4C1-2

フェーズ間の制約差分情報および制約-変数間の依存関係を用いたHydLa処理系の最適化

Optimization of HydLa Implementation Using Difference Information of Constraints between Phases and Relation between Variables and Constraints

小林輝哉[*1] Teruya KOBAYASHI　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情報理工学専攻
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

Hybrid systems are dynamical systems with both continuous and discrete changes of states. HydLa is a hybrid systems modeling language based on constraints. Hyrose, an implementation of HydLa, aims at the verification of hybrid systems. Hyrose provides some features which are useful for the verification of hybrid systems. However, Hyrose may take very long time to simulate a large HydLa program. One of the causes of this is the redundancy in constraint solving. To identify variables which should be recalculated, we propose a technique to analyze the difference between the previous and current phases and the relation between variables and constraints. By using the above information, we reduced the number of consistency checks and the judgments of guard conditions that are performed after the first two phases of simulation. The time complexity of consistency checks was reduced to $O(1)$ from $O(N)$, while the time complexity of the judgments of guard conditions was reduced to $O(N^2)$ from $O(N^3)$.

1. はじめに

ハイブリッドシステム [2] は，複数の連続系が離散事象の発生により切り替わるシステムであり，制御工学や生命工学など幅広い分野での応用が可能である．そのような分野では，複雑なシステムの挙動を理解するための解析や，システムの望ましくない挙動を検出するための検証の需要がある．そのため，複雑なシステムのモデリングを可能にする記述力を持つハイブリッドシステムモデリング言語と，解析および検証を高速かつ高度に行えるシミュレータは非常に有用である．

ハイブリッドシステムモデリング言語 HydLa[5] は，対象とするシステムの性質や構造を時相論理式と微分方程式で表される制約を組み合わせて宣言的に記述することで，KeYmaera[4] に代表される手続き型のモデリング手法やハイブリッドオートマトン [1] に比べて，システムの直観的な記述を可能としている．HydLa の処理系 Hyrose[3] は数式処理による誤差のないシミュレーションと，記号実行による不定値を含む HydLa プログラムのシミュレーション，さらにモデルの挙動の定性的な場合分けを可能としている．Hyrose はシミュレーションの高度化を主目的に開発が行われてきたが，高速化に関しての研究はほとんど行われておらず，大規模なモデルのシミュレーションには非常に多くの時間がかかる．

本研究ではこの問題を解決するため，Hyrose の実行を最適化することを目的とする．従来の Hyrose では，モデル中の複数のオブジェクトの一部だけが離散変化に関わるような例題に対して冗長な制約求解処理を行うために，実行時間が増大するという問題があった．本研究ではフェーズ間の制約の差分情報と，制約-変数間の依存関係を解析し利用することで，そうした冗長性を排除するための最適化手法を提案および実装した．

連絡先: 小林輝哉, 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, teruya(at)ueda.info.waseda.ac.jp

2. ハイブリッドシステムモデリング言語 HydLa

HydLa は制約階層に基づく宣言型のハイブリッドシステムモデリング言語である．HydLa では，時相論理式および微分方程式を用いて表される制約と，制約間の優先関係を表す階層構造である制約階層を宣言することでモデリングを行う．HydLa におけるモデルの挙動は，制約階層を満たす制約集合のうち，無矛盾かつ極大な集合を充足する各変数の挙動である．図 1 は本論文でベンチマークとして扱うビリヤードのモデルである．図 1 ではボールが直線状に等間隔に並んでいる．図 1 を HydLa でモデリングしたものを図 2 に示す．図 2 のプログラムは各ボールの質量は等しく，各ボールの衝突は完全弾性衝突としてモデリングされている．

