日本ソフトウェア科学会第 29 回大会 (2012 年度) 講演論文集

# 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アルゴリズム

竹口 輝　和田 亮　松本 翔太　細部 博史　上田 和紀

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システムをハイブリッドシステムと呼ぶ. HydLa はハイブリッドシステムを制約によって記述する言語であり, 精度保証されたシミュレーションを行うことで, システムの検証に役立てることを目標としている.

これまで, HydLa で記述されたハイブリッドシステムの無限時間における性質を検証する手法が明らかでなかった. HydLa プログラムをハイブリッドシステムのモデリング手法の一つであるハイブリッドオートマトンに変換することで, 既存のハイブリッドオートマトンを用いた検証器を利用でき, 無限時間におけるシステムの到達可能性や安全性の検証が可能となる. 本論文では, HydLa プログラムのハイブリッドオートマトンへの変換アルゴリズムを述べる.

## 1　はじめに

ハイブリッドシステム [1] 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システムを指す. 物理学, 制御工学, 生命工学など広範囲に適用可能であり, そのシミュレーション・検証系は重要な役割を果たす.

HydLa [4] は制約概念に基づくハイブリッドシステム制約言語である. 数式処理システムを用いた記号実行によって, システムの検証に役立てることを目標としている.

これまで, 文献 [6] で提案された非決定実行アルゴリズムによって, ハイブリッドシステムの有界モデル検査は行うことができたが, 無限時間におけるモデル検査には対応していなかった. また, HydLa とハイブリッドオートマトンの記述力の違いも明らかでなかった.

本論文では HydLa プログラムのハイブリッドオートマトンへの変換アルゴリズムを述べる. 有限状態のハイブリッドオートマトンに変換することで, 既存のハイブリッドオートマトンを用いた検証器を利用することができる. さらに HydLa とハイブリッドオートマトンの記述力の関係を考察する上での手がかりにもなる.

## 2　関連研究

文献 [9] では, SIMULINK や ZELUS で用いられるハイブリッドデータフロー言語をハイブリッドオートマトンへ変換する手法が提案されている. 文献 [10] では, HydLa と同じく制約を用いてハイブリッドシステムを記述する言語である Hybrid cc をハイブリッドオートマトンへ変換し, ハイブリッドオートマトンを用いた検証器の一つである HyTech を用いて検証する手法が提案されている. この手法は入力を線形制約で記述されるプログラムに限定し, さらに有界モデル検査を対象としている.

本論文では, 制約概念に基づく宣言型言語である HydLa をハイブリッドオートマトンへ変換し, 無限時間におけるモデル検査に役立てることを目指す. ま

---

An algorithm for converting hybrid constraint programs to hybrid automata.

Akira Takeguchi, Ryo Wada, Shota Matsumoto, 早稲田大学大学院基幹理工学研究科情報理工学専攻, Waseda University.

Hiroshi Hosobe, 国立情報学研究所, National Institute of Informatics.

Kazunori Ueda, 早稲田大学理工学術院情報理工学科, Waseda University.

た, 提案する変換アルゴリズムは非線形制約や不定値を含む制約を入力として与えることができる.

## 3 ハイブリッドオートマトン

ハイブリッドオートマトン [2] [8] とは, 離散状態を表すノードと遷移を表すエッジから構成されるオートマトンの各ノードに, 連続変数の時間発展の記述を加えたものであり, ハイブリッドシステムのモデリング手法として広く扱われている. ハイブリッドオートマトンを用いた検証ツールとして HyTech, PHAver などが挙げられる [3].

### 3.1 ハイブリッドオートマトンの構成要素

ハイブリッドオートマトンは次の構成要素からなる.

- システムの軌道 (連続状態とも呼ぶ) を表す実数変数の有限集合 $X = \{x_1, x_2, \ldots, x_n\}$. 以下の説明のため, $X' = \{x'_1, x'_2, \ldots, x'_n\}$ を実数変数の時間微分の集合, $\tilde{X} = \{\tilde{x_1}, \tilde{x_2}, \ldots, \tilde{x_n}\}$ を実数変数の離散変化直後の値の集合とする.
- 制御グラフと呼ばれる有限有向多重グラフ $(V, E)$. ノード $V$ を制御モード (離散状態とも呼ぶ), エッジ $E$ を制御スイッチと呼ぶ.
- 初期条件 *init*, 不変条件 (invariants)*inv*, フロー条件 *flow*. これらはそれぞれ各制御モード $v \in V$ に付随する関数で,
  - 各初期条件 *init(v)* は $X$ に属する自由変数に関する述語,
  - 各不変条件 *inv(v)* も $X$ に属する自由変数に関する述語,
  - 各フロー条件 *flow(v)* は $X \cup X'$ に属する自由変数に関する述語,

  である.
- ジャンプ条件 *jump*. 各エッジすなわち制御スイッチ $e \in E$ に付随する関数で, 各ジャンプ条件 *jump(e)* は $X \cup \tilde{X}$ に属する自由変数に関する述語であり, 遷移のための条件と遷移時の変数値の変化を表す.
- イベントの有限集合 $\Sigma$, および各制御スイッチにイベントを対応づける関数 $event{:}E \Rightarrow \Sigma$.

