Why3を用いたハイブリッドシステムの検証

石井 大輔　中島 震　Guillaume Melquiond

本論文では演繹的プログラム検証プラットフォーム Why3 を用い，ハイブリッドオートマトンの安全性検証を行う方法を提案する．提案手法はハイブリッドオートマトンを Why3 の記述言語にエンコードした後，Why3 を介して検証条件を生成，それを SMT ソルバー等の Why3 のバックエンド証明器で判定することにより，検証を実施する．実験結果では提案手法による複数の検証事例を示す．

1　はじめに

時間にともなう連続変化と離散変化からなるハイブリッドシステムは，物理現象と計算機を統合する系 (cf. cyber-physical systems) の直截的なモデルであり，その高信頼な検証技術の確立は重要課題である．

ハイブリッドシステムの検証手法として大まかに 2 つのアプローチが提案されている．1 つ目はモデル検査法に基づくアプローチで，HyTech [10]，PHAVer [6]，HybridSAL [19] 等の多くのツールが開発され，実用的な問題に応用されてきた．2 つ目は演繹的推論に基づくアプローチである．理論的な面では多くの研究がなされてきたものの，ツール実用化の面では遅れていた．例外的な研究成果として KeYmaera [15] があり，モデル検査アプローチでは扱えなかったパラメタを含む系や非線形系等，広範なクラスの問題が検証可能であることを示した．演繹的推論アプローチの欠点は，系が大きくなると，検証プロセスが自動的でなくなり，ユーザにモデルの理解と検証戦略の選択等を強いる点である．

汎用プログラムを対象とするプログラム検証の分野では，演繹的推論アプローチによる開発が活発に進んでおり，実用的なツールが実現されている．本研究は，そのような演繹的プログラム検証ツールをハイブリッドシステムの検証にどの程度利用できるのか探るものである．具体的には，プログラム検証プラットフォーム Why3 [3] (3 節) を用いてハイブリッドオートマトン (2 節) の安全性検証を行う方法を提案する．提案手法ではハイブリッドオートマトンを Why3 の記述言語へと簡潔にエンコードする (4 節)．その後，Why3 の処理系が最弱事前条件の計算により検証に必要な条件群を自動生成し，それらを SMT ソルバー等の Why3 のバックエンド証明器により妥当性判定することで，検証を実施する (4.4 節)．本研究は，Why3 の柔軟な機構を利用し，簡潔で見通しのよい検証プロセスを目指す点を特徴とする．複雑な例題の検証においては単純なエンコーディングでは不十分な場合があるため，5 節ではエンコーディングを詳細化するための複数の方法を述べる．提案手法を文献由来の複数の例題に適用し，その有効性を示す (6 節)．7 節では関連研究について述べる．

2　ハイブリッドオートマトン

ハイブリッドオートマトン (hybrid automata, 以下「HA」) [9] はハイブリッドシステムを表す数学モデルの 1 つである．

Daisuke Ishii, Shin Nakajima, 国立情報学研究所, National Institute of Informatics
Guillaume Melquiond, INRIA Saclay–Île-de-France/ Université Paris Sud 11

$$\frac{\phi(0)=\nu \quad t>0 \quad \forall \tilde{t}\in(0,t]\ (\frac{d\phi(\tilde{t})}{dt}=F_l(\phi(\tilde{t}))\wedge I_l[\phi(\tilde{t})])}{\langle l,\nu\rangle \xrightarrow{t} \langle l,\phi(t)\rangle} \quad (1) \qquad \frac{G_{l_1,l_2}[\nu_1] \quad T_{l_1,l_2}[\nu_1,\nu_2] \quad I_{l_2}[\nu_2]}{\langle l_1,\nu_1\rangle \xrightarrow{0} \langle l_2,\nu_2\rangle} \quad (2)$$

図 1　ハイブリッドオートマトンの操作的意味論

定義 1 ハイブリッドオートマトンを以下の要素からなる 7 つ組 $HA=\langle L, V, Init, \mathcal{G}, \mathcal{T}, \mathcal{F}, \mathcal{I}\rangle$ で表す:

- ロケーションの有限集合 L.
- 実変数の有限集合 $V=\{x_1,\ldots,x_n\}$. 系のすべての付値の集合を $\mathbb{R}^V$ で表す.
- 初期条件 $Init \subseteq L\times\mathbb{R}^V$.
- ガード条件の族 $\mathcal{G}=\{G_{l,l'}\}_{l\in L,l'\in L}$. ただし $G_{l,l'}$ は $\mathbb{R}^V$ に関する条件である.
- リセット条件の族 $\mathcal{T}=\{T_{l,l'}\}_{l\in L,l'\in L}$. ただし $T_{l,l'}$ は $\mathbb{R}^V\times\mathbb{R}^V$ に関する条件である.
- ベクトル場の族 $\mathcal{F}=\{F_l\}_{l\in L}$. ただし $F_l : \mathbb{R}^V\to\mathbb{R}^V$ とする.
- ロケーション不変条件の族 $\mathcal{I}=\{I_l\}_{l\in L}$. ただし I_l は $\mathbb{R}^V$ に関する条件である.

