2012 年度 修士論文

ハイブリッド制約言語 HydLa の記号実行シミュレータ Hyrose の実装と最適化

提出日： 2013 年 1 月 26 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：5111B100-3
松本 翔太

概要

ハイブリッドシステムとは，時間の経過によって状態が連続変化したり，状態や方程式系自体が離散変化したりする動的システムを指す．物理学や制御工学，生命工学などさまざまな分野の問題がハイブリッドシステムとしてモデル化できる．HydLa はハイブリッドシステムの記述や検証を目的とした制約に基づく宣言型言語であり，システムを表現する数式や論理式をそのままプログラムに記述することができる．HydLa では制約間に優先順位を設けることで，システムの通常の動作と例外的な動作を簡潔に記述可能である．この特徴により，他のハイブリッドシステムモデリング手法であるハイブリッドオートマトンや Hybrid CC などと比較し，システムの取りうる全状態を列挙する必要が無いという利点がある．

我々は HydLa 処理系である Hyrose を開発しており，HydLa プログラムの高信頼なシミュレーションや検証を行うことを目的にしている．本研究ではまず，パラメータを含むモデルを記述した HydLa プログラムに対しても記号実行によるシミュレーションが可能となるよう，Hyrose を拡張した．拡張後の Hyrose では，システムの振る舞いがパラメータによって定性的に変化する場合，自動で場合わけを行うことで考えうるすべての振る舞いを求めることができる．この拡張によって，元々のハイブリッドシステムで設定できるパラメータのほか，モデリング誤差や，環境や経年による変化もパラメータとして扱うことができるようになった．実装の過程で，従来の HydLa 言語の仕様やシミュレーションアルゴリズム上では詳細な定義がなかった事項について，明確化を行った．

更に，以上のように実装した Hyrose に対して，実行アルゴリズムの最適化，実装を行うことで，シミュレーション能力を改善した．制約数の多いプログラムのシミュレーションを視野に入れた最適化を施したアルゴリズムも考案した．

Abstract

Hybrid Systems are dynamical systems with continuous changes of states and discrete changes of states or differential equation systems. Problems in various fields such as physics, control engineering, and biology can be modeled as hybrid systems. HydLa is a constraint-based declarative language for the modeling and verification of hybrid systems, and HydLa programs can directly include mathematical expressions and logical formulae that describe hybrid systems. In HydLa, normal and exceptional behaviors can be described concisely by setting priorities between constraints. With this feature, HydLa has an advantage that programmers don't have to enumerate all states, while it is necessary in other hybrid system modeling methods such as Hybrid automata and Hybrid CC.

We are developing Hyrose, an implementation of HydLa, for the purpose of dependable simulation and verification of HydLa programs. In this research, we extended Hyrose to be able to simulate HydLa programs with parameters symbolically. Extended Hyrose can perform automatic case analysis of systems whose behaviors change qualitatively depending on parameters. This extension enabeld Hyrose to handle not only parameters in original hybrid systems, but also modeling errors, environmental changes and changes over time as parameters. In the process of implementing this extension, we clarified several matters that had no detailed definition in the existing specification of HydLa and the simulation algorithm.

In addition, we improve simulation power of Hyrose by optimization of simulation algorithm and its implementation. We also devised an optimized algorithm that took account of the simulation of program with many constraints.

目次

第 1 章　はじめに 1
1.1　研究の背景と目的 1
1.2　論文の構成 2

第 2 章　ハイブリッド制約言語 HydLa 3
2.1　ハイブリッドシステム 3
2.2　HydLa 言語の概要 3
2.3　記法 4
2.4　本論文で扱う構文の注意点 5
2.5　プログラム例 5

第 3 章　HydLa の非決定実行アルゴリズム 7
3.1　記号定数 7
3.2　アルゴリズムの全体像 8
3.3　補助関数 10
3.4　アルゴリズムの実行例 13

第 4 章　HydLa の記号実行シミュレータ Hyrose 16
4.1　Hyrose の概要 16
4.2　精度保証 17
4.3　全解探索機能 19
4.4　有界モデル検査機能 19

第 5 章　Hyrose の実装 20
5.1　処理系の構成 20
5.2　解軌道の表現 21

5.3 暗黙的な連続性 22
5.4 時刻 0 の左極限値に関するガード条件 22

第 6 章 例題による動作確認 24
6.1 溝のある地面を跳ねる質点 24
6.2 電気回路 27
6.3 カーリング 28
6.4 その他の例題 31

第 7 章 Hyrose の実行性能の最適化 33
7.1 CalculateClosure 内のガード条件判定回数削減 33
7.2 制約の依存関係を用いた最適化 35

第 8 章 関連研究 43

第 9 章 まとめと今後の課題 45
9.1 まとめ 45
9.2 今後の課題 45

謝辞 47

参考文献 48

発表文献 50

図目次

2.1 天井に向かって投げ上げられる質点のモデリング例 6
2.2 投げ上げられる質点の軌道 6

3.1 HydLa 処理系のシミュレーションの非決定実行アルゴリズム 9
3.2 CheckConsistencyPP のアルゴリズム 11
3.3 CheckConsistencyIP のアルゴリズム 12
3.4 CalculateMCS のアルゴリズム 12
3.5 CalculateClosure のアルゴリズム 15

4.1 Mathematica を用いた解軌道の出力例 18

5.1 Hyrose 構成図（太字は本研究での主要部分） 21
5.2 解軌道木の例 22

6.1 溝のある地面を跳ねる質点の図 25
6.2 溝のある地面を跳ねる質点のプログラム 26
6.3 電気回路（文献 [10] より引用） 28
6.4 電気回路のプログラム 28
6.5 電圧 vc の振舞い 29
6.6 電流 il の振舞い 29
6.7 カーリングのプログラム 30
6.8 開始する速度を区間値とした場合のストーンの軌道 31
6.9 加速度を区間値とした場合のストーンの軌道 31

7.1 CalculateClosure の改善版アルゴリズム 34
7.2 最適化前後のガード条件導出判定時間 36

7.3 複数の質点が独立に跳ねるプログラム 36
7.4 図 7.3 の依存関係グラフ（上が PP，下が IP） 37
7.5 制約モジュール間の依存関係を利用する CalculateMCS のアルゴリズム 38
7.6 改善後の *CalculateMCS* 関数による図 7.3 の探索
（灰色のノードが枝刈で調べる必要のなくなったモジュール集合） 40
7.7 質点同士の衝突を考えたプログラム 41
7.8 質点 y1 と質点 y2 が衝突した時刻の探索
（灰色のノード：枝刈されたモジュール集合
緑色のノード：通常通り計算されたモジュール集合
赤色のノード：F2 に関しての計算結果を再利用したモジュール集合
白色のノード：調べられなかったモジュール集合） 42

表目次

2.1 HydLa の構文 5

6.1 電気回路のプログラムの実行時間（左が $0 \leq \mathtt{vc}_0 \leq 5$，右が $\mathtt{vc}_0 = 2$） . 30
6.2 その他の例題による実験結果 32

7.1 例題実行時のガード条件導出判定時間の割合 33
7.2 複数の質点が独立に跳ねる例題の実行結果 37

8.1 関連研究との比較 43

第 1 章

はじめに

1.1 研究の背景と目的

ハイブリッドシステム [5] とは，時間の経過によって状態が連続変化したり，状態や方程式系自体が離散変化したりする動的システムを指す．ハイブリッドシステムのモデリングツールは物理学や制御工学，生命工学などさまざまな分野で重要な役割を果たす．

HydLa はハイブリッドシステムの記述や検証を目的とした宣言型言語であり，上田研究室 HydLa 班で提案，開発が行われている．HydLa ではシステムを表現する数式や論理式をそのままプログラムとして記述可能とすることで，プログラミングを専門としない人々でも容易に利用できるようにすることを考えている．また，HydLa では制約間に優先順位を設け，システムの通常の動作と例外的な動作を簡潔に記述することを可能にしている．これらの特徴により，HydLa ユーザが直感的かつ簡潔なモデリングを行うことができると期待される．

Hyrose は HydLa の処理系であり，HydLa プログラムの高信頼なシミュレーションや検証を行うことを目的としている．Hyrose は基本的に，多くの状態変数を持つ大規模なシステムを扱うことよりも，複雑な挙動を持つシステムの精密な解析や理解を目指している．HydLa プログラムは一般に，振る舞いが一意に定まらないようなハイブリッドシステムを記述することができるが，従来の Hyrose ではそのような不確定性を持つプログラムを正しくシミュレーションすることはできなかった．しかしモデリング誤差や，システムの特性の環境・経年による変化を考慮したモデルを扱う場合など，不確定性を持つプログラムを扱えることは，現実のハイブリッドシステムの高信頼なシミュレーションを行うために重要な意味を持つ．更に不確定性を扱えることで，特定の性質を満たすためのパラメータの条件を求める，といった用途にも Hyrose を応用しやすくなる．このために，不

確定性を正しく扱う非決定実行アルゴリズムが文献 [11] において提案された．

本研究では，Hyrose をより有用なハイブリッドシステムの記号実行シミュレータとするため，文献 [11] を元に，アルゴリズムや言語仕様の詳細化とその実装を行った．更にシステム検証の足がかりとして，シミュレーションを行った範囲でシステムが指定された条件を満たすことを保証する，有界モデル検査機能の実装も行った．その後，実装した Hyrose を用いていくつかの例題について実験を行い，実験から得られた知見を元にアルゴリズムの最適化について，調査，提案，実装などを行った．

1.2 論文の構成

本論文の構成を示す．2 章で，ハイブリッド制約言語である HydLa について，その記法とプログラム例を紹介する．3 章で，HydLa プログラムを正しくシミュレーションするための非決定実行アルゴリズムを述べる．4 章で，HydLa の処理系である Hyrose の設計について述べる．5 章で，Hyrose の構成や，実装について述べる．6 章で実装した Hyrose を用いて例題のシミュレーションを行った結果について述べる．7 章で実装後，更に Hyrose に対して施した最適化について述べる．8 章で HydLa や Hyrose の関連研究について述べる．9 章で本研究のまとめや今後の課題について述べる．

第 2 章

ハイブリッド制約言語 HydLa

HydLa は上田研究室 HydLa 班によって提案，開発されているハイブリッドシステム 2.1 のモデリング言語である．本章では，HydLa が記述対象としているハイブリッドシステムの説明と，HydLa 自体に関する説明を行う．

2.1 ハイブリッドシステム

ハイブリッドシステムとは hybrid dynamical system とも呼ばれ，時間経過に伴い，状態が連続変化したり，状態や方程式系自体が離散変化したりする系である．ハイブリッドシステムは物理学や制御工学，生命学などさまざまな範囲に適用可能な概念であり，例として以下のようなものが挙げられる．

- 衝突を含む剛体の運動（衝突が離散変化，それ以外の運動が連続変化）．
- 摩擦を受ける物体の運動（静止摩擦力と動摩擦力の切り替えが離散変化，それ以外の運動が連続変化
- ロボットアームの制御（制御信号が離散変化，アームの運動が連続変化）．
- スイッチがある電気回路（スイッチの切り替えが離散変化，それ以外の状態が連続変化.

2.2 HydLa 言語の概要

HydLa は制約を用いてハイブリッドシステムのモデリングを行う宣言型言語である．システムが満たすべき個々の条件を，等式・不等式・微分方程式を原子論理式とする時相

論理式で記述することでモデリングを行う．HydLa プログラムは，時相論理式で記述した制約（これを「制約モジュール」と呼ぶ）の集合に，制約モジュール間の優先関係の集合を加えたものである．HydLa プログラム中に出現する変数は，すべて時刻についての関数変数であり，それらの変数の時刻 $t > 0$ における軌道のうち，プログラムの仕様を満たすものが実行結果（解軌道）となる．

2.3 記法

本論文で紹介する HydLa 処理系が対象としている HydLa[15] の構文を表 2.1 に示す．HydLa 言語についての詳細は，文献 [15] において与えられているので，ここでは概略のみを説明する．HydLa では，等式や不等式など，数式間で満たされるべき関係を原子制約 (atomic constraint) としている．各数式には，変数や数値そのものだけではなく，時間微分値や左極限値を，それぞれ E' や $E-$ のように含めることができる (expression)．

原子制約は論理積や含意記号で結合することができる．含意記号で結合された制約を条件つき制約と呼び，後件の制約は前件の制約が満たされる場合にのみ有効になる．なお，条件つき制約の条件部 (guard) には，論理和による結合を含めることができる．この条件部をガード条件と呼ぶ．上記の結合に加え，“[]” 記号によって時相論理式における always，つまり「その後常に有効である」制約も記述することができる．

以上のように記述された制約を制約モジュール (module) と呼ぶ．HydLa プログラムは全体として，この制約モジュールの半順序集合 (module set) を宣言している．“*MS1*, *MS2*” は制約モジュールに優先順位をつけない合成であり，“*MS1* << *MS2*” は *MS1* より高い優先順位を *MS2* に与える合成である．優先順位が低いモジュールは，それより優先順位が高い半順序集合が採用されている時のみ採用される．この優先順位を満たすようなモジュールの集合を解候補モジュール集合と呼び，HydLa プログラムの意味する解軌道は，各時刻において無矛盾な解候補モジュール集合のうち，極大となる制約モジュール集合の制約を満たすものとなる．

また, HydLa プログラム内ではモジュールの半順序集合や制約に名前をつけることができる (definition)．半順序集合に名前 *MSname* をつけたい場合は，$MSname(\vec{X})\{(MS)\}$ のように記述することができ，$\vec{X}$ は束縛変数の列を表す．$Cname(\vec{X})$ <=> C も同様に制約の定義文である．各定義文や半順序集合の宣言文は，“.” 記号によって区切る．

