様相論理を使用したヒープ検証方式

田辺 良則[1,2] 湯浅 能史[1,3] 関澤 俊弦[1] 高橋 孝一[1]

述語抽象化は，ソフトウェアモデル検査における強力なフレームワークである．本研究では，述語抽象化を用いてヒープを扱うプログラムの検証を行う手法を提案する．我々の手法の特徴は，抽象化に用いる「述語」の表現に，様相論理の一種であるノミナル付き 2 方向 CTL (2CTLN) を用いることである．ヒープを構成するノードをポインタが指す，という関係を，様相としてとらえている．述語抽象化における主要なステップは，述語に関してプログラム実行文に対する最弱前条件を求めることと，充足可能性判定を行うことである．我々はこの 2 つの手続きを実行するアルゴリズムを考案し，以上の原理に基づく抽象化検証ツールを試作することで，この枠組みが検証問題に適用可能であることを示した．実用化にむけての研究課題についても論じる．

A heap verification method using modal logics

Yoshinori Tanabe, Yoshifumi Yuasa, Toshifusa Sekizawa and Koichi Takahashi

[1] 産業技術総合研究所 システム検証研究センター, Research Center for Verification and Semantics, National Institute of Advanced Industrial Science and Technology (AIST).

[2] 東京大学大学院 情報理工学系研究科, Graduate School of Information Science and Technology, the University of Tokyo.

[3] 科学技術振興機構 CREST, Japan Science and Technology Agency.

本研究は，科学技術振興機構戦略的創造研究推進事業 (CREST) 研究領域「情報社会を支える新しい高性能情報処理技術」研究課題「検証における記述量爆発問題の構造変換による解決」の一部として行われた．

1 はじめに

近年，モデル検査による検証手法は，大きな成功を収めている．この成功は主にハードウェア分野や，ソフトウェアにおける設計の分野で達成されてきた．そこでは，検証すべきシステムを形式化した遷移系が有限であり，有限遷移系を対象とするモデル検査手法はうまく合致している．大規模な検証対象をモデル検査可能とするための重要な手法に，抽象化がある．元来，大きな有限遷移系のサイズを，ある種の性質を保ったまま小さくする手法であるが，無限遷移系にも適用することができる．この手法を利用して，プログラムのソースコードを直接モデル検査して検証しようというのが，ソフトウェアモデル検査である．プログラムの実行過程を形式化して得られる遷移系は，実質的には無限遷移系であり，抽象化が欠かせない．ソフトウェアモデル検査の抽象化では，述語抽象化 [7] と呼ばれる枠組みが有力であり，それに基づくツールも開発されている．

しかし，従来のこうしたツールでは，ヒープの性質の検証は困難であった．述語抽象化の手法では，いくつかの述語に着目した上で，プログラムの実行に関するその最弱前条件を求める手続きと充足可能性の判定手続きが必要になる．ヒープに関する性質を表現した上で，これらの手続きを効率的に行うための標準的な枠組みがないことが，困難の原因である．

著者らは，グラフ書換系の性質を抽象化の手法によって検証する研究を行ってきており，これまで，セルオートマトンの性質の検証方式の提案 [4] などを

行った．抽象化および書換系の性質の記述のためには，個々のグラフで成立する条件を記述することが必要であるが，そのためには様相論理を用いている．エッジを様相と考えて，グラフをクリプキ構造とみなすのである．ヒープを扱うプログラムもグラフ書換系の一種であるから，これまでの手法が適用できると考えられる．具体的には，上述のヒープに関する性質の表現を様相論理を用いて行った上で，述語抽象化の枠組みに乗せるというものである．

本研究ではこの考え方に基づき，ヒープを扱うプログラムが定める無限遷移系から，抽象化を行って有限遷移系である抽象構造を得る具体的な方法を提案する．この抽象化は，十分広い範囲の仕様に関して健全なものである．すなわち，得られた抽象構造において仕様が成立していれば，もとのプログラムが仕様をみたすことが保証される．そして，抽象構造は有限遷移系であるから，仕様の成立はモデル検査の手法によって確認することができる．

一般に述語抽象化をベースにした検証において，検証者は次の作業を行う．(1) 与えられた述語から抽象構造を構築すること．(2) 抽象構造をモデル検査すること．(3) モデル検査から反例が得られた場合，対応する具体構造の反例が存在するかどうか，判定すること．(4) 存在しない場合，その反例を除去する新しい述語を作ること．本研究では，この (1) の部分を完全に自動化した．(2) は既存のモデル検査ツールで自動化できている．(3),(4) は現状では検証者の判断が必要であり，これらの自動化は今後の課題となっている．本稿では，課題解決のための方向についても論じる．

ヒープに関する仕様の検証に関して，さまざまなアプローチによる研究が行われている．

一つは，Hoare 流の証明を行うものである．これは，通常の Hoare 論理を拡張することで，ヒープを扱うプログラムに関する推論ができるようにしたもので，separation logic [6] と呼ばれる．対話型検証に向いたアプローチである．

ヒープの形状を解析する heap analysis は，自動検証を指向したアプローチである．代表的なものに TVLA [8] と呼ばれるシステムがある．これは，抽象解釈の理論に基づいて，ヒープを扱うプログラムの抽象実行を行うものである．抽象構造におけるノードが代表する複数の具体構造のノードにおいて，述語が常に成立するか，常に不成立か，どちらでもないかを表現するために，3 値論理が用いられている．

本論文の構成は以下の通りである．第 2 節では，ヒープの数学的表現であるポインタ構造を定義し，第 3 節でヒープを扱うプログラム言語 PML を導入したのち，第 4 節で，仕様を記述する言語を確定する．以上の準備のもと，第 5 節で抽象構造の作成方法を述べ，第 6 節で，本方式による検証の流れを説明し，第 7 節で，試験的な実装を用いた検証の実際を述べる．第 8 節で，実用化についての課題をあげ，第 9 節でまとめを述べる．

2 ポインタ構造

この節では，プログラムのヒープを表現する数学的構造を定義する．ヒープには，全体としてデータとともにポインタが格納されている．[6] のようにヒープは構造のない整数の列であり，ポインタは整数を番地と解釈することで得られるとする見方と，[8] のようにヒープを複数のポインタが格納されたセルの集合とみる見方とがある．我々は，様相論理を応用する立場から，後者に近い方法をとる．

Var, Val, Fld を空でない有限集合とする．Var の要素をプログラム変数，Val の要素を値，Fld の要素をフィールドと呼ぶ．Val の一つの要素 d_0 を固定する．

定義 1　有限集合 N, 関数 $p : N \times \mathrm{Fld} \to N$, 関数 $v : N \to \mathrm{Val}$, 関数 $\rho : \mathrm{Var} \to N$, N の要素 nil からなる 5 つ組 $\mathcal{S} = (N, p, v, \rho, \mathrm{nil})$ をポインタ構造と呼ぶ．N の要素をセルと呼ぶ．ポインタ構造全体のなすクラスを PtrStr' と書く．N, p, v, ρ, nil を，各々 $N_{\mathcal{S}}$, $p_{\mathcal{S}}$, $v_{\mathcal{S}}$, $\rho_{\mathcal{S}}$, $\mathrm{nil}_{\mathcal{S}}$ と書く．また，特別なオブジェクト $\perp$ を考え，$\mathrm{PtrStr}' \cup \{\perp\}$ を PtrStr と書く． ■