列 $\langle l_0,\nu_0\rangle \xrightarrow{t_1} \sigma_1 \xrightarrow{0} \sigma_2 \xrightarrow{t_3} \cdots \xrightarrow{0} \sigma_{2k}$ を HA の長さ k の実行と呼ぶ. ただし $\langle l_0,\nu_0\rangle\in L\times\mathbb{R}^V$, $\sigma_i\in L\times\mathbb{R}^V$, $Init[\langle l_0,\nu_0\rangle]\wedge I_{l_0}[\nu_0]$ が成り立つとする. $\xrightarrow{t_i}$ を $t_i>0$ のとき連続変化, $t_i=0$ のとき離散変化とよび, 図 1 の 2 つのルールで定義する.

本論文では, 複数の離散変化が同時刻に起こらず, 最初に $\langle l_0,\nu_0\rangle \xrightarrow{0} \sigma_1$ のように離散変化が起こることもないものと仮定する. 一般的な実行に関する検証は今後の課題とする.

定義 2 安全性条件 (不変性ともいう) を式 $\Box P$ で表す. ただし P は $L\times\mathbb{R}^V$ に関する条件である. 式 P は, 各実行の初期状態に関して成り立ち ($\sigma_0\models P$), 各連続変化および離散変化において保存される ($\sigma_i\models P$).

例 1 (水槽の水位制御 [2] [13]) 水位制御装置付きの水槽を図 2 の HA でモデリングする. ロケーション *off* および *sw-on* では水槽の出力孔から水が放出されるため, 水位 y が一定速度 $rate_{out}$ で減少し, ロケーション *on* および *sw-off* では制御装置が水を注入するため, y が一定速度 $rate_{in}$ で増加する. 制御装置は y が値 *low* あるいは *high* に達するのに応じてロケーション *on* と *off* を切り替える. さらにロケーション *sw-on* および *sw-off* によって上記の切り替えに遅延 *delay* が発生することを表している. ここでは水位 y が限界値 *min* と *max* の間につねに収まっていることを, つまり次の安全性条件を検証する:

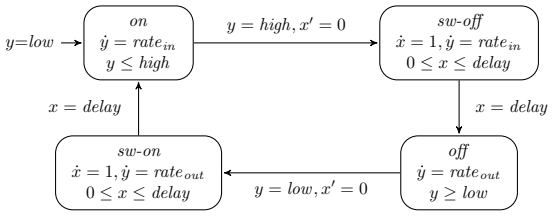

図 2　水槽の水位制御のモデル.

$$\Box(min\le y\le max). \quad (3)$$

また各定数パラメタを下記のように制約する:

$$min\le max \wedge low\ge min \wedge high\le max \wedge$$
$$high>low \wedge delay>0 \wedge$$
$$max\ge high+rate_{in}\cdot delay \wedge rate_{in}=2 \wedge$$
$$min\le low+rate_{out}\cdot delay \wedge rate_{out}=-3. \quad (4)$$

3　Why3 フレームワーク

本研究では検証のプラットフォームとして **Why3** [3] (バージョン 0.80) を用いる (Why [5] の後継). Why3 は, OCaml 風の構文で記述されたアノテーション付きプログラムを入力とし, プログラムの正当性を証明するための検証条件を計算し, 既存の証明器群を用いて各論理式の妥当性判定を行う作業を自動化するツールである. Why3 による検証プロセスを図 3 に示す. Why3 のプリプロセッサは入力として C や Java のプログラムや HA を受け取り, Why3 の記述言語でエンコードする. 記述言語は **Why3** と **WhyML** と呼ばれる 2 種類の言語からなり, それぞれ事前・事後条件を記述するための論理言語, プログラムを記述するためのプログラミング言語となっている. WhyML プログラムに対して, Why3 言語の記述をアノテートする