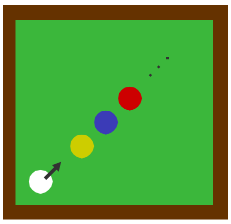

図 1: ビリヤードモデル

3. HydLa の処理系 Hyrose

Hyrose は HydLa の処理系であり，入力として HydLa プログラムを受け取り，受け取ったプログラム内の各変数の軌道（解軌道）を出力する．Hyrose はハイブリッドシステムを検証する目的で開発されており，数式処理による誤差のないシミュレーションと，記号実行による不定値を含む HydLa プログラムのシミュレーション，さらにモデルの挙動の定性的な場合分けを可能としている．このように Hyrose はハイブリッドシステムを検証する上で有用な機能を備えている．しかし Hyrose

```
INIT(x,y,x0,y0,xv,yv) <=> x=x0&y=y0&x'=xv&y'=yv.
CONS(x) <=> [](x''=0).
COL(x1,y1,x2,y2) <=> [](
 (x1--x2-)^2+(y1--y2-)^2=4 =>
  x1'=1/4 *
   ((x2--x1-)*(x2'-*(x2--x1-)+y2'- *(y2--y1-)) -
   (y2--y1-)*(-x1'-*(y2--y1-)+ y1'- *(x2--x1-))) &
  y1'=1/4 *
   ((y2--y1-)*(x2'-*(x2--x1-)+y2'-*(y2--y1-)) +
   (x2--x1-)*(-x1'-*(y2--y1-)+y1'-*(x2--x1-))) &
  x2'=1/4 *
   ((x2--x1-)*(x1'-*(x2--x1-)+y1'-*(y2--y1-)) -
   (y2--y1-)*(-x2'-*(y2--y1-)+y2'-*(x2--x1-))) &
  y2'=1/4 *
   ((y2--y1-)*(x1'-*(x2--x1-)+y1'-*(y2--y1-)) +
   (x2--x1-)*(-x2'-*(y2--y1-)+y2'- *(x2--x1-))) ).
COL_WALL(z) <=>
  [](z- + 1 = 40 | z- - 1 = -40 => z' = -z'-).

INIT(x0,y0,0,0,20,20),
INIT(x1,y1,5,5,0,0),INIT(x2,y2,10,10,0,0),
(CONS(x0),CONS(y0))<<(COL_WALL(x0),COL_WALL(y0),
COL(x0,y0,x1,y1),COL(x0,y0,x2,y2)),
(CONS(x1),CONS(y1))<<(COL_WALL(x1),COL_WALL(y1),
COL(x1,y1,x2,y2) ),
(CONS(x2),CONS(y2))<<(COL_WALL(x2),COL_WALL(y2)).
```

図 2: 3 個のボールのビリヤードをモデリングした HydLa プログラム

には大規模なモデルのシミュレーション非常に多くの時間を要するという問題がある．

3.1 Hyrose の実行アルゴリズム

本節では，Hyrose の実行アルゴリズムについて述べる．詳細なアルゴリズムは文献 [3] によって与えられているため，本節では本研究において特に重要な部分についてのみ述べる．本実行アルゴリズムではモデルの離散変化を計算する Point Phase (PP) と，連続変化を計算する Interval Phase (IP) を繰り返すことで，軌道を計算していく．各 Phase においては成り立つべき制約の連言である制約ストアを求める必要があり，このために制約の無矛盾性判定とガード条件の導出判定を繰り返し，その閉包を計算している．無矛盾性判定とは，制約ストアが充足可能か否かの判定である．ガード条件の導出判定とは，与えられた制約から各条件つき制約のガード条件が導出可能か否かの判定である．ここで導出可能だと判断された条件つき制約の後件は制約ストアに追加され，無矛盾性判定の対象となる．IP においては上記の計算に加え，離散変化時刻の計算も行なっている．離散変化時刻の計算とは，各ガード条件について条件の成否が変化する時刻を求め，その中で最小のものを選択するという計算である．離散変化時刻の計算においては副次的に離散変化条件となるガード条件も求めており，この条件を利用することが提案手法において重要である．

3.2 既存アルゴリズムの冗長性

既存の実行アルゴリズムは，毎 PP，IP ごとに与えられた HydLa プログラムのすべての制約について無矛盾性判定とガード条件の導出判定を行っている．N ボールビリヤードの例では，$2N$ 個の条件なし制約が追加された制約ストアに対して無矛盾性判定が行われ，${}_NC_2 + 2N$ 個の条件つき制約に対してガード条件判定が行われる．このようなモデルの規模の増大による制約ストアの肥大化とガード条件判定回数の増加が実行時間の増大の原因の一つとなっている．しかし，N ボールビリヤード等の多数のオブジェクトが登場するモデルにおいては，一回の離散変化によって軌道が変化する変数はプログラム内の一部の変数のみである場合が多く，プログラム内のすべての変数を毎フェーズ計算するのは冗長な処理であると言える．また他の冗長性として，PP における左極限値のみに言及したガード条件（Prev ガード条件）の導出判定が挙げられる．PP における Prev ガード条件の導出可能性は，直前の IP の離散変化時刻の計算において判明しているため，導出判定を行う必要がない．なぜなら，離散変化の原因となった Prev ガード条件は必ず導出状態が変化し，離散変化の原因とならなかった Prev ガード条件は導出状態が変化することがないためである．この二つの冗長性を排除するための最適化手法を次節で説明する．

