_{j,n}\}$ が矛盾する時,
$h_i = (H_0, H_1, ..., H_i)$, $h_j = (H_0, H_1, ..., H_j)$ として

$$S(H) \equiv \{S(h_j) \mid \neg\exists c_{j,n}\,(c_{j,n} \in h_j \vee i < j)\}$$

$i = j$ の時

$$S(H) \equiv \emptyset$$

5.2.1 syntax

ContConstr、Dconstr ついては論文参照 [3]
ac は ask 制約、N は数値、x は変数、X() は関数

$$\begin{aligned}
HAgent &:== Preferencial\ Agent \mid Agent \\
Preferencial &:== \text{"}required\text{"} \mid \text{"}strong\text{"} \mid \text{"}medium\text{"} \mid \text{"}weak\text{"} \\
Agent &:== ContConstr \mid DConstr \mid Agent \mid if\ ac\ then\ Agent \\
&\quad \mid if\ ac\ then\ Agent\ else\ Agent \mid unless\ ac\ then\ Agent \\
&\quad \mid hence\ Agent \mid always\ Agent \mid do\ Agent\ watching\ ac \\
&\quad \mid do\ Agent\ hencewatching\ ac \mid do\ Agent\ until\ ac \mid do\ Agent\ henceuntil\ ac \\
&\quad \mid while\ ac\ do\ Agent \mid time\ Agent\ on\ ac \mid when\ ac\ do\ Agent \\
&\quad \mid wait\ N\ do\ Agent \mid new\ x\ in\ Agent \mid X(t1, ..., tn) \\
&\quad \mid forall\ X(Y)\ do\ Agent \mid sample(x1, ..., xn) \\
&\quad \mid evalexpr\ Variable\ =\ Expression\ in\ Agent
\end{aligned}$$

5.2.2 semantics

- c は制約
- Γ はエージェントの集合（処理系でいうパージング後のキュー）
- σ はストア
- $\sigma^\star$ は σ からピックアップした c を除いた制約の集合
  c が σ に tell されている状態（入っている状態）は，c と $\sigma^\star$ とが矛盾しない
- $next, else$ は論文参照 [2]
- P（Preference）は優先度であり，優先度の高いものほど値が低い

Tell1

$$\langle(\Gamma, c), \sigma, next, else\rangle \rightarrow \langle\Gamma, (\sigma \cup \{c\}), next, else\rangle$$

Tell2-1

$$\frac{(\sigma^\star \not\vdash \{c1 \wedge c2\}) \ \wedge \ P.c1 < P.c2}{\langle(\Gamma, c1), (\sigma^\star, c2), next, else\rangle \rightarrow \langle\Gamma, (\sigma^\star \cup \{c1\}), next, else\rangle}$$

Tell2-2

$$\frac{(\sigma^\star \not\vdash \{c1 \wedge c2\}) \ \wedge \ P.c1 > P.c2}{\langle(\Gamma, c1), (\sigma^\star, c2), next, else\rangle \rightarrow \langle\Gamma, \sigma, next, else\rangle}$$

5.3　例題を用いた制約階層化の有効性検証

5.3.1　記述性の多様化

状態遷移させるモデルであると気づけず記述できない場合に有効
昔のファイルで動作過程を忘れてしまったモデルに追加項目を加える場合にも有効である.
以下は書き換え可能な構文の現状（コードの上側）と階層化導入後（コードの下側）の例である.

```
x=0,
always{
  cont(x),
  if(x=10) then x'=-prev(x') else x'=10
}                                                    現状
-------------------------------------------------------------
x=0,
always{
  weak x'=10, cont(x),
  medium if(x=10) then x'=-prev(x')
}                                                  階層化導入後
```

if else 構文の書き換え

```
x=0,
y=123,

A=(){
  always{
    x'=1,
    y'=-3,
    if(x=y) then {z=2, T}
  }
},

B=(){
  always {unless(T) then z=1}
},                                                    現状
-------------------------------------------------------------
x=0,
y=123,

A=(){
  always medium{
    x'=1,
    y'=-3,
    if(x=y) then z=2
  }
},

B=(){
  always weak z=1
},                                                    階層化導入後
```

タグのための unless 書き換え

```
x = 10, L0(),
always L0 = () {
  do hence {
    x' = 10
  } until(x=20),
  when(x=20) do L1()
},

always L1 = () {
  do hence {
    x' = 5
  } until(x=30),
  when(x=30) do L2()
},

always L2 = () {
  do hence {
    x' = -5
  } until(x=10),
  when(x=10) do L0()
},                                                    現状
-------------------------------------------------------------------
x=10,
 always {
    weak x'=10,
    medium if(x>=20) then x'=5,
    strong when(x=30) do{
      do always x'=-5 watching(x=10)
    }
}                                                     階層化導入後
```