(hydla program)	$P ::= (D. \mid MS.)*$
(definition)	$D ::= MSname(\vec{X})\{(MS)\} \mid Cname(\vec{X}) <=> C$
(module set)	$MS ::= M \mid MS, MS \mid MS << MS \mid MSname(\vec{E})$
(module)	$M ::= A \mid M \wedge M \mid \mathrm{G}{=>}M \mid []M \mid Cname(\vec{E})$
(guard)	$G ::= A \mid G \wedge G \mid G \vee G$
(atomic constraint)	$A ::= E(relopE)+$
(expression)	$E ::=$ 通常の式 $\mid E' \mid E-$

表 2.1 HydLa の構文

2.4 本論文で扱う構文の注意点

前述の通り HydLa の構文は，文献 [15] でも与えられているが，本論文で扱う HydLa 処理系が対象としている構文には，以下のような差異がある．

- 存在量化子 ∃ を用いる記法を使用できない．
- ガード条件 G に論理和が含められる．
- 原子制約 A として $0 < x < 1$ のような，3 つ以上の数式間での等式もしくは不等式の記述を許すよう変更したものとなっている．

2.5 プログラム例

図 2.1 に，HydLa プログラムの例を示す．この例は，高さ 15 の位置にある天井に向かって，その下から初期速度 10 で投げ出される質点をモデリングしたものである．このプログラムに出現する変数は，質点の高さを表す y，およびその速度（一階微分），加速度（二階微分）をあらわす y’，y’’ である．また，y- や y’- は y や y’ の左極限値を表す．1–3 行目は制約の定義であり，1 行目ではシステムの初期状態を表す INIT，2 行目では質点が重力によって下に引き寄せられることを表す FALL （連続変化についての制約），3 行目では質点が天井に衝突する条件と衝突した際の振る舞いを表す BOUNCE（離散変化についての制約）を定義している．最後に，4 行目で制約の半順序集合を構成している．この例では，FALL を BOUNCE より弱い優先度で合成しているので，シミュレーション時は，FALL より BOUNCE を優先して採用することになる．つまり，FALL がこのシステムに

```
INIT <=> 9 <= y /\ y <= 11 /\ y' = 10.
FALL <=> [](y'' = -10).
BOUNCE <=> [](y- = 15 => y' = -4/5 * y'-).
INIT, (FALL << BOUNCE).
```

図 2.1 天井に向かって投げ上げられる質点のモデリング例

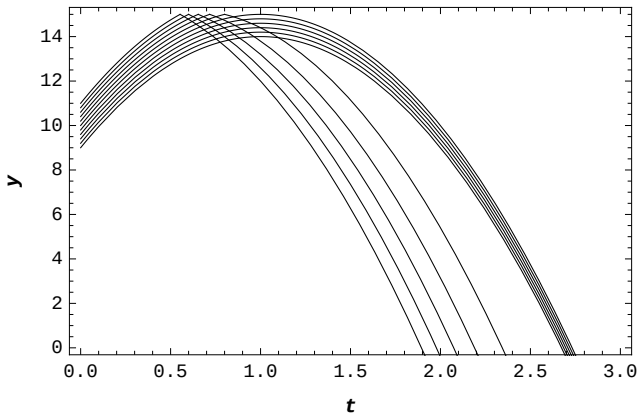

図 2.2 投げ上げられる質点の軌道

おける通常時の動作であり，BOUNCE が天井との衝突という例外的な動作である，と解釈できる．

このプログラムは INIT の中で，y の値を $9 \leq \mathrm{y} \wedge \mathrm{y} \leq 11$ と不等式制約で与えているので，y の初期値（以下 y_0 とする）は 9 以上 11 以下の任意の値が許され，解軌道は無限個考えられることになる．さらにこのモデルの場合，y_0 が 10 以上か否かによって BOUNCE 中の y- = 15 という条件がいつか成立するか，もしくは永遠に成立しないかが変化する．また，y_0 がちょうど 10 であるか否かによって，離散変化時に FALL と BOUNCE とが矛盾するかどうかも変化する．

このモデルの解軌道の一部をグラフ化したものを，図 2.2 に示す（y_0 を，9 から 11 まで，0.2 刻みで変化させた）．天井に衝突するか否かの境界となる，$\mathrm{y}_0 = 10$ を少し超えた部分では，y_0 の変化に対する軌道の変化率が大きいことが分かる．詳しくは 4 章以降で述べるが，Hyrose を用いると質点の位置を表す数式のうち，y_0 の関係する項の和は，衝突後の時刻 t において，$18(1-t)\sqrt{\mathrm{y}_0/5-2} - 13\mathrm{y}_0/5$ であることが分かる．この式の第 1 項を見ると $\mathrm{y}_0 = 10$ の正の近傍で変化率が $-\infty$ になっていることが確認できる．このように，変数値の軌道をパラメータを含む数式として求めることで，パラメータがシステムに与える影響を正確に把握することができる．

第 3 章

HydLa の非決定実行アルゴリズム

本章では，HydLa プログラムの実行結果である解軌道群がどのように得られるかを説明する．解軌道群を正しく求めるためのアルゴリズムは自明ではないが，文献 [11] において，不等式によって記述された制約を含むプログラムにも対応可能な，非決定実行アルゴリズムが提案されている．本章で述べるのは，処理系実装において得られた知見をもとに，記号定数（3.1 節）の概念を導入し，さらなる詳細化を施したアルゴリズムである．この詳細化により，シミュレーションの中ですでに決定した値とこれから決定する値とを切り分け，場合分けの処理を的確に行うことができるようになった．

なお，HydLa プログラムは一般に複数の解軌道を許す場合があるが，本章で紹介するアルゴリズムは，非決定的選択を行う各箇所で選択肢の一つを選択することでその中の一部だけが求まる．すべての解軌道を求める場合はすべての選択肢についてシミュレーションを行うようにすればよい．なお，非決定的選択とは具体的には以下の選択を指す．

- *GetElement* によって，集合の要素から任意の一つを選ぶ（図 3.1，図 3.2，図 3.3，図 3.5）．
- 制約モジュール集合の選択の際，候補の中で優先度が極大のものから任意の一つを選ぶ（図 3.4）．

3.1 記号定数

例えば図 2.1 のプログラムを満たす解軌道は無限個考えられるが，各解軌道は y の初期値によって，天井に対して「ぶつかる，接する，届かない」の 3 種類に分類できる．ここで同じ種類に分類された解軌道は，y の初期値をパラメータとすれば各時刻で充足する制

約が一致している（制約が一致しているので，もちろん y について解いた式も一致している）．このような同じ種類の解軌道を，本論文では「定性的に等しい解軌道」と呼び，異なる種類の解軌道を，「定性的に異なる解軌道」と呼ぶ．また，それらの分類を行うためにアルゴリズムが導入するパラメータを，記号定数と呼ぶ．

3.2 節で述べるアルゴリズムの中では，システムの離散変化時の変数値が一意的でない場合に記号定数が導入される．必ずしも時刻 0 の変数値に対してのみではないことに注意されたい．記号定数を用いることは，解軌道の定性的な分類だけでなく，異なる時刻間での変数値の関係（例えば初期値が軌道全体に与える影響）の正確な把握のためにも役立っている．

3.2 アルゴリズムの全体像

HydLa の非決定実行アルゴリズムを図 3.1 に示す．まず *SolveCH* によって，*HydLaProgram* から制約階層の処理を行い，各時刻で採用される制約モジュールの集合の候補リスト *MS* を得る（1 行目）．*MS* 内の要素は集合の包含関係に従って，トポロジカルソートされて並んでいる．M_{all} と V は，それぞれプログラム内で宣言されている全モジュールの集合と全変数の集合を表す（2,3 行目）．なお，V 内では，プログラム中に出現する範囲で，各変数の n 階微分も独立に保存している．以降，本論文で単に「変数」という場合，変数とその微分値の両方を指すものとする．

図 3.1 のアルゴリズムは時刻がシミュレーション終了時刻に到達するまで，離散変化を扱うポイントフェーズ (PP，特定の時刻) と連続変化を扱うインターバルフェーズ (IP，2 つの PP を両端とする開区間) の計算を交互に実行する．このことから，HydLa 言語自体は区分的に連続でない関数の一部も表現可能であるものの，このアルゴリズムが扱う関数は区分的に連続な関数に限定される．図中では，6 行目から 13 行目までが PP に対応し，14 行目から 22 行目までが IP に対応している．T は PP では現在時刻を，IP では時刻の両端を表す変数である（IP 開始時の T が左端に，IP 終了時の T が右端に対応している）．5 行目の while 文の継続条件「$T <_{CP} MaxT$」では，終了時刻 $MaxT$ と T を，記号定数の条件 CP を考慮して比較する．

S は制約の連言であり，制約ストアと呼ぶ．制約ストアには各変数の時刻に関する式や微分方程式も制約として入れることができる．高階関数 *CalculateMCS*（*MCS* は Maximal Consistent Set の略）では，そのフェーズにおける極大な無矛盾モジュール集合を求め [4]，対応する制約ストアを返す（9, 16 行目）．本関数は，最後の引数として PP または IP 用の無矛盾性判定関数 *CheckConsistency* を受け渡している．ここで

Input: HydLa プログラム *HydLaProgram*, シミュレーション終了時刻 *MaxT*
1: $MS := TopologicalSort(SolveCH(HydLaProgram))$
2: $M_{all} := MaxModuleSet(MS)$
3: $V := GetVariables(HydLaProgram)$
4: $T := 0$; $S := true$; $CP := true$; $E := \emptyset$
5: **while** $T <_{CP} MaxT$ **do**
6: //PP
7: $S := SubstituteTime(S, T)$
8: $(S, CP, E, \_, \_, \_) :=$
9: $CalculateMCS(S, MS, E, CP, T,$
$CheckConsistencyPP)$
10: **if** $S = false$ **then**
11: break
12: **end if**
13: $(S, CP) := AddParameters(S, CP, V)$
14: // IP
15: $(S, CP, E, M, A_-, A_+) :=$
16: $CalculateMCS(S, MS, E, CP, T,$
$CheckConsistencyIP)$
17: $S := SolveDifferentialEquation(S)$
18: **if** $S = false \vee \neg IsUnique(S, V)$ **then**
19: break
20: **end if**
21: $(MinT, CP) := GetElement(CompareMinTime($
$\{FindMinTime(S \wedge CP \Rightarrow g) | (g \Rightarrow c) \in A_-\}$
$\cup \{FindMinTime(S \wedge CP \Rightarrow \neg g) | (g \Rightarrow c) \in A_+\}$
$\cup \{FindMinTime(S \wedge CP \wedge M_-) |$
$M_- \in (M_{all} \setminus M)\}$
$\cup \{FindMinTime(S \wedge CP \wedge \neg M_+) | M_+ \in M\}$
$\cup \{(MaxT - T, true)\}))$
22: $T := MinT + T$
23: **end while**

図 3.1 HydLa 処理系のシミュレーションの非決定実行アルゴリズム

CalculateMCS 関数の実行中に，極大無矛盾モジュール集合の選択や，*CheckConsistency* 関数の *GetElement* の選択による解軌道の場合分けが発生した場合には，記号定数 *CP* の条件を絞り込んだものを返り値として受け取る．*CalculateMCS* 関数の 3 番目の返り値は，後件に always 記号がついている条件つき制約のうち，シミュレーション中に条件が一度でも成立したものの集合であり，こうした制約を展開済み always 制約と呼ぶ．展開済み always 制約は強い衝撃を受けて物体が壊れるなど，特定条件を満たした際に系の性質が永久に変化するような例題を記述する際に有用である．*CalculateMCS* 関数の結果，制約ストアが *false*（矛盾）となる場合，解軌道は存在しないものとしてシミュレーションを終了する（11,19 行目）．

PP，IP と実行し，IP で成り立つべき制約ストアが得られたら，ストア内の微分方程

式を *SolveDifferentialEquation* によって解く（17 行目）．こうして各変数が従う制約の，時刻を用いた表現が得られるので，この情報をもとに「次の離散変化時刻の候補と，その離散変化が最も早く起こる場合の記号定数の条件との組」の集合を *FindMinTime* と *CompareMinTime* [7] によって得る（21 行目）．*FindMinTime* では，特定の離散変化条件を満たす最小の正の時刻を求める関数であり，必要に応じて記号定数の条件の場合分けを行う．*CompareMinTime* では *FindMinTime* によって得られた結果同士を比較し，必要に応じて場合分けを行いながら最小となる時刻と，その場合の記号定数の条件の組の集合を求める．*FindMinTime* と *CompareMinTime* を終えた後は，得られた集合の要素から非決定的に 1 つを選び，T をその時刻に更新する（22 行目）．以上の繰り返しを，T がシミュレーション終了時刻 $MaxT$ に到達するまで繰り返すのが，HydLa の実行アルゴリズムの全体像である．

フェーズの実行結果として得られた制約ストアは，「直前のフェーズに関して成り立つ制約」として，次のフェーズに引き継がれる．ただし単純にフェーズを実行した直後の制約ストアには，$9 \leq \mathrm{y} \leq 11 \wedge \mathrm{y}' = 10 \wedge \mathrm{y}'' = -10$ のように不等式制約が含まれていたり，ある変数についての制約が全く含まれていなかったりする可能性が考えられ，そうした変数の値はフェーズ内で一意に定まらないことになる．このアルゴリズムでは PP の最後に *AddParameters* によって，値が一意に定まらない変数の出現を検知し，出現した場合はその値に対応する記号定数を導入し，置き換える（13 行目）．IP 中に *IsUnique* によって値が一意に定まらない変数の出現を検知した際は，このアルゴリズムでは対応していない解軌道であるため，シミュレーションを中断する (19 行目)．以上の処理によって，次のフェーズに引き継がれる制約ストア内では，各変数の値が等式制約によって表現される．

