2J1-1

制約の静的解析を用いたHydLa処理系の最適化

Optimization of a HydLa implementation using static analysis of constraints

河野文彦*1 Fumihiko KONO　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 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

Hybrid systems are dynamical systems with both continuous and discrete changes of states. Hybrid systems are applicable to various fields. HydLa is a hybrid system modeling language based on constraint hierarchy. In HydLa, models are written as constraint hierarchies. The trajectories of a HydLa program are defined by maximul consistent module sets. We are implementing Hyrose to simulate and verify HydLa programs. Hyrose simulates HydLa programs without errors by using symbolic formula manipulation. It is important to optimize Hyrose because the simulation of complex models takes long time. The bottleneck of Hyrose is in the consistency checking of constraints. We propose a static analysis method for calculating necessary and sufficient conditions of inconsistency that can be checked efficiently at runtime, and use these conditions to optimize the consistency checking of constraints. This paper describes the optimization method based on constraint analysis, and evaluates performance improvements using benchmark programs.

1. はじめに

ハイブリッドシステム [3] とは離散変化と連続変化を含むシステムであり, 物理学をはじめ, 制御工学や生命工学など幅広い分野に応用が可能である.

HydLa[7] は制約階層 [1] に基づくハイブリッドシステムモデリング言語であり, プログラミングを専門にしない技術者の利用を目標としている. HydLa では論理記号や数式によって制約を定義し, 定義した制約に優先度を与えることにより制約階層を記述することで, ハイブリッドシステムのモデリングを行う. HydLa プログラム内で定義された制約を制約モジュールと呼び, 制約階層を満たす制約モジュールの集合を解候補制約モジュール集合と呼ぶ. HydLa における制約階層では制約モジュールに優先度をつけ, 半順序集合を定義し, 制約モジュールを採用する際の部分集合が upward closed となるように制約を採用する [2]. HydLa プログラムの仕様を満たす各変数の軌道を解軌道と呼び, 解軌道が HydLa プログラムの実行結果となる. HydLa では制約階層において極大かつ無矛盾な解候補制約モジュール集合を採用することで解軌道を求める.

HydLa の処理系 Hyrose[4] は HydLa プログラムのシミュレーションを行うことで, 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に開発されている. Hyrose は非決定実行により HydLa プログラムが許すすべての解軌道を求めることや, 数式処理による精度保証や記号実行による誤差の無いシミュレーションなどを行えるが, 挙動の複雑なシステムに対して実行時間の増大が激しいという問題も抱えているため, Hyrose を高速化することは重要である. Hyrose の処理全体に対する各処理の割合を調べたところ, ボトルネックは制約モジュール集合の無矛盾性判定であることが分かった. そこで制約を静的解析し, その解析結果を利用して制約モジュール集合の無矛盾性判定の高速化を行った. 本稿では, 制約の静的解析法とその解析結果の利用法を提案し, その実装について述べ, その後, 評価結果を述べる.

連絡先: 河野文彦，早稲田大学大学院基幹理工学研究科情報理工学専攻，〒 169-8555 新宿区大久保 3-4-1 63 号館 5 階 02 号，03-5286-3340，kouno(at)ueda.info.waseda.ac.jp

2. HydLa の記述例

床を跳ねる質点をモデリングした HydLa プログラムを図 1 に示す.

