数理解析研究所講究録
第 1554 巻 2007 年 194-201

述語抽象化洗練を用いたリアルタイムプログラムの自動検証手法

Automatic Verification for Real-time Programs using Predicate Abstraction and Refinement

(Extended Abstract)

駒形龍太* 山根智†
Ryota Komagata Satoshi Yamane
金沢大学大学院 自然科学研究科 電子情報工学専攻
Division of Electrical and Computer Engineering,
Graduate School of Natural Science & Technology, Kanazawa University

概要

本論文では, まずリアルタイムプログラムについて定義を行う. そして, その上での効率的な安全性検証手法を提案する. 提案手法により, 組込システムのようなリアルタイムな動作, 情報処理的な動作を持つシステムに対する, 効率的な自動検証が可能となる. 検証の効率化には, 述語抽象化とその洗練による手法を用いる. さらに本論文では, 検証問題における停止性について述べる.

Abstract. In this paper, we first define the *real-time program*. And next, we present a efficient verification method for real-time programs. This method enables effective automated verification for systems which has Real-time-operation and Data-processing-operation like embedded systems. This verification method is based on predicate abstraction and refinement. Furthermore, we discuss the termination of this verification method.

Keywords: formal method, real-time programs, predicate abstraction and refinement

1 まえがき

近年, ソフトウェアに対する信頼性の要求の高まりから, 対象とするシステムにおけるある性質に対し, 数学的な証明を与えることができる, **形式的検証法 (Formal Method)** が注目されている. 本論文では形式的検証法に則った, 検証を伴うソフトウェア開発モデルの提案を行う.

ソフトウェアの**安全性 (Safety Properties)** の検証には様々な研究が行われており, システムの制御的な側面からはハイブリッドオートマトンによる検証 [3], 時間オートマトンによる検証 [2] など, 連続的に変化するシステムの要因に関する研究が行われている. また一方でプログラミング言語に対する検証として, C 言語に対する検証 [5] が行われ, デバイスドライバに適用し成果を挙げている.

しかし, 形式的検証法を実際の規模のシステムに適応するにあたり, **状態爆発 (State Explosion)** が問題となる. この問題を解決するために過去に様々な研究が行われてきた [7]. そして, システム検証における強力な抽象化法として**述語抽象化法 (Predicate Abstraction)**[8] が考案された. 述語抽象化法では, システムの状態空間を述語を用いて抽象化して表現することにより, 状態空間の縮小が可能となる.

そしてさらに, 述語抽象化を用いて効率的な検証を行っていくための枠組みとして, **述語抽象化とその洗練の枠組み (CounterExample Guided Abstraction and Refinement(CEGAR))**[6] が考え出された.

この枠組みのC言語への適用としては [5], [9] が報告されており, 他にも Java 言語などに対してもツールによる実装が報告されている [11]. そしてハ

*E-mail: komagata@csl.ec.t.kanazawa-u.ac.jp
†E-mail: syamane@is.t.kanazawa-u.ac.jp
‡〒 920-1192 金沢市角間町 金沢大学大学院 自然科学研究科 電子情報工学専攻

イブリッドオートマトンへの適用としては [4] が報告されている．また我々もこの枠組みを確率線形ハイブリッドオートマトン [12] に適用し，比較実験によりその成果を実証した [13]．

本論文では新しい仕様記述言語としてリアルタイムプログラム (*Realtime Program*) を定義し，その上で CEGAR による安全性検証を行う．リアルタイムプログラムではソフトウェアの重要な要素として，リアルタイム処理，データ処理をモデル化する．このような処理をモデル化することにより，組込みソフトウェアのような用途に応じて様々な動作を必要とするソフトウェアにおけるの信頼性の高いシステム仕様記述が可能となる．

リアルタイム性を持つシステムに対する CEGAR による検証としては [10] が報告されているが，本手法では更にデータ処理を記述可能とすることで適用範囲を広げている．

以降２章では，本論文の仕様記述言語であるリアルタイムプログラムについて定義し，３章では述語抽象化とその洗練を用いた安全性検証の適用について述べる．そして４章では本検証法の停止性について述べ，最後に５章でまとめとする．

