2009 年度 修士論文

ハイブリッドシステムモデリング言語 HydLaの実行アルゴリズムの提案と実装

提出日　：　2010 年 1 月 28 日
指導　　：　上田　和紀 教授

早稲田大学大学院基幹理工学研究科
情報理工学専攻
学籍番号：5108B107-0

廣瀬　賢一

概要

時間の経過に伴って状態が連続変化したり，状態や方程式自体が離散変化したりする系をハイブリッドシステムと呼ぶ．たとえば，状態を表す連続関数が時間にしたがって切り替わるモデルはハイブリッドシステムの１つであり，そのようなモデルは物理学，生命工学，制御工学等をはじめとする様々な分野に存在するため，その適用分野は広い．そのため，ハイブリッドシステムを容易に扱うことができる枠組みの作成が期待されている．

ハイブリッドシステム記述のための既存手法としてハイブリッドオートマトンや Hybrid CC などを挙げることができるが，それらはユーザがモデルの取りうる状態をすべて列挙し，記述をおこなわなければならないため，記述性に欠けるという問題が存在した．そこで，現在ハイブリッドシステムモデリング言語として制約概念と制約階層に基づくモデリング言語 HydLa の提案をおこなっている．しかし，現状では HydLa には宣言的意味論が与えられただけであり，その実行手法については議論がおこなわれていなかった．

本研究ではまず，HydLa モデルに対するシミュレーション実行の為の数式処理を用いた手法の提案をおこなう．定義の呼び出しの解決をはじめとする HydLa プログラムの変換手法，制約モジュール集合間の強弱関係を表す関係グラフを構築することによる制約階層の求解手法，制約の意味を解析するための型付け，実行時におけるポイントフェーズ・インターバルフェーズといったフェーズの分割，および各フェーズにおける処理の詳細を中心に述べていく．

さらに，その手法を実際に取り入れ作成をおこなったシミュレーション実行処理系について述べる．処理系は，各処理を部品化することで，それぞれを任意に入れ替え可能とし，現在同時に作成されている区間演算を用いた数値計算による処理系と共通処理を共有する統合処理系として作成した．各処理の抽象・部品化をおこない，任意に入れ替え可能とした設計について述べる．

最後に，HydLa を用いてモデリングをおこなうにあたって基本となる記述手法や注意すべき点，制約階層等を用いた応用的な記述手法についても考察する．

本研究により，HydLa モデルに対するシミュレーション実行のアルゴリズムが具体化された．そして，様々な求解手法を統一的に扱うことができるシミュレーション実行処理系を作成することで，HydLa プログラムを複数の手法を使用して実際に動作させることが可能となった．

目次

図目次

表目次

リスト目次

第 1 章

はじめに

1.1　研究の背景と目的

時間の経過に伴って状態が連続変化したり，状態や方程式自体が離散変化したりする系をハイブリッドシステムと呼ぶ．たとえば，状態を表す連続関数が時間にしたがって切り替わるモデルはハイブリッドシステムの 1 つであり，そのようなモデルは物理学，生命工学 [3, 19]，制御工学等 [2] をはじめとする様々な分野に存在するため，その適用分野は広い．そのため，ハイブリッドシステムを容易に扱うことができる枠組みの作成が期待されている．

ハイブリッドシステム記述のための既存手法としてハイブリッドオートマトン [13] や Hybrid CC[5, 6, 10, 11, 12] などを挙げることができるが，それらはユーザがモデルの取りうる状態をすべて列挙し，記述をおこなわなければならないため，記述性に欠けるという問題が存在した．そこで，現在ハイブリッドシステムモデリング言語として制約概念と制約階層 [4] に基づくモデリング言語 HydLa[24, 25] の提案をおこなっている．しかし，現状では HydLa には宣言的意味論が与えられただけであり，その実行手法については議論がおこなわれていなかった．

本研究ではまず，HydLa モデルに対するシミュレーション実行の為の数式処理を用いた手法の提案をおこなう．定義の呼び出しの解決をはじめとする HydLa プログラムの変換手法，制約モジュール集合間の強弱関係を表す関係グラフを構築することによる制約階層の求解手法，制約の意味を解析するための型付け，実行時におけるポイントフェーズ・インターバルフェーズといったフェーズの分割，および各フェーズにおける処理の詳細を中心に述べていく．

さらに，その手法を実際に取り入れ作成をおこなったシミュレーション実行処理系について述べる．処理系は，各処理を部品化することで，それぞれを任意に入れ替え可能とし，現在同時に作成されている区間演算を用いた数値計算による処理系と共通処理を共有

する統合処理系として作成した．各処理の抽象・部品化をおこない，任意に入れ替え可能とした設計について述べる．

最後に，HydLa を用いてモデリングをおこなうにあたって基本となる記述手法や注意すべき点，制約階層等を用いた応用的な記述手法についても考察する．

1.2　本論文の構成

以下に本論文の構成を記す．

- 2 章
  ハイブリッドシステムについて解説する．
- 3 章
  ハイブリッドシステムモデリング言語 HydLa について解説する．
- 4 章
  HydLa のシミュレーション実行をおこなうための，提案するアルゴリズムについて述べる．
- 5 章
  シミュレーション実行処理系の実装について述べる．
- 6 章
  HydLa を用いたハイブリッドシステムのモデリングについて考察する．
- 7 章
  まとめおよび今後の課題について述べる．

第 2 章

ハイブリッドシステム

本章では対象となるハイブリッドシステムについて述べる．

2.1　概要

時間の経過に伴って状態が連続変化したり，状態や方程式自体が離散変化したりする系をハイブリッドシステムと呼ぶ．たとえば，状態を表す連続関数が時間にしたがって切り替わるモデルはハイブリッドシステムの 1 つであり，そのようなモデルは物理学，生命工学，制御工学等，をはじめとする様々な分野に存在するため，その適用分野は広い．そのため，ハイブリッドシステムを容易に扱うことができる枠組みの作成が期待されている．

ハイブリッドシステムの例として図 2.1 の軌道を描く，ボールが自由落下し，地面で跳ね返るモデル (以下，BouncingParticle モデル) を挙げる．ボールが重力によって落下することで，高さと速度が連続的に移り変わる変化を “連続変化” と呼び，ボールが地面と衝突することでボールの速度が瞬間的に変化し，高さの軌道も連続的でなくなる変化を “離散変化” と呼ぶ．このような連続的変化と離散変化を交互に繰り返す系のことをハイブリッドシステムと呼ぶ.

2.2　既存のハイブリッドシステム記述手法

本節では，ハイブリッドシステムをモデリングするための既存の手法について述べる．

2.2.1　ハイブリッドオートマトン

ハイブリッドシステムをオートマトンを使用する事により，記述をおこなう手法である．ノードに連続状態を表す微分方程式（フロー制約）を記述し，エッジに離散変化が起こる条件（ガード条件）および，その際に起こる変化を記述する式（リセット制約）の記

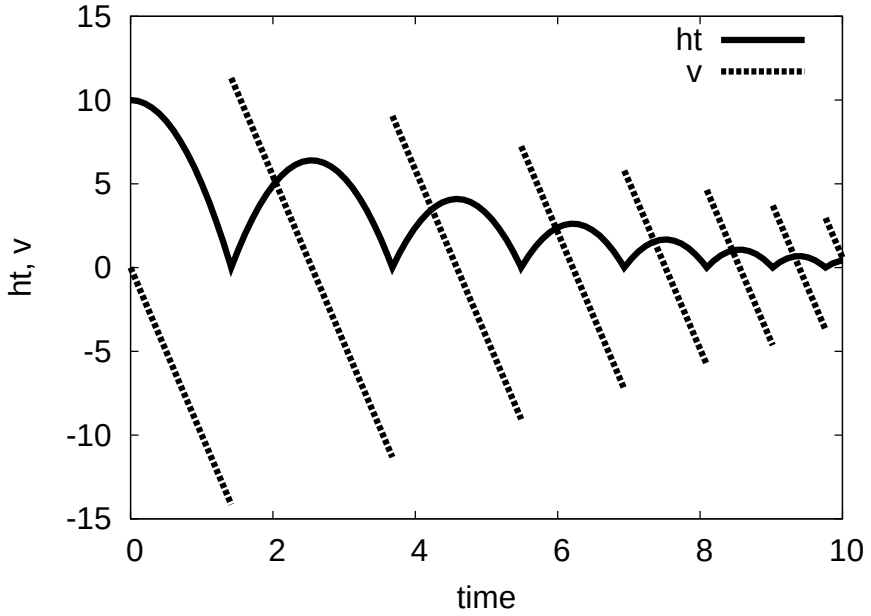

図 2.1　BouncingParticle モデル

述をおこなうことにより表現する．

ハイブリッドオートマトンを利用して検証をおこなうツールとして以下のものが挙げられる．

- PHAVer
- HyTech
- HyperTech

2.2.2　Hybrid cc

並行制約プログラミング (Concurrent Constraint Programming, cc)[21] は制約ストアに対して制約を追加する “tell” と，現在の制約ストアからある制約が導出可能か調べる “ask” を使用する事により計算を進めていくモデルである．

cc では条件が成り立っていないという否定情報を扱うことができなかった．そのため，否定情報を扱うことができる Default cc というモデルが提案された．

Hybird cc は Default cc を連続的に扱うことができるように拡張したモデルである．連続を扱うために微分方程式を記述可能とし，制約を連続的に適用可能とするための “hence” 識別子を導入するといった特徴がある．

なお，cc から派生したモデルとして他に以下のようなモデルが存在する．これらの概

念は直接は Hybrid cc の元になっていないが，Hybrid cc はこれらの概念を包含したものとなっている．

- Timed cc[20]
  cc に対してフェーズという概念を追加することで，その時点では無効となっている制約についても扱うことができるようにしたモデルである．次のフェーズに無効となっていた制約を引く継ぐことが可能なため，より柔軟なモデリングが可能となっている．
- Timed Default cc[22]
  Timed cc に対して各フェーズに対して制約を伝播させる機能を追加したモデルである．

リスト 2.1 は，BouncingParticle モデルを Hybrid cc を用いて記述をおこなったものである．y は高さ，y′ は速さ，y″ は加速度を表す．このモデルでは高さが 0 になった（地面と衝突した）場合，反発係数（0.8）で跳ね返り，衝突していなかった場合は重力加速度（10）で落下すると記述されている．

このように，直観的な記述化可能となっているが，**??**章で示す通り取りうるすべての状態を列挙・記述しなければならないため，複雑なモデルを扱うには向かないという欠点がある．

リスト 2.1　Hybrid cc による，BouncingParticle モデル

```
1  #define g 10
2  #define e 0.8
3
4  y  = 10,
5  y' = 0,
6
7  hence {
8      cont(y),
9
10     if(y=0) then y' = -e * prev(y')
11     else y'' = -g
12 },
13
14 sample(y)
```

第 3 章

ハイブリッドシステムモデリング言語 HydLa

本章では本研究において対象となるハイブリッドシステムモデリング言語 HydLa について述べる．この章の内容は主に文献 [24, 25, 28, 27] に基づいて再構成したものである．

3.1 概要

HydLa は 2008 年よりハイブリッドシステムをモデリングするための言語として開発されている．その特徴として以下のものが挙げられる．

1. 宣言型
   論理式を用いて系を記述する．数学や論理学の記法を最大限利用することで，新たに習得べき概念や記法を最小限にすることを目指している．
2. 制約ベース
   プログラム中のすべての変数は “時刻の関数” となっている．それらに関して等式や不等式，微分方程式からなる制約式を記述することにより，直観的なモデリングが可能となることを目指している．
3. 制約階層の採用
   制約感に優先度を設けることで，互いに矛盾しうる複数の制約を記述することを可能としている．それによって簡潔なモデリングを可能とすることを目標としている．