状態遷移モデルの書き換え

5.3.2 フールプルーフ設計

HCC 特有の連続性宣言である cont,lcont をユーザが意識することなくコーディングできるようにするため，制約階層を用いてこの点を解消する．
以下のコードは cont,lcont を考えていないため，ユーザの期待通りに動かないものである．
ユーザの期待としては，$v = 3$ を初期値として速度 -2 で進み，$v = 0$ になると速度 0 になり急停止するモデルである．

```
v=3,
hence{
  if(v=0) then v'=0 else v'=-2
}
```

制約の抜け落ちる例

上記コードを計算させると初期値以外は値が定まらない．理由は interval phase で $v = 3$ に値が引き継がれないからである．よって、lcont(v) が必要となる．

詳細に理由を説明すると，初期値となる point phase では $v = 3$ は読み込まれ、hence の中は読み込まれない．次に interval phase に入るが，$v = 3$ は読み込まれず、hence の中だけ読み込まれる．変数の値が引き継がれないので，時間的には同時刻であるが、point phase の $v = 3$ は interval phase（積分計算除く）では $v = -inf$〜inf となる。v が区間値であるから if 文の ask 条件は満たされない．これにより積分に必要な微分値を導出できる制約がなく $v' = -inf$〜inf，よって，この後の interval phase の積分計算では $v = -inf$〜inf・$v' = -inf$〜inf となる．

lcont(v) を付加したことで $v = 0$ となるところまでは値の定まる計算ができる．しかし，$v = 0$ で停止するということが表現できない．理由は point phase に $v = 0$ の値が引き継がれないからである．よって、cont(v) が必要になる．

こちらも詳細に理由を説明すると，$v = 0$ の point phase にはいる直前の interval phase では $v = 0$・$v' = -2$ である。point phase に入ると、if 文の $if(v = 0)then v' = 0 else v' = -2$ が読み込まれるが v,v' ともに値は引き継がれないため，$v = -inf$〜inf であり，この if 文はリダクションされない．従って $v' = -inf$〜inf となり以降 v,v' の値は定まらない．

次は制約が過剰にあり矛盾する例を紹介する．
ユーザの期待としてはボールを自由落下させ，床に衝突し反発係数 1 で跳ね返り，元の位置まで戻ってくるのを繰り返すモデルである．

```
y=10,
always{
  cont(y),
  y''=-9.8,
  if(y=0) then y'=-prev(y')
}                                                    現状
-------------------------------------------------------------
y=10,
always{
  cont(y),
  cont(y''),
  if(y=0) then y'=-prev(y') else y''=-9.8
}                                                  階層化導入後
```

このコードの上側は，$y = 10$ から $y = 0$ まで計算され，$y = 0$ になった point phase で $y'' = -9.8$ によって引き継がれた y' の値と $y' = -prev(y')$ が異なるため矛盾としてエラーを返す．
一方コードの下側は，$y = 0$ の point phase で何の支障もなく計算され，期待した動作をする．
ここで問題なのは下側のコードも上側同様，$y = 0$ の point phase で $y'' = -9.8$ は存在する．しかし，下側の cont(y") で引き継いだ $y'' = -9.8$ という値は y' に影響を与えない．つまり、y' の値は引き継がれず、y' に関しては $y' = -prev(y')$ の制約のみでエラーを起こさないということである．

制約階層を用いて cont,lcont を除去　上記のコードのように非常に簡単なコードでさえも，ユーザが内部処理を考えてコーディングしなくてはならないのが現状である．そこで，全変数に対し cont,lcont を制約階層の最も弱い階層に置くことで解消する．これを cont,lcont をデフォルト制約にすると言う事にする．

まず，変数がシミュレート時間内でずっと連続量であった場合 cont,lcont は記述してあると正常に動く場合はあるが異常を示す例はない．一方，変数がシミュレート時間内で離散変化する場合はどうかが重要となる．変数が連続量とりつつ，ある一時点だけ離散変化するものが一般的であるが，常に離散変化する変数もあったとする．変数を離散変化させるには必ず離散変化を起こさせる制約が必要となる．現状で cont と書いてあるとエラーとなるが，階層構造を用いると cont のほうが弱い制約になるので無視され離散変化の制約のほうが優先される．つまり、cont,lcont をデフォルト制約とすることで連続変化，離散変化ともにユーザの期待どおりに表現可能となる．そして，ユーザはもう cont,lcont よって起こる影響を考える必要はない．

cont,lcont と同様に重力加速度，磁場による力など，常にデフォルトで制約が

かかっているような状況下でミスを減らす場合にもこれらの力を弱い階層の制約として扱うことで解消可能となる．

5.3.3 可読性の向上

5.3.1 の書き換えの例のように，階層化を効率的に利用することでコードの量は確かに減る．しかし，減ったコードが必ずしも可読性がいいとは言いきれない．また，クラスごとに機能を分散させてあるコードの場合，階層化を使う場面は減り，少ししか使用されないようでは可読性に大差はない．