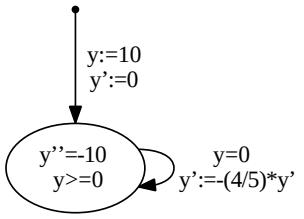


**図 1 床で弾む質点モデルのハイブリッドオートマトン**

ハイブリッドオートマトンの実行は初期制御モードおよび初期変数値から始まり, 時間の経過に伴って条件を満たすノードとエッジに遷移を繰り返すことで, ハイブリッドシステムを表現する.

### 3.2 ハイブリッドオートマトンのモデリング例

例として, 床で弾む質点モデルを取り上げる. このモデルは, 空中にある質点が重力によって自然落下し, 床 (質点の高さが 0) に達したとき反発係数にしたがって跳ね返る, というハイブリッドシステムである.

図 1 に床で弾む質点モデルのハイブリッドオートマトンを示す. 変数 $\mathrm{y}$ は質点の高さを表す. 初期条件 $\mathrm{y} := 10$, $\mathrm{y}' := 0$ として初期ノードに遷移し, フロー条件 $\mathrm{y}'' = -10$ によって床に達するまで $\mathrm{y}$ は連続的に減少する. 床に達するとジャンプ条件 $\mathrm{y} = 0$ が満たされる. 不変式 $\mathrm{y} \geq 0$ が満たされなくなると, エッジに遷移してジャンプ条件 $\mathrm{y}' := -(4/5) * \mathrm{y}'$ によって速度が離散変化する. このようにハイブリッドオートマトンは状態遷移が視覚的に理解しやすいという利点を持つ.

## 4 ハイブリッド制約言語 HydLa

HydLa [4] は制約概念に基づくハイブリッドシステム制約言語であり, 数学と論理学の記法を用いた時相論理式による記述によって, モデルを簡潔に表現することが可能である.

HydLa プログラム中に出現する全ての変数は時間についての関数変数である. それらの変数に関する等式・不等式・微分方程式からなる制約式を記述することで, ハイブリッドシステムの直観的なモデリングを行うことができる. また, 制約間に優先度を与えることで, お互いに矛盾しうる制約の記述が可能である.