直観的な意味は以下の通りである．N が台集合，すなわち，ヒープをセルの有限集合と考える．各セル間には，複数種類のポインタが張られている．この種類をフィールドと呼ぶことにし，セル n_1 のフィールド

⟨*program*⟩ ::= ⟨*atomic prog*⟩ | ⟨*compound prog*⟩
⟨*atomic prog*⟩ ::= `skip;` | `abort;` | ⟨*var*⟩`:=NULL;`
| ⟨*var*⟩`:=`⟨*var*⟩`;` | ⟨*var*⟩`:=`⟨*var*⟩`.`⟨*fld*⟩`;`
| ⟨*var*⟩`.val:=`⟨*val*⟩`;` | ⟨*var*⟩`.`⟨*fld*⟩`:=`⟨*var*⟩`;`
| ⟨*var*⟩`:=new();`
⟨*compound prog*⟩ ::= ⟨*program*⟩⟨*program*⟩
| `if(`⟨*cond*⟩`){`⟨*program*⟩`}else{`⟨*program*⟩`}`
| `while(`⟨*cond*⟩`){`⟨*program*⟩`}`
⟨*cond*⟩ ::= ⟨*var*⟩`==NULL` | ⟨*var*⟩`==`⟨*var*⟩
| ⟨*var*⟩`.val==`⟨*val*⟩ | `!`⟨*cond*⟩ | ⟨*cond*⟩`||`⟨*cond*⟩
⟨*var*⟩ ::= (any element of Var)
⟨*val*⟩ ::= (any element of Val)
⟨*fld*⟩ ::= (any element of Fld)

図 1 PML の文法

f がセル n_2 を指していることを，$p(n_1, f) = n_2$ で表現する．また，各セルは値を持っており，セル n の値が d であることを，$v(n) = d$ で表す．さらに，セルを指すプログラム変数があり，変数 x がセル n を指していることを，$\rho(x) = n$ で表す．セルのフィールドや変数が,「何も指していない」こともあるが，あとの処理を簡単にするため，仮想的なセル nil を導入して，$p(n, f) =$ nil や $\rho(x) =$ nil となることをもって，何も指していないことを表現する．これによって，p と ρ は全域関数となる．また，$\perp$ は，プログラムが「異常終了」したことを表すために用いる．

3 プログラム言語

前節で定義したポインタ構造を操作するプログラムを記述するのに十分な表現力を持った，手続き型プログラム言語 PML (Pointer Manipulation Language) を導入する．

図 1 に，PML の文法を示す[†1]．PML のプログラム (図 1 の ⟨*program*⟩) 全体の集合を Prog と書く．原子プログラム (図 1 の ⟨*atomic prog*⟩) 全体の集合を AProg と書く．複合プログラム (図 1 の ⟨*compound prog*⟩) 全体の集合を CProg と書く．条件 (図 1 の ⟨*cond*⟩) 全体の集合を Cond と書く．アクションの集合 Action を，Action = AProg ∪ Cond で定義する．

```
y := NULL;
while (!(x == NULL)) {
  t := y;
  y := x;
  x := x.next;
  y.next := t;
}
```

図 2 PML プログラム例 reverse

PML プログラムの直観的な意味は以下の通りである．PML のプログラムは，初期状態にも依存して，正常に終了するか，異常終了するか，無限に実行し続けるかのいずれかである．`skip;` は，最も簡単な正常終了するプログラム，`abort;` は，最も簡単な異常終了するプログラムである.「NULL ポインタの先の参照」を行った場合にも異常終了する．6 種類用意されている代入文のうち，最初の 3 つは単に変数が指すセルを変更するものであり，後の 3 つはヒープ自体を変化させる．x`.val:=`v`;` によって，セルの値が変わり，x`.`f`:=`y`;` によって，セルのポインタが指すセルが変わり，x`:=new();` によって，新しいセルがヒープに追加される．

PML プログラムの例 reverse を，図 2 に示す．このプログラムを，変数 `x` にリストが与えられたとき (変数 `x` の指すセルからポインタ `next` をたどって nil に到達するとき) に実行すると，終了時にはリストが逆転し，その先頭を変数 `y` が指すようになる．

PML プログラム P の正確な意味は，P が引き起こす遷移系として以下で定める．そのために，まず，$a \in$ Action に対して，$[\![a]\!] \subseteq$ PtrStr × PtrStr を定義する．以下で $\mathcal{S}, \mathcal{S}' \in$ PtrStr$'$ とする．$(\mathcal{S}, \mathcal{S}') \in [\![a]\!]$ の意味は，$\mathcal{S}$ においてアクション a を実行した結果が $\mathcal{S}'$ になることである．ただし a が条件の時には，a が成立するときだけ「実行可能」であり，その場合ポインタ構造は変化しない，と解釈する．表 1 で 2 つの述語を定義している．cb$(c, \mathcal{S})$ は，$\mathcal{S}$ で条件 c を評価すると異常終了する，ct$(c, \mathcal{S})$ は，$\mathcal{S}$ で c を評価すると真に

†1 PML は，以下に述べる (具体および抽象) 遷移系の構成が簡単になるように最小限のコマンドセットを採用している．このため，たとえば `x := y.f1.f2;` という表現はできないが，同じ内容は，一時変数を用いて `tmp := y.f1; x := tmp.f2;` とすることで記述可能である

なる，という意味である．これらを用いて，$c \in \mathrm{Cond}$ について，$[\![c]\!]$ を以下を満たす最小の集合として定義する: $(\bot,\bot) \in [\![c]\!]$, $\mathrm{cb}(c,\mathcal{S}) \implies (\mathcal{S},\bot) \in [\![c]\!]$, $\neg\mathrm{cb}(c,\mathcal{S}) \wedge \mathrm{ct}(c,\mathcal{S}) \implies (\mathcal{S},\mathcal{S}) \in [\![c]\!]$. また，表 2 で 2 つの述語 $\mathrm{pb}(P,\mathcal{S})$ と $\mathrm{pt}(P,\mathcal{S},\mathcal{S}')$ を定義しており，前者は，$\mathcal{S}$ で P を実行すると異常終了する，後者は，$\mathcal{S}$ で P を実行すると $\mathcal{S}'$ になる，という意味である†3．これらを用いて，$P \in \mathrm{AProg}$ について，$[\![P]\!]$ を以下を満たす最小の集合として定義する: $(\bot,\bot) \in [\![P]\!]$, $\mathrm{pb}(P,\mathcal{S}) \implies (\mathcal{S},\bot) \in [\![P]\!]$, $\neg\mathrm{pb}(P,\mathcal{S}) \wedge \mathrm{pt}(P,\mathcal{S},\mathcal{S}') \implies (\mathcal{S},\mathcal{S}') \in [\![P]\!]$.

