2017 年度 修士論文

制約階層に基づく ハイブリッドシステムモデリング言語 HydLa の静的誤り検出手法

提出日： 2018 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・情報通信専攻

学籍番号：5116F036-5
小山 峻平

概要

小山 峻平

時間経過に伴い連続的な変化と離散的な変化を繰り返すシステムのことをハイブリッド（動的）システム（以降，ハイブリッドシステム）と呼ぶ．床を跳ねるボールや，車のブレーキ制御など様々な現象はハイブリッドシステムとして表現することが可能である．制約階層に基づく宣言型言語 HydLa はこのハイブリッドシステムをシミュレーションするためのモデリング言語である．

HydLa のようなモデリング言語を用いて検証したいシステムをモデル化する際，記述されたモデルが記述者の意図を正確に反映していないという問題がしばしば発生する．本研究ではこれをモデリングエラーと呼ぶ．

HydLa は制約プログラミングパラダイムに基づくモデリング言語であり，制約間に優先度を設定することができる制約階層の考え方を導入している．それにより，手続き型言語のように直接手続きを書かなくてもよくプログラムの記述量を削減できる反面，プログラムのフローを追うことが難しくなっている．そのためモデリングエラーが発生しても検出や修正は難しい．

本研究では，モデリングエラーの修正を容易にすることを目的として，対象となるモデリングエラーの定義，静的なモデリングエラーの検出手法の提案，提案手法の HydLa 処理系 HyLaGI への実装をおこなった．提案手法の特徴は，制約階層を考慮した上で HydLa プログラムが取る状態を過不足なく検証し，プログラム内で発生しないモデリングエラーを誤って検出しないよう考慮していることである．手法の基本方針は，全制約の冪集合から，モデリングエラーを引き起こす制約の集合を導き出す．しかし，HydLa は制約階層の考え方を採用しているため，全制約の冪集合の中に，シミュレーション中に決して到達しないと判明している状態が存在する．例えば，矛盾している制約間に階層構造により優先度が設定されている場合は，その制約が同時に成立することはない．このような条件を記述量を少なく設定できることが制約階層のメリットであるため，提案する誤り検出手法においても同時に成立しない状態についてモデリングエラーを検出することは避けたいと考えた．それを実現するために，提案手法ではガード条件となる制約式やガード後件の制約式を考慮して制約の組み合わせである候補制約集合を作成し，その候補制約集合に対してモデリングエラーの検出をおこなう．実際に提案手法の実装をおこない，モデリングエラー検出の結果を使うことで，比較的容易にモデルの修正が可能になることを確認した．

Abstract

Shunpei KOYAMA

Hybrid systems are dynamical systems involving continuous and discrete changes. Many dynamical systems can be represented as hybrid systems; examples include a ball bouncing on a floor and a car with braking. HydLa is one of modeling tools for hybrid systems.

When we describe models for the verification of hybrid systems using a modeling language like HydLa, we often encounter a problem that a program does not reflect the programmer's intention. We call it Modeling Errors.

HydLa is a constraint-based modeling language for hybrid systems. HydLa allows one to specify priorities between constraints, which is called constraint hierarchy. We can describe programs easily by constraint hierarchy, but it is not easy to detect the causes of modelling errors and to correct them.

The purpose of this reserch is to make it easy to detect the cause of modeling errors and to fix them. We define modeling errors, propose a method for statically detecting the causes of modelling errors, and implement the proposed method. A feature of the proposed method is that it prevents erroneous detection. The basic idea of the proposed method is to derive errors from the powerset of all constraints of a HydLa program. However, not all of the subsets of constraints are adopted in the simulation of a HydLa program. For example, if a constraint hierarchy is specified between conflicting constraints, the simulation never reaches the state in which these constraints become effective at the same time. The purpose of the proposed method is to prevent detecting such states as errors. This is realized by creating candidate constraint sets. We implemented these methods in HyLaGI and made sure that we could fix the errors easily by using the result of error detection.

目次

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

第 2 章　HydLa 3
2.1　ハイブリッドシステム 3
2.2　HydLa 言語 3
2.3　HydLa 処理系 HyLaGI 5
2.4　HydLa 統合開発環境 webHydLa 7

第 3 章　HydLa における制約階層概念とそれによる問題点 8
3.1　制約プログラミング 8
3.2　制約階層 8
3.3　HydLa の各制約の状態 9
3.4　モデリングエラー 13

第 4 章　提案手法 19
4.1　目的 19
4.2　アルゴリズム 19
4.3　検出例 21

第 5 章　手法の実装について 25
5.1　HyLaGI 本体との関係 25
5.2　実装概要 26

第 6 章　例題 28

6.1 氷の張った湖面で跳ねる質点のモデル 28
6.2 壁の付近で跳ねる質点のモデル 32

第 7 章 自動修正についての考察 40
7.1 プログラムの修正の種類 40
7.2 自動修正が困難な理由 42

第 8 章 まとめと今後の課題 43
8.1 まとめ 43
8.2 今後の課題 43

謝辞 44

参考文献 45

発表文献 47

Appendix **ソースコードの変更概略** 48

図目次

2.1 床で跳ねる質点のモデル 4
2.2 床で跳ねる質点のモデルの解軌道 5
2.3 HyLaGI が対象としている HydLa の構文 6
2.4 床で跳ねる質点のモデルのフェーズ 6 までのシミュレーション結果 . . . 7

3.1 Basic HydLa の構文 9
3.2 制約モジュール BOUNCE 10
3.3 *HM*，*HMC* の定義 10
3.4 制約モジュールの状態の定義 11
3.5 *MS* の定義 14
3.6 モデリングエラーの定義 14
3.7 氷の張った湖面で跳ねる質点のモデル 16
3.8 エラーを含む，氷の張った湖面で跳ねる質点のモデル 16
3.9 エラーを含む，氷の張った湖面で跳ねる質点のモデルの実行結果 17

4.1 モデリングエラー検出アルゴリズム 19
4.2 MakeCandidateConstraintSets 20
4.3 SolveCandidateConstraintSets 21
4.4 制約モジュール BOUNCE，BOUNCE2 22
4.5 制約モジュール BOUNCE，WALL 23

6.1 (再掲) エラーを含む，氷の張った湖面で跳ねる質点のモデル 29
6.2 (再掲) エラーを含む，氷の張った湖面で跳ねる質点のモデルの実行結果 . 30
6.3 図 6.1 の HydLa プログラムに対するモデリングエラー検出例 31
6.4 修正後の氷の張った湖面で跳ねる質点のモデル 31

6.5 修正後の氷の張った湖面で跳ねる質点のモデルの実行結果 (フェーズ 6 まで) 32
6.6 壁の付近で跳ねる質点のモデル 33
6.7 壁の付近で跳ねる質点のモデルの実行結果 34
6.8 図 6.6 の HydLa プログラムに対するモデリングエラー検出例 34
6.9 修正後の壁の付近で跳ねる質点のモデル 35
6.10 修正後の壁の付近で跳ねる質点のモデルの実行結果 (フェーズ 6 まで) . . 36
6.11 微小値を加える修正後の壁の付近で跳ねる質点のモデル 37
6.12 微小値を加える修正後の壁の付近で跳ねる質点のモデルの実行結果 (フェーズ 6 まで) 38
6.13 微小値を加える修正後の壁の付近で跳ねる質点のモデルのの解軌道 . . . 39

7.1 モデリングエラーを含む，床で跳ねる質点のモデル 41

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 [1] とは，時間の経過に伴い連続的な変化と離散的な変化を繰り返すシステムのことである．HydLa[2] は制約階層に基づくハイブリッドシステムモデリング言語である．HydLa は各変数の挙動を示した制約の組み合わせによってシステムの挙動を表す．HydLa の他にハイブリッドシステムを記述し，シミュレーションする手法として，ハイブリッドオートマトン [3] を用いたものや，制約を用いた HybridCC[4] などが挙げられる．

システムを検証する際，先ずはシミュレーションのためのモデルを作成するモデル化をおこなう必要がある．しかし，検証したいシステムをモデル化する際，記述されたモデルが記述者の意図を正確に反映していないという問題がしばしば起き，これをモデリングエラーと呼ぶ．具体的には，シミュレーション実行時に設定している制約式自体には問題はないが，シミュレーション結果が得られなかったり，余計な情報を含む結果が得られたりすることである．例えば，車のブレーキをかけるシステムの，ブレーキをかける閾値等について検証したい場合があるとする．この時，本来は得られたシミュレーション結果より，ブレーキをかける閾値となる制約等に問題がないかを検証する．しかし，モデリングエラーが発生していると閾値等の制約自体の検証ではなく，その制約の組み合わせ方などモデル化の問題を検証しなければならない．システムの検証時にユーザーが，システムの設計に問題があるのか，システムには問題はないがモデル化に問題があるのかを適切に判断できないと検証の効率は大幅に低下する．モデリングエラーが頻発する言語を使用してシステムの検証を行う際には，その判別を適切に行うことが必要となる．

HydLa は制約プログラミングパラダイムに基づくモデリング言語であり，制約間に優

先度を設定することができる制約階層の考え方を導入している．それにより，手続き型言語のように直接手続きを書かなくてもよくプログラムの記述量を削減できる反面，プログラムのフローを追うことが難しくなっている．そのためモデリングエラーが発生しても検出や修正は難しい．本研究では，モデリングエラーの修正を容易にすることを目的として，静的にモデリングエラーを検出する手法を提案する．

本研究の関連研究としては，HydLa のシミュレーション実行中にシミュレーション継続に問題がある場合に原因となる制約 (unsat core[5]) を検出する手法が存在する [6]．しかし，提案されている手法ではシミュレーションを行うことが前提となっており，シミュレーション時に与えられる初期値や実行時間によってはエラーを検出できない場合がある．それに対し本研究の提案手法では，与えられた HydLa プログラムより静的に原因となる制約，状態を検出する．

1.2　論文構成

まず 2 章では，ハイブリッドシステムモデリング言語である HydLa と，その処理系について説明を行う．次に 3 章で，HydLa の特徴である制約階層概念と，それにより発生する問題点について述べる．4 章では，静的にモデリングエラーを検出するための提案手法について述べる．5 章では，4 章で述べた提案手法を HydLa 処理系 HyLaGI に対しておこなった実装について述べる．6 章では，例題を用いて提案手法の検出内容とそれによるプログラムの修正について説明をおこなう．7 章では，プログラムの自動修正についての考察をおこなう．最後に 8 章で本論文をまとめ，今後の課題を述べる．

第 2 章

HydLa

この章では，まずハイブリッドシステムモデリング言語 HydLa について説明する．その後 HydLa の処理系 HyLaGI，統合開発環境として開発されている webHydLa について触れる．