2 リアルタイムプログラム

この節では本研究の検証対象とする仕様記述言語であるリアルタイムプログラム(*RP*) の定義を行う．*RP* は直感的には時間オートマトン [2] を離散的に変化するデータ変数 (*data variables*) で拡張したモデルである．データ変数によりシステムのデータ処理的を，クロック変数によりシステムのリアルタイム処理をモデル化することができる．

2.1 準備

まず RP で扱う変数について以下で定める．

Definition 1 [データ変数とクロック変数]

データ変数は整数値をとる変数であり，データ変数の有限集合 D とする．またクロック変数は非負実数値をとる変数であり，クロック変数の有限集合を X とする．D, X を以下のように形式的に定義する．

$D = \{d_1, d_2, \ldots, d_n | d_i \in \mathbb{Z}\}$
$X = \{x_1, x_2, \ldots, x_m | x_i \in \mathbb{R}\}$

クロック変数はシステムの遅延により全ての変数が同じ速度で変化する．さらにシステムの離散的変化により個別に値が 0 にリセットされる．またデータ変数はシステムの離散的変化により定められた演算式に従いその値を変える．本論文では定数の加算・減算を演算として扱い，演算式を以下のよう定義する．

Definition 2 [データ変数における演算式]

$$d ::= d + k | d - k | k$$

ただし k は整数値を持つ定数である．

また，データ変数とクロック変数における制約として以下のものを定義する．$\sim \in \{<, >, =, \leq, \geq\}$ である．

データ変数 d におけるデータ制約 ϕ^D は $d \sim c$ または $d_1 - d_2 \sim c$ の形で表される．ここで $\sim \in \{<, >, =, \leq, \geq\}$ である．

クロック変数 x における制約はロケーションの不変条件や遷移可能条件に現れ，クロック変数 x におけるクロック制約 ϕ^X は $x \sim n$ または $x_1 - x_2 \sim n$ の形で表される．ここで，$\sim \in \{<, >, =, \leq, \geq\}$ である．

ϕ^D, ϕ^X を以下で形式的に定義する．ここで，k, n は整数値を持つ定数である．

Definition 3 (クロック制約とデータ制約)

$\phi^D ::= d \sim k | d_n - d_p \sim k | \phi^D_1 \wedge \phi^D_2$
$\phi^X ::= x \sim n | x_m - x_o \sim n | \phi^X_1 \wedge \phi^X_2$

2.2 構文

Definition 4 (RP の構文)

リアルタイムプログラムは以下の７つ組みにより構成される．

$$RP = \langle L, D, X, E, l_0, inv, T \rangle$$

- ロケーションの有限集合　L
- データ変数の有限集合　D
  データ変数の評価 (**valuation**) μ はそれぞれのデータ変数 $d \in D$ に整数 $\mu(x) \in \mathbb{Z}$ を割り付ける関数である．
- クロック変数の有限集合　X
  クロック変数の評価 (**valuation**) ν はそれぞれのクロック変数 $x \in X$ に実数 $\nu(x) \in \mathbb{R}^{\geq 0}$ を割り付ける関数である．
- イベントの有限集合　E
- 初期状態　l_0
  l_0 は初期ロケーションである．データ変数，ク

ロック変数の初期割り付けは全て0であるものとする.

- 不変式の関数　$inv : S \rightarrow \Psi^X$
  それぞれのロケーション $l \in L$ に**不変条件 (invariant)** $inv(l) \subseteq \Phi^X$ を割り付ける.
- **遷移関係の有限集合**
  $T \subseteq L \times E \times GUARD \times ACT \times L$
  $l \in L$ から $l' \in L$ への**遷移** $t \in T$ は以下の6つ組みで定義される.

$$\langle l, event, guard, act, l' \rangle \in T$$

  - l は遷移元のロケーションである.
  - $event \in E$ は遷移におけるイベントである.
  - $guard \in \Phi^V \cup \Psi^X$ はガード条件であり, 遷移を制限する.
  - act はアクション式でありクロックリセット, データ演算式からなる. クロックリセットは, $x := 0$ の形で表す. ここで $x \in X$ である. データ演算式は定義で示したとおりである.
  - l' は遷移先のロケーションである.

Example1 [リアルタイムプログラム]
以下にリアルタイムプログラムの簡単な例を示す.