4. 提案手法

図 3 に再計算が必要な制約のみを計算する閉包計算のアルゴリズムを示す．既存のアルゴリズムでは，各フェーズですべての変数について計算を行っていたが，本アルゴリズムでは 2 回目以降のステップ（1 つの PP から IP までの計算）において，直前の IP からの差分，すなわち再計算が必要な変数とそれらを含む制約のみを計算対象とすることで計算量を削減する．再計算が必要な変数とは，あるステップにおいて離散変化し軌道が変化する変数のことである．PP では直前の IP の計算結果を，IP では直前の PP の計算結果を利用することで，そのフェーズでどの変数が離散変化するかを閉包計算を行う前に調べ，不要な再計算を排除する．

本手法のアルゴリズムの概要を説明する．本手法では，既存アルゴリズムの `CalculateClosure` 関数に大きく分けて 3 つの処理を追加した．まず直前のフェーズの計算結果から離散変化する変数を求める処理である．PP においては，導出可能性が自明な Prev ガード条件の導出判定の省略も行い，その結果に基づいて離散変化する変数を求める．2 つ目は離散変化する変数を含まない制約に関する計算の省略である．これにより直前の IP からの差分のみが計算対象となり，不要な再計算が排除される．そして 3 つ目が新たに追加された離散変化する変数の整合性の確認と再計算である．最初の処理では離散変化すると判定されなかった変数が，ガード条件判定によって新たに離散変化することが判明することがある．この際，離散変化しないと仮定して行っていたそれまでの計算の整合性のために，閉包計算をやり直す場合がある．以降，各処理に関して詳細を説明する．

4.1 Prev ガード条件判定の省略と離散変化する変数の算出

まず軌道が変化することが閉包計算の開始時点でわかる変数を求める (図 3：2〜10 行目)．現在のフェーズが PP の場合，3.2 節で述べたように，導出可能性がすでに判明している Prev ガード条件の導出判定を離散変化条件を用いて省略する（図 3：3 行目）．この際，離散変化条件となったガード条件は必ず導出状態が変化するため，フェーズ間の制約の差分が生じる．この差分を利用して，離散変化する変数を探す (図 3:5 行目)．ここで求めた離散変化する変数の集合 CV 内の変数を含まない制約は，この後で行う Prev ガード条件以外の導出によって新たに CV に追加されない限り，無矛盾性判定およびガード条件導出判定の対象から外され，計算が省略される．また，現在のフェーズが IP の場合は，直前の PP の CV を用いる（図

Require: 現在のフェーズタイプ $Phase$，直前のフェーズの値に関する制約ストア S_{prev}，直前のフェーズで成立した条件付き制約の集合 A_{+prev}，直前のフェーズで成立しなかった条件付き制約の集合 A_{-prev}，直前のフェーズで離散変化した変数の集合 CV_{prev}，離散変化の原因の集合 CD，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 $CheckConsistency(S)$
Ensure: 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，離散変化する変数の集合

```
1: A+ := E
2: if Phase = PP then
3:    (A+, A−) := CollectPrevGuard(M, CD, A+, A−)
4:    S := CollectTell(M, A+, S_prev)
5:    CV := FindChangingVariables(S, S_prev)
6: else
7:    CV := CV_prev
8: end if
9: repeat
10:   S := CollectChangingTell(M, A+, S, S_prev, CV)
11:   (TF, CP) := CheckConsistency(S, CP)
12:   if TF = False then
13:      return (False, φ, CP, φ, φ, φ)
14:   end if
15:   Expanded := False
16:   BranchedAsks := CollectChangingAsk(M, A+, A−, CV)
17:   S := AddPreviousValue(S, S_prev, CV)
18:   for all (g ⇒ c) ∈ BranchedAsks do
19:      (TF, CP) := CheckConsistency(S ∧ g, CP)
20:      if TF ≠ False then
21:         (TF, CP) := CheckConsistency(S ∧ ¬g, CP)
22:         if TF ≠ False then
23:            continue
24:         end if
25:         Expanded := True
26:         if IsAlways(c) then
27:            E := E ∪ (g ⇒ c)
28:         end if
29:         A+ := A+ ∪ (g ⇒ c)
30:         if ((g ⇒ c) ∈ A−prev) then
31:            CV := ExpandChangingVariables(c, S, CV)
32:         end if
33:      else
34:         A− := A− ∪ (g ⇒ c)
35:         if ((g ⇒ c) ∈ A+prev) then
36:            CV_tmp := CV
37:            CV := ExpandChangingVariables(c, S, CV)
38:            if CV ⊃ CV_tmp then
39:               S := EliminatePreviousValue(S, S_prev, CV)
40:               continue
41:            end if
42:         end if
43:      end if
44:   end for
45: until ¬Expanded
46: return (S, E, CP, A−, A+, CV)
```