次に PML プログラム P の制御流れ図 $\mathrm{CFG}(P)$ を定める．これは，節の有限集合 $G = |\mathrm{CFG}(P)|$，開始節 $s \in G$, 終了節 $e \in G$, エッジの集合 $E \subseteq G \times G$, エッジのラベル関数 $l : E \to \mathrm{Action}$ からなる 5 つ組 (G, s, e, E, l) である．(1) 原子プログラム P については，$\mathrm{CFG}(P)$ の節は 2 点 s_P, e_P からなり，エッジは s_P から e_P に向かうもののみであり，このエッジにつくラベルは P 自身である．(2) CProg に対する制御流れ図は，図 3 のように定める．この図は，$\mathcal{G}_i = \mathrm{CFG}(P_i)$ とした場合の各複合プログラムに対する制御流れ図を示したものである． 例として，前出のプログラム reverse の制御流れ図を図 4 に示す．

$G' \subseteq G$ とする．G の要素の列 $(g_0,\ldots,g_n)$ $(n \geq 1)$ で，$g_0, g_n \in G'$, $g_i \in G \setminus G'$ $(0 < i < n)$, $(g_i, g_{i+1}) \in E$ $(0 \leq i < n)$ となるもの全体の集合を $\mathrm{path}(G')$ と書く．

これらを用いて，遷移系を定義する．

定義 2 P を PML のプログラム，$G \subseteq |\mathrm{CFG}(P)|$ とする．$\to_G$ を，以下を満たす最小の $\mathrm{PtrStr} \times G$ 上の関係として，G によって定まる遷移系 $\mathcal{T}(G) = (\mathrm{PtrStr} \times G, \to_G)$ を定義する: $\mathcal{S},\mathcal{S}' \in \mathrm{PtrStr}$, $(g_0,\ldots,g_n) \in \mathrm{path}(G)$, $(\mathcal{S},\mathcal{S}') \in [\![l(g_0,g_1)]\!] \circ \cdots \circ [\![l(g_{n-1},g_n)]\!]$ ならば，$(S,g_0) \to_G (S',g_n)$. ただし，$\circ$ は，関係の合成である．特に，$\mathcal{T}(|\mathrm{CFG}(P)|)$ を，P が引き起こす遷移系と呼ぶ． ∎

表 1 述語 cb, ct

c	$\mathrm{cb}(c,\mathcal{S})$	$\mathrm{ct}(c,\mathcal{S})$
x==NULL	偽	$\rho(x) = \mathrm{nil}$
x_1==x_2	偽	$\rho(x_1) = \rho(x_2)$
x.val==d	$\rho(x) = \mathrm{nil}$	$v(\rho(x)) = d$
!c	$\mathrm{cb}(c,\mathcal{S})$	$\neg\mathrm{ct}(c,\mathcal{S})$
c_1 || c_2	$\mathrm{cb}(c_1,\mathcal{S})$ $\vee\mathrm{cb}(c_2,\mathcal{S})$	$\mathrm{ct}(c_1,\mathcal{S})$ $\vee\mathrm{ct}(c_2,\mathcal{S})$

$\mathcal{S} = (N,p,v,\rho,\mathrm{nil})$ とする．

表 2 述語 pb, pt

P	$\mathrm{pb}(P,\mathcal{S})$	$\mathrm{pt}(P,\mathcal{S},\mathcal{S}')$†2
skip;	偽	$\mathcal{S} = \mathcal{S}'$
abort;	真	偽
x:=NULL;	偽	$\rho'(x) = \mathrm{nil}$
x:=y;	偽	$\rho'(x) = \rho(y)$
x:=y.f;	$\rho(y) = \mathrm{nil}$	$\rho'(x) = p(\rho(y),f)$
x.val:=d;	$\rho(x) = \mathrm{nil}$	$v'(\rho(x)) = d$
x.f:=y;	$\rho(x) = \mathrm{nil}$	$p'(\rho(x),f) = \rho(y)$
x:=new();	偽	$N' = N \uplus \{n\}$ $p'(n,f) = n$ $(f \in \mathrm{Fld})$ $v'(n) = d_0$ $\rho'(x) = n$

$\mathcal{S} = (N,p,v,\rho,\mathrm{nil})$, $\mathcal{S}' = (N',p',v',\rho',\mathrm{nil}')$, とする．

†2 下の 6 つの代入文については，表中に明示してあるもの以外は，$\mathcal{S}$ と $\mathcal{S}'$ が同じであることを示す．

†3 new() の意味がやや直観とは異なっているのは，後に述べる前条件の計算を簡単にするためである．ポインタフィールドを NULL で初期化することが必要なら，x:=new(); x.f:=NULL; のようにプログラムを書けば良い．

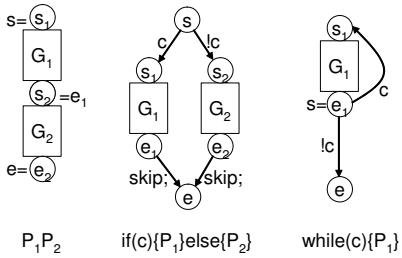

図 3 制御流れ図

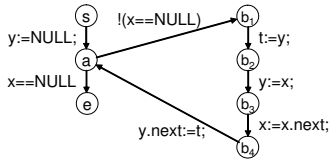

図 4 reverse の制御流れ図

4 仕様記述

プログラムの検証を行うためには，プログラムが満たすべき仕様を記述することが必要である．そのための言語を導入する．仕様記述言語には，(1) ある時点のヒープの状態を表現する能力，とともに (2) 状態の時間的な変化を表現する能力，も必要になる．

4.1 ヒープの状態記述

ポインタ構造を,「ポインタで指す」ことを様相としてとらえることによってクリプキ構造とみなし，セルが持つ性質を様相論理式で表現することで，ヒープの状態を記述しようというのが，基本的な考え方である．この目的のためには，最小の様相論理 **K** では記述能力に不足があるので，以下の拡張を採用する．

まず,「ポインタをたどることによってあるセルに到達する」ことを表現したい．このためには，不動点演算子を導入した様相 μ 計算を用いればよい．本研究では，抽象化のために必要な計算量を小さくするために，その部分体系である CTL (Computational Tree Logic) を用いる．

次に，セルが「ある性質を持つセルをポインタで指す」だけではなく,「ある性質を持つセルによってポインタで指される」ことも表現したい．このために，様相は順方向と逆方向の 2 方向が指定できるものとする．

さらに，セルが「プログラム変数によって指される」ことを表現する必要がある．このためにはプログラム変数を様相論理の原子論理式と考えればよいのであるが，クリプキ構造上にこの原子論理式を満たす状態は 1 つしかあってはならない．このような制約を持つ原子論理式 ノミナル [1] を導入する．

以上の拡張をした様相論理を，以下，本稿では ノミナル付き 2 方向 CTL (2CTLN) と称する．2CTLN は，命題定数の集合 PC とノミナルの集合 Nom，および様相の集合 Mod を指定することによって定まる．また，各 $m \in \mathrm{Mod}$ に対して $\overline{m} \in \mathrm{Mod}$ が定まっていて，$\overline{\overline{m}} = m$ を満たしているものとする．2CTLN の論理式 φ は，次で与えられる:

$$\varphi ::= p \mid n \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi@n$$

$$\mid \mathbf{E}_m\mathbf{X}\,\varphi \mid \mathbf{E}_m(\varphi\,\mathbf{U}\,\varphi) \mid \mathbf{E}_m(\varphi\,\mathbf{R}\,\varphi)$$