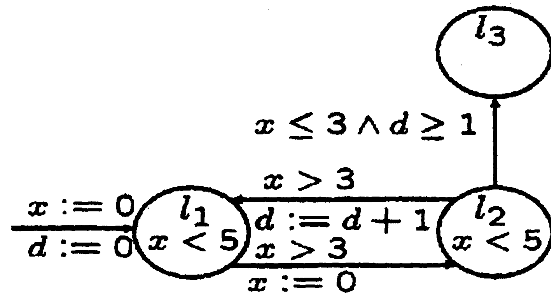

図 1: リアルタイムプログラムの例

上図を用いてリアルタイムプログラムの動作を説明する.

初期状態：　$(l_1, x = 0 \wedge d = 0)$
時間経過：　$(l_1, x = 4 \wedge d = 0)$
l_1から l_2への遷移：$(l_2, x = 0 \wedge d = 0)$
時間経過：　$(l_2, x = 3.5 \wedge d = 0)$
l_2から l_1への遷移：$(l_1, x = 3.5 \wedge d = 1)$

このように, 時間経過とロケーション間の遷移を繰り返すことによりリアルタイムプログラムは動作していく.

2.3 意味論

次に RP の意味論について定義する.

Definition 5 (RP の動作状態)
RP の動作状態は $q = \langle l, \mu, \nu \rangle$ であり, このとき各項は次のとおりである.

1. ロケーション $l \in L$
2. データ変数の評価関数 $\mu : D \rightarrow \mathbb{Z}$
3. クロック変数の評価関数 $\nu : X \rightarrow \mathbb{R}^+$

2.3.1 RP の実行

RP のあるロケーションにはそのロケーションの不変条件が真であるときにのみ, 滞在してよい. すなわち, 不変条件が偽になる前に離散遷移は起こらなければならない.

また RP において有限の時間区間において無限回の離散遷移は行われないものとする (nonzenoness).

リアルタイムプログラムの実行列 R_{UN} は, 以下のような有限もしくは無限の列である：

$$R_{UN} : \langle l_0, \mu_0, \nu_0 \rangle \xrightarrow{l_1} \langle l_1, \mu_1, \nu_1 \rangle \xrightarrow{l_2} \langle l_2, \mu_2, \nu_2 \rangle \xrightarrow{l_3} \cdots$$

ここで, $q_0 = \langle l_0, \mu_0, \nu_0 \rangle$ は初期状態であり, l_0 は初期ロケーション, μ_0, ν_0 のすべての変数には初期値が割り付けられており, q_i は RP の動作状態, $l_i \in (E \cup \mathbb{R}^+)$ はラベルである.

2.3.2 RP の遷移系

RP を遷移系として定義する.

Definition 6 [遷移系]
リアルタイムプログラム RP の遷移系 $TS_{RP} = (Q_{RP}, \rightarrow, I_{RP})$ は以下のように定義される.

- $Q_{RP} \subseteq L \times [\mathbb{Z}^n]^D \times [\mathbb{R}^{+^m}]^X$ は状態 (l, μ, ν) の集合である. ただし ν は $inv(l)$ を満たしているものとし, これを $\nu \in inv(l)$ と書く.
- $\rightarrow$ は状態間の遷移関係であり**離散-ステップ関係** $\rightarrow^{edge}$ と, **時間-ステップ関係** $\rightarrow^{time}$ の和集合である.
  - **離散-ステップ関係**
    $\rightarrow^{edge}$ は $event \in E$ において,

$$\frac{(l, event, guard, act, l') \in T, (\mu, \nu) \in guard, \nu' \in inv(l')}{(l, \mu, \nu) \rightarrow^{edge} (l', \mu', \nu')}$$

    を満たす. ここで, μ', ν' は act によって μ, ν を更新したものである.