2.1 ハイブリッドシステム

時間経過に伴い連続的な変化と離散的な変化を繰り返すシステムのことをハイブリッド（動的）システム [1] と呼ぶ．床を跳ねるボールや，車のブレーキ制御など様々な現象をハイブリッドシステムとして表現することが可能である．床を跳ねるボールは，ボールの自由落下を重力加速度で加速し続ける連続変化とし，床に触れた時にボールの速度が反転する離散的な変化を離散変化として表現することが可能である．車のブレーキ制御は，車の位置の変化を連続変化，ブレーキの有無によって加速度が増減する変化を離散変化として表現することが可能である．ハイブリッドシステムは，これまでに挙げた例のように物理学や制御工学などの分野に応用が可能な概念である．

2.2 HydLa 言語

HydLa[2] とは Hybrid dynamical Language の略であり，ハイブリッドシステムをモデリングするための制約階層に基づいた宣言型言語のことである．先のハイブリッドシステムの説明で挙げたように物理学や制御工学などの分野で扱う多くの問題はハイブリッドシステムとして扱うことが可能である．それらのハイブリッドシステムをモデリングし，シミュレーションすることを目的として開発されている．

主な特徴は，原子論理式を制約として扱い，それら制約の組み合わせからなる制約モジュール間に優先度からなる制約階層を設定することでモデルを表現することである．これにより，記述量の増大を抑えることができる．

また．HydLa は対象ユーザを，プログラミングを専門としない技術者としている [7]．そのため，数学や論理学を理解していれば記述にあたって新たに習得すべき必要のある記法や概念を大幅に減らすことができ，簡潔なモデリングが可能であるように設計されている．

2.2.1　HydLa によるモデリング例

図 2.1 に HydLa プログラムの例を示す．示すプログラムは，床で跳ねる質点のモデルである．

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

図 2.1　床で跳ねる質点のモデル

このプログラムは，(INIT) y の初期値 10, 初速度 0, (FALL) y の加速度 -10, (BOUNCE) y が接地した時 ($y\text{-} = 0$)，y の速度が反発係数 $(4/5)$ によって減衰して反転する ($y' = -4/5 * y'\text{-}$) という意味を持つ制約の組み合わせから成る．HydLa プログラム内に存在する変数は時刻 t に関する関数であり，プログラム内の変数 y は $y(t)$ のことを表す．また，HydLa プログラム内に存在する変数と記号 - の組み合わせは，その変数の時刻 t における左極限値を表す．$y\text{-}$ は変数 y の時刻 t における左極限値を表している．この意味に従い，時間を $t = 0$ から進めることでシミュレーションを行うことで得られる解軌道を図 2.2 に示す．図 2.2 は縦軸を高さ y，横軸を時間 t として，質点の動く解軌道を示している．

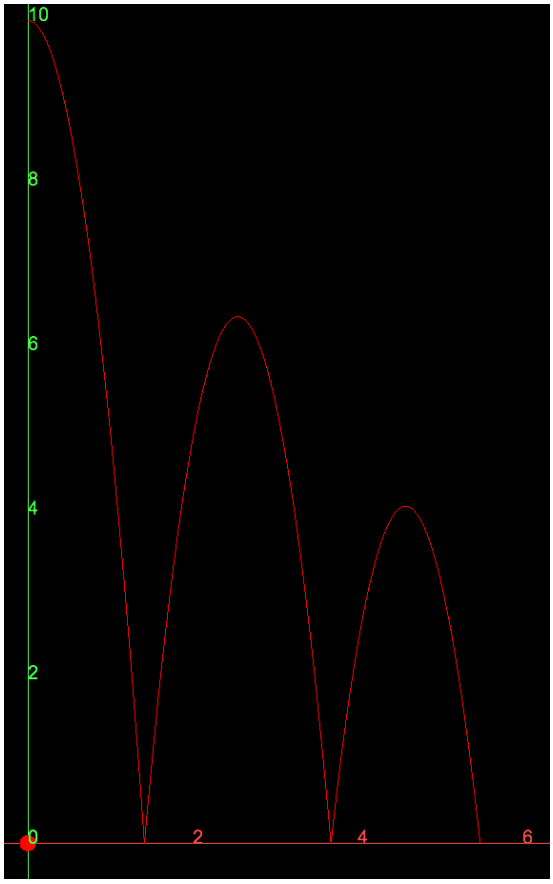

図 2.2　床で跳ねる質点のモデルの解軌道

2.3　HydLa 処理系 HyLaGI

HyLaGI は HydLa 言語により記述されたモデルの示す解軌道を数式処理により求める処理系である．HyLaGI は C++ で実装されており，約 3 万行のプログラムから構成される．HydLa 言語で記述されたプログラムを与えると，そのプログラムを満たす変数の解軌道を出力する．バックエンドに Mathematica による数式処理を用いており，計算誤差のないシミュレーションが可能になっている．

図 2.3 に，HydLa 言語処理系 HyLaGI[8] が対象としている HydLa の構文を示す．

図 2.3 に示した構文の他にも，文献 [9] に記述されているリスト記法に対応する構文が存在するが，複数の制約をまとめて表現できるようにしたたシンタックスシュガーであり，本質的には図 2.3 の構文に対応している．

(hydla program)	H	::=	($DF.$ | DC .)$*$
(definition)	DF	::=	$Dname(\vec{E})\{DC\}$ | $Cname(\vec{X})$<=> C
(declaration)	DC	::=	M | $Dname(\vec{E})$ | DC, DC | DC << DC | (DC)
(module)	M	::=	C
(constraint)	C	::=	A | $Cname(\vec{E})$ | $C \wedge C$ | G => C | [] C | (C)
(guard)	G	::=	A | $G \wedge G$ | $G \vee G$ | (G)
(atomic constraint)	A	::=	E (*relop* E)+
(expression)	E	::=	通常の式 | E' | E ^E | E-

図 2.3　HyLaGI が対象としている HydLa の構文

2.3.1　HyLaGI の実行アルゴリズム

HyLaGI は，時間変数である t が 0 であるときを始点とし，t の値を増加させていくことで時間変化に伴う各変数の変化をシミュレートする．ハイブリッドシステムの状態である連続変化をここでは IP（Interval Phase），離散変化をここでは PP（Point Phase）とする．$t = 0$ の時に PP として，以降 IP と PP を交互に繰り返すことでシミュレーションを進める．処理系では，プログラム中に存在する制約モジュール同士の優先度より解候補となる制約モジュールの集合を導出，その中で極大無矛盾となるような制約モジュール集合を各フェーズごとに導出する．そして，その制約に従うように変数を求めることで解軌道を求め，次の離散変化時刻を求める．設定されたフェーズ数，または時間が経過するとシミュレーションを終了する．以上のアルゴリズムで HyLaGI は実行される．

2.3.2　HyLaGI のシミュレーション例

図 2.1 に示した床で跳ねる質点のモデルの HydLa プログラムを，HyLaGI にてフェーズ 6 まで実行した際の実行結果について図 2.4 に示す．

シミュレーション結果は，各フェーズごとに変数 (及びその微分値) の式として与えられ，それを元に解軌道をプロットしたものが図 2.2 となる．

```
------ Result of Simulation ------
---------Case 1---------
---------1---------
---------PP 1---------
unadopted modules: {}
positive  :
negative  :
t : 0
y : 10
y' : 0
y'' : -10
---------IP 2---------
unadopted modules: {}
positive  :
negative  :
t : 0->2^(1/2)
y : (t^2+(-2))*(-5)
y' : t*(-10)
y'' : -10
---------2---------
---------PP 3---------
unadopted modules: {FALL}
unsat mod : {BOUNCE, FALL}
unsat cons : {y''=-10, y'=-4/5*y'-}
positive  : y-=0=>y'=-4/5*y'-
negative  :
t : 2^(1/2)
y : 0
y' : 2^(1/2)*8
```

```
---------IP 4---------
unadopted modules: {}
positive  :
negative  : y-=0=>y'=-4/5*y'-
t : 2^(1/2)->2^(1/2)*13/5
y : 18*2^(1/2)*t+(-26)+t^2*(-5)
y' : 2*(2^(1/2)*9+t*(-5))
y'' : -10
---------3---------
---------PP 5---------
unadopted modules: {FALL}
unsat mod : {BOUNCE, FALL}
unsat cons : {y''=-10, y'=-4/5*y'-}
positive  : y-=0=>y'=-4/5*y'-
negative  :
t : 2^(1/2)*13/5
y : 0
y' : 2^(1/2)*32/5
---------IP 6---------
unadopted modules: {}
positive  :
negative  : y-=0=>y'=-4/5*y'-
t : 2^(1/2)*13/5->2^(1/2)*97/25
y : t^2*(-5)+2^(1/2)*t*162/5+(-2522)/25
y' : t*(-10)+2^(1/2)*162/5
y'' : -10
# number of phases reached limit
```

図 2.4　床で跳ねる質点のモデルのフェーズ 6 までのシミュレーション結果

2.4　HydLa 統合開発環境 webHydLa

webHydLa は HydLa 言語のブラウザ上で動作する統合開発環境である．シミュレーション結果を 3D 空間で確認することが可能であり，アニメーションやパラメータの表示にも対応している．実装は主に JavaScript を使用しており，サーバーサイドには Python，処理系には HyLaGI を使用している．主な機能は，プログラムエディタ部分，オプション設定部分，描画部分に分かれる．

第 3 章

HydLa における制約階層概念とそれによる問題点

この章では，HydLa に用いられている考え方である制約プログラミングパラダイムと制約階層について説明する．その後，HydLa における各制約の状態を定義し，その定義を使用してモデリングエラーの定義をする．特に記述しないが，本章で定義する各制約の状態とモデリングエラーは，ある時刻 t における状態を指す．これは，プログラムを実行するにあたり時刻 t を変化させる HydLa プログラムでは，各時刻ごとに制約集合の状態が変化するためである．

3.1 制約プログラミング

制約プログラミングとは，制約でモデルを表現するプログラミングパラダイムである．ここで述べている制約とは，解の特性を表す方程式や不等式からなる原子論理式のことであり，それらの制約を満たす解の組み合わせを計算する．手続きを記述する必要がないため，プログラムの記述が容易であることがメリットとして挙げられる．一方，手続きを記述しないためプログラムの細かい挙動がわかりにくいことがデメリットとして挙げられる．

3.2 制約階層

制約階層 [12] とは，制約間に優先度を導入したものである．ハイブリッドシステムの制約条件を過不足なく適切に与えることは容易ではない．HydLa は制約階層構造を取り入

れることによって制約条件を適切に与えやすく設計されている．

3.3　HydLa の各制約の状態

3.3.1　Basic HydLa

本研究では，図 2.3 にて紹介した HydLa の構文よりも簡潔にした Basic HydLa の構文を対象とする．Basic HydLa の構文を図 3.1 に示す．

(basic hydla program)	BH	::=	$(\ CM\ .\ \mid\ HC\ .\ )*$
(constraint module)	CM	::=	$Cname$ <=> M
(hierarchical constraint)	HC	::=	$CM\ \mid\ HC,\ HC\ \mid\ HC$ << HC
(module)	M	::=	G => $C \mid$ []$(G$ => $C)$
(constraint)	C	::=	$A\ \mid\ C\ \wedge\ C\ \mid$ [] C
(guard)	G	::=	$A\ \mid\ G\ \wedge\ G$
(atomic constraint)	A	::=	$E\ (\ relop\ E\ )+$
(expression)	E	::=	通常の式 $\mid\ E' \mid E$-

図 3.1　Basic HydLa の構文

プログラムは制約モジュール (CM) と制約階層 (HC) の組み合わせより構成される．制約モジュールはモジュール (M) に名前 ($Cname$) をつけたものであり，モジュールは前件 (G) と後件 (C) の組み合わせから構成される．この前件 (G) の制約を ask 制約，後件 (C) の制約を tell 制約と表現する．モジュール内の記号 => は，論理包含を示す記号であり，G => C と記述された場合は，前件 (G) が真ならば後件 (C) も真であるということを示す．前件と後件はそれぞれ原子論理式 (A) の論理和で構成されている．構文中に現れる [] は，時相論理演算子 Globally を示しており，続く制約が常に真であることを示す．E 内の記号 ’ は E の 1 階微分値，記号 - は E の左極限値を示す．

3.3.2　各制約モジュールの状態

Basic HydLa の構文において，HydLa の各制約モジュールがシミュレーション中に取る状態について以下にまとめる．制約モジュールは，前件の制約 (ask 制約) が満たされるならば (=>)，後件の制約 (tell 制約) も満たされるという意味を持つ．例えば図 3.2 の制約モジュールでは，ask 制約 $y\text{-} = 0$ が満たされれば，tell 制約 $y' = -y'\text{-}$ も満たされると

いう意味である．

BOUNCE <=> [](y- = 0 => y' = -y'-).