ただし，$p \in$PC, $n \in$ Nom, $m \in$ Mod とする．2CTLN 論理式全体の集合を 2CTLN と書く．

true, false, $\wedge$, $\rightarrow$ 等の一般的な略記は特に断らずに用いる．以下で定義する仕様論理式でも同じ．また，$\mathbf{E}_m(\mathrm{true}\,\mathbf{U}\,\varphi)$ を $\mathbf{E}_m\mathbf{F}\,\varphi$ と略記する．

2CTLN 論理式を解釈するためのクリプキ構造 $\mathcal{K} = (K, R, \lambda)$ は、台集合 K と、遷移関係 $R : \mathrm{Mod} \to K \times K$，ラベル関数 $\lambda : \mathrm{PC} \cup \mathrm{Nom} \to \mathscr{P}(K)$ で定まる．ただし，各 $n \in$ Nom に対して $\lambda(n)$ が単元集合であることと，$m \in$ Mod に対して $R(\overline{m}) = R(m)^{-1}$ であることが要請される．$s \in K$ と $\varphi \in$ 2CTLN に対し，$\mathcal{K}$ において s が φ を満たすという関係 $\mathcal{K}, s \models \varphi$ (略して $s \models \varphi$) を，図 5 のように定義する．これは，通常の CTL における定義を，複数の様相およびノミナルに対応するように拡張したものである．ただし，ここで m-path とは，隣接する要素 s, s' が $(s, s') \in R(m)$ をみたすような K の要素の無限列または有限列で極大なものである．m-path π の i 番目の要素を π_i，添字の集合を $\mathrm{dom}(\pi)$ と記述している．

$s \models p \iff s \in \lambda(p)$.
$s \models n \iff s \in \lambda(n)$.
$s \models \neg\varphi \iff s \not\models \varphi$.
$s \models \varphi_1 \vee \varphi_2 \iff s \models \varphi_1$ または $s \models \varphi_2$.
$s \models \varphi@n \iff s' \models \varphi$. ここに $\lambda(n) = \{s'\}$.
$s \models \mathbf{E}_m\mathbf{X}\,\varphi \iff \exists s'.\,(s, s') \in R(m),\ s' \models \varphi$.
$s \models \mathbf{E}_m(\varphi_1\,\mathbf{U}\,\varphi_2) \iff \exists \pi : m\text{-path}.\,\pi_0 = s,$
$\exists i \in \mathrm{dom}(\pi).\,\pi_i \models \varphi_2, \forall j < i.\,\pi_j \models \varphi_1$.
$s \models \mathbf{E}_m(\varphi_1\,\mathbf{R}\,\varphi_2) \iff \exists \pi : m\text{-path}.\,\pi_0 = s,$
$\forall i \in \mathrm{dom}(\pi).\,\pi_i \not\models \varphi_2 \implies \exists j < i.\,\pi_j \models \varphi_1$.

図 5 2CTLN 意味論

我々の応用では，PC = Val, Nom = Var∪{NULL}, Mod = Fld ∪ $\{\overline{f} \mid f \in \mathrm{Fld}\}$ とする．そして，次の定義によってポインタ構造をクリプキ構造とみなす．

定義 3 ポインタ構造 $\mathcal{S} = (N, p, v, \rho, \mathrm{nil})$ に対応するクリプキ構造とは，$\mathcal{K} = (N, R, \lambda)$ であって，$f \in$ Fld, $d \in$ Val, $x \in$ Var に対して $R(f) = \{(n, p(n, f)) \mid n \in N\}$, $R(\overline{f}) = \{(p(n, f), n) \mid n \in N\}$, $\lambda(d) = \{n \in$

$N \mid v(n) = d\}$, $\lambda(x) = \{\rho(x)\}$, $\lambda(\mathrm{NULL}) = \{\mathrm{nil}\}$ によって定まるものである．$s \in N$ と $\varphi \in$ 2CTLN に対して，$\mathcal{K}, s \models \varphi$ のとき，$\mathcal{S}, s \models \varphi$ と書く． ■

2CTLN 論理式を用いて，ヒープの状態を記述することができる．例えば，$\varphi_1 = (\mathbf{E}_{\mathtt{next}}\mathbf{F}\, y)@x$ は，変数 x によって指されるセルからスタートして `next` フィールドが指す先を追っていくと，変数 y によって指されるセルに到達することを意味している．(具体的には，ある $s \in N$ について $\mathcal{S}, s \models \varphi_1$ となることと，$\mathcal{S}$ がこの性質をもつヒープを表すポインタ構造であることとが同じ意味になる．) また，$\varphi_2 = (\mathbf{E}_{\mathtt{next}}\mathbf{X}\, \mathbf{E}_{\overline{\mathtt{next}}}\mathbf{X}\, y)@x$ は，変数 x によって指されるセルと変数 y によって指されるセルの `next` フィールドが同じセルを指すことを意味している．

しかし，任意の 2CTLN 論理式が，ヒープの状態記述に使えるわけではない．例えば，論理式 $\varphi_3 = x \vee y$ を考えると，$\mathcal{S}, s \models \varphi_3$ が成り立つかどうかは $s \in N$ に依存して変わる．φ_3 は，ヒープ全体の状態を記述しているというよりは，ヒープ内の個々のセルの状態を記述していると解するべきである．一方，論理式 φ_1 については，$s \models \varphi_1$ が成立するか否かは s に依存しない．φ_2 も同様である．そこで，このような論理式をヒープの状態記述に用いることにする:

定義 4 $\varphi@x$ の形の 2CTLN 論理式，特別な原子論理式 abort，およびこれらをブール結合 (演算子 $\neg$, $\vee$ による結合) して得られる論理式を p-論理式という．ただし，φ は@が現れない[†4] 論理式，x はノミナルである．p-論理式全体の集合を PForm と書く． ■

原子論理式 abort は，プログラムが異常終了したことを表すのに用いる．

$S \in$ PtrStr において p-論理式が成立する，という関係を次のように定める: $S \models \varphi@x \iff S \neq \bot$ かつ ある $s \in N_S$ に対して (したがってすべての $s \in N_S$ に対して) $\mathcal{S}, s \models \varphi@x$. $S \models \mathrm{abort} \iff S = \bot$. ブール結合に関しては，自然に定義を拡張する．

4.2 仕様論理式

PML プログラム P の仕様は，$G \subseteq |\mathrm{CFG}(P)|$ が定める遷移系 $\mathcal{T}(G)$ が満足すべき時相論理を用いて記述することができる．我々は，よく知られた時相論理の体系である LTL (Linear Temporal Logic) をベースにして，仕様記述言語を定義する．$Q \in$ PForm, $g \in G$ とするとき，次のように定められる σ を本稿では仕様論理式と呼ぶ.