このように，与えられた問題における数学的定義をできるだけそのまま記述し，実行や解析をおこなうことを目標としている．プログラミングを専門としない技術者でも利用できる事を目指している点からはモデリング言語と呼ぶことがふさわしいが，論理学における

用語法との混乱を避けるために，HydLa で記述したものはモデルではなくプログラムと呼ぶ．

3.2　記述例

2 章で扱った BouncingParticle モデルを HydLa を用いて記述をおこなった．結果は，リスト 4.3 である．

関数変数 ht と v を用い，ht はボールの高さ，v はボールの移動速度を表している．1–3 行目において INIT，FALL，BOUNCE という 3 つの制約を定義している．INIT は関数 ht と v の時刻 0 における初期値を定義している．FALL および BOUNCE は，always 演算子 [] により時刻 0 以降に成り立つべき制約であることを示している．FALL ではボールの連続変化 (自由落下) を表す微分方程式を記述し，BOUNCE ではボールが床に到達した際の v の離散変化 (ボールの反発) を記述している．4 行目で制約モジュールの合成をおこなっている．FALL と BOUNCE について，両者が矛盾した場合は BOUNCE を優先的に採用するように合成し，さらに優先順序をつけずに INIT を合成している．

リスト 3.1　HydLa による，BouncingParticle モデル

```
1 INIT   <=> ht = 10 /\ v = 0.
2 FALL   <=> [](ht' = v /\ v'  = -10).
3 BOUNCE <=> [](ht- = 0 => v = -(4/5)*v-).
4 INIT, FALL << BOUNCE.
```

3.3　基本構文

HydLa の構文を以下に示す．それぞれの意味については後述する．

$$
\begin{aligned}
\text{(hydla program)} \quad & H ::= (D. \mid P.)^* \\
\text{(definition)} \quad & D ::= Pname(\overrightarrow{X})\{P\} \mid Cname(\overrightarrow{X}) \texttt{<=>} C \\
\text{(program)} \quad & P ::= MS \\
\text{(module set)} \quad & MS ::= M \mid MS, MS \mid MS \texttt{<<} MS \mid Pname(\overrightarrow{E}) \\
\text{(module)} \quad & M ::= C \\
\text{(constraint)} \quad & C ::= E \; relop \; E \mid Cname(\overrightarrow{E}) \\
& \quad \mid C \texttt{/\textbackslash} C \mid C \texttt{\textbackslash/} C \mid C \texttt{=>} C \mid \texttt{[]}C \\
\text{(expression)} \quad & E ::= \text{通常の式} \mid E\texttt{'} \mid E\texttt{-}
\end{aligned}
$$

3.3.1　名前

HydLa において扱う名前は複数存在する．

関数変数　関数を領域とする変数を表す，関数変数の領域は，時刻 $t \geq 0$ で定義された実数値関数の集合であり，その具体系が満たすべき制約条件を HydLa プログラムが与える，ただし，HydLa では制約は時相論理式で与えるため，時刻を引数として明示的に与えることはできない．

定義名　プログラムに命名するものと制約に命名するものの二種類がある．定義名は引数を取ることができる，制約を表す定義名は，最終引数として暗黙の時刻引数を持つ．

束縛変数　述語定義内に現れる束縛変数を表す．

数値定数　その値を常に返す定数関数とみなされる．

3.3.2　プログラム

HydLa プログラムは一般に複数の定義およびプログラムからなる．また，プログラムは制約モジュール集合から構成される．

3.3.3　制約モジュール

制約モジュール集合は制約モジュールから構成され，制約モジュール集合に対しては，“<<” および “,” の 2 種類の演算が定義されている．“MS_1 , MS_2” は優先順序をつけない合成であり，“MS_1 << MS_2” は制約モジュール集合 MS_2 に MS_1 よりも高い優先度を与える合成である．なお，“<<” は “,” よりも高い結合優先度を持つ．実行時にどのように制約モジュール集合を採用するかは第**??**節を参照のこと．

3.3.4　定義

定義は，制約およびプログラムに対する命名となる．

$\overrightarrow{X}$ は束縛変数の列である．定義は必ず成り立つべきものであり，優先度付与の対象とはならない．

プログラム定義

プログラム定義は，定義内で制約階層を扱うことができるという特徴をもつ．

プログラムのレベルで使用することは可能であるが，制約のレベルで使用することはで

きない．そのため，P1，P_2，P_3 をそれぞれプログラム定義としたとき，

```
P1, P2 << P3.
```

のような使い方はできるが，以下の 2 行のような使用はそれぞれ不可能である

```
P1 /\ P2.
P3 => y=2.
```

制約定義

制約定義は，定義内で制約階層を扱うことはできないが，他の制約内から参照することが可能であるという特徴を持つ．

そのため，

```
C1 \/ C2 => x=1.
[](C4 /\ y=2).
```

といった使用が可能である．

3.3.5　制約

HydLa の制約は，関係式，制約の呼び出し，連言，宣言，含意，[](always) からなる時相論理式である．

E *relop* E の形の制約を原子制約と呼ぶ (*relop* $\in$ {=, !=, <, <=, >, >=})．原始制約のうち，C => C の形の制約の左辺に現れるもの以外を tell 制約と呼ぶ．

C => C の形の制約は含意制約 (ask 制約) と呼び，左辺の制約式が他のその時点で有効となっている tell 制約の集合から導出可能であった場合，右辺の制約式の解釈をおこなう．

tell 制約は連節で結ぶことができないが含意条件内の制約同士は連節で結ぶことは可能である．

3.3.6　式

HydLa では，制約システムをパラメタとして考え，記述可能な制約式のクラスをとくに定めない．ただし連続・離散変化の記述に必須となる時間微分 E', および左極限 E- をとるための演算子を用意している．

また式の中では時刻 t への明示的な言及はおこなわない．たとえば式 x+y+1 の意味は

$x(t) + y(t)+1$，x' の意味は $dx(t)/dt$ となる．

3.4　HydLa の制約階層

本節では，HydLa の特徴である制約階層について述べる．

3.4.1　制約階層

制約階層は，制約間に優先順位を設けることによって，記述性向上をはかる概念である．制約階層においては全順序階層が用いられることが多いが，HydLa では半順序階層を採用している．半順序を用いた制約階層に関する研究は文献 [**?**, **?**] 等が存在するが，HydLa では制約モジュールを採用する際の部分集合が upward closed となるように制約を採用する点が既存の制約階層と異なる．

一般に，半順序集合 $(S_0, \leq)$ の部分集合 S が upward closed とは

$$\forall x \forall y [x \leq y \wedge x \in S \Rightarrow y \in S]$$

を満たすことであり，HydLa の $\ll$ は強半順序 $<$ $(\lneq)$ に対応する．

3.4.2　制約の採用の可否の判断手法

制約の採用・不採用の選択は，制約モジュール単位で，かつ各時刻ごとにおこなわれる．具体的には，プログラムの実行が進んで時刻 t の直前までの実数値関数が求まっているとき，時刻 t における関数値は，制約モジュールの upward closed な部分集合のうちその時刻において無矛盾なもののなかで，集合の包含関係に関して極大なものを満たすように決定する．

制約部分集合が時刻 t において無矛盾かどうかは，時刻 t の直前までの実数値関数の形にも依存する．また，極大集合が一意に決まるとは限らないことと，極大無矛盾集合が与える制約条件が十分強いとは限らないことから，実数値関数の値が一意的に求まるとは限らない．複数の解を許容するのは制約プログラミング共通の特徴である．

3.5　HydLa プログラムの解軌道

HydLa プログラムの (解) 軌道とは，HydLa プログラム中の関数変数の値を具体化した実ベクトル関数 $\mathbb{R} \to \mathbb{R}^n$($n$ は変数の数) である．図**??**に跳ね返るボールの軌道を示している．図の時間軸を区分けして幅 0 の閉区間と開区間の列を考える $(t_1 = 0, t_2, t_3 \in \mathbb{R})$.

$$[t_1, t_1], (t_1, t_2), [t_2, t_2], (t_2, t_3), [t_3, t_3], \ldots$$

連続変化 (例: 空中でのボールの移動) を，となりあう閉区間と開区間を結合した区間上の連続微分可能な関数 $\vec{x}_i : [t_i, t_{i+1}) \to \mathbb{R}^n$ とする ($i \in \mathbb{N}$)．離散変化 (例: 床での速度変化) を，各閉区間の時刻における，左側の関数の左極限値からその時刻の値 (右側の関数の初期値) への変換 $\vec{x}_{i-1}(t_i)_- \mapsto \vec{x}_i(t_i)$ とする．軌道とは，時間軸上に並んだ連続変化と離散変化の列である (時刻 0 における初期値の宣言も離散変化とみなす)．上記の連続変化と離散変化は，プログラム中の制約を満たすように具体化される．その際，各時間区間について**??**節で述べる方法により，モデル中の制約モジュールから無矛盾極大なものを選択する．

第 4 章

HydLa のシミュレーション実行の為の手法の提案

本章では，HydLa プログラムのシミュレーション実行を行うための手法を提案する．

4.1 シミュレーション手法の流れ

HydLa をシミュレーション実行する為の手法について述べる．

1. 入力されたプログラムを解析し，プログラム内に存在する制約定義やプログラム定義の展開をおこなうことによって，呼び出しが存在しないプログラムに変換する．
   プログラムを解析するための詳細な構文定義については 4.2 節で，展開手法については 4.3 節で述べる．
2. 呼び出しが存在しないプログラムから，解候補モジュール集合の集合を導出する事で，制約階層が存在しない HydLa プログラムの集合を作成する．
   解候補モジュール集合の集合の導出手法は 4.4 で述べる．
3. 解候補モジュール集合ごとにシミュレーション実行をおこない，無矛盾であったモジュール集合のうち，極大元であったもの（無矛盾極大モジュール集合）の解を実行結果の解とする．
   無矛盾極大モジュール集合の導出手法は**??**節で，解候補モジュール集合に対するシミュレーション手法は 4.6 節で述べる．

4.1.1　流れの例

リスト 4.3 の BouncingParticle モデルのプログラムについて考える．このプログラムにおける制約呼び出しに関する記述を展開するとリスト 4.1 となり，これにより，“フラットな HydLa プログラム” を得ることができる．

リスト 4.1　リスト 4.3 から制約呼び出しを展開したプログラム

```
1 (ht = 10 /\ v = 0),
2 ([](ht' = v /\ v'  = -10))
3   << ([](ht- = 0 => v = -(4/5)*v-)).
```

リスト 4.1 に対して 4.4 節において定義される関数 ms を適用するとリスト**??**の解候補モジュール集合の集合を得ることができる．なお，このモジュール集合の集合は呼び出しが展開済みのため多少分かりにくいが，展開前は以下の通りである．

リスト 4.2　リスト 4.1 の解候補モジュール集合の集合

```
1  {
2   {(ht = 10 /\ v = 0)},
3   {([](ht- = 0 => v = -(4/5)*v-))},
4   {([](ht- = 0 => v = -(4/5)*v-)),
5    ([](ht' = v /\ v'  = -10))},
6   {(ht = 10 /\ v = 0),
7    ([](ht- = 0 => v = -(4/5)*v-))},
8   {(ht = 10 /\ v = 0),
9    ([](ht' = v /\ v'  = -10)),
10   ([](ht- = 0 => v = -(4/5)*v-))}
11 }
```

リスト 4.3　リスト 4.1 の解候補モジュール集合の集合

```
1 {{INIT},
2  {BOUNCE},
3  {BOUNCE, FALL},
4  {INIT, BOUNCE},
5  {INIT, BOUNCE, FALL}}
```

4.2　HydLa の詳細な構文定義

3.3 章で定義された HydLa の基本構文を拡張し，詳細な構文定義をおこなった．

まず，演算子に対する優先順位および結合性の定義をおこなった．定義した結果を表 4.1 に示す．この表において，上位にあるほど優先順位が高く，表内の同じ行に掲載されている演算子は同じ優先度をもつ．