アルゴリズム内には明記していないが，各フェーズの時刻もしくは時刻の区間と，制約ストア S との組をつないだリストに，最終的な全記号定数の条件 CP の情報を付加したものが，1 つの解軌道の表現になる．

3.3 補助関数

本節では，図 3.1 に現れる主要関数について説明する．

Input: 制約ストア S，記号定数の条件 CP
Output: 充足可能性，記号定数の条件
1: $V := GetVariables(S)$
2: $CP_{tmp} := \exists V(S \wedge CP)$
3: **if** $CP_{tmp} = false$ **then**
4: **return** $(false, CP)$
5: **else if** $CP_{tmp} = CP$ **then**
6: **return** $(true, CP)$
7: **else**
8: **return** $GetElement(\{(true, CP_{tmp}),$
9: $(false, CP \wedge \neg CP_{tmp})\})$
10: **end if**

図 3.2 CheckConsistencyPP のアルゴリズム

3.3.1 CheckConsistencyPP

図 3.2 に PP における無矛盾性判定関数の処理内容を示す．最初に制約ストア内に出現する変数の集合 V を求め，その後制約ストアと記号定数の条件を同時に満たすような変数値が存在するかの判定を行う．この結果得られる条件 CP_{tmp} は，制約が充足可能であるための記号定数の条件を表しており，そのような条件が存在しない場合（$CP_{tmp} = false$ の場合）は $false$ と元々の条件 CP を返す．充足可能な記号定数の条件が存在する場合は，引数として与えられた記号定数の条件と一致するかを判定する．一致する場合は，必ず制約が充足可能な場合なので，$true$ と元々の条件 CP を返す．一致しない場合は，記号定数の条件によって充足可能性が変化する場合なので，「$true$ と CP_{tmp} の組」か，「$false$ と，CP を満たしかつ CP_{tmp} ではない条件の組」のいずれかを非決定的に返す．

3.3.2 CheckConsistencyIP

図 3.3 に，*CheckConsistencyIP* のアルゴリズムを示す．この関数の処理は本文中の *CheckConsistencyPP* と基本的に同じであり，制約ストア内の微分方程式を解くことと（1 行目），充足可能性判定を IP 開始時刻の正の近傍で行うこと（3 行目）だけが異なる．なお，本関数内で IP の開始時刻は常に 0 として扱っており，実際の時刻の適用はフェーズが IP から PP に切り替わるときに行っていることに注意されたい．

Input: 制約ストア S，記号定数の条件 CP
Output: 充足可能性，記号定数の条件
1: $S_t := SolveDifferentialEquation(S)$
2: $V := GetVariables(S_t)$
3: $CP_{tmp} := \exists V(Inf\{t \mid \exists_t(S_t \wedge t > 0)\} = 0) \wedge CP)$
4: **if** $CP_{tmp} = false$ **then**
5: **return** $(false, CP)$
6: **else if** $CP_{tmp} = CP$ **then**
7: **return** $(true, CP)$
8: **else**
9: **return** $GetElement(\{(true, CP_{tmp}),$
10: $(false, CP \wedge \neg CP_{tmp})\})$
11: **end if**

図 3.3 CheckConsistencyIP のアルゴリズム

Input: 制約ストア S，解候補モジュール集合のリスト MS，展開済み always 制約の集合 E，記号定数に関する条件 CP, 時刻 T, 無矛盾性判定関数 $CheckConsistency$
Output: 制約ストア，記号定数に関する条件，展開済み always 制約の集合，極大となるモジュール集合，成り立たないガード条件の集合，成り立つガード条件の集合
1: **for** $M \in MS$ **do**
2: **if** $T > 0$ **then**
3: $M := EliminateNotAlways(M)$
4: **end if**
5: $(S_{tmp}, E_{tmp}, CP, A_-, A_+) :=$
6: $CalculateClosure(S, M, CP, E, CheckConsistency)$
7: **if** $S_{tmp} \neq false$ **then**
8: **return** $(S_{tmp}, CP, E_{tmp}, M, A_-, A_+)$
9: **end if**
10: **end for**
11: **return** $(false, CP, E, \emptyset, \emptyset, \emptyset)$

図 3.4 CalculateMCS のアルゴリズム

3.3.3 CalculateMCS

図 3.4 に，*CalculateMCS* のアルゴリズムを示す．この関数は各フェーズで極大無矛盾となるモジュール集合を求め，モジュール集合に対応する制約ストア等の情報を得るための関数である．*MS* 内のモジュール集合のうち，大きいものから順に制約ストアの閉包を *CalculateClosure* で求め，最初に無矛盾であると分かったモジュール集合について，結果を返す．7.2 節において，この関数の最適化について言及する．

3.3.4 CalculateClosure

図 3.5 に，*Calcula teClosure* のアルゴリズムを示す．この関数内では与えられたモジュール集合に対し，制約ストアの閉包を求める．制約ストアの閉包は制約ストアの無矛盾性判定（5 行目）と，各ガード条件の導出判定および導出された場合の後件の追加（11〜25 行目）を，制約ストアが変化しなくなる（26 行目，$\neg Expanded$）まで繰り返すことによって得られる．ガード条件の導出判定において，「変数の値を絞り込めば導出可能」な条件が現れた場合は *BranchedAsks* に保持しておき，もしループ終了時にそのような条件が残っていたなら，*CalculateClosure* 関数を再帰的に呼び出すことで，当該条件が導出される場合とされない場合の制約ストアを調べる（27 行目〜43 行目）．もし両方が無矛盾なら軌道の場合分けが発生する．

3.4 アルゴリズムの実行例

例として，上記のアルゴリズムを用いて，図 2.1 のプログラムを $MaxT$ (> 1) 秒実行する過程を説明する．まず，制約モジュール集合の候補 MS は，$\{\{\mathtt{INIT}, \mathtt{FALL}, \mathtt{BOUNCE}\}, \{\mathtt{INIT}, \mathtt{BOUNCE}\}\}$ となる．M_{all} と V はそれぞれ $\{\mathtt{INIT}, \mathtt{FALL}, \mathtt{BOUNCE}\}$ と $\{\mathtt{y}, \mathtt{y'}, \mathtt{y''}\}$ になる．

最初の PP に入る．*SubstituteTime* は制約ストア内の式に時刻 T を代入する処理だが，制約ストアに制約が追加されていないので何も起きない．この PP では，$\{\mathtt{INIT}, \mathtt{FALL}, \mathtt{BOUNCE}\}$ が極大無矛盾集合となり，対応する制約ストアとして $9 \leq \mathtt{y} \leq 11 \wedge \mathtt{y'} = 10 \wedge \mathtt{y''} = -10$ を得る．ここで，y の値が一意に定まらないため，時刻 0 での y に対応する記号定数 py を導入し，制約 $9 \leq py \leq 11$ を CP に追加し，制約ストアを $\mathtt{y} = py \wedge \mathtt{y'} = 10 \wedge \mathtt{y''} = -10$ とする．

次に IP に入る．このフェーズでも $\{\mathtt{INIT}, \mathtt{FALL}, \mathtt{BOUNCE}\}$ が極大無矛盾モジュール集合となる．ただし，INIT と BOUNCE はそれぞれ初期時刻ではないことと，ガード条件が成り立たないことから，実質的に意味を持つのは FALL だけである．対応する制約ストアは，y の初期値を $\mathtt{y}_0$ のように表すと，$\mathtt{y}_0 = py \wedge \mathtt{y'}_0 = 10 \wedge \mathtt{y''} = -10$ となる．この制約ストアを *SolveDifferentialEquation* によって時刻 t を用いた表現に変換すると，$\mathtt{y} = py + 10\mathtt{t} - 5\mathtt{t}^2 \wedge \mathtt{y'} = 10 - 10\mathtt{t} \wedge \mathtt{y''} = -10$ が得られる．この制約ストアと先ほどの記号定数の条件をもとに，天井と衝突する時刻を *FindMinTime* によって求めると，天井に衝突する時刻は高さの初期値によって $1 - \sqrt{py/5 - 2}$ となる場合と ∞(衝突しない)

となる場合が考えられる．この 2 つの時刻をそれぞれ $CompareMinTime$ で $MaxT - T$ と比較する．ここで，T は 0 なので $MaxT - T$ は $MaxT$ に等しい．$MaxT > 1$ なので，最終的に $py \geq 10$ の場合には $1 - \sqrt{py/5 - 2}$ が，$py < 10$ の場合には $MaxT$ が $MinT$ に代入される．ここでは天井に衝突する場合を選び，$MinT = 1 - \sqrt{py/5 - 2}$，$CP$ が $10 \leq py \leq 11$ となるものとする．T はまだ $MaxT$ に達していないので，シミュレーションは次の PP に移る．

次の PP は，質点と天井との接触に対応している．このフェーズにおける極大無矛盾モジュール集合を求める際に，FALL と BOUNCE が矛盾するかどうかが py に依存する．つまり，py がちょうど 10 なら，質点は天井と接するだけで，速度の離散変化は起きない．一方，$py > 10$ なら速度の離散変化が発生し，FALL と BOUNCE は矛盾する．ここでは $py > 10$ の場合を選ぶことにする．この場合，極大無矛盾モジュール集合は $\{\mathtt{INIT}, \mathtt{BOUNCE}\}$ となり，制約ストアは $\mathtt{y} = 15 \wedge \mathtt{y'} = -10\sqrt{py/5 - 2}$ となる．y'' の値は一意に定まらないため，この時刻における y'' の値に対応する記号定数 $py2$ を導入する．ただしこの記号定数は他のフェーズの変数の値に関係しないため，実質的な意味を持たない．以上より，制約ストアは $\mathtt{y} = 15 \wedge \mathtt{y'} = -10\sqrt{py/5 - 2} \wedge \mathtt{y''} = py2$ となり，記号定数の条件は $10 < py \leq 11 \wedge -\infty < py2 < \infty$ となる．

その後，シミュレーションは IP に入るが，以降離散変化は起きないので，この IP が最後のフェーズとなる．基本的に y の初期値以外は最初の IP と同様なので，説明は省略する．

Input: 直前のフェーズの値に関する制約ストア S_{prev}，現在の制約モジュール集合 M，記号定数に関する条件 CP, 展開済み always 制約の集合 E，無矛盾性判定関数 $CheckConsistency(S)$
Output: 制約ストア，展開済み always 制約の集合，記号定数に関する条件，成立しない条件付き制約の集合，成立する条件付き制約の集合

```
 1: A+ := E
 2: repeat
 3:    S :=CollectTell(M, A+, Sprev);
 4:    (TF, CP) :=
 5:    CheckConsistency(S, CP)
 6:    if TF = false then
 7:       return (false, ∅, CP, ∅, ∅)
 8:    end if
 9:    Expanded := false
10:    BranchedAsks := ∅
11:    A− := CollectAsk(M, A+);
12:    for (g ⇒ c) ∈ A− do
13:       (TF, CP) :=
14:       CheckConsistency(S ∧ g, CP)
15:       if TF ≠ false then
16:          (TF, CP) :=
17:          CheckConsistency(S ∧ ¬g, CP)
18:          if TF ≠ false then
19:             BranchedAsks := BranchedAsks ∪ (g ⇒ c)
20:             continue
21:          end if
22:          Expanded := true
23:          if IsAlways(c)) then
24:             E := E ∪ (g ⇒ c)
25:          end if
26:          A+ := A+ ∪ (g ⇒ c)
27:          A− := A− \ (g ⇒ c)
28:       end if
29:    end for
30: until ¬Expanded
31: if BranchedAsks≠ ∅ then
32:    g := GetGuard(GetElement(BranchedAsks))
33:    (Str, A−tr, A+tr, Etr, CPtr) :=
34:      CalculateClosure(S ∧ g,
35:       M, CP, E, CheckConsistency)
36:    (Sfa, A−fa, A+fa, Efa, CPfa) :=
37:      CalculateClosure(S ∧ ¬g,
38:       M, CP, E, CheckConsistency)
39:    if Str ≠ false ∧ Sfa ≠ false then
40:       return GetElement({
             (Str, Etr, CPtr, A−tr, A+tr),
             (Sfa, Efa, CPfa, A−fa, A+fa)})
41:    else if Str ≠ false then
42:       return (Str, Etr, CPtr, A−tr, A+tr)
43:    else if Sfa ≠ false then
44:       return (Sfa, Efa, CPfa, A−fa, A+fa)
45:    else
46:       return (false, ∅, CP, ∅, ∅)
47:    end if
48: end if
49: return (S, E, CP, A−, A+)
```

図 3.5 CalculateClosure のアルゴリズム

第 4 章

HydLa の記号実行シミュレータ Hyrose

本章では，HydLa の記号実行シミュレータである Hyrose について，その概要や設計について述べる．

4.1 Hyrose の概要

HydLa のシミュレータである Hyrose は，HydLa プログラムをシミュレーション実行することで，ハイブリッドシステムの性質の理解や検証に役立てることを目標として開発された．

Hyrose は 3 章に示したアルゴリズムを採用しており，全体の実装は約 25700 行の C++ コードからなっている．処理系開発プロジェクトに携わった人数は 12 人であり，Trac と Subversion を利用してチーム開発を行ってきた．

ハイブリッドシステムの性質検証のため，Hyrose は 3 章のシミュレーションアルゴリズムに従った基本的なシミュレーションに加え，プログラムが許すすべての解軌道をシミュレーションする全解探索機能（4.3 節）と，求めた解軌道が指定された条件を満たすかを調べる有界モデル検査機能（4.4 節）を有している．さらに，Hyrose が求める解軌道は数式処理によって誤差を含まないものになっており，シミュレーション結果の信頼性を保証している（4.2 節）．