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

INIT, FALL<<BOUNCE.
```

**図 2　床で弾む質点モデルの HydLa プログラム**

プログラムに記述された制約を満たす各変数値の時刻に沿った振る舞い (解軌道) を HydLa プログラムの解と呼ぶ. また, 宣言的意味論に従って時刻 0 から指定されたシミュレーション終了時刻までの解軌道を求めることを, HydLa プログラムを実行すると呼ぶ.

### 4.1　HydLa のモデリング例

3.2 節で用いた床で弾む質点モデルの HydLa プログラムを図 2 に示す. 変数 y は質点の高さを表している. 「'」は時間に関する微分を表している. また, 変数の右側に「-」を付加することでその変数の各時刻における左極限値を表すことができる.

このプログラムでは 3 つの制約が宣言されている. INIT で y, y' の初期値を記述し, FALL で質点の加速度を記述している. [] は時相演算子 always であり, 時刻 0 以降全ての時刻で成り立つ制約を表す. BOUNCE は質点の跳ね返りを表し, この系における離散変化条件とそのときの振る舞いが記述されている. BOUNCE のように => を含む制約を条件付き制約と呼ぶ.

4 行目で制約の優先順位に基づいて半順序集合を構成している. 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 << の左辺よりも右辺が優先されて採用されることを表し, 「,」は左右の優先度が等しいことを表す. つまり BOUNCE は各時刻において FALL よりも優先して採用される.

### 4.2　非決定実行アルゴリズム

HydLa は不定値や区間値を持つ系の記述が可能なため, モデルによっては定性的に異なる解軌道を複数持つ場合があり, その解軌道群を正しく求めるアルゴリズムは自明でない. 文献 [6] において HydLa の非決定実行アルゴリズムが提案されているが, ここではアルゴリズムの流れを簡潔に説明する.

まず, 入力された HydLa プログラムから制約階層の処理を行い, 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.

次に入力されたシミュレーション終了時刻まで, 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) を交互に実行する. 各フェーズでは無矛盾かつ極大な制約モジュール集合を採用し, そのフェーズで成り立つべき制約の連言 $S$ を求める. $S$ は制約ストアと呼ばれ, 各フェーズにおいて成立する制約と, 前フェーズ終了時の変数の値に関する制約が含まれる.

各フェーズを実行することによって得られる制約ストアは一通りとは限らず, 解が分岐した場合複数の制約ストアが得られる. その場合, 任意の制約ストアを非決定的に 1 つ選び, 分岐したフェーズを再実行する. 全ての制約ストアについて実行を行うことで, HydLa プログラムの満たす全ての解軌道を求めることができる [5].

IP ではさらに次の離散変化時刻 $T$ を求める. 離散変化が起きない場合は $\infty$ とする. 入力されたシミュレーション終了時刻より $T$ が大きい場合, シミュレーションを終了する.

## 5　変換アルゴリズムの概要

提案する HydLa プログラムのハイブリッドオートマトンへの変換アルゴリズムの概要を述べる. 詳細は 6 節で示す. 本アルゴリズムは HydLa プログラムの実行に沿って行われ, 可能な限り冗長性を回避したハイブリッドオートマトンの生成を目指す.

### 5.1　HydLa とハイブリッドオートマトンの対応

ハイブリッドオートマトンにおけるエッジは, HydLa における PP に該当し, ハイブリッドオートマトンにおけるノードは, HydLa における IP に該当する. さらに, ハイブリッドオートマトンにおける遷移は, ある IP から次の IP までの一連のフェーズの移り変わりを意味する.

各フェーズをハイブリッドオートマトンへ変換するために必要な情報は, そのフェーズで採用した制約モジュール集合と成り立つガード条件の集合である. つ

まり, HydLa プログラムのハイブリッドオートマトンへの変換は, 無限時間において, HydLa プログラムが許す制約モジュール集合と成り立つガード条件の集合の組を全て求めることと同値である.

本アルゴリズムでは各フェーズの情報として, 上記 2 つの要素に加えて制約ストアも保持するが, これは 5.4 節で述べるエッジの通過可能性判定で用いられる.

### 5.2 入力とする HydLa プログラム

HydLa は局所変数生成機能 ∃[4] を用いて制約モジュール集合と成り立つガード条件の集合の組が無限個存在するプログラムを記述できるが, 本アルゴリズムはこのようなプログラムを入力として扱わない. 5.1 節の対応に従って, 変換後のハイブリッドオートマトンが有限状態となる HydLa プログラムのみ入力として扱う.

### 5.3 生成されるハイブリッドオートマトンの不変式

従来のハイブリッドオートマトンでは不変式を満たしている間は, たとえエッジの条件が満たされていたとしてもノードに留まることができるが, 本アルゴリズムで生成されるハイブリッドオートマトンでは不変式を記述しない代わりに, 条件の満たすエッジが存在する場合, 即座にそのエッジに遷移する, と考える.

### 5.4 エッジの通過可能性判定とループ判定

無限時間について言及するためには, 各ノードに含まれるエッジの通過可能性判定が不可欠である. エッジの通過可能性判定とは, そのノードから新たにエッジが生成される可能性はあるか, つまり新たなエッジを通過する可能性があるか, を判定することである. 通過可能性の判定は, 以前に遷移したことがあるエッジに関しては行わない.

エッジの通過可能性判定は, そのノードが含まれるループ内での不変条件を用いることで容易となる. そのため, 本アルゴリズムでは, 非決定実行アルゴリズムに沿ってエッジとノードを求めていく中で, ループの判定を行っている. 本アルゴリズムにおけるループは, 単純閉路に限定せず, ループ内で複数回同じノードを通ることを許す. 一般に, 求められたループの遷移列が長いほど, 不変条件に含まれる情報量が多くなり判定が容易となるため, ループを判定する際は, 可能性のあるループの中で最長のループを求める必要がある.

ループを回るとは, 最長のループとして判定された遷移列の始端から終端までの遷移が連続して生じることを指す. また, ループを回る回数が多いほど, 通過可能性判定を必要とするエッジの数が減少する. なぜなら, 通過可能性を判定する以前に遷移したエッジは, 判定する必要がないからである. そのため, 仮に無限回ループを回ったと仮定すると, エッジの通過可能性を判定する必要はなくなる. 本アルゴリズムでは, ループ回数が入力によって指定された最大回数に達するまでループを回り, その後エッジの通過可能性を判定する.

### 5.5 通過可能性判定の結果の対応

エッジの通過可能性を判定した結果として, 以下の 3 種類が考えられる. また, これは後述する *checkOccurrence*(6.2 節) の戻り値と対応する.

- 必ず通過する
- 通過しない
- 判定不可能

必ず通過すると判定されたエッジがある場合, ループ回数が指定された最大回数に達していたとしても, そのエッジに遷移するまでループを回る. 判定不可能なエッジが存在する場合, そのエッジに遷移した後の状態を調べる必要があるため, そのエッジを通過するような制約ストアを生成して実行する. このとき, 値が定まる変数はその値とし, 値が定まらない変数は不定値とした制約ストアが生成される.

### 5.6 変換アルゴリズムの直観的な説明

ハイブリッドオートマトンへ変換するにあたって, HydLa プログラムの満たす全ての解軌道を調べる必要がある. 本アルゴリズムでは, 全ての解軌道をスタックを用いた深さ優先探索によって調べる.

本アルゴリズムにはシミュレーションステップ, ループ判定ステップ, エッジ判定ステップがある. シミュ

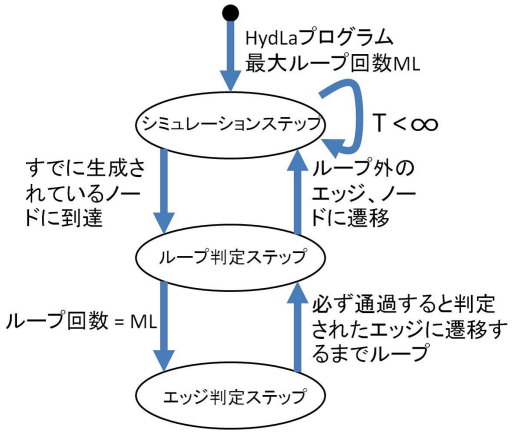


**図 3　変換アルゴリズムの流れを表すオートマトン**

レーションステップでは非決定実行アルゴリズムに沿って PP と IP の計算を行い, 状態遷移列を求める. ループ判定ステップでは最長のループの発見とループを回った回数のカウントを行う. エッジ判定ステップではループ判定ステップで求められたループ内の, 各ノードにおけるエッジの通過可能性を判定する.

図 3 にアルゴリズムの流れをオートマトンで示す. 以下の操作を, スタックに積まれた全ての分岐に対して行う.

まず, 入力として HydLa プログラムと最大ループ回数を受け取り, シミュレーションステップに入る. シミュレーションステップでは, PP と IP の計算を非決定実行アルゴリズムに沿って行い, 状態遷移列を求める. 分岐が発生した場合, 得られた制約ストアを全てスタックに積む. 次の離散変化時刻が $\infty$ の場合, 実行を終了する.

状態遷移列を求める過程で, すでに生成されているノードに到達した場合, ループ判定ステップに移行する. ループ判定ステップでは, 非決定実行アルゴリズムに沿った計算に加え, 遷移がループ内で閉じているか, ループを 1 周したか, を判定する. ループ外に抜けた場合はシミュレーションステップに戻る. ループを最大ループ回数まで回った場合, エッジ判定ステップに移行する.

エッジ判定ステップでは, ループ内に含まれる各ノードにおいて, 必ず通過すると判定されたエッジがある場合, そのエッジに対応する離散変化が生じるまでループ判定ステップを繰り返す. 必ず通過すると判定されたエッジがなく, 判定不可能なエッジが存在する場合, 判定不可能なエッジを通過するような制約ストアを生成し, スタックに積んで実行を終了する.

## 6　変換アルゴリズム

提案する HydLa プログラムのハイブリッドオートマトンへの変換アルゴリズムを図 4 に示す.

入力として HydLa プログラム $HP$ と, 最大ループ回数 $ML$ を受け取り, $HP$ に対応するハイブリッドオートマトンを出力する.

シミュレーションステップとして, HydLa プログラムの実行に沿って PP と IP を交互に行っていく. その中でハイブリッドオートマトンの生成に必要となる情報を $state$ に追加する. ここで $state$ は状態の遷移を表すリストである. また, 図 4 に記述されている PP, IP は文献 [6] で示された PP のアルゴリズムと IP のアルゴリズムを, 成り立つガード条件の集合 $A_+$ を戻り値に含むように拡張した関数である. この拡張は自然な形で行うことができる.

分岐がある場合, $stack$ を用いた深さ優先探索によって全ての分岐について $state$ を求める. $stack$ が空集合のとき, 求められた全ての $state$ からハイブリッドオートマトンを生成しハイブリッドオートマトンへの変換が完了する.

IP を実行した結果得られる, 成り立つガード条件の集合 $A_+$ とその時採用されている制約モジュール集合の組と同一のものが $state$ に含まれていない場合, PP に戻る. 含まれている場合, ループ判定ステップに移行し, $state$ からループの候補となる状態遷移列の集合 $LS$ を求める. $LS$ は現時点までに含まれるそのノードを終端とし, かつ同じノードを始端とする全ての状態遷移列の集合として求められる.

ループ判定ステップに入ると, それ以降の状態遷移が $loop$ に追加され, $checkContinue$(6.1 節) によって $LS$ と $loop$ の比較が行われる. $loop$ を部分列に持つ $LS$ の要素の中で最長の要素が $loop$ と等しい場合, ループしたと判断し, ループ回数をカウントする変数 $count$ をインクリメントする. $LS$ 内に $loop$ を部分