演算子	名称	結合性
’	微分	左から右
-	左極限	
+ -	単項プラス・マイナス	右から左
* /	乗算・除算	左から右
+ -	加算・減算	
= != <= < >= >	関係演算子	
=>	含意制約	右から左
[]	Always 制約	
& /\	論理積	左から右
| \/	論理和	
,	並列合成	
<<	弱合成	

表 4.1　演算子の優先順位および結合性

そして，表 4.1 の優先順位を元に HydLa の詳細構文を定義した．リスト 4.4 が作成した詳細構文である．ただし，作成した詳細構文は文献**??**における基本構文をもとに作成したために，3.3 章の基本構文とは若干異なっている．

リスト 4.4　HydLa の詳細な構文定義

```
1      hydla_program = statements
2         statements = {(def_statement | program) '.'}
3            program = program_parallel
4   program_parallel = program_ordered {parallel program_ordered}
5    program_ordered = program_factor {weaker program_factor}
6     program_factor = program_caller
7                    | '(' program_parallel ')'
8                    | module
9      def_statement = constraint_def
10                   | program_def
11       program_def = program_callee '{' program '}'
12     constraint_def = constraint_callee equivalent constraint
13  constraint_callee = constraint_name formal_args
14  constraint_caller = constraint_name actual_args
```

```
15    program_callee = program_name formal_args
16    program_caller = program_name actual_args
17            module = constraint
18        constraint = logical
19           logical = logical_term {logical_or logical_term}
20      logical_term = always_term {logical_and always_term}
21       always_term = [always] ask
22               ask = ask_logical implies constraint
23                   | tell
24                   | constraint_caller
25                   | '(' logical_term ')'
26              tell = expression comp_op expression
27        expression = arithmetic
28        arithmetic = arith_term {add arith_term}
29                   | sub
30        arith_term = unary {mul unary}
31                   | div
32             unary = [positive| negative] limit
33             limit = diff [previous]
34              diff = factor {differential}
35            factor = variable
36                   | number
37                   | '(' expression ')'
38       ask_logical = ask_logical_term {logical_or ask_logical_term}
39  ask_logical_term = comparison {logical_and comparison}
40        comparison = expression comp_op expression
41                   | constraint_caller
42                   | '(' ask_logical ')'
43            number = digit_p {digit_p} ['.' digit_p {digit_p}]
44          variable = identifier
45    bound_variable = identifier
46   constraint_name = identifier
47      program_name = identifier
48        identifier = (alpha_p | '_') {alpha_p | digit_p | '_'}
49       actual_args = ['(' [expression {',' expression}] ')']
50       formal_args = ['(' [bound_variable {',' bound_variable}] ')']
51           comp_op = less_eq
52                   | less
53                   | greater_eq
54                   | greater
55                   | equal
56                   | not_equal
57        equivalent = "<=>"
58           implies = "=>"
59            always = "[]"
60      differential = '''
61          previous = '-'
62             weaker = "<<"
63          parallel = ','
64              less = '<'
65           less_eq = "<="
66           greater = '>'
67        greater_eq = ">="
68             equal = '='
69         not_equal = "!="
70       logical_and = "&" | "/\"
71        logical_or = "|" | "\/"
72               add = '+'
73               sub = '-'
74               mul = '*'
75               div = '/'
76          positive = '+'
77          negative = '-'
```

これにより，HydLa プログラムの詳細な構文が定義され，hydLa プログラムは LL(k) により実際にパース可能な構文となっている事を確認した．

4.3　制約およびプログラム呼び出しの展開

4.4　解候補モジュール集合の導出手法

本節では、HydLa プログラムが与えるモジュールの半順序構造からある時刻における解候補モジュール集合を求める手法を示す.

解候補モジュール集合は以下に定義する関数 ms を用いて導出する．たとえば，$ms((A, B \ll C))$ はモジュールの半順序集合 $A, B \ll C$ の解候補モジュール集合の集合を求める.

解候補であるモジュール集合の集合は以下の規則により構築される．なお M は制約モジュール，MS は制約モジュール集合，X および Y は解候補モジュール集合の集合である.

$$ms(M) = \{\{M\}\} \tag{4.1}$$
$$ms((MS_1, MS_2)) = parallel(ms(MS_1), ms(MS_2)) \tag{4.2}$$
$$ms((MS_1 \ll MS_2)) = ordered(ms(MS_1), ms(MS_2)) \tag{4.3}$$
$$parallel(X, Y) = X \cup Y \cup \{x \cup y \mid x \in X, y \in Y\} \tag{4.4}$$
$$ordered(X, Y) = Y \cup \{x \cup y \mid x \in X, y \in Y\} \tag{4.5}$$

以下に，この規則の適用例を挙げる.

$$\begin{aligned} ms((A, B)) &= parallel(ms(A), ms(B)) \\ &= parallel(\{\{A\}\}, \{\{B\}\}) \\ &= \{\{A\}, \{B\}, \{A, B\}\} \end{aligned}$$

$$\begin{aligned} ms((A \ll B)) &= ordered(ms(A), ms(B)) \\ &= ordered(\{\{A\}\}, \{\{B\}\}) \\ &= \{\{B\}, \{A, B\}\} \end{aligned}$$

$$\begin{aligned} ms((A \ll B, C \ll D)) &= parallel(ms((A \ll B)), ms((C \ll D))) \\ &= parallel(\{\{B\}, \{A, B\}\}, \{\{D\}, \{C, D\}\}) \\ &= \{\{B\}, \{D\}, \{A, B\}, \{C, D\}, \{B, D\}, \\ &\quad \{A, B, D\}, \{B, C, D\}, \{A, B, C, D\}\} \end{aligned}$$

上の定義から，制約間に優先順位をつけない合成をおこなう演算子 “,” は結合則を満たすことが容易にわかる.

一方，“≪” の結合則については，

$$\begin{aligned}
ms(((A \ll B) \ll C)) &= \mathit{ordered}(ms((A \ll B)), ms(C)) \\
&= \mathit{ordered}(\{\{B\}, \{A, B\}\}, \{\{C\}\}) \\
&= \{\{C\}, \{B, C\}, \{A, B, C\}\} \\
ms((A \ll (B \ll C))) &= \mathit{ordered}(ms(A), ms((B \ll C))) \\
&= \mathit{ordered}(\{\{A\}\}, \{\{C\}, \{B, C\}\}) \\
&= \{\{C\}, \{A, C\}, \{B, C\}, \{A, B, C\}\}
\end{aligned}$$

のため，結合則を満たさないことがわかる．また，$(A \ll (B \ll C))$ と記述した場合，制約モジュール A は B が矛盾を起こしている場合でも採用されうることがわかる．なお，演算子 “≪” は左結合的であり，

$$(A \ll B \ll C) = ((A \ll B) \ll C)$$

である．

分配則 $(A \ll (B, C)) = (A \ll B, A \ll C)$ は満たされない．それぞれの解候補は

$$\begin{aligned}
ms((A \ll B, A \ll C)) &= \mathit{parallel}(ms((A \ll B)), ms((A \ll C))) \\
&= \mathit{parallel}(\{\{B\}, \{A, B\}\}, \{\{C\}, \{A, C\}\}) \\
&= \{\{B\}, \{C\}, \{A, B\}, \{A, C\}, \{B, C\}, \\
&\quad \{A, B, C\}, \{A, B, C\}, \{A, A, B, C\}\} \\
ms((A \ll (B, C))) &= \mathit{ordered}(ms(A), ms((B, C))) \\
&= \mathit{ordered}(\{\{A\}\}, \{\{B\}, \{C\}, \{B, C\}\}) \\
&= \{\{B\}, \{C\}, \\
&\quad \{A, B\}, \{A, C\}, \{B, C\}, \{A, B, C\}\}
\end{aligned}$$

となり，両者を集合の集合として扱うのならば分配法則が満たされる．しかし，第 6.3 節で述べるように HydLa のモジュール集合は多重集合とみなすべきであり，同一モジュールを複数回記述した場合，それぞれは区別して扱う必要がある．そのため，$(A \ll (B, C)) \neq (A \ll B, A \ll C)$ となり，分配則は満たさないことがわかる．

4.5　無矛盾極大モジュール集合の導出手法

HydLa プログラムを実行して関数の具体形を定めていくときは，当該時刻の直前までの関数形と矛盾しない解候補モジュール集合のなかで，集合の包含関係に関して極大なもののうちの 1 つをその時点における制約条件として満たさなければならない．

解である無矛盾極大モジュール集合とは次の条件を満たす MMS の事である．ここで，無矛盾な解候補モジュール集合からなる集合を $\mathcal{MS}$ とする．

$$MMS \in \mathcal{MS} \wedge \neg(\exists MS \in \mathcal{MS}(MS \supsetneq MMS)) \tag{4.6}$$

4.5.1　無矛盾極大モジュール集合を 1 つ導出する手法

HydLa では解となる無矛盾極大モジュール集合が複数存在する場合，そのどれか 1 つを採用すれば良いことになっている．

解候補モジュール集合の集合から，無矛盾極大モジュール集合のうちの 1 つを導出するには，解候補モジュール集合の集合から要素数が多い順に矛盾が存在しないものを探索すればよい．

要素数の多い順に探索をおこなうことで極大が求められることは，解候補モジュール集合の集合が以下の性質を持つことからわかる．ここで，$\mathcal{MS}$ は解候補モジュール集合からなる集合であり，$size(MS)$ はモジュール集合 MS に含まれるモジュール数を表す．

$$\neg(\exists MS1 \in \mathcal{MS}\ \exists MS2 \in \mathcal{MS}\ (size(MS1) \leq size(MS2) \Rightarrow MS1 \supset MS2) \qquad (4.7)$$

4.5.2　無矛盾極大モジュール集合をすべて導出する手法

解となる無矛盾極大モジュール集合が複数存在する場合にすべての解を導出する手法について述べる．この手法は，全解探索をおこなうシミュレーション実行時に必要となる．

導出手法は以下の通りである．

1. 4.4 章で述べた関数 ms を用いることで，解候補モジュール集合の集合を導出する．
2. 導出された解候補モジュール集合の集合に対して，集合の包含関係による依存関係のグラフの構築をおこなう．
3. グラフの各ノードに対して無矛盾性を確認し，矛盾が存在した場合はそのノードおよびそれを包含する集合であるノードを削除する．この際，任意の順にノードを探索してよい．
4. 残ったグラフのノードの内，極大元であるものが無矛盾極大モジュール集合となる．むろん，極大元は複数存在する可能性がある．

例として制約階層の関係が `A << B, C << D` の場合について考える．$ms((A \ll B, C \ll D)) = \{\{A,B,C,D\},\{B,C,D\},\{A,B,D\},\{C,D\},\{B,D\},\{A,B\},\{D\},\{B\}\}$ の為，依存関係のグラフは図 4.1 となる．この例において，$\{A,B,C,D\}$ に矛盾が生じなければ無矛盾極大モジュール集合は図 4.1 のルートノードである $\{A,B,C,D\}$ となる．しかし，たとえば B と C の間に矛盾が存在する場合，$\{A,B,C,D\}$ と $\{B,C,D\}$ の 2 つのモジュール集合も矛盾する．矛盾が存在するモジュール集合を取り除くと図 4.2 となる．結果，無矛盾極大モジュール集合は図 4.2 の極大元である $\{C,D\}$ および $\{A,B,D\}$ となり，双方共に解となる．

