1F3-5

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

Categorizing modeling errors of hybrid modeling language HydLa

小山峻平 [*1] Shunpei KOYAMA　松本翔太 [*1] Shota MATSUMOTO　上田和紀 [*2] Kazunori UEDA

[*1]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
Graduate School of Fundamental Science and Engineering, Waseda University

[*2]早稲田大学理工学術院
Faculty of Science and Engineering, Waseda University

HydLa is a declarative language for modeling hybrid systems. HydLa allows us to describe hybrid models using differential equations, inequalities and guards. Simulation of a HydLa model sometimes returns an unexpected result, but it is not easy to find causes of modeling errors. This is because the output of HydLa execution is complicated and the currently available implementation does not support static detection of modeling errors. The goal of this research is (i) to categorize modeling errors of HydLa to find errors statically and (ii) to propose a method to detect the errors.

1. 背景と目的

ハイブリッドシステムとは，連続変化と離散変化が交互に起こるシステムのことである．ハイブリッドシステムのモデリング言語として HydLa 言語が存在する．HydLa 言語は，プログラムを専門としない技術者が，数式や論理式を書きそれに優先順位をつけることで，簡潔にハイブリッドモデルを記述できることを目的に設計されている．

現状の HydLa における問題点として，もっともらしい HydLa プログラムを記述したにもかかわらず制約の不足などの要因により意図した軌道が求まらない場合があり，その時にハイブリッドモデルの設計に問題があるのか，それともハイブリッドモデルを表現した HydLa プログラムに問題があるのかがわからないという点を挙げることができる．

本研究は，静的にモデリングエラーの原因を推定することを目的としている．モデリングエラーとは，設計者の意図したモデルがあり，そのモデルに沿って記述したにもかかわらず意図した軌道を導出することができない場合のことを表す．本研究では，HydLa プログラミング時に比較的よく起こるモデリングエラーの具体例を挙げ，その原因やエラー回避策を体系化し，エラー体系を提案する．そして，エラーの検出手法を提案，実装の方針について述べる．

本研究は処理系で実行することでのみ得ることのできたエラーを，実行を必要とせずにのみ検出する手法を提案することで，実行に時間のかかる複雑なプログラムのデバッグに効果をもたらすと期待される．

2. HydLa

HydLa[1] 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宣言型言語である．ハイブリッドシステム [2] とは，時間経過に伴い，状態や方程式が連続変化と離散変化を繰り返す系のことをいう．HydLa は制約階層に基づいた言語であり，制約を方程式や論理式を用いて記述しそれらの制約を制約モジュールとしてまとめて表現することができる．そして制約モジュール間に優先度を設定することで，ハイブリッドシステムを簡潔に表現することができる．

図 1 に HydLa プログラムの例を示す．示すプログラムは，床でバウンドする質点のモデルである．

```
INIT <=> y=10 & y'=0.
FALL <=> [](y''=-9.8).
BOUNCE <=> [](y-=0 => y'=-4/5*y'-).

INIT, FALL << BOUNCE.
```

図 1: 床で跳ねる質点のモデル

この HydLa プログラムでは，初期状態 `INIT`，自由落下 `FALL`，床でバウンド `BOUNCE` に関する制約を各制約モジュールにて記し，それに優先度をつけることで床でバウンドする質点のモデリングを行っている．制約に時相論理演算子 `[]` をつけることで時刻 $t = 0$ 以降に適用される制約であることを表す．

HydLa の対象ユーザには，プログラミングを専門としない技術者も含まれる [3]．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．

3. モデリングエラーの体系化

Acumen[4] や HydLa のようなハイブリッドシステムモデリング言語では，静的にエラーを検出する手法に関して，あまり研究が進んでいない．

そこで本研究では，現状の HydLa の問題点である，詳細な原因がわかりにくいモデリングエラーを体系化する。体系化するにあたって，モデリングエラーをその状況と内容によって分類してその原因を分析した．はじめに実行時の結果に基づいて大きく二分した (表 1(A))．