図 3.2　制約モジュール BOUNCE

このようにそれぞれの制約モジュールの論理包含の意味として，以下の 2 つの状態が考えられる．

- ask 制約が成立するので tell 制約が成立する．
- ask 制約が成立しないので tell 制約は成立してもしなくともよい．

ただし，HydLa の制約モジュールには，優先度 (制約階層) を設定することが可能である．制約階層内で，A << B と記述された場合は，制約モジュール A と制約モジュール B の制約式が矛盾する場合，制約モジュール A を無視して制約モジュール B の制約式を優先するという意味になる．この制約階層により，上述した 2 つの状態に加えもうひとつ状態が考えられる．

- ask 制約の成立不成立にかかわらず，tell 制約は成立してもしなくてもよい．

上記の 3 つの状態について述べる前に，制約モジュール CM に対して優先度が高い制約モジュールについて，図 3.3 に $HM(CM)$，$HMC(CM)$ として定義する．

$$\begin{array}{rlcl} \textit{(higher module set)} & HM(CM) & := & \{m \mid CM \texttt{ << } m\} \\ \textit{(higher module constraints)} & HMC(CM) & := & \bigwedge_{m \in HM(CM)} (G(m) \texttt{ => } C(m)) \end{array}$$

図 3.3　HM，HMC の定義

図 3.3 における $G(CM)$, $C(CM)$ はそれぞれ，制約モジュール CM の前件，後件を示す．$HM(CM)$ は与えられた制約モジュール CM よりも優先度が高い制約モジュールの集合である．制約階層にて与えられる半順序集合を元に，制約モジュールが狭義の半順序で与えられる制約モジュール CM より優先度が高い制約モジュールの集合を $HM(CM)$ とする．$HMC(CM)$ は与えられた制約モジュールよりも優先度が高い制約モジュールの論理包含の連言である．

3.3.3　各制約モジュールの状態の定義

前述した各制約モジュールが取る状態をまとめて以下に示す.

- ask 制約の成立不成立にかかわらず，tell 制約は成立してもしなくてもよい.
- ask 制約が成立するので tell 制約が成立する.
- ask 制約が成立しないので tell 制約は成立してもしなくともよい.

示した制約モジュールの状態を定義する．図 3.4 に 3 種類の制約モジュールの状態の定義をそれぞれ示す.

$(ignored)$	$ignored(CM)$	$:=$	$(HMC(CM) \wedge G(CM) \wedge \neg C(CM) \wedge HM(CM) \neq \emptyset)$
			$\vee \exists m \in HM(CM)(ignored(m))$
$(valid)$	$valid(CM)$	$:=$	$HMC(CM) \wedge G(CM) \wedge C(CM) \wedge \neg ignored(CM)$
$(failed)$	$failed(CM)$	$:=$	$\neg(HMC(CM) \wedge G(CM)) \wedge \neg ignored(CM)$

図 3.4　制約モジュールの状態の定義

ignored(**無視**)

無視状態とは，ask 制約が満たされているかどうかにかかわらず，後件を無視した状態で制約充足問題を解いてもいい状態のことを指す．前述した各制約モジュールが取る状態の中では,「ask 制約の成立不成立にかかわらず，tell 制約は成立してもしなくてもよい.」に該当する．制約モジュールが無視状態を取るのは，以下の 2 つの条件のどちらかを満たす場合である.

- ガードが成立するが、後件が自分より優先度が高い制約の論理包含の連言と矛盾する.
- 自分より上の優先度を持つ制約モジュールが無視されている.

valid(**有効**)

ask 制約が満たされており，tell 制約を満たさなくてはならない (有効な) 状態のことを指す．前述した各制約モジュールが取る状態の中では,「ask 制約が成立するので tell 制約が成立する.」に該当する．制約モジュールが有効状態を取るのは，以下の 3 つの条件を全て満たす場合である.

- ガードが成立する
- 後件が自分より優先度が高い制約の論理包含の連言と矛盾しない
- 自分より上の優先度を持つ制約モジュールが無視されていない

$failed$(**無効**)

ask 制約が満たされておらず，tell 制約は制約充足問題に影響しない (無効な) 状態のことを指す．前述した各制約モジュールが取る状態の中では，「ask 制約が成立しないので tell 制約は成立してもしなくともよい.」に該当する．制約モジュールが無効状態を取るのは，以下の 2 つの条件を全て満たす場合である．

- ガードが成立しない
- 自分より上の優先度を持つ制約モジュールが無視されていない

3.3.4　各制約モジュールの状態の例

例題を用いて，各制約モジュールの各状態について説明をする．図 2.1 に示した，床で跳ねる質点のモデルを用いて説明する．

$0 \leq t < \sqrt{2}$ の時，質点は空中 $(0 < y)$ に存在する．このときの制約モジュール FALL, BOUNCE の状態は,

FALL　以下の理由により有効

ガードが $true$ なので成立する.

後件が自分より優先度が高い制約の論理包含の連言 ($y\text{-} = 0$ => $y' = -4/5 * y'\text{-}$) と矛盾しない.

自分より上の優先度を持つ制約モジュール (BOUNCE) が無視されていない

BOUNCE　以下の理由により無効

ガード $y\text{-} = 0$ が成立しない.

自分より上の優先度を持つ制約モジュール $(\emptyset)$ が無視されていない.

$t = \sqrt{2}$ の時，質点は床 $(y = 0)$ に存在する．このときの制約モジュール FALL, BOUNCE の状態は,

FALL　以下の理由により無視

ガードが $true$ なので成立する.

後件が自分より優先度が高い制約の論理包含の連言 ($y\text{-} = 0$ => $y' = -4/5 * y'\text{-}$) と矛盾する.

BOUNCE　以下の理由により有効

ガードが $y\text{-} = 0$ なので成立する.

後件が自分より優先度が高い制約の論理包含の連言 ($\emptyset$) と矛盾しない.

自分より上の優先度を持つ制約モジュール ($\emptyset$) が無視されていない.

図 2.1 のモデルは上述した 2 つの制約モジュールの状態の集合をプログラムの状態として持つ．そのプログラムの状態を繰り返すことでシミュレーションが進むことになる.

3.4　モデリングエラー

モデリングエラーとは，システムをモデル化する際，記述されたモデルが記述者の意図を正確に反映していないという問題のことである．特に，制約プログラミングのように処理の手続きが外から見えにくいモデルでは，原因の発見が困難になるためプログラマの経験に頼る部分が多くなることや，長時間の修正作業が必要になるという問題がある.

3.4.1　HydLa におけるモデリングエラー

先に述べたように，HydLa プログラムは前件 (ask 制約) と後件 (tell 制約) の組み合わせである制約モジュールの間に優先度による階層構造 (制約階層) を与えることで構成される．図 2.1 の例では，制約モジュール BOUNCE の前件が ($y\text{-} = 0$)，後件が ($y' = -y'\text{-}$) である．前件が満たされた場合のみ，後件の制約が有効になる．それぞれ有効になっている後件が矛盾する場合，制約階層に従い，極大無矛盾になるように制約モジュールを無視することでシミュレーションを行っている．このシミュレーション実行時に，シミュレーションが進まないことや，値が定まらないなど，プログラマの意図と異なる解軌道が得られることがある．その中でも，制約プログラミングでシステムをモデル化する際に生ずるもののことをモデリングエラーと呼ぶ.

本研究では HydLa のモデリングエラーを，制約過多と制約不足，非決定状態として定義して扱う．制約式が矛盾しているのにもかかわらず，それらの制約間に優先度が付いていない場合を制約過多，各変数に対して適当な制約式が存在しない場合を制約不足とする．また，制約過多の一種として，非決定状態を分けて扱う．非決定状態は，制約式同士が矛盾しそれらの制約間に優先度が付いていない場合に，どちらかを非決定的に選択し解軌道を求める場合のことである．そのため，ケース分岐が発生し結果が複数通り得られる

ことになる．

まず，図 3.5 にて，HydLa のシミュレーション時に解軌道を求める制約の集合 (MS) や，それに付随する関数についてについての定義を行う．

(module set)	MS	:=	$\{m \mid valid(m)\}$
(module set constraints)	MSC	:=	$\bigwedge_{m \in MS} (G(m) \wedge C(m))$
(variables)	$var(MS)$	:=	MS 内の制約式に含まれる自由変数の集合
(program variables)	$pvar$	:=	プログラム全体の制約式に含まれる自由変数の集合
(conflict module set)	$CMS(CM)$	:=	$\{m \mid \neg(CM \wedge m) \wedge m \in MS\}$

図 3.5　MS の定義

MS はある時刻 t において有効 ($valid$) となる制約モジュールの集合である．MSC は MS に含まれる制約式の連言である．$var(MS)$ は MS 内の制約式に含まれる自由変数の集合を示し，$pvar$ はプログラム全体の制約式に含まれる自由変数の集合を示している．$CMS(CM)$ は与えられた CM と矛盾する MS 内の制約モジュールの集合のことを示している．

次に，図 3.6 にモデリングエラーの定義を示す．

$(over\text{-}constrained)$	$OC(CM)$	:=	$valid(CM) \wedge \neg MSC$
			$\wedge \exists (m_1, m_2) \in (CMS(CM) \cup CM)$
			$(HM(m_1) = \emptyset \wedge HM(m_2) = \emptyset \wedge m_1 \neq m_2)$
$(under\text{-}constrained)$	$UC(v)$	:=	$v \in pvar \wedge v \notin var(MS)$
$(non\text{-}deterministic)$	$ND(CM)$	:=	$valid(CM) \wedge \neg MSC$
			$\wedge \forall m \in CMS(CM)(HM(m) \neq \emptyset)$
			$\wedge HM(CM) \neq \emptyset \wedge CMS(CM) \neq \emptyset$

