修 士 論 文

定性シミュレーションの効率化に関する研究

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

落合 紀雄

2004 年 3 月

修 士 論 文

定性シミュレーションの効率化に関する研究

指導教官　平石 邦彦 教授

審査委員主査　平石 邦彦 教授
審査委員　金子 峰雄 教授
審査委員　宮地 充子 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

110030 落合 紀雄

提出年月: 2004 年 2 月

目 次

図 目 次

表 目 次

第1章　はじめに

定性シミュレーション [1][8] は, システムの定性的振る舞いのみを考慮した解析手法であり, ハイブリッドシステムでシステムに対する不完全な情報しか得られない場合に対して有効なシミュレーション法である．そのシミュレーションの対象として, システムバイオロジーの分野で, 遺伝子発現過程のシミュレーションの応用が期待される [9].

既存の定性シミュレータとして, QSIM[1] があるが, QSIM のアルゴリズムは，到達可能な状態の候補を生成し, 定性微分方程式に整合する状態のみを残すという操作の繰り返しであり, 非常に多くの状態を同時に扱う必要がある. そのため, 扱う変数の数が多くなると, 生成する状態数が爆発的に増え, シミュレーションを行うことができなくなる. そこで, 既存の離散状態システムに対する効率的解析手法である BDD (Binary Dicision Diagram) の利用を検討する.

BDD は論理関数の表現方法の 1 つであり, 真理値表や論理式の積和形表現に比べよりコンパクトに関数を表現できるため, 効率的に扱える論理関数の範囲を飛躍的に広げたという点でさまざまな分野に大きな影響を与えている. 例えば, LSI 設計システムの基盤技術においては, 論理関数を計算機上で効率よく表現し, 高速に論理演算を行うことが重要であり, 論理関数のデータ表現に BDD を用いる処理方法は, 記憶効率や計算速度の面で優れており, 盛んに用いられている [6].

定性シミュレーションは, 到達可能な状態の候補を生成し定性微分方程式に定義された変数間の満たすべき制約条件を次々に適用していくことにより, 到達可能な状態の候補から到達可能状態が求まり, この操作を新たな到達状態に遷移しなくなるまで繰り返す. 言い換えれば, 1 ステップで到達可能な状態の集合はすべての制約条件を満たす到達可能な状態の候補であるといえる. そこで, 定性シミュレーションを BDD で表現することにより, 定性シミュレーションの効率化を図ることを試みる. 各制約条件それぞれについて成り立つ状態を求める操作は, BDD で表現された各制約条件について論理積をとり, その BDD を満たすものを求める操作に相当する. つまり, BDD でコーディングすることにより, 次状態の候補から次状態を求める操作が 1 回の論理演算で終わることになる.

ところで, BDD を用いた定性シミュレーション法では, 到達可能な状態集合が計算できたとしてもどのような経路を通って到達可能状態に到達したのか判別できない. そこで, 経路に依存した性質が成り立つか判定するためにモデル検査の手法を適用する.

CTL (Computational Tree Logic) は時相論理の一種である. 時相論理とは命題の真偽値が時間的に変化する論理体系であり, 有限状態機械の満たすべき仕様を記述するのに適している. 定性シミュレーションは初期状態から遷移可能な状態へ次々と状態が遷移して

いく点において, 有限状態機械と見なすことができる. そこで, 定性シミュレーションにおいて時相論理式で記述されるような性質の検証を行うため, CTL を用いた検証手法である CTL 検査を行う.

本研究では, BDD を用いて定性シミュレーションの実装を行い, 従来のアルゴリズムに基づき Java で実装された定性シミュレータである JQSIM と計算時間とメモリ効率に関する比較実験を行った. 実験にはバスタブを多数連結したモデルを使用した.

定性シミュレーションは，連続系，あるいは，ハイブリッド系を定性的に解析する手法である. この手法は，システムバイオロジーの分野での遺伝子発現過程のような，システムに対する不完全な情報しか得られない場合に有効な手法となる.

既存の定性シミュレーションツールとして, QSIM [1] があるが, QSIM のアルゴリズムは，到達可能な状態の候補を生成し, 定性微分方程式に整合する状態のみを残すという操作の繰り返しであり，非常に多くの状態を同時に扱う必要がある．そのため，扱う状態数が多くなると，すぐに計算機資源を使い果たしてしてまうため，扱える状態数が非常に限られたものとなる．そこで，離散状態システムに対して開発された解析手法である BDD (Binary Dicision Diagram 2 分決定木)[5] の適用について考える. Akers [2] や Bryant [3] により定式化され多くの実用的な論理関数を比較的コンパクトに表現できる BDD を利用すれば, 論理関数の操作を計算機上で効率的に行うことができる. 定性シミュレーションでは次の到達可能状態を求めるのに, 制約条件を次々に適用していき, すべての制約条件を満たす状態を求めていく. そこで, BDD を使えば, 各制約条件を論理積で結合することにより一度に制約条件を満たす状態を求めることができる.

本研究では, 定性シミュレーションにおいて初期状態から到達可能な定性状態集合を BDD を用いて求める手法について検討する. さらに, 初期状態から到達可能な定性状態までの経路に依存した性質が成立するかどうかを調べるために CTL を用いた記号モデル検査 [4] の手法を適用する.

第2章　BDD（Binary Dicision Diagram）

BDD はグラフによる論理関数の表現である [5]. 同じ論理関数を示す BDD は一つではない. その中でも, ある条件の下で求まる既約な順序付き BDD(2.2 参照) は, 論理関数に対する表現が一意に定まり標準形になるという特徴を持つ. またこの BDD は多くの論理関数が現実的な節点数で表現できたり, 論理関数に対する演算が表現のサイズに比例する時間で行えるなど著しい特徴を持つ. 本章では, BDD の定義を示し, 既約な順序付き BDD について述べる.

2.1　BDD の定義

BDD は, 変数でラベル付けされた非終端節点である変数節点と, 論理値でラベル付けされた終端節点である定数節点からなる. 変数の値が 0, 1 のときたどる枝はそれぞれ 0 枝, 1 枝と呼ばれる. 変数節点 v の 0 枝で指される節点を $\mathrm{low}(v)$, 1 枝で指される節点を $\mathrm{high}(v)$ と表わす. また, 変数節点 v にラベル付けされた論理値を $\mathrm{value}(v)$ で表わす. BDD の各節点は１つの論理関数を表わす. 節点 v の表わす論理関数 f_v は, 次のように定義される.

$$
\begin{aligned}
f_v &= \mathrm{value}(v)\ \text{(恒偽関数, 恒真関数)} && \cdots\ v\ \text{が定数節点のとき} \\
f_v &= \overline{\mathrm{var}(v)} \cdot f_{\mathrm{low}(v)} + \mathrm{var}(v) \cdot f_{\mathrm{high}(v)} && \cdots\ v\ \text{が変数節点のとき}
\end{aligned}
$$

言い換えれば, v を変数節点, $f|_{x=0}$, $f_{x=1}$ をそれぞれ f の $x=0, x=1$ に関するコファクタとすると,

$$
\begin{aligned}
f_{\mathrm{low}(v)} &= f_v|_{\mathrm{var}(v)=0} \\
f_{\mathrm{high}(v)} &= f_v|_{\mathrm{var}(v)=1}
\end{aligned}
$$

が成立する. BDD の根が表わす論理関数を, その BDD が表わす論理関数という.