- 時間-ステップ関係
  $\rightarrow^{time}$ は $t \in \mathbb{R}^{\geq 0}$ において，
  $$\frac{\forall 0 \leq t' \leq t. \nu + t' \in inv(l)}{(l, \mu, \nu) \rightarrow^{time} (l, \mu, \nu + t)}$$
  を満たす．

- I は RP の初期状態 q_0 である．

遷移系 TS_{RP} は反射的である．

2.4 リアルタイムプログラムにおける安全性検証

安全性検証のための手法として，エラー状態への到達可能性解析を用いる．まず，到達可能性問題を形式的に定義する．

Definition7 (到達可能性問題)

リアルタイムプログラム RP= $\langle L, l_0, D, X, E, inv, T\rangle$ に対して L_T をターゲットとするロケーションの集合とする．RP の初期状態 $q_0 = (l_0, \mu_0, \nu_0)$ から始まり，$l_n \in L_T$ となる状態 $q_n = (l_n, \mu_n, \nu_n)$ に到達する RP の実行が存在するとき，かつ，そのときに限り，答えは"$Y_{ES}, reachable$" となる．それ以外は"N_O" である．

本論文では，危険な状態をターゲットとした初期状態からの深さ優先探索により，到達可能性解析を行う．

2.4.1 記号的到達可能性解析

まず，RP の記号的状態について定める．

Definition8 (RP の記号的状態)

RP における記号的状態を $R = \langle l, P_D, P_X\rangle$ で表す．ここで P_D はデータ変数の評価値 ν の集合であり，$P_D = \{\phi_1^D \wedge \phi_2^D \wedge \ldots \wedge \phi_n^D | n \in \mathbb{N}\}$ の形で表現される．また，P_X はクロック変数の評価値 μ の集合であり，$P_X = \{\phi_1^X \wedge \phi_2^X \wedge \ldots \wedge \phi_n^X | m \in \mathbb{N}\}$ の形で表現される．記号的到達可能性解析では，RP の記号的状態に対して記号演算を行うことにより遷移後の状態を求める．

Definition9 (Successor 演算)

記号的状態 $\langle l, P_D, P_X\rangle$ に対して次のように演算を定義する．まず，P_X における前進射影を $\nearrow P_X$ と定義する．このような $\nu \in \nearrow P_X$ の必要十分条件は，$\exists t \in \mathbb{R}. \nu - t \in P_X$ である．

Time Successor:

$\mathbf{time_succ}\langle l, P_D, P_X\rangle := \langle s, P_D, \nearrow P_X \cap inv(l)\rangle$

$\mathbf{time_succ}\langle l, P_D, P_X\rangle$ は $\langle l, P_D, P_X\rangle$ から，時間-ステップ関係により遷移可能な状態の集合である．

Discrete successor:

遷移 $T = (l, event, guard, act, l')$ における discrete successor を以下のように定義する．

$\mathbf{disc_succ}(e, \langle l, P_D, P_X\rangle) :=$
$\langle l', ((P_D \cap guard)[act]), ((P_X \cap guard)[act]) \cap inv(l')\rangle$

$\mathbf{disc_succ}\langle l, P_D, P_X\rangle$ は $\langle l, P_D, P_X\rangle$ から，遷移-ステップ関係 e により遷移可能な状態の集合である．

次に **Time Successor, Discrete Successor** を用いて，**Successor** 演算 **Succ** を次のように定義する．

$\mathbf{Succ}(e, \langle l, P_D, P_X\rangle) :=$
$\mathbf{time_succ}\ (\mathbf{disc_succ}\ (e, \langle l, P_D, P_X\rangle))$

$\mathbf{Succ}(e, \langle l, P_D, P_X\rangle)$ は，$\langle l, P_D, P_X\rangle$ から遷移-ステップ関係 e により遷移可能な状態を集合を求め，遷移先のロケーションで時間-ステップ関係により遷移可能な状態の集合を求める演算である．

Succ を繰り返し，深さ優先探索により初期状態から遷移可能な全ての状態を求めることにより，到達可能性問題を解くことができる．

Example2 [記号的到達可能性解析]

以下に記号的到達可能性解析の例を示す．

図 1 のモデルに対して，**successor** 演算を適用する．

$\mathbf{Succ}(In \rightarrow l_1, \langle l_1, d = 0, x = 0\rangle))$
$= \langle l_1, d = 0, 0 \leq x \leq 5\rangle$ （初期状態の計算）
$\mathbf{Succ}(l_1 \rightarrow l_2, \langle l_1, d = 0, 0 \leq x \leq 5\rangle))$
$= \langle l_2, d = 0, 0 \leq x \leq 5\rangle$ （l_1から l_2 への遷移）
$\mathbf{Succ}(l_2 \rightarrow l_1, \langle l_2, d = 0, 0 \leq x \leq 5\rangle))$
$= \langle l_1, d = 1, 3 \leq x \leq 5\rangle$ （l_2から l_1 への遷移）