図 3.6　モデリングエラーの定義

図 3.6 において，ある制約モジュール CM に対して $HM(CM) \neq \emptyset$ ということは，制約モジュールがその HydLa プログラム内で最高優先度であることを示す．以降，最高優先度を持つ制約モジュールのことを，プログラムに必須な制約ということで requred 制約と表現する．

over-constrained(**制約過多**)

制約過多状態は，有効になる制約モジュールが余分に存在し，それらが矛盾することでシミュレーションが実行不可能になる，つまり解が求まらない状態である．制約過多状態

になるのは，以下の 3 つの条件を全て満たす場合である．

- 制約モジュールが有効である
- 有効になっている制約モジュールが全体として矛盾する
- 非決定状態ではなく、required 制約が 1 つのみではない

under-constrained(**制約不足**)

制約不足状態は，制約集合内に値を決定するための制約モジュールが足りない，つまり解が余計に存在してしまう状態である．制約不足状態になるのは，以下の 1 つの条件を満たす場合である．

- 有効になっている制約集合内の制約により明記されていない変数がある

non-deterministic(**非決定状態**)

非決定状態は，有効になる制約モジュールが余分に存在し，それらが矛盾するが，それらの制約モジュールを非決定的に選択することで以降のシミュレーションを続ける状態のことである．制約モジュールの有効か無視の状態が非決定的であるということより，非決定状態とする．非決定状態になるのは，以下の 3 つの条件を全て満たす場合である．

- 制約モジュールが有効である
- 有効になっている制約モジュールが全体として矛盾する
- 矛盾した制約モジュールの集合の中に、required の制約モジュールを含まない

ここで述べた 3 種類のモデリングエラーの中で非決定状態は，意図的に条件分岐を引き起こすモデルを作成することがあるため必ずしもモデリングエラーとは言えない．しかし，意図しない条件分岐を起こすモデルは明らかにモデリングエラーであるため，モデリングエラーの一種として扱うことが適当であるとした．

3.4.2　HydLa におけるモデリングエラーの例

次にモデリングエラーの例を具体例を元に示す．

図 3.7 に図 2.1 にて示した床を跳ねるボールのモデルを変更して，氷の張った湖面で跳ねる質点のモデルを示す．図 2.1 のモデルに，x 軸の概念を追加，さらに，$4 < x < 6$ の範囲は氷が溶けており水面が見えているとして制約 `WATER` を追加した．

```
1 INIT <=> y = 10 & y' = 0 & x = 0 & x' = 1.
2 FALL <=> [](y'' = -10).
3 MOVE <=> [](x'' = 0).
4 FLOOR <=> [](y-=0 => y' = -4/5 * y'-).
5 WATER <=> [](y-=0 & 4<x<6
6             => y' = 4/5 * y'- & x' = 1/2 * x'-).
7
8 INIT, (FALL, MOVE) << FLOOR << WATER.
```

図 3.7　氷の張った湖面で跳ねる質点のモデル

HydLa では制約として使用する論理式の組み合わせ方や制約階層の与え方について自由度が高いため，同じモデルに対して多様な記述が可能である．そのため図 3.7 のプログラムにはモデリングエラーが存在しないが，モデリングエラーを引き起こしてしまうモデルも作成される可能性がある．例えば，質点のデフォルトの挙動を示す制約モジュール FALL と MOVE を一つの制約モジュール FALL_MOVE として記述してしまうと制約不足のモデリングエラーを含む図 3.8 のプログラムが作成される．

```
1 INIT <=> y = 10 & y' = 0 & x = 0 & x' = 1.
2 FALL_MOVE <=> [](y'' = -10 & x'' = 0).
3 FLOOR <=> [](y-=0 => y' = -4/5 * y'-).
4 WATER <=> [](y-=0 & 4<x<6
5             => y' = 4/5 * y'- & x' = 1/2 * x'-).
6
7 INIT, FALL_MOVE << FLOOR << WATER.
```

図 3.8　エラーを含む，氷の張った湖面で跳ねる質点のモデル

図 3.8 のモデルでは，$y\text{-} = 0 \wedge (x \leq 4 \vee 6 \leq x)$ の時に制約モジュール FLOOR のみが有効になり制約モジュール FALL_MOVE が無視されることで変数 x に対しての制約式が存在しなくなってしまう．それにより，制約不足のモデリングエラーが発生する．

制約不足のモデリングエラーを含む図 3.8 のモデルを HyLaGI でシミュレーションした結果を図 3.9 に示す．

```
1  ------ Result of Simulation ------
2  ---------Case 1---------
3  ---------1---------
4  ---------PP 1---------
5  unadopted modules: {}
6  positive :
7  negative :
8  t : 0
9  x : 0
10 y : 10
11 x' : 1
12 y' : 0
13 x'' : 0
14 y'' : -10
15 ---------IP 2---------
16 unadopted modules: {}
17 positive :
18 negative :
19 t : 0->2^(1/2)
20 x : t
21 y : (t^2+(-2))*(-5)
22 x' : 1
23 y' : t*(-10)
24 x'' : 0
25 y'' : -10
26 ---------PP 3---------
27 unadopted modules: {FALL_MOVE}
28 unsat mod : {FALL_MOVE, FLOOR}
29 unsat cons : {y''=-10, y'=-4/5*y'-}
30 positive : y-=0=>y'=-4/5*y'-
31 negative :
32 t : 2^(1/2)
33 y : 0
34 y' : 2^(1/2)*8
35 # number of phases reached limit
36
37 Simulation Time : 0.576681 s
38 Finish Time : 0.577777 s
39
40 WARNING: x is completely unbound at phase...
41 %% PhaseType: 1
42 %% id: 3
43 %% step: 2
44 %% parent_id:2
45 %% unadopted modules: {FALL_MOVE}
46 %% inconsistent modules: {FALL_MOVE, FLOOR}
47 %% inconsistent constraints: {y''=-10, y'=-4/5*y'-}
48 %% positive_asks
49 y-=0=>y'=-4/5*y'-
50 %% negative_asks
51 %% current_time: 2^(1/2)
52 %% end_time: 2^(1/2)
53 --- variable map ---
54 y <=> 0
55 y' <=> 2^(1/2)*8
```

図 3.9　エラーを含む，氷の張った湖面で跳ねる質点のモデルの実行結果

このようなプログラム中におけるエラーの詳細な原因に関して，手続き型言語 [10] [11] では研究があるが HydLa のような宣言型言語では研究が進んでいない.

第 4 章

提案手法

本章では，HydLa プログラム内に存在しているモデリングエラーの静的検出手法を提案する．

4.1 目的

これまでに示したモデリングエラーを検出するための手法を考案し，HydLa 言語処理系 HyLaGI 上に実装した．提案手法は大きく 3 つの手順から構成される．制約階層を考慮した上で HydLa プログラムが取る状態を過不足なく検証し，プログラム内で発生しないモデリングエラーを誤検出しないために以下の手法をとる．

4.2 アルゴリズム

提案アルゴリズムは図 4.1 のようになる．本手法には，時相論理演算子 Globally([]) を含まない制約式は対象としない．理由は，時相論理演算子 Globally([]) を含まない制約式は時刻 $t = 0$ のみで成立するものであるため，普通にシミュレーション実行した時にはじめに問題が現れ，修正が比較的容易であるためである．

Require: HydLa プログラム HydLaProgram
Ensure: モデリングエラーの内容 ME
1: $CCS := MakeCandidateConstraintSets(HydLaProgram)$
2: $CCS_{tell} := SolveCandidateConstraintSets(CCS)$
3: $ME := GetModelingErrors(CCS, CCS_{tell})$

図 4.1　モデリングエラー検出アルゴリズム

提案アルゴリズムの詳細について説明する．

MakeCandidateConstraintSets

同時に前件を満たす制約モジュールの組み合わせを求める．この制約モジュールの組み合わせのことを候補制約集合とする．ここで，前件が同時に満たされない (例，$x = 1, x = 2$) 制約モジュールの集合は検出対象から外すことができる．提案アルゴリズムは図 4.2 のようになる．

Require: HydLa プログラム HydLaProgram
Ensure: 候補制約集合 CCS
1: $CCS := \{\}$
2: $MSS := makeMSS(HydLaProgram)$
3: **for** *each* $MS \in MSS$ **do**
4: $A := SolveAsk(MS)$
5: **if** $A! = \{\}$ **then**
6: **if** $A \notin CCS$ **then**
7: $CCS := CCS \cup A$
8: **end if**
9: **end if**
10: **end for**

図 4.2　MakeCandidateConstraintSets

SolveCandidateConstraintSets

求めた候補制約集合ごとに後件も含めて解を求め，それを候補制約集合解とする．ここで，特殊な解 (例，$x = -x$ の制約において解が $x = 0$ の場合) を含む場合とそれ以外に分割することができる．提案アルゴリズムは図 4.3 のようになる．

GetModelingErrors

得られた候補制約集合と候補制約集合解より，候補制約集合解に解がない場合を制約過多，解が不定となる場合を制約不足とする．その場合の制約集合，変数の値を求め，提示する．

Require: 候補制約集合 CCS
Ensure: 候補制約集合解 CCS_{tell}
1: CCS_{tell} := {}
2: **for** *each* $CS \in CCS$ **do**
3: A := $SolveTell(CS)$
4: **if** $A! = \{\}$ **then**
5: **if** $CheckConflictDiscrete(A) = false$ **then**
6: A := $ResolveWithoutUnsatCore(A)$
7: **end if**
8: **if** $CheckVal(A) = false$ **then**
9: A := $ResolveSplitRange(A)$
10: **end if**
11: **else**
12: A := $ResolveWithoutUnsatCore(A)$
13: **end if**
14: CCS_{tell} := $CCS_{tell} \cup A$
15: **end for**

図 4.3　SolveCandidateConstraintSets

4.3　検出例

制約不足のモデリングエラーを含む図 3.8 のモデルを用いて，提案手法の検出の流れを説明する．

4.3.1　入出力

入力として，図 3.8 の HydLa プログラムを与える．出力として，モデリングエラーの内容を得る．今回の例で得たいモデリングエラーの内容は，以下の通りである．

- 変数 x が足りない制約不足の状態である．
- その時の制約モジュール集合は，{FLOOR} である
- その時の変数の左極限値は，$y\text{-} = 0$，$x \leq 4\ or\ 6 \leq x$ である．