Hyrose は制約求解のバックエンドとして，数式処理ソフトウェアである Mathematica [17] と REDUCE [2] のいずれか一つを利用する．このため，C++ コードの一部に Mathematica コードや REDUCE コードを文字列として含んでいる．以降，本論文では

特に断りが無い限り，Mathematica を利用した場合について扱うものとする．

Hyrose を用いて図 2.1 プログラムを実行した場合の出力例を図 4.1 に示す．シミュレーション終了時刻は 3 とした．この例は天井に衝突する場合に相当しており，最下部における「parameter condition」が，この場合の記号定数 parameter[y，0，1] と parameter[y，2，5] の条件を示している．parameter[y，0，1] とは，「y」の「0」階微分（つまり y そのもの）の第「1」フェーズにおける値を表す記号定数を意味する．(10, 11] は，10 < parameter[y，0，1] ≤ 11 を表している．parameter[y，2，5] についても同様であり，(-inf，inf) の inf とは ∞ を意味する，

4.2 精度保証

ハイブリッドシステムは離散変化を伴うため，計算結果の誤差によって本来のシステムの振舞いとは異なる，定性的に誤った（3.1 節）結果を求めてしまう可能性がある．このため，処理系開発においては計算精度を保証することが重要な課題であった．精度を保証する方法には，数式処理の他に区間演算 [8]（精度保証数値計算）もあり，Hyrose 開発プロジェクトの中では区間演算によるアプローチも試みられてきたが，今回は数式処理によるシミュレーションを優先採用して実装を行った．なお，無理数を含む数式の大小比較などのごく小さな処理単位では，数値近似および区間演算も補助的に使用している．

数式処理を用いる利点は，以下が挙げられる．

1. 計算結果の誤差が発生しないため，正確な結果を求められる（区間演算の場合，区間幅が爆発して結果の有用性がなくなる可能性がある）
2. 誤差が発生しないため，アルゴリズムの動作を論理的に確認しやすい．
3. パラメータを含むシステムの記号実行を行うことができる．
4. 区間演算は本質的に不等式制約の対に対する演算であるが，不等式制約自体は数式処理でも対処可能である．このため，数式処理に基づく処理系への区間演算の導入は，その逆と比較してスムーズだと期待される．

特に，2 番目の利点は Hyrose の開発目的にも合致していた．また，HydLa 言語は制約を用いてパラメータを含むシステムを自然に記述できるという特徴を持つ．この特徴を生かすためにも，3 番目は重要な利点だと考えられる．

逆に，数式処理が区間演算に比べて難しい点としては以下が挙げられる．

1. 数式に対応するデータ構造は，浮動小数点数の区間の表現に比べてやや複雑である．

```
#---------Case 1---------
#---------1---------
---------PP---------
time    : 0
y       : parameter[y, 0, 1]
y'      : 10
y''     : -10

---------IP---------
time    : 0->1+(-1)*(-2+
          parameter[y, 0, 1]*1/5)^(1/2)
y       : t*(t+(-2))*
          (-5)+parameter[y, 0, 1]
y'      : (t+(-1))*(-10)
y''     : -10

#---------2---------
---------PP---------
time    : 1+(-1)*(-2+
          parameter[y, 0, 1]*1/5)^(1/2)
y       : 15
y'      : (-8)*(-2+
  parameter[y, 0, 1]*1/5)^(1/2)
y''     : parameter[y, 2, 5]

---------IP---------
time    : 1+(-1)*(-2+parameter[y, 0, 1]*
          1/5)^(1/2)->3
y       : 36+10*t+t^2*(-5)+18*(-2+
          parameter[y, 0, 1]*1/5)^(1/2)+
          t*(-18)*(-2+parameter[y, 0, 1]
          *1/5)^(1/2)+parameter[y, 0, 1]*
          (-13)/5
y'      : 10+t*(-10)+(-18)*(-2+
          parameter[y, 0, 1]*1/5)^(1/2)
y''     : -10


#---------parameter condition---------
parameter[y, 0, 1]      : (10, 11]
parameter[y, 2, 5]      : (-inf, inf)
# time ended
```

図 4.1 Mathematica を用いた解軌道の出力例

2. 離散変化を重ねることで，数式の項の数および係数の有理数表現が増大し，処理コストが爆発する可能性がある．
3. 解析的に解くことのできない微分方程式に対応できない．
4. 数式の簡約を適切に行わないと，処理速度に悪影響が出る．

2. や 3. のように，数式処理だけでは対処できない問題が存在する．そこで数式処理の枠組の中で区間演算を行うプロトタイピングを経て，将来的に数式処理技術と高速区間演算技術を連携させたシミュレータを実装することを考えている．

4.3 全解探索機能

全解探索機能は，HydLa プログラムが許す解軌道をすべて求める機能であり，実行時オプション「`--nd`」を指定することで利用できる．パラメータを含むシステムに対し，パラメータの値によるシステムの振舞いの変化を調べるなどの用途が考えられる．また，4.4 節で紹介する有界モデル検査機能と組み合わせて，解軌道が特定の条件を満たすためのパラメータ値の範囲を求めることにも応用できる．この機能を実現するため，Hyrose は未展開のフェーズを記録するコンテナを用意して深さ優先もしくは幅優先探索を行うことで，すべての場合についてシミュレーションを行っている．

4.4 有界モデル検査機能

有界モデル検査機能は，シミュレーションによって求めた解軌道が指定された条件を満たすかを調べる機能である．プログラム内に「`ASSERT(x > 0)`」のようにシステムの満たすべき条件を記述し，シミュレータを実行することで利用できる．条件に違反した場合はその旨を出力し，条件に違反するまでにシステムがどのような振舞いをしていたかをあわせて出力する．もし記号定数の値によって条件に違反するかが変化する場合は，違反する場合と違反しない場合とに分け，違反しない場合についてはシミュレーションを続行する．この機能を利用することで，有限時間内のシステムの振舞いを検証できるほか，ある状態への到達可能性を調べるために，その状態の条件の否定を記述してシミュレーションを行う，といった利用法も考えられる．

第 5 章

Hyrose の実装

本章では，4 章で述べた Hyrose の実装および，それに付随して必要とされた事項について述べる．

5.1 処理系の構成

Hyrose の構成図を図 5.1 に示す．Parser，ConstraintHierarchySolver，Simulator，PhaseSimulator，VirtualConstraintSolver はそれぞれ独立したモジュールとして設計しており，実装の入れ替えや拡張を容易なものとしている．また，Mathematica と REDUCE という大きく異なるバックエンドの統一的な利用を目指した点も特徴である．

各クラスの役割としては，まず Parser で HydLa プログラムを構文解析木に変換し，それを元に ConstraintHierarchySolver でプログラムに記述された半順序集合の優先順位を満たす制約モジュール集合（解候補モジュール集合）の集合を求める．この解候補モジュール集合の集合を元に，Simulator 以下のクラスが軌道の求解を行う．SequentialSimulator と InteractiveSimulator はそれぞれ Simulator クラスの実装であり，前者は逐次的に指定された終了条件を満たすまでシミュレーションを続けるのに対し，後者はユーザの入力を対話的に受けつけながらシミュレーションを行う．Simulator クラスは各フェーズのシミュレーションに PhaseSimulator クラスを利用する．SymbolicPhaseSimulator は PhaseSimulator の数式処理を用いた実装であり，制約求解などの処理に SymbolicVirtualConstraintSolver を用いる．MathematicaConstraintSolver と REDUCEConstraintSolver は SymbolicVirtualConstraintSolver の実装であり，それぞれ対応する数式処理ソフトウェアを利用している．これら 2 つの ConstraintSolver のコードはその大部分が Mathematica や REDUCE のコードを表現するテキストからなっ

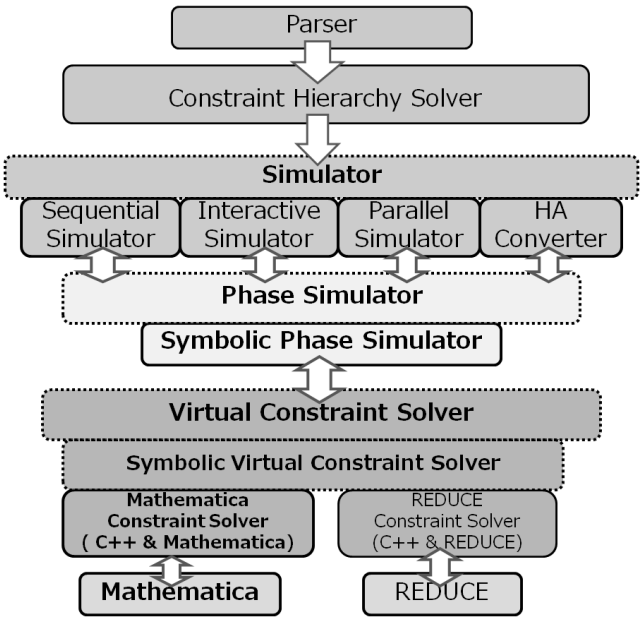

図 5.1　Hyrose 構成図（太字は本研究での主要部分）

ている．

5.2　解軌道の表現

Hyrose はシミュレーションの結果である解軌道を表現するために，フェーズを節点とする木構造を作成する．図 5.2 に，木構造の例を示す．各節点は，図 3.1 における PP や IP 終了時の制約ストアなどの情報を保持しており，上から PP か IP かの種別，フェーズの時刻，変数値制約，記号定数の条件となっている．変数の値は記号定数を用いてすべて等式で表現されることに注意されたい．なお，この例では変数や記号定数に関する制約のうち，特に重要なもののみを表示している．また，展開済み always 制約については，今回の例には出現しないため省略している．図中の root は，解軌道木の根を意味しており，どのフェーズにも対応していない．最初の PP が根にならないのは，プログラムによっては最初の PP の時点ですでに複数通りに分岐する可能性があるためである．

解軌道木を，根から葉に辿って行く経路 1 本 1 本が，それぞれ定性的に異なる解軌道を表現している．各経路の葉が持つ記号定数の条件が，それぞれの軌道に至るための条件となっている．図 5.2 の例では，記号定数 y_0 が $[9, 10)$ の場合と，10 の場合と，$(10, 11]$ の場合との 3 通りに解軌道が分類されていることが分かる．

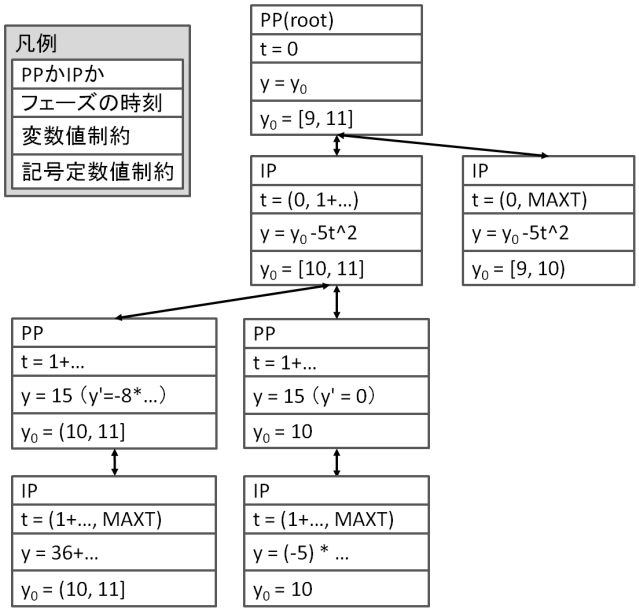

図 5.2　解軌道木の例

5.3　暗黙的な連続性

HydLa の宣言的意味論では，変数の時刻に関する微分値に関する制約が存在する場合は，暗に当該変数の左連続性と右連続性を仮定するものとされている．それらの連続性は微分値に関する制約自体とは独立に，1 つ上の優先度を持ったものと仮定することになっているが，これをそのまま実装すると，状態変数の個数の増加に伴って解候補モジュール集合の数が爆発するため，暗黙の連続性制約については実装上特別に扱うことが必要と判断した．現在の Hyrose では連続性の仮定を簡単化し，「制約ストア，もしくはアルゴリズム中で判定対象となるガード条件の後件に変数 x に関する微分制約（$x', x'', \ldots$ に関する制約）が出現する場合，直前のフェーズからの x の連続性を仮定する」ものとして実装している．このように実装したことで，今まで実験してきた例題プログラム（6 章）からは，宣言的意味論との差異を生むようなプログラムは見つかっていない．

5.4　時刻 0 の左極限値に関するガード条件

Hyrose のシミュレーションにおいて，正の時刻における各変数の左極限値はそこまでのシミュレーション結果から明らかである．しかし Hyrose のシミュレーションは時刻 0

を始点としているため，時刻 0 における左極限値は定義されない．このため，時刻 0 の PP において左極限値を含むガード条件の判定をどのように扱うかが実装上の問題となった．解決にあたっては，以下の 4 つの案の長所と短所を検討した．

1. 常に偽とする．— 実装は容易であり，確認した例題の範囲ではプログラムの直感的な意味を反映している．
2. 常に真とする．— 実装は容易だが，多くの例題の動作が想定外のものになる．例えば図 2.1 のプログラムについて，時刻 0 で INIT と BOUNCE が矛盾し，解軌道が存在しないことになる．
3. 制約が時刻 0 から負の近傍へ遡及することを許し，判定する． — いくつかの例題について望ましい動作をするが，言語設計の基本方針であり実装の前提ともなっている「制約は過去の値に影響しない」という性質と矛盾する．
4. 左極限値を完全に不定なものとして判定する． — 解軌道群の中に必ず正しいものを含むことにはなるが，その分だけ冗長な場合分けが発生しやすい．