5.3.4 デバグへの有効性

実装の過程において，制約矛盾となる際，その２つ以上の制約を特定する必要がある．その特定された制約をユーザに示すことでデバグの効率は大きく変わる．ただし，これは階層化以前の問題であり，階層化自体はデバグ自体には無関係である．

第6章　HCCでの処理アルゴリズムの設計

6.1　現状のHCCインタプリタの処理

1. モード，シミュレーション時間を指定してプログラム実行
2. Hcc プログラムファイルの読み込み
3. プログラム解析　各エージェントをキューに格納
4. (a) point phase 処理 (b) interval phase 処理
   (a) → (b) → (a) → (b) → (a) → ··· を繰り返す．途中エラーを検出したら実行終了．各フェーズごとではキューからエージェントを一つずつ取り出してキューが空になるまで処理を続ける
5. 指定シミュレート時間になり次第，４の繰り返し実行を中断してサンプル値を出力

6.1.1　point phaseでの処理

現状の処理系では，エージェントをキューで管理している．point phase ではこのキューからエージェントを取り出し，キューが空になるまで順に処理を行う．処理はエージェントの種類で分岐し，それぞれ異なる処理が行われる．
制約ストアへの Tell 操作は HccTell() という関数で行い，制約を制約ストアに入れるエージェントが算術制約・非算術制約・cont 制約・論理制約のときに HccTell() は実行される．HccTell() 実行中に制約の矛盾である変数の上書きを検出すると制約エラー処理に移行する．エラーを返さずに計算が全て終了したら interval phase の処理に移行する．

6.1.2　interval pahseでの処理

interval phase も point phase ほぼ同様の操作を行う．異なる点は，算術制約に属する微分方程式などの連続制約を用いた積分計算が加わる．また，各変数の値

の変化を常にチェックしている機能があり、point phase となる点を見つけ次第、interval phase での計算を中断し point phase 処理に移行する.

6.2 理想とする記述法

階層構造を作ることで制約矛盾する際、if 文の数珠繋ぎで解消するという意識をすることなく，並行プログラミングできる

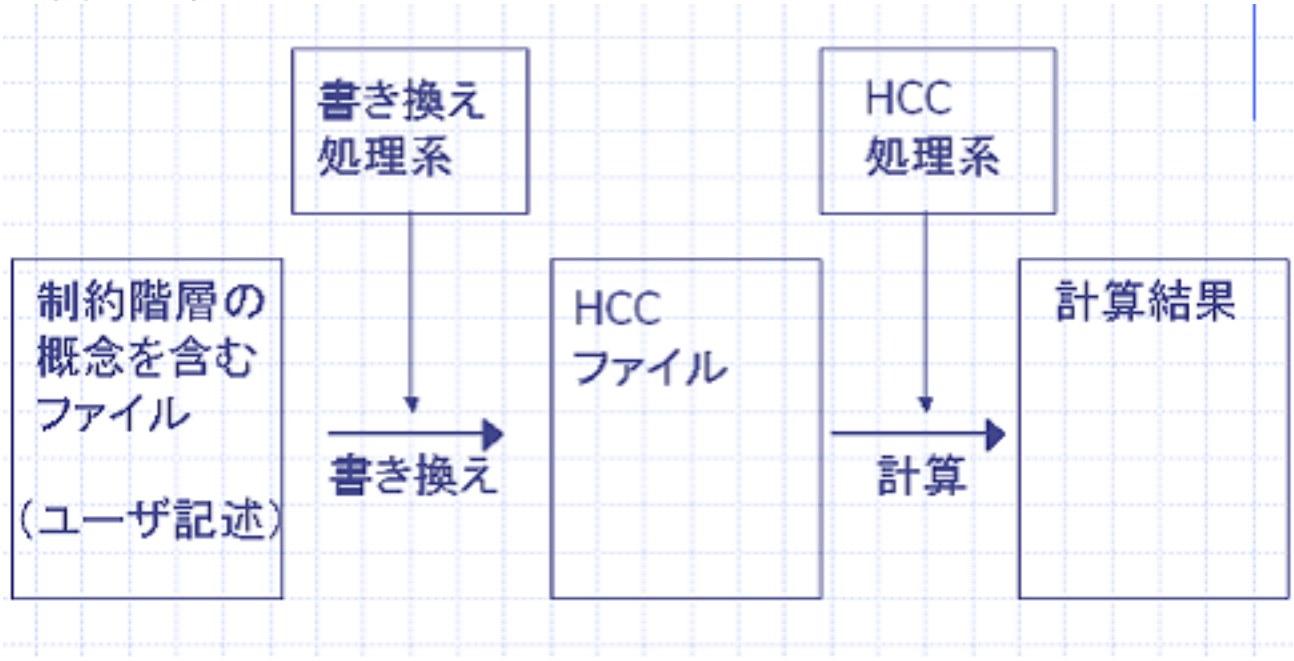
```
x=10, y=20,
y'=0, x'=1,
always{
  weak  {cont(y), cont(x), y''=-9.8, x''=0,}
  medium{ if(y=0) then y'=-0.8*prev(y'),
      if(x=20) then x'=-prev(x'),
  }
  strong{ if(y=0 && prev(y')>-0.00001) then always y'=0,
        if(x=-1) then {x'=-prev(x'), y'=20},
}
sample(x,y)
```