これらの情報を提案手法にて検出する．

4.3.2 MakeCandidateConstraintSets

図 4.2 の関数は，HydLa プログラムを入力として受け取り，候補制約集合とその時の変数の条件を返す．2 行目の *makeMSS* で受け取った HydLa プログラムの制約階層を元に制約モジュール集合を作成する．制約モジュールの中で，FALL_MOVE のように $G(CM)$ が存在しない制約モジュールは，$G(CM) = true$ として扱う．$G(CM) = true$ は，どのような状態でもガードが $ture$ であるので無効 $(failed)$ の状態になることはないという特徴がある．$G(CM) \neq true$ の制約モジュールの冪集合の各要素に全ての $G(CM) = true$ の制約モジュールを加えた集合を *MSS* とする．今回の例では，*MSS* は {FALL_MOVE，FLOOR，WATER}，{FALL_MOVE，WATER}，{FALL_MOVE，FLOOR}，{FALL_MOVE} の 4 種類である．

3 行目から作成した *MSS* の要素の制約モジュール集合一つづつについて処理をおこなう．4 行目の *SolveAsk* で集合内の前件の制約式を全て連立させて解 A を求める．求めた A に解が存在し (5 行目)，*CCS* に包含されていないなら (6 行目)，候補制約集合として，その制約モジュール集合と解 A を *CCS* に追加する．A に解が存在しない場合とは，$x = 1, x = 2$ のような矛盾する $G(CM)$ が *MSS* 内に存在する場合などである．式を満たす x が存在しないので，その *MSS* は解制約モジュール集合となることはなく，その状態について探索をおこなうことは無意味なので候補制約集合として追加しない．6 行目で述べる包含とは，A と解が一致し制約モジュール集合が包含される時に $true$ を返す．例えば，プログラムが図 4.4 に示す 2 つの制約モジュールにて構成されている場合，MSS は {BOUNCE，BOUNCE2}，{BOUNCE}，{BOUNCE2}，{} の 4 種類である．

BOUNCE <=> [](y- = 0 => y' = -y'-).
BOUNCE2 <=> [](y- = 0 => y' = -x'-).

図 4.4　制約モジュール BOUNCE，BOUNCE2

この時，A が $y\text{-} = 0$ となり一致する *MSS* は，{BOUNCE，BOUNCE2}，{BOUNCE}，{BOUNCE2} である．この場合，制約集合内の制約数が極大になる {BOUNCE，BOUNCE2} に関して後述する *SolveCandidateConstraintSets* にて探索を行えば残りの制約集合を探索する必要はないので追加しない．

今回の例の *MSS* から生成される候補制約集合 (*CCS*) は，

- {{FALL_MOVE，FLOOR，WATER}，$y\text{-} = 0, 4 \leq x \leq 6$}
- {{FALL_MOVE，FLOOR}，$y\text{-} = 0$}

- {{FALL_MOVE}, }

の 3 種類である．この候補制約集合を関数の出力として返す．

4.3.3 SolveCandidateConstraintSets

図 4.3 の関数は，候補制約集合を入力として受け取り，候補制約集合解を返す．

2 行目から受け取った *CCS* 要素の候補制約集合一つづつについて処理をおこなう．3 行目の *SolveTell* で集合内の前件，後件全ての制約式を連立させて解 A を求める．求めた A に解が存在しない (12 行目) なら，解が存在しない原因の制約モジュール ($unsatcore$) を取り除いて再計算 (*ResolveWithoutUnsatCore*) する．求めた A に解が存在する (4 行目) なら，次の 2 つの確認をおこなう．

- *CheckConflictDiscrete*
- *CheckVal*

CheckConflictDiscrete は，ある変数に対して微分値が存在する時，その変数は連続であるという，HydLa における暗黙の連続性を解 A が満たしているかを判定する関数である (5 行目)．もし，暗黙の連続性を満たしていない場合は，それらの制約式を含む制約モジュールを $unsatcore$ として，制約階層に基づいていくつかの制約モジュールを除外して再計算 (*ResolveWithoutUnsatCore*) する．*CheckVal* は，ある変数の左極限値が離散変化前後で一致するかを判定する関数である (8 行目)．候補制約集合の制約式は，ある一部の特殊な状態においてのみ成立し，それ以外の状態では解が存在しない場合がある．例えば図 4.5 に示す 2 つの制約モジュールを満たす y'- は 0 のみである．この場合，離散変化前は y'- に特に制約がないため $-inf$ から inf までの値をとることができるが，離散変化後は y'- $= 0$ 以外の y'- では矛盾してしまう．この矛盾してしまう y'- $= 0$ 以外の y'- の場合については矛盾する制約モジュールを削減することで解が求まる可能性がある．そのため，候補制約集合の条件を場合分け (y'- $= 0$ の場合と y'- $= 0$ 以外の場合) して再計算をおこなう．

BOUNCE <=> [](y- = 0 => y' = -y'-).
WALL <=> [](x- = 2^(1/2) => x' = -x'- & y' = -4/5 * y'-).

図 4.5　制約モジュール BOUNCE，WALL

以上の処理の結果として得られた，制約モジュール集合とその解を候補制約集合解

(CCS_{tell}) に追加する (15 行目). 今回の例で生成される候補制約集合解 (CCS_{tell}) は,

- $\{\{$`WATER`, $(y'\text{-} \neq 0)\}$, $y\text{-} = 0$, $4 \leq x \leq 6$, $x'\text{-} = 2 * x'$, $x' = 1/2 * x'\text{-}$, $y' = -4/5 * y'\text{-}$, $y'\text{-} = -5/4 * y'\}$
- $\{\{$`FLOOR`, `WATER`, $(y'\text{-} = 0)\}$, $y\text{-} = 0, 4 \leq x \leq 6$, $y'\text{-} = 0$, $x'\text{-} = 2 * x'$, $y' = 0$, $x' = 1/2 * x'\text{-}\}$
- $\{\{$`FLOOR`$\}$, $y\text{-} = 0$, $y' = -4/5 * y'\text{-}$, $y'\text{-} = -5/4 * y'\}$
- $\{\{$`FALL_MOVE`$\}$, $x'' = 0$, $y'' = -10\}$

の 4 種類である. この候補制約集合解を関数の出力として返す.

4.3.4　GetModelingErrors

候補制約集合と候補制約集合解を入力として受け取り, モデリングエラーの内容を返す処理をおこなう. 与えられた候補制約集合と候補制約集合解より, 候補制約集合解に解がない場合を制約過多, 解が不定となる場合を制約不足とする. また候補制約集合解が複数解を持つ場合は, 非決定状態とする. その場合の制約集合, 変数の値を求め, 提示する.

第 5 章

手法の実装について

この章では，4 章において挙げた提案手法の実装方法について説明する．また実装に際して，関連する処理系の構成，実装について触れる．

5.1 HyLaGI 本体との関係

提案手法の実装にあたって，以下の 2 点を特に考慮した．

5.1.1 HyLaGI 本体の処理に手を加えない

本手法により実現される機能は，HyLaGI 本体の処理には直接関係しないものであるため，option の一つとしての実装が適当であると考えた．そのため，実装部分を独立させ，保守，更新を行いやすいように設計した．実装コードは，ほぼ全て新規に作成した HyLaGI/src/debug 内に存在し，HyLaGI/src/core/main.cpp より --debug_constraint オプションで呼び出せるようにした．

5.1.2 数式処理に Mathematica 以外の処理系を使う

現在，数式処理を HyLaGI は全て Mathematica に依存している．Mathematica は商用であるがゆえに性能は高いが，多くのソースコードが公開されておらず，今後の HyLaGI の実装において問題が発生することが懸念される．そのため今回の実装では，python のライブラリである sympy[13] を使用した．最低限の計算は可能である上に，微分方程式も扱うことができる点は他のフリーの数式処理ソフトと比較し優秀であったため，今回使用することにした．

5.2　実装概要

5.2.1　手法実装部

make_ask_map

提案手法の図 4.2 の処理を実装している関数である．与えられた制約階層から前件が同時に成立する制約モジュールの集合である候補制約集合を作成する関数．前件が同時に成立しても，他の候補制約集合に完全に包含される制約モジュールの集合を候補制約集合としないことで以降無駄な制約集合に対して解を求めなくても済むように工夫している．

make_tell_map

提案手法の図 4.3 の処理を実装している関数である．候補制約集合から，後件も含めて解を求める．候補制約集合全体で解がないときは，制約階層に従って制約を減らして再計算し，これ以上制約を減らせない場合に矛盾する制約モジュールを求める．また，解として得られた変数の範囲が， make_ask_map で求めた候補制約集合の変数の範囲より狭くなる場合は，候補制約集合を分割して，条件を追加する．

5.2.2　計算処理部

前述したように，計算には python のライブラリである sympy[13] を使用した．

make_val_map

与えられた連立方程式の解を求める sympy の関数 solve に求めたい連立方程式の情報を渡す関数．しかし， sympy の関数 solve は多変数不等式を連立方程式に含めた場合，解をうまく求めることができなかった．そのため，今回の実装では以下の 3 つの手順を踏むことで多変数不等式を含む連立方程式の解を求めた．

1. findequality
   等式のみを全て連立させて解を求める．求めた解は，各変数ごとに保持しておく．
2. findinequality
   一つずつ不等式を満たす変数の値を求める．求めた変数の値は，findequality とは別に各変数ごとに保持しておく．
3. findval

これまで求めたそれぞれの変数に対する値が複数存在する場合，それらを満たす変数の値を求める．

なお，上記のどの手順の間でも解が求まらなければ，その連立方程式は解なしとする．

第 6 章

例題

この章では第 4 章で提案し，第 5 章で実装した機能を例題を用いて考察する．本章では，4 つのモデリングエラーを含む HydLa プログラムを紹介，検出例を示し，モデリングエラーを含まないプログラムへの修正例を示す．

6.1 氷の張った湖面で跳ねる質点のモデル

6.1.1 扱うモデルの説明

第 3 章にて扱った氷の張った湖面で跳ねる質点のモデルである．一部 $(4 < x < 6)$ の湖面で氷が溶けており，その場所で接地したらそのまま水中に入るが，それ以外の場所で接地した場合は跳ね返るというモデルである．図 3.7 にはモデリングエラーを含まないモデルを示しているが，同章にて述べている通り HydLa は複数通りのモデルを記述することが可能である．そのため，モデリングエラーを含んでいるモデルを用いて例を示す．

6.1.2 プログラムと出力結果

図 3.8 に示した，モデリングエラーを含む氷の張った湖面で跳ねる質点のモデルと HyLaGI で得られる実行結果を，図 6.1，図 6.2 に再掲する．

このモデルは，変数 x に対する制約が不足している $UC(x)$ の状態であるが，実行結果を見ただけではどの状況で変数 x が不足しているのかがわかりにくい．ここで述べている状況とは，左極限値を含む各変数の値と採用されている制約集合のことである．HydLa プログラムにおいては，エラーの状況がわかると比較的容易に修正が可能である．

```
1 INIT <=> y = 10 & y' = 0 & x = 0 & x' = 1.
2 FALL_MOVE <=> [](y'' = -10 & x'' = 0).
3 FLOOR <=> [](y-=0 => y' = -4/5 * y'-).
4 WATER <=> [](y-=0 & 4<x<6
5              => y' = 4/5 * y'- & x' = 1/2 * x'-).
6
7 INIT, FALL_MOVE << FLOOR << WATER.
```