以上のことから，現在は「常に偽とする」案を実装している．

第 6 章

例題による動作確認

本章では，今回実装した Hyrose を用いて例題に対するシミュレーションおよび検証を行った結果について論じる．なお，本章に掲載しているグラフは見やすさのため有限個の解軌道を描画したものになっているが，Hyrose の出力自体は記号定数を用いることで無限個の解軌道を表現したものになっている．なお，本章では，7 章で述べる最適化を施していない Hyrose を使用している．

6.1 溝のある地面を跳ねる質点

最初に，2 次元空間上で質点を投げて目標地点へ到達させる例題を考える．モデルの概要を図 6.1 に示す．

- 水平軸 x と垂直軸 y からなる 2 次元空間の中で，$\mathrm{x} = 0, \mathrm{y} = 10$ の位置から質点を $\mathrm{x} > 0$ の方向に投げる．
- x 方向への初速度は 20 以下の正の値．
- y 方向への初速度は 0.
- 空気抵抗は無視し，質点には重力により下方へ -10 の加速度が働く．
- 高さ $\mathrm{y} = 0$ に地面があり，途中に溝が掘られている．
- 溝は深さが 7，幅が 3，左端が 7 の長方形である．
- 質点は溝の中の壁と反発係数 1 で衝突し，地面および溝の底と反発係数 4/5 で衝突する．

このモデルにおいて，質点が溝の向こうの地面に達するためには，初期速度がどのような条件を満たす必要があるかを Hyrose によって求める．

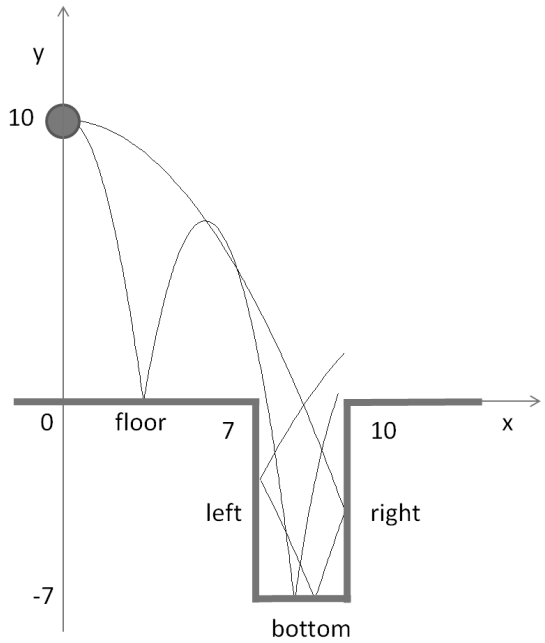

図 6.1 溝のある地面を跳ねる質点の図

HydLa プログラムを図 6.2 に示す．図中の x が質点の左右方向の位置を，y が上下方向の位置を表すものとする．INIT はモデルの初期状態を表している，FALL と BOUNCE が上下方向の運動に，XCONST と XBOUNCE が左右方向の運動に対応する制約になっている．ASSERT の中に記述されている条件は，目標状態が満たすべき条件の否定を表しており，こうすることで初期速度の条件を Hyrose によって求めることができる．

このプログラムを，時刻 20，離散変化の回数を 6 回までという条件で Hyrose を用いてシミュレーションしたところ，溝の向こうに質点が到達するための x' の初期値 $\mathtt{x'}_0$ の範囲は次のいずれかであることが分かった．

1. $1250/(405\sqrt{2}+\sqrt{317}+9\sqrt{1387}) \leq \mathtt{x'}_0 \leq 1250/(405\sqrt{2}-\sqrt{317}+9\sqrt{1387})$
2. $125\sqrt{2}/97 \leq \mathtt{x'}_0 \leq 35/(13\sqrt{2})$
3. $35/(13\sqrt{2}) < \mathtt{x'}_0 \leq (1125\sqrt{2}+225\sqrt{67}+25\sqrt{197})/(928+81\sqrt{134})$
4. $-40\sqrt{197}/(928+81\sqrt{134})+360(5\sqrt{2}+\sqrt{67})/(928+81\sqrt{134}) \leq \mathtt{x'}_0 < 25\sqrt{2}/13$
5. $25\sqrt{2}/13 \leq \mathtt{x'}_0 \leq 7/\sqrt{2}$
6. $(117\sqrt{85}+13\sqrt{485})/256 < \mathtt{x'}_0 < 10\sqrt{5/17}$
7. $10\sqrt{5/17} \leq \mathtt{x'}_0 \leq 10\sqrt{5/17}$
8. $10\sqrt{5/17} < \mathtt{x'}_0 \leq (9\sqrt{85}+\sqrt{485})/16$

```
INIT <=> y = 10 /\ y' = 0
  /\ x = 0 /\ 0 < x' <= 20.
FALL <=> [](y'' = -10).
XCONST <=> [](x'' = 0).
XBOUNCE <=> []((x- = 7 \/ x- = 10)
  /\ y- < 0 => x' = -x'-).
BOUNCE <=> [](y- = -7 \/
  (x- <= 7 \/ x- >= 10) /\ y- = 0
  => y' = -(4/5) * y'-).
ASSERT(!(y >= 0 /\ x >= 10)).
INIT, FALL << BOUNCE, XCONST << XBOUNCE.
```

図 6.2 溝のある地面を跳ねる質点のプログラム

9. $5\sqrt{2} \leq \mathtt{x'}_0 \leq 20$

個々の場合を数値で表し，定性的挙動（目標状態に至るまでの衝突系列）を示すと

1. $[1.36027, 1.40428]$, (floor,floor,bottom)
2. $[1.82244, 1.90375]$ (floor,floor)
3. $(1.90375, 2.02803]$ (floor,bottom)
4. $[2.643, 2.71964)$ (floor,right,bottom,left)
5. $[2.71964, 4.94975]$ (floor)
6. $(5.33196, 5.42326)$ (bottom,right,left)
7. $[5.42326, 5.42326]$ (bottom+right,left)
8. $(5.42326, 6.56241]$ (right,bottom,left)
9. $[7.07107, 20]$ ()

となる．

このシミュレーションにおいて，Hyrose は 50 通りの場合分けを行い，出力は 7234 行におよんだ．

質点の初期速度に加えて，左壁の位置も区間制約で与える実験も行ったが，左壁の位置を幅 1 の区間制約 `6<=edge<=7` に変更するだけで，探索空間が大幅に増大するのに加え，特定の初期速度において解の表現に 4 次方程式の根が出現して計算時間が大幅に増加する現象が観察された．このことから，複数のパラメータをもつモデルのシミュレーションは，現在の HydLa でも原理的に可能であるものの，パラメータ空間を十分小さく分割した複数のモデルを用いて求解する必要があることが判明した．

6.2 電気回路

次に，図 6.3 に示す電気回路のシミュレーション結果を紹介する．回路にはスイッチがあり，スイッチが s_1 と s_2 のどちらの位置にあるかによって回路の振舞いが変化する．今回は，容量 C のコンデンサにかかる電圧 v_C と，インダクタンス L のコイルに流れる電流 i_L に着目する．スイッチが s_1 にあるとき，v_C と i_L はそれぞれ次の微分方程式に従う．

- $\frac{d}{dt}i_L = -\frac{R_L}{L}i_L + \frac{1}{L}v_S$
- $\frac{d}{dt}v_C = -\frac{1}{C}\frac{1}{R_C + R_0}v_C$

同様に s_2 にあるときは次の式に従う．

- $\frac{d}{dt}i_L = -\frac{1}{L}(R_L + \frac{R_C R_0}{R_C + R_0})i_L - \frac{1}{L}\frac{R_0}{R_C + R_0}v_C + \frac{1}{L}v_S$
- $\frac{d}{dt}v_C = -\frac{1}{C}\frac{R_0}{R_C + R_0}i_L - \frac{1}{C}\frac{1}{R_C + R_0}v_C$

これらの式を元に，$L = 1, R_L = 1, v_S = 5, R_C = 1, R_0 = 1, C = 1$ とし，i_L の初期値を 0，v_C の初期値を 0 以上 5 以下とした場合の HydLa プログラムを図 6.5 に示す．このプログラムではタイマーを設け，タイマーが 1 になるごとにスイッチを切り替え，タイマーを 0 にリセットするようにしている．プログラム内の `INIT` はシステムの初期条件に，`TIMER` はタイマーの進行に，`SWITCH` はスイッチの切り替えに，`STATE1` はスイッチが s_1 にあるときの振る舞いに, `STATE2` は s_2 の場合にそれぞれ対応している．

このプログラムに対する Hyrose の出力をグラフ化したものを，図 6.5 と図 6.6 に示す．一本一本の線は，v_C を 0 から 5 まで 0.5 ずつ変化させた場合の各軌道を表している．図 6.5 を見ると，時刻が経過するにつれて v_C の初期値による振舞いの差は少なくなり，ある一定の振舞いに収束していくことが分かる．一方図 6.6 では，最初にスイッチを切り替えた時点から v_C の初期値による影響を僅かに受けるが，その後 v_C の振舞いに合わせて再び収束していくことが分かる．以上のことから，このモデルは十分な時間がたてば，v_C の初期値に関わらず一定の振舞いをするであろうことが推測できる．

このプログラムについて，実際に Hyrose のシミュレーションにかかった時間を表 6.1 に示す．比較例として，初期電圧 $\mathtt{vc}_0 = 2$ とした場合の実験結果も付した．実験環境は，

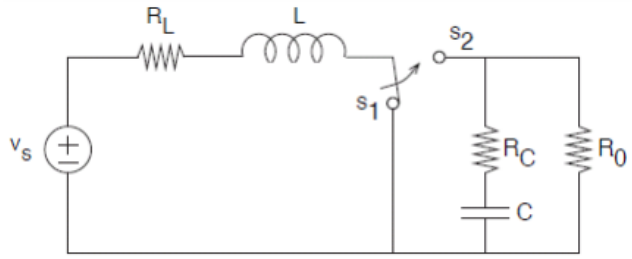

図 6.3　電気回路（文献 [10] より引用）

```
INIT <=> 0 <= vc <= 5 /\ il = 0 /\
  s = 0 /\ timer=0.
TIMER <=> [](s' = 0 /\ timer' = 1).
SWITCH <=> [](timer- = 1 => timer = 0 /\
  s = 1 - s-).
STATE1 <=> [](s- = 0 => il' = -il + 5 /\
  vc' = -1/2 * vc).
STATE2 <=> [](s- = 1 => il' = -3/2 *
  il - 1/2 * vc + 5 /\
  vc' = 1/2 * il - 1/2 * vc).

INIT, TIMER << SWITCH, STATE1, STATE2.
```

図 6.4　電気回路のプログラム

OS: Debian (etch)，CPU: Quad-Core AMD Opteron(tm) 2.3GHz，メモリ: 16Gbyte，となっている．

6.3　カーリング

最後にシステムの制御を行う例題として，カーリングのストーンの動きを簡単化したモデルを扱う．ストーンは 1 つだけとし，衝突は考えない．ゲームの目的を，「スタート地点から目標範囲内（距離が 9〜11 の間）に止める」こととする．空間は一次元とし，初期位置を 0 として，初期速度を 1 とする．投球後，何もしないでいると摩擦によってストーンは $-1/10$ の加速度を受けるものとし，スウィーピングを行った場合は摩擦が減少する．摩擦によって速度が負になることはない．スウィーパーは目標地点に到達する前に速度がある値を下回ったらスウィーピングを開始する．この時，スウィーピングの開始速度やスウィーピング時の加速度を区間値とし，ストーンを適切な位置に止めるための条件を調べる．

以上を表す HydLa プログラムを，図 6.7 に示す．図中の y がストーンの位置を表して

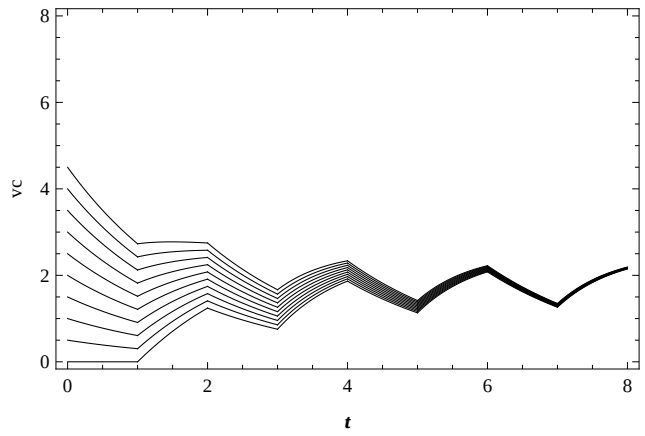

図 6.5 電圧 vc の振舞い

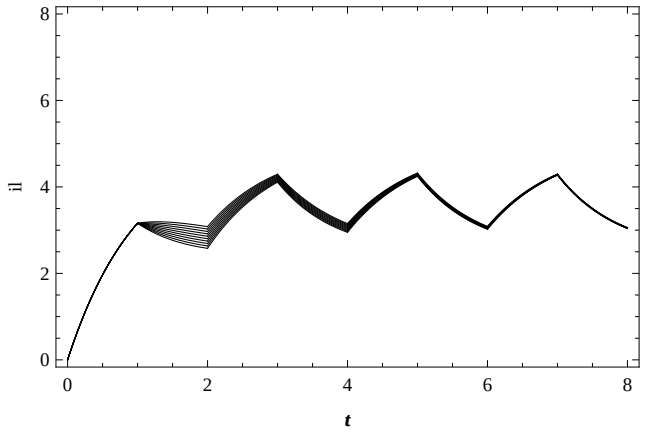

図 6.6 電流 il の振舞い

おり，INIT がモデルの初期状態，FRICTION が通常時の摩擦に，SWEEPING がスウィーパーの判断とスウィーピング中の摩擦にそれぞれ対応している．border と fric はそれぞれ定数であり，border はスウィーパーがスウィーピングを開始する速度を，fric はスウィーピングを行っている間のストーンの加速度を表している．このプログラムを Hyrose でシミュレーションした結果の解軌道群を，図 6.8 と図 6.9 に示す．

border に幅を持たせた場合，求める条件は $2/\sqrt{15} \leq \mathtt{border} \leq 2/\sqrt{5}$ であることが分かった．この条件は概算すると，$0.516 \leq \mathtt{border} \leq 0.894$ である．この条件を満たす場合，ストーンは $8 + 15\mathtt{border}^2/4$ で停止する．$\mathtt{border} < 2/\sqrt{15}$ の場合，スウィーピングを行っても $\mathtt{y} = 5 + 15\mathtt{border}^2$ の地点でストーンは停止してしまい，目標範囲には届かない．一方 $2/\sqrt{5} < \mathtt{border}$ の場合，目標範囲を通り過ぎた後 $\mathtt{y} = 8 + 15\mathtt{border}^2/4$ で停止する．