```
Input: HydLa プログラム HP , 最大ループ回数 ML
Output: ハイブリッドオートマトン HA
  (MS, CC) := parse(HP)
  state := [ ]; stack := {}; S := true;
  T := 0; phase :=′ PP′; loop := [ ];
  LS := {}; stateSet := {}
  repeat
    (phase, state, S, T) := stack.pop
    //シミュレーションステップ
    while T < ∞ do
      if phase =′ IP′ then
        goto PPEnd
      end if
      for M ∈ MS do
        (SS, MS_tmp, A_+) := PP(S, M, MS, T)
        while |SS| ≥ 1 do
          for S ∈ SS do
            stack.push(′PP′, state, S, T)
          end for
          (_, state, S, T) := stack.pop
          if |SS| = 1 then
            state.add({′PP′, S, M, A_+})
            loop.add({′PP′, S, M, A_+})
            if loop ∉ LS then
              LS := {}; loop := [ ]
            end if
            MS := MS_tmp
            goto PPEnd
          end if
          (SS, MS_tmp, A_+) := PP(S, M, MS, T)
        end while
      end for
      break
      PPEnd:
      for M ∈ MS do
        (SS, T_tmp, MS_tmp, A_+) := IP(S, M, MS, T, ∞)
        while |SS| ≥ 1 do
          for S ∈ SS do
            stack.push(′IP′, state, S, T)
          end for
          (_, state, S, T) := stack.pop
          if |SS| = 1 then
            state.add({′IP′, S, M, A_+})
            if state.contains({′IP′, _, M, A_+})&
            (subset(loop, ls ∈ LS)  ||  isEmpty(LS))
            then
              //ループ判定ステップ
              if checkContinue(
              state, {′IP′, S, M, A_+}, CC, loop, LS,
              known, unknown, count, ML) = false
              then
                for (tmpstate, guards) ∈ unknown
                do
                  T := (T_tmp, ∞)
                  S :=generateS(guards, tmpstate)
                  stack.push(′PP′, tmpstate, S, T)
                end for
                goto End
              end if
            else
              LS := {}; loop := [ ]
            end if
            goto IPEnd
          end if
          (SS, T_tmp, MS_tmp, A_+) := IP(S, M, MS, T)
        end while
      end for
      break
      IPEnd:
      phase :=′ PP′; MS := MS_tmp; T := T_tmp
    end while
    END:
    stateSet.add(state)
  until notEmpty(stack)
  HA := generateHA(stateSet)
  return HA
```

**図 4　変換アルゴリズム**