図 6.1　(再掲) エラーを含む，氷の張った湖面で跳ねる質点のモデル

6.1.3　モデリングエラー検出結果

このモデルに対して，提案手法を実行すると検出できるモデリングエラーは図 6.3 のようになる．

この検出結果より，モデリングエラーが発生する時の各変数の値と，制約モジュール集合が得られる．それを元に行ったプログラムの修正について以下に示す．

6.1.4　プログラム修正例

図 6.3 の検出結果よりわかることは以下の 3 点である．

- 変数 x が足りない制約不足の状態である．
- その時の制約モジュール集合は，{FLOOR} である
- その時の変数の左極限値は，$y\text{-} = 0$ である．

これより，新しく変数 x に関する制約を作成する．制約モジュール集合が {FLOOR} の時は，氷面に質点が接地した時である．この時，変数 x の速度 x' は変化しないことが想定される．よってこの場合は，制約モジュール FLOOR に変数 x に関する制約 $x' = x'\text{-}$ を追加した，FLOOR2 を作成し，制約モジュール FLOOR と置き換える．

$$\texttt{FLOOR2 <=> []}(y\text{-} = 0 \texttt{ => } y' = -4/5 * y'\text{-} \ \&\ x' = x'\text{-}).$$

制約 $x' = x'\text{-}$ は，x' の値が左極限と同じであることを明示している．この制約により，制約モジュール集合 {FLOOR2} の時に変数 x に関する制約が存在することになるので，モデリングエラーは解消できる．実際に修正したモデルと HyLaGI で得られる実行結果を，

```
------ Result of Simulation ------
---------Case 1---------
---------1---------
---------PP 1---------
unadopted modules: {}
positive :
negative :
t : 0
x : 0
y : 10
x' : 1
y' : 0
x'' : 0
y'' : -10
---------IP 2---------
unadopted modules: {}
positive :
negative :
t : 0->2^(1/2)
x : t
y : (t^2+(-2))*(-5)
x' : 1
y' : t*(-10)
x'' : 0
y'' : -10
---------PP 3---------
unadopted modules: {FALL_MOVE}
unsat mod : {FALL_MOVE, FLOOR}
unsat cons : {y''=-10, y'=-4/5*y'-}
positive : y-=0=>y'=-4/5*y'-
negative :
t : 2^(1/2)
y : 0
y' : 2^(1/2)*8
# number of phases reached limit

Simulation Time : 0.576681 s
Finish Time : 0.577777 s

WARNING: x is completely unbound at phase...
%% PhaseType: 1
%% id: 3
%% step: 2
%% parent_id:2
%% unadopted modules: {FALL_MOVE}
%% inconsistent modules: {FALL_MOVE, FLOOR}
%% inconsistent constraints: {y''=-10, y'=-4/5*y'-}
%% positive_asks
y-=0=>y'=-4/5*y'-
%% negative_asks
%% current_time: 2^(1/2)
%% end_time: 2^(1/2)
--- variable map ---
y <=> 0
y' <=> 2^(1/2)*8
```

図 6.2　(再掲) エラーを含む，氷の張った湖面で跳ねる質点のモデルの実行結果

```
1 CCS →FLOOR.
2 prevVAL →y=0.
3 ME →Under-constrained.
4 x is undefined.
```

図 6.3　図 6.1 の HydLa プログラムに対するモデリングエラー検出例

図 6.4，図 6.5 に示す．

```
1 INIT <=> y = 10 & y' = 0 & x = 0 & x' = 1.
2 FALL_MOVE <=> [](y'' = -10 & x'' = 0).
3 FLOOR2 <=> [](y-=0 => y' = -4/5 * y'- & x' = x'-).
4 WATER <=> [](y-=0 & 4<x<6
5              => y' = 4/5 * y'- & x' = 1/2 * x'-).
6
7 INIT, FALL_MOVE << FLOOR2 << WATER.
```

図 6.4　修正後の氷の張った湖面で跳ねる質点のモデル

図 6.5 を見ると，フェーズ 6 までシミュレーションできていることがわかる．この結果は，図 3.7 に示した HydLa プログラムと制約モジュールの数や内容が異なるが，同じ解軌道を得られる．

```
------ Result of Simulation ------
---------Case 1---------
---------1---------
---------PP 1---------
unadopted modules: {}
positive  :
negative  :
t : 0
x : 0
y : 10
x'  : 1
y'  : 0
x'' : 0
y'' : -10
---------IP 2---------
unadopted modules: {}
positive  :
negative  :
t : 0->2^(1/2)
x : t
y : (t^2+(-2))*(-5)
x'  : 1
y'  : t*(-10)
x'' : 0
y'' : -10
---------2---------
---------PP 3---------
unadopted modules: {FALL_MOVE}
unsat mod : {FALL_MOVE, FLOOR2}
unsat cons  : {y''=-10, y'=-4/5*y'-}
positive  : y-=0=>y'=-4/5*y'-&x'=x'-
negative  :
t : 2^(1/2)
x : 2^(1/2)
y : 0
x'  : 1
y'  : 2^(1/2)*8
---------IP 4---------
unadopted modules: {}
positive  :
negative  : y-=0=>y'=-4/5*y'-&x'=x'-
t : 2^(1/2)->2^(1/2)*13/5
x : t
y : 18*2^(1/2)*t+(-26)+t^2*(-5)
x'  : 1
y'  : 2*(2^(1/2)*9+t*(-5))
x'' : 0
y'' : -10
---------3---------
---------PP 5---------
unadopted modules: {FALL_MOVE}
unsat mod : {FALL_MOVE, FLOOR2}
unsat cons  : {y''=-10, y'=-4/5*y'-}
positive  : y-=0=>y'=-4/5*y'-&x'=x'-
negative  :
t : 2^(1/2)*13/5
x : 2^(1/2)*13/5
y : 0
x'  : 1
y'  : 2^(1/2)*32/5
---------IP 6---------
unadopted modules: {}
positive  :
negative  : y-=0=>y'=-4/5*y'-&x'=x'-
t : 2^(1/2)*13/5->2^(1/2)*97/25
x : t
y : t^2*(-5)+2^(1/2)*t*162/5+(-2522)/25
x'  : 1
y'  : t*(-10)+2^(1/2)*162/5
x'' : 0
y'' : -10
# number of phases reached limit
```

図 6.5　修正後の氷の張った湖面で跳ねる質点のモデルの実行結果 (フェーズ 6 まで)

6.2　壁の付近で跳ねる質点のモデル

6.2.1　扱うモデルの説明

図 2.1 に示した床を跳ねる質点のモデルに壁を導入したモデルである．床には摩擦は存在しないが，壁には摩擦が存在し，壁に衝突した時に y 方向 (垂直方向) の速度が $4/5$ になるというモデルを作成する．

6.2.2　プログラムと出力結果

作成した HydLa プログラムと HyLaGI で得られる実行結果を，図 6.6，図 6.7 に示す．このモデルは，図 2.1 のモデルに新たに変数 x(水平方向) を追加して，それに伴い制約モジュール MOVE，WALL を追加したものである．制約モジュール MOVE は，変数 x のデフォルトの挙動を示す．制約モジュール WALL は，質点が壁に衝突した時に x 方向 (水平方向) にはそのまま跳ね返り，y 方向 (垂直方向) の速度は摩擦により $4/5$ になることを示す制約である．

```
1  INIT <=> y = 10 & y' = 0 & x = 0 & x' = 1.
2  FALL <=> [](y'' = -10).
3  MOVE <=> [](x'' = 0).
4  BOUNCE <=> [](y- = 0 => y' = -4/5 * y'-).
5  WALL <=> [](x- = 2^(1/2) => x' = -x'- & y' = 4/5*y'-).
6
7  INIT, (FALL, MOVE) << (BOUNCE, WALL).
```

図 6.6　壁の付近で跳ねる質点のモデル

このモデルは，制約モジュール BOUNCE，WALL が制約過多となる OC(BOUNCE)，OC(WALL) の状態である．具体的には制約モジュール BOUNCE の式 $y' = -4/5 * y'$- と制約モジュール WALL の式 $y' = 4/5 * y'$- とが y'- $\neq 0$ の時に矛盾する．しかし，図 6.7 に示した実行結果では，どの制約の制約式が制約過多なのかわかりにくい．

6.2.3　モデリングエラー検出結果

このモデルに対して，提案手法を実行すると検出できるモデリングエラーは図 6.8 のようになる．

この検出結果より，モデリングエラーが発生する時の各変数の値と，制約モジュール集合が得られる．それを元に行ったプログラムの修正について示す．

6.2.4　プログラム修正例

図 6.8 の検出結果よりわかることは以下の 3 点である．

```
1  ------ Result of Simulation ------
2  ---------Case 1---------
3  ---------1---------
4  ---------PP 1---------
5  unadopted modules: {}
6  positive :
7  negative :
8  t : 0
9  x : 0
10 y : 10
11 x' : 1
12 y' : 0
13 x'' : 0
14 y'' : -10
15 ---------IP 2---------
16 unadopted modules: {}
17 positive :
18 negative :
19 t : 0->2^(1/2)
20 x : t
21 y : (t^2+(-2))*(-5)
22 x' : 1
23 y' : t*(-10)
24 x'' : 0
25 y'' : -10
26 ---------PP 3---------
27 unadopted modules: {}
28 unsat mod : {BOUNCE, FALL, WALL}
29       {MOVE, WALL}
30       {BOUNCE, WALL}
31 unsat cons : {y''=-10, y'=-4/5*y'-, y'=4/5*y'-}
32       {x''=0, x'=-x'-}
33       {y'=-4/5*y'-, y'=4/5*y'-}
34 positive : y-=0=>y'=-4/5*y'-
35       x-=2^(1/2)=>x'=-x'-&y'=4/5*y'-
36 negative :
37 t : 2^(1/2)
38 # execution stuck
```

図 6.7　壁の付近で跳ねる質点のモデルの実行結果

```
1  CCS →BOUNCE, WALL, (y'- != 0).
2  prevVAL →y=0, x=sqrt(2).
3  ME →Over-constrained.
4  y' = -4/5 * y'-, y' = 4/5 * y'-
```

図 6.8　図 6.6 の HydLa プログラムに対するモデリングエラー検出例

- 制約式 $y' = -4/5 * y'$- と $y' = 4/5 * y'$- が矛盾する制約過多の状態である．
- その時の制約モジュール集合は，{BOUNCE，WALL，(y'- != 0)} である
- その時の変数の左極限値は，y- $= 0$，x- $= \sqrt{2}$ である．

検出結果より，モデリングエラーが発生する状況は，制約 BOUNCE と WALL が同時に有効になり，変数 y の速度が 0 ではない時であるとわかる．つまり，壁と床の境目にちょうど落下する時に発生する状況である．このモデリングエラーを修正する方法はいくつか存在するが，今回はより上位の制約モジュールを新たに作成することで修正を行う．新たに作成する制約モジュールを CORNER とする．質点が角に衝突する時の挙動は 3 体同時衝突の問題であり，挙動は自明ではないが，今回は x 方向の速度が壁に衝突した際の速度変化 ($x' = -x'$-)，y 方向の速度が床に衝突した際の速度変化 ($y' = -4/5 * y'$-) をするものとして CORNER を作成する．

CORNER <=> [](y- $= 0$ & x- $= \sqrt{2}$ => $y' = -4/5 * y'$- & $x' = -x'$-).

