2011 年度 修士論文

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

提出日： 2012 年 1 月 27 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： 5110B058-2
渋谷　俊

ハイブリッドシステムとは時間の経過に伴って状態が連続変化したり，状態や方程式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ハイブリッドシステムの柔軟な表現力を活かして物理学や制御工学，生命工学などの分野でシステムの目標状態への到達可能性検証や仕様を満たすようなパラメータ調整など重要な役割を果たしている．HydLa は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であり，HydLa 処理系 Hyrose は精度保証されたシミュレーションを行ってシステム検証に役立てることを目標としている．これまでの Hyrose はシミュレーション終了までの実行時間や状態変化のステップ数を指定して実行し，その時点までの結果を得ていた．このため HydLa プログラムを無限時間まで実行した時の性質がどのようになるか明らかでなかった．

今回，HydLa プログラムをハイブリッドオートマトンに変換する手法を示す．ハイブリッドオートマトンは連続変化を表すノードと離散変化を表すエッジによりハイブリッドシステムを表し，ハイブリッドオートマトンを用いたシミュレーションツールや検証器の研究が進められている．HydLa プログラムを変換したハイブリッドオートマトンの状態と遷移には HydLa プログラムのすべての実行パターンを含むため，ハイブリッドオートマトンを用いた検証器の利用が期待できる．検証器にハイブリッドオートマトンを直接記述するのではなく HydLa プログラムから変換するため，ユーザーがモデリングする際に数式を数学と論理学の記法で記述できる，優先順位のある制約で完結に記述できる，といった HydLa の特徴を活かすことができる．生成するハイブリッドオートマトンは冗長な状態や遷移をできるだけ少ないものとし，変換には有限回数の記号実行を行い，プログラムの実行時間によらないものを目指した．

提案手法である変換方法を Hyrose にハイブリッドオートマトンを出力する機能として実装した．変換により生成されたハイブリッドオートマトンを用いて，入力元の HydLa プログラムと対応していること，漏れ無く冗長のないハイブリッドオートマトンになっていることや HydLa プログラムの性質を確認した．ハイブリッドオートマトンを扱う検証ツールの利用やプログラムの初期値に不定な値を持つ場合について考察する．

目次

図目次

表目次

第 1 章

はじめに

1.1 研究の目的と背景

(まだ概要のコピー＋参考文献なので加筆削除必要) ハイブリッドシステムのモデリングツールはハイブリッドシステムの柔軟な表現力を活かして物理学や制御工学，生命工学などの分野でシステムの目標状態への到達可能性検証や仕様を満たすようなパラメータ調整など重要な役割を果たしている．[5] HydLa は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であり，HydLa 処理系 Hyrose は精度保証されたシミュレーションを行ってシステム検証に役立てることを目標としている．[9][11][13] これまでの Hyrose はシミュレーション終了までの実行時間や状態変化のステップ数を指定して実行し，その時点までの結果を得ていた．このため HydLa プログラムを無限時間まで実行した時の性質がどのようになるか明らかでなかった．

今回，HydLa プログラムをハイブリッドオートマトンに変換する手法を示す．ハイブリッドオートマトンは連続変化を表すノードと離散変化を表すエッジによりハイブリッドシステムを表し，ハイブリッドオートマトンを用いたシミュレーションツールや検証器の研究が進められている．[4][1] HydLa プログラムを変換したハイブリッドオートマトンの状態と遷移には HydLa プログラムのすべての実行パターンを含むため，ハイブリッドオートマトンを用いた検証器の利用が期待できる．検証器にハイブリッドオートマトンを直接記述するのではなく HydLa プログラムから変換するため，ユーザーがモデリングする際に数式を数学と論理学の記法で記述できる，優先順位のある制約で完結に記述できる，といった HydLa の特徴を活かすことができる．生成するハイブリッドオートマトンは冗長な状態や遷移をできるだけ少ないものとし，変換には有限回数の記号実行を行い，プログラムの実行時間によらないものを目指した．

提案手法である変換方法を Hyrose にハイブリッドオートマトンを出力する機能として実装した．変換により生成されたハイブリッドオートマトンを用いて，入力元の HydLa プログラムと対応していること，漏れ無く冗長のないハイブリッドオートマトンになっていることや HydLa プログラムの性質を確認した．ハイブリッドオートマトンを扱う検証ツールの利用やプログラムの初期値に不定な値を持つ場合について考察する．

1.2 論文構成

本論文の構成は下記の通りである．

第 2 章

ハイブリッドシステムモデリング言語 HydLa の概要について解説する．

第 4 章

ハイブリッドオートマトンの仕様やハイブリッドオートマトンを用いたツールについて述べる．

第**??**章

関連研究について紹介し，HydLa との比較する．

第 5 章

提案するハイブリッドオートマトン変換アルゴリズムについて述べる．

第 6 章

提案手法の実装について述べる．

第 7 章

いくつかの例題を用いて変換を試し，生成したハイブリッドオートマトンについて考察する．

第 8 章

変換手法について考察と今後の課題について述べる．

第 2 章

ハイブリッドシステムモデリング言語 HydLa

2.1 ハイブリッドシステム

ハイブリッドシステム [5] とは時間の経過に伴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システムを指す．

ハイブリッドシステムは物理学や制御工学，生命工学などの分野で，システムの目標状態への到達可能性検証や，仕様を満たすようなパラメータ調整に重要な役割を果たす．

2.2 HydLa

HydLa[9] は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であり，次のような特徴を持つ．

- 制約を宣言することでプログラムを記述する．
- 状態や条件を論理式と数式を組み合わせて表現する．
- 制約階層を用いることで，適用する制約間の優先順位を表現する．