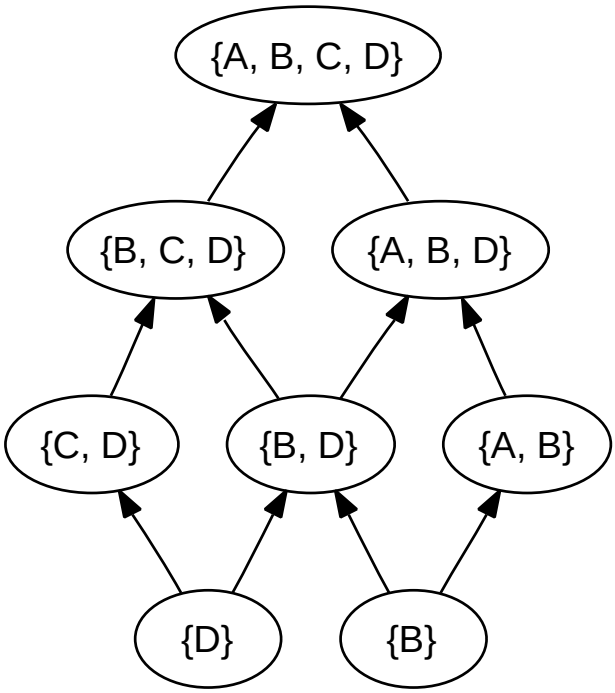

図 4.1　$ms((A \ll B, C \ll D))$ の依存関係のグラフ

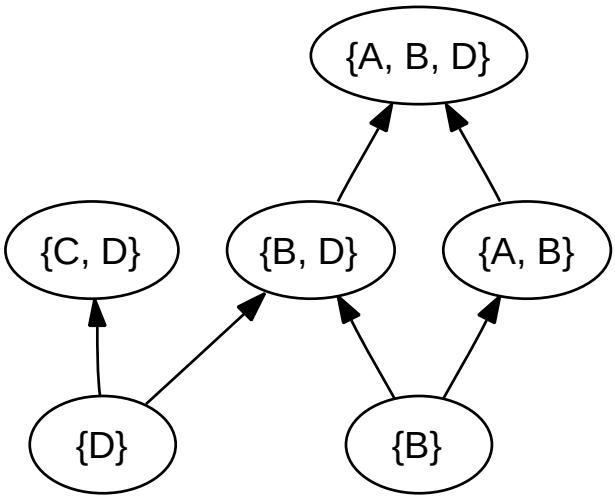

図 4.2　$ms((A \ll B, C \ll D))$ から B と C が矛盾する場合の矛盾するモジュール集合を取り除いた依存関係のグラフ

4.6　解候補モジュール集合の実行手法

解候補モジュール集合をシミュレーション実行する手法について述べる．ここで述べる手法は数式処理を用いたシミュレーション実行のためのアルゴリズムである．数値計算を用いた不確実値を持つモデルにおける実行手法については文献 [27, 26] を参照のこと．

実行は “離散フェーズ”，“連続フェーズ” の 2 フェーズから構成され，それぞれ交互に実行される．なお，各フェーズの開始時に毎回，極大元となる無矛盾なモジュール集合の導出がおこなわれ，そのモジュール集合が実行される．

4.6.1　簡約規則

各フェーズでは次の簡約規則を用いて制約式の解釈をおこなう．ここで，P はプログラム，c は制約式，g は含意条件，σ は制約ストア，Δ は次のフェーズで使用するプログラムである．

$$\text{tell} : \langle c, \sigma, \Delta\rangle \rightarrow \langle \epsilon, \sigma \cup \{c\}, \Delta\rangle$$

$$\text{ask} : \frac{\sigma \vdash g}{\langle (g \texttt{ => } P), \sigma, \Delta\rangle \rightarrow \langle P, \sigma, \Delta\rangle}$$

$$\text{always} : \langle (\texttt{[]}\, P), \sigma, \Delta\rangle \rightarrow \langle P, \sigma, (\Delta \ \ \texttt{/\textbackslash} \ \ (\texttt{[]}\, P))\rangle$$

$$\text{conjunction} : \frac{\langle P_1, \sigma, \Delta\rangle \xrightarrow{*} \langle P'_1, \sigma, \Delta\rangle}{\langle (P_1 \ \ \texttt{/\textbackslash} \ \ P_2), \sigma, \Delta\rangle \rightarrow \langle (P'_1 \ \ \texttt{/\textbackslash} \ \ P_2), \sigma, \Delta\rangle}$$

4.6.2　離散フェーズ

離散フェーズは離散変化を扱うフェーズであり，このフェーズで導き出された値は連続フェーズにおける ODE(Ordinary Differential Equations) の初期値となる．

4.6.3　連続フェーズ

連続フェーズは連続変化を扱うフェーズであり，このフェーズでは含意制約の導出状態が変化するか，制約階層をまたいだ制約間の関連性によって無矛盾極大モジュール集合が変化するまで計算がおこなわれ，最終的に導き出された値は離散フェーズにおける prev 変数の値となる．

Input: 前フェーズの変数表 X，制約モジュール集合 C
Output: 新しい変数表

```
1.  S := createStore(C)
2.  repeat
3.    T := C 中の tell 制約
4.    S := S ∪ T
5.    if S = ∅ then
6.      return ∅
7.    end if
8.    A := C 中の ask 制約
9.    expanded := false
10.   for g => c ∈ A do
11.     if ENTAIL(X, S, D, g) = true then
12.       C := C \ {g => c} ∪ {c}
13.       expanded := true
14.     end if
15.   end for
16. until expanded = true
17. return createVariableMap(S)
```

図 4.3　離散フェーズにおけるアルゴリズム

第 5 章

HydLa シミュレーション実行処理系の実装

本章では作成した統合処理系について述べる

5.1 処理系の概要

提案したアルゴリズムを元に，HydLa のシミュレーション実行処理系を作成した．現在の処理系の構成図は図 5.1 であり，Windows および Linux においてネイティブコードとしてコンパイル可能である．作成には C++ 言語を使用している．

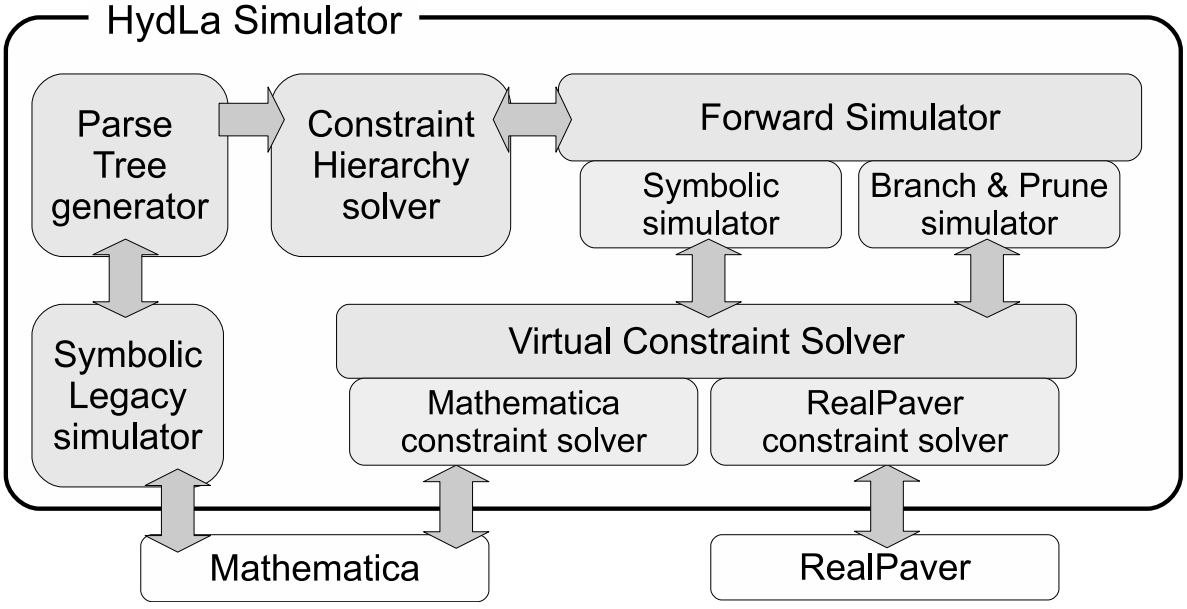

図 5.1 HydLa Simulator の version0.5 における構成図

ライブラリとして，Boost C++ Libraries[1] を使用しており，Boost のライブラリは iostreams，Conversion，Iterators，Operators，Foreach，Function，Lambda，Preprocessor，Xpressive，Regex，StringAlgorithms，SmartPointers，Bind，Spirit，Test，Program options といったものを使用している．

また，制約求解のために Mathematica[23] および RealPaver[9] を使用している．

実行可能な環境をなるべく広く取るために，環境依存の関数等を使用せざるを得ないときは，環境による違いを吸収するためのライブラリを経由することで環境に縛られないようにする方針をとっている．

開発には 3 名が関わり，処理系の全体の規模は約 20000 行である．自身の記述量は約 15000 行であり，処理系全体の設計及び，図中の色つきの部分の作成をおこなった．

処理系における根幹は構文木生成処理 (Parse Tree genarator)，制約階層処理 (Constraint Hierarchy solver)，シミュレーション実行処理 (Forward Simulator)，制約求解処理 (Virtual Constraint Solver) であり，それぞれが独立し任意に入れ替え・拡張可能となっている．そのため，Symbolic Legacy simulator のように構文木生成処理のみを利用するシミュレータを結合したり，Mathematica constraint solver や RealPaver constraint solver のように任意に求解手法を追加可能である．

5.1.1　処理系の構造

処理系のバージョン毎の比較をおこなう．

Version 0.3

version0.3 における構成図は図 5.2 である．

Parse Tree Generator において，入力されたプログラムをパースし，パースツリーの生成をおこなう．

Intermediate Code Generator において，パースツリーをもとに中間コードの生成をおこなう．中間コードについては**??**節を参照のこと．

Mathematica Symbolic Simulator において入力された中間コードにしたがって，実行をおこなう．この部分はすべて Mathematica を用いて記述されている．

Version 0.4

version0.4 における構成図は図 5.3 である．

本バージョンから，区間演算による処理系と数式処理による処理系が統合された．以前のバージョンまでは，パース以外のすべての処理が Mathematica を用いて記述されていたが，区間演算を用いた処理との共通化を計るために，すべて C++ を用いて書き直され

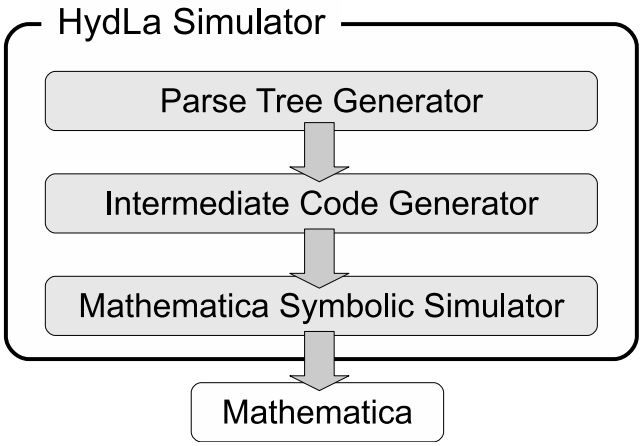

図 5.2　HydLa Simulator の version0.3 における構成図

た．区間演算による処理手法については文献 [27, 26] を参照のこと．

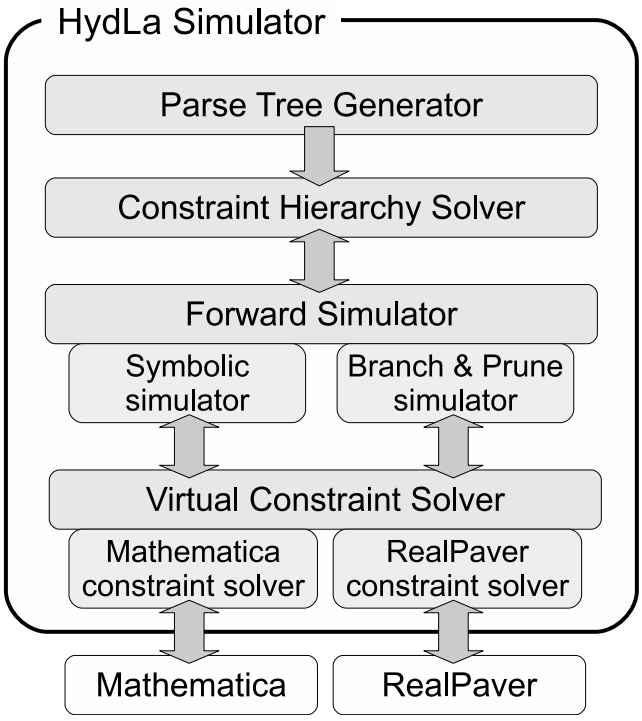

図 5.3　HydLa Simulator の version0.4 における構成図

Version 0.5

version0.5 における構成図は図 5.1 である．

version0.3 での “Mathematica Symbolic Simulator” を version0.4 に統合したもので

ある．これにより，以前の求解手法も利用可能となった．version0.3 と version0.4 での処理アルゴリズムは異なっており，どちらがが優位かは簡単には言えないため，有効であるといえる．