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

INIT, FALL << BOUNCE.
```

図 1: 床を跳ねる質点の HydLa プログラム

HydLa プログラム中の変数はすべて時刻に関する関数であり, y は $y(t)$ を意味する. HydLa プログラムに使用される記号を以下に示す.

- `<=>`：制約モジュールを定義する.
- 変数 `'`：変数の時間微分を表す.
- 変数 `-`：各時刻での変数の左極限. prev 演算子と呼ぶ.
- `[]`：時相演算子 always ($\forall t \geq 0$) を表す. `[]` がない制約は時刻 0 でのみ成立する制約となる.
- `=>`：左辺をガード条件とする条件つき制約を表す.
- `,`：制約モジュールに優先度をつける. この記号の左辺と右辺の制約モジュールは同じ優先度となる.
- `<<`：制約モジュールに優先度をつける. この記号の左辺の制約モジュールよりも右辺の制約モジュールが強い優先度となる.

図 1 では `INIT` と `FALL`, `BOUNCE` という制約モジュールを定義し, `INIT, FALL << BOUNCE` で制約モジュールに優先度をつけ, 制約階層を定義することで, 床を跳ねる質点のモデルを記述している. `INIT` は質点の初期位置と初期速度を表し, `FALL` は自然落下を表し, `BOUNCE` は質点の床との衝突を表している.

図 1 のプログラムの実行結果である y の軌道を図 2 に示す.

3. Hyrose の最適化手法

Hyrose は HydLa の処理系であり, HydLa プログラムのシミュレーションを行い, 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ている.

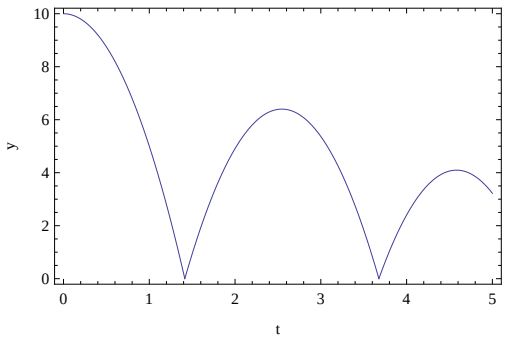

図 2: 図 1 のプログラムにおける y の軌道

Hyrose は離散変化の計算フェーズであるポイントフェーズ (PP) と連続変化の計算を行うインターバルフェーズ (IP) を繰り返すことで, HydLa プログラムのシミュレーションを行っている.

3.1 Hyrose のボトルネック

Hyrose で例題をシミュレーションしたときの各処理にかかった時間の割合を図 3 に示す.

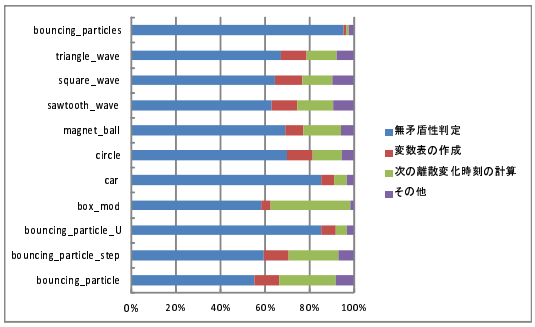

図 3: Hyrose の各処理の割合

シミュレーションした例題について非決定実行で 20 フェーズ実行するときの情報を表 1 に示す. 表 1 中の N_b は解軌道の数, N_g はガード条件の数, N_s は解候補制約モジュール集合の数を表わす. また, bouncing_particle という文字列を b_p と省略している.

表 1: シミュレーションした例題

プログラム名	N_b	N_g	N_s	簡単な説明
b_ps	1	5	32	複数の跳ねる質点
triangle_wave	1	1	2	三角波
square_wave	1	1	4	方形波
sawtooth_wave	1	1	2	のこぎり波
magnet_ball	183	3	14	磁場と重力場中にある質点
circle	1	1	4	円の中を跳ねる質点
car	1	6	4	車の運転制御
box_mod	1	4	4	障害物がある中で跳ねる質点
b_p_U	1	1	2	曲面の床を跳ねる質点
b_p_step	1	2	2	段差のある床を跳ねる質点
b_p	1	1	2	床を跳ねる質点 (図 1)

図 3 を見ると無矛盾性判定がボトルネックとなっていることが分かる. 無矛盾性判定とは制約モジュール集合が無矛盾であることを判定するための処理である. 無矛盾性判定では, 調べる制約モジュール集合に含まれるすべての制約モジュールの定義を確認し, それらの定義に含まれるすべてのガード条件の成否判定を行う. その後, 採用すべきすべての制約の連言を算出し, その制約の連言について矛盾か無矛盾かの判定を行っている.

3.2 Hyrose の最適化手法の提案

Hyrose のボトルネックとなっている無矛盾性判定を高速化するための新たな無矛盾性判定の方法を本節で示す.