HydLa で記述したプログラムのシミュレーションを行う HydLa 処理系 Hyrose が現在早稲田大学上田研究室で開発中である．

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 [4] を用いてハイブリッドシステムを表現するものが多い [1]．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[2][3] が数少ない例外である．簡潔な記述ができるが計算の

(program)	$P ::= (MS,DS)$
(module set)	$MS ::= M$ の集合を要素とする半順序集合
(definitions)	$DS ::=$ 互いに異なる左辺を持つ D の集合
(definition)	$M \Leftrightarrow C$
(constraint)	$C ::= A \mid C \wedge C \mid \mathrm{G} \Rightarrow \mathrm{C} \mid \Box C \mid \exists x.C$
(guard)	$G ::= A \mid G \wedge G$
(atomic constraint)	$A ::= ErelopE$
(expression)	$E ::=$ 通常の式 $\mid E' \mid E-$

表 2.1 HydLa 言語の構文

```
INIT   <=> ht=10 /\ v=0.
FALL   <=> [](ht'=v /\ v'=-10).
BOUNCE <=> [](ht- =0 => v=-(4/5)*v-).
INIT, (FALL << BOUNCE).
```

図 2.1 床で弾むボールの挙動

精度が保証されていないため厳密なシミュレーションを行うことができない.

2.3 HydLa 言語の構文

HydLa 言語の構文を表 2.1 に示す．HydLa プログラム P は階層のある制約からなる制約モジュール集合 MS と制約定義 DS から構成される.

2.4 HydLa によるモデリング例

床で弾むボールの挙動について HydLa を使ってモデリングしたプログラムを図 2.1 に示す.

このプログラムでは 1 行目から 3 行目で制約を定義し，4 行目で使用する制約を呼び出している．`<=>` の左辺は制約定義名を，右辺は論理記号や方程式によってその内容を表す．変数 `ht` はボールの高さ，`v` はボールの落下速度を表す.

制約 `INIT` は `ht` と `v` の時刻 0 での値を記述している．HydLa ではすべての変数は時刻の関数であり，時相演算子 `[]`(always) が付いていない制約は時刻 0 での性質を表し，付いている制約は時刻 0 以降の性質を表す.

図 2.2 床で弾むボールの実行

制約 FALL は落下運動を示す．右辺全体に [] が付いており，括弧内が時刻 0 以降の ht’ と v’ の性質を表す．ht’ と v’ はそれぞれ ht と v の時刻に対する一次微分である．

制約 BOUNCE は跳ね返りを表す．この制約は => があるため条件付き制約である．条件付き制約は => の左辺のガード条件が満たされた場合に，右辺の式が成立することを表す．変数 ht や v の後ろに付いている - はガード条件の判定時刻における当該変数の左極限値を表す．

4 行目で制約の優先順位に従って半順序集合を構成しており，半順序集合の各要素を制約モジュールと呼ぶ．<< で関係付けられた制約は，右辺が左辺より優先されて適用される．「,」で区切られた制約モジュール間には階層関係はない．この書き方により各時刻で FALL より BOUNCE が優先して適用される．

このプログラムの実行結果を図 2.2 に示す．落下開始から ht=0 となるまでが連続変化，ht=0 で離散変化となる．その後再び跳ね返るまで連続変化，跳ね返りで離散変化となる．以降この連続変化と離散変化を繰り返す．

2.5 Hyrose

HydLa 処理系 Hyrose は HydLa プログラムを宣言的意味論に従って正しく実行することを目的としたものであり，開発改良が進められている [7][8][13]．数式処理を主として用いて，数式処理では解けない計算について区間計算を組み合わせることで，精度保証されたシミュレーションを行うことを目標としている．精度保証されたシミュレーションはシステム検証においても重要な要素技術となる．HydLa プログラムの初期値に不定な値を含む場合，非決定実行シミュレーションができ，すべての解起動を求める全解探索実行ができる．

2.6 HIDE

Hyrose の GUI 環境として HIDE[10] がある．プログラムの実行結果として解起動だけでなく，シミュレーションに用いた制約や全解探索実行した場合の各軌道や分岐点の情報も確認することができる．(HIDE の機能にオートマトン変換初期版が乗っていることを書くか，考察で GUI 機能との連携を書く)

第 3 章

HydLa の非決定シミュレーション実行アルゴリズム

3.1 HydLa の非決定シミュレーション実行アルゴリズム

HydLa 処理系は HydLa プログラムを宣言的意味論に従って正しく実行することを目的としたものであり，開発改良が進められている [7][8]．数式処理を主として用いて，数式処理では解けない計算について区間計算を組み合わせることで，精度保証されたシミュレーションを行うことを目標としている．精度保証されたシミュレーションはシステム検証においても重要な要素技術となる．

論文 [8] のアルゴリズムは数式処理によるシミュレーションに目的を限定し，解軌道が一意に決まらないモデル，たとえばシステムの初期値が範囲で与えられたモデルやパラメタを用いて表現したモデルは対象としていなかった．本論文のアルゴリズムは，ガード条件以外の場所に現れる制約が一意な解を持たないようなモデルも扱えるよう拡張したものとなっている．アルゴリズム中で使用している微分方程式の求解や制約の充足可能性判定などの手続きが計算可能である限り，具体的な数値を用いずにシミュレーションができ，検証などへの展開が期待できる．