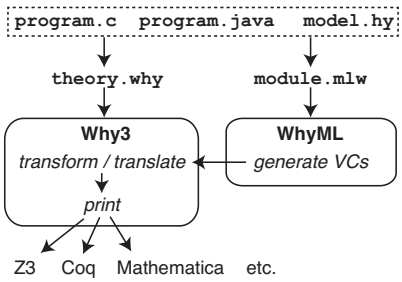

図 3　**Why3** の検証プロセス

file	::=	(*theory* | *module*)∗	(1)
theory	::=	`theory` *tid decl*∗ `end`	(2)
decl	::=	`use import` *tid*	(3)
	|	`type` *id* `=` *type-decl*	(4)
	|	`function` *id targs* `:` *ty* (`=` *fml*)	(5)
	|	`predicate` *id targs* (`=` *fml*)	(6)
	|	`axiom` *id* `:` *fml*	(7)
module	::=	`module` *mid m-decl*∗ `end`	(8)
m-decl	::=	`use import` (*tid*| *mid*)	(9)
	|	`let` *id args* `=` *pgm-defn*	(10)
	|	`val` *id targs* `:` *ty* (*annot*)	(11)

図 4　**Why3/WhyML** 言語の基本構文

ことができる．Why3 はアノテートされた WhyML プログラムから必要な検証条件を導出し，検証条件を各バックエンド証明器の入力形式に変換する．バックエンド証明器としては，SMT ソルバー (例: Z3[†1])，定理証明器 (例: Coq[†2])，計算機代数システム (例: Mathematica[†3]) 等が用意されている．

Why3 の検証手法は **Floyd-Hoare 論理** [11] に基づいており，プログラムの各ブロックに対して事前条件と事後条件を導出し，検証条件を生成する (場合によっては補助的な表明や不変条件も用いる)．事前・事後条件はプログラムの状態を参照する自由変数をもつ一階述語論理式である．Why3 は事前・事後条件が付随した各ブロックについて最弱事前条件 [4] の計算によって検証条件を求める．

記述言語の基本構文を図 4 に示す．Why3 言語と WhyML 言語による記述は，それぞれセオリーあるいはモジュールを定義する (ルール 1)．セオリーは型，関数，述語，公理等の定義からなる (ルール 4–7)．モジュールは，関数定義 (ルール 10)，関数のシグネチャのみの定義 (ルール 11)，それらの定義中にアノテートした仕様記述からなる．図では省略したが，関数定義では参照型 (例: `i := !i+1`)，ループ (例: `loop` 構文)，パターンマッチング (例: `match` 構文)，レコード (例: `{loc=On; x=10.}`) 等を記述できる (4 節の例を参照されたい)．

4　Why3 を用いた HA の安全性検証

本研究では HA を，その実行をシミュレートするプログラムに変換する．このプログラムを実行中の「状態」は HA 実行中のある「状態」 ($\sigma \in L \times \mathbb{R}^V$) を表している．また HA の安全性条件をプログラムの状態に関する条件に書き換える．その上で，プログラム実行中のすべての状態が条件をみたすことを確認できれば，HA の任意の実行が安全性条件をみたすことを示すことができる (ただしプログラムに明示されない HA の連続変化中の状態についても確認する必要がある)．

提案手法は，HA $\langle L, V, Init, \mathcal{G}, \mathcal{T}, \mathcal{F}, \mathcal{I} \rangle$ と安全性条件 $\Box P$ を，図 5 に示すように Why3 セオリー `Model` と WhyML モジュール `Controller` とにエンコードする．`Model` は HA を記述するのに必要となる型，述語，関数を定義する．`Controller` は外部環境と通信する離散的なコントローラを実装する．さらに，`Model` に用意した型・述語・関数を用いて `Controller` をアノテートすることにより，連続変化の記述と離散変化の記述 (コントローラ実装) とが結合される．

Why3 フレームワークを用いてエンコードしたプログラムの正当性を示す．すなわち，プログラム実行中の各状態が，アノテートされた事前・事後条件，表明，ループ不変条件等をみたすことを示す．Why3 はプログラムの停止性を検証することも可能だが，HA の実行は無限長となりうるため，提案手法では停止性は検証しない (無限長となる場合にはプログラムの実行も停止しない)．

以下，例 1 の HA と安全性条件 (3) を Why3 にど

†1 `http://research.microsoft.com/en-us/um/redmond/projects/z3/`
†2 `http://coq.inria.fr/`
†3 `http://www.wolfram.com/mathematica/`

```
theory Model
  use import real.Real (* 実数セオリーを導入. *)
  ... (* 4.1, 4.2 節を参照. *)
end

module Controller
  use import real.Real (* 実数セオリーを導入. *)
  use import Model (* Model 中の定義を導入. *)
  use import ref.Ref (* 参照型を使用. *)
  ... (* 4.3 節を参照. *)
end
```

図 5　HA をエンコードする Why3 プログラムのスタブ

のようにエンコードし (4.1–4.3 節)，Why3 を用いてどのように検証するのか (4.4 節) を述べる．

4.1　HA 状態のエンコーディング

HA の状態をエンコードするために，Model セオリー中に 2 つのユーザ定義型を用意する．まず HA のロケーション L を列挙型 loc で表す:

```
type loc = On | Sw_off | Off | Sw_on
```