$\sigma ::= Q \mid g \mid \neg\sigma \mid \sigma \vee \sigma \mid \sigma\, \mathcal{U}\, \sigma$

ここで，$\mathcal{U}$ は，標準的な LTL の演算子である．仕様論理式についても，標準的な略記を用いる．特に，$\mathrm{true}\, \mathcal{U}\, \sigma$ を $\Diamond\sigma$，$\neg\Diamond\neg\sigma$ を $\Box\sigma$ と書く．

$\mathcal{T}(G)$ における原子論理式の真偽については，$\mathcal{S} \in$ PtrStr, $Q \in$ PForm, $g, g' \in G$ に対し，$(\mathcal{S}, g) \models Q \iff \mathcal{S} \models Q$, $(\mathcal{S}, g) \models g' \iff g = g'$ と定める．

reverse プログラムについては，満たすべき仕様の例として，次のものが考えられる．(1) 開始時に変数 x から到達できるセル c は，終了時に変数 y から到達できる．(2) x がリストを指している場合，開始時に x から到達できるセル c_1 の `next` フィールドが c_2 を指していたら，終了時には c_2 の `next` フィールドは c_1 を指す．(3) プログラムは異常終了しない．(4) while ループは有限回で抜ける．すなわち，プログラムの実行は終了する． これらの仕様を表す論理式を，表 3 に示す．ただし，G としては，図 4 における s,e,a からなる集合をとっている．u, u_1, u_2 は，(reverse には現れない) プログラム変数である．

表 3 仕様論理式の例

仕様	論理式
(1)	$\sigma_1 = \Box(s \wedge (\mathbf{E}_{\mathtt{next}}\mathbf{F}u)@x \wedge \neg\mathrm{NULL}@u \rightarrow \Box(e \rightarrow (\mathbf{E}_{\mathtt{next}}\mathbf{F}u)@y)$
(2)	$\sigma_2 = \Box(s \wedge (\mathbf{E}_{\mathtt{next}}\mathbf{F}\mathrm{NULL})@x \wedge (\mathbf{E}_{\mathtt{next}}\mathbf{F}u_1)@x \wedge (\mathbf{E}_{\mathtt{next}}\mathbf{X}u_2)@u_1 \wedge \neg\mathrm{NULL}@u_2 \rightarrow \Box(e \rightarrow (\mathbf{E}_{\mathtt{next}}\mathbf{X}u_1)@u_2))$
(3)	$\sigma_3 = \Box(s \wedge \neg\mathrm{abort} \rightarrow \Box\neg\mathrm{abort})$
(4)	$\sigma_4 = \Diamond e$

†4 この制限は本質的ではない．制限のない論理式に対して，制限付きの論理式で同値なものが存在する．

5 抽象構造

5.1 述語抽象化

前節までで導入した方法で記述された仕様を PML のプログラム P が満たすという関係は，無限遷移系である $\mathcal{T}(G)$ を用いて定義されている．したがって，定義をそのまま適用したのでは，全数調査による検証を行うことはできない．そこで，述語抽象化の枠組みを用いて $\mathcal{T}(G)$ の抽象化である有限遷移系を作成する．

述語抽象化では，遷移系上にいくつかの述語を設定し，これらの述語 (抽象化述語) の真偽の組み合わせをもって，具体遷移系の状態を抽象化する．抽象遷移は，通常，これらの述語のアクションに関する最弱前条件と充足可能性を用いて決定される．我々の方式では，具体遷移系は $\mathcal{T}(G)$ であり，述語には p-論理式を用いる．したがって，p-論理式のアクションに関する最弱前条件を求める手続きと，p-論理式の充足可能性判定手続きが確立できればよい．

実は，健全な述語抽象化を行うためには，必ずしも最弱前条件を求めることは必要ではなく，次の定義を満たす条件が求められればよい:

定義 5 以下の条件を満たす Action × PForm から PForm への関数 pre を，前条件関数という: 任意の $\mathcal{S}_1, \mathcal{S}_2 \in \mathrm{PtrStr}$, $a \in \mathrm{Action}$, $Q \in \mathrm{PForm}$ に対して，$(\mathcal{S}_1, \mathcal{S}_2) \in [\![a]\!]$，$\mathcal{S}_2 \models Q$ ならば，$\mathcal{S}_1 \models \mathrm{pre}(a, Q)$. ∎

通常の意味での a に関する Q の前条件は，$\mathrm{pre}(a, Q)$ でなく，$\neg\mathrm{pre}(a, \neg Q)$ である．定義 5 を満たす (つまらない) 関数の例としては，$\mathrm{pre}(a, Q) = \mathrm{true}$ がある．しかし，このような関数を用いたのでは，意味のある抽象化はできない．実用的な前条件関数として，(1 方向)CTL に対応するものが提案されている ([10]). これを 2CTLN に拡張することによって，本方式による抽象化に利用可能な関数 pre を作ることができる．(付録参照)

$\vec{a} = (a_1, \ldots, a_n)$ をアクションの有限列とするとき，$\mathrm{pre}(a_1, \ldots, \mathrm{pre}(a_n, Q)\ldots)$ も，同じ記号 $\mathrm{pre}(\vec{a}, Q)$ で表すことにする．

次に充足可能性であるが，健全な述語抽象化のために必要な判定手続きは，以下の定義を満たすものである:

定義 6 PForm から $\{0,1\}$ への関数 sat で，$\mathcal{S} \models Q$ なるポインタ構造 $\mathcal{S}$ が存在するならば $\mathrm{sat}(Q) = 1$ となるものを，充足可能判定関数と呼ぶ. ∎

(つまらない) 例として，$\mathrm{sat}(Q) = 1$ なる恒等関数はこの定義を満たす．抽象化のために実用になる関数については，5.2 節で述べる．

$g_0, \ldots, g_n \in |\mathrm{CFG}(P)|$ に対して，アクションの有限列 $l(g_0, g_1), \ldots, l(g_{n-1}, g_n)$ を $\mathrm{act}(\pi)$ と書く．

定義 7 pre を前条件関数，sat を充足可能判定関数，$Q_1, \ldots, Q_n$ を p-論理式，P を PML のプログラム，$G \subseteq |\mathrm{CFG}(P)|$ を，$\mathrm{path}(G)$ が有限集合となるものとする．pre, sat, $Q_1, \ldots, Q_n$, P, G によって定まる抽象遷移系 $\mathcal{T}^{\mathrm{A}} = (A, \rightarrow_A)$ を以下で定義する: $A = B' \times G$. ここに，$B = \{b \mid b : \{1, \ldots, n\} \rightarrow \{0,1\}\}$, $B' = B \cup \{\bot\}$. $(b_1, g_1) \rightarrow_A (b_2, g_2) \iff \pi \in \mathrm{path}(G)$ が存在して，$\mathrm{sat}(Q(b_1) \wedge \mathrm{pre}(\mathrm{act}(\pi), Q(b_2))) = 1$. ここに，$b \in B$ に対して $Q(b) = \bigwedge_{b(i)=0} \neg Q_i \wedge \bigwedge_{b(i)=1} Q_i$, $Q(\bot) = \mathrm{abort}$. ∎

この抽象化は，仕様論理式に関して健全である．すなわち，次が成り立つ:

定理 8 σ を仕様論理式とするとき，$\mathcal{T}^{\mathrm{A}} \models \sigma$ ならば，$\mathcal{T}(G) \models \sigma$ である. ∎

$\mathcal{T}^{\mathrm{A}}$ は有限遷移系だから，$\mathcal{T}^{\mathrm{A}} \models \sigma$ が成立するかどうかはモデル検査によって確認することができる．これが成立すれば，この定理によって，プログラムが仕様を満たすことが検証できたことになる．