```
checkContinue(state, nowIP, CC, loop, LS,
known, unknown, count, ML){
if isEmpty(LS) then
  LS :=possibleAllLoops(state, nowIP)
  loop := [ ]; count := 0;
  known := {}; unknown := {}
end if
loop.add(nowIP)
if loop =MaxElement(LS) then
  count + +
  if count = ML then
    //エッジ判定ステップ
    (known, unknown) :=
    checkNode(loop, state, CC, known, unknown)
    if isEmpty(known) then
      return false
    end if
    count := −∞
  end if
  loop := {nowIP}
end if
return true
}
```

**図 5　*checkContinue***

列に持つ要素がない, つまり *loop* が $LS$ 内に含まれる全てのループ候補と異なる場合, もしくは生成されていないノード, エッジに達した場合, ループ判定ステップから抜ける.

*count* が $ML$ に達すると, エッジ判定ステップに移行する. エッジ判定ステップではループ内に含まれる各ノードで, そのノードに含まれるまだ生成されていない全てのエッジについて, 通過可能性を判定する. 必ず通過すると判定されたエッジ (*known*) がある場合, そのエッジに達するまでループを回る. 判定不可能なエッジ (*unknown*) がある場合, *state* とエッジの状態から *generateS* によって制約ストアを生成し, *stack* に追加して実行を終了する.

*generateS* によって生成される制約ストアは遷移前のノードで成り立つ制約の左極限値と遷移するエッジで成り立つガード条件の制約が含まれる. 言及されていない変数に関しては全て不定値とする.

### 6.1 *checkContinue*

*checkContinue* のアルゴリズムを図 5 に示す. *checkContinue* はループ判定ステップにおいて, 最長のループの発見とループを回った回数のカウントを行う. ループ判定ステップに移行した直後の場合, *possibleAllLoops* によって *state* からループの候補と

```
checkNode(loop, state, CC, known, unknown){
if |loop| > 1 then
  (pp, ip) := lastStep(loop)
  checkEdge(ip, loop, state, CC, known, unknown)
  checkNode(loop.remove(pp, ip), state,
  CC, known, unknown)
end if
}
```

**図 6 *checkNode***