次に状態 $(l,(x,y)) \in L \times \mathbb{R}^V$ をレコード型 state で表す:

```
type state = {loc:loc; x:real; y:real}
```

フィールド loc がロケーションを保持し，他のフィールドは V 中の変数の値を保持する．

4.2　モデルの記述

Model セオリーには個々のモデルの内容を関数と述語の定義としてエンコードしていく．

モデル中の実数パラメタは real 型の関数記号で表す:

```
function max:real     function min:real
function low:real     function high:real
function delay:real
function rate_in:real function rate_out:real
```

上記パラメタに関する制約 (4) を公理 Constraints として記述する:

```
axiom Constraints :
  min <= max /\ low >= min /\ high <= max /\
  high > low /\ delay > 0. /\
  max >= high+rate_in*delay /\ rate_in = 2. /\
  min <= low+rate_out*delay /\ rate_out = -3.
```

初期条件 $Init$ を state 型のオブジェクト s に関する述語 init として定義する:

```
predicate init (s:state) = s.loc=On /\ s.y=low
```

ロケーション不変条件 I を述語 inv として定義する:

```
predicate inv (s:state) =
  match s.loc with
  | On     -> s.y<=high
  | Off    -> s.y>=low
  | Sw_on  -> 0.<=s.x<=delay /\ s.y<=high
  | Sw_off -> 0.<=s.x<=delay /\ s.y>=low
  end
```

s.loc にパターンマッチングを適用して各ロケーション $l \in L$ について不変条件 I_l を記述する．

ベクトル場 $\mathcal{F}$ にもとづく連続変化を述語 flow として定義する:

```
predicate flow (d:real) (s_:state) (s:state) =
  s.loc = s_.loc /\
  match s.loc with
  | On     ->
    s.x = s_.x /\ s.y = s_.y + rate_in *d
  | Off    ->
    s.x = s_.x /\ s.y = s_.y + rate_out*d
  | Sw_on  ->
    s.x = s_.x + d /\ s.y = s_.y + rate_out*d
  | Sw_off ->
    s.x = s_.x + d /\ s.y = s_.y + rate_in *d
  end
```

flow は，各ロケーション $l \in L$ について，状態 s_ がロケーション s.loc において時間 d だけ連続変化した後，状態 s となることを表明している．flow の定義はあらかじめ微分方程式を解いた閉形式として記述する．

ガード条件 $\mathcal{G}$ を述語 guard として定義する:

```
predicate guard (s:state) (dst:loc) =
  match s.loc, dst with
  | On, Sw_off  -> s.y = high
  | Off, Sw_on  -> s.y = low
  | Sw_on, On   -> s.x = delay
  | Sw_off, Off -> s.x = delay
  | _ -> false
  end
```

パターンマッチングによって離散変化前後のロケーション $(l_1,l_2) \in L \times L$ を列挙し，G_{l_1,l_2} を定義する．最後の分岐は未定義の離散変化，あるいは同じロケー

ションに留まることを禁止するためのものである．

リセット条件 $\mathcal{T}$ を関数 trans として定義する:

```
function trans (s_:state) (dst:loc) :state =
  match s_.loc, dst with
  | On,Sw_off  -> {loc=dst; x=0.; y=s_.y}
  | Off,Sw_on  -> {loc=dst; x=0.; y=s_.y}
  | Sw_on,On   -> {loc=dst; x=s_.x; y=s_.y}
  | Sw_off,Off -> {loc=dst; x=s_.x; y=s_.y}
  | _ -> s_
  end
```

trans は状態 s がロケーション s_.loc から dst へ離散変化後の状態を返す．$\mathcal{G}$ 同様に離散変化前後のロケーションの組み合わせ毎に T_{l_1,l_2} を定義していく．

安全性条件 (3) を述語 property として定義する:

```
predicate property (s:state) = min <= s.y <= max
```

4.3 コントローラの記述

(有限あるいは無限長の) HA 実行のシミュレーションのメインループを Controller モジュールの evolve 関数に実装する:

```
let evolve s_
  assumes { init s_ }
= let state = ref s_ in
  loop
    invariant { inv !state /\ property !state }
    let d,s = wait !state in
    assert { forall t:real. (0.<=t<=d ->
             forall s:state.
             (flow t !state s -> property s)) };
    let dst = detect s in
    state := trans s dst
  end
```