2.2 既約な順序付き BDD

論理式 $x_1\overline{x_2} + x_3$ を表わす BDD は 図 2.1 や 図 2.2 など複数存在する. その中で重要なものが既約な順序付き BDD (ROBDD : reduced ordered BDD) である. ある変数の全順序が存在し, 根から定数節点に至るすべてのパスについて, 変数の順序がその全順序関係に矛盾しないとき, その BDD は順序付き BDD と呼ばれる. 図 2.1 の BDD について $x_1 < x_2 < x_3$ という変数の全順序を考えたとき, 根からどのようなパスを通っても変数の出現する順序はこの全順序に従っているので, この BDD は順序付き BDD (ordered BDD) であるといえる. 節点 v が $\mathrm{low}(v) = \mathrm{high}(v)$ を満たすとき, v は冗長な節点と呼ばれる. このような節点は BDD の表現する論理関数を変更することなく削除することができる. また, 節点 u と v が $\mathrm{low}(u) = \mathrm{high}(v)$ を満たすとき, u と v は等価であると言う. このとき u と v は同一の論理関数を表現するため, 等価な節点の一方を BDD の表現する論理関数を変更することなく削除することができる. 冗長な節点と等価な節点の削除を可能な限り行う操作を既約化という. 順序付き BDD に既約化を施して得られるものが, 既約な順序付き BDD である. また図 2.2 の BDD は, 図 2.1 の BDD 対して既約化を行って得られたものである.

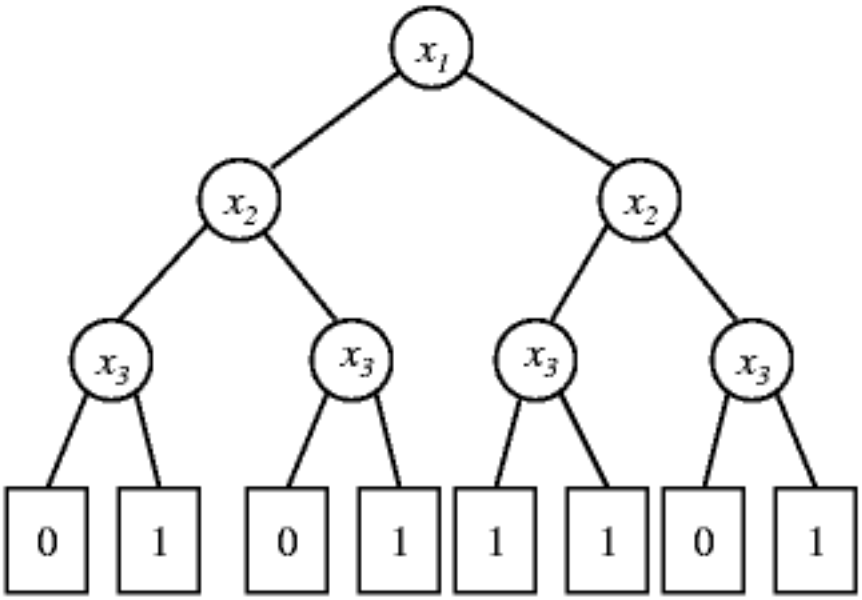

図 2.1: $x_1\overline{x_2} + x_3$ を表わす真理値表に等しい BDD

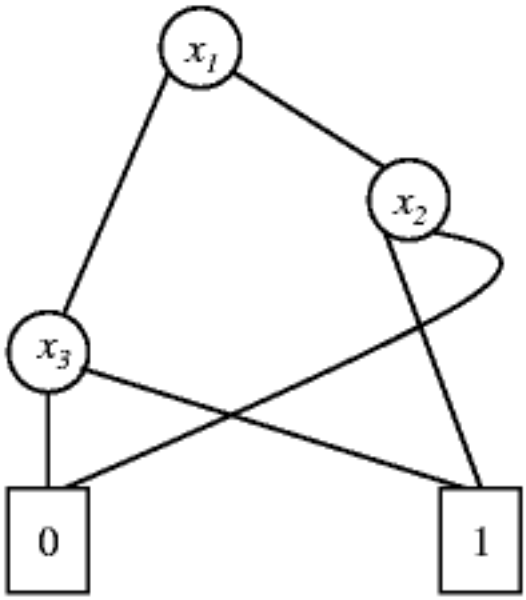

図 2.2: $x_1\overline{x_2} + x_3$ を表わす既約な順序付き BDD

第3章　定性シミュレーション

定性シミュレーションは, 到達可能な状態の候補を生成し, 定性微分方程式 (QDE: Qualitative Differential Equation) に整合する状態のみを残す操作を繰り返すことにより系の振る舞いを導出するものである. 本章では, 定性シミュレーションを行うために, シミュレーションを行う対象となるシステムをモデル化するための諸定義について述べ, それを用いて定性シミュレーションを行うシステムの構造を定義する定性微分方程式について述べ, 最後にその実行方法について述べる.

3.1　諸定義

定性シミュレーションを行うためには, 不完全情報を持つシステムをモデル化するために, 記号や符号を用いる. つまり, ある特別な実数に対応させた記号による順序関係や, 符号を用いる必要がある. そこで, 記号や符号に関する形式的な定義を与える.

定義 (reasonable 関数) $\mathfrak{R}^* := \mathfrak{R} \cup \{-\infty, \infty\}$. 区間 $[a,b] \subseteq \mathfrak{R}^*$ に対し, 関数 $f : [a,b] \to \mathfrak{R}^*$ は以下を満たすとき区間 $[a,b]$ 上で *reasonable* であるという.

1. f は $[a,b]$ で連続.
2. f は (a,b) で連続微分可能.
3. f は任意の有界な区間において有限個の臨界点 (critical point) を持つ.
4. $\mathfrak{R}^*$ 上で片側極限 $\lim_{x \to a^+} f'(t)$ および $\lim_{x \to b^-} f'(t)$ が存在する. $f'(a)$, $f'(b)$ をこれらの極限値とする.

定義 (定量空間 (Quantity Spaces)) 定量空間とは記号 (ランドマーク値 (landmark value) と呼ぶ) の全順序集合

$$l_1 < l_2 < \cdots < l_k$$

である. ランドマーク値には $\mathfrak{R}^*$ の値が対応する.

定義 (landmark time point)
時点 $t \in [a,b]$ は集合

$$\{t \in [a,b] \mid f(t) = x, x \text{ は } f \text{ のランドマーク値として表現されている}\}$$

の境界要素であるとき, ランドマーク時点 (landmark time point) であるという. 境界要素としているのは, $f(t)$ の値がある区間で一定の場合を扱うためである.

3.1.1 定性値 (Qualitative Values)

reasonable 関数 $f(t)$ の定量空間 $l_1 < l_2 < \cdots < l_k$ に関する定性値 $QV(f,t)$ は組 $\langle qmag, qval \rangle$ である. ここで,

$$qmag = \begin{cases} l_j & \text{if } f(t) = l_j \\ (l_j, l_{j+1}) & \text{if } f(t) \in (l_j, l_{j+1}) \end{cases}$$

$$qdir = \begin{cases} inc & \text{if } f'(t) > 0 \\ std & \text{if } f'(t) = 0 \\ dec & \text{if } f'(t) < 0 \end{cases}$$

- $a = t_0 < \cdots < t_n = b$ をランドマーク時点とする. $t_i < s < t < t_{i+1}$ ならば $QV(f,s) = QV(f,t)$ である.
- t_i, t_{i+1} を隣接するランドマーク時点とする. 区間 (t_i, t_{i+1}) に対する定性値を

  $$QV(f, t_i, t_{i+1}) \equiv QV(f,t) t \in (t_i, t_{i+1})$$

  により定義する.
- 関数 f の定性的挙動 (qualitative behavior) とは系列

  $$QV(f,t_0)QV(f,t_0,t_1)QV(f,t_1)\cdots QV(f,t_{n-1},t_n), QV(f,t_n)$$

  である.

複数の関数からなる系 $F = \{f_1, \cdots, f_m\}, f_i : [a,b] \to \mathfrak{R}^* (i = 1, \cdots, m)$ に対しては,

- 系 F のランドマーク時点は各 f_j のランドマーク時点の和集合とする. F のランドマーク時点がすべての $f_j, j = 1, \cdots, m$ のランドマーク時点とは限らない.
- 定性状態 (qualitative state) は各関数のランドマーク値のベクトルである.

$$
\begin{aligned}
QS(F,t_i) &= \langle QV(f_1,t_i),\cdots,QV(f_m,t_i)\rangle \\
QS(F,t_i,t_{i+1}) &= \langle QV(f_1,t_i,t_{i+1}),\cdots,QV(f_m,t_i,t_{i+1})\rangle
\end{aligned}
$$

系 F の定性的挙動とは系列

$$
QS(F,t_0)QS(F,t_0,t_1)QS(F,t_1)\cdots QS(F,t_{n-1},t_n),QS(F,t_n)
$$

である.
定義より, $QS(F,t_i)$ と $QS(F,t_i,t_{i+1})(QS(F,t_i,t_{i+1})$ と $QS(F,t_{i+1}))$ について, 少なくとも 1 つは $(QV(f_j,t_i)\neq QV(f_j,t_i))$ となる $f_j\in F$ が存在する.