5.2 充足可能性判定

本稿で採用している 2CTLN を含む体系である，ハイブリッド様相 μ 計算の論理式の充足可能性を判定する方法は，すでに知られている ([9]). しかし，その手続きは，交代木オートマトンの決定化などを含む複雑なものであり，我々の知る範囲では実装は報告されていない．

2CTLN に限ることによって，タブロー法によるより効率的な判定手続き ([11]) が得られる．この手続

きは[9] と異なり，健全かつ完全なものではない．しかし，この手続きによって定義される関数 sat は，定義 6 にいう充足可能判定関数の条件を満たしているので，抽象構造の作成に用いることができる．

6 検証の流れ

PML プログラム P が与えられたとき，本方式による検証の流れは，次のようになる．

1. $G \subseteq |\mathrm{CFG}(P)|$ で，$\mathrm{path}(G)$ が有限集合となるものを選択する．たとえば，while ループの先頭をすべて含めればよい．また，仕様論理式に記述したい節はすべて G に含める．
2. 仕様論理式 σ を記述し，$\mathcal{T}(G) \models \sigma$ が，検証したい仕様を表すようにする．
3. 抽象化述語として，有限個の p-論理式 $Q_1, \ldots, Q_n$ を選択する．この際，仕様論理式に原子論理式として現れる p-論理式はすべて含める．
4. P, G, $Q_1, \ldots, Q_n$ から抽象構造 $\mathcal{T}^{\mathrm{A}}$ を作成し，モデル検査によって $\mathcal{T}^{\mathrm{A}} \models \sigma$ であるかどうかを判定する．「Yes」なら，定理 8 によって，$\mathcal{T}(G) \models \sigma$ であり，検証が終わる．
5. 「No」なら，モデル検査によって得られた反例を検討する．それが真の反例 (遷移系 $\mathcal{T}(G)$ においても対応する反例が存在するもの) であれば $\mathcal{T}(G) \not\models \sigma$ であり，検証が終わる．
6. 真の反例でない場合，その反例を生じさせないように p-論理式を追加して，4 に戻る．

上の 4 における抽象構造作成は，本方式において自動化されている．5 の検討および 6 での新しい p-論理式の追加は，(現状では) 検証者が手作業で行うことになる．

7 実装と検証例

7.1 MLAT

前節までに述べた検証方式を，MLAT (Modal Logic Abstraction Tool) というツール (試作版) として実装した．MLAT は，6 節で述べたステップ 1 と 4 を自動化する．ステップ 4 におけるモデル検査ツールには，NuSMV [2] を使用している．

```
%%Decl
  Var x,y,t,u;
  Field next;
  Label start, end;
%%Source
  start:
    y := NULL;
  // _auto1: (ここにラベルが生成される)
    while (!(x == NULL)) {
      t := y;
      y:= x;
      x:= x.next;
      y.next:=t;
    }
  end:
%%Pred
  q1 = x ==> E<next>F (u & !NULL);
    // ノード u は，変数 x からつながっている．
  q2 = y ==> E<next>F (u & !NULL);
    // ノード u は，変数 y からつながっている．
%%Spec
  s1 = [](start && q1 -> [](end -> q2));
    // start で q1 なら，end で q2 (成立)
  s2 = [](end -> q2);
    // end で q2 (成立しない)
  s3 = [](start && !abort -> []!abort)
    // 異常終了しない (成立)
```

図 6 MLAT 入力ファイル例

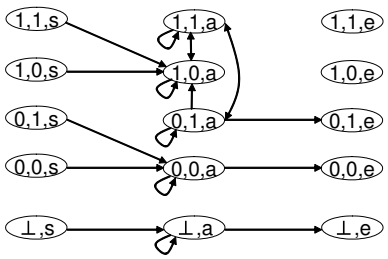

図 7 MLAT が生成した抽象遷移

入力ファイル例を図 6 に示す．最初の節 Decl は，変数などの宣言，次の節 Source は，PML プログラムである．仕様論理式に記述する制御流れ図の節位置を指定するために，ラベルを置けるようになっている．この例では `start` と `end` の 2 つのラベルが置かれている．Pred 節には，抽象化述語を記述する．たとえば，`q1` は，$(\mathbf{E}_{\mathtt{next}}\mathbf{F}\,(u \wedge \neg\mathrm{NULL}))@x$ を表している．Spec 節には，仕様論理式を記述する．`s1` は表 3 の σ_1 と，`s3` は σ_3 とそれぞれ同値な仕様論理式を表している．

この入力に対し，MLAT は $G \subseteq |\mathrm{CFG}(P)|$ とし

て，開始点，終了点，while ループの先頭，およびラベルがおかれた位置を選択し，抽象化述語としては Pred 節に与えられた述語を用いて抽象遷移系を作成する．作成された抽象遷移系をグラフ状に表したものを図 7 に示す．例えばノードの $(1,0,s)$ は，`q1` が真，`q2` が偽という組み合わせであり，プログラムの実行位置は s であることを表している．たとえば `s1` が成り立つことは，$(1,1,s)$ または $(1,0,s)$ から出発して，$(1,0,e)$ または $(0,0,e)$ に至るパスがないことからわかる．

7.2 検証例

4.2 節にあげた仕様の (1),(2),(3) について，MLAT を用いて検証することができた．

(1) と (3) については，図 6 に示したソースで検証が行える．つまり，抽象化述語としては仕様論理式に現れる p-論理式のみを採用して，検証することができる．MLAT は，抽象遷移系の生成および NuSMV の呼出によるモデル検査まで含めて，約 1.2 秒で処理を終了している．(Pentium 4, 2.4GHz, 512MB メモリ，Windows XP)

(2) については，抽象化述語として仕様論理式に現れる p-論理式のみを用いた場合には，モデル検査において反例が生じてしまう．もちろん，これは真の反例ではない．検証者が反例を解析するなどして述語 $u_2@x$, $u_1@y$, $(\mathbf{E}_n\mathbf{F}u_2)@y$ を加えた結果，MLAT は仕様論理式が成り立つことを検証できた．MLAT の実行に要した時間は，同じ環境で約 37 秒である．

8 実用化への課題

前節で述べた実装によって，本研究で提案した方式で実際に検証が行えることが確認された．この節では，この方式を実用化するために残されている研究課題について述べる．

まず，検証の自動化が課題としてあげられる．現状では，多くの場合に自動的に検証を行うことはできず，検証者による判断が必要になる．どの仕様にも適用できる完全な自動化を達成することは原理的に不可能であるが，より多くの問題に対して自動検証が行えることが望ましい．

検証者の判断は，以下の 2 点で必要である．(1) 反例の検討: モデル検査によって反例が得られたとき，それが真の反例であるかどうか調べる必要がある．反例から，実行経路および各時点における抽象化述語の真偽を抽出し，具体遷移系において，そのような遷移が可能であるかどうかを判定することになる．(2) 述語の選択: 最初の述語の選択および，モデル検査による反例が真の反例では無い場合に追加する述語の選択は，検証者の判断による．

述語抽象化において上記 2 点を自動化する手法に，反例に基づく詳細化 [3] と呼ばれるものがある．この手法は，モデル検査によって得られた反例のパスを逆方向にたどりながら最弱前条件をもとめていき，矛盾に到達するかどうかで，真の反例であるかどうかを判定する．さらに真の反例でない場合，矛盾への到達状況から，その反例を除去する述語を生成することも行う．本方式にこの手法をそのまま適用すると，反例解析と述語生成が無限に繰り返されてしまい，検証に至らないことが多い．適当な近似を行うことで，有限回で必要な述語が得られる方法を見いだすことが課題となる．