5.2　Parse Tree Generator

HydLa プログラムをパースするために，Boost の Spirit ライブラリを使用した．パースするために使用した詳細構文については 4.2 節を参照のこと．

パースツリーは図 5.8 のノードクラスによって構築される．すべてのツリーノードは Node クラスを継承しており更にそれぞれの子ノードの数に応じて，子ノードを持たない末端ノードである FactorNode クラス，子ノードを 1 つ持つ UnaryNode クラス，子ノードを 2 つ持つ BinaryNode クラスを継承している．これにより，子ノードの数による処理の切り分けがおこなえるようになっている．

次に，パースツリーの例を示す．Bouncing Particle モデルにおけるパースツリーは図 5.5 である．

5.2.1　Parse Tree の同型性判定

パースツリーは互いに同じ構造を持っているかどうかを判定可能である．判定には “厳密判定”，“同義判定” の 2 種類が存在し，以降でそれぞれについて述べる．

厳密な判定

厳密に判定をおこなう場合，ツリーとして同型であるかを調べる．そのため，$a = b$, $b = a$ の二つは別の Tree とみなされる．

同じ意味を持つ構造の判定

$a = b$ と $b = a$ は同じ構造とみなされる

$A /\backslash B /\backslash C$ の取りうる Tree 構造は 12 通りある．一例を図 5.6 に挙げる．

このような構造同士の比較をするために以下のアルゴリズムを適用する．

含意制約 => や除算 / といった交換律を満たさない演算に対してはこの手法を使用することができないため，ツリー構造の厳密な判定と同様のアルゴリズムを用いて比較する必要がある．

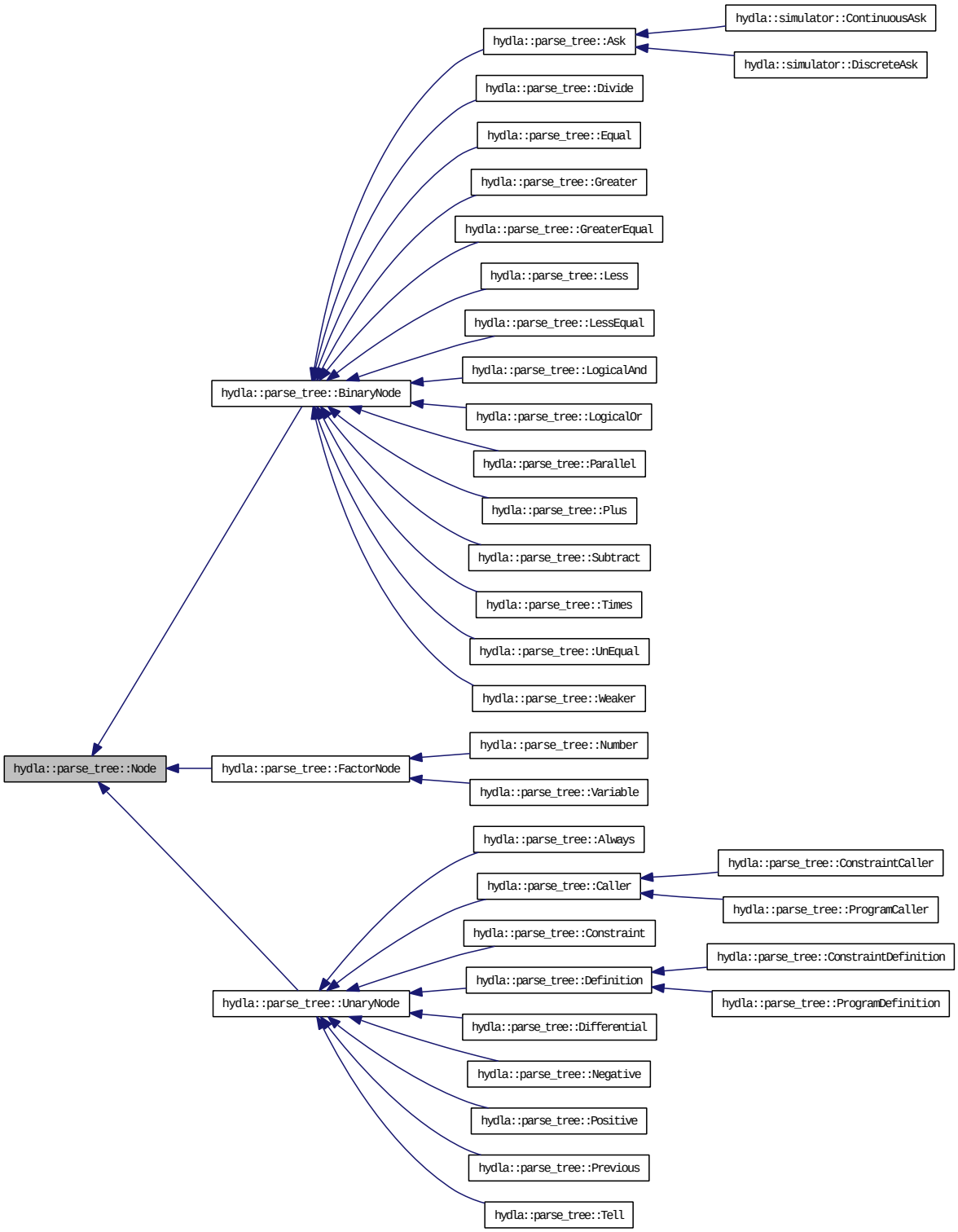

図 5.4　Node クラスの継承関係

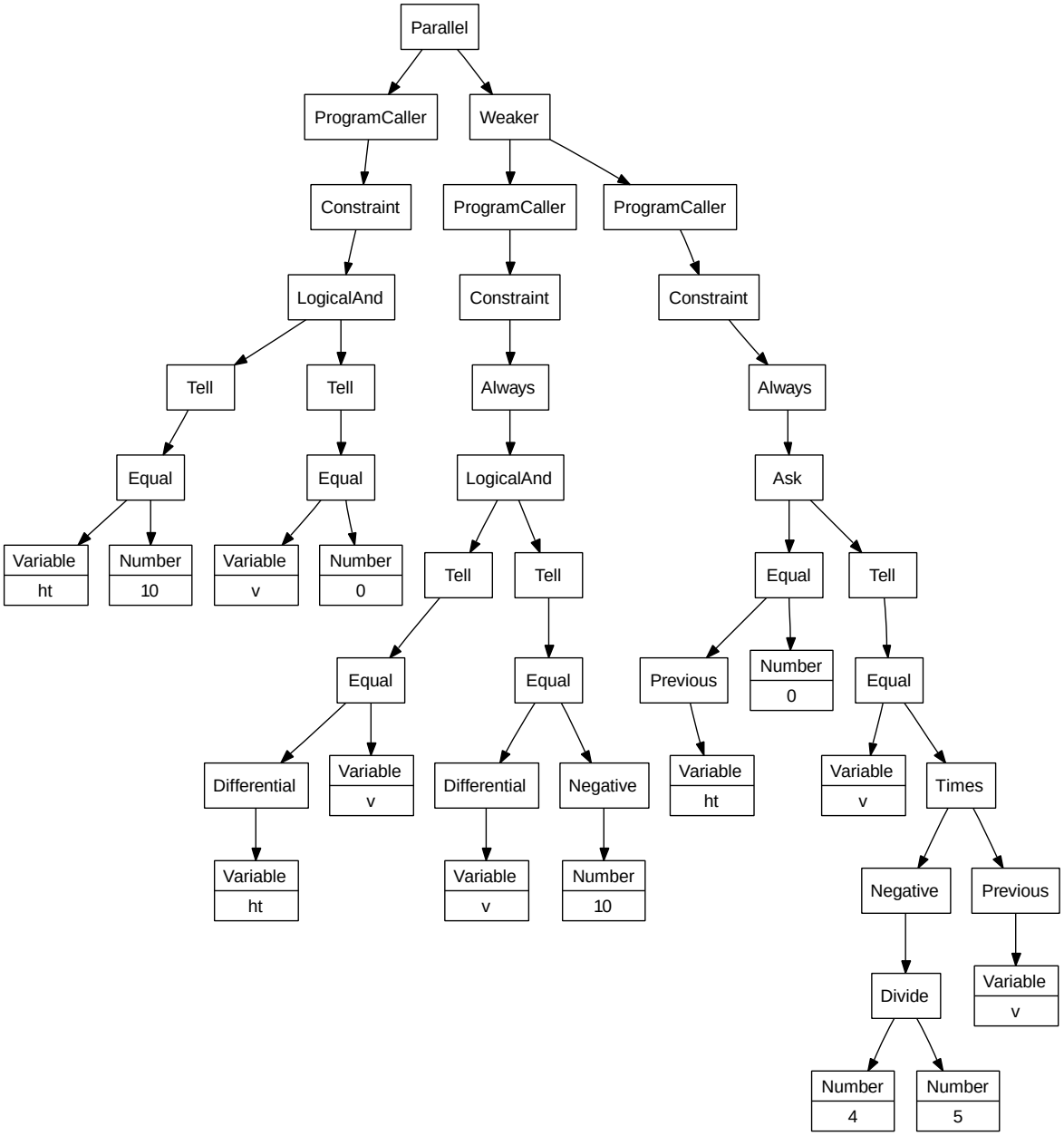

図 5.5　Bouncing Particle モデルの Parse Tree

5.2.2　ガード条件内の選言の分解

ガード条件内に選言が存在する Ask 制約が存在する場合提案されているアルゴリズムでは適用できない.