図 1 のモデルでバウンドするフェーズの実行結果をもとに

連絡先: 小山峻平，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, 〒 169-8555 新宿区大久保 3-4-1 63 号館 5 階 02 号, 03-5286-3340, koyama(at)ueda.info.waseda.ac.jp

概要 (A)	詳細	原因	検出法	回避法
実行ができない	execution stuck(B)	連続変数に対する離散変化制約 (1)	変数をその微分値に関する制約を比較	制約を追加する
		制約過多	同じ変数に関する制約を比較	制約を減らす
挙動がおかしい (C)	変数に意図しない数が入る (D)	ガード制約式関係演算子ミス	制約が採用されているフェーズを確認	式の等号不等号を見直す
		ガード制約不足	制約集合を確認	制約を追加する
		制約不足	制約集合を確認	制約を追加する
		微分値制約不足 (3)	制約集合を確認	変数を追加する
		ガード前後の変数の一意性 (2)	ガード前後の制約式を比較	左極限を採用する
	優先度ミス	優先度ミス	出力結果を確認	優先度順を変更する

表 1: HydLa のモデリングエラーの体系表

して，分類に関して述べる．まず，モデリングエラーのないプログラムの実行結果を図 2 に示す.

```
---------PP 3---------
unadopted modules: {FALL}
unsat mod : {BOUNCE, FALL}
unsat cons : {y''=-(98/10), y'=(-4)/5*y'-}
positive  : y-=0=>y'=(-4)/5*y'-
negative  :
t : 10/7
y : 0
y' : 56/5
```

図 2: 実行例 (一部抜粋)

図 2 はこのフェーズ以降も続き，意図している振る舞いが求められる．

実行ができないとは，たとえば図 3 のような出力が得られる場合のことを示す.

```
---------PP 3---------
unadopted modules: {}
unsat mod : {BOUNCE, FALL}
    {BOUNCE}
unsat cons : {y''=-(98/10), y=3, y'=(-4)/5*y'-}
    {y=3, y'=(-4)/5*y'-}
positive  : y-=0=>y=3&y'=(-4)/5*y'-
negative  :
t : 10/7
# execution stuck
```

図 3: 実行ができない例 (一部抜粋)

図 3 は，図 1 のプログラムの制約 BOUNCE に，新たに方程式 y=3 を付け足したものである．本来は先のフェーズまで実行しようとしていたが，フェーズ 3 で実行が止まり"execution stuck" を返すプログラムの出力である．

実行はできるが挙動がおかしいとは，たとえば図 4 のような出力が得られる場合のことを示す.

図 4 は，図 1 のプログラムに，壁の制約 WALL を付け足したものである．本来は変数 x の値は定数であるが，パラメータ p[x, 0, 3] が代入されてしまうプログラムの出力である．

```
---------PP 3---------
unadopted modules: {}
unsat mod : {BOUNCE, FALL}
unsat cons : {y''=-(98/10), y'=(-4)/5*y'-}
positive  : y-=0=>y'=(-4)/5*y'-
negative  :
t : 10/7
x    : p[x, 0, 3]
y    : 0
x'    : p[x, 1, 3]
y'    : 56/5
```

図 4: 挙動がおかしい例 (一部抜粋)

表 1 の中で，実行ができない場合である execution stuck に関してはモデリングエラーの原因で二分した (表 1(B)).

連続変数の離散変化エラーは，採用される制約集合内で同一変数に関して離散変化制約と連続変化制約が同時に設定されてしまうことにより起こるエラーであり，制約過多のエラーは採用される制約集合内にて同一変数に関する制約が複数存在し，それぞれが矛盾するときに起こるエラーである．

次に，実行はできるが挙動がおかしい場合については処理系の出力結果の詳細で二分した (表 1(C))．詳細のうち意図しない値が入る場合に関しては，モデリングエラーの原因で更に五分した (表 1(D)).