```
checkEdge(ip, loop, state, CC, known, unknown){
reachableGuards := searchEdge(ip, state, CC)
for guards ∈ reachableGuards do
  if checkOccurrence(guards, ip, state) = true then
    known.add(append(state, loop), guards)
  else if checkOccurrence(guards, ip, state)
   = unknown then
    unknown.add(append(state, loop), guards)
  end if
end for
}
```

**図 7 *checkEdge***

なる状態遷移列の集合 $LS$ を求める．最長のループを発見するために, *loop* を部分列に持つ $LS$ の要素の中で最長の要素が *loop* と等しいかどうか調べる．等しい場合, ループ回数をカウントする変数 *count* をインクリメントする．等しくない場合, true を返し実行を続ける．*count* が $ML$ と等しい場合, エッジ判定ステップに移行し, *checkNode*(6.2 節) によって, ループに含まれる各ノードにおいてエッジの通過可能性を判定する．

エッジの通過可能性を判定した結果, 必ず通過すると判定されたエッジ (*known*) が存在する場合, *count* を $-\infty$ として, そのエッジが成り立つまでループを回る．必ず通過すると判定されたエッジ (*known*) が存在しない場合, $false$ を返し実行を終了する．

### 6.2 *checkNode* と *checkEdge*

*checkNode* のアルゴリズムを図 6 に, *checkEdge* のアルゴリズムを図 7 に示す． *checkNode* と *checkEdge* によってエッジの通過可能性判定を行う．

*checkNode* ではループ内に含まれる全てのノードに関して再帰的に *checkEdge* を適用する．*loop* は 1 ループの状態遷移列を含むリストであるため, *checkEdge* を実行したのち *loop* の最後の要素から *pp* と *ip* を取り除きつつ再帰的に実行することで, *loop* の要素が 1 つとなった時, 全てのノードに対して *checkEdge* が行われたことになり処理を終了する．

*checkEdge* ではまだ生成されていないエッジで成り立つガード条件の集合 *reachableGuards* を *searchEdge* によって集め, *reachableGuards* の各要素に対して *checkOccurrence* によって通過可能性を判定する．*checkOccurrence* の戻り値は 5.5 節で示した．また, 7 節の例題の中で *checkOccurrence* の動作例を示す．

### 6.3 アルゴリズムの性能

本アルゴリズムによって生成されるハイブリッドオートマトンの冗長性は *checkOccurrence* の性能に依存する．ここで, *checkOccurrence* は判定不可能となるエッジが多い場合性能が低いといい, 判定不可能となるエッジが少ない場合性能が高いという．

*checkOccurrence* の性能が高い場合, エッジの通過可能性判定において *unknown* となるエッジが少なく, ハイブリッドオートマトンの冗長性を回避できる．*checkOccurrence* の性能が低い場合, エッジの通過可能性判定において多くのエッジが *unknown* と判定され, その全てに関してエッジを生成する必要があるため, 結果として生成されるハイブリッドオートマトンは冗長なものとなる．

$ML$ が 1 で *checkOccurrence* の結果が全て *unknown* となるとき, 制約から明らかに矛盾するエッジ, ノード以外の全ての要素を含むハイブリッドオートマトンが生成される．また, *checkOccurrence* の性能が高く判定不可能なエッジが存在しない場合, $ML$ の値によらず無駄のないハイブリッドオートマトンを生成することができる．

### 6.4 アルゴリズムの停止性

本アルゴリズムはループを含まない HydLa プログラムを入力として与えた場合停止しない．例えば, 無理数を 1 桁ごとに求め, 各数字に対応する状態へ遷移する HydLa プログラムを考える．このプログラムは有限状態しか持たないが, 無理数は循環しないため, 状態遷移列にループは存在しない．部分的なループは存在するがそれよりも大きい $ML$ を入力として与え

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

INIT, FALL<<(ROOF & BOUNCE).
```

**図 8　天井と床で弾む質点モデルの HydLa プログラム**

た場合, 停止しない.

しかし, 実問題をモデリングした HydLa プログラムを考えた場合, ループを含まない状態遷移を繰り返す可能性は低く, 実用上問題はないと考えられる. また, あるステップ数までループを発見できなかった場合, エッジの到達可能性を判定する, というように実装の上で対応することも可能である.

## 7　例題

提案するアルゴリズムが正しく動作することを例題によって示す. また, 例題における *checkOccurrence* の動作を示す.

### 7.1　天井と床で弾む質点モデル

3.2 節で用いた例題を発展させ, 天井と床で弾む質点モデルを考える. 初期速度を 10, 初期高さを開区間 $(9, 11)$ としたプログラムを図 8 に示す.

このプログラムは実行の過程で以下の 3 つのケースに場合分けされる. 以下, 条件付き制約に関して, ガード条件が成立している場合は制約名の添え字に $+$ を記述し, 成立していない場合は $-$ を記述することで区別する.