RealPaver で選言は利用できない

PositiveAsk，NegativeAsk のリストへ正しく収納できない

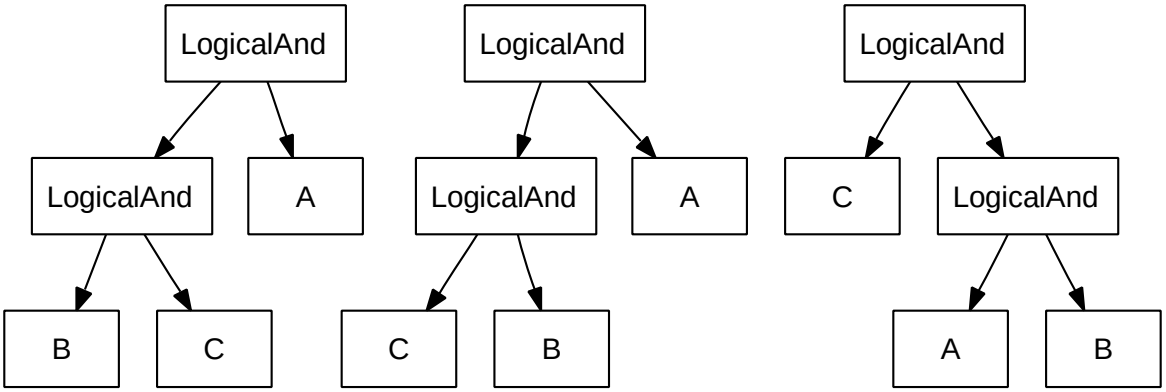

図 5.6　$A /\backslash B /\backslash C$ と同じ意味をもつツリー構造の一例

この問題を解決するには，以下の式からわかる通り

$$\begin{aligned} A \wedge B \Rightarrow C &= \neg(A \vee B) \vee C \\ &= (\neg A \wedge \neg B) \vee C \\ &= (\neg A \vee C) \wedge (\neg B \vee C) \\ &= (A \Rightarrow C) \wedge (B \Rightarrow C) \end{aligned}$$

$A \wedge B \Rightarrow C$ および $(A \Rightarrow C) \wedge (B \Rightarrow C)$ は論理式において同一であるため，入力された HydLa プログラムを選言のない Ask 制約へ変換すればよい．

この変換をおこなうために，次の 2 つの作業を実行する．

1. 与えられたプログラムに対し，分配束を適用することで，ガード条件を DNF(Disjunctive normal form) 形式に変換する
2. DNF 形式のガード条件を分解し，それぞれ独立したガード条件として書き換える．その際，Ask 制約の右辺は元の制約の右辺と同一とし，新たに作成された Ask 制約はそれぞれ連言を用いて合成する．

5.3　Constraint Hierarchy Solver

Constraint Hierarchy Solver では，4.4 節の手法に基づいて，制約階層の求解をおこなう．

ModuleSet クラスによってモジュール集合を表し，このクラスを保持する ModuleSetContainer によって解候補モジュールの集合の集合を表現する．

ModuleSetContainer およびそのサブクラスを表したものが図 5.7 である．ModuleSetList は 4.5.1 節の 1 つの解候補モジュールを導出するためのクラスである．Module-

SetGraph は 4.5.2 節のすべての解候補モジュールを導出するためのクラスである．

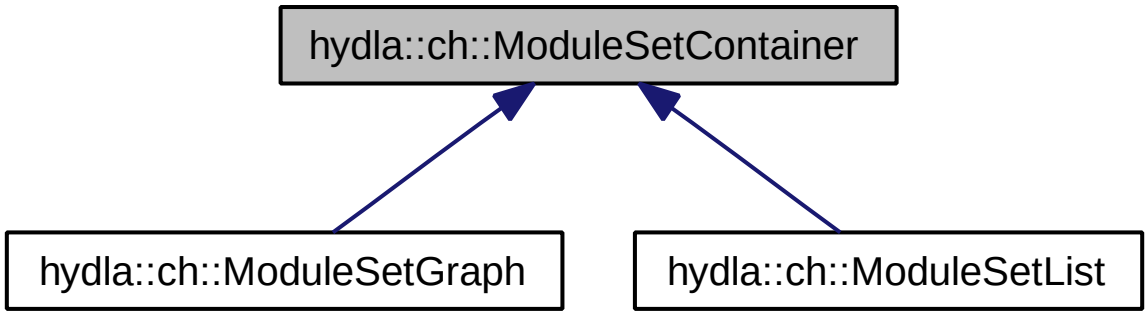

図 5.7　ModuleSetContainer のクラス図

5.4　Simulator

Simulator はシミュレーション実行をおこなうにあたっての基本機能を提供するためのクラスである．これを継承してさまざまな手法を用いたシミュレータを作成する．Simulator はテンプレートクラスとなっており，各フェーズの状態を保持するための PhaseState クラスを指定する必要がある．PhaseState はシミュレーション手法ごとに異なって来るため，それぞれの手法にあったクラスを指定すれば良いが，DefaultPhaseState という基本的な機能を提供する PhaseState が存在するため，それを利用してもよい．なお，DefaultPhaseState を利用するためには，“時刻クラス”，“変数クラス”，“値クラス”を指定する必要がある．

このように，さまざまな型を指定できるようにすることにより，汎用的な Simulator クラスを作成している．

5.5　Virtual Constraint Solver

Virtual Constraint Solver は制約処理をおこなうための機構であり，さまざまな求解手法を統一的に扱うことを目標として作成されている．

現在は，Mathematica を使用した MathematicaVCS と RealPaver を使用した RealPaverVCS が存在する．

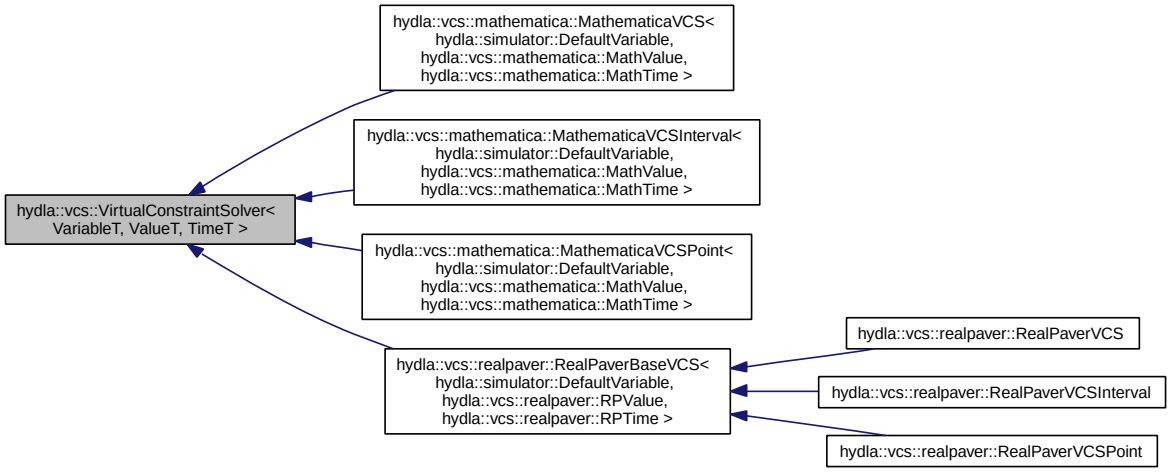

図 5.8

5.6　Symbolic Legacy Simulator

version0.3 時点における Mathematica で作成されたシミュレーション実行処理系を結合したものである．

他のシミュレーション実行処理系とは異なり，処理系には特別な中間コードを与えることでそれを解釈し，実行を行うようになっている．

中間言語は HydLa のプログラムを前置記法に変換したものであり，たとえば，ボールが自由落下し地面において跳ね返るモデル

```
INIT   <=> ht=10 \& v=0.
FALL   <=> [](ht' = v \& v'  = -10).
BOUNCE <=> [](ht- = 0 => v = -(4/5) * v-).
INIT, FALL << BOUNCE.
```

の中間言語は

```
group[
 unit[tell[Equal[ht,10]],
      tell[Equal[v,0]]],
 order[unit[always[ask[Equal[prev[ht],0],
                   tell[Equal[v,Times[Minus[Divide[4,5]],
                                      prev[v]]]]]]],
       unit[always[tell[Equal[Derivative[1][ht],v]],
```

```
                                    tell[Equal[Derivative[1][v],Minus[10]]]]]]]
```

となる．ここで，可読性を上げるために改行，インデント等を任意に挿入している．

この中間言語を元にシミュレーション実行処理系は，制約階層の処理，離散フェーズの処理，連続フェースの処理，含意条件の処理といったものをおこない，シミュレーション実行をおこなう．

第 6 章

HydLa を用いたモデリングに関する考察

6.1　制約階層が有用となる例題

本節ではハイブリッドシステムのモデリングをおこなうにあたって制約階層を用いることで記述がより簡潔となることを例題を用いて示す．制約階層を用いることでユーザの記述量を減らし，より直感的な表現をおこなうことができるようになる．

6.1.1　壁や床，箱等で跳ね返るボール

この例は，複数の含意条件に対する否定を取るにあたって制約階層が有用となる例題である．

重力が存在する空間に壁や床，箱等の障害物が配置され，その中でボールが運動をおこなうモデルである．モジュール WALL，FLOOR，BOX は壁や床，箱といった空間の境界における跳ね返りを記述した制約を与える．モジュール FALL と X_MOVE は自由落下，等速運動といったボールの連続変化を記述している．bx，by がボールの座標であり，このモデルにおけるボールの軌道は図 6.1 となる．

```
WALL(pos) <=>
 [](bx- = pos => bx' = -e*bx'-).
FLOOR(pos) <=>
 [](by- = pos => by' = -e*by'-).

BOX(x1,y1,x2,y2) <=>
  []((x1 <= bx- /\ bx- <= x2
```

```
      /\ (by- = y1 \/ by- = y2)
                  => by' = -e*by'-)
   /\ (y1 <= by- /\ by- <= y2
      /\ (bx- = x1 \/ bx- = x2)
                  => bx' = -e*bx'-)).

CONST  <=> [](e = 77/100).
INIT   <=> bx=4/3 /\ by=20
         /\ bx'=1  /\ by'=0.
FALL   <=> [](by'' = -10).
X_MOVE <=> [](bx'' =   0).

INIT, CONST,
(FALL /\ X_MOVE)
 << (BOX(5,   6, 6, 14)
  /\ BOX(1, 9/2, 4, 15)
  /\ WALL(0) /\ WALL(6)
  /\ FLOOR(0)).
```

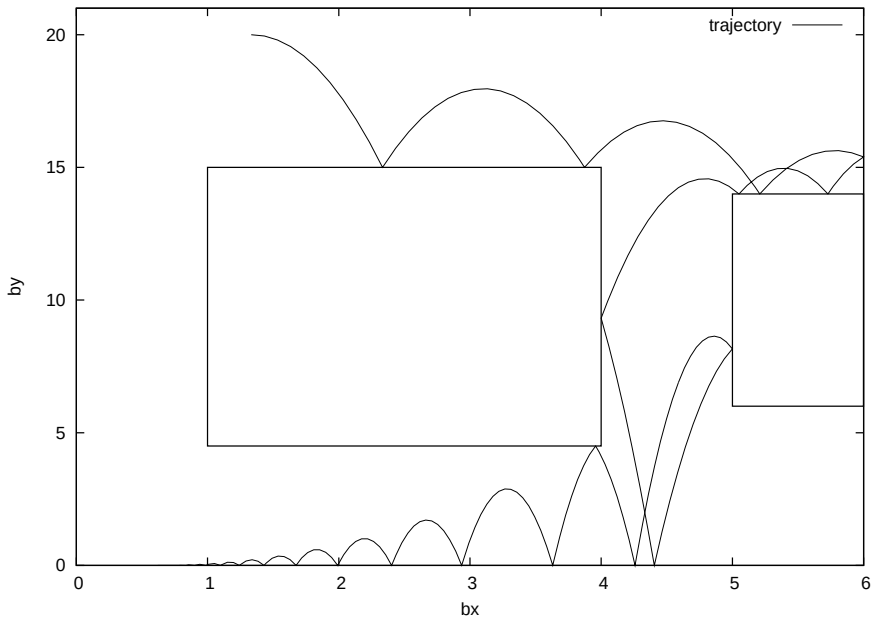

図 6.1　壁や床，箱等で跳ね返るボールの軌道

このモデルにおいて，制約階層を用いない場合、壁や床，箱といった障害物に衝突した際に矛盾が生じないように，FALL と X_MOVE を書き換えなければならない．すなわち，WALL，FLOOR，BOX と競合しないように含意条件を追加し，下記のように記述しなければならない．

```
FALL <=> []((by != 0)
 /\ (x<5 \/ 6<x \/ (y!=6   /\ y!=14))
 /\ (x<1 \/ 4<x \/ (y!=9/2 /\ y!=15))
                          => by'' = -10).
X_MOVE <=> []((bx != 0)
 /\ (bx != 6)
 /\ (y<6   \/ 14<y \/ (x!=5 /\ x!=6))
 /\ (y<9/2 \/ 15<y \/ (x!=1 /\ x!=4))
                            => bx'' = 0).
```