3.1.1 ポイントフェーズとインターバルフェーズ

HydLa の非決定実行アルゴリズムを図 3.1 に示す．まず *SolveCH* によりプログラムから制約階層の処理を行い [6]，解候補となる制約モジュール集合のリスト *MS* を求める．*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．

```
Input: HydLa プログラム HP, シミュレーション終了時刻 MaxT
  MS := TopologicalSort(SolveCH(HP))
  T := 0;  S := true
  while T <_S MaxT do
    for M ∈ MS do
      (SS, MS_tmp) := PP(S, M, MS, T)
      while |SS| > 1 do
        S := GetElement(SS)
        (SS, MS_tmp) := PP(S, M, MS, T)
      end while
      if |SS| = 1 then
        S := GetElement(SS);  MS := MS_tmp
        goto PPEnd
      end if
    end for
    break // 全ての制約モジュール集合が矛盾
    PPEnd:
    for M ∈ MS do
      (SS, T_tmp, MS_tmp) := IP(S, M, MS, T, MaxT)
      while |SS| > 1 do
        S := GetElement(SS)
        (SS, T_tmp, MS_tmp) :=
          IP(S, M, MS, T, MaxT)
      end while
      if |SS| = 1 then
        S := GetElement(SS);  MS := MS_tmp
        T := T_tmp
        goto IPEnd
      end if
    end for
    break // 全ての制約モジュール集合が矛盾
    IPEnd:
  end while
```

図 3.1 HydLa の非決定シミュレーション実行アルゴリズム

本論文のアルゴリズムはシミュレーション時刻が終了するまで，離散変化を扱うポイントフェーズ (PP) 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) 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，本論文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．

それぞれのフェーズでは無矛盾かつ極大な制約モジュール集合に基づいて実行を行う [**?**]．PP と IP のアルゴリズムを図 3.2，3.3 に示す．アルゴリズム内に記述しないが，各フェーズでの計算結果を出力として表示するものとする．以降ではシミュレーション開始時刻からの経過時間を T，ある IP における直前の PP の時刻からの経過時間を t を用いて表す．

Input: 制約ストア S，制約モジュール集合 M，解候補モジュール集合のリスト MS， 現在のシミュレーション時刻 T
Output: 制約ストアの集合 SS，新しい解候補モジュール集合のリスト MS
if $T > 0$ **then**
$M := EliminateNotAlways(M)$
end if
$(SS, \_, \_, MS) := CalculateClosure(S, M, MS,$
$CheckConsistencyPP)$
return (SS, MS)

図 3.2 PP のアルゴリズム

Input: 制約ストア S，現在の制約モジュール集合 M，解候補モジュール集合のリスト MS，現在のシミュレーション時刻 T，最大シミュレーション時刻 $MaxT$
Output: 制約ストアの集合 SS，IP 終了時刻 $EndT$，新しい解候補モジュール集合のリスト MS
$M := EliminateNotAlways(M)$
$M_{all} := MaxModule(MS)$
$(SS, A_-, A_+, MS) := CalculateClosure(S, M,$
$MS, CheckConsistencyIP)$
if $|SS| \neq 1$ **then**
return $(SS, MaxT, MS)$
end if
$S_t := SolveDifferentialEquation(GetElement(SS))$
$(MinT, SS) := CompareMinTime($
$\{FindMinTime(S_t \wedge g) | (g \Rightarrow c) \in A_-\}$
$\cup \{FindMinTime(S_t \wedge \neg g) | (g \Rightarrow c) \in A_+\}$
$\cup \{FindMinTime(S_t \wedge M_-) | M_- \in (M_{all} \setminus M)\}$
$\cup \{FindMinTime(S_t \wedge \neg M_+) | M_+ \in M\}$
$\cup \{(MaxT - T, true)\})$
if $|SS| > 1$ **then**
return $(SS, MaxT, MS)$
end if
$S := SubstituteMinTime(S_t, MinT)$
return $(\{S\}, MinT + T, MS)$

図 3.3 IP のアルゴリズム

S はそのフェーズで成立すべき制約の連言で制約ストアと呼ぶ．制約ストアには前フェーズ終了時の変数の値に関する制約が入る．すなわち PP では左極限値に関する制約，IP ではフェーズ開始時点で成立する制約が含まれる．S にはさらに各フェーズにおいて成立する制約が入る．

シミュレーション時刻 T は具体値であるとは限らず，パラメタを含むことがあり，パラメタの情報は S に含まれている．シミュレーション終了判定の $T <_S MaxT$ は，T がパラメタを含む場合，そのパラメタについて S を参照して比較を行うことを表す．

制約モジュール集合 M には，always が付いていない開始時刻 $(T = 0)$ でのみ成立する制約が含まれる．この制約は $T = 0$ より後のフェーズでは不要であるので

Input: 制約ストア S，現在の制約モジュール集合 M，解候補モジュール集合のリスト MS，無矛盾性判定関数 $CheckConsistency(S)$
Output: 制約ストアの集合 SS，成立しない条件付き制約の集合 A_-，成立する条件付き制約の集合 A_+，新しい解候補モジュール集合のリスト MS

```
A_- := ∅;  A_+ := ∅
repeat
  S := CollectTell(M, S)
  if ¬CheckConsistency(S) then
    return (∅, ∅, ∅, MS)
  end if
  A_- := A_- ∪ CollectAsk(M)
  Expanded := false
  for (g ⇒ c) ∈ A_- do
    (S_true, S_false) := ((S ∧ g), (S ∧ ¬g))
    if CheckConsistency(S_true)
     ∧ CheckConsistency(S_false) then
      return ({S_true, S_false}, A_-, A_+, MS)
    end if
    if CheckConsistency(S_true) then
      M := DeleteGuard(M, (g ⇒ c))
      if CheckAlways(c) then
        MS :=
          {DeleteGuard(M_i, (g ⇒ c)) | M_i ∈ MS}
      end if
      A_- := A_- \ {g ⇒ c};  A_+ := A_+ ∪ {g ⇒ c}
      Expanded := true
    end if
  end for
until ¬Expanded
return ({S}, A_-, A_+, MS)
```