6.3 制約階層化の３つの手法案

6.3.1 書き換え規則アルゴリズム

ユーザに記述してもらう理想とする（制約の階層化を意識しない）ソースから，Hybrid CC で矛盾の生じないソースへ変換するプリコンパイラをつくる. 次頁に載せている評価基準に従い，規則を保ちつつ書き換える処理系を作成する.
次頁の図はプリコンパイラから HCC 処理系へと計算する処理の流れ図である.

計算の流れ概念図



階層構造間の評価基準として次を採用する.

```
if S(C1) ≠ φ then {
  if S(C1) ∩ S(C2) ≠ φ then
    S(H)=S(C1) ∩ S(C2)
  else
    S(H)=S(C1)
}
else
    S(H)=S(C2)
-------------------------------------------------------------
C1,C2 は制約
S(C) は制約 C から求まる解集合
S(H) は求める解集合
Strong C1   Weak  C2
```

上記のアルゴリズムは最もシンプルな 2 層の制約階層処理方式で，強い制約を C1, 弱い制約を C2 とすると,

- C1 が存在しないならば $H = C2$
- C1 が存在し，$C1 \cap C2$ の解が空集合にならなければ $H = C1 \cap C2$
- C1 が存在し，$C1 \cap C2$ の解が空集合になるなら $H = C1$

簡単な例題における処理の流れ

以下のコードはボールを自由落下させ，地面に衝突しては繰り返すモデルである．

現状の処理系におけるユーザの記述

```
y = 10, y' = 0,

hence {
    cont(y),

    if(y=0) then{
      y'= -0.8*prev(y')
    }else {
      y''= -9.8
    }
}
```

階層化プリコンパイラ実行前のユーザの記述

```
y = 10, y' = 0,

hence {
  cont(y),
  weak y''=-9.8,
  strong if(y=0) then y'= -0.8*prev(y')
}
```

階層化プリコンパイラ実行後のファイル

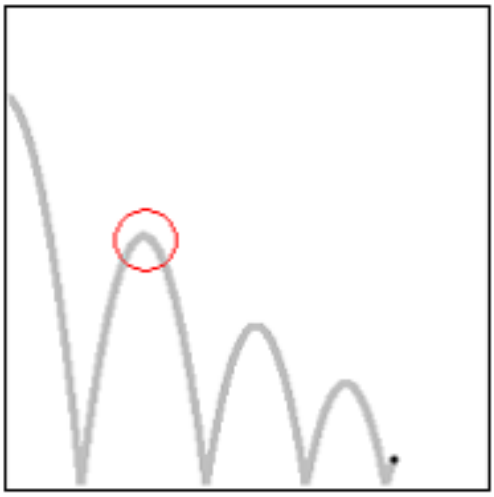
```
y = 10, y' = 0,

hence {
   cont(y),

   if(y=0) then y'=-0.8*prev(y'),

/*以下を自動的に追加*/
   if (y'=-0.8*prev(y')) then {
       if(y=0) then{
           if(y''=-9.8 && y'=-0.8*prev(y')) then {
               y''=-9.8, y'=-0.8*prev(y')
           },
           unless(y''=-9.8 && y'=-0.8*prev(y')) then{
               y'=-0.8*prev(y')
           }
       }
   },
   unless (y'=-0.8*prev(y')) then y''=-9.8
```

4行目の $if(y = 0)\ then\ y' = -0.8 * prev(y')$ は不必要に思うかもしれない．しかし，現状の HCC 処理系のままでは追加部分の if 文は解釈されいため必要となる．

問題点 1

プリコンパイラ実行後のファイルはこのままでは HCC 処理系はエラーを起こして終了してしまう．原因はソースコード最後の行の unless ($y' = -0.8$*prev(y')) then $y'' = -9.8$ である．