以上のように，**successor** 演算により初期状態から遷移可能な状態集合を求めることができる．上記の計算で求められた状態集合は Example 1 の動作例を含み，記号的到達可能性解析で，モデルの全てのふるまいをまとめて求められることが見て取れる．

3 述語抽象化とその洗練による自動検証手法

前節で定めた記号的到達可能性解析手法により，計算機上でリアルタイムプログラムの安全性検証を行うことが可能となった．しかし，状態爆発はいまだに問題であり，本節ではまず，状態爆発を解消するための述語抽象化法について述べる．

3.1 述語抽象化

RP の変数の状態空間に対して述語の k 次元ベクトル $\Pi = [\pi_1, \pi_2, \ldots \pi_k]$ を与え，状態空間を各述語 $\pi_i \in \Pi$ を満たす領域 $\mathcal{H}(\pi_i)$ とそうでない領域 $\overline{\mathcal{H}(\pi_i)}$ に分割する．これにより変数の状態空間全体は，たかだか 2^k 個の領域に分割される．

述語抽象化は検証すべき状態空間の縮小に貢献するが，その反面，検証に必要な情報を失ってしまう可能性がある．

Example 3 [述語抽象化]
図 1 のモデルに対して述語抽象化を適用した例を示す．述語 $\{x \geq 3\}$ と $\{d = 0\}$ を用いて状態空間を抽象化する．これにより，状態空間を 2 つの述語の真偽値に関して以下のように 4 つに分割することが可能となる (T:真, F:偽).

$Pred1 : \{x \geq 5\}, \{d = 0\}$ =T, T
$Pred2 : \{x \geq 5\}, \{d = 0\}$ =T, F
$Pred3 : \{x \geq 5\}, \{d = 0\}$ =F, T
$Pred4 : \{x \geq 5\}, \{d = 0\}$ =F, F

次に抽象化によって作られた状態空間を持つモデルにおける解析について述べる．

3.2 抽象モデルにおける到達可能性解析

本来の RP の遷移系を具体モデル $\mathcal{C}$ とし，具体モデルを抽象化することによって作られる抽象モデルを $\mathcal{A}$ とする．

抽象化によって抽象構造を作る場合，安全性検証においては**到達不能性**を保存するような抽象化を行う．このとき，$\mathcal{A}$ における到達可能性解析で得られる結果は以下のように扱う．

- エラーへ到達せず : $\mathcal{A}$ は到達不能性を保存しているので，エラーへ到達しなかったという結果は $\mathcal{C}$ においても正しい．
- エラーへ到達 : $\mathcal{A}$ は到達可能性を保存していないので，$\mathcal{C}$ においてもエラーへ到達するかは判断できない．

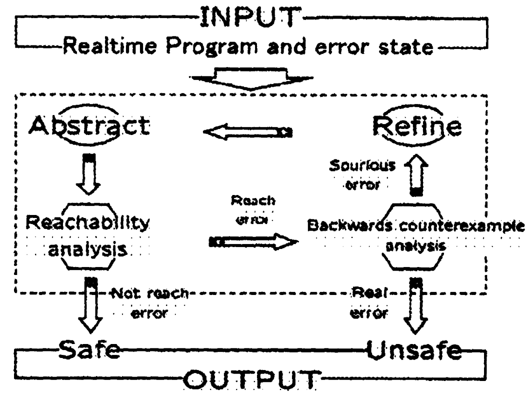

図 2: CEGAR の枠組み

Example 4 [抽象モデルにおける到達可能性解析]
図 1 のモデルに対し、述語 $\{x \geq 5\}$ を用いて抽象化した抽象モデル，
Pred1=$\{x \geq 5\}$ =T, Pred2=$\{x \geq 5\}$ =F
におけるエラー状態 ($Error = l_3$) への到達可能性解析の例を示す．

遷移:$In \rightarrow l_1$　$Pred1$ =F, $Pred2$ =T
遷移:$l_1 \rightarrow l_2$　$Pred1$ =F, $Pred2$ =T
遷移:$l_2 \rightarrow l_3$　$Pred1$ =T, $Pred2$ =T