1. $9 < y < 10$ 天井に衝突せず, 床で弾む
   一度床で跳ねた後の IP で, ループ判定ステップに入る. もう一度跳ねた後の IP で *count* がインクリメントされ, $ML$ に達するまで繰り返し床で弾む. $ML$ に達したとき, 通過可能性を判定するエッジとして,
   (a) $\{\mathtt{FALL}, \mathtt{ROOF}_+, \mathtt{BOUNCE}_+\}$
   (b) $\{\mathtt{FALL}, \mathtt{ROOF}_-, \mathtt{BOUNCE}_+\}$
   (c) $\{\mathtt{FALL}, \mathtt{ROOF}_+, \mathtt{BOUNCE}_-\}$
   (d) $\{\mathtt{ROOF}_+, \mathtt{BOUNCE}_+\}$
   (e) $\{\mathtt{ROOF}_+, \mathtt{BOUNCE}_-\}$
   (f) $\{\mathtt{ROOF}_-, \mathtt{BOUNCE}_-\}$
   の 6 つあるが, 全て起こり得ないことが容易に判定できるので, *known*, *unknown* 共に空集合となり, 実行が終了する. 生成されるハイブリッドオートマトンを図 9 に示す.
2. $y = 10$ 天井に接して, 床で弾む
   天井と衝突するとき, y' は 0 となるため FALL と ROOF は矛盾しない. その後, ケース 1 と同様に床で跳ね, ループ判定ステップに入る. 通過可能性を判定するエッジとしては,
   (a) $\{\mathtt{FALL}, \mathtt{ROOF}_+, \mathtt{BOUNCE}_+\}$
   (b) $\{\mathtt{FALL}, \mathtt{ROOF}_-, \mathtt{BOUNCE}_+\}$
   (c) $\{\mathtt{ROOF}_+, \mathtt{BOUNCE}_+\}$
   (d) $\{\mathtt{ROOF}_+, \mathtt{BOUNCE}_-\}$
   (e) $\{\mathtt{ROOF}_-, \mathtt{BOUNCE}_-\}$
   の 5 つであるが, どれも起こり得ないことが判定できる. 生成されるハイブリッドオートマトンを図 10 に示す.
3. $10 < y < 11$ 天井に衝突してから, 床で弾む
   天井と衝突するとき, FALL と ROOF は矛盾する. ケース 1, ケース 2 と同様にエッジの通過可能性が判定され, 生成されるハイブリッドオートマトンは図 11 のようになる.

*checkOccurrence* の動作を解説する. 3 つのケースにおいて *checkOccurrence* の動作の違いはなく, 判定するエッジの数が異なるだけであるため, ここではケース 1 で行われる (a) から (f) までの判定方法を解説する. まず, ループ内の全てノードにおいて y'' は一定であり, 負の値を持つことから, 離散変化以外で y' は単調減少となり, かつループ内の各 IP での初期速度も減少することから, y の最大値が減少することがわかり, ROOF のガード条件が成り立たないことがわかる. よって $\mathtt{ROOF}_+$ となる (a), (c), (d), (e) は通過しないと判定できる. 同様の理由から (b) を通過する条件 $\mathtt{y} = 0$ かつ $\mathtt{y'} = 0$ が成り立たないことがわかる. (f) は ROOF と BOUNCE のガード条件が成り立たず, かつ FALL と矛盾する場合はありえないため, 通過しないことがわかる. 以上により, (a) から (f) の全てのエッジを通過することはない, と判定できる.

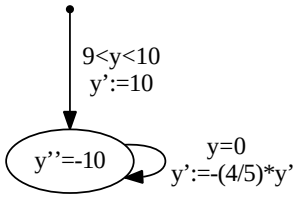


**図 9　初期高さ** $(9, 10)$ **の場合のハイブリッドオートマトン**

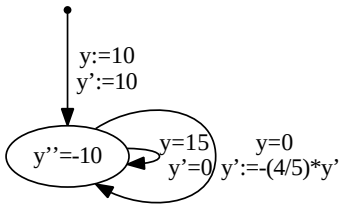


**図 10　初期高さ** 10 **の場合のハイブリッドオートマトン**

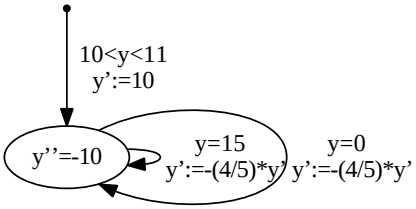


**図 11　初期高さ** $(10, 11)$ **の場合のハイブリッドオートマトン**

```
INIT <=> d > 0 & y = 10 & y' = 0 & s = 0.
S <=> []( s'=0 ).
DUR <=> []( d' = 0 ).
FALL <=> [](y'' = -10).
BOUNCE1 <=> []( y- = 5 & s- = 0 =>
          y' = -(4/5) * y'- & d = d- * 1/2 ).
BOUNCE2 <=> []( y- = 0 =>
                    y' = -(4/5) * y'- ).
BREAK <=> []( d <= 0 => s = 1).

INIT,S<<BREAK,(FALL,DUR)<<(BOUNCE1&BOUNCE2).
```

**図 12　耐久値のある床で弾む質点モデルの HydLa プログラム**

### 7.2　耐久値のある床で弾む質点モデル

高さ 5 に 0 より大きい耐久値 d を持つ板がある場合の床で弾む質点モデルを考える. 耐久値 d が 0 以下となったとき, 板は壊れ, 以降板で跳ね返ることはない. プログラムを図 12 に示す. BOUNCE1 は板での跳ね返りを表す条件付き制約であり, その後件で耐久値 d が 1/2 倍されている. 床での跳ね返りは BOUNCE2 によって表されている. BREAK は板が壊れたときにガード条件が成り立ち, スイッチ s が 1 となる. 以降, BOUNCE1 のガード条件が満たされることはない.