下図の◯の部分前までずっと unless (偽) then $y'' = -9.8$ がリダクションされ $y'' = -9.8$ という制約が有効になっていた．折り返しの部分を制約 $y'' = -9.8$ を用いて計算すると $y' = -0.8 * prev(y')$ が真となる場合に遭遇する．これは HCC 処理系ではデフォルトエラーとなる．



問題 1 の改良と実行結果

これを回避するために $y' = -0.8 * prev(y')$ の部分にタグを代わりに使用し改良を施す．この改良によって，同様な不具合はなく動くことを確認した．

問題点 2

このコードは階層化プリコンパイラを実行後のソースである.

```
y = 10, y' = 0,
x = 0, x' = 1,
always {
      cont(y),cont(x),

  if(y=0) then {A, y' = -0.8 * prev(y')},
  if(x=20) then {B, x'=-prev(x')} else {C, x''=0},

/*以下追加コード*/
  if A then {
  if(y''=-9.8 && A) then {
  y''=-9.8, y'=-0.8*prev(y')
  },
  unless(y''=-9.8 && A) then{
      y'=-0.8*prev(y')
  }
  },
  if B then {
    if(y''=-9.8 && B) then {
       y''=-9.8, x'=-prev(x')
    },
    unless(y''=-9.8 && B) then{
      x'=-prev(x')
    }
  },
  if C then {
     if(y''=-9.8 && C) then {       *
       y''=-9.8, x''=0              *
     },                             *
     unless(y''=-9.8 && C) then{
       x''=0
     }
  },
  unless (A||B||C) then y''=-9.8    **
}
```

前頁のコードは weak 制約を 2 変数にして検証を行ったものである.
この実験により，ある時間 $*$ で示した行が真とならなければいけないところを $**$ の行の条件が有効になることで妨げる結果となってしまった.

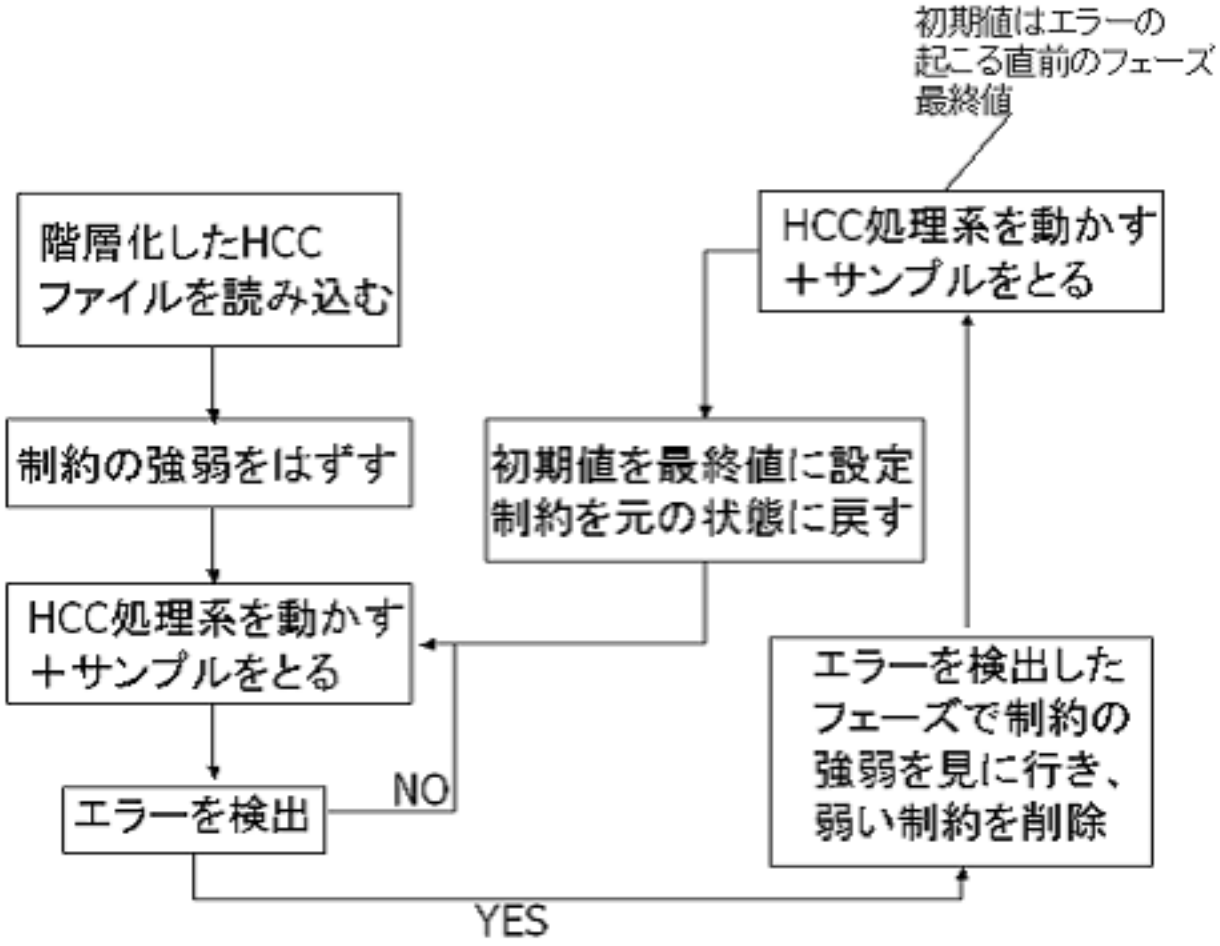
問題 2 の改良と実行結果
そこで $**$ の行の判定を変数ごとに区切って行う必要があると考え、Unless A then $y'' = -9.8$ としてユーザに weak 制約で指定する変数を記述させる方法で改良した.