3.2 定性微分方程式

定性微分方程式は 4 項組 $\langle V,Q,C,T\rangle$ から成り立っている.

- V は変数の集合であり, 各変数の値は時間の reasonable 関数である.
- Q は定量空間 (quantity space) の集合で, 各々は V の各変数に対応している.
- C は V の変数上に定義された制約の集合であり, V の各変数は C の制約に必ず出現しなければならない.
- T は遷移 (transition) の集合であり, その QDE が適用可能な領域の境界を定義する.

3.2.1 符合領域 (Domain of Signs)

$S=\{+,-0\}$ を符合領域という. S に無視 (ignorance) を表わす記号 ? を加えた $S'=\{+,-,0,?\}$ を拡張符合領域という.
$x\in\mathfrak{R}^*$ に対し以下を定義する.

- $$
sign(x)=\begin{cases}[+] & (x>0)\\ [-] & (x<0)\\ [0] & (x=0)\end{cases}
$$
- $[x]_{x_0}=sign(x-x_0)$. $[x]_0$ は単に $[x]$ と書く.
- $[\dot{x}]=[dx/dt]=sign(dx/dt)$.

表 3.1: Qualitative addition table.

+	[+]	[0]	[−]
[+]	[+]	[+]	[+]/[0]/[−]
[0]	[+]	[0]	[−]
[−]	[+]/[0]/[−]	[−]	[−]

表 3.2: Qualitative multiplication table.

×	[+]	[0]	[−]
[+]	[+]	[0]	[−]
[0]	[0]	[0]	[0]
[−]	[−]	[0]	[+]

- $$[x]_\infty = \begin{cases} [+] & (x = +\infty) \\ [-] & (x = -\infty) \\ [0] & (x \text{ is finite}) \end{cases} \tag{3.1}$$

定義 (定性的な加法 · 乗法) $\mathcal{S}'$ 上の関係として, 加法と乗法を以下のように定義する.

$$+ : \mathcal{S} \times \mathcal{S} \times \mathcal{S} \to \{T, F\}$$
$$\times : \mathcal{S} \times \mathcal{S} \times \mathcal{S} \to \{T, F\}$$

3.2.2 定性的制約

変数間には, 各時刻において満たすべき定性的な制約がある. 各制約 A, MULT, MINUS, d/dt, CONSTANT は数学的な関係と似ており以下のように定義される.

- $(ADD\ x\ y\ z) \equiv x(t) + y(t) = z(t)$.
- $(MULT\ x\ y\ z) \equiv x(t) \cdot y(t) = z(t)$.
- $(MINUS\ x\ y) \equiv y(t) = -x(t)$.
- $(d/dt\ x\ y) \equiv \frac{d}{dt}x(t) = y(t)$.

- $(CONSTANT\ x) \equiv \frac{d}{dt}x(t) = 0.$

他に, 関数についての定性的な関係を表わす制約 M+, M- がある. その変数を関係付ける関数 M+ は, 単調増加関数のクラス M+ に由来する.

- $(M^+\ x\ y) \equiv y(t) = f(x(t)), f \in M^+.$
- $(M^-\ x\ y) \equiv y(t) = -f(x(t)), f \in M^-.$

ここで, M^+ は, $f'(t) > 0, t \in (a, b)$ であるような reasonable 関数 $f : [a, b] \to \Re^*$ 全体の集合 (厳密に単調増加).

3.2.3 対応値 (Corresponding Values)

各制約において同時に取ることのできる qmag の組のことであり, シミュレーションの履歴を制約ごとにとったものである. 例えば, 変数 A, B, C に $A + B = C$ の関係があり, ある時刻 t において Ac, Bc, Cc の値を取っていたとする. このとき, t より後の t' において $A = A1$, $B = B1$, $C = C1$ の値をとり, $A1 > Ac$, $B1 > Bc$ であれば, $C1 > Cc$ でなくてはならない. このように, シミュレーションの過程において, $A = Ac$, $B = Bc$, $C = Cc$ の組合わせが $A + B = C$ の制約を満足したことを記憶しておき, 状態遷移によって生成される可能な遷移の絞り込みに利用される.

3.3 定性シミュレーションシステムの実現

定性シミュレーションは, 以下の処理を繰り返すことにより実現される.

1. 初期条件 (定性的状態に関する部分記述) を入力する
2. 初期条件から制約を満たす初期状態を求める
3. 初期状態から P 遷移 (表 3.3) を参照し次状態の候補を求め, 制約を満たす次状態 (次のステップにおいて現在の状態となる) を求める
4. 現在の状態から I 遷移 (表 3.4) を参照し次状態の候補を求め, 制約を満たす次状態 (次のステップにおいて現在の状態となる) を求める
5. 過去に到達した状態と同じ状態にしか到達していなければシミュレーションは終了となる. 新しい状態に到達しているならば P 遷移を参照する段階からシミュレーションを繰り返す

表 3.5 に JQSIM によるバスタブモデル (図 5.2) における振る舞いを示す. これは状態が最終状態に至るまでにどのように変化するかを表わしており, この場合, 最終状態に至るまでに 3 通りの振る舞いの仕方があることを表わしている.

表 3.3: P-Successors: point to interval.

$QV(v, t_i)$	$\Leftrightarrow$	$QV(v, t_i, t_{i+1})$
$< l_j, std >$		$< l_j, std >$
$< l_j, std >$		$< (l_j, l_{j+1}), inc >$
$< l_j, std >$		$< (l_{j-1}, l_j), dec >$
$< l_j, inc >$		$< (l_j, l_{j+1}), inc >$
$< l_j, dec >$		$< (l_{j-1}, l_j), dec >$
$< (l_j, l_{j+1}), inc >$		$< (l_j, l_{j+1}), inc >$
$< (l_j, l_{j+1}), dec >$		$< (l_j, l_{j+1}), dec >$
$< (l_j, l_{j+1}), std >$		$< (l_j, l_{j+1}), std >$
$< (l_j, l_{j+1}), std >$		$< (l_j, l_{j+1}), inc >$
$< (l_j, l_{j+1}), std >$		$< (l_j, l_{j+1}), dec >$

表 3.4: I-Successors: interval to point.

$QV(v, t_i, t_{i+1})$	$\Leftrightarrow$	$QV(v, t_{i+1})$
$< l_j, std >$		$< l_j, std >$
$< (l_j, l_{j+1}), inc >$		$< l_{j+1}, std >$
$< (l_j, l_{j+1}), inc >$		$< l_{j+1}, inc >$
$< (l_j, l_{j+1}), inc >$		$< (l_j, l_{j+1}), inc >$
$< (l_j, l_{j+1}), inc >$		$< (l_j, l_{j+1}), std >$
$< (l_j, l_{j+1}), dec >$		$< l_j, std >$
$< (l_j, l_{j+1}), dec >$		$< l_j, dec >$
$< (l_j, l_{j+1}), dec >$		$< (l_j, l_{j+1}), dec >$
$< (l_j, l_{j+1}), dec >$		$< (l_j, l_{j+1}), std >$
$< (l_j, l_{j+1}), std >$		$< (l_j, l_{j+1}), std >$