表 6.1 電気回路のプログラムの実行時間（左が $0 \leq \mathtt{vc}_0 \leq 5$，右が $\mathtt{vc}_0 = 2$）

指定時刻	実行時間 (s)
1	0.803
2	1.707
3	2.414
4	3.514
5	4.746
6	6.248
7	7.614
8	9.485
9	11.096
10	13.373

指定時刻	実行時間 (s)
1	0.779
2	1.540
3	2.127
4	2.956
5	3.691
6	4.619
7	5.661
8	6.855
9	7.888
10	9.397

```
INIT <=> y = 0 /\ y' = 1 /\
  0 < border < 1 /\  [](border' = 0) /\
  -1/20 < fric < -1/100 /\ [](fric' = 0).
FRICTION <=> [](y' > 0 => y'' = -1/10) /\
  (y' <= 0 => y'' = 0)).

SWEEPING <=> [](y < 9 /\ 0 < y' < border
  => y'' = -1/40).
    // 開始する速度を区間値とする場合

/*
SWEEPING <=> [](y < 9 /\ 0 < y' < 3/4
  => y'' = fric).
*/ // 加速度を区間値とする場合


INIT, FRICTION << SWEEPING.

ASSERT(y' != 0 \/ 9 <= y <= 11).
```

図 6.7 カーリングのプログラム

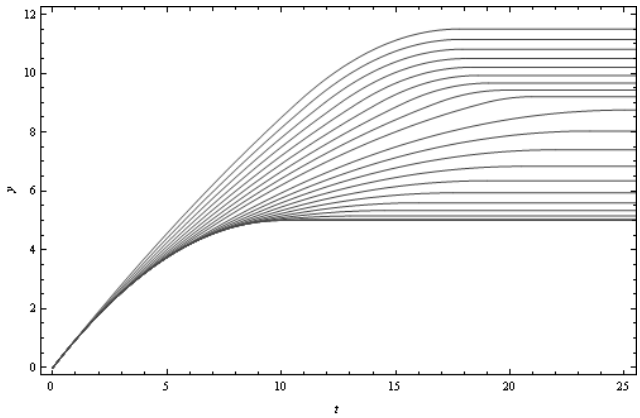

図 6.8　開始する速度を区間値とした場合のストーンの軌道

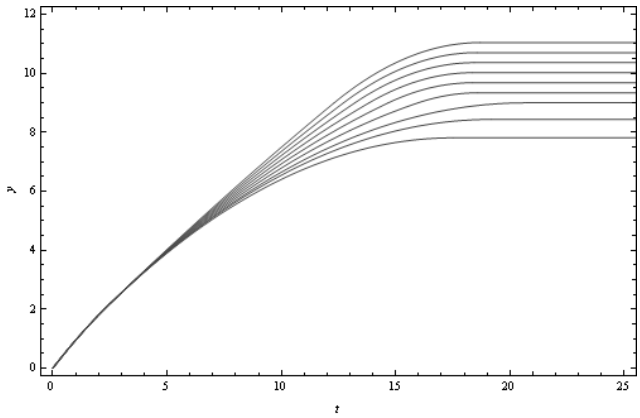

図 6.9　加速度を区間値とした場合のストーンの軌道

一方，fric に幅を持たせた場合，求める条件は $-9/218 \leq \texttt{fric} \leq -13/1090$ であることが分かった．この条件は概算すると，$-0.041 \leq \texttt{fric} \leq -0.0119266$ である．この条件の範囲内では，ストーンは $\texttt{y} = 189/16 + 545\texttt{fric}/8$ で停止する．$\texttt{fric} < -9/218$ の場合，$\texttt{y} = (70 - 9/\texttt{fric})/32\texttt{fric}$ の地点で停止してしまい，目標範囲には届かない．$-13/1090 < \texttt{fric}$ の場合，目標範囲を通り過ぎた後 $\texttt{y} = 189/16 + 545\texttt{fric}/8$ で停止する．

6.4　その他の例題

ここで紹介した例題以外にも，これまで 32 個の例題を用いて処理系の動作を確認してきた．そのうちのいくつかを表 6.2 に示す．本論文で詳述した例題はいずれもパラメータを 1 つに限定してシミュレーションを行ったが，これまで実験してきた中にはパラメータ

表 6.2　その他の例題による実験結果

例題の概要	結果
質点が地面で跳ねる	初期高さ，初期速度，重力加速度の 3 つを パラメータとして与えてもシミュレーシ ョン可能．61 通りの場合分けを確認 （時間内に地面と衝突する回数による場合分け）
複数の質点が跳ねる	質点を 10 個まで増やしてもシミュレーション可能.
インパルス関数	微分方程式によらない軌道もシミュレーション可能
燃料タンクを交換しながら 飛行する熱気球 [14] （飛行を続けるための条件を求める）	燃料タンクの容量と，交換にかかる時間の 2 つをパラメータとしてシミュレーション を行うことができた．特定の条件を満たす ためのパラメータの関係を得ることができた. 確認した範囲で，40 通りの場合分けを行った

を複数含むものも含まれている．しかし複数のパラメータを持つプログラムのシミュレーションは現在採用している数式処理ソフトウェアの求解能力とその利用法の下では容易ではなく，数式の複雑化にともない制約ソルバが応答しなくなるケースも単数パラメータの場合に比べて増加する．特にそれぞれのパラメータが離散変化条件に複雑に影響するようなプログラムのシミュレーション能力の向上は今後の課題となっている.

第 7 章

Hyrose の実行性能の最適化

本章では，ここまでの章で説明した Hyrose について，更に実行性能の最適化を行ったので，その最適化について論じる．

7.1 CalculateClosure 内のガード条件判定回数削減

Hyrose を用いて例題のシミュレーションを行う中で，ガード条件判定に時間がかかる場合が多くあった．表 7.1 に，各例題について，シミュレーション全体の実行時間に占めるガード条件導出判定時間の割合を示す．これらの例題では，全体時間の 60%〜70% をガード条件導出判定が占めていることが分かる．このため，導出判定の回数を削減することができれば，実行時間の削減に直結すると考え，以下の 2 つの改良案を提案・実装した．

表 7.1　例題実行時のガード条件導出判定時間の割合

例題の概要	ガード条件導出判定時間の割合（%）
熱気球	60.5
溝と質点	63.0
複数の質点	70.2
円盤の衝突	66.1
磁力の切り替えと質点	70.2

```
 1: A_- := ∅; A_+ := E
 2: repeat
 3:    S :=CollectTell(M, A_+, S_prev);
 4:    (TF, CP) :=
 5:    CheckConsistency(S, CP)
 6:    if TF = false then
 7:       return (false, ∅, CP, ∅, ∅)
 8:    end if
 9:    Expanded := false
10:    BranchedAsks := CollectAsk(M, A_+, A_-);
11:    for (g ⇒ c) ∈ BranchedAsks do
12:       (TF, CP) :=
13:       CheckConsistency(S ∧ g, CP)
14:       if TF ≠ false then
15:          (TF, CP) :=
16:          CheckConsistency(S ∧ ¬g, CP)
17:          if TF ≠ false then
18:             continue
19:          end if
20:          Expanded := true
21:          if IsAlways(c)) then
22:             E := E ∪ (g ⇒ c)
23:          end if
24:          A_+ := A_+ ∪ (g ⇒ c)
25:       end if
26:       A_- := A_- ∪ (g ⇒ c)
27:    end for
28: until ¬Expanded
29: （以降，図 3.5 と同様）
```

図 7.1 CalculateClosure の改善版アルゴリズム

7.1.1 制約増加の単調性の利用

3.3.4 節で説明したように，*CalculateClosure* 関数内では制約ストア内の制約は単調に増加するだけで，減少することはない．このことから，ガード条件の導出判定において「現在の制約ストアと矛盾する」と判断された条件は，*CalculateClosure* 関数が終了するまで制約ストア内の制約と矛盾し続けることがわかる．しかし現在のアルゴリズムでは，「導出される」と判明した条件以外はすべて *CalculateClosure* の 1 ループごとに計算し直しているので，最大で（(ループ回数 − 1) ×（矛盾すると判断されたガード条件））回だけ無駄なガード条件判定を行っていることになる．この無駄な判定を削減した，*CalculateClosure* 関数のアルゴリズムを図 7.1 に示す．なお，図には変更の対象となる部分のみを示す．

改善前の図 3.5 のアルゴリズムでは，「まだ導出されていないガード条件の集合」であ

る $A\_$ に対して導出判定の for ループを回していたのに対し，図 7.1 のアルゴリズムでは，「論理的に導出されないが矛盾もしない条件」である *BranchedAsks* に対してのみガード条件導出判定を行う．なお改善後のアルゴリズムにおける $A\_$ の直感的な意味は，「制約ストアと矛盾することが判明したガード条件」に変化している．

7.1.2 左極限値に関するガード条件の利用

3 章のアルゴリズムでは，*CalculateMCS* 関数中で *CalculateClosure* 関数は解候補モジュール集合ごとに独立に呼び出され，計算結果は共有されない．一方で，各 PP における変数の左極限値は直前の IP で確定しており，採用するモジュール集合と無関係に定まる．ここから，「左極限値のみに言及し，現在時刻の変数値に影響されないガード条件」の判定は 1 つの PP に 1 回だけ行い，あとはその結果を流用すれば良いことがわかる．これを実現するため，プログラム中から左極限値に関する条件のみを抜き出し，それらの判定結果を 1 度の *CalculateMCS* 内では再利用することで，ガード条件導出判定回数の削減を行った．

7.1.3 ガード条件削減の評価

図 7.2 に，以上の 2 つの最適化前後を行ったことによるガード条件導出判定時間を示す．この図を見ると，どの例題についてもガード条件導出判定の時間を削減できたことが分かる．特に効果が顕著な複数の質点の例題については，判定時間を 90% 以上削減することに成功した．

7.2 制約の依存関係を用いた最適化

図 7.3 のようなプログラムを考える．このプログラムは，複数の質点が高さ 10 から同時に落下を始め，高さ 0 にある地面と衝突し，反発係数 1 で跳ねあがるモデルを記述したものである．質点の高さに対応する変数は，それぞれ `y1`，`y2`，`y3` ... となっている．このモデルでは，質点同士の運動は完全に独立であり，互いに衝突したりはしないものとする．

このプログラムの質点の数を 1 個〜10 個に増やしながら現在の Hyrose で 3 フェーズ目まで実行した各場合の実行時間を表 7.2 に示す．表 7.2 を見ると，質点の数に対して実行時間が指数関数的に増加していくことがわかる．この理由として，このプログラムの解候補モジュール集合の要素数は，2^n（n は質点の数）であることとが挙げられる．このプ

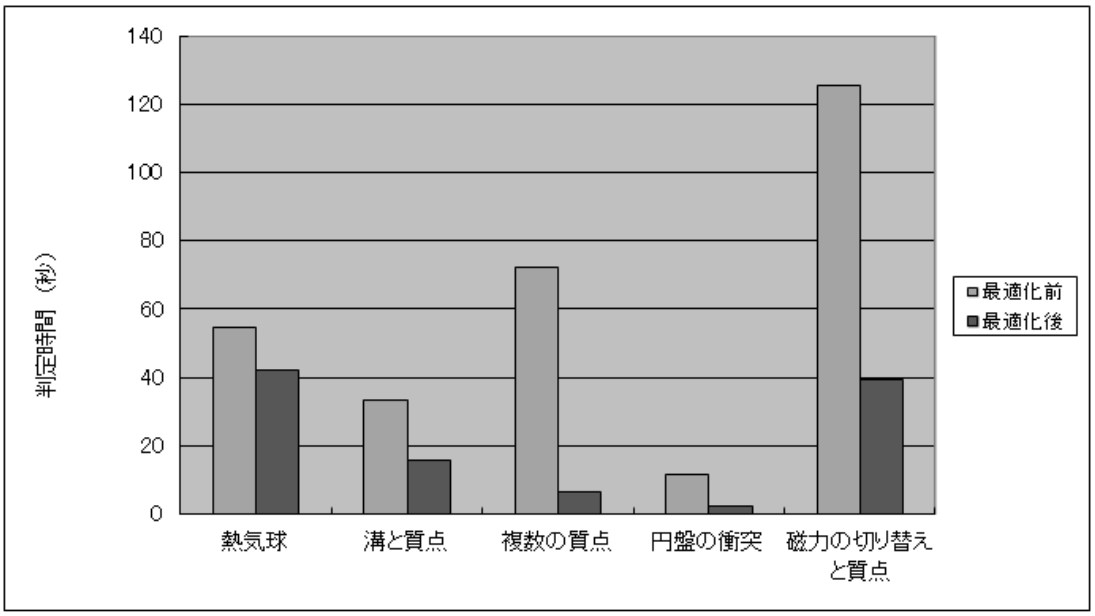

図 7.2　最適化前後のガード条件導出判定時間

```
INIT(y) <=> y = 10 & y' = 0.
FALL(y) <=> [](y'' = -10).
BOUNCE(y) <=> [](y- = 0 => y' = -y'-).

BALL(yarg){INIT(yarg), FALL(yarg) << BOUNCE(yarg)}.

BALL(y1), BALL(y2), BALL(y3), BALL(y4), BALL(y5),
 BALL(y6), BALL(y7), BALL(y8), BALL(y9), BALL(y10).
```