この制約を制約モジュール BOUNCE と WALL より優先度を高く設定することにより，制約モジュール集合 {BOUNCE，WALL，(y'- != 0)} の時は BOUNCE と WALL の代わりに CORNER が制約モジュールに含まれることになるので，モデリングエラーは解消できる．実際に修正したモデルと HyLaGI で得られる実行結果を，図 6.9，図 6.10 に示す．

```
1  INIT <=> y = 10 & y' = 0 & x = 0 & x' = 1.
2  FALL <=> [](y'' = -10).
3  MOVE <=> [](x'' = 0).
4  BOUNCE <=> [](y- = 0 => y' = -4/5 * y'-).
5  WALL <=> [](x- = 2^(1/2) => x' = -x'- & y' = 4/5*y'-).
6  CORNER <=> [](y-=0 & x-=2^(1/2) => y' = -4/5 * y'- & x' = -x'-)
       .
7
8  INIT, (FALL, MOVE) << (BOUNCE, WALL) << CORNER.
```

図 6.9　修正後の壁の付近で跳ねる質点のモデル

図 6.10 を見ると，フェーズ 6 までシミュレーションできていることがわかる．

```
------ Result of Simulation ------
---------Case 1---------
---------1---------
---------PP 1---------
unadopted modules: {}
positive  :
negative  :
t : 0
x : 0
y : 10
x'  : 1
y'  : 0
x'' : 0
y'' : -10
---------IP 2---------
unadopted modules: {}
positive  :
negative  :
t : 0->2^(1/2)
x : t
y : (t^2+(-2))*(-5)
x'  : 1
y'  : t*(-10)
x'' : 0
y'' : -10
---------2---------
---------PP 3---------
unadopted modules: {FALL, MOVE, WALL}
unsat mod : {BOUNCE, CORNER,
              FALL, WALL}
      {CORNER, MOVE, WALL}
      {BOUNCE, CORNER, WALL}
      {CORNER, WALL}
unsat cons  : {y''=-10, y'=-4/5*y'-,
             y'=4/5*y'-, y'=-4/5*y'-}
      {x''=0, x'=-x'-, x'=-x'-}
      {y'=-4/5*y'-, y'=4/5*y'-,
             y'=-4/5*y'-}
      {y'=4/5*y'-, y'=-4/5*y'-}
positive  : y-=0=>y'=-4/5*y'-
      x-=2^(1/2)=>x'=-x'-&y'=4/5*y'-
      y-=0&x-=2^(1/2)
              =>y'=-4/5*y'-&x'=-x'-
negative  :
t : 2^(1/2)
x : 2^(1/2)
y : 0
x'  : -1
y'  : 2^(1/2)*8
---------IP 4---------
unadopted modules: {}
positive  :
negative  : y-=0=>y'=-4/5*y'-
      x-=2^(1/2)=>x'=-x'-&y'=4/5*y'-
      y-=0&x-=2^(1/2)=>y'=-4/5*y'-&x'=-x'-
t : 2^(1/2)->2^(1/2)*13/5
x : 2^(1+1/2)+t*(-1)
y : 18*2^(1/2)*t+(-26)+t^2*(-5)
x'  : -1
y'  : 2*(2^(1/2)*9+t*(-5))
x'' : 0
y'' : -10
---------3---------
---------PP 5---------
unadopted modules: {FALL}
unsat mod : {BOUNCE, FALL}
unsat cons  : {y''=-10, y'=-4/5*y'-}
positive  : y-=0=>y'=-4/5*y'-
negative  :
t : 2^(1/2)*13/5
x : 2^(1/2)*(-3)/5
y : 0
x'  : -1
y'  : 2^(1/2)*32/5
x'' : 0
---------IP 6---------
unadopted modules: {}
positive  :
negative  : y-=0=>y'=-4/5*y'-
t : 2^(1/2)*13/5->2^(1/2)*97/25
x : 2^(1+1/2)+t*(-1)
y : t^2*(-5)+2^(1/2)*t*162/5+(-2522)/25
x'  : -1
y'  : t*(-10)+2^(1/2)*162/5
x'' : 0
y'' : -10
# number of phases reached limit
```

図 6.10　修正後の壁の付近で跳ねる質点のモデルの実行結果 (フェーズ 6 まで)

また，今回説明した上位の制約モジュールを作成する以外に，微小値 [14] を挿入する修正方法も存在する．厳密にはモデリングエラーを完全に修正することはできていないが，修正したモデルと HyLaGI で得られる実行結果を，図 6.11，図 6.12 に示す．

```
1  INIT <=> y = 10 & y' = 0 & x = 0 & x' = 1
2           & 0 < eps < 0.1 & [](eps' = 0).
3  FALL <=> [](y'' = -10).
4  MOVE <=> [](x'' = 0).
5  BOUNCE2 <=> [](y- = -eps => y' = -4/5 * y'-).
6  WALL <=> [](x- = 2^(1/2) => x' = -x'- & y' = y'-).
7
8  INIT, (FALL, MOVE) << (BOUNCE2, WALL).
```

図 6.11　微小値を加える修正後の壁の付近で跳ねる質点のモデル

微小値 eps を追加し，前件を置き換えることにより前件が同時に成立しないように変更した．今回の例では BOUNCE の前件を eps で置き換えた．変更した BOUNCE2 を示す．

$$\texttt{BOUNCE2 <=> []}(y\text{-} = eps \texttt{ => } y' = -4/5 * y'\text{-}).$$

BOUNCE2 の前件より，床は eps 沈んでいることになるので，シミュレーションでは図 6.13 のように先に壁に当たったとして処理される．

この修正により，図 6.12 に示した通り，シミュレーションは可能になっている．しかし，初期値を変更した時に再度シミュレーション不可能になってしまう可能性が残っているので，完全に修正できたとは言うことはできない．

```
------ Result of Simulation ------
---------parameter condition(global)-
p[eps, 0, 1] : (0, 1/10)
---------Case 1---------
---------1---------
---------PP 1---------
unadopted modules: {}
positive  :
negative  :
t : 0
eps : p[eps, 0, 1]
x : 0
y : 10
eps' : 0
x' : 1
y' : 0
x'' : 0
y'' : -10
---------IP 2---------
unadopted modules: {}
positive  :
negative  :
t : 0->2^(1/2)
eps : p[eps, 0, 1]
x : t
y : (t^2+(-2))*(-5)
eps' : 0
x' : 1
y' : t*(-10)
x'' : 0
y'' : -10
---------2---------
---------PP 3---------
unadopted modules: {MOVE}
unsat mod : {MOVE, WALL}
unsat cons : {x''=0, x'=-x'-}
positive  : x-=2^(1/2)=>x'=-x'-&y'=y'-
negative  :
t : 2^(1/2)
eps : p[eps, 0, 1]
x : 2^(1/2)
y : 0
eps' : 0
x' : -1
y' : 2^(1/2)*(-10)
y'' : -10
---------IP 4---------
unadopted modules: {}
positive  :
negative  : x-=2^(1/2)=>x'=-x'-&y'=y'-
t : 2^(1/2)->(2+p[eps, 0, 1]*1/5)^(1/2)
eps : p[eps, 0, 1]
x : 2^(1+1/2)+t*(-1)
y : (t^2+(-2))*(-5)
eps' : 0
x' : -1
y' : t*(-10)
x'' : 0
y'' : -10
---------3---------
---------PP 5---------
unadopted modules: {FALL}
unsat mod : {BOUNCE2, FALL, INIT}
unsat cons : {y''=-10, y'=-4/5*y'-}
positive  : y-=-eps=>y'=-4/5*y'-
negative  :
t : (2+p[eps, 0, 1]*1/5)^(1/2)
eps : p[eps, 0, 1]
x : 2^(1+1/2)+(-1)*(2+p[eps, 0, 1]*1/5)^(1/2)
y : -1*p[eps, 0, 1]
eps' : 0
x' : -1
y' : 5^(-1/2)*8*(10+p[eps, 0, 1])^(1/2)
x'' : 0
---------IP 6---------
unadopted modules: {}
positive  :
negative  : y-=-eps=>y'=-4/5*y'-
t : (2+p[eps, 0, 1]*1/5)^(1/2)
    ->(2+p[eps, 0, 1]*1/5)^(1/2)*13/5
eps : p[eps, 0, 1]
x : 2^(1+1/2)+t*(-1)
y : -26+t^2*(-5)+18*t*(2+p[eps, 0, 1]*1/5)^(1/2)
      +p[eps, 0, 1]*(-18)/5
eps' : 0
x' : -1
y' : t*(-10)+18*(2+p[eps, 0, 1]*1/5)^(1/2)
x'' : 0
y'' : -10
---------parameter condition(Case1)---------
p[eps, 0, 1] : (0, 1/10)
# number of phases reached limit
```

図 6.12　微小値を加える修正後の壁の付近で跳ねる質点のモデルの実行結果 (フェーズ 6 まで)