表 3.5: JQSIM が出力するバスタブ 1 連モデルにおける振る舞い

Behavior1 AMOUNT1	OUTFLOW1	INFLOW1	NETFLOW1
$< 0, inc >$	$< 0, inc >$	$< if^*, std >$	$< (0, \infty), dec >$
$< (0, full), inc >$	$< (0, \infty), inc >$	$< if^*, std >$	$< (0, \infty), dec >$
$< full, std >$	$< (0, \infty), std >$	$< if^*, std >$	$< 0, std >$
Behavior2 AMOUNT1	OUTFLOW1	INFLOW1	NETFLOW1
$< 0, inc >$	$< 0, inc >$	$< if^*, std >$	$< (0, \infty), dec >$
$< (0, full), inc >$	$< (0, \infty), inc >$	$< if^*, std >$	$< (0, \infty), dec >$
$< (0, full), std >$	$< (0, \infty), std >$	$< if^*, std >$	$< 0, std >$
Behavior3 AMOUNT1	OUTFLOW1	INFLOW1	NETFLOW1
$< 0, inc >$	$< 0, inc >$	$< if^*, std >$	$< (0, \infty), dec >$
$< (0, full), inc >$	$< (0, \infty), inc >$	$< if^*, std >$	$< (0, \infty), dec >$
$< full, inc >$	$< (0, \infty), inc >$	$< if^*, std >$	$< (0, \infty), dec >$
$< (full, \infty), inc >$	$< (0, \infty), inc >$	$< if^*, std >$	$< (0, \infty), dec >$
$< (full, \infty), std >$	$< (0, \infty), std >$	$< if^*, std >$	$< 0, std >$

第4章　モデル検査

CTL(Computational Tree logic) は時相論理の一種である. 時相論理とは命題の真偽値が時間的に変化する論理体系であり, 有限状態機械の満たすべき仕様を記述するのに適している. 定性シミュレーションは初期状態から遷移可能な状態へ次々と状態が遷移していく点において, 有限状態機械と見なすことができる. 本章では, そのような定性シミュレーションにおいて時相論理式で記述されるような性質の検証を行うため, CTL を用いた検証手法である CTL モデル検査について述べる [7].

4.1　Kripke 構造

CTL モデル検査 においては有限状態機械は Kripke 構造と呼ばれるラベル付けされた状態遷移図でモデル化される.

定義 ($Kripke$ 構造) AP を原始命題の集合とする. AP 上の $Kripke$ 構造 M は 4 項組 $M = (S, S_0, R, L)$ で表わされる.

- S は状態の有限集合.
- $S_0 \subseteq S$ は初期状態の集合.
- $R \subseteq S \times S$ は節点集合上の 2 項関係で, 状態間の遷移関係であり, すなわち状態 $s \in S$ ごとに $R(s, s')$ となる状態 $s' \in S$ が存在する. また, 任意の状態は少なくとも１つ以上の次状態を持つものとする.
- $L : S \to 2^{AP}$ は各状態で真となる原始命題の集合を持つその状態にラベル付けした関数である.

初期状態の集合 S_0 については考えないこともある. そのような場合, 定義から初期状態の集合を省く. 状態 s から 構造 M の $path$ は $s = s_0$ かつ $R(s_i, s_{i+1})$ が任意の $i \geq 0$ で成り立つような状態の無限列 $\pi = s_0 s_1 s_2 \cdots$ である. g が CTL の状態論理式であるとする. もし g が M の状態 s_0 で成り立つならば, このとき $M, s_0 \models g$ と表記する.

4.1.1 CTL 論理式

原始命題 AP の集合からなる CTL 論理式は以下のように定義される.

定義 (状態論理式, パス論理式)
状態論理式

- 原始命題 $p \in AP$ は状態論理式である.
- f_1, f_2 が状態論理式であるならば, $\neg f_1$, $f_1 \vee f_2$, $f_1 \wedge f_2$, そして $f_1 \rightarrow f_2$ も状態論理式である.
- もし g が パス論理式であるならば, Eg と Ag は状態論理式である.

パス論理式

- もし f_1 と f_2 が状態論理式であるならば, Xf_1, Ff_1, Gf_1, そして f_1Uf_2 はパス論理式である.

ここで,

- Eg は g があるパスで成り立つことを表わす.
- Ag は g がすべてのパスで成り立つことを表わす.
- Xf は f が次の状態で成り立つことを表わす .
- Ff は f がパス上のある状態で成り立つことを表わす .
- Gf は f がパス上の全ての状態で成り立つことを表わす.
- f_1Uf_2 は f_2 が真になるまで f_1 が真であることを表わす.

これらの時相演算子を概念的に表わしたものが図 4.1-図 4.6 である. 図 4.1 は AFf を表わし, すべてのパスにおいて将来 f が成り立つことを表わす.
EFf (図 4.2) は将来 f が成り立つパスが存在することを表わす.
AGf (図 4.3) はすべてのパスの全ての節点において f が成り立つことを表わす.
$E[fUg]$ (図 4.6) は将来 g が成り立つまで常に f が成り立っているパスが存在することを表わす.
$A[fUg]$ (図 4.5) はどのパスにおいても将来 g が成り立つまでに常に f が成り立っていることを表わす.

4.2 CTL モデル検査のアルゴリズム

CTL のモデル検査は, CTL 論理式 φ が成り立つ状態集合を求め, その中に初期状態 s_0 が含まれるどうかを調べることで行うことができる. つまり, CTL 論理式が求まれば, CTL 論理式と初期状態 s_0 との論理積を求め, 論理式となれば成り立ち, $False$ となれば成り立たないということになり, モデル検査を行うことができる. そこで, 基礎となる CTL 論理式 EXf, $E[f_1Uf_2]$, EGf が成り立つ状態集合を求める計算が, 1 ステップの状態遷移関数の最小, 最大不動点として特徴付けられることによって示す.

4.2.1 不動点表現

$M = (S, R, L)$ は有限 Kripke 構造であるとする. 各 CTL 論理式 f は集合 $\{s| < M, s > \models f\} \in S$ によって特定される.

各 CTL 論理式をプログラム上で実装する前に, 本プログラムで必要となる bdd ライブラリ関数について説明しておく. 本プログラムは, 1989 年に, Synopsys Inc, Mountain View, CA. と Carnegie Mellon 大学で作られた bdd version 3.34 を使用する. bdd_dagptr_and(bddm, A, B) は, 論理式 A と論理式 B の論理積を表わす. 関数 bddSHIFT(f) は, 到達可能状態を表わす論理関数 f を現在の状態を表わす論理関数に置き換える. 関数 bddEXIST(f) は現在の状態を表わす bdd 変数と到達可能状態を表わす bdd 変数によって構成された論理関数 f から, 論理関数 f を満たす現在の状態を表わす論理関数を出力する. bdd_dagptr_free(bddm, h1) は論理関数 h1 で使用したメモリ領域を解放する関数である.

- EXf
  EXf は $EXf := \exists x'[f(x') \wedge R(x, x')]$ と定義され, f が成り立つ状態集合 X を求め, $\exists x' \in X$, $(x, x') \in R$ であるような状態 x の集合 X' を求める. X' が EXf が成り立つ状態集合となる.