図 3.4　CalculateClosure のアルゴリズム

EliminateNotAlways で取り除く．M_{all} は *HP* から制約階層を除いたもので *MS* の先頭要素である．

PP や IP を実行した結果，制約ストアの集合 *SS* が得られ，*SS* の要素数により次の処理が異なる．

SS が空集合の場合，採用した制約モジュール集合の制約間に矛盾があったことを意味する．現在のフェーズで実行中の制約モジュール集合 M は失敗とし，M については何もせず，*MS* のリストの中で次に優先度の高い制約モジュール集合を用いてフェーズの実行をやり直す．採用できる制約モジュール集合が存在しない場合，シミュレーション実行を中止する．

SS の要素が複数の場合，ガード条件以外の制約が不等式やパラメタをもつなどの理由で解軌道が一意に決まらず，初期値やパラメタの範囲によって解が分岐したことを意味する．*SS* の要素である制約ストアには各分岐が満たすべき制約がそれぞれ入っている．

CheckConsistencyPP(S){
  return $\exists(S)$
}

CheckConsistencyIP(S){
  $S_t :=$ *SolveDifferentialEquation*(S)
  return($Inf\{t \mid \exists_{\setminus t}(S_t \wedge (t > 0))\} = 0$)
}

図 3.5 CheckConsistency

GetElement により任意の制約ストアを非決定的に 1 つ選び，このフェーズをやり直すことでその分岐を実行する．

SS の要素が 1 つの場合，その制約ストアを次のフェーズに引き継ぎ，シミュレーション実行を進める．

3.1.2 CalculateClosure

PP や IP の *CalculateClosure* (図 3.4) では，ガード条件が成立する場合に後件の制約を追加する閉包計算を繰り返し，そのフェーズ開始時点でガード条件が成立しない条件付き制約の集合 A_- と成立する条件付き制約の集合 A_+ を求める．このとき，*CollectAsk* は条件付き制約がどの制約モジュールに含まれていたものか区別する．これにより同じ形の条件付き制約が A_- の要素に含まれたとしても，後で条件付き制約を取り出す際に互いを区別できるようになる．

S が各変数の値を一意に決定しない場合，ガード条件が成立する場合と成立しない場合が同時に存在することがあり，このとき解が分岐する．解を正しく求めるために *CalculateClosure* はガード条件が成立する場合と成立しない場合に絞りこんだ制約ストアをそれぞれ返す．

成立するガード条件は，*DeleteGuard* で条件付き制約の中から取り除き，後件だけを残す．成立するガード条件の後件に always が含まれるとき，*MS* の要素である各モジュール集合からも，制約モジュールに含まれている該当するガード条件部分を取り除く．

3.1.3 CheckConsistency

CalculateClosure では制約ストア S 内の無矛盾性判定を PP 用の *CheckConsistencyPP*，IP 用の *CheckConsistencyIP* (図 3.5) により判定する．$\exists(S)$ は S の存在閉包を

表す．*CheckConsistencyIP* では制約ストア S の中の微分方程式を *SolveDifferentialEquation* で解いて，各変数の値を t に関する式で表した S_t を求める．さらに IP 開始直後に S_t が成立するかどうか判定している．*Inf* は実数値集合の最大下界 (greatest lower bound) を求める．$\exists_{\backslash t}$ は変数 t を除いた各変数に関する存在閉包を表す．判定の結果，最大下界が 0 の場合，$t = 0$ の正の近傍で S_t は満たされる．

3.1.4 ガード条件の判定

条件付き制約のガード条件の判定は *CalculateClosure* の for ループの最初で行っている．S_{true} と S_{false} はガード条件 g が成立する場合と $\neg g$ が成立する場合に絞りこんだ制約ストアを表す．両制約ストアの充足可能性を *CheckConsistency* で判定することで，S の下で g および $\neg g$ が成立しうるか否か判定する．両方とも成立しうる場合は解が分岐する．

3.1.5 FindMinTime と CompareMinTime

IP では次の離散変化時刻を *FindMinTime* と *CompareMinTime* で求める．SS の要素である制約ストア S から *SolveDifferentialEquation* によって t に関する式 S_t を求める．S_t，A_-，A_+，IP 開始時点で採用していない制約モジュール，IP 開始時点で採用している制約モジュールから，次の離散変化時刻を，直前の PP の時刻を $t = 0$ として *FindMinTime* で求める．その際，引数として渡す論理式が満たされない場合は離散変化時刻として ∞ を返す．

制約に不等式やパラメタが含まれていると，次の離散変化が起こる要因が一通りに定まらず，解が分岐する場合がある．解が分岐するとき，各分岐は，それぞれ異なる理由による離散変化の結果を表している．*FindMinTime* では離散変化時刻だけでなく，必ずその離散変化が起こるように強化した制約も返す．*CompareMinTime* では *FindMinTime* の結果から次の離散変化時刻を場合分けし，各分岐で満たすべき S をそれぞれ用意し，S の集合である SS を返す．