ガード制約式関係演算子ミスのエラーは，制約モジュール内のガード条件前件の等号，不等号が適当でないときに起こるエラーのことである．ガード制約不足のエラーは，制約モジュール内のガード条件の制約式が不足しているときに起こるエラーのことである．制約不足のエラーは，採用される制約がフェーズ間で変わった際，これまで制約されていた変数に関しての制約がなくなるときに起こるエラーのことである．微分値不足のエラーは，変数に関する制約を設定したときにその変数の微分値に関する制約がないときに起こるエラーのことである．ガード前後の変数の一意性のエラーは，ガード条件前後で同じ変数に関する制約がありそれらが矛盾しているときに起こるエラーのことである．

優先度ミスのエラーは，プログラム内で設定する制約間の優先度の順序が異なることで起こるエラーのことである．

それぞれの体系について原因とそのエラーの回避方法についても体系表に記載した.

4. エラー検出手法

各エラーを静的に検出する手法とそのエラーに対する回避法に関して考察を行い，表 1 にまとめた．本節では，エラーの検出を一連の流れで行う手法に関して考察する．プログラム内の以下の 3 点に着目する.

1. 個々の制約モジュール単体
2. 優先度が付けられている 2 つの制約モジュール A と B
3. 制約モジュールの優先度をつけている部分

それぞれを対象とした検出手法を述べる.

4.1 個々の制約モジュール単体

まず，各制約モジュール内で制約されている変数をリストにする．ここで作成したリストは以降の変数の比較において用いるものである.

リストより，制約 INIT のような時相論理演算子 always[] がついていない制約を除いた制約モジュールで，(x,x') または (x,x) のように一つの変数に複数の制約があるかどうかを確認する．（x,x）のようにそれらが全く同じ変数の場合は，矛盾している (左極限付きの変数は定数として扱い，変数に含まない) なら警告する．必ず矛盾するかが保証出来ない場合，エラーの可能性があることだけ警告する.

ここで検出できるエラーは，矛盾する変数がガード前後にあるならガード前後の変数の一意性エラー，ガード後にあるなら制約過多エラーもしくは，連続変数に対する離散変化制約エラーである.

4.2 優先度付けられている 2 つの制約モジュール A と B

制約モジュールの優先度に基づいて，優先度付けられている 2 つの制約モジュールの変数のリストを比較する．ここでは優先度を (A << B) として考えていく．ここで検証することは 2 点であり，

1. A 内で，x'' が制約されているとき，B 内で x を制約していれば警告.
2. 制約されている変数の包含関係が $A \subseteq B$ もしくは，$A \cap B = \emptyset$ となっていなければ警告.

として検証する.

1. では複数の制約モジュールにまたがって状態で x'' と x が設定されているなら x' に関する制約がなくてはならないことを利用して，x' が制約されているなら連続関数に対する離散変化制約エラー，x' が制約されていないなら微分値不足エラーを検出できる．2. では A と B が矛盾した時に A が採用されないことを利用して，$A \supset B$ であるなら B の変数不足，A の変数過多に関するエラーを検出できる.

ここで検出できるエラーは，微分値不足エラーと，変数不足，過多に関するエラー，連続変数に対する離散変化制約エラーである.

4.3 制約モジュールの優先度をつけている部分

最後に，優先度が付いている制約モジュールの集合について比較する．優先度付き制約モジュールが複数ある場合を扱う．(例えば A << (B, C) など) ここで検証することは 1 点で，

1. 時相論理演算子 [] がついたガード条件の無い制約より下の優先度に，時相論理演算子 [] がついたガード条件のある制約があり, 両制約モジュールが同じ変数に関して制約している時，エラーの可能性があるとして警告する.

として検証する.

ここで検出できるエラーは，優先度エラーである.

5. 例題を用いた検出と修正例

実際に複数のモデリングエラーを含むプログラムの例を用いて，提案手法によりエラー検出を行い検出結果をもとに手動で修正を加えることで，プログラムが適当なものになるかを評価した.