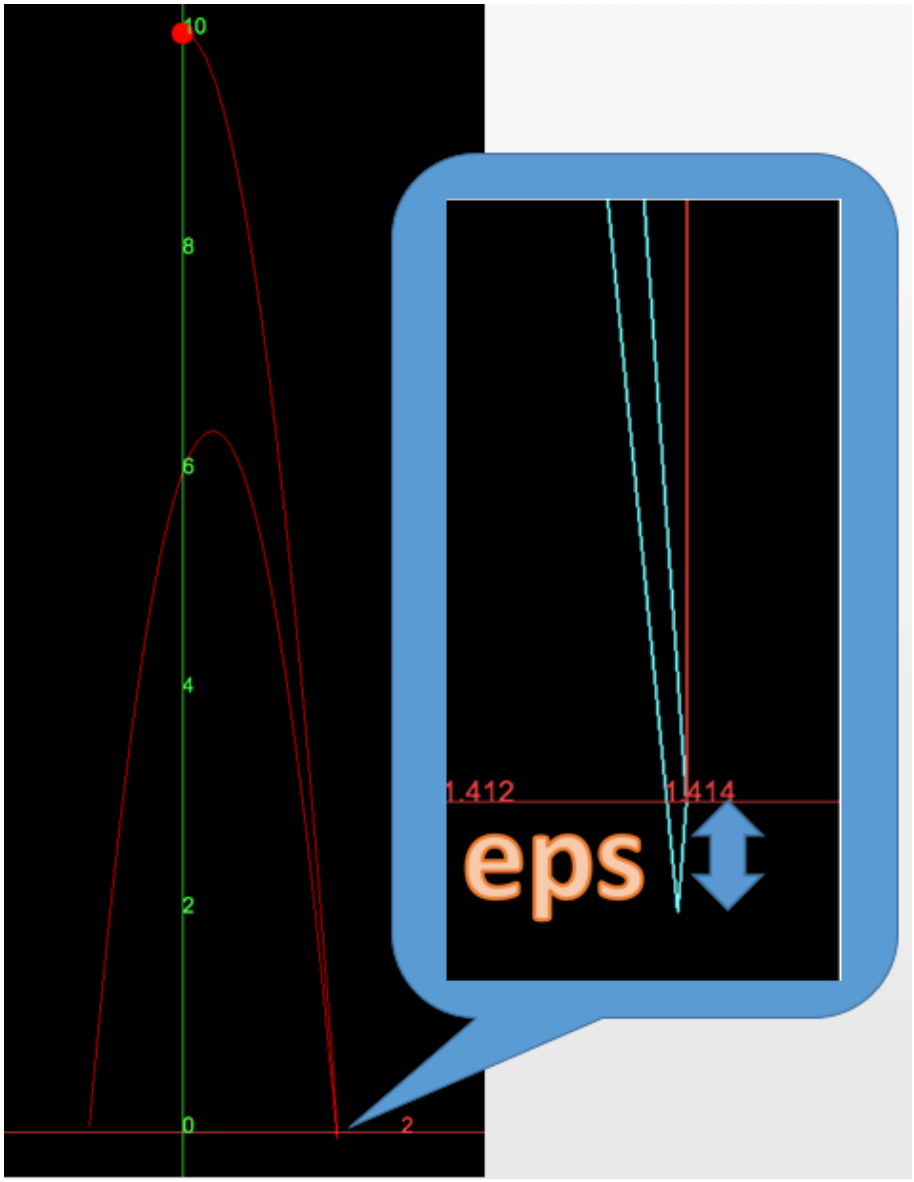

図 6.13　微小値を加える修正後の壁の付近で跳ねる質点のモデルのの解軌道

第 7 章

自動修正についての考察

第 6 章では提案手法で得られた検出結果より，プログラムの修正案を手動で作成したものを示した．しかし，本研究の最終目的は HydLa プログラムの修正である．しかし，プログラムの自動修正は現時点では難しいという結論に至った．本章では，現時点までで判明しているプログラムの修正方法についての紹介をおこない，本論文にて自動修正が困難だとする理由について説明する．

7.1 プログラムの修正の種類

現在考えられている，プログラムの修正方法について紹介する．今回紹介する修正方法以外にも修正方法は存在し，今回紹介する修正方法を含め，修正方法ついてはまだ十分な検証がなされていないのが現状である．

7.1.1 上位の制約モジュールの追加

図 6.9 のモデル修正に使用した方法である．主に制約過多のモデリングエラーが存在する場合に使用することができる．モデリングエラーが発生する状態の制約を新たに過不足なく設定することで，問題となる制約集合が成立しないようにする方法である．

7.1.2 制約式の追加

図 6.4 のモデル修正に使用した方法である．主に制約不足のモデリングエラーが存在する場合に使用することができる．該当する制約モジュールの後件に連言で不足している変数の制約式を追加することで，制約不足の状態を解消する方法である．

7.1.3　微小値の挿入

図 6.11 のモデル修正に使用した方法である．主に制約過多のモデリングエラーが存在する場合に使用することができる．第 6 章でも述べているが，初期値の変更することで問題は再発する．問題は残ったままなので厳密には修正とは言えないが，3 体同時衝突問題などはこのようにしてモデル化することがあるので，修正案の一つとして挙げた．

7.1.4　制約階層の変更

制約階層を適切に設定することで，プログラムを修正する方法である．図 7.1 に HydLa プログラムの例を示す．示すプログラムは，モデリングエラーを含む床で跳ねる質点のモデルである．

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

図 7.1　モデリングエラーを含む，床で跳ねる質点のモデル

このモデルは，図 2.1 に示したモデルと異なり，制約モジュール FALL と BOUNCE の間に優先度が存在しない．これだと，質点が接地した時に変数 y に関する制約が過多となるモデリングエラーを含んでいる．このモデルの制約階層を，図 2.1 の制約階層になるように適切に変更することで，モデリングエラーは修正される．

7.1.5　制約モジュールの分割

一つの制約モジュールを 2 つ以上の制約モジュールに分割することで，適切な制約モジュール集合を選択できるようにする方法である．図 3.8 のモデルを図 6.4 に修正するのではなく，図 3.7 のモデルになるように，制約モジュール FALL_MOVE を 2 つに分割することで，制約不足のモデリングエラーを修正することができる．

7.2　自動修正が困難な理由

主な理由は，HydLa プログラムの記述の自由度が高いことで，プログラムからは記述者の意図を正確に推測することができなかったことにある．本論文内で紹介したモデリングエラーの修正方法として，複数の方法をそれぞれ例に挙げた．それらの修正方法の中でどれが一番適しているかを自動的に判定することは困難であり，ユーザーしか正確にはわからない．図 2.1 のモデルを本論文では，床を跳ねる質点のモデルと説明したが，制御系のシステムでも同じような HydLa プログラムで表現することが可能である．その場合，プログラムから静的に判定することは不可能である．それにより，複数の修正案を作成してもどの修正案が適当であるかを順位付けして提示することが難しく，全ての修正案を提示することは現実的ではない．また，各モデルにおいてそれぞれのテンプレートのようなものが作られれば，そのテンプレートの対象範囲内でのモデルに関しては自動修正が可能であると推測しているが，全てのプログラムに適用できる自動修正は現実的ではないと考えている．

第 8 章

まとめと今後の課題

8.1 まとめ

本論文では，HydLa のモデリングエラーを HydLa プログラムから静的に検出する手法を提案した．本論文に提案した手法は，シミュレーション実行を介さずにエラーの原因を探すことが可能であり，これによりモデル化の手間を大きく削減する可能性を持つ．実行時にエラーが出力されてもエラーの原因がよくわからなかったプログラムに対して，エラーの原因を探る一つの方策になると期待される．

8.2 今後の課題

本研究の最終目的は HydLa プログラムの修正である．第 6 章にて示したようなプログラムの修正案を自動で出力するところまでが本研究の目的である．本論文では，原因の静的検出に止まり，プログラムの自動修正には至らなかった．原因としては主に，HydLa プログラムの記述の自由度が高いことで，プログラムからは多くの記述者の意図を推測することができなかったことにある．それにより，とても多くの修正案が導かれてしまい，その修正案に妥当性の順位をつけることができなかった．それでもいくつかの修正案のグループを作成することができたが，それを自動的に作成，記述者に提示する手法を提案するには至らなかった．本論文では至らなかったが本研究の最終的なゴールは，文献 [5] のようなプログラム自動修正を HydLa プログラム上で実現することである．そのための検証をおこなっていくことが今後の課題である．

謝辞

本研究を進めるにあたり様々な方の指導，助言をいただきました．まず，ご指導を賜わった上田和紀教授に深く感謝致します．また，学部生時代より HydLa について何度も助言をくださった松本翔太さんに深く感謝いたします．同じ HydLa 班として一緒に HyLaGI のデバッグに苦しんだり，色々なことを相談した別納健市くんにも深く感謝いたします．また，班は違いましたが，色々な相談をしたり，息抜きに邪魔をしにいったりした松澤望くんにも深く感謝いたします．来年からもよろしく．私にはなかった独特の世界観で，いつも興味深い話をしてくれた恒川雄太郎くんにも深く感謝いたします．わからないことがあった時，一緒に考えてくれた坂爪裕也くんにも深く感謝いたします．色々な刺激をもらった上田研究室の後輩の皆様にも深く感謝いたします．最後に，入学以前から現在に至るまで，常に支えて頂いた家族に深く感謝致します．

2018 年 1 月　小山 峻平

参考文献

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

[2] 上田和紀, 石井大輔, 細部博史 : 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.

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

[4] Gupta, V., Jagadeesan, R., Saraswat, V., Bobrow, D. G., Programming in Hybrid Constraint Languages, In Hybrid Systems II, Vol. 999 of LNCS, Springer-Verlag, 1995, pp. 226–251.

[5] 網代 育大, 上田 和紀 : Kima: 並行論理プログラム自動修正系. コンピュータソフトウェア, Vol. 18, No. 0, 2001, pp. 122–137.

[6] 和田亮 : ハイブリッド制約言語 HydLa の対話的な実行方式の実装, 早稲田大学大学院基幹理工学研究科, 修士論文, 2014.

[7] 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, SSV2008(第 5 回システム検証の科学技術シンポジウム), 2008.

[8] 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, Vol. 30, No. 4, 2013, pp. 18–35.

[9] 河野文彦 : リスト記法の導入と動的な制約集合導出によるハイブリッド制約言語 HydLa とその処理系のスケーラビリティ向上, 早稲田大学基幹理工学研究科, 修士論文, 2014.

[10] S. C. Johnson : Lint, a C Program Checker, Comp. Sci. Tech. Rep., Bell Laboratories, 1978.

[11] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, Raymie Stata : Extended static checking for Java. In Proceedings of

the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). SIGPLAN Notices, Vol. 37, No. 5, 2002, pp. 234–245.

[12] Borning, A., Freeman-Benson, B., Wilson, M. : Constraint Hierarchies, Lisp and Symbolic Computation, Vol. 5, No. 3, 1992, pp. 223–270.

[13] SymPy Development Team : SymPy main Page. `http://www.sympy.org/en/index.html`, (最終参照 2018-01-19).

[14] 若槻 祐彰, 松本 翔太, 伊藤 剛史, 和田 努, 上田 和紀 : ハイブリッド制約処理系 HyLaGI による微小誤差を用いたモデル解析, 日本ソフトウェア科学会第 32 回大会, 2015.

発表文献

[1] 小山峻平，上田和紀，松本翔太: ハイブリッドシステムモデリング言語 HydLa におけるモデリングエラーの体系化，人工知能学会第 30 回全国大会，1F3-5，2016（登壇発表）.

[2] 小山峻平，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 における静的検証，第 14 回 ディペンダブルシステムワークショップ (DSW 2016)，2016（ポスター発表）.

[3] 小山峻平，上田和紀: 制約階層に基づくハイブリッドシステムモデリング言語 HydLa の静的誤り検出手法，情報処理学会第 80 回全国大会，2018 （発表予定）.

Appendix ソースコードの変更概略

以下に，今回の実装のために行ったソースファイルの変更の概略を示す．

- core/main.cpp
  debug_constraint オプションの追加
- debug/debug_main.cpp
  main.cpp から始めに呼ばれる．それぞれの関数はここから呼び出す．
- debug/json_parser.cpp
  HydLa プログラムは parser/ParseTree.cpp でパースされるが，それを使いやすい形に成形する．
- debug/solve_sym.cpp
  手法本体の実装箇所．
- debug/solve_sym.py
  python 側の実装．細かいところは全て debug/solve_sym.cpp にて実装しているので，単純な命令のみ．

このソースコードは以下の場所にて公開されている．
https://github.com/HydLa/HyLaGI