よって l_3 に到達する状態集合が存在し，3 度の遷移でエラー状態に到達することが分かる

Example 4 のように，エラーへ到達した場合においてはエラーの真偽を決定するために更なる検証が必要である．そこで，次のような枠組みを用いる．

3.3 述語抽象化とその洗練の枠組み (CounterExample Guided Abstract and Refinement(CEGAR))

図 2 に CEGAR[6] のモデル図を示す．

CEGAR による検証の流れ

1. 具体モデル $\mathcal{C}$ とエラーロケーションを入力する．
2. 選択された述語の集合から，抽象モデル $\mathcal{A}$ を構成する (Abstract). 最初の述語は 0 個であるとする．
3. $\mathcal{A}$ において到達可能性解析を行う (Reachability analysis). エラーへ到達しなかった場合，システムは安全であると出力し，停止する．
4. 3 でエラーへ到達した場合，得られたエラーパス (反例と呼ぶ) が $\mathcal{C}$ でも存在する実反例か，$\mathcal{C}$ では存在しない偽反例かを後方反例解析により判定する (Backwards counterexample analysis).

実反例だったときシステムは危険であると出力し停止する.

5. 4で偽反例であると判定されたとき, その偽反例を排除する述語を追加し (Refine), 1に戻る.

次に後方反例解析, 述語の追加による抽象モデルの洗練について説明する.

3.3.1 後方反例解析

後方反例解析では抽象モデルにおける反例が具体構造でも存在するのかどうかをチェックする. 具体的には, 到達可能性解析の結果得られたエラーパス $EPass = (l_0, \ldots, l_n)(l_n \in L_T)$ が具体構造でも存在するかどうかを l_n から後ろ向きに解析する.

後方解析の際, **最弱前条件 (weakest precondition)** と呼ばれる概念を用いてある状態に到達可能な最大の状態集合を求めていく. 一般に, ある遷移 e の直後における条件 ϕ に対し, e の直前における条件文 μ のうち, e により遷移可能であるものの中で最も弱いものを, その遷移に関する ϕ の最弱前条件という. 例えば, 遷移における演算式 $d = d + 1$ に関する条件 $d \geq 5$ の最弱前条件は $d \geq 4$ である.

後方反例解析の結果, 以下の2通りの結果が起こりえる.

1. 初期状態まで解析が行われ, エラー状態へ到達可能な状態集合が見つかる. つまり, 本物のエラーが見つかる.
2. 解析途中に状態集合が空集合となる. つまり, そのパスは具体モデルでは存在しない spurious なパスであるということになる.

1の場合は, エラーパスを出力し, 検証を終える. 2の場合では, 今回用いられた述語では, システムの安全性を検証するためには不十分であるということになる. よって, さらに述語を追加することによって抽象モデルを洗練し, 再度の到達可能性解析を試みる.

Example 5 [後方反例解析]

Example 4 の結果のエラーパスが具体モデルにおいても存在するかを, 後ろ向きに確かめる.

遷移:$l_2 \to l_3$　$x \leq 3 \wedge d \geq 1$

遷移:$l_1 \to l_2$　$3 < x < 5 \wedge d \geq 1$

遷移:$In \to l_1$　$d = 0 \wedge d \geq 1 \Rightarrow \emptyset$

l_3 に到達するためには, 入力の時点で $d \geq 1$ となっていなければならない. しかし図1では入力で $d = 0$ と割り付けられているため, そのような状態集合は存在しない. よって最弱前条件は空集合となり, このエラーパスは具体モデルでは存在しない spurious なパスだということが分かる.

3.3.2 述語の追加

述語の追加では sprious なエラーパス $EPass = (l_0, \ldots, l_n)$ を消去可能な述語を選択する. このエラーパスは遷移における変数の状態空間に関する情報の不足により発生したものであるので, 述語はエラーパス中の遷移に現れた遷移可能条件, 不変条件, 遷移における割り付けを利用して選択すればよい.

追加された述語により新たな抽象モデル $\mathcal{A}'$ が作られる.

Example 6 [述語の追加及び再度の探索]

Example 5 の後方反例解析により, d の値を表現する述語が不足していたため, spurious なエラーパスが生まれてしまったことが分かる. よって d の値を分割する述語 $\{d = 0\}$ を追加することにより抽象モデルを洗練し, さらなる到達可能性解析を行う.