もう一つの課題は，効率である．現在使用している充足可能性判定手続きは，抽象化述語の数が増大するとともに，急速に必要な計算量が増大する．(充足可能性判定問題の複雑さは EXPTIME-完全である.) 現状では，reverse における仕様 (2) 程度が限界であり，これ以上述語数を増やすと，現実的な時間内では検証が終了しない．これを解決するために，充足可能性判定手続きのさらなる効率化とともに，導入した述語をプログラムの全範囲で用いるのではなく，必要なところだけで用いるようにすることによって，抽象遷移系のサイズを抑える方式が有望であると考え，検討中である．

最後に，活性性質の検証方法の確立があげられる．検証項目を大きく分類すると,「常にこの性質が成り立つ」という安全性と,「いつかはこの性質が成り立つ」という活性に分けられる．4.2 節であげた例では，(1),(2),(3) が安全性であり，(4) は活性である．安全性の検証は，本方式をはじめとする述語抽象化の手法になじむが，活性性質の検証は，難しい場合が多い．

近年研究されている遷移述語抽象化 [5] などの手法を用いることによって，活性検証が可能になる場合があると考えており，検討を行っている．

9 おわりに

本研究では，述語抽象化の枠組みを用いて，ヒープを扱うプログラムの検証を行う方法を提案し，その方式に基づいた試作実装を行い，プログラムの安全性に関する仕様が検証できることを示した．今後は，自動検証の範囲を広げることや，活性性質の検証など，実用化に向けての課題の解決を行っていく計画である．

参考文献

[1] Blackburn, P., de Rijke, M., and Venema, Y.: *Modal Logic*, Cambridge University Press, 2001.

[2] Cimatti, A., Clarke, E., Giunchiglia, F., and Roveri, M.: NuSMV: a new Symbolic Model Verifier, *Proceedings Eleventh Conference on Computer-Aided Verification (CAV'99)*, Lecture Notes in Computer Science, No. 1633, Trento, Italy, Springer, July 1999, pp. 495–499.

[3] Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., and Veith, H.: Counterexample-Guided Abstraction Refinement, *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, London, UK, Springer-Verlag, 2000, pp. 154–169.

[4] Hagiya, M., Takahashi, K., Yamamoto, M., and Sato, T.: Analysis of Synchronous and Asynchronous Cellular Automata using Abstraction by Temporal Logic, *FLOPS2004: The Seventh Functional and Logic Programming Symposium*, Lecture Notes in Computer Science, Vol. 2998, 2004, pp. 7–21.

[5] Podelski, A. and Rybalchenko, A.: Transition predicate abstraction and fair termination, *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, ACM Press, 2005, pp. 132–144.

[6] Reynolds, J. C.: Separation Logic: a Logic for Shared Mutable Data Structures, *Proceedings of the 17th IEEE Symposium on Logic in Computer Science*, 2002, pp. 55–74.

[7] S. Graf and H. Saidi: Construction of Abstract State Graphs with PVS, *Proc. 9th INternational Conference on Computer Aided Verification (CAV'97)*, Lecture Notes in Computer Science, Vol. 1254, Springer Verlag, 1997, pp. 72–83.

[8] Sagiv, M., Reps, T., and Wilhelm, R.: Parametric Shape Analysis via 3-valued Logic, *ACM Transactions on Programming Languages and Systems*, Vol. 24,No. 3(2002), pp. 217–298.

[9] Sattler, U. and Vardi, M. Y.: The Hybrid μ-Calculus, *Proceedings of the International Joint Conference on Automated Reasoning*, LNCS, Vol. 2083, Springer Verlag, 2001, pp. 76–91.

[10] Tanabe, Y., Takai, T., Sekizawa, T., and Takahashi, K.: Preconditions of properties described in CTL for statements manipulating pointers, *Supplemental Volume of the 2005 International Conference on Dependable Systems and Networks (DSN-2005)*, 2005, pp. 228–234.

[11] 湯浅能史, 田辺良則, 関澤俊弦, 高橋孝一: 時相論理による述語抽象化のための充足可能性判定手続き, 日本ソフトウェア科学会第 *21* 回大会, 2005.

付録: 最弱前条件

定義 5 をみたす関数 pre を定義する．これは，[10] で導入したものを，2 方向の様相に対応するように拡張した関数である．

本質的に定めなければならないのは $\mathrm{pre}(a, \mathrm{abort})$ と $\mathrm{pre}(a, \varphi@x)$ である．これらを定義すれば，$\mathrm{pre}(a, \neg Q) = \neg\mathrm{pre}(a, Q)$ および $\mathrm{pre}(a, Q_1 \vee Q_2) = \mathrm{pre}(a, Q_1) \vee \mathrm{pre}(a, Q_2)$ を用いて全体に拡張できる．

まず，$c \in \mathrm{Cond}$ に対しては，表 4 の p_1 と p_2 を用いて，$\mathrm{pre}(c, \mathrm{abort}) = p_1(c)$，$\mathrm{pre}(c, \varphi@x) = \neg p_1(c) \wedge p_2(c) \wedge \varphi@x$ と定める．

原子プログラム P に対する $\mathrm{pre}(P, \mathrm{abort})$ と，$x\texttt{.}f\texttt{:=}y\texttt{;}$ 以外の原子プログラム P に対する $\mathrm{pre}(P, \varphi@z)$ は，表 5 のように定める．ただし，表中の $p_3(x, \varphi)$ は，φ の構成に関する帰納法により，表 6 で定義されるものである．残った $p_2(x\texttt{.}f\texttt{:=}y, \varphi@z)$ の定義は次のようになる．まず，C を φ の部分論理式で主論理記号が $\mathbf{E}_m\mathbf{X}$, $\mathbf{E}_m\,\mathbf{U}$, $\mathbf{E}_m\,\mathbf{R}$ のいずれか (ただし，$m = f$ または $m = \overline{f}$) であるものの全体とする．H を C から $\{0, 1\}$ への関数全体の集合とす

表 4 Cond **に関する前条件**

c	$p_1(c)$	$p_2(c)$
x`==NULL`	false	nil@x
x`==`y	false	y@x
x`.val==`d	nil@x	d@x
`!`c	$p_1(c)$	$\neg p_2(c)$
c_1`||`c_2	$p_1(c_1) \vee p_1(c_2)$	$p_2(c_1) \vee p_2(c_2)$

る．そこで，$p_2(x.f\texttt{:=}y, \varphi@z) = \bigvee_{h\in H}(p_4(\varphi,h)@z \wedge \bigwedge_{\psi\in C} p_5(\psi,h)@p_6(\psi))$ と定義する．ここで，p_4, p_5, p_6 は，表 7 で定義されるものである．なお，様相 f については，$\mathbf{E}_f(\varphi_1 \mathbf{R} \varphi_2) \equiv \neg\mathbf{E}_f(\neg\varphi_1 \mathbf{U} \neg\varphi_2)$ である (前向き様相に関しては枝分かれが起こらないため) ことを利用して，あらかじめ $\mathbf{E}_f \mathbf{R}$ が φ に現れないようにしておく．