state 型の引数 s_は初期状態を表し，事前条件を記述するための assumes 節内で s_が初期条件をみたすことを表明している．事後条件は記述していない．loop 内の繰り返し処理は HA の離散・連続変化を計算し，結果を state 参照型の変数 state に保持する．各繰り返しでは，それぞれ，連続変化，ガード条件の評価，離散変化を計算するサブルーチン wait，detect，trans を呼ぶ．invariant 節は，各繰り返しの計算前に成り立つべきループ不変条件を記述するためのもので，ここでは state がロケーション不変条件と安全性条件をみたすことを記述する．

wait 関数は連続変化を計算する:

```
val wait (s_:state) :(real,state)
  ensures { let d,s = result in d>0. /\
    flow d s_ s /\ forall t:real. (0.<t<=d ->
    forall s:state. (flow t s_ s -> inv s) ) }
```

wait は，状態 s_が与えられると，連続変化の経過時間と結果の状態の組 result $= (\mathtt{d}, \mathtt{s})$ を返す．WhyML 言語ではシグネチャだけを記述し，戻り値に関して事後条件 (ensures 節) として，ルール (1) の前提条件を theory の述語を用いて記述している．

detect 関数は現在の状態 s_があるロケーションへの離散変化のガード条件をみたすかどうかを調べ，離散変化後のロケーションを返す:

```
val detect (s_:state) :loc
  ensures { guard s_ result }
```

ここでは wait 関数同様に事後条件として計算内容を記述する．detect は，wait が計算した連続変化後の状態がすぐに次の離散変化することを要請していることに注意されたい．

最後に，trans 関数が離散変化を計算する:

```
let trans s_ dst
  requires { guard s_ dst /\ inv s_ }
  ensures { result = trans s_ dst /\ inv result}
= trans s_ dst
```

trans は，現在の状態と離散変化後のロケーションのロケーションが与えられると，Model セオリーの trans 関数を用いて状態を計算する．事前・事後条件にはルール (2) の前提条件を記述し，trans が HA の離散変化を計算していることを確認している．

4.4 安全性の検証

上記のエンコード結果を入力として Why3 を実行すると，下記のような，evolve 関数に関する 4 つの検証条件 VC_1–VC_4 と，trans 関数に関する検証条件 VC_5 が生成される:

- VC_1: すべての初期状態がループ不変条件 (ロケーション不変条件と安全性条件) をみたすこと．
- VC_2: ループ内の表明．連続変化中につねに安全性条件をみたすこと．
- VC_3: trans 関数呼び出し時に事前条件 (ある離散変化のガード条件) をみたすこと．
- VC_4: ループ実行後にループ不変条件をみたす

こと．

- VC_5: `trans` 関数の事前・事後条件が妥当であること．

上記すべての検証条件の妥当性判定に成功すれば，元の HA の安全性を検証したことになる．VC_2–VC_5 は，ループ不変条件をみたす任意の状態 σ_n が，$\sigma_n \xrightarrow{d_{n+1}} \sigma_{n+1} \xrightarrow{0} \sigma_{n+2}$ のようにある時間 d_{n+1} だけ (安全性を保ちながら) 連続変化し，その後離散変化して再度ループ不変条件をみたす状態 σ_{n+2} に到達することを確認しており，VC_1 を基礎とした帰納法によって検証を行っている．

サブルーチンを `wait` と `detect` とに分けたのはあるロケーションに留まり続けるような系を検証するためである．すなわち提案手法では，VC_2 は妥当だがその後ガード条件がみたされることがない場合でも，すべての検証条件が妥当となる．実際，VC_3 と VC_4 は `detect` の `ensures` 節を仮定するが，`ensures` 節はガード条件の成立を要請するので偽となり，両検証条件は妥当となる．

バックエンド証明器の Z3 を用いて，上記の VC_2 以外の検証条件をそれぞれ 0.05 秒以内で証明できる．VC_2 の判定に失敗するのは，ロケーション `Sw_on` と `Sw_off` のロケーション不変条件が y の上限あるいは下限のみしか制約しておらず，メインループの連続変化の計算においても上限あるいは下限のみを求め，安全性条件を判定するのに不十分なためである．検証を成功させるためにはループ不変条件を以下のように修正する:

```
invariant { inv !state /\ property !state /\
 (!state.loc = Sw_off  -> !state.y = high) /\
 (!state.loc = Sw_on   -> !state.y = low) }
```

修正後，Z3 を用いて VC_2 を 0.04 秒で証明することができた．

5 検証戦略

前節で述べたエンコーディングでは，`Controller` 内で HA 実行を 1 回ずつの連続変化と離散変化のループに抽象化して記述している．複雑な安全性条件を検証するためには，エンコードを詳細化し，HA 実行中の状態をより具体的に記述した上で証明を行う必要がある．本節ではエンコード方法を調整するための検証戦略について述べる．