述語 $\{x \geq 3\}, \{d = 0\}$ を用いた到達可能性解析では,

遷移:$In \to l_1$　$Pred1, 2, 3, 4$ =T,F,F,F

遷移:$l_1 \to l_2$　$Pred1, 2, 3, 4$ =T,F,F,F

遷移:$l_2 \to l_3$　$Pred1, 2, 3, 4$ =F,F,F,F

(遷移不可のため他のエッジを探索する)

遷移:$l_2 \to l_1$　$Pred1, 2, 3, 4$ =F,T,F,F

遷移:$l_1 \to l_2$　$Pred1, 2, 3, 4$ =F,T,F,F

遷移:$l_2 \to l_3$　$Pred1, 2, 3, 4$ =F,T,F,T

となり, さきほどのエラーパスは排除されるが, 他のパスを通ることによりエラー状態に到達することが分かる.

今回の到達可能性解析で得られたエラーパスを後方反例解析により判定すると, 具体モデルでも存在するエラーパスであることが分かる. よって, 危険であると出力し, 検証を終える.

4 検証の停止性

この節では前節までに示した検証手法における停止条件について考察する.

まず, リアルタイムプログラム RP の記号的到達可能性解析は停止しない場合が存在する. まず遷移によりエラー状態に到達する場合, いずれ検証は停止する. しかしエラー状態に到達せず, さらに不動点が計算できない場合, RP の記号的到達可能性解

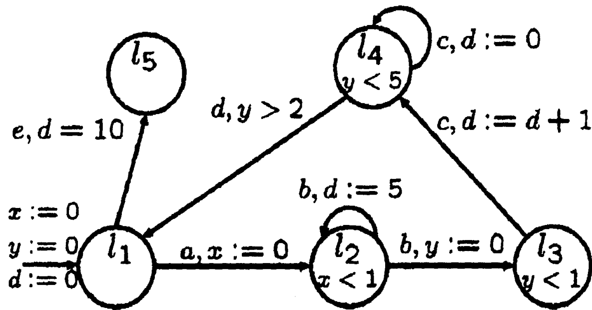

図 3: CEGAR の導入により検証が停止する

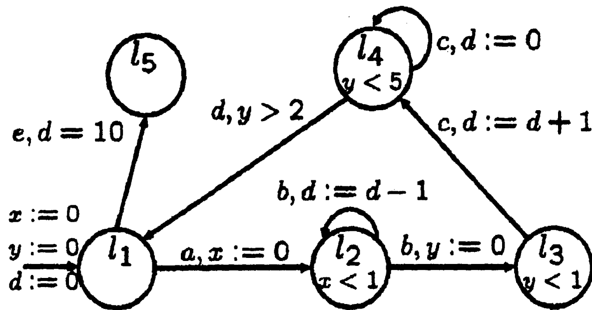

図 4: CEGAR を導入しても検証が停止しない

析は停止しない.

図3, 図4の例において, l_5 に対する記号的到達可能性解析は停止しない. これはモデル中で変数 d に対する加算命令が無限回実行可能であることから, 状態集合が発散してしまうためである.

しかし, 検証が停止した場合は, 出力された結果は正しい.

4.1 CEGAR 検証の停止性

CEGAR による検証の停止問題は次のように分解して考えることができる.

- 1回の検証ループは停止するか?
- 検証ループの繰り返しは停止するか (述語の追加は収束するか)?

まず, 1回の検証ループの停止について考えると (図2参照), 一回の検証ループはそれぞれ有限の計算となるので停止する.

次に検証ループの繰り返しは有限回で停止するかについて考える.

まず, 遷移によりエラー状態に到達する場合, さきほどと同様に停止する.

そしてエラー状態に到達しない場合, 検証ループの繰り返しにより述語の追加が繰り返されることになる. 述語の追加を考えるに当たってデータ変数, クロック変数のそれぞれに対して考えていく必要がある.

クロック変数に関してはその状態空間は有限の述語数で表現することができることが示されているが [1], データ変数に関しては加算・減算の演算により, 述語の追加は停止しない.

CEGAR による検証の停止問題をさきほどの例 (図3, 図4) で考える. 図3, 図4のモデルでは, 遷移 $l_2 \rightarrow l_2$ におけるデータ変数への演算式のみが異なっており, 他は全て等しい. 2つのモデルで l_5 へ の到達可能性を検証する際, l_1 から l_5 への遷移において d に関する遷移可能条件 ($d = 10$) があることから, d の値が検証に影響を与えることが分かる.