今回使用したプログラムの例は，気球のシミュレーションである．このモデルは，気球を上昇させるモデルであり気球の燃料を補給するタイミングを制御している．燃料タンクの最大容量に幅をもたせることで，最大容量に違いでどのような挙動を示すかをシミュレートするものである．図 5 のプログラムを記述し処理系にて実行したところ，最後まで実行ができなかった.

```
INIT <=> x = 0 & x' = 0 & fire = 2
         & tank = full & tank' = -1 & 5<=full<=7.
FULL <=>[](full'=0).
UP <=> [](x''=0.5).
DOWN <=> [](fire=0 ∨ fire=3=> x''=-1).
FIRE <=> [](fire'=0).
TANK <=>[](tank''=0).
HIGHCUT <=> [](x-=20 => fire=0 & tank'=0).
SPEEDCUT <=> [](fire=1 & x'-=1 => fire=0 & tank'=0).
FUELEMP <=> [](tank-=0 & fire-=2 => fire=3 & tank'=0).
HIGHBURN <=> [](x'-=-1 & fire-=0=> fire=1 & tank'=-1
                                    & tank=full).
EMPBURN <=> [](x'-=-1 & fire-=3 => fire=2
                              & tank'=-1 & tank=full).

INIT, FULL, UP << DOWN, (FIRE, TANK) <<
(HIGHCUT,FUELEMP, SPEEDCUT, HIGHBURN, EMPBURN).
```

図 5: 気球の上昇のシミュレーション

このプログラムを検出手法を用いてエラー箇所を検出する．まず，各制約モジュール内で制約されている変数をリストにした．以下に一部を示す.

- SPEEDCUT：fire, tank', fire
- HIGHBURN：fire, tank', tank
- EMPBURN：fire, tank', tank

作成したリストを確認すると，制約 HIGHBURN，EMPBURN にて tank,tank' と 2 つの変数 tank に関する制約が定義されていること，制約 SPEEDCUT にて fire，fire と 2 つの変数 fire に関する制約が定義されていることがわかる．よって，この制約 HIGHBURN，EMPBURN の変数 tank に関する制約は連続変数に対する離散変化制約エラーである可能性があることを警告し，制約 SPEEDCUT の変数 fire に関する制約はガード前後の変数の一意性エラーである可能性があることを警告する.

次に制約モジュールの優先度に基づいて，優先度付けられている 2 つの制約の変数のリストを比較する．優先度が付いている制約モジュールの関係は UP << DOWN, (FIRE, TANK) << (HIGHCUT, OILEMP, SPEEDCUT, HIGHBURN, EMPBURN) となっているため，それぞれ 2 つの制約の組にして確認する．変数のリストを元に検証すると，

- TANK << HIGHBURN

- TANK << EMPBURN

の2つの組み合わせで，(A << B の時) 制約 A 内で，tank'' が制約されているとき，制約 B 内で tank を制約していることがわかる．よって，この制約 TANK と制約 HIGHBURN，制約 TANK と制約 EMPBURN の間の変数 x に関する制約は，微分値制約不足エラーである可能性があることを警告する．

もうひとつの確認事項である，制約されている変数の包含関係が $A \subseteq B$ もしくは，$A \cap B = \emptyset$ となっていない，という条件は当てはまらないので今回のプログラムでは警告をしない．

最後に，優先度が付いている制約モジュールの集合について比較する．制約モジュールの優先度を定めている箇所より確認すると，時相論理演算子 [] がついたガード条件の無い制約より下の優先度に，時相論理演算子 [] がついたガード条件のある制約がは無いことがわかる．よって，確認している条件には当てはまらないので今回のプログラムでは警告をしない．

以上の結果より，

- 制約 HIGHBURN の変数 tank に関する制約は連続変数に対する離散変化制約エラー (表 1(1)) である可能性があること
- 制約 EMPBURN の変数 tank に関する制約は連続変数に対する離散変化制約エラー (表 1(1)) である可能性があること
- 制約 SPEEDCUT の変数 fire に関する制約はガード前後の変数の一意性エラー (表 1(3)) である可能性があること
- 制約 TANK と制約 HIGHBURN，制約 TANK と制約 EMPBURN の間の変数 tank に関する制約は，微分値制約不足エラー (表 1(2)) である可能性があること