ループ展開． この検証戦略では `evolve` 関数内のメインループの内容を複製する．複製コードは，ループ内あるいはループ前に展開する．ループ内に展開することにより，複数ステップ周期の不変条件を記述できるようになる．またループ外に展開することにより，例外的な初期の複数ステップにわたる挙動をメインループから分離することができる．

ループ不変条件の追加． 4.4 節において行ったように，ループ不変条件を追加するのが有用な場合がある．この戦略はループ不変条件に $L \times \mathbb{R}^V$ に関する表明を追加する．

表明の追加． この戦略はあるステップの実行コードの終わりに $L \times \mathbb{R}^V$ に関する表明を追加する．

上記の表明において現在地点の前後の HA 状態や連続変化時間について記述することもできる．たとえばループ不変条件において，ループ内の最初の 1 ステップの連続変化時間と実行後の状態とについて，下記のように述べることができる:

```
let d1,s1 = wait !state in let _ = detect s1 in
  (* ここで !state は更新しない. *)
loop
  invariant { a_predicate_on d1 s1 }
```

以下，検証戦略を用いた検証の例を述べる．

例 2 例 1 は，4.4 節で述べた方法のほか，ループ不変条件に `(!state.loc = On \/ !state.loc = Off)` を追加し，ループ内が 2 ステップ分になるように展開しても検証することができる．

例 3 (ガスバーナー [2]) 図 6 の線形 HA を考える．HA は，2 つのロケーション $L = \{leaking, non\text{-}leaking\}$ と，ガス漏れ時間，ロケーション滞在時間，総実行時間を表す 3 つの変数 $V = \{t, x, y\}$ を持つ．この HA について下記の安全性条件を検証する:

$$\Box(y \geq 60 \implies 20t < y) \tag{5}$$

前件 $y \geq 60$ や後件 $20t < y$ を評価するためにはエンコードディングを詳細化する必要がある．

この HA の実行例を図 7 に示す．実行では 2 つの

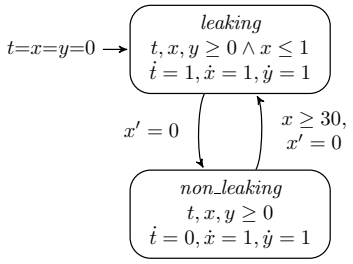

図 6　ガスバーナーのモデル.

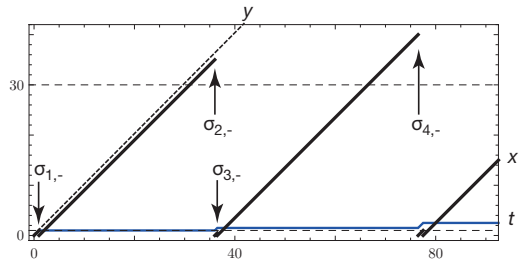

図 7　ガスバーナーの実行例.

ロケーションを交互に訪れるので，ループ内で 2 ステップ分展開すれば，ループ不変条件でのロケーションを特定できる．最初の 4 ステップについて下記の性質が成り立つ:

$\sigma_0 \models \mathit{leaking} \wedge t = 0 \wedge x = 0 \wedge y = 0$

$\sigma_{1,-} \models \mathit{leaking} \wedge 0 \leq t \leq 1 \wedge 0 \leq x \leq 1 \wedge 0 \leq y \leq 1$

$\sigma_{2,-} \models \mathit{non\text{-}leaking} \wedge 0 \leq t \leq 1 \wedge x \geq 30 \wedge y > 30$

$\sigma_{3,-} \models \mathit{leaking} \wedge 0 \leq t \leq 2 \wedge 0 \leq x \leq 1 \wedge y > 30$

$\sigma_{4,-} \models \mathit{non\text{-}leaking} \wedge 0 \leq t \leq 2 \wedge x \geq 30 \wedge y > 60$

ただし $\sigma_{i,-}$ は i 番目の離散変化前の状態を表す．$\sigma_{4,-}$ に至って安全性条件の前件 $y \geq 60$ が成り立ち，以降の実行では，条件 $y \geq 60 \wedge 20t < y$，さらには条件 $y > 60 \wedge 30t < y$ がつねに成り立つ.

以上から，ループ前に 4 ステップ分展開し，ループ不変条件を下記のように設定する:

```
invariant { inv !state /\ property !state /\
  !state.loc = Leaking /\ !state.x = 0. /\
!state.y > 60. /\ !state.t < (1./30.)*!state.y}
```

6　実験結果

提案エンコード手法を HyTech の HA 記述言語 (検証戦略も記述できるように拡張) から Why3/WhyML 言語へのトランスレータとして実装した†4.