考察
この書き換え規則によるプリコンパイラを作る方法での階層化は，グローバル変数をデフォルト制約として扱う場合，有効である (有効性を示すコードを付録に添付してある). しかし，デフォルト制約を時間ごとに切り替えることには不向きであり，3 階層以上の階層構造を作るには問題 2 での改良は使用できない. 例え仮に 3 階層以上にも適用可能であったとしても，指定変数が増えすぎ，ユーザの記述が煩雑となり可読性に優れる改良にはならないことが容易に推測されるためである.

6.3.2 HCC 処理系外部から操作による階層化アルゴリズム

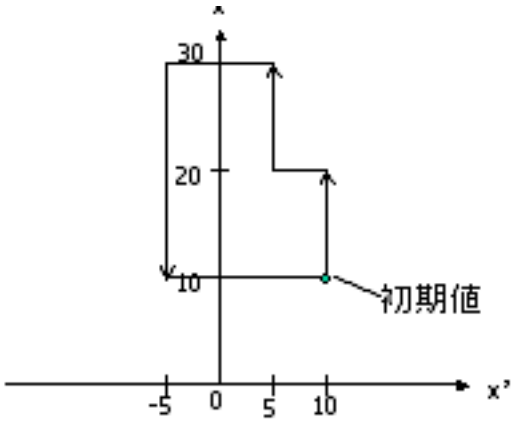
1. ユーザに記述してもらった理想とするコードを読み込む（ファイル A）
2. 制約の強弱をコードからはずしたファイル B を作成
3. ファイル B で HCC 処理系を動かしサンプルを取る（計算結果 1）
   (a) 正常にシミュレート時間が終わった場合
   計算結果を返す
   (b) HCC 処理系がエラー検出し停止した場合
      i. エラーを検出したフェーズで，衝突した制約 2 つを特定する
      ii. 2 つの制約の強弱をファイル A から識別する
      iii. 弱い制約のほうを削除したファイル C を作成
      iv. エラーを検出したフェーズのみファイル C で HCC 処理系を動かし，サンプルを取る（計算結果 2）
      v. 計算結果 2 を計算結果 1 に結合させる
      vi. 計算結果 2 の最終値を初期値としたファイル A を作成手順 3 にもどる

前頁アルゴリズムの概念図



例題を用いた考察

例１ ハイブリッドオートマトンでの状態遷移モデル
初期値 $x = 0$ から速度 $x' = 10$ で上昇し，$x = 20$ になると速度 $x' = 5$ に減速しながら上昇，$x = 30$ に到達すると $x' = -5$ で下降する．そして $x = 10$ になると初期状態に戻るモデルである
図とコードを次頁に載せる




```
x=10,
always {
   weak x'=10,
   medium if(x>20) then x'=5,
   when(x=30) do{
        do strong always x'=-5 watching(x=10)
   }
}
```