3.2 例題

本論文で提案したアルゴリズムの実行が HydLa の宣言的意味論 [**?**] に従うことを例題によって示す．5 節と同様に，シミュレーション開始時刻からの経過時間を `T`，ある IP において直前の PP の時刻からの経過時間を `t` を用いて表す．

```
A <=> x=0.
B <=> [](y=1).
C <=> [](x'=1 /\ (x>3 => y=2)).
A, (B << C).
```

図 3.6 ある時刻の直後からガード条件が成立する例

```
INIT  <=> x=0 /\ y=0.
CON   <=> [](y'=0).
CROSS <=> [](x=5 /\ y=2).
INC   <=> [](x'=1).
INIT, (CON << (CROSS << INC)).
```

図 3.7 極大が変化することによる離散変化の例

```
INIT   <=> 9<ht /\ ht<11 /\ v=10.
FALL   <=> [](ht'=v /\ v'=-10).
ROOF   <=> [](ht- =15 => v=-(4/5)*v-).
BOUNCE <=> [](ht- =0  => v=-(4/5)*v-).
INIT, (FALL << (ROOF, BOUNCE)).
```

図 3.8 不等式を含むことで分岐する例

図 3.6 について考える．制約 C のガード条件が成立し，y=2 となるのは T=3 の直後からである．今回のアルゴリズムでは T=3 の PP で x=3 となり，次の IP でガード条件 x>3 が成立するか判定する．もしこのとき PP の判定のように前回のフェーズ終了時の値 x=3 とガード条件 x>3 を連立させると *false* となる．そこで IP 開始時点で制約ストアから求めた x と t の関係式 x=t+3 とガード条件 x>3 を連立したものが t=0 の直後で成り立つことを検査している．

図 3.7 について考える．HydLa の宣言的意味論では制約モジュール集合の中から無矛盾かつ極大なものを時々刻々と満たすことを求めている [?]．初めの IP で CROSS と INC が矛盾するため，制約階層の優先順位に従って CON と INC が採用される [6]．このことから x'=1 により x は増加し，やがて T=5 で x=5 となると，CON のかわりに CROSS が採用可能になり，CROSS を採用した場合，y=2 となる．つまり極大な制約モジュール集合が変化することから，離散変化が起こる．本論文のアルゴリズムでは *FindMinTime* により現在採用されていない制約が採用可能になる時刻を求めている．

初期値が幅を持つと，実行結果が分岐する場合があり，ここでは図 3.8 をもとに考える．

```
INIT(q,q0,v,v0) <=> q=q0 /\ v=v0.
CONT(q,v) <=> [](q'=v /\ v'=0).
COLLISION(q1,v1,q2,v2)
    <=> [](q1- =q2- => v1=v2- /\ v2=v1-).
EPS <=> eps>0.
EPS,INIT(qa,-4,va,5),
INIT(qb,1,vb,0),INIT(qc,1+eps,vc,0),
(CONT(qa,va),CONT(qb,vb),CONT(qc,vc))
<<(COLLISION(qa,va,qb,vb) /\
   COLLISION(qb,vb,qc,vc)).
```

図 3.9 隣接したボールの衝突の例

図 2.1 と同じく ht が高さ，v が落下速度であるが，v=10 により鉛直方向上向きに投げ上げ，ROOF により床だけでなく ht=15 にある天井で衝突した時も跳ね返るモデルとなっている．この INIT は不等式を含むために，天井に衝突する場合と衝突せずに落下する場合の両方が考えられる．*FindMinTime* は ROOF のガード条件より，9<ht<11 を 10<=ht<11 (天井に衝突する) と 9<ht<10 (天井に衝突しない) に分ける．*CompareMinTime* では ROOF や BOUNCE のガード条件が原因となって起こる離散変化時刻とシミュレーション終了時刻を比較し，各分岐ごとにそれぞれ次の離散変化時刻を求める．

図 3.9 について考える．複数の隣接した球がきわめて短い時間の間に連続して衝突を繰り返すモデル (ニュートンのゆりかご) の一部である．球は質量 1，反発係数 1 で，大きさを持たない質点とする．隣接して配置するために内側の球は動かずに最も外側の球だけが飛び出すように見える．物体の衝突後の速度は運動量保存の法則と反発係数の定義を連立したものから求めている．球 A,B,C の座標を qa,qb,qc，速度を va,vb,vc とし，球 A を静止した球 B,C に衝突させる．INIT は位置と速度の初期値，CONT は速度と加速度，COLLISION は衝突した場合の速度変化を表す．このモデルをシミュレーションするには静止した球を隣接して配置することが必要であるが，隣接してしまうと球 B と C の座標が重なる．すると T=1 の PP で球 A と B，B と C の COLLISION の条件付き制約がどちらも成り立ち，vb に矛盾する制約が課される．よって少しだけ間隔を空けることにする．

この間隔を具体的な数値ではなく，パラメタ eps として与える．これにより実行時間や途中式に eps を含んだままシミュレーションを行うことができる．T=1 で球 A と B が衝突し，次に T=1+0.2eps で球 B と C が衝突する．T>1+0.2eps の IP で座標の t に関する式は qa=1,qb=1+eps,qc=5t となり球 A,B は動かないが，球 C は動き続けることがわかる．eps を 0 に近づける極限を考えると，球 A,B が隣接し動かず，端の球 C だけが飛び出すことになる．

第 4 章

ハイブリッドオートマトン

4.1 ハイブリッドオートマトン

ハイブリッドオートマトンは状態を表すノードと遷移を表すエッジから成り立つ.
ハイブリッドオートマトンの構成要素は次のものから成る.