  プログラム上では, 以下のように実装される.
  4 行目で f は現在の状態を表わす BDD 変数で表わされているため, f を次の状態を表わす BDD 変数で置き換えている. 5 行目で, 次の状態を表わす BDD 変数で定義された h1 と 1 ステップの状態遷移関数を表わす NEXT との論理積 h2 を取り, 6 行目において, h2 の論理式を成り立たたせるような現在の状態を表わす BDD 変数を求め, それが次の状態で f が成り立つような, つまり EXf が成立する状態の集合を表わす.

```
1:  bdd_dagptr bddEX(bdd_dagptr f)
2:      {
3:      bdd_dagptr h1, h2, h3;
```

```
4:      h1 = bddSHIFT(f);
5:      h2 = bdd_dagptr_and(bddm, h1, NEXT);
6:      h3 = bddEXIST(h2);
7:      bdd_dagptr_free(bddm, h1);
8:      bdd_dagptr_free(bddm, h2);
9:      return(h3);
10: }
```

- $E[fUg]$
  $E[fUg]$ は, 論理式 $Z = g \vee (f \wedge g)$ である Z の最小不動点として定義される. $\tau : P(S) \to P(S)$ は $\tau(Z) = g \vee (f \wedge EXZ)$ によって定義される操作とする. そのとき,

$$\begin{aligned} E[f_1 U f_2] &= \cup_i \tau^i(False) \\ &= \emptyset \cup \tau(False) \cup \tau^2(False) \cup \tau^3(False) \cup \cdots . \end{aligned}$$

  となる. これは g がいつか成り立つまで f が常に成り立ち続けることを表わす. プログラム上は, E[f U g] を求めており, 4 行目で False を表わす bddZERO を代入し, 7 行目までが前処理となる. 次の状態が False となるものは存在しないため 7 行目の z2 には 将来いつか成り立つ g が代入されることになる. 11 行目からがプログラム本体となるが, 14 行目において将来 z1 となるもの (この場合は, g) を求め h1 に代入する. 15 行目で次の状態が z1 となるものの中で f が成り立つものを求めている. これにより, g が成り立つまで f が成り立ち続けることになる. z2 の求まった状態が, 1 ステップ前に求まった状態と同じものとなったとき while 文の実行は終了し, その論理式が E[f U g] を満たす論理式となる.

```
1:  bdd_dagptr bddEU(bdd_dagptr f, bdd_dagptr g)
2:  {
3:      bdd_dagptr z1, z2, h1, h2;
4:      z1 = bddZERO;
5:      h1 = bddEX(z1);
6:      h2 = bdd_dagptr_and(bddm, f, h1);
7:      z2 = bdd_dagptr_or(bddm, g, h2);
8:      bdd_dagptr_free(bddm, h1);
9:      bdd_dagptr_free(bddm, h2);
10:
11:     while(bdd_dagptr_equal(z1, z2) == 0){
12:         bdd_dagptr_free(bddm, z1);
13:         z1 = z2;
```

```
14:       h1 = bddEX(z1);
15:       h2 = bdd_dagptr_and(bddm, f, h1);
16:       z2 = bdd_dagptr_or(bddm, g, h2);
17:       bdd_dagptr_free(bddm, h1);
18:       bdd_dagptr_free(bddm, h2);
19:    }
20:    return(z2);
21: }
```

- EGf

  EGf は 論理式 $Z = f \wedge EXZ$ である Z の最大不動点として定義される. $\xi : P(S) \rightarrow P(S)$ は $\xi(Z) = f \wedge EXZ$ によって定義される操作であるとする. そのとき,

$$\begin{aligned} EGf &= \cap_i \xi^i(True) \\ &= S \cap \xi(True) \cap \xi^2(True) \cap \xi^3(True) \cap \cdots . \end{aligned}$$

  となる. これは f があるパス上で常に成り立っていることを表わしている. プログラム上は, 4 行目で True を表わす bddONE を g1 に代入し, 5 行目で次の状態が True となるものを求め, 6 行目でその状態において g1 が成り立つものを求める. これが前処理となる. 9 行目より while 文に入り, 11 行目で EX(g1) を満たす論理式を求め, その状態において g1 が成り立つ状態を 12 行目で求めている. 9 行目の while 文の条件式で求まった論理式 g2 と その前のステップで求まった論理式 g1 が等しいかどうか比較し同じ論理式になるまでこの操作を繰り返す.

```
1:  bdd_dagptr bddEG(bdd_dagptr f)
2:  {
3:     bdd_dagptr g1, g2, h1, h2, h3;
4:     g1 = bddONE;
5:     h1 = bddEX(g1);
6:     g2 = bdd_dagptr_and(bddm, g1, h1);
7:     bdd_dagptr_free(bddm, h1);
8:
9:     while(bdd_dagptr_equal(g1, g2) == 0){
10:       g1 = g2;
11:       h1 = bddEX(g1);
12:       g2 = bdd_dagptr_and(bddm, g1, h1);
13:       bdd_dagptr_free(bddm, h1);
14:    }
15:    return(g2);
```

16: }

また, AXf, EFf, AGf, AFf については上記の EX, $E[f_1Uf_2]$, EGf の操作を使って

- $AXf \equiv \neg EX\neg f$
- $EFf \equiv E[TrueUf]$
- $AGf \equiv \neg EF\neg f$
- $AFf \equiv \neg EG\neg f$

によって定義される.

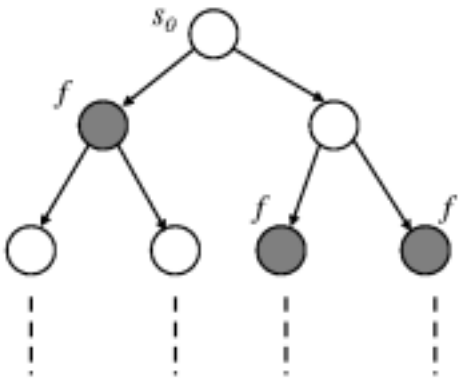

図 4.1: $M \models AFf$

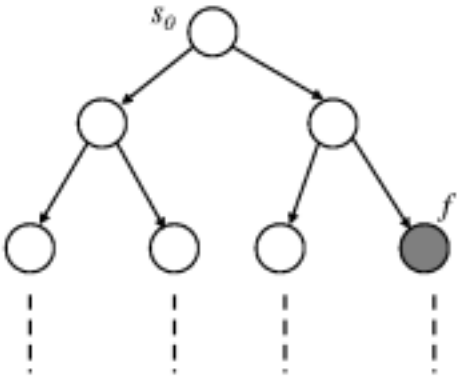

図 4.2: $M \models EFf$

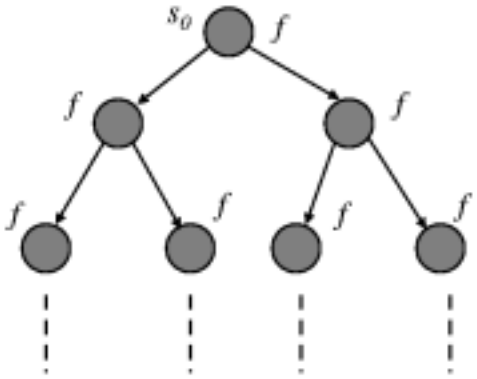

図 4.3: $M \models AGf$

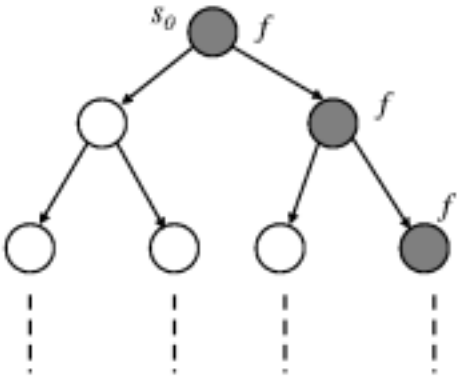

図 4.4: $M \models EGf$

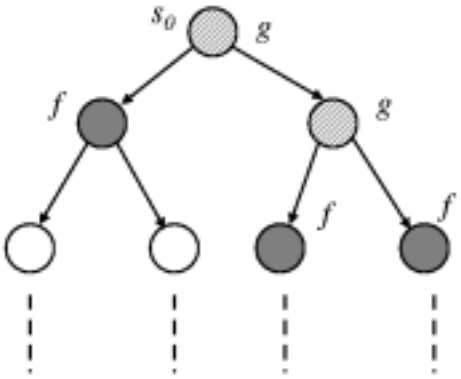

図 4.5: $M \models A[fUg]$

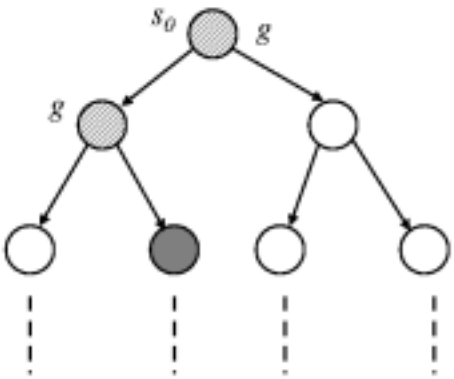

図 4.6: $M \models E[fUg]$

第5章　BDD を用いた定性シミュレーション

2 章で, BDD が効率的に論理関数を表現することについて述べた. 本章では BDD を用いて定性シミュレーションの効率化を図る方法について述べる. つまり, 定性シミュレーションにおいて到達可能な状態を求める方法を、どのように論理演算に置き換えるのか, その方法について述べる. 本章では, BDD を用いた定性シミュレーションの実装方法について以下の順に説明していく.