[10] と同様にして，次を示すことができる:

定理 9 関数 pre は，前条件関数である．実際，任意の $\mathcal{S}_1, \mathcal{S}_2 \in \mathrm{PtrStr}$, $a \in \mathrm{Action}$, $Q \in \mathrm{PForm}$ に対して $(\mathcal{S}_1, \mathcal{S}_2) \in [\![a]\!]$ ならば，$\mathcal{S}_2 \models Q$ と $\mathcal{S}_1 \models \mathrm{pre}(a, Q)$ は同値になる． ∎

これからわかるように，$\mathrm{pre}(a, Q)$ は，Q の a に関する通常の意味の最弱前条件になっている．

表 5 AProg に関する前条件

P	$p_1(P)$	$p_2(P, \varphi@z)$
`skip;`	false	$\varphi@z$
`abort;`	true	false
x`:=NULL;`	false	$(\varphi@z)[\mathrm{NULL}/x]$
x`:=`y`;`	false	$(\varphi@z)[y/x]$
x`:=`y`.`f`;`	nil@y	$\varphi[\mathbf{E}_{\overline{f}}\mathbf{X}\, y/x]@z \quad (x \neq z)$ $(\mathbf{E}_f\mathbf{X}\varphi[\mathbf{E}_{\overline{f}}\mathbf{X}y/x])@y \quad (x = z)$
x`.val:=`d`;`	nil@x	$\varphi[d \vee x/d]$ $[d' \wedge \neg x/d']_{d'\in \mathrm{Val}\setminus\{d\}}@z$
x`.`f`:=`y`;`	nil@x	(本文参照)
x`:=new();`	false	$\varphi[\mathrm{false}/x]@z \quad (x \neq z)$ $p_3(x, \varphi) \quad (x = z)$

$\varphi[\xi/\psi]$ は，φ に現れる ψ を ξ で置き換えることを表す．

表 6 `new()` に関する論理式

φ	$p_3(x, \varphi)$
NULL	false
$y \in \mathrm{Var}$	false $\quad (x \neq y)$ true $\quad (x = y)$
$d \in \mathrm{Val}$	false $\quad (d \neq d_0)$ true $\quad (d = d_0)$
$\neg\psi$	$\neg p_3(x, \psi)$
$\psi_1 \vee \psi_2$	$p_3(x, \psi_1) \vee p_3(x, \psi_2)$
$\mathbf{E}_f\mathbf{X}\,\psi$	$p_3(x, \psi)$
$\mathbf{E}_f(\psi_1 \mathbf{U} \psi_2)\psi$	$p_3(x, \psi_2)$
$\mathbf{E}_f(\psi_1 \mathbf{R} \psi_2)\psi$	$p_3(x, \psi_2)$

表 7　p_4, p_5, p_6 の定義

φ	$h(\varphi)$	$p_4(\varphi, h)$	$p_5(\varphi, h)$	$p_6(\varphi)$
原子論理式	*	φ	—	—
$\neg\varphi_1$	*	$\neg\chi_1$	—	—
$\varphi_1 \vee \varphi_2$	*	$\chi_1 \vee \chi_2$	—	—
$\mathbf{E}_g\mathbf{X}\,\varphi_1$	*	$\mathbf{E}_g\mathbf{X}\,\chi_1$	—	—
$\mathbf{E}_f\mathbf{X}\,\varphi_1$	0	$\neg x \wedge \mathbf{E}_f\mathbf{X}\,\chi_1$	$\neg\chi_1$	y
	1	$x \vee \mathbf{E}_f\mathbf{X}\,\chi_1$	χ_1	
$\mathbf{E}_{\overline{f}}\mathbf{X}\,\varphi_1$	0	$\mathbf{E}_{\overline{f}}\mathbf{X}\,\chi_1$	$\neg\chi_1$	x
	1	$y \vee \mathbf{E}_{\overline{f}}\mathbf{X}\,(\neg x \wedge \chi_1)$	χ_1	
$\mathbf{E}_g(\varphi_1\,\mathbf{U}\,\varphi_2)$	*	$\mathbf{E}_g(\chi_1\,\mathbf{U}\,\chi_2)$	—	—
$\mathbf{E}_f(\varphi_1\,\mathbf{U}\,\varphi_2)$	0	$\mathbf{E}_f((\chi_1 \wedge \neg x)\,\mathbf{U}\,\chi_2)$	$\neg\mathbf{E}_f((\chi_1 \wedge \neg x)\,\mathbf{U}\,\chi_2)$	y
	1	$\mathbf{E}_f(\chi_1\,\mathbf{U}\,((x \wedge \chi_1) \vee \chi_2))$	$\mathbf{E}_f((\chi_1 \wedge \neg x)\,\mathbf{U}\,\chi_2)$	
$\mathbf{E}_{\overline{f}}(\varphi_1\,\mathbf{U}\,\varphi_2)$	0	$\mathbf{E}_{\overline{f}}(\chi_1\,\mathbf{U}\,\chi_2)$	$\neg\mathbf{E}_{\overline{f}}(\chi_1\,\mathbf{U}\,\chi_2)$	x
	1	$x \vee \mathbf{E}_{\overline{f}}[(\neg x \wedge \chi_1)\,\mathbf{U}\,(\neg x \wedge (\chi_2 \vee (y \wedge \chi_1)))]$	$\mathbf{E}_{\overline{f}}(\chi_1\,\mathbf{U}\,\chi_2)$	
$\mathbf{E}_g(\varphi_1\,\mathbf{R}\,\varphi_2)$	*	$\mathbf{E}_g(\chi_1\,\mathbf{R}\,\chi_2)$	—	—
$\mathbf{E}_f(\varphi_1\,\mathbf{R}\,\varphi_2)$	*	(本文参照)		
$\mathbf{E}_{\overline{f}}(\varphi_1\,\mathbf{R}\,\varphi_2)$	0	$q_1(\chi_1, \chi_2, x, y)$	$\neg q_2(\chi_1, \chi_2, x, y)$	x
	1	$q_1(\chi_1, \chi_2, \text{false}, \text{false})$	$q_2(\chi_1, \chi_2, x, y)$	

*は，値に関係ないことを示す.

$g \neq f$, $g \neq \overline{f}$, $\chi_1 = p_4(\varphi_1, h)$, $\chi_2 = p_4(\varphi_2, h)$.

$q_1(\chi_1, \chi_2, w_1, w_2) = w_1 \vee \mathbf{E}_{\overline{f}}[\{w_2 \vee \chi_1 \vee (\mathbf{E}_{\overline{f}}\mathbf{X}\,v_1 \wedge \neg\mathbf{E}_{\overline{f}}\mathbf{X}\,\neg v_1)\}\,\mathbf{R}\,\{\chi_2 \wedge \neg v_1\}]$.

$q_2(\chi_1, \chi_2, w_1, w_2) = \chi_2 \wedge [\chi_1 \vee w_2 \vee \neg\mathbf{E}_{\overline{f}}\mathbf{X}\,\text{true} \vee \mathbf{E}_{\overline{f}}\mathbf{X}\,q_1(\chi_1, \chi_2, \text{false}, w_2)]$.