新たな無矛盾性判定を行うにあたって, まずは各制約モジュール集合 MS に含まれるすべての制約の連言 LC に対して式 (1) を満たすような C を求める.

$$C \Leftrightarrow \neg LC \tag{1}$$

ただし, C は変数として prev 演算子を伴うもののみが出現する式とする. このような C を以降 Prev False Conditions (PFC) と呼ぶことにする. このような PFC を調べることで, ある制約モジュール集合の無矛盾性の判定は PFC の成否判定に置き換えることができる.

prev つき変数は各時点での左極限を表しており, それらの値は前のフェーズから得られる. 従って, PFC の成否は前のフェーズからの情報のみに依存するため, これにより, 従来の無矛盾性判定において, (i) 制約モジュール集合に含まれるすべての制約モジュールの定義を確認する, (ii) それらの定義に含まれるすべてのガード条件の成否判定を行う, (iii) 採用すべきすべての制約の連言を算出する, (iv) その制約の連言について矛盾か無矛盾かの判定を行う, というステップを踏んで行っていた処理を PFC の成否判定だけで置き換えることができるため, 実行時間を短縮できると期待できる.

HydLa プログラムでは時刻が 0 の場合と時刻が 0 より大きい場合とでは時相演算子 (`[]`) がつかない制約の扱いが異なるが, 時刻 0 では左極限が存在しないので, PFC を得られたとしても利用することができない. 従って本稿で述べるすべてのアルゴリズムの適用は時刻が 0 より大きい場合を対象とする.

3.3 制約の解析アルゴリズム

各解候補制約モジュール集合に対応する PFC を求めるアルゴリズムを図 4 に示す. このアルゴリズムではガード条件に prev つきでない変数 (通常変数と呼ぶ) が含まれていた場合, PFC に通常変数が含まれてしまう. その場合, 目的としていた PFC を求めることができない. 従ってこのアルゴリズムで PFC を求める場合, ガード条件に通常変数が含まれていないことが前提となる.

図 4 内に出現する関数についての説明を以下に示す.

- $CollectAsk(MS)$: 制約モジュール集合 MS 内のガード条件を含む制約を集めて返す関数. 本稿においては時相演算子がついた制約が対象となる.
- $CollectTell(MS, A_+)$: 制約モジュール集合 MS 内のガード条件を含まないすべての制約と A_+ 内のすべての後件の連言を返す関数. 本稿においては時相演算子がついた制約が対象となる.
- $CalculateFalseConditions(S \wedge G)$: $S \wedge G$ が矛盾する条件を返す関数. 図 5 にてアルゴリズムの詳細を示す.

図 4 は制約モジュール集合 MS に対応する矛盾の条件を求めるアルゴリズムである. 例として図 1 における制約モジュール集合 {`INIT, FALL, BOUNCE`} に図 4 のアルゴリズムを適用した結果を式 (2) に示す.

$$\mathrm{y}^- = 0 \wedge \mathrm{y}'^- \neq 0 \tag{2}$$

各制約モジュールはシミュレーション中に変化しないので, シミュレーション開始前に PFC を求めることが可能となる.

ただしガード条件の成否によって採用される制約が異なるので, 解析時にはすべてのガード条件の成否の組み合わせに対して計算を行う必要がある. このため, 図 4 の 3 行目から始まる for all によって全組み合わせに対する計算を行っている.

Require: 制約モジュール集合 MS
Ensure: 制約モジュール集合 MS に対応する PFC
  $FC := False$
  $A := CollectAsk(MS)$
  for all $A_+ \subseteq A$ **do**
    for all $(g \Rightarrow c) \in A$ **do**
      $G := True$
      if $(g \Rightarrow c) \in A_+$ **then**
        $G := G \wedge g$
      else