図 7.3　複数の質点が独立に跳ねるプログラム

ログラムでは全ての質点が同時に地面に衝突するため，衝突の時刻においては最大のモジュール集合から最小のモジュール集合まで，全ての解候補モジュール集合 1 つ 1 つに対して計算行われ，全体のシミュレーション時間も解候補モジュール集合の要素数に比例して増加してしまうことになる．しかし少なくともこのプログラムに関しては，質点は互いに独立に動くので，計算量のオーダーは，$O(n)$ であることが望ましい．このためには，Hyrose 自身が「それぞれの質点に関する制約モジュールは互いに矛盾しない」という情報を利用するようにすればよい．

表 7.2　複数の質点が独立に跳ねる例題の実行結果

質点の数	実行時間 (s)
1	0.141
2	0.234
3	0.453
4	1.031
5	2.625

質点の数	実行時間 (s)
6	6.906
7	17.703
8	46.234
9	116.016
10	284.500

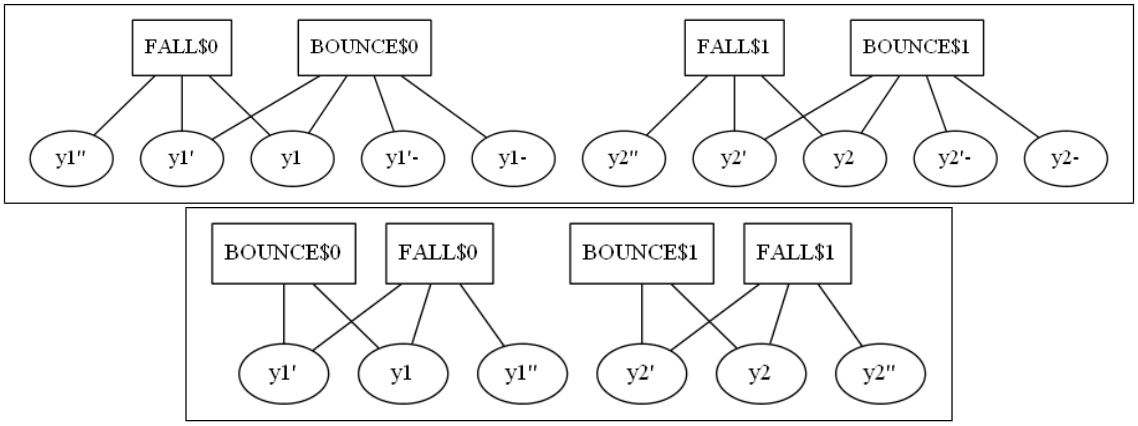

図 7.4　図 7.3 の依存関係グラフ（上が PP，下が IP）

7.2.1　最適化のためのアルゴリズム

制約モジュール同士が互いに矛盾しないための十分条件は，「共通の変数が出現しないこと」である．このため，まずは各制約モジュールに出現する変数およびその左極限値を調べ，それぞれをノードとする 2 部グラフを作成する．以降，この 2 部グラフのことを依存関係グラフと呼ぶ．

図 7.4 に図 7.3 のプログラムに対する依存関係グラフを示す．IP では PP と異なり，変数自体の値と左極限値とは常に一致するため，依存関係グラフは IP と PP で異なるものになる．図中で，枠が長方形になっているノードが制約モジュールに，枠が楕円形のノードが変数に対応している．制約モジュール名の末尾に付加されている”$0” といった文字列は，同名のモジュールの中での出現順を表している．

Input: 制約ストア S, 解候補モジュール集合のリスト MS, 展開済み always 制約の集合 E, 記号定数に関する条件 CP, 時刻 T, 無矛盾性判定関数 $CheckConsistency$, 制約間の依存関係グラフ R
Output: 制約ストア, 記号定数に関する条件, 展開済み always 制約の集合, 極大となるモジュール集合, 成り立たないガード条件の集合, 成り立つガード条件の集合

```
 1: MMap := ∅
 2: START:
 3: for M ∈ MS do
 4:    (S_ret, E_ret, A_-ret, A_+ret) := (∅, ∅, ∅, ∅)
 5:    if T > 0 then
 6:       M := EliminateNotAlways(M)
 7:    end if
 8:    CMS := R.GetConnectedComponents(M)
 9:    for CM ∈ CMS do
10:       if MMap.hasKey(CM) then
11:          (S_ret, E_ret, A_-ret, A_+ret) += MMap.getValue(CM)
12:       end if
13:       (S_tmp, E_tmp, CP, A_-tmp, A_+tmp) :=
            CalculateClosure(S, CM, CP, E, CheckConsistency)
14:       if S_tmp ≠ false then
15:          (S_ret, E_ret, A_-ret, A_+ret) += (S_tmp, E_tmp, A_-tmp, A_+tmp)
16:          MMap.insert(CMS, (S_tmp, E_tmp, A_-tmp, A_+tmp))
17:       else
18:          MS := MS \ {M_c ∈ MS | CM ⊆ M_c}
19:          goto START
20:       end if
21:    end for
22:    return (S_ret, CP, E_ret, M, A_-, A_+)
23: end for
24: return (false, CP, E, ∅, ∅, ∅)
```

図 7.5 制約モジュール間の依存関係を利用する CalculateMCS のアルゴリズム

依存関係グラフにおいて，連結でない制約モジュール同士には共通の変数が出現しない．このため，連結成分ごとに制約の解が求められていれば，あとはそれらの結果を結合するだけで，本来のアルゴリズムと同様の結果を得ることができる．また逆に，ある連結成分に対応する制約集合が矛盾する場合，その制約集合を含む制約モジュール集合は必ず矛盾する．このことを利用して，探索を行うモジュール集合を大幅に削減することができる．

以上のようにして制約間の依存関係を利用する *CalculateMCS* 関数の処理内容を図 7.5 に示す．

図 7.5 のアルゴリズムを説明する．まず入力として新たに依存関係グラフ R を追加している．R はシミュレーション開始後変化することは無いため，*CalculateMCS* 関数の呼び出し側が与えるものとする．*MMap* は，制約モジュール集合に対して，*CalculateClosure* の *CP* 以外の結果を対応させるマップであり，*CalculateMCS* 関数内で同一の連結成分

に対する計算の結果を再利用するために使用する．$(S_{ret}, E_{ret}, A_{-ret}, A_{+ret})$ はそれぞれ本関数の返り値として用いるための変数であり，モジュール集合 M ごとに初期化しておく．8 行目で，依存関係グラフ R のうち，解候補モジュール集合 M に含まれるノードのみを辿った場合の連結成分のリスト CMS を得る．その後，CMS の要素であるモジュール集合 CM について，制約求解を行う．ここでもし既に CM に対する計算を行っていた場合，$MMap$ から計算結果を取り出して利用する (11 行目)．この時利用している演算子”$+=$” は，左辺の各要素に対して，右辺の各要素を加える，という意味である．CM がまだ計算したことがないモジュール集合だった場合，$CalculateClosure$ 関数によって計算を行う (13 行目)．計算した結果が矛盾 ($false$) で無かった場合，その結果を返り値とする変数群と $MMap$ に追加する (15,16 行目)．矛盾していた場合 MS から CM を含むモジュール集合を除き，探索の枝刈りを行う．以上の処理を，全ての CM が矛盾しないモジュール集合 M が見つかるまで行うというのが，図 7.5 のアルゴリズムである．

7.2.2 アルゴリズムに対する考察

本アルゴリズムでは元のアルゴリズムである図 3.4 と比較して，探索の枝を大幅に削減できる可能性がある．例えば，7.3 のプログラムに対して本アルゴリズムを適用した場合，PP における探索の対象となるモジュール集合を図 7.8 に示す．なお，この図では `BOUNCE` や `INIT` などのモジュール集合を省略し，`FALL` を `F1` のように記述している．`F` に続く数字は，それが何番目の質点に関係するものであるかを表している．緑色のノードは実際に探索されるモジュール集合を表しており，灰色のノードは枝刈りによって探索する必要のなくなったモジュールを表している．この図を見ればわかるように，本アルゴリズムを適用することで探索するモジュール集合を 2^n から $n+1$ まで減らすことができるといえる．こうした枝刈りによる改善効果は，制約モジュールの数が多いほど，またシミュレーションが要素数の小さなモジュール集合にまで及ぶときほど効果は大きいと考えられる．

更に，枝刈り自体が不可能な場合でも，本アルゴリズムには同一の連結成分に対する計算結果の再利用を行えるという優位性がある．図 3.4 のアルゴリズムでは，そもそも同一のモジュール集合に対する計算は一度しか行われないために，計算結果を記録していても再利用を行うことはできない．

当初は依存関係を利用するため単純に，「連結成分ごとに無矛盾極大集合を求めればよいのではないか」と考えていた．連結成分ごとに計算が行えるとすると，計算の独立性が更に高まり，枝刈りをする必要すら無くなることに加え，モジュール集合に対する計算結果を再利用するまでもなく直接使用できる．しかし連結成分ごとに計算を行うことで正し

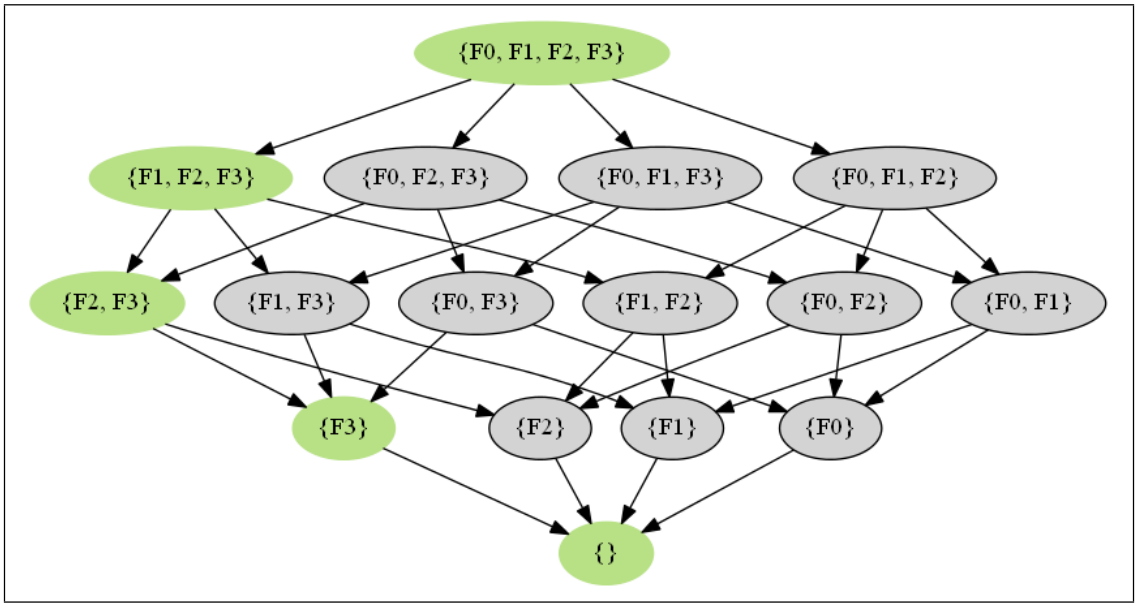

図 7.6　改善後の *CalculateMCS* 関数による図 7.3 の探索
（灰色のノードが枝刈で調べる必要のなくなったモジュール集合）

く解を求めることは，一般には不可能である．というのも，共通の変数のを持たないモジュール同士でも，優先関係を持っている可能性はあるからである．例えば，“`(x=1 << y=1, y=2) << 1=1]`.” のようなプログラムを考えると，初期時刻において y の値は 1 でも 2 でも良いが，x の値は y の値が 1 であるか否かによって変化する．このような振る舞いは，制約モジュール `y=1` と `x=1` との優先関係を考慮した上でシミュレーションを行わなければ計算できない．もし異なる連結成分をまたがるような優先関係が存在しないならば，単純な連結成分ごとの計算も可能であると思われる．

7.2.3　左極限値に関する制約でのみ連結する制約モジュール集合

7.2.1 節のアルゴリズムでは，共通の変数が 1 つも出現しないモジュールの集合が存在する場合に，計算を高速化することができるが，実用的なプログラムでそのようなモジュール集合が出現することは少ないと考えられる．というのも，そのようなプログラムは複数のプログラムに分割してそれぞれを実行しても，同じ結果が得られる場合が多いからである（7.2.2 節の最後に示すようなプログラムは例外だが，こうしたプログラム自体もあまり利用されることは無いと思われる）．

図 7.3 のような複数の物体が出現する場合は，物体同士の衝突を考えたプログラムの方

```
INIT(y, iy) <=> y = iy & y' = 0.
FALL(y) <=> [](y'' = -10).
BOUNCE(y) <=> [](y- = 0 => y' = -y'-).
COLLISION(yl, yr) <=> [](yl- = yr- => (yl' = yr'- & yr' = yl'-)).

INIT(y1, 10), INIT(y2, 5), INIT(y3, 3), (FALL(y1), FALL(y2), FALL(y3)) <<
  (BOUNCE(y1), BOUNCE(y2), BOUNCE(y3),
COLLISION(y1, y2), COLLISION(y2, y3),
COLLISION(y3, y1)).
```