図 3: 再計算が必要な制約のみを計算する $CalculateClosure$ のアルゴリズム

Require: 現在の制約ストア S,直前のフェーズの値に関する制約ストア S_{prev}
Ensure: 離散変化する変数の集合

```
1: M := CollectModules(S)
2: M_prev := CollectModules(S_prev)
3: CV := CollectVariables(M △ M_prev)
4: I := M ∩ M_prev
5: repeat
6:    Expanded := False
7:    CM := CollectChangingModule(I, CV)
8:    CV_tmp := CV ∪ CollectVariables(CM)
9:    if CV_tmp ⊃ CV then
10:      Expanded := True
11:      CV := CV_tmp
12:   end if
13: until ¬Expanded
14: return CV
```

図 4: $FindChangingVariables$ のアルゴリズム

3：7 行目）.

図 4 に離散変化する変数を求めるアルゴリズムを示す．まず現在の制約ストアに含まれる制約の集合 S と直前のフェーズの制約ストア S_{prev} に含まれる制約の集合の対称差を取り，それに含まれる変数の集合を離散変化する変数の集合 CV とする (図 4：1 行目)．ここである変数が求めた CV に含まれなかったとしても，S と S_{prev} 双方に含まれる制約の中で CV の変数と関係を持っていた場合，その変数も連動して離散変化することになる．そのため，S と S_{prev} 双方に含まれる制約の集合から，CV の変数を持つ制約を探し，その制約に含まれる変数を CV に追加する処理を不動点に達するまで行う (図 4：5 行目〜13 行目)．こうして最終的に求まった CV が離散変化する変数の集合として返される (図 4：14 行目)．

4.2 離散変化する変数を含まない制約に関する計算の省略

CV の変数を含む制約および CV の変数の現在のフェーズでの計算結果のみを制約ストアに追加する（図 3：10 行目）ことで，不要な無矛盾性判定を省略する．また，再計算が必要な変数を含むガード条件のみを導出判定の対象とする (図 3：16 行目) ことで，判定回数を削減する．その際 CV に含まれない変数を含むガード条件を判定できるようにするために，直前の IP における CV に含まれない変数の解を制約ストアに追加する (図 3：17 行目)．

4.3 新たに追加された離散変化する変数の整合のための再計算

ガード条件導出判定の結果，前のフェーズから導出状態が変化し，その制約に新たな変数が含まれていた場合，新たに離散変化する変数が増える (図 3：31 行目，37 行目)．直前のフェーズで導出された制約が現在のフェーズで導出されなくなった場合，それまで再利用していた直前の IP の軌道を制約ストアから除き，再計算する (図 3：38 行目〜41 行目)．これは，相対的に制約が多い状態で求めた変数の軌道を仮定して再利用していたことになるので，誤った計算結果となる可能性があるためである．

5. 評価実験

図 2 に示した例題に対し，ボールの個数を 3 個から 23 個まで増やしていき，従来の手法と提案手法それぞれで 3 ステップ

シミュレーションを行った．実験は表 1 に示した実機上の VM で行った．

表 1: 実験環境

	VM	実機
OS	Debian 7.2	CentOS 6.4
CPU	QEMU Virtual CPU version 1.0 2.8GHz × 2	Quad-Core AMD Opteron(tm) 2.3GHz, Quad-core × 2
Memory	4Gbyte	16Gbyte