このとき図4のモデルでは, d に対する加算・減算命令により d の値は $\pm\infty$ に発散する. よって厳密に遷移に関して状態集合を計算するためには, d の全ての値に対して述語により区別を与えなければならない. これより述語数は無限に発散し, 検証は停止しない.

図3のモデルにおいても, d に対する加算命令は存在し, d の値は $+\infty$ に発散しうる. しかし図3では, d の値がひとたび10を超えればそれらを区別する必要はなく, 述語 $\{d \geq 10\}$ でまとめて表現することができる. よって最終的な述語数は有限個となり, 検証は停止する.

よって, CEGAR による検証においても停止しないような入力 (図4) は存在するが, 変数の値が単調に増加し, 発散する場合 (図3) は検証は停止する.

5 むすび

本論文ではソフトウェアのリアルタイム処理, データ処理をモデル化した仕様記述言語であるリアルタイムプログラムを定義し, その上での効率的な安全性の検証手法を示した. また, 述語抽象化とその洗練における検証法で重要な問題となる検証の停止条件についての考察を行った.

CEGAR の枠組みを用いることにより, 検証を各フェーズに切り分けて行うことができ, 計算量の爆発を抑制することができる. 本手法を実装するに当たって, 効果的に抽象モデルを洗練していくことが重要となる. よって, 最適な述語の追加を行うためのアルゴリズムを検討する必要がある.

今後の課題としては，まず本手法を実装し大規模なシステムにおいてその有効性を示すことが挙げられる．さらに複雑なソフトウェアの動作をモデルに取り入れることで，より現実的なシステムの検証を行う．

また UML のステートチャートに検証を適用することにより，システムの仕様記述において，一般的に用いられているモデルに対する検証手法を提案することも，今後の課題となる [14].

参考文献

[1] R. Alur, C. Courcoubetis, N. Halvwachs, D.L. Dill, and H. Wong-Toi. Minimization of timed transition systems.*CONCUR 92*, pp.340-354. 1992.

[2] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183-235, 1994.

[3] R. Alur, C. Coucoubetis, T.A. Henzinger, R.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3-34, 1995.

[4] R. Alur, T. Dang, and F. Ivancic. Counter-Example Guided Predicate Abstraction of Hybrid Systems, LNCS 2619:208-223, 2003

[5] T. Ball and S.K. Rajamani. Checking Temporal Properties of Software with Boolean Programs. *In Workshop on Advances in Verification(with CAV 2000)*, 2000.

[6] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu and H. Veith. Counterexample-Guided Abstraction Refinement. *Conference on Computer Aided Verification CAV 2000*, LNCS 1855, pp.154-169, 2000.

[7] E.M. Clarke, O. Grumberg, and D.A. Peled, Model Checking. *MIT Press*, 1999.

[8] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. *LNCS*, 1254:pp72-83, 1997.

[9] T.A. Henzinger, R. Jhala, R. Majumdar and G. Sutre. Lazy Abstraction. *ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, pp.58-70, 2002.

[10] O. Möller, H.Rueb, and Maria. Sorea. Predicate abstraction for dense real-time systems. *ENTCS*, 65(6), 2002.

[11] 田辺良則, 高井利憲, 髙橋孝一. 抽象化を用いた検証ツールの調査. 産業技術総合研究所算譜科学グループ研究速報, AIST-PS-2003-007, 2003.

[12] Y. Mutsuda, T. Kato, S. yamane. Symbolic Reachability Analysis of Probabilistic Linear Hybrid Automata, *IEIEC A: on Fundamentals of Electronics, Communications and Computer Sciences*. Vol.E88A No.11, pp.2972-2981, 2005.

[13] 加藤位明, 陸田陽介, 山根智. 述語抽象化とその洗練による確率線形ハイブリッドオートマトンの到達可能解析手法. 電子情報通信学会研究報告 *CST2006*, pp19-24, 2006.

[14] 徳田学, 山根智. リアルタイムオブジェクトチャートによる仕様記述と検証. 電子情報通信学会研究報告 *SS2005-31*, pp25-30, 2006.