5.1　実装方法

5.1.1　状態の論理関数表現

変数は, 0 からはじまる整数に対応付けて表わす. プログラム上では変数名をマクロとして各整数に対応づけている. 各変数の定性値を論理関数であらわすために, 各変数ごとに論理変数をランドマーク値の数 + (ランドマーク値の数 - 1) + 4 個用意する. (ランドマーク値の数 + ランドマーク値の数 - 1) で, qmag の位置を表わし, 残りの 4 個の論理変数で qval の位置を表わす. 変数の定性値は, かく変数ごとに定性値に 0 から数字を割り振ることで表わし, 変数の qmag を 4 の商で, qval を 4 の剰余で表わす. 定性値の qmag の位置は, 定性値に割り振られた数に対する 4 の商で求まり, qval は, 4 の剰余で求まる. なお, 剰余が 0 には符号 [+] が, 1 には [−], 2 には [0], 3 には [?] が対応づけられている. 例えば、ランドマーク値の数が３つで、定性値の値が１２であるならば、この qmag の位置は, 12 を 4 で割った商が 3 であるので, 0 から数えて４番目の位置の qmag を表わし、１２を４で割った剰余が０であるので、qval は、[+] を表わす。なお, 到達可能な状態を表わすための変数として, 上記の方法と同じ方法で別にもう１つの変数を表現する. また各変数のランドマーク値について, 0 , $-\infty$, $+\infty$ の位置が qmag のどの整数に対応するのか, つまり, 定性値を 4 で割った商のどの値に対応するのか定義しておく. また, すべての変数の数も定義する.

5.1.2 1 ステップ遷移関数の表わし方

各変数ごとに 現在の状態から次の状態への 1 ステップの状態遷移関数を作る. 時点から時区間への状態遷移関数 (P-successor) 表 3.3 と, 時区間から時点の状態遷移関数 (I-successor) 表 3.4 を作る. これは各状態ごとに現在の定性値から次に遷移可能な定性値を論理積をとることで表わす. ある変数について, 遷移可能な場合を定義したら, それらの論理和を取り, ある変数についての 1 ステップの状態遷移関数とする. すべての場合について 1 ステップの状態遷移関数を定義したら, それらについて論理積を取る. それが 1 ステップ状態遷移関数となる.

5.1.3 制約条件の表わし方

制約条件を到達可能な状態を表わす BDD 変数について定義する. 制約条件に表れる状態について, 制約を満たすすべての定性値の関係について論理積取る. それらについて論理和を取り, それがその制約についての制約条件となる.

5.1.4 QDE が変わる場合

系が遷移する場合, つまり, QDE が変わる場合, QDE を区別するための BDD 変数を定義する. 到達可能状態を調べるため状態については同じ BDD 変数を使い, QDE を表わす BDD 変数と 論理積を取ることで系の違いを表現する. また, 系が変化する場合の条件も一緒に定義する.

5.1.5 入力状態

初期状態として, 各状態が取りうる定性値の qmag, qdir をそれぞれ定義する. もし qmag, qdir の値が未知のものがあれば未知のものとして定義する. プログラム上は, qmag の位置が決まっていれば, ランドマーク値の位置を指定し, qdir は [+] ならば 0 を, [−] ならば 1 を, [0] ならば 2 を指定する. もし qmag も qdir も未知の値であれば -1 を指定する.

5.1.6 出力状態

到達可能状態を表わす BDD を出力する.

5.1.7　シミュレーションの仕方

遷移関数, 制約条件, QDE が遷移する場合はその時の条件すべてについて論理積を取る. それを 完全な 1 ステップ遷移関数と呼ぶ. 初期条件と制約条件について論理積を取り, それが初期条件を満たす初期状態の集合となる. 求まった初期状態と完全な 1 ステップ遷移関数の論理積を取り, 1 ステップ後に到達可能な状態の集合を表わす BDD を求める. 以上の処理を繰り返し, 現在の状態を表わす BDD と 1 ステップ後に到達可能な状態の集合を表わす BDD が同じものであればシミュレーション終了となる.

$$
\begin{aligned}
R_0 \quad &= \quad \{x_0\} \\
R_1 \quad &= \quad R_0 \cup \{\ \mathrm{x} \text{ — } \exists\, \mathrm{x'} : \mathrm{x'} \in R_0 \wedge \delta(x', x)\} \\
&\cdot \\
&\cdot \\
&\cdot \\
R_i \quad &= \quad R_{i-1} \cup \{\mathrm{x} \text{ — } \exists\, \mathrm{x'} : \in R_{i-1} \wedge \delta(x', x)\ \} \\
\text{until} \quad R_i \quad &= R_{i-1}
\end{aligned}
$$

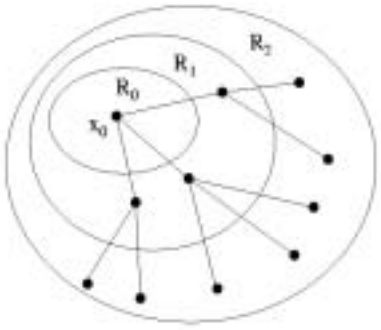

図 5.1: 到達状態計算

5.2 例題：バスタブモデル

本シミュレーションで使用するモデルであるバスタブモデル (図 5.2) について説明する. ある大きさのバスタブに水を注いでいく. バスタブの底には, 排水口がついている. このモデルは, バスタブに貯まる水の量を表わす AMOUNT1, 排水口から流出する流量を表わす OUTFLOW1, バスタブ内部の流量を表わす NETFLOW1, バスタブに注がれる水の量を表わす INFLOW1 の 4 つの変数によって表わされる. 各変数間には, 満たすべき物理的な制約条件が存在する. バスタブの水の量 AMOUNT1 が増えれば, 流出する流量 OUTFLOW1 も大きくなる. また, バスタブ内部の流量は, 流入する流量と, 流出する流量の差で表わされる. また, バスタブ内部の水の量の変化の方向は, バスタブ内部の流量の大きさと対応している. このシミュレーションは, このバスタブに水を注いでいく過程でどのように状態が変化してか, 言い換えれば, どのような状態に到達する可能性があるのかを見るものである. バスタブモデルは, 変数の数が小さいため, どのような到達可能状態が存在するのか手作業でも求めることが可能である. しかし, バスタブを縦に連結したモデルである (図 5.3) のような, バスタブ 3 連モデル (図 5.3) について考えてみるとどうであろうか. この場合, 考えるべき変数の数は 10 変数ある. バスタブが 1 連のときは手作業でも到達可能状態を求めることが可能であるが, バスタブが 3 連にもなると, 比較する変数の数も大きくなり, いくら変数間の制約条件が簡単なものであっても到達可能状態をすべて求めるのは困難な作業となる. そこで, バスタブを連結するごとに比較する変数の組み合わせが大きくなっていくこのモデルを用いて, 定性シミュレーションの効率化が図られているのかシミュレーションによって確かめていく.