このように，ボールを動かすための連続制約と衝突し離散変化を起こすための離散制約が矛盾しないようにするために，制約階層を使えば不要となるいくつかの条件式をすべて書き下さなければならない．

さらに，BOX や WALL，FLOOR に与えられていた引数の値を即値として FALL 内に直接記述しなければならなくってしまっており，この後さらに箱や壁等をモデルに追加するごとに FALL を変更しなければならない．そのため，モジュール性が損なわれてしまう．

6.1.2　ナビゲーション

以下のモデルは，文献 [?] に掲載されているベンチマーク問題である．本来は到達可能性検証をおこなう問題だが，ここでは具体的な初期値から前向きにシミュレーションをおこなう．

平面に格子状に区切られた 6 つの領域があり，それぞれに異なる “流れ” が存在する．そのどこかに初速度を持った質点を置くと，質点は領域ごとの “流れ” に一致するように速度を変化させながら移動していく．px，py が質点の位置，vx, vy が質点の速度，vdx, vdy が質点の位置における “流れ” を表している．領域と，例題における質点の軌道を図 6.2 に示す．制約 L(x,y,vdx,vdy) は，点 (x,y) を左下とした広さ 1 の領域での “流れ” を記述している．

```
INIT <=>
 px=1/2 /\ py=3/2 /\ vx=1/2 /\ vy=1/2.
LAWS <=>
 [](px' = vx /\ py' = vy).
INERTIA <=> [](vx' = 0 /\ vy' = 0).
V(vdx, vdy) <=>
    vx' = -12/10*(vx - vdx)
 /\ vy' = -12/10*(vy - vdy).
L(x, y, vdx, vdy) <=>
 [](x<px /\ px<x+1 /\ y<py /\ py<y+1
       => V(vdx, vdy)).

INIT, LAWS,
INERTIA
 << (L(0,0,1,0) /\ L(0,1,0,-1)
  /\ L(1,0,1,0) /\ L(1,1,0.707,0.707)
  /\ L(2,0,0,0) /\ L(2,1,0,-1)).
```

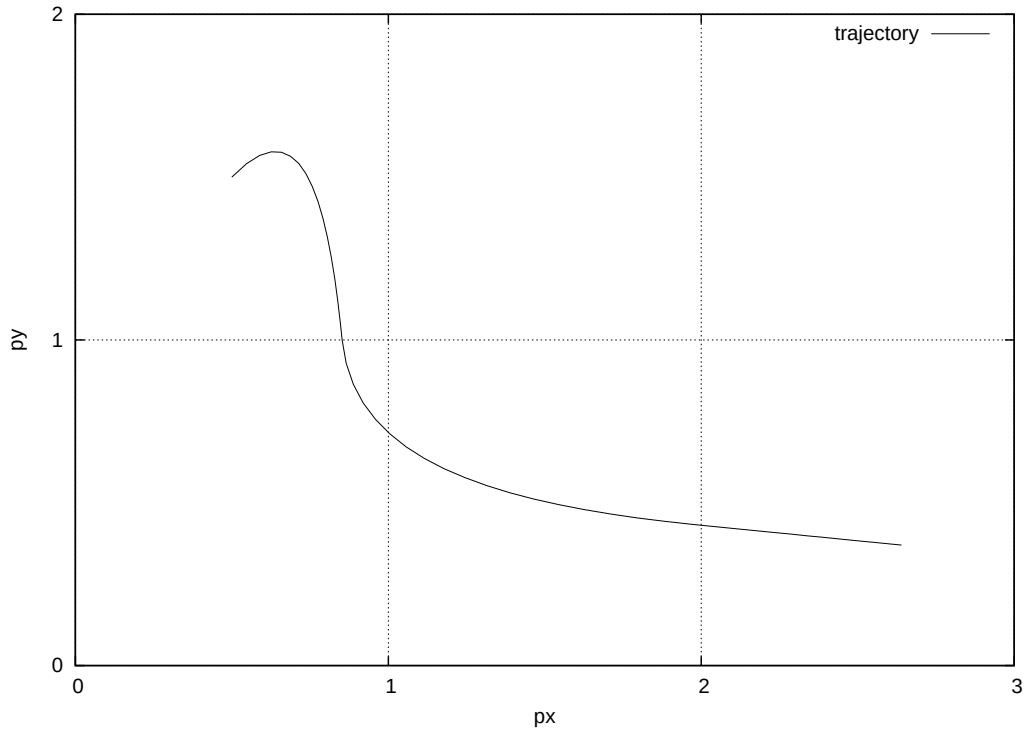

図 6.2　ナビゲーション問題の領域と質点の軌道

制約 L(x,y,vdx,vdy) は各領域における流れを与えるが，領域の境界については制約を与えない．ここでは境界上の質点は慣性により直前と同様に動くと考えることにする．このモデルではそのことを制約 INERTIA と制約階層によって表現している．制約 L(x,y,vdx,vdy) 内に等号をつけて境界上の動作を決定すると，境界上で複数の制約が矛盾を起こしたり，境界上では必ずどちらかの領域の制約にしたがうような記述になってしまう．境界上で必ずどちらかの領域の制約にしたがうようにすると，たとえば質点が左から境界に到達した場合でも右から到達した場合でも境界上ではどちらかの制約にしたがうこととなり，慣性によって動くという趣旨にあわない．制約階層を用いることによって，より簡潔な形で境界上での慣性を表現することができる．

6.2　強い優先度を持つモジュール同士の結合

制約モジュールを制約階層の強い位置に配置する場合には注意をして記述をおこなわなければならない場合がある．

次のモデルは，等速直線運動をおこなうボールが -5 および 10 の位置に存在する壁と衝突し，跳ね返るものである．

```
INIT       <=> x=0, x'=1.
MOVE       <=> [](x''=0).
WALL(pos) <=> [](x-=pos => x' = -x'-).
INIT, MOVE<<(WALL(-5), WALL(10)).
```

このモデルでは，x の値が pos となりボールが壁と衝突する際に WALL の制約と MOVE の制約が矛盾を起こす．そのため，WALL モジュールの制約の強さを MOVE モジュールよりも強くすることで，反射が起こるようにしている．しかし，図 6.3 の INIT を除いた ms((MOVE << (WALL(-5), WALL(10)))) の Hasse 図を見ればわかるように，{{MOVE, WALL(10)}, {WALL(10), WALL(-5)}, {MOVE, WALL(-5)}}の各組み合わせの優先度はすべて同一となっている．そのため，たとえば x=10 のとき MOVE と WALL(10) は矛盾を起こし，ボールが正しく跳ね返るためには{WALL(10), WALL(-5)} が採用される必要があるが，{MOVE, WALL(-5)}が採用され，跳ね返らずに壁をすり抜けることも正しい挙動となってしまう．

この問題を解決するためには WALL 同士を “,” ではなく “∧” で結合するようにすれば良い．つまり，

```
INIT, MOVE<<(WALL(-5) /\ WALL(10)).
```

とすることにより，WALL(-5) と WALL(10) は別々のモジュールではなく双方を連

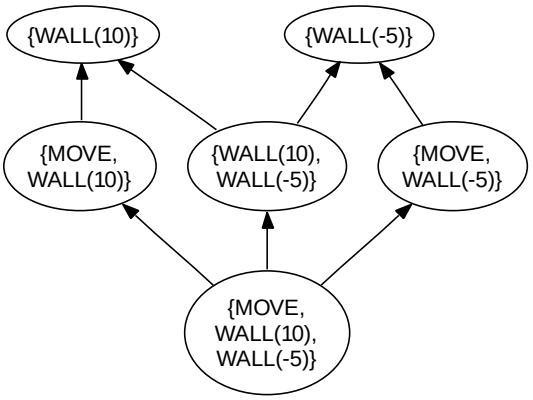

図 6.3　2 つの壁によってボールが跳ね返るモデルにおける解候補モジュール集合の Hasse 図

結した 1 つのモジュールとなる．解候補モジュール集合の集合は INIT を除くと $\{\{\texttt{WALL(-5)} \wedge \texttt{WALL(10)}\}, \{\texttt{WALL(-5)} \wedge \texttt{WALL(10)}, \texttt{MOVE}\}\}$ となるために，壁とぶつかり MOVE と WALL(10) が矛盾を起こす際も解のモジュール集合は $\{\texttt{WALL(-5)} \wedge \texttt{WALL(10)}\}$ が選ばれ，壁によって反射を起こす挙動のみが正しい解となる．

同様の問題は第 6.1 節で取り上げた 2 つの例題双方においても発生する．これらの例題は最上位の制約同士の結合に “∧” を用いているが，“,” を用いて結合をすると期待どおりの結果以外のものも解として有効となってしまう．たとえば壁や床，箱等で跳ね返るボールの例題において BOX や WALL，FLOOR といった制約を “,” を用いて結合をするとボールが壁や床，箱をすり抜ける解が有効になってしまう．

今回の場合は壁や床，箱といった物体が階層を持つ制約モジュールではなく，階層を持たない制約として定義されていたために “∧” を用いることですり抜けを回避できた．しかし，もしこれらが階層を持つ制約モジュールであった場合 “∧” を用いた結合はおこなえないため，この問題を回避できない．また，今回は “∧” で結合をおこなわなければならない制約はすべて同一のモジュール集合内に存在したが，プログラムが構造化することで，異なったモジュール集合に制約が分散してしまい，上手くいかない例題が存在する可能性がある．現在のところ階層を持つ制約モジュール同士を “∧” によって結合することができないことにより問題が発生する例題は見つかっていない．もし，これらの問題が存在する例題が見つかった場合には，制約階層の構文と意味論の拡張が必要となるかもしれない．

6.3　同じ制約モジュールを複数回記述した場合の挙動

HydLa の制約は “状態” を持つ．このことを説明するために以下の例を考える．

```
[](a=1 => []b=2)
```

この制約が含意条件 a=1 が成り立つ時点で採用された場合，

```
[](b=2)
```

へと展開がおこなわれ，その時点以降は [](b=2) という制約として扱われる．このように，含意条件の後に [] を伴った制約が出現する場合，その制約が採用されたことの効果が将来に波及する可能性があるため，制約は状態を持っていると見ることができる．

このことから，構文的に同じ制約を複数回記述した場合，それぞれを互いに独立した制約と見るかどうかでプログラムの意味が変化する．HydLa では制約を多重集合とみなし，個々の制約は別のものであると解釈する．

同じ制約を複数回記述した場合に，それらを区別することで結果が異なったものとなる例を以下に挙げる．

```
TIMER  <=> t=0 /\  [](t'=1).
A      <=> [](t=5 => [](f=1)).
B      <=> [](x=1 /\ (t<10  => x=2)).
C      <=> [](y=1 /\ (10<=t => y=2)).
```

上記の制約定義のもとで，TIMER，A<<B，A<<C および TIMER，A<<(B，C) という 2 つのプログラムを考える．現在の HydLa の仕様では制約定義の呼び出しはその場に展開されるため，同じ制約定義を複数回呼び出した場合はそれぞれを区別して考える．そのため，TIMER，A<<B，A<<C における 2 つの A は，同じ制約ではあるが別々のインスタンスということとなり，TIMER，A<<(B，C) における A は B と C に共有される同一のインスタンスということとなる．以下では変数 f に注目して議論を進め，解説のために A<<B，A<<C の二つの A はそれぞれ A1, A2 と命名し，TIMER，A1<<B，A2<<C と表記をおこなう．

1. $0 \leq$ t < 5 において
   モジュール宣言 TIMER，A1<<B，A2<<C および TIMER，A<<(B，C) 双方共に f に関する制約は有効とはなっていないため，f は任意の値をとる．
2. t $= 5$ において
   (a) TIMER，A1<<B，A2<<C について
   B は矛盾を起こしているために A1 は無効となっている．そのため A1 における含意条件の展開はおこなわれない．一方で，C は矛盾を起こしていないために A2 は有効となっている．そのため，A2 の制約は [](f=1) となり，f=1 となる．