本アルゴリズムを実行すると, まず一度板で跳ねた後の IP で, ループ判定ステップに入る. その後 $ML$ に達するまでループを回り, 各エッジの通過可能性を判定する.

ここで本プログラムにおける *checkOccurrence* の動作を解説する. ループ内に含まれる全てのノードにおいて d'=0 であるため d はエッジでのみ変化することがわかる. さらにループ内に含まれるエッジは一つしかなく, そのエッジを通るごとに d が 1/2 倍されていくことがわかる. よって, d の値は初項 1, 公比 1/2 の等比数列であることが判断でき, 公比が 0 より大きいため, d は 0 に到達することはなく, BREAK のガード条件が成立するエッジを通過することはない, と判定できる.

仮に, BOUNCE1 の後件における d の更新の式が $\mathtt{d} = -1 + \mathtt{d}\text{-}$ である場合の *checkOccurrence* の動作を考える. d がエッジでのみ変化することは同様であるが, この場合等比数列ではなく, 公差 $-1$ の等差数列となっていることがわかる. そのため, 必ず $\mathtt{d} \leq 0$ となることが判定できる. 一般に, ループ内の全てのノードにおいて変化せず, エッジにおいて等比数列, もしくは等差数列の項で表される変数の正負判定が条件となるガード条件は判定可能である.

しかし, *checkOccurrence* の性能が低い場合や耐久値の変化がより複雑な場合, d が 0 以下になるかどうか判定できない可能性がある. このとき, 本アルゴリズムは $\mathtt{d} = 0$, $\mathtt{y}\text{-} = 5$, $\mathtt{y''}\text{-} = -10$, $\mathtt{s} = 1$, $\mathtt{s'}\text{-} = 0$, $\mathtt{s}\text{-} = 0$, $\mathtt{d'}\text{-} = 0$ を含む制約ストアを生成して, 板が壊れた先の状態を実行する.

板が壊れないことが判断できた場合に生成されるハイブリッドオートマトンを図 13 に, 判断できなかった場合に生成されるハイブリッドオートマトンを図 14 に示す. 図 14 を見ると, 冗長となるエッジとノードが生成されていることがわかる.

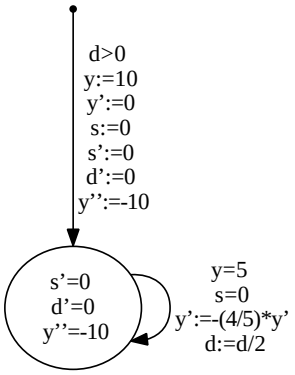


**図 13　板が壊れないことが判断できた場合のハイブリッドオートマトン**

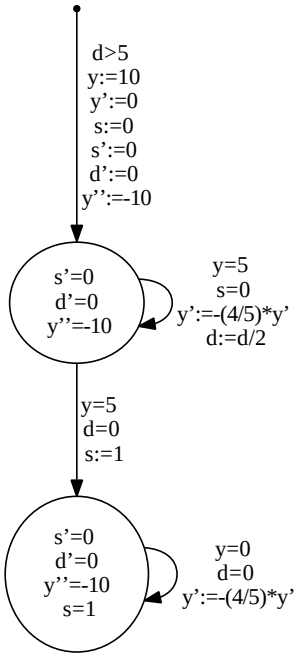


**図 14　板が壊れないことが判断できなかった場合のハイブリッドオートマトン**

## 8　まとめと今後の課題

本論文では, HydLa プログラムのハイブリッドオートマトンへの変換アルゴリズムを提案した．HydLa プログラムをハイブリッドオートマトンへ変換することで既存のハイブリッドオートマトンを用いた検証器を利用でき, 無限時間におけるシステムの到達可能性や安全性の検証が可能となる．

今後は, 本アルゴリズムを実装し, 実際にハイブリッドオートマトンの生成を行うことで, 既存の検証器を用いた検証手法や HydLa とハイブリッドオートマトンの関係性などを明らかにする．

**謝辞** 本研究を行うにあたり，貴重な意見を頂いた上田研究室 HydLa 班の皆様に感謝いたします．また，本研究の一部は, 科学研究費補助金 (基盤研究 (B) 23300011) の補助を得て行った．

## 参 考 文 献

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

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

[3] 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.

[4] 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.

[5] 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, 講演論文集, 6E–4, 2011.

[6] 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.

[7] 渋谷俊: 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, 早稲田大学大学院基幹理工学研究科, 修士論文, 2012.

[8] 潮俊光, 金澤尚史: ハイブリッドシステムの非線形現象, Fundamentals Review, 電子情報通信学会, Vol. 1, No. 1, 2007, pp. 41–50.

[9] P. Schrammel and B. Jeannet: From Hybrid Data-Flow Languages to Hybrid Automata: A Complete Translation, HSCC, 2012, pp. 167–176.

[10] M. Falaschi, A. Policriti, A. Villanueva: Time Limited Model Checking, In Proceedings of International Workshop on Specification Analysis and Validation for Emerging Technologies in Computational Logic, 2001.