- 変数
  システムの変数.
- ガード条件
  状態がガード条件を満たす時，遷移する
- リセット
  遷移の際に行われる変数の代入を表す.
- 不変式 (invariant)
  変数が状態中で満たすべき条件．この不変式が満たされないときに遷移する
- フロー
  状態中で変数に適用される微分方程式.
- 初期値
  変数の初期値

ハイブリッドオートマトンは時間経過ごとに対応するノード間を遷移することで，ハイブリッドシステムの連続的な時間変化と離散的な状態の切り替わりを表現する.

4.2 ハイブリッドオートマトンを用いたツール

ハイブリッドオートマトンを用いたツールには Matlab/Simulink，hyvisual，Keymaera，iSAT などがありハイブリッドシステムのシミュレーションや検証に用いられる．これらは GUI を利用し加算素子や積分素子などを組み合わせ構成した電気回路により微分方程式を表すものや，オートマトンのノードにラベルを付け，遷移するための条件を満たしたらラベルを切り替えて実行するものである．

計算処理部分に MATLAB，mathematica，SAT を用いている．

4.3 ハイブリッドオートマトンの例

(参考文献から何かのハイブリッドオートマトンを graphviz で出力する．)

ハイブリッドオートマトンの例として〜のシステムをハイブリッドオートマトンで記述した例を図〜に示す．

出力には graphviz を用いた．

〜の遷移を繰り返すことがわかる．

第 5 章

ハイブリッドオートマトン生成アルゴリズム

5.1 入力とする HydLa プログラムの仕様

HydLa 記法は柔軟な記述力ゆえ，HydLa プログラムの制約には様々な書き方がある．例えば

- 制約モジュールに時相演算子 always なしの制約とありの制約
- ガード条件のない制約と条件付き制約などをつなげた制約
- 条件付き制約の後件にさらに条件付き制約がある制約

のような記述ができる．

提案する変換手法で入力として扱う HydLa プログラムは

- １つの制約モジュールには時相演算子なしの制約とありの制約が混在しない
- １つの制約モジュールにはガード条件のない制約と条件付き制約が混在しない
- 条件付き制約の後件にさらに条件付き制約が存在しない

ものとする．入力として扱わない HydLa プログラムの例を図 5.1 に示す．入力として扱わない HydLa プログラムでも制約の内容を分割して複数の制約として表すことで，提案する変換手法で扱えるようになると考えている．

これは制約モジュール名に条件付き制約の成立，不成立をラベルとして表記するときのわかりやすさのためであり，評価する制約の中身，そしてシミュレーションに用いる方程式群にまでには影響しないため，本来この制限は不要であるが，説明を簡易にするためこ

```
// 時相演算子 always なしの制約と always ありの制約が混在
SAMPLE1 <=> a=1 /\ [](b=2).

// ガード条件のない制約と条件付き制約が混在
SAMPLE2 <=> [](a=1) /\ [](b=2 => c=3).

// 条件付き制約の後件にさらに条件付き制約が存在
SAMPLE3 <=> [](a=1 => (b=2 => c=3)).
```

図 5.1 入力として扱わない HydLa プログラムの例

のような制限を設ける．またプログラムの初期値に不定な値を含まないものとする．非決定実行との対応については考察で述べる．Hyrose は初期値に不定な値を含む場合の分岐に対応しているため，ハイブリッドオートマトンの遷移経路が分岐するが，まず一意に定まる場合を説明するためである．

5.2 HydLa の実行に則したハイブリッドオートマトン変換アルゴリズムの概要

HydLa の実行に基づいて HydLa プログラムを変換し，ハイブリッドオートマトンを生成する．HydLa の実行とは HydLa プログラムを入力とし，時刻 0 の PointPhase から始まり，PointPhase と IntervalPhase の各フェーズにおいて，極大で無矛盾な解候補モジュール集合を非決定に選択してシミュレーションを行い，指定したシミュレーション終了条件まで各フェーズのシミュレーションを交互に繰り返し，解軌道を求めることである．

このフェーズの切り替わりのときに各変数のフェーズ終了時の値を変数表に格納し引き継ぐ．次フェーズでは変数表の各変数の値を前フェーズ終了時の値 (prev) として扱い，シミュレーションを行う．もし同じ種類のフェーズ (PointPhase または IntervalPhase) で異なる時間やステップのフェーズでも同じ変数表を用いてフェーズの実行を行うと，同じ解候補モジュール集合が採用され，条件付き制約の成立が判定された結果，同一の方程式を用いて解軌道が計算されるため，出力は一致する．(三角波の例だと何度でも `f-=2` でリセットされる)

HydLa プログラムに現れる変数と条件付き制約のガード条件を対応させ，特徴変数としてマークし，条件付き制約の成立・不成立すべての場合を覆うようにする．(バウンシングボールだと速度がどの値でも床に衝突すれば弾む) (床が壊れるバウンシングボール

の場合，はじめの床の衝突だけでなく，いつか壊れた時の床の衝突もシミュレーションできる)