        $G := G \wedge \neg g$
      end if
    end for
    $S := CollectTell(MS, A_+)$
    $C := CalculateFalseConditions(S \wedge G)$
    $C := C \wedge G$
    if C **then**
      return $True$
    else
      $FC := FC \vee C$
    end if
  end for
  return FC

図 4: 制約モジュール集合の解析アルゴリズム

A は HydLa プログラム内のガード条件を含む制約であり, A_+ はその中でガード条件が成立すると仮定した制約を表す. 4 行目から始まる for all によって G にガード条件に関する制約を追加している. 11 行目の *CollectTell* で S に A_+ に含まれるすべてのガード条件の後件と MS 内のガード条件を含まない制約の連言を代入している. 制約の連言である S は制約ストアと呼ばれる.

12 行目の *CalculateFalseConditions* でここまでに計算した S と G に対応する PFC を算出し, C に代入する. C が $True$ だった場合には調べている制約モジュール集合 MS は必ず矛盾することが分かるので, その場で $True$ を返す. C が $True$ でなければ, FC に C と G の連言を論理和で加える. ここで G を論理積でつなげることは, PFC にガード条件の成否を前提として加えることを意味する. ここまでの処理をガード条件の成否の全組み合わせについて繰り返す.

今回の実装では求められた PFC に通常変数が含まれていた場合にも対応できるような実装を行った. 実装の詳細については 3.4 節で述べる.

Require: 制約ストア S
Ensure: 与えられた S に対応する PFC
  $V := GetVariables(S)$
  $V := RemovePrevVariables(V)$
  return $\neg\exists V(S)$

図 5: CalculateFalseConditions のアルゴリズム

図 5 に *CalculateFalseConsitions* のアルゴリズムを示す. *CalculateFalseConditions* は与えられた制約ストア S に対応するポイントフェーズでの PFC を返す関数である. 例として図 1 における制約モジュール集合 {`INIT, FALL, BOUNCE`} について `BOUNCE` のガード条件が成立する仮定の下で図 5 を適用した結果を式 (3) に示す.

$$\mathtt{y\text{-}} \neq 0 \vee \mathtt{y'\text{-}} \neq 0 \tag{3}$$

1 行目の *GetVariables* で S 内の変数を V に代入し, 2 行目の *RemovePrevVarialbes* で V から prev つき変数を取り除く. 3 行目で与えられた S に対応する PFC を返している.

3.4 実装

実装は図 4 のアルゴリズムをすべての解候補制約モジュール集合に適用するように行った. 図 4 のアルゴリズムでは PFC を求める前提としてガード条件に通常変数を含まない必要があったが, 実際の HydLa プログラムではガード条件に通常変数が含まれる場合がある. その場合には PFC に通常変数が含まれてしまうため, 通常変数に関する式を False と置き換えることで求められた条件を prev に関する変数のみの式に変換している. 以降, この変換を行って得られた条件を Prev False Sufficient Conditions (PFSC) と呼ぶ. この処理を行って求められた PFSC は式 (4) を満たすが, 式 (5) を満たさない場合がある.

$$C \Rightarrow \neg LC \tag{4}$$

$$C \Leftarrow \neg LC \tag{5}$$