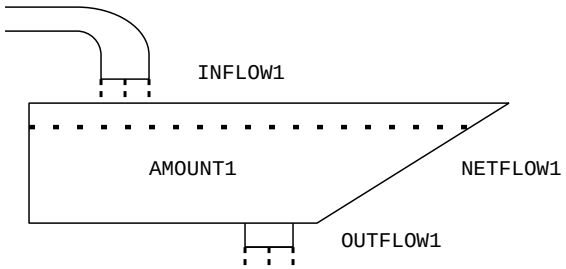

図 5.2: バスタブモデル

バスタブ 1 連モデルをモデル化してみる. このモデルは, 4 つの変数 AMOUNT1, INFLOW1, OUTFLOW1, NETFLOW1 からなる. 変数 AMOUNT1 はバスタブ中の水の量を表わす. 変数 INFLOW1 はバスタブに流入する水の流量を表わす. 変数 OUTFLOW1 は流出する水の流量を表わす. 変数 NETFLOW1 はバスタブの中で変化する水の流量を表わす. 各変数は, 次のランドマーク値をとるものとする.

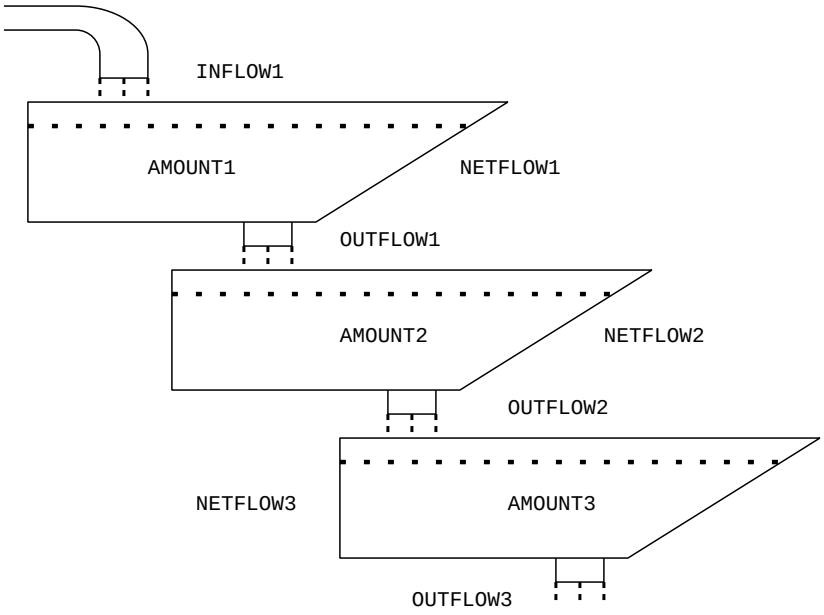

図 5.3: バスタブ 3 連モデル

AMOUNT1 : $0 \cdots$ full $\cdots \infty$
OUTFLOW1 : $0 \cdots \infty$
INFLOW1 : $0 \cdots$ if* $\cdots \infty$
NETFLOW1 : $-\infty \cdots 0 \cdots \infty$

ここで if* はある一定の速度を表わす.

変数間には満たすべき物理的な制約条件,
INFLOW1 は一定
OUTFLOW1 = M+(AMOUNT1)
INFLOW1 + NETFLOW1 = OUTFLOW1
d(AMOUNT1)dt = NETFLOW1
が存在する.

バスタブ 2 連目以降において流入する水の量としては 1 段上のバスタブの OUTFLOW1 が対応する. 従って, バスタブ 1 連モデルでは変数の数は 4 個であるが, 以後バスタブが１つ連結される度に, 変数が 3 つづつ増えていく. バスタブ 10 連モデルでは扱う変数の数は 31 個となる.

第6章　シミュレーション結果

6.1　BDD を用いたシミュレーションの正しさ

本シミュレーションが正確に実行されているのか確認するためにいくつかのモデルについてシミュレーションを行った. バスタブ 1 連モデル (図 5.2) は, 変数の数が 4 つしかなく, 手作業でも到達可能状態を求めることが可能である. そこで, 手作業により求まった到達可能状態を表わす BDD と, シミュレーションにより求まった到達可能状態を表わす BDD が等しいものとなるか比較したところ同じものとなった. 同様の作業を Ball-Bouncing モデル, Thermostat モデルについても行ったところこれも等しい結果となった. 3 つのモデルに対して正確に実行されており, 本シミュレーションは正確に実行されているといえる.

6.2　JQSIM でのシミュレーション結果

JQSIM は Java で実装された定性シミュレーションを行うプログラムであり, 従来の定性シミュレーションのアルゴリズムに従って BDD を用いずに作られており, 系の定性挙動を出力する. 本シミュレーションを, 1 個の 1.062GHz UltraSPARK IIIi プロセッサ, OS は Soraris8, メモリ 512 MB の環境で行った.

バスタブ 1 連から 10 連モデルについて, JQSIM と BDD を用いた定性シミュレータで到達可能状態, 及び到達可能状態を表わす BDD を求めるのにかかる時間を比較した (表 6.1, 表 6.2).

表 6.1: JQSIM での到達可能状態を求めるのにかかる実行時間

JQSIM	実行時間	到達可能状態数	振る舞いの数
バスタブ 1 連	1.25 秒	9	3
バスタブ 2 連	2.53 秒	379	178
バスタブ 3 連	19 分 37.70 秒	7789	5857
バスタブ 4 連	実行不能	実行不能	実行不能

バスタブ 4 連モデルに関しては, 実行してから 129 時間 12 分 22.32 秒後に 84008 の到達可能状態を求めてメモリ不足により実行不能となった.

6.3 BDD を用いた定性シミュレーションの結果

表 6.2: BDD を用いた定性シミュレータでの到達可能状態を表わす BDD を求めるのにかかる実行時間

JQSIM	実行時間	BDD size
バスタブ 1 連	0.0200	74
バスタブ 2 連	0.1200	191
バスタブ 3 連	0.2900	278
バスタブ 4 連	0.4900	603
バスタブ 5 連	0.8100	793
バスタブ 6 連	1.2300	983
バスタブ 7 連	1.6600	1173
バスタブ 8 連	2.2200	1363
バスタブ 9 連	2.7600	1553
バスタブ 10 連	4.2900	1743

6.4 CTL モデル検査を用いた検証

バスタブ 1 連モデルについて, 以下の検証を行った.

- 将来, 初期状態から, 次の状態が (OUTFLOW1 = ((0, ∞), [+])) であるパスが存在する.
  初期状態 $\wedge$ EX(OUTFLOW1 = ((0, ∞))
  これは, 論理関数を表わす BDD が出力し, 成立することを表わす.

- 初期状態から、将来, AMOUNT1 = (full, +) に到達するパスが存在する
  初期状態 $\wedge$ EF(AMOUNT1 = (full, +))
  これは, 論理関数を表わす BDD が出力し, 成立することを表わす.

- 初期状態であるならば, すべてのパスで AMOUNT1 = (full, +) または (full, -) または (full, 0) に到達する
  初期状態 $\wedge$ AF(AMOUNT1 = (full, +) または (full, -) または (full, 0))
  この結果は, 恒偽関数となる. これは初期状態から常には, AMOUNT1 = (full, +) または (full, -) または (full, 0) に到達しないということを表わす.

- 初期状態から、OUTFLOW1 = ((0, ∞, +) である状態へ到達できるパスが常に存在する.
  初期状態 $\wedge$ AF(OUTFLOW1 = ((0, ∞), +))
  これは, 論理関数を表わす BDD が出力し, 成立することを表わす.