というエラー原因を検出できる．

検出したエラー原因を元に，プログラマによる修正を加えたプログラムを図 6 に示す．

```
INIT <=> x = 0 & x' = 0 & fire = 2 & tank = full
                        & tank1 = -1 & 5<=full<=7.
                                       //tank' を tank1 に
FULL <=>[](full'=0).
UP <=> [](x''=0.5).
DOWN <=> [](fire=0 ∨ fire=3=> x''=-1).
FIRE <=> [](fire'=0).
TANK <=>[](tank'=tank1).
                                       //tank' を tank1 に
TANK1 <=>[](tank1=0).
                                       //tank1 の制約
HIGHCUT <=> [](x-=20 => fire=0 & tank1=0).
SPEEDCUT <=> [](fire-=1 & x'-=1 => fire=0 & tank1=0).
FUELEMP <=> [](tank-=0 & fire-=2 => fire=3 & tank1=0).
HIGHBURN <=> [](x'-=-1 & fire-=0=> fire=1 & tank1=-1
                                            & tank=full).
EMPBURN <=> [](x'-=-1 & fire-=3 => fire=2 & tank1=-1
                                            & tank=full).
                                       //tank' を tank1 に

INIT, FULL, UP << DOWN, (FIRE, TANK, TANK1)
<< (HIGHCUT, FUELEMP, SPEEDCUT, HIGHBURN, EMPBURN).
```

図 6: 気球の上昇のシミュレーション (修正後)

修正したプログラムは指定したフェーズまで実行ができたことを確認した．図 7 に実行結果のプロットを示す．

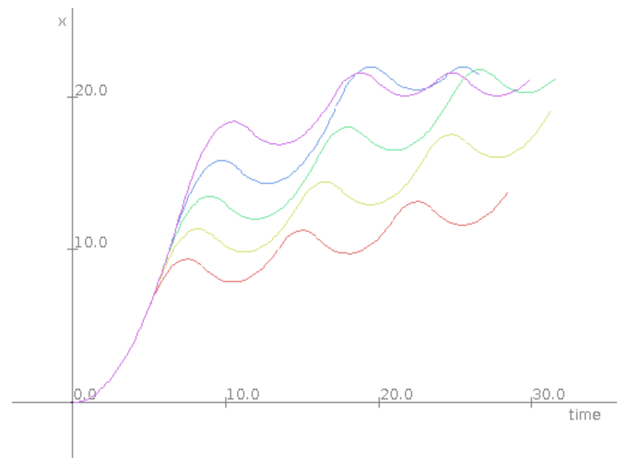

図 7: 気球の上昇のシミュレーション結果 (修正後)

6. まとめと今後の課題

6.1 まとめ

本論文では，HydLa のモデリングエラーの一部分の体系化と，そのモデリングエラーをプログラムから静的に検出する手法を提案した．実際のプログラムを用いてエラーを検出して，手動でプログラムを変更することで評価を行った．本論文で提案した手法は，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思われる．

6.2 今後の課題

本論文で提案した検出手法を今後処理系に組み込むことによって，ユーザがモデリングエラーを推測するための指標となることが期待される．以下に今後の課題を述べる．

1. 処理系への実装
2. 検出手法の改良
3. さらなるエラー原因の考察
4. プログラムの修正方法に関する提案手法の考察

本論文では，検出手法を提案することにとどまった．モデリングエラーの検出を行ったが，その後のプログラムの修正についてはプログラマ依存であるのが現状である．よって，手法の改良，処理系への実装をおこない，そのプログラムの修正方法に関しても何らかの情報を提供することができるようになることで HydLa の開発環境をよりよくしていくことを目的として今後研究をおこなう．

参考文献

[1] 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.

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

[3] 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.

[4] Taha, Walid, et al. A core language for executable models of cyber-physical systems (preliminary report). In: Distributed Computing Systems Workshops (ICDCSW), 2012 32nd International Conference on. IEEE, 2012. pp. 303-308.