例として図 1 のプログラムで `BOUNCE` のガード条件を `y=0` に置き換えたプログラムを考える. 置き換えた後のプログラムの解軌道は図 1 の解軌道と変わらないものとなる. 置き換えた後のプログラムにおける制約モジュール集合 {`INIT, FALL, BOUNCE`} に図 4 を適用すると式 (6) が得られる.

$$(\mathtt{y\text{-}} \neq 0 \vee \mathtt{y'\text{-}} \neq 0) \wedge \mathtt{y} = 0 \tag{6}$$

式 (6) ではプログラム中に現れない `y-` が表れているが, これは `y'-` が存在することにより, 変数 `y` がその点において連続であることから `y` = `y-` が導き出せるためである. 式 (6) 内の通常変数に関する式を False と置き換えると結果として False を得る. これが置き換えた後のプログラムにおける PFSC となるが, これは式 (5) において LC が充足不可能であれば常に成り立つ. しかし置き換えたプログラムでは LC が充足可能な場合がある (`BOUNCE` のガード条件が成立しない場合). 従って PFSC は式 (5) を満たさない場合があることが分かる.

このことにより制約モジュール集合 MS の無矛盾性判定を PFSC の成否判定に置きかえることができないが, PFSC が成立した場合には MS は矛盾すると判断できる. 今回の実装では PFSC の成否判定を行い, 成立した場合には MS が矛盾すると判断し (このように PFSC により MS が矛盾すると判断できることを prev 矛盾検出と呼ぶ), 成立しなかった場合には従来の無矛盾性判定を行う. また, PFSC はポイントフェーズについてのものをシミュレーション開始前に算出し, ポイントフェーズを対象として PFSC を利用する.

4. PFSC を用いた無矛盾性判定の評価実験

評価実験では表 1 に示している例題において非決定実行で 20 フェーズ分のシミュレーションを行った. その測定を行ったときの実験環境を表 2 に示す.

表 2: 実験環境

OS	Debian(etch)
CPU	Quad-Core AMD Opteron(tm) 2.3 GHz
Memory	16 Gbyte

4.1 実験結果

PFSC を利用して制約モジュール集合の矛盾判定を行うようにした場合の実行時間削減率を図 6 に示す.

図 6 を見ると, 実行時間削減率は bouncing_particles で最も高く約 70% であり, box_mod で最も低く約 −10% だった. 全体を見ると 11 例題中 9 例題において実行時間削減効果があったことが分かった.

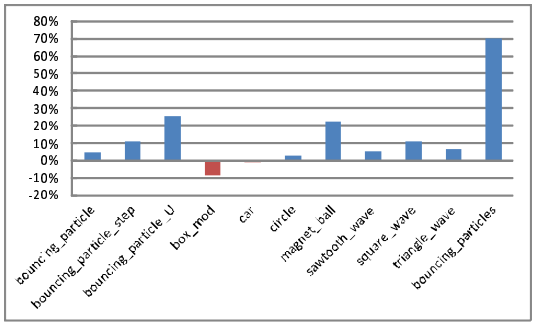

図 6: PFSC を利用した場合の実行時間削減率

4.2 PFSC を用いた無矛盾性判定についての考察

今回の実装では, PFSC を用いて prev 矛盾検出が行えなかった場合は従来の無矛盾性判定を行うので, PFSC を用いた判定のうち, 制約モジュール集合が矛盾すると判断できた割合 (以下この割合を PFSC の適合率と呼ぶ) が高いほど実行時間を減少させることができると考えられる. そこで各例題について, PFSC の適合率を図 7 に示す.

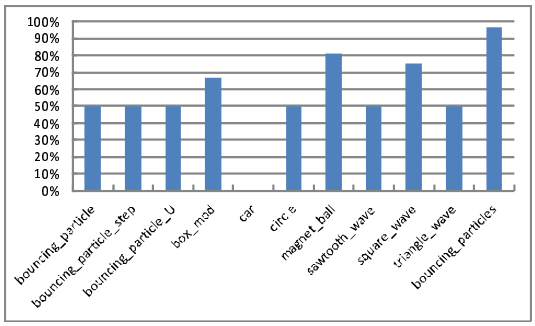

図 7: PFSC の適合率

図 6 と図 7 を比べると, PFSC の適合率が高くても実行時間削減率が高くなっていない例題も存在していることが分かる. 例えば bouncing_particles や magnet_ball などは PFSC の適合率が高いほど実行時間を減少できるという予想に従った結果が出ているが, bouncing_particle_U や box_mod についてはこの予想に従わない結果が出ていることが分かる.

PFSC を用いた無矛盾性判定を行うことによって短縮できる実行時間は, 1 フェーズ中で i 番目に無矛盾性判定を行う制約モジュール集合に対する従来の無矛盾性判定法でかかる時間を b_i, PFSC の判定時間を a_i, PFSC によって制約モジュール集合が矛盾すると判定できた場合に 1, それ以外の場合に 0 となる関数を t_i とすると式 (7) で表される.

$$\sum_i (b_i \times t_i - a_i) \tag{7}$$