5.1 実験結果

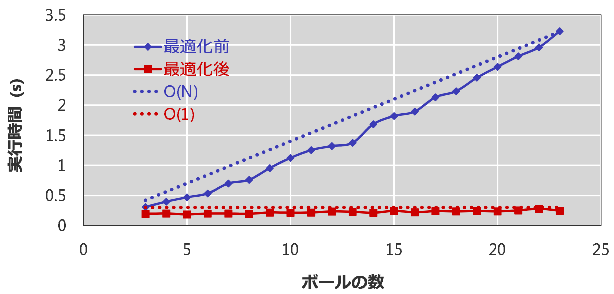

図 5: 無矛盾性判定の実行時間

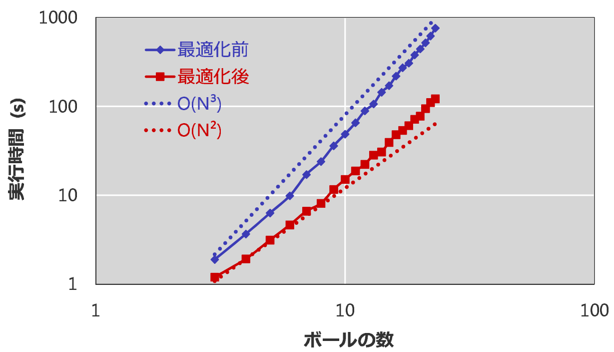

図 6: ガード条件判定の実行時間

提案手法では，直前の IP からの差分を利用した最適化を行っているため，3 ステップ中の後半の 2 ステップが最適化される．図 5 に最適化された部分の無矛盾性判定の実行時間のグラフを，図 6 に最適化された部分のガード条件判定の実行時間のグラフを示す．図 5 はボールの数に対する実行時間のオーダーが $O(N)$ から $O(1)$ 近くまで削減されたことを示した．また，図 6 ではボールの数に対する実行時間のオーダーが $O(N^3)$ から $O(N^2)$ 近くまで削減される結果となった．

5.2 考察

5.2.1 無矛盾性判定の削減

従来の実行アルゴリズムでは，PP，IP 共に $2N$ 個の制約が追加された制約ストアに対して無矛盾性判定が行われていたが，本手法により直前のフェーズで衝突した 2 球の x 軸と y 軸方向に対する運動に関する制約のみに計算対象が絞られるため，追加される制約数が 4 個に削減される．このことは，$O(N)$ から $O(1)$ 近くまで実行時間のオーダーが削減された結果と対応している．完全に $O(1)$ とならなかった理由としては，各制約に対して変化する変数を含むか判定する処理が全体の計算時間にわずかに影響を及ぼしたものと考えられる．

5.2.2 ガード条件導出判定の削減

従来の実行アルゴリズムでは，${}_NC_2 + 2N$ 個のガード条件が毎フェーズごとに導出判定されていた．提案手法により PP では本例題の条件付き制約はすべて Prev ガード条件で構成されているため，ガード条件判定回数が 0 になった．IP では直前の PP で衝突した 2 球のうちどちらか 1 つとそれ以外の球との衝突に関するガード条件（$2N - 3$ 個）と，衝突した 2 球それぞれの壁との衝突に関するガード条件 (4 個) のみが判定対象となったので，$2N + 1$ 個のガード条件が判定されるようになった．

また，ガード条件判定は制約ストアとガード条件の論理積を計算することで行われるため，その計算量は制約ストアに追加されている制約の数にも依存すると考えられる．制約ストアはガード条件判定を行う直前でプログラムに記述された制約のかわりに前の IP で求めた離散変化しない変数の解を制約として追加するので，制約ストアの制約数は N 個になっている．この N 個の制約数の制約ストアに対して，1 ステップで最適化前は $2({}_NC_2 + 2N)$ 回の導出判定を行い，最適化後は $2N + 1$ 回の導出判定を行うことになる．したがって，最適化前後で計算時間のオーダーは $O(N^3)$ から $O(N^2)$ になると推測でき，このことは実験結果と対応している．完全に $O(N^2)$ とならなかった理由としては，各ガード条件に対して変化する変数を含むか判定する処理が全体の計算時間にわずかに影響を及ぼしたものと考えられる．

6. まとめと今後の課題

モデルの規模の増大に伴う無矛盾性判定とガード条件導出判定の実行時間の増大という問題に対し，フェーズ間の制約差分情報および制約-変数間の依存関係を用いて再計算が必要な変数のみ計算を行う手法を構築し，評価実験により無矛盾性判定の実行時間が N から 1 近くまで，ガード条件導出判定の実行時間が $O(N^3)$ から $O(N^2)$ 近くまで削減されることを確認した．今後の課題としては，本手法では最適化されない最初のステップにおける処理性能の向上が挙げられる．本手法が対象するモデルは，一般に各制約間の依存性が低いため，計算対象となる制約集合を分割出来る．本手法で用いた制約-変数間の依存関係を調べる手法を適用し，制約集合を分割した上で計算を並列化することで，最初のステップにおける処理性能の向上が期待できる．

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

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

[3] 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.

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

[5] 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.