以上の検証に関して, 実際に使用したプログラムと, その実際の実行結果を示す.

```
  printf("*************************************************************\n");

  printf("EX\n");
  printf("初期状態から、次の状態で OUTFLOW1 が ((0, inf), +) である状態へ到達
できるパスが存在する. \n");
  printf("初期状態 ^ EX(OUTFLOW1 = ((0, inf), +)) \n\n");

  tameshi_ctl = bdd_dagptr_and(bddm, QDE[0], bdd_hennsuu[OUTFLOW1][4]);
  tameshi_ctl1 = bddEX(tameshi_ctl);
  tameshi_ctl1 = bdd_dagptr_and(bddm, gennzai, tameshi_ctl1);

  if(tameshi_ctl1 == bddONE)
    printf("tameshi_ctl1 = bddONE\n");
  else if(tameshi_ctl1 == bddZERO)
    printf("tameshi_ctl1 = bddZERO\n");
```

```
  else
    printf("tameshi_ctl1 != bddZERO\n");

  printf("************************************************************\n");

  printf("EF\n");
  printf("初期状態が、AMOUNT = (full, +) である状態へ到達できるパスが存在する. \n ");
  printf("初期状態 ^ EF(AMOUNT = (full, +))\n\n");

  tameshi_ctl = bdd_dagptr_and(bddm, QDE[0], bdd_hennsuu[AMOUNT1][8]);
  tameshi_ctl1 = bddEF(tameshi_ctl);
  tameshi_ctl1 = bdd_dagptr_and(bddm, gennzai, tameshi_ctl1);

  if(tameshi_ctl1 == bddONE)
    printf("tameshi_ctl1 = bddONE\n");
  else if(tameshi_ctl1 == bddZERO)
    printf("tameshi_ctl1 = bddZERO\n");
  else
    printf("tameshi_ctl1 != bddZERO\n");

  printf("************************************************************\n");

  printf("AF\n");
  printf("初期状態であるならば、常に amount = (full, +) or (full, -) or (full, 0) で
ある状態に到達するパスが存在する. \n ");
  printf("初期状態 ^ AF(AMOUNT = (full, + or - or 0))\n\n");

  tameshi_ctl = bdd_dagptr_and(bddm, QDE[0], bdd_hennsuu[AMOUNT1][8]);
  tameshi_ctl2 = bdd_dagptr_and(bddm, QDE[0], bdd_hennsuu[AMOUNT1][9]);
  tameshi_ctl3 = bdd_dagptr_and(bddm, QDE[0], bdd_hennsuu[AMOUNT1][10]);
  tameshi_ctl = bdd_dagptr_or(bddm, tameshi_ctl, tameshi_ctl2);
  tameshi_ctl = bdd_dagptr_or(bddm, tameshi_ctl, tameshi_ctl3);
  tameshi_ctl1 = bddAF(tameshi_ctl);
  tameshi_ctl1 = bdd_dagptr_and(bddm, gennzai, tameshi_ctl1);

  if(tameshi_ctl1 == bddONE)
    printf("tameshi_ctl1 = bddONE\n");
  else if(tameshi_ctl1 == bddZERO)
```

```
    printf("tameshi_ctl1 = bddZERO\n");
  else
    printf("tameshi_ctl1 != bddZERO\n");

  printf("*****************************************************************\n");

  printf("AF\n");
  printf("初期状態から、outflow1 が ((0, inf), +) である状態へ到達できるパス
が常に存在する. \n ");
  tameshi_ctl = bdd_dagptr_and(bddm, QDE[0], bdd_hennsuu[OUTFLOW1][4]);
  tameshi_ctl1 = bddAF(tameshi_ctl);
  tameshi_ctl1 = bdd_dagptr_and(bddm, gennzai, tameshi_ctl1);

  if(tameshi_ctl1 == bddONE)
    printf("tameshi_ctl1 = bddONE\n");
  else if(tameshi_ctl1 == bddZERO)
    printf("tameshi_ctl1 = bddZERO\n");
  else
    printf("tameshi_ctl1 != bddZERO\n");

  printf("*****************************************************************\n");
```

以下に実行結果を示す.

```
**************************************************************
EX
初期状態から、次の状態で OUTFLOW1 が ((0, inf), +) である状態へ到達できるパスが存在する.
初期状態 ^ EX(OUTFLOW1 = ((0, inf), +))

tameshi_ctl1 != bddZERO
**************************************************************
EF
初期状態が、AMOUNT = (full, +) である状態へ到達できるパスが存在する.
 初期状態 ^ EF(AMOUNT = (full, +))

..
tameshi_ctl1 != bddZERO
**************************************************************
AF
初期状態であるならば、常に amount = (full, +) or (full, -) or (full, 0) である状態に到達するパスが存在する.
 初期状態 ^ AF(AMOUNT = (full, + or - or 0))

.
tameshi_ctl1 = bddZERO
**************************************************************
AF
初期状態から、outflow1 が ((0, inf), +) である状態へ到達できるパスが常に存在する.
 .
tameshi_ctl1 != bddZERO
**************************************************************
```

第7章　考察

7.1　効率化について

従来のアルゴリズムに基づいて作られた定性シミュレータ JQSIM は到達可能状態を求めるのにバスタブ 4 連モデルでは, 実行して 129 時間後に, メモリ不足によりシミュレーションが途中で終了してしまったが, BDD を用いた定性シミュレーションでは, 到達可能な状態を表わす BDD を出力するのに 0.49 秒しかかからなかった. また, バスタブ 10 連モデルにおいてでさえ, わずか 20 秒ほどでシミュレーションは終了した. 以上の結果より BDD を用いた定性シミュレーション法は, JQSIM よりも計算速度において効率化されているといえる.

記憶効率に関して考えてみる. JQSIM はバスタブ 4 連モデルにおいて, シミュレーションが途中で終了し実行不能となったが, 到達可能状態として 84008 の状態を求めている. 一方, BDD を用いた定性シミュレーション法ではバスタブ 4 連モデルの到達可能状態を表わす BDD size はわずか 603 である. これは, JQIM によって求まった 84008 状態分以上を表現しているといえ, 記憶効率においても, JQSIM より BDD を用いた定性シミュレーション法の方が効率化されているとえる.

7.2　記号モデル検査について

バスタブ 1 連モデルは手作業でもその振る舞いを確認することが可能である. また, JQSIM においても状態数が少ないためその振る舞いを確認することが容易である. そこで, その結果と照らし合わせてみて, 本シミュレーションで求まった検証結果は妥当であるといえる.

第8章　おわりに

本研究で, BDD を用いて定性シミュレーションの効率化を図る手法について検討し, 実装を行い, BDD を用いない既存のアルゴリズムでプログラムされた JQSIM との比較実験を行った. その結果, BDD を用いたことにより計算速度, 記憶効率の観点から定性シミュレーションが効率化されていることが確認された. また, 記号モデル検査についてもその有効性が確認された.

本シミュレーションの対象としてシステムバイオロジーの分野における遺伝子発現過程のシミュレーションへの応用が期待される. それは, 遺伝子発現課程はハイブリッドシステムとして記述できるが, 酵素反応であるため正確なパラメータを与えることが困難であるという理由による. 本シミュレーションが不完全情報を持つシステムに対してモデル化されており, 大規模な状態数を扱うのに適していることから, システムバイオロジーの分野での応用に非常に期待が持てるシミュレーション法であるといえる.

謝辞

本研究を行うにあたり, 多大なるご指導・御鞭撻を賜りました平石 邦彦 教授に深く感謝の意を表します. また, 本研究をまとめるにあたり, 有益な御助言を頂きました宋 少秋助手, に心より感謝します.

また, 日頃から多大なる議論と激励を頂きました平石研究室の諸先輩方、平石研究室の皆様に厚くお礼申し上げます.

参考文献

[1] B.Kuipers:Qualitative Reasoning - Modeling and Simulation with Incomplete Knowledge, The MIT Press(1994).

[2] Akers, SS. B. : Binary Dicision Diagrams, IEEE Trans. Comput., pp.509-516 (1978).

[3] Bryant, R. E. : Graph-Based Algorithms for Boolean Function Manipulation, IEEE Trans. Comput.,pp.677-691(1986).

[4] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled : Model Checking, The MIT Press(1999)

[5] 石浦 菜岐佐 : BDD とは, 情報処理, Vol.34 No. 5, pp. 585-592(1993).

[6] 湊 真一: 計算機上での BDD の処理技法, 情報処理, Vol.34 No. 5, pp. 593-599(1993).

[7] 平石 裕実、浜口 清治 : 論理関数処理に基づく形式的検証手法, 情報処理, Vol 35 No. 8, pp. 710-718(1994).

[8] 淵 一博 監修 溝口文雄、古川康一、安西祐一郎、共編 “定性推論”, 共立出版株式会社,1989

[9] 北野 宏明 編 : “システムバイオロジーの展開”, シュプリンガー・フェアラーク東京 (2001)