図 7.7 質点同士の衝突を考えたプログラム

ががより有用である．例としては，ビリヤードのようなモデルが挙げられる．そこで本節では，そういったプログラムに対しても計算量が爆発しない方法を考える．図 7.7 に，3 つの質点が異なる高さから同時に落下を始め，質点同士や床との衝突を起こす HydLa プログラムを示す．なお，3 つ以上の物体が同時に衝突する場合は考えておらず，もしそのような状況が起きた場合は衝突を表す制約同士が矛盾して，シミュレーションは中断される．

このプログラムでは，質点同士の衝突に対応する制約モジュール COLLISION が追加されたため，すべてのモジュールが連結されており，7.2.1 節のアルゴリズムでは高速化できない．しかしここで，7.1.2 節でも扱った，「左極限値のみに言及するガード条件」に着目することで，最適化を行うことを考える．本プログラムでは，COLLISION と BOUNCE 内のガード条件がそれにあたり，かつこれらの制約モジュールはそのような条件付き制約しか持たない．このため，ガード条件が成り立たない時刻では，その制約モジュール自体が意味を持たないため，依存関係グラフの中で無視することができるようになり，衝突に関係ない質点に関する制約モジュール集合は，独立なモジュール集合だと考えることができる．

図 7.8 に，図 7.7 のプログラムのシミュレーションにおいて，変数 y1 で表される質点と変数 y2 で表される質点とが衝突する時刻におけるモジュール集合の探索過程を示す．なお，本時刻で最終的に採用されるモジュール集合は {F2} である．この時刻では，変数 y3 に関する制約モジュール（図中では F2 としている）がその他の制約モジュールと独立であるため，最初の {F0,F1,F2} の求解において，F2 の計算結果と，{F0, F1} が矛盾するという情報を得ることができる．すると，F2 の計算結果を図中の赤色のノードで表されるモジュール集合にて再利用することができるほか，{F0, F1} が矛盾するという情報から，{F0, F1} のシミュレーションを省略することができる．

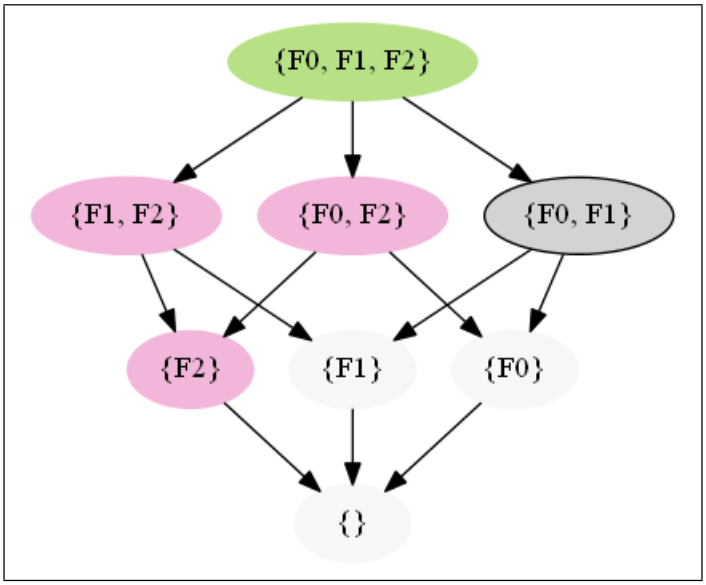

図 7.8　質点 y1 と質点 y2 が衝突した時刻の探索
（灰色のノード：枝刈されたモジュール集合
緑色のノード：通常通り計算されたモジュール集合
赤色のノード：F2 に関しての計算結果を再利用したモジュール集合
白色のノード：調べられなかったモジュール集合）

以上のように，左極限値のみに言及するガード条件だけを含むモジュール集合に着目することで，完全に独立ではない制約モジュール集合に対してもある程度の高速化を行うことができる．しかし，そもそも全質点が同時に衝突することを考慮したようなプログラムにおいて，全質点が同時に衝突する時刻では，この手法を用いても高速化を行うことはできない．

第 8 章

関連研究

Hyrose と関連研究との比較を表 8.1 に示す．ハイブリッドシステムのモデリングツールにはハイブリッドオートマトン [12] を用いてシステムを表現するものが多い [6] ．一方，制約による記述はオートマトンによる記述と比較して，システムの全状態を列挙する必要が無いため，記述量を減らしやすい．しかし HydLa のように制約を用いてハイブリッドシステムを扱う言語は少なく，Hybrid cc [3] [16] が数少ない例外である．Hybrid cc では簡潔な記述ができ，区間演算による精度保証も行われているが，離散変化を厳密に扱うことはできていない．Hyson[9] は不確定性を持つ Simulink プログラムに対して区間演算を用いてシミュレーションを行うツールであるが，現時点では解軌道の精度保証は行われていない．

また，Hyrose と同じく，数式処理を用いてハイブリッドシステムの検証を行うツールとして KeYmaera [1] が存在する．Hyrose を KeYmaera と比較すると，KeYmaera が

表 8.1　関連研究との比較

モデリング手法&ツール	モデリング形式	パラメータの扱い	精度保証
Hybrid Automata [12]	オートマトン	（実装依存）	（実装依存）
Hyson [9]	ブロック線図（Simulink）	可能	区間演算
Hybrid CC [3] [16]	制約	不可能	無
KeYmaera [1]	手続型（検証器）	可能	数式処理
HydLa	制約	可能	数式処理

ハイブリッドプログラムという手続き型言語に近いモデリングを行うのに対し，宣言型言語である HydLa を用いる点が違いとしてあげられる．KeYmaera はシステムの性質検証のみを目的としているのに対し，Hyrose はシステムのシミュレーションを基本としている点も異なる．

第 9 章

まとめと今後の課題

9.1 まとめ

本研究では，ハイブリッド制約言語 HydLa の処理系であった Hyrose を，パラメータを含む HydLa プログラムに対しても記号実行を行い，適切な場合分けを行って全解軌道を求めることができる記号実行シミュレータとして拡張した．実装した Hyrose を用いていくつかの例題について動作確認を行い，更に Hyrose の処理能力を向上させるためのいくつかの改善案を提案，その一部を実装した．

9.2 今後の課題

以下に，本研究の過程で今後行うべきだと判断した事項を挙げる．

9.2.1 依存関係を用いた最適化の実装

7.2 節で提案した改善案を実装することで，多くの物体が出現するようなモデルに対してのシミュレーションを高速化できると考えられる．また，この手法を実装する以外にも，制約同士が矛盾した時，どの制約が矛盾したのかを知ることができれば，依存関係を用いる場合とは別の方向から探索の枝刈りを行うことが考えられる．

9.2.2 HydLa 言語の拡張と実装

現在の HydLa では，同じ性質を持つ物体が複数出現する際に，それらに対応する制約を個々に宣言する必要がある．そうした場合に，手続き型言語の配列のように，1 つの文

でまとめて宣言できるようにすると，記述性が向上すると思われる．それ以外にも，∃記号を用いた記法は，現在提案されただけで実装されていないため，この記法の実装を行うことも課題としてあげられる．

9.2.3 バックエンドシステムの自作

現在の Hyrose では，バックエンドの数式処理システムとして外部のソフトウェアを利用しているため，Hyrose 自体それらのソフトウェアの仕様や処理能力に制限されてしまっている部分がある．更に Mathematica は有料である点が多くのユーザの利用を期待する上で障害となり得る．REDUCE では Hyrose で必要となる計算機能を完全にはカバーしきれていない．加えて，外部のソフトウェアである関係上，Hyrose とバックエンドの通信にかかるオーバーヘッド時間も影響する可能性が考えられる．これらの問題を解決するため，Hyrose にとって必要な機能だけを搭載した制約ソルバを独自に実装し，組み込むことが考えられる．こうすることで，Hyrose を使用するために特別なソフトが必要無くなることや，保守管理の上でのメリットなどが期待できる．

謝辞

指導教官としてご指導くださった上田和紀教授に心より感謝を申し上げます．HydLa 班ゼミや，ナント出張でお世話になった細部先生と石井さんに感謝を申し上げます．私の研究にご協力いただいた HydLa 班の皆さんに感謝を申し上げます．明るく話してくれた櫻庭君に感謝を申し上げます．独特のセンスで笑わせくれた清水君に感謝を申し上げます．仕事のことや幅広い方面の知識を教えてくれた徐君に感謝を申し上げます．見ている方まで元気になるような頑張りを見せてくれた目黒君に感謝を申し上げます．いつも研究やそれ以外の相談に乗ってくれた上田研の先輩，後輩の皆さんに感謝を申し上げます．これまでさまざまな方面から支援してくれた家族に感謝を申し上げます．皆さま，どうもありがとうございました．

2013 年 2 月　松本　翔太

参考文献

[1] A.Platzer and J.-D. Quesel : KeYmaera : A Hybrid Theorem Prover for Hybrid Systems. In IJCAR 2008, LNCS 5195, Springer-Verlag, 2008, pp. 171–178.

[2] Anthony C. Hearn, REDUCE, http://reduce-algebra.com/.

[3] B. Carlson and V. Gupta : Hybrid cc with Interval Constraints, in Proc. HSCC ' 98, LNCS 1386, Springer, 1998, pp. 80–94.

[4] 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.

[5] J. Lunze : Handbook of Hybrid Systems Control: Theory, Tools, Applications, Cambridge University Press, 2009.

[6] L. Carloni, R. Passerone, A. Pinto and A. L. Sangiovanni-Vincentelli : Languages and Tools for Hybrid Systems Design, Foundations and Trends in Design Automation, Vol. 1 No. 1, 2006, pp. 1–204.

[7] 松本翔太, 櫻庭翔, 高田賢士郎，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 の実装，日本ソフトウエア科学会第 28 回大会講演論文集，6E–4, 2011.

[8] Moore R. E. : Interval Analysis, Prentice–Hall Englewood Cliffs, N.J, 1966.

[9] S. Mimram, O. Bouissou and A. Chapoutot : HySon: Set-based Simulation of Hybrid Systems, to be presented at IEEE International Symposium on Rapid System Prototyping (RSP'2012), 2012.

[10] Paulo Tabuada : Verification and Control of Hybrid Systems，Springer-Verlag, 2009.

[11] 渋谷俊，高田賢士郎，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 処理系における実行アルゴリズム，コンピュータソフトウェア，Vol.28, No.3, 2011 pp. 167–172.

[12] T. Henzinger : The Theory of Hybrid Automata, in Proc. LICS '96, IEEE Computer Society Press, 1996, pp. 278–292.

[13] 高田賢士郎, 渋谷俊, 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 の数式処理実行系, 情報処理学会 第 73 回全国大会 1B–5, 2011.

[14] 竹口輝, 松本翔太, 上田和紀 : ハイブリッドシステムモデリング言語 HydLa を用いたシステム解析, ディペンダブルシステムワークショップ＆シンポジウム（DSW&DSS2011）, 2011.

[15] 上田和紀, 細部博史, 石井大輔 : ハイブリッド制約言語 HydLa の宣言的意味論, コンピュータソフトウェア, Vol.28, No.1, 2011, pp. 306–311.

[16] V. Gupta, R. Jagadeesan and V. Saraswat, D. Bobrow : Programming in Hybrid Constraint Languages, in Hybrid Systems II, LNCS 999, Springer, 1995, pp. 226–251.

[17] Wolfram Research, Inc., Mathematica,
http://www.wolfram.co.jp/products/
mathematica/index.html.

発表文献

[1] 松本翔太, 上田和紀: ハイブリッド制約言語 HydLa の記号実行シミュレータ Hyrose, コンピュータソフトウェア, 投稿中（19 pages）.

[2] Kazunori Ueda, Shota Matsumoto, Akira Takeguchi, Hiroshi Hosobe, Daisuke Ishii: HydLa: A High-Level Language for Hybrid Systems. In Proc. Second Workshop on Logics for System Analysis (LfSA 2012, affiliated with CAV 2012), July 2012, pp.3-17（共著）.

[3] Shota Matsumoto, Akira Takeguchi, Hiroshi Hosobe, Kazunori Ueda : Hybrid Constraint Language HydLa and Its Implementation, Hybrid Systems: Computation and Control, 2012 ポスター.

[4] Kakeru Sakuraba, Kazunori Ueda, Hiroshi Hosobe, Shun Shibuya, Shota Matsumoto: Simulation with guaranteed accuracy using hybrid system modeling language HydLa, Asian Symposium on Programming Languages and Systems, 2011 ポスター（共著）.

[5] 松本翔太, 上田和紀 : ハイブリッド制約言語 HydLa の記号実行シミュレータ Hyrose, 日本ソフトウェア科学会第 29 回大会 5C-4, 2012 登壇発表（16 pages）.

[6] 竹口輝, 和田亮, 松本翔太, 細部博史, 上田和紀 : ハイブリッド制約言語プログラムのハイブリッドオートマトンへの変換アルゴリズム, 日本ソフトウェア科学会 第 29 回大会 2A-3, 2012. 登壇発表（共著, 10 pages）.

[7] 竹口 輝, 松本 翔太, 上田 和紀 : ハイブリッドシステムモデリング言語 HydLa を用いたシステム解析, ディペンダブルシステムワークショップ&シンポジウム 2011 登壇発表（共著, 8 pages）.

[8] 松本 翔太, 櫻庭 翔, 高田 賢士郎, 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 の実装, 日本ソフトウェア科学会第 28 回大会 6E-4, 2011 登壇発表（16 pages）.

[9] 松本翔太, 高田賢士郎, 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 の処理系による非決定性の扱い, 第 13 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, 2011 ポスター.