提案手法の実用性を評価するため，複数の例題を検証する実験を行った．また，比較のために同じ例題を既存ツール (HyTech, PHAVer, KeYmaera) を用いて検証した．実験結果を表 1 に示す．各列は，各 HA のロケーション数，変数の数，ループ展開数 (ループ前/ループ内)，Why3 が生成した検証条件の数，バックエンド証明器が全検証条件を判定するのに要した CPU 時間，HyTech (H) あるいは PHAVer (P) による検証に要した時間，KeYmaera による検証に要した時間，を示している (「–」は検証が失敗したことを示す)．バックエンド証明器には Z3 (バージョン 3.2), Mathematica (バージョン 8.0.4)，CVC3 (バージョン 2.4.1) を用いた．実験は 3.4GHz Intel Xeon プロセッサと RAM 4GB を備えたマシン上で実施した.

6.1　例題

Water tank (例 1). 提案手法を用い，4.4 節で述べたように検証できる．HyTech は効率よくこの問題を検証することができる．また KeYmaera でも検証することができるが，モデルに (4.4 節のものよりも複雑な) ループ不変条件を追加する必要がある.

Gas burner (例 3). 例 3 で述べたように，検証戦略を適用して検証することができる．HyTech は効率よくこの問題を検証することができる．KeYmaera ではループ不変条件を追加しても検証できなかった.

Temperature control [2]. あるロケーションから複数のエッジが出ている HA である．[2] は安全性の必要十分条件として，隣り合う 3 つの連続変化の経過時間に関する条件を示しているが，これをループ不変条件として設定することで検証することができた．HyTech は効率よくこの問題を検証することができる．KeYmaera ではループ不変条件を追加しても検証できなかった.

Bouncing ball. 平坦な床の上で跳ね返るボールをモデリングした非線形 HA である．ボールの初期位置，初速，床の反発係数をパラメタ化し，反発係数が 1 以下ならばボールの高さがボールの初期エネ

†4 http://www.dsksh.com/whybrid/ にて公開.

表 1　実験結果.

例題	locs	vars	unroll	VCs	CPU 時間	モデル検査器	KeYmaera
water tank (例 1)	4	2	0/1	5	0.15s	–	1.8s
gas burner (例 3)	2	3	4/2	15	1.08s	0.004s (H)	–
temp. control	4	3	4/2	15	5.77s	0.012s (H)	–
bouncing ball	1	2	1/1	7	36.0s	–	0.9s
highway 12	13	12	0/2	7	87.3s	25.5s (P)	–
highway 12 (opt.)			0/2	7	2.46s		
highway 24 (opt.)	25	24	0/2	7	14.0s	–	–

ルギーを超えないことを検証した．4 つの検証条件は Mathematica を用いて証明した．このようにパラメタ化した HA はモデル検査器では検証できない．KeYmaera は同様のモデルを効率よく検証できる．

Highway [12]. n 台の自律走行する車からなる高速道路のモデルである．$n = 12$ と $n = 24$ のインスタンスを検証した．$n = 12$ の場合は Z3 で検証できたものの，$n = 24$ の場合は検証が 20 分以内に終了しなかった．これは `trans` 関数中の分岐数の増加によるものだと考えられる．実際どの分岐でも変数値をリセットすることがないので，`trans` 関数を分岐を用いない形に変更したところ，CVC3 を用いて効率よく検証を行うことができた (“highway 12/24 (opt.)”)．PHAVer を用い，$n = 12$ の場合は検証できたが，$n = 24$ の場合にはメモリを使い切ってしまい検証できなかった．KeYmaera は同モデルを検証できなかった．

6.2　考察

モデル検査器 HyTech および PHAVer を用いて 3 つの例題を効率よく検証することができたが，他の問題は検証できなかった．提案手法はいくつかの点で優れているといえる．まず，提案手法はパラメタを含むモデルを扱うことができる．Bouncing ball はパラメタを含むためにモデル検査器で検証できなかった (具体化すれば検証できる)．次に，提案手法はスケーラビリティが高い．[12] の実験では，PHAVer は highway の例を $n = 15$ までしか検証できていない．

KeYmaera を用いて多くのハイブリッドシステムを自動的に検証できるが，HA としてモデル化したハイブリッドシステムに関しては検証がうまくいかないことが多い．Water tank の例ではモデルにループ不変条件を付加する必要があった．自動検証がうまくいかない場合，ユーザは定理証明器を対話的に用いて演繹的な証明を行うこともできるが，KeYmaera は 141 の推論ルールを備え，使いこなすのは習熟を要する [15]．本稿の提案手法では，自動検証がうまくいかない場合には 5 節で述べた検証戦略を適用するが，戦略の数は限られており，またループ展開の試行やループ不変条件の導出はある程度自動化することが可能だと考えている．