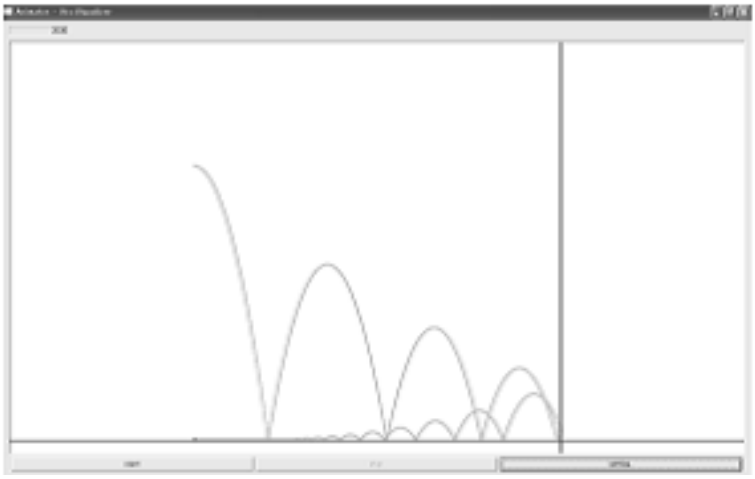
問題点 1
エラーではなく値が-inf〜inf になるため，ポイントフェーズにはいる点をエラーのみでは判断できない

例 2 多少複雑なバウンシングボールモデル $x = 10, y = 20, y' = 0, x' = 1$ を初期値とし，水平投射されたボールが壁 $x = 20$ と地面 $y = 0$ で衝突し跳ね返る．速度がなくなっていくと，地面との間で zeno 問題が起きるが，$y = 0$ かつ $y' = 0$ と検値されたら y 方向の振動を止めるようなモデルである．

```
x=10, y=20, y'=0,x'=1,
always{
  cont(y),
  cont(x),
  weak y''=-9.8,  x''=0,
  if(y=0) then{
    if(prev(y')>-0.00001) then always y'=0
      else  storng y'=-0.8*prev (y')
  },
  if(x=>20) then strong x'=-prev(x')
}
```

問題点 2

ポイントフェーズで制約エラーになり変数の値が書き換わるモデルは，ポイントフェーズから値を引き継いで新たに始めなければならないが HCC 処理系ではその設定ができない．



結論

問題点 1 に対してはフェーズごとに制約をチェックする必要がある．
問題点 2 に対しては現状の HCC 処理系では外部ファイルを用いて，ポイントフェーズから値を引き継いで新たに始めることができないので，処理系内部より値を設定する必要がある．

6.3.3 HCC 処理系内部のストア入出力操作による階層化アルゴリズム

6.3.2 の問題点（フェーズごとに制約をチェックする．ポイントフェーズから値を引き継いで新たに始めることができないこと）を踏まえて、point phase と interval phase を別々にエラーが発生するかみる．point phase と interval phase の処理はほぼ同様なので 1 つのほうのアルゴリズムを示す.
Γ はキュー，σ は制約のストア，Σ は矛盾する制約のストア、A はエージェント、P は優先度 (値が低いほうが優先度が高い)、C は制約、ERR はエラー

1. parsing
2. dequeue(A) (from Γ)
3. store(P, A)
4. tell A
5. if $\neg$ ERR then compulating, and goto 2
   else goto (a)
   (a) which constrant($C2 \in \sigma$) does last executed Agent($C1 \in A$) contradict?
   (b) switch ($a = P.C1 - P.C2$){
   case $a > 0$:
   goto 2
   case $a < 0$:
   $\Sigma \cup \{C2\}$
   continue
   case $a = 0$:
   return ERR
   }
   (c) $initialize(\sigma)$, $initialize(\Gamma)$
   (d) parsing except for Σ
   (e) goto 2

第7章　制約階層の実装

7.1　6.3.3手法での実装

1. 制約とその優先度の保持する配列をつくる
2. 制約の優先度順にキューを作リ変える (R,S,M,W) → R がなくなるまで enqueue しなおす
3. キューとストア（indexicals）の入ってる制約全てのコピーを作る
4. 変数の値も別に保持
5. タグを付け替える機能
6. TellHCC(), 衝突した制約 c1
   if (!tellHcc(c)) {
   もう一つの制約をチェックする関数 () → return c2
   switch($a = P.c1 - P.c2$){
   case $a > 0$:
   goto 2
   case $a < 0$:
   $\Sigma \cup \{C2\}$
   continue
   case $a = 0$:
   $ts- > halt = HccConstrErr$;
   }
   }
7. キュー，ストアの初期化

```
NullifyQ(&metaq); NullifyQ(&defaultQ)
restoreState(ptQ);NullifyQ(&NQ);
```

7.2 実装の際の問題点

エラー起こる起こらないに限らず以下の 2 点は必須

- タグの付け替え
- 評価の順番

```
requied x+y=3
weak x=0
weak y=2
```

yを導いた制約 R : $x+y=3, W$: $x=0 \rightarrow W^{\star}$: $y=3$
dequeue(W:y=2) →エラーとするには？
案 1 ：$R=0, W=3$ として新しいタグ $Z=1.5$ →エラーが返せない
案 2 ：$R \wedge W=W$