従って式 (7) の値が正の値となれば実行時間が短縮されたことになる. 式 (7) を大きくするためには PFSC の適合率に加え, a_i が小さくなる必要がある. 仮に i 番目の制約モジュール集合に対して $(b_i < a_i)$ が成り立つと, その制約モジュール集合の無矛盾性判定の実行時間は必ず遅くなることになる. ここで実際に box_mod のシミュレーションでの処理の割合を PFSC を利用する場合と利用しない場合を比較すると図 8 が得られる.

図 8 を見ると PFSC の判定時間 $(\sum_i a_i)$ が大きいため短縮された分の無矛盾性判定の時間を超えてしまい, 結果としてシミュレーション時間が増えていることが分かる.

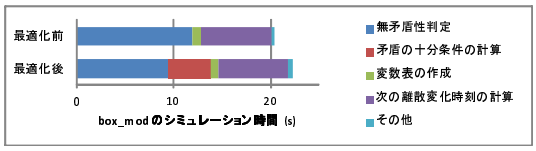

図 8: PFSC を利用する場合と利用しない場合の box_mod のシミュレーション時間

これらの結果から PFSC を利用した無矛盾性判定では以下の条件を満たす HydLa プログラムに対して実行時間削減率が大きくなることが分かった.

- PFSC の適合率が高い
- PFSC の判定にかかる時間が短い

これら 2 つの条件のうち, どちらか 1 つもしくは 2 つの条件を満たさなければ実行時間の削減率が高くならないことが分かった.

5. まとめと今後の課題

PFSC を利用して無矛盾性判定を行うようにし, 例題において測定を行った結果 11 例題中 9 例題実行時間を削減することができた. その測定結果から PFSC を利用して無矛盾性判定を行う場合, PFSC の適合率が高く, PFSC の判定時間が短いほど実行時間が大きく削減できることが分かった.

今後の課題として以下の課題が挙げられる.

- ガード条件に通常変数が含まれていても PFC が求められるようにアルゴリズムを改良する.
- PFSC または PFC を利用した無矛盾性判定を IP における無矛盾性判定に適用する.

一つ目の課題を達成し, PFC を求めることで, 制約モジュール集合の無矛盾性判定を完全に PFC の判定に置き換えることができる. その場合, 短縮できる実行時間は $\sum_i (b_i - a_i)$ となる. この式における a_i は PFC の判定時間を表す. PFC を利用した場合に短縮できる実行時間は $\sum_i \{b_i \times (1 - t_i)\}$ で表わすことができる. ただし, PFC の判定時間と PFSC の判定時間は同じだけかかると仮定し, その値を共通で a_i としている.

$b_i > 0$ であり, t_i は 1 か 0 の値しかとらないので, 短縮できる実行時間は常に 0 以上となる. 従って PFC を求めることができれば, 実行時間をさらに削減できる.

次に二つ目の課題を達成することにより, IP における無矛盾性判定時間を短縮することが期待できる. 本研究の一部は, 科学研究費 (基盤研究 (B) 23300011) の補助を得て行った.

参考文献

[1] A.Borning, B.Freeman-Benson, M.Wilson, “Constraint hierarchies”, Lisp and Symbolic Computation, Vol.5, pp.221–268, 1992.

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

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

[4] 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.

[5] 渋谷俊, 高田賢士朗, 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.

[6] 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.

[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, 第 5 回システム検証の科学技術シンポジウム (SSV2008), 2008, pp.1–6.