7　関連研究

ハイブリッドシステムのための演繹的検証ツールが複数提案されている [13] [1] [8] [18]．提案手法と同様，これらのツールではハイブリッドシステムを利用する検証フレームワークの記述言語 (例: STeP [13]，PVS [1]，SAL [7], Event-B [18]) にエンコードする．いずれのツールにおいても検証プロセスが完全には自動化されておらず，ユーザが付加的な情報を入力したり，対話的な定理証明を行ったりする必要がある．

Platzer らが開発した KeYmaera [15] [14] が近年，実用的な検証ツールとして注目されている．KeYmaera は独自のモデリング言語を備え，ハイブリッドシステムを hybrid program として記述し，differential (algebraic) dynamic logic で仕様や検証条件を付加する．また，検証には独自の定理証明器を用いる [14]．これに対して本研究では，Why3 という汎用の

演繹的検証フレームワークを用いることにより，より「軽量な」検証手法を提案している．提案する検証プロセスは、階層的な枠組みになっている．

ハイブリッドシステムの多項式不変条件を生成する代数的手法 [17] [16] が提案されている．将来課題として，これらを提案手法に組み込むことが考えられる．

8 まとめと今後の課題

HA の安全性検証は多くの研究が蓄積されてきた重要な研究課題である．本研究では，モデル検査法に基づく既存ツール群とは対照的に，演繹的手法による線形および非線形 HA の検証ツールを提案した．提案手法は Why3 を利用して簡潔に HA を表現し，帰納法とループ展開に基づく検証戦略により検証を行う．複数の典型的な HA の検証に成功するとともに，既存ツールでは扱えなかったインスタンスについても検証することができた．

今後の課題としては，検証戦略の自動発見や，同アプローチによるより複雑な HA の検証がある．提案手法で扱う HA のクラスを一般化したり，新しい証明器を導入したりする際には，Why3 の拡張可能な機構が役立つことが期待できる．

謝辞 本研究の一部は科研費 23-3810 の補助を得て行った．

参考文献

[1] E. Ábrahám-mumm, M. Steffen, and U. Hannemann. Verification of hybrid systems: Formalization and proof rules in PVS. In *ICECCS*, pages 48–57, 2001.

[2] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[3] F. Bobot, J.-C. Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *International Workshop on Intermediate Verification Languages (Boogie)*, 2011.

[4] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[5] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV*, *LNCS* 4590, pages 173–177, 2007.

[6] G. Frehse. PHAVer: Algorithmic verification of hybrid systems past HyTech. *International Journal on Software Tools for Technology Transfer (STTT)*, 10(3):263–279, 2008.

[7] R. Ghosh, A. Tiwari, and C. Tomlin. Automated symbolic reachability analysis: with application to delta-notch signaling automata. In *HSCC*, *LNCS* 2623, pages 233–248, 2003.

[8] I. Hasuo and K. Suenaga. Exercises in nonstandard static analysis of hybrid systems. In *CAV*, *LNCS* 7358, pages 462–478, 2012.

[9] T. A. Henzinger. The theory of hybrid automata. *Verification of Digital and Hybrid Systems (NATO ASI Series F: Computer and Systems Sciences)*, 170:265–292, 2000.

[10] T. A. Henzinger, P-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *STTT*, 1:110–122, 1997.

[11] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, 1969.

[12] S. K. Jha, B. H. Krogh, J. E. Weimer, and E. M. Clarke. Reachability for linear hybrid automata using iterative relaxation abstraction. In *HCSS*, *LNCS* 4416, pages 287–300, 2007.

[13] Z. Manna and H.B. Sipma. Deductive verification of hybrid systems using STeP. In *HSCC*, *LNCS* 1386, pages 305–318, 1998.

[14] A. Platzer. *Logical Analysis of Hybrid Systems*. Springer, 2010.

[15] A. Platzer and J. D. Quesel. KeYmaera: A hybrid theorem prover for hybrid systems. In *IJCAR*, *LNCS* 5195, pages 171–178, 2008.

[16] E. Rodriguez-Carbonell and A. Tiwari. Generating polynomial invariants for hybrid systems. In *HSCC*, *LNCS* 3414, pages 590–605, 2005.

[17] S. Sankaranarayanan, H.B. Sipma, and Z. Manna. Constructing invariants for hybrid systems. In *HSCC*, *LNCS* 2993, pages 539–554, 2004.

[18] W. Su, J.-R. Abrial, and H. Zhu. Complementary methodologies for developing hybrid systems with Event-B. In *ICFEM*, pages 230–248, 2012.

[19] A. Tiwari. HybridSAL relational abstracter. In *CAV*, *LNCS* 7358, pages 725–731, 2012.