(もし解候補モジュール集合と条件付き制約の組み合わせを全て網羅しようとすると，解候補モジュール集合の採用順序について無視することとなり，冗長なノードやエッジが大量に発生することを後述する実験で確かめる)

b ハイブリッドオートマトンを生成するにあたって，各ノードとエッジを

- 各フェーズで採用している解候補モジュール集合
- 各フェーズで成立しているガード条件
- 初期値に関する制約であるか

について場合分けを行う．

このように分ける理由は

- 各フェーズで採用している解候補モジュール集合
  HydLa の実行は各フェーズにおいて極大で無矛盾な解候補モジュール集合を採用するためフェーズごとに異なるためである．
- 各フェーズで成立しているガード条件
  HydLa の実行は同じ解候補モジュール集合を採用していても，条件付き制約の成立によって後件の制約を評価するかが決定され，シミュレーションに用いる微分方程式が異なるためである．HydLa では条件付き制約のガード条件が成立する場合，評価することになった後件の制約により，他の条件付き制約が成立するかどうかを繰り返しチェックする閉包計算を行う．このため同じ解候補モジュール集合を採用していても条件付き制約の成立によって実行結果が異なる．
- 初期値に関する制約であるか
  ハイブリッドオートマトンの初期値に関する制約は HydLa プログラムの時刻 0 の PointPhase の実行に対応する．時相演算子 always が付いている制約は時刻 0 以降に使用し，付いていない制約は時刻 0 の時点だけで使用するためである．

これらの点に着目しながら HydLa プログラムのシミュレーションをハイブリッドオートマトンの生成が終了するまで実行し，実行結果としてハイブリッドオートマトンに必要な情報を集め，変換を行う．まず時刻 0 の PointPhase と直後の IntervalPhase を実行し，ハイブリッドオートマトンの初期値と初期ノードを生成する．この初期ノードの変数表を特徴変数について解析し，変数表とガード条件の強さを比べ，次にこの初期ノードから実行可能な PointPhase を条件付き制約成立のすべてのパターンを制約モジュール集合

Input: HydLa プログラム *HP*
Output: ハイブリッドオートマトン *HA*
1: {*MSS*, *guards*} := prepare(*HP*);
2: {*variable_map*, *MS*, *positive_asks*} := simulate(*PP*,null,*MSS*);
3: store{*result_store*,PP1, null, *MS*, *positive_asks*};
4: {*variable_map*, *MS*, *positive_asks*} := simulate(IP,*variable_map*,*MSS*);
5: store{*result_store*,IP1, PP1, *MS*, *positive_asks*};
6: {*variable_map*} := analyze(*variable_map*, *guards*);
7: push({*variable_map*}, *stack*);
8: **while** *notEmpty*(*stack*) **do**
9: *variable_map*:= pop(*stack*);
10: {*variable_map*, *MS*, *positive_asks*} := simulate(*PP*,*variable_map*,*MSS*);
11: store{*result_store*,*PPm*, *IPl*, *MS*, *positive_asks*};
12: {*variable_map*, *MS*, *positive_asks*} := simulate(IP,*variable_map*,*MSS*);
13: store{*result_store*,*IPn*, *PPm*, *MS*, *positive_asks*};
14: **if** $isStored(ms, pa, rs)$ **then**
15: {*variable_map*} := analyze(*variable_map*, *guards*);
16: push({*variable_map*}, *stack*);
17: **end if**
18: **end while**
19: *HA* := convert(*result_store*);

図 5.2 HydLa プログラムのハイブリッドオートマトン変換アルゴリズム

の優先順位に従ってすべて実行し，次に各 PointPhase の直後から始まる IntervalPhase を実行しのエッジとノードのペアを生成する．これ以降も同様に生成されたノードから次に実行可能な PointPhase をすべて実行するが，すでに生成されたノードと同じ解候補モジュール集合を採用し，条件付き制約の成立の組み合わせなものは同じ状態であるとし，PointPhase と IntervalPhase の実行を行わない．このように実行する可能性のある PointPhase とその直後の IntervalPhase のペアをオンザフライに実行し，すでに実行した IntervalPhase からは次の PointPhase を実行しないとして，遷移可能なすべての解候補モジュール集合と条件付き制約成立の組み合わせの変化を取得する．

変数表は prev に関して等号で制約を表す．特徴変数に関する制約を判別する．

5.3 変換アルゴリズム

この変換アルゴリズムを図 5.2 に示す．

1. t=0 の PP を実行し (t=0 であるため変数表を用いない)，変数表， 使用した解候補モジュール集合，成立したガード条件を得る
2. 直前のフェーズのない PP の結果として使用した解候補モジュール集合，成立したガード条件を保存する

ハイブリッドオートマトン	HydLa プログラム
変数	変数
ガード条件	条件付き制約のガード条件
リセット	条件なし制約と条件付き制約の後件
不変式	条件付き制約のガード条件の否定
フロー	条件なし制約と条件付き制約の後件
初期値	時刻 0 の PointPhase で用いる制約

表 5.1 ハイブリッドオートマトンの各要素と HydLa プログラムの対応

3. *variable_map* を用いて IP を実行
4. PP1 の直後の IP1 の結果として *MS*, *positive_asks* を保存
5. 変数表からガード条件に含まれる特徴変数を分析して新しい *variable_map* の集合を生成
6. *variable_map* を *stack* に push
7. *variable_map* を用いて PP を実行
8. *variable_map* を *stack* から pop
9. 変数表からガード条件に含まれる特徴変数を分析して新しい *variable_map* の集合を生成
10. *variable_map* を *stack* に push
11. *IPl* 終了時から (PPm) を実行
12. *variable_map* を用いて IP を実行
13. *PPm* 終了直後から *IPn* を実行