（b）TIMER, A<<(B, C) について

B は矛盾を起こしているが C は矛盾を起こしていない．そのため，A は有効となっているために A の制約は [](f=1) となり，f=1 となる．

3. 5 < t < 10 において

（a）TIMER, A1<<B, A2<<C について

C は矛盾を起こしておらず有効であり，A2 は [](f=1) という制約となっているために，f=1 となる．

（b）TIMER, A<<(B, C) について

C は矛盾を起こしておらず有効であり，A は [](f=1) という制約となっているために，f=1 となる．

4. 10 ≤ t において

（a）TIMER, A1<<B, A2<<C について

C は矛盾を起こすために，A2 は無効となる．一方で B は矛盾を起こしていないために A1 は有効となる．t=5 において A1 は含意の展開がおこなわれなかったために，A1 は [](t=5 => [](f = 1)) のままである．そのため，f に関する有効な制約が存在しないため，f は任意の値を取る．

（b）TIMER, A<<(B, C) について

C は矛盾を起こすが B は矛盾を起こさない．そのため，A は有効となる．TIMER, A1<<B, A2<<C とは違い，A は 1 つのインスタンスであるために t=5 において展開がおこなわれている．そのため，A の制約は [](f=1) であり，f=1 となる．

以上のように，A<<B, A<<C と A<<(B, C) の 2 つは異なった結果となる．HydLa の制約定義は使用するごとに展開されるためにこのようになる．しかし，もし制約定義宣言を展開せずに別の場所に存在するインスタンスへのリファレンスとして扱う場合は A<<B, A<<C と A<<(B, C) の記述は同一のものとなる．現在のところ，リファレンスとして扱う必要が存在する例は見つかっていない．そのため，HydLa では数学における原則にしたがい，制約定義の呼出しは定義を用いて展開する意味論を採用している．

6.4　ask 制約を伴わない離散変化

HydLa では ask 制約を用いて離散変化を生じさせる様に記述することが想定されているが，tell 制約と制約階層を組み合わせることで離散変化を生じさせることが可能である．

例えば，以下のプログラムは $5 \leq t < 10$ において x

```
INIT  <=> t=0.
```

```
TIMER <=> [](t'=1).
RISE  <=> [](5<=t /\ t<10 => x=1).
INIT, TIMER, RISE.
```

```
INIT  <=> t=0.
TIMER <=> [](t'=1).
RISE  <=> [](5<=t /\ t<10 /\ x=1).
FLAT  <=> [](x=0).
INIT, FLRISE<<TIMER}.
```

片方微分方程式

```
INIT  <=> x=0, y=0
INC   <=> [](x'=1)
CROSS <=> [](x=5, y=2)
INIT, INC >> CROSS
```

t=5 の時点で y が 2 へと離散変化する両方微分方程式

```
INIT  <=> a=10, b=0, c=0
INC   <=> [](a'=-2, b'=3)
CROSS <=> [](a=b, c=10)
INIT, INC >> CROSS
```

t=2 の時点で c が 10 へと離散変化する

第 7 章

まとめと今後の課題

7.1　まとめ

本研究により，HydLa モデルに対するシミュレーション実行のアルゴリズムが具体化された．そして，様々な求解手法を統一的に扱うことができるシミュレーション実行処理系を作成することで，HydLa プログラムを複数の手法を使用して実際に動作させることが可能となった．

7.2　今後の課題

まず，現状では連続変化制約と離散変化制約の矛盾性を認識できないという問題が存在する為，矛盾を検出するための手法の導入が挙げられる．また，処理系の並列化を挙げることができる．全解探索をおこなう実行の場合，分岐が多量に発生するために非常に時間ががかかってしまう．分岐が起こった場合複数の状態が存在することになるが，それぞれは完全に独立したものであるため，高い並列効果を期待できる．

謝辞

本研究をおこなうにあたってお世話になりました方々に感謝の意を示したいと思います。

まず、研究をおこなうにあたって多岐にわたるご指導，助言をしていただいた上田 和紀教授には深く感謝致します．また、細部 博史准教授，石井 大輔氏には研究指導を始め，論文執筆時等さまざまな場面で大変お世話になりました．そして，HydLa 班同期である大谷 順司氏をはじめとする先輩方，後輩の皆様には研究はもちろんのこと，生活面においても数多くの相談に乗っていただき，大変お世話になりました．最後に，研究室での生活を非常に楽しいものとしてくださった上田研同期の皆様に感謝したいと思います．

2010 年 1 月　廣瀬 賢一

参考文献

[1] *Boost C++ Libraries*, http://www.boost.org/.

[2] Bemporad, A., Morari, M., Verification of Hybrid Systems via Mathematical Programming, In *Hybrid Systems: Computation and Control, volume 1569 of Lecture Notes in Computer Science*, pp. 31–45. Springer Verlag, 1999.

[3] Bockmayr, A., Courtois, A., Using Hybrid Concurrent Constraint Programming to Model Dynamic Biological Systems, In *ICLP '02: Proceedings of the 18th International Conference on Logic Programming*, pp. 85–99, London, UK, 2002. Springer-Verlag.

[4] Borning, A., Freeman-Benson, B., Wilson, M., Constraint hierarchies, *Lisp Symb. Comput.*, Vol. 5, No. 3, pp. 223–270, 1992.

[5] Carlson, B., Gupta, V., Hybrid CC and interval constraints, In Henzinger, T. A., Sastry, S., editors, *Hybrid Systems 98: Computation and Control*, Lecture notes in computer science, Vol 1386, pp. 80–94. Springer Verlag, April 1998.

[6] Carlson, B., Gupta, V., Hybrid cc with Interval Constraints, In *HSCC '98, LNCS 1386*, pp. 80–95. Springer, 1998.

[7] Chiu, C. K., Lee, J. H. M., Extending HCLP with partially ordered hierarchies and composite constraints, *J. Exp. Theor. Artif. Intell.*, Vol. 10, No. 1, pp. 5–24, 1998.

[8] Freeman-Benson, B. N., Kaleidoscope: mixing objects, constraints, and imperative programming, In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pp. 77–88, New York, NY, USA, 1990. ACM.

[9] Granvilliers, L., Benhamou, F., Algorithm 852: RealPaver: an interval solver using constraint satisfaction techniques, *ACM Trans. Math. Software*, Vol. 32, No. 1, pp. 138–156, 2006.

[10] Gupta, V., Jagadeesan, R., Saraswat, V. A., Computing with continuous change,

Sci. Comput. Program., Vol. 30, No. 1-2, pp. 3–49, 1998.

[11] Gupta, V., Jagadeesan, R., Saraswat, V. A., Models for Concurrent Constraint Programming, In *International Conference on Concurrency Theory*, pp. 66–83, 1996.

[12] Gupta, V., Jagadeesan, R., Saraswat, V., Bobrow, D., Programming in hybrid constraint languages, In Antsaklis, P., Kohn, W., Nerode, A., Sastry, S., editors, *Hybrid Systems II*, Vol. 999 of *Lecture Notes in Computer Science*. Springer Verlag, November 1995.

[13] Henzinger, T. A., The Theory of Hybrid Automata, In *LICS'96*, pp. 278–292. IEEE Computer Society Press, 1996.

[14] Ishii, D., Ueda, K., Hosobe, H., Simulation of Hybrid Systems based on Hierarchical Interval Constraints, In *2nd International Conference on Simulation Tools and Techniques (SIMUTools'09)*, 2009 (poster).

[15] Ivanov, A. P., On multiple impact, *Journal of Applied Mathematics and Mechanics*, Vol. 59, No. 6, pp. 887 – 902, 1995.

[16] Lunze, J., Lamnabhi-Lagarrigue, F., *Handbook of Hybrid Systems Control: Theory, Tools, Applications*, Cambridge University Press, 2009.

[17] Lygeros, J., Lecture notes on hybrid systems, In *Notes for an ENSIETA workshop*, 2004.

[18] Moore, R. E., *Interval Analysis*, Prentice-Hall Englewood Cliffs, N.J.,, 1966.

[19] Mysore, V. P., *Algorithmic Algebraic Model Checking: Hybrid Automata and Systems Biology*, PhD thesis, New York University, 2006.

[20] Saraswat, V., Jagadeesan, R., Gupta, V., Foundations of Timed Concurrent Constraint Programming, In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pp. 71–80, Paris, France, 4–7 July 1994. IEEE Computer Society Press.

[21] Saraswat, V., Rinard, M. C., Concurrent Constraint Programming, In *Seventeenth Annual ACM Symposium on the Principles of Programming Languages*, pp. 232–245, San Francisco, California, 1990.

[22] V.A.Saraswat, R.Jagadeesan, V.Gupta, Timed Default Concurrent Constraint Programming, *Journal of Symbolic Computation*, Vol. 22, No. 5–6, pp. 475–520, November–December 1996, Extended abstract appeared in the *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, San Francisco, January 1995.

[23] Wolfram Research, Inc., *Mathematica*,

http://www.wolfram.co.jp/products/mathematica/index.html.

[24] 上田和紀, 石井大輔, 細部博史, 制約概念に基づくハイブリッドシステムモデリング言語, 日本ソフトウェア科学会第 25 回大会論文集, 5A-2, 2008.

[25] 上田和紀, 石井大輔, 細部博史, 制約概念に基づくハイブリッドシステムモデリング言語 HydLa, 第五回システム検証の科学技術シンポジウム予稿集, pp. 1–6, 2008.

[26] 大谷順司, 廣瀬賢一, 石井大輔, 細部博史, 上田和紀, ハイブリッドシステムモデリング言語 HydLa の区間制約に基づく全解シミュレーション実行処理系, 情報処理学会創立 50 周年記念（第 72 回）全国大会, 1M-1, 2010.

[27] 大谷順司, 廣瀬賢一, 石井大輔, 上田和紀, 不確実値を持つハイブリッドシステムの高信頼なシミュレーション手法, 第 6 回ディペンダブルシステムシンポジウム論文集, pp. 145–153, 2009.

[28] 廣瀬賢一, 大谷順司, 石井大輔, 細部博史, 上田和紀, 制約階層によるハイブリッドシステムのモデリング手法, 日本ソフトウェア科学会第 26 回大会論文集, 2D-2, 2009.

[29] 廣瀬賢一, 大谷順司, 石井大輔, 細部博史, 上田和紀, 制約概念に基づくハイブリッドシステムモデリング言語 HydLa の実装, 第 11 回プログラミングおよびプログラミング言語ワークショップ, C3, 2009.

発表文献

[1] 大谷順司, 廣瀬賢一, 石井大輔, 細部博史, 上田和紀, ハイブリッドシステムモデリング言語 HydLa の区間制約に基づく全解シミュレーション実行処理系, 情報処理学会創立 50 周年記念全国大会 (第 72 回全国大会) 1M-1, 2010.

[2] 大谷順司, 廣瀬賢一, 石井大輔, 上田和紀, 不確実値を持つハイブリッドシステムの高信頼なシミュレーション手法, 第 6 回ディペンダブルシステムシンポジウム, pp.145–153, 2009.

[3] 廣瀬賢一, 大谷順司, 石井大輔, 細部博史, 上田和紀, 制約階層によるハイブリッドシステムのモデリング手法, 日本ソフトウェア科学会第 26 回大会, 2D-2, 2009.

[4] 廣瀬賢一, 大谷順司, 石井大輔, 細部博史, 上田和紀, 制約概念に基づくハイブリッドシステムモデリング言語 HydLa の実装, 第 11 回プログラミングおよびプログラミング言語ワークショップ, C3, 2009.

[5] 廣瀬賢一, 石井大輔, 上田和紀, 区間演算を用いた ODE Solver における任意精度演算の導入とパラメタ最適化, FIT2008(第 7 回情報科学技術フォーラム), A-007, 2008.