評価の順番
$W: x=0 \rightarrow W: y=2 (\rightarrow W: x=0 and W: y=2) \rightarrow R: x+y=3$ →エラー
$W: x=0, W: y=2$ の 2 つとも削除すると値がでない
片方だけ優先させると，評価の順番によって値が変わる

評価の順番は R,S,M,W の順，タグの付け替えは案 2 で対策

```
requied x+y=3
strong z=0
medium z=5
weak x=3
weak y=2
-------------------------------------------------------
評価順番
R→  S        →    M              →    W    →   W
     ↓             ↓                  ↓        ↓
   S:z=0        M:z=5 破棄          W:x=3    W:y=2 エラー
                                    W:y=0
```

```
requied x+y+z=3
strong z=0
medium x=2
medium y+z=5
weak y=2
-------------------------------------------------------
R→  S       →  M    →  M              →  W
    ↓          ↓    ↓
  S:z=0      M:x=2  M:y=5 エラー
             M:y=1
```

```
requied x+y+z=3
strong y+z=5
medium x+y=3
weak y=2
-------------------------------------------------------
R→    S       →       M              →    W
      ↓               ↓                   ↓
    S:x=-2          M:y=5              W:y=2 破棄
                    M:z=0
```

```
requied x+y+z=3
strong y+z=4
medium x=1
weak y=2
-------------------------------------------------------
R→    S       →       M              →    W
      ↓               ↓                   ↓
    S:x=-1          M:x=1 破棄          W:y=2
                                        W:z=2
```

第8章　まとめと今後の課題

8.1　まとめ

ハイブリッド並行制約プログラミングに制約階層を適用することにより，記述性の多様化・デバグへの有効性・フールプルーフ設計・可読性の向上を評価基準に多くの例題をもとに検証を行い有効性を示した．特にフールプルーフ設計の点において，コーディングにおいて制約過多，制約不足といった状況の解消支援をする本研究の目的に適した検証が行えたと思われる．

３通りのアルゴリズムに従い，階層化設計について各々実験を行った．その実験結果をもとに現状最良と思われる設計を考案した．設計の中の１つの項目である矛盾しあう２つの制約を見つけるというアルゴリズムはそれだけでもデバグに非常に有効である．

8.2　今後の課題

今後の課題として２点あげる．
１点目は，実装し，評価することで設計したアルゴリズムが実際に正しく挙動するかを証明することである．２点目は制約を時間ごとに強弱を入れ替えたりする操作を可能とする制約階層を実現することである．これによって，より記述の多様性が増し，多くの制約を明示的に強弱が宣言できる．それに加え，コード量を激減させうるので，本当の意味で可読性が高まる．

謝辞

この論文を書くにあたって，ご指導してくださった上田和紀教授に深く感謝いたします．そして，階層化の概念を適切に説明し，アドバイスも頂いきました細部博史氏，記号を用いた数式化を教えて下さった石井大輔氏と大野善之氏に感謝いたします．また，描画班のみなさま・描画班 OB の飛田氏には論文の内容や構成への適切なアドバイスなど多くの助言をいただき本当に助かりました．最後に上田研究室のメンバー全てに重ねて感謝をいたします．

参考文献

[1] Alan Borning, Bjorn Feldman-Benson, Molly Wilson:Constraint Hierarchies,1992, OOPSLA '87 Proceedings, pages 48-60, October, Page 27 of 31

[2] Bjorn Carlson and Vineet Gupta：Hybrid cc with interval constraints,1997

[3] Bjorn Carlson and Vineet Gupta：The hcc Programmer's Manual,1999

[4] 渋井俊昭,Boonmee Choompol,川田重夫:可読性を重視したプログラムの生成に関する研究 情報処理学会研究報告 [ハイパフォーマンスコンピューティング]Vol.95,No.118,pp.1-6 社団法人情報処理学会

[5] 細部博史:ユーザインタフェースのための線形等式・不等式制約解消系 コンピュータソフトウェア,Vol.19,No.6,pp.13-20,日本ソフトウェア科学会,2002.11.

[6] 中村好一: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 早稲田大学大学院理工学研究科 情報ネットワーク専攻,2004

[7] 笹嶋唯: ハイブリッド並行制約プログラミングにおける制約エラー説明機能の設計と実装 早稲田大学理工学部コンピュータ・ネットワーク工学科卒業論文,2006

[8] 中木量也:制約に基づく描画ツール Constrator のデータ構造と制約処理系の階層化 早稲田大学理工学部情報学科卒業論文,2002

[9] 中木量也:制約に基づく描画ツール Constrator のデータ構造と制約処理系の階層化 早稲田大学理工学部情報学科卒業論文,2002

[10] 飛田伸一: ハイブリッド並行制約プログラミングにおける多物体衝突モデルの高速化 早稲田大学理工学部情報学科卒業論文,2004