この変換手法で初期値とノードとエッジを作成した後，各ノードとエッジにハイブリッドオートマトンの構成要素であるフローなどを付け加えることでハイブリッドオートマトンとなる．

ノードのフローとエッジのリセットにはガード条件を除いた制約が，エッジのガード条件にはガード条件が，ノードの不変式にはガード条件の否定したものが入る．ハイブリッドオートマトンの各要素と HydLa プログラムの対応を表 5.1 に表す．

最も単純な例（初期値制約に関する時刻 0 の PointPhase(PP1) を実行し，以降ある IntervalPhase (IP1) とある PointPhase (PP2)(採用する解候補モジュール集合と成立するガード条件が各フェーズで固定) を繰り返す）を図 5.3 に示す．

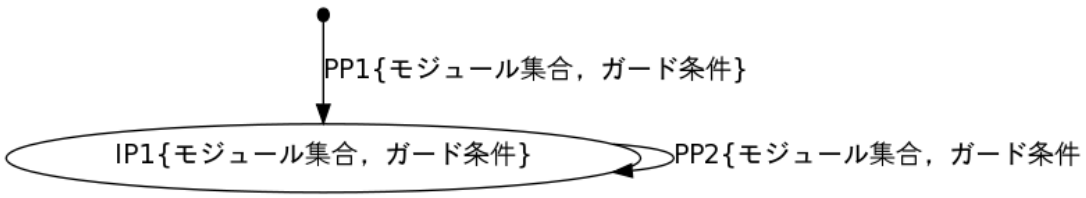

図 5.3　出力するハイブリッドオートマトンの例

第 6 章

実装

6.1 ハイブリッドオートマトンへの変換部分の実装

Hyrose に第 5 章を基にしてハイブリッドオートマトン出力用モードとして実装した. 極大で無矛盾な解候補モジュール集合の選択や, 制約間の無矛盾性判定を繰り返し用いる.

第 7 章

実験

7.1 実験

提案したアルゴリズムを例題を用いて確認する．

7.2 三角波の例題

三角波の HydLa プログラムについてハイブリッドオートマトンへの変換を示す．入力とする三角波の HydLa プログラムを図 7.1 に示す．

変換アルゴリズムに沿って進めると，時刻 0 の PointPhase を行い，採用するモジュール集合は INIT，INC，DROP となり，成立するガード条件はない．PointPhase 終了時に変数表には f=0，f’=1 が入る．次に時刻 0 の直後から始まる IntervalPhase を f-=0，f’-=1 を用いて行い，採用するモジュール集合は INC，DROP となり，成立するガード条件はない．IntervalPhase 終了時に変数表には f=2，f’=1 が入る．

ここからガード条件の成立で取りうるパターンを考えると，特徴変数は f で変数表から f- = 2 であるかを判別する．この場合 f- = 2 であるため，この IntervalPhase からガード条件が成立する PointPhase を実行し，採用するモジュール集合が DROP となる．

```
INIT <=> f=0.
INC  <=> []{f'=1}.
DROP <=> []{f- = 2 => f=0}.
INIT, (INC << DROP).
```

図 7.1　三角波の HydLa プログラム

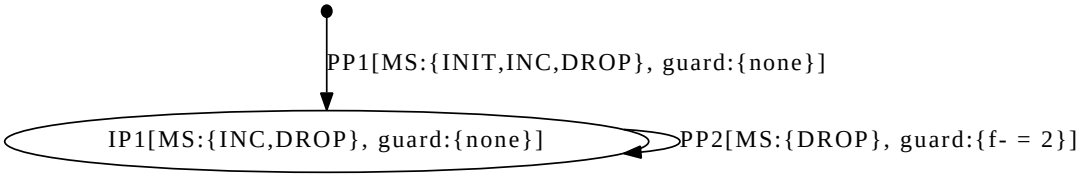

図 7.2 三角波のプログラムの変換結果

PointPhase 終了時に変数表には `f=0, f'=1` が入る．この PointPhase の時刻の直後から始まる IntervalPhase を `f=0, f'=1` を用いて行い，採用するモジュール集合は `INC, DROP` となり，成立するガード条件はない．この採用したモジュール集合と成立するガード条件の組み合わせは既に実行済みであるため、変換はここで終了となる．

変換結果のオートマトンをラベルで示したものを図 7.2 に示す．

`IP1` と `PP2` を繰り返し遷移することがわかる．

7.3 床で跳ね返るボール

7.4 DC-DC コンバーター

7.5 一定量の荷重が加わると壊れる床で跳ね返るボール

第 8 章

考察と今後の課題

8.1 考察

考察

8.2 今後の課題

今後の課題について述べる.

謝辞

上田先生、細部先生、石井さん、HydLa 班の皆様、ありがとうございました。

2012 年 2 月 渋谷 俊

参考文献

[1] 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.

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

[3] 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.

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

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

[6] 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.

[7] 廣瀬賢一 : 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の提案と実装, 早稲田大学大学院基幹理工学研究科, 修士論文, 2010.

[8] 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 処理系の実行アルゴリズムの検討, 第 8 回ディペンダブルシステムワークショップ, 2010.

[9] 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.

[10] Kakeru Sakuraba, Kazunori Ueda, Hiroshi Hosobe, Shun Shibuya, Shota Matsumoto : Simulation with guaranteed accuracy using hybrid system modeling language HydLa, APLAS, poster, 2011.

[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 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.

[12] 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.

[13] 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, 2011.