日本ソフトウェア科学会第 27 回大会 (2010 年度) 講演論文集

ヒープ中のオブジェクトの状態に関するモデル検査器

松田 元彦　前田 俊行　米澤 明憲

モデル検査器を使って，デバイスドライバ等の排他ロック等に対する呼出し整合性の検査を行う方法が提案されている．しかし，デバイスドライバではヒープ中にロックが置かれることがあるので，ヒープに関する記述を扱う必要がある．そのため，モデル検査器に Separation Logic の部分言語に対して提案されている決定手続きを導入した．具体的には，決定手続きをモデル検査器で使用する SMT ソルバーに組込むとともに，リスト構造を走査するループの検査を容易にするため，プログラマが挿入するディレクティブをいくつか定義した．ロック等に関する検査では，ヒープ自体に関する制約は自明であるが，ヒープ中の一つのオブジェクトの状態遷移を追跡する必要がある．大域的な状態の追跡はモデル検査器が得意な部分であるが，それをヒープ中のオブジェクトにも適用できるようになった．そのモデル検査器の設計と実装について述べる．

1　はじめに

モデル検査器の典型的な利用に，カーネル・モジュール内の排他ロックや実行コンテキストに関する整合性の検査がある．排他ロックについては，ロックに続けてアンロックを呼び出すという呼出し順序の整合性を検査する．実行コンテキストについては，割込み処理に課せられる制限等を検査する．例えば，カーネル内では割込み処理中はスリープできない．そのため，メモリ割当てルーチン等の呼出しに対して実行コンテキストに合った特定のフラグを渡すといった整合性を検査する必要がある．

モデル検査器により，大域的な排他ロックや実行コンテキストに関する検査は容易に行える．しかし，カーネル内ではロック等はリストに繋がれた構造体の一部としてヒープに置かれることも多い．図 1 にヒープ中にあるロックの例を示す．例では，リストの要素 p に対してロックした状態で本体の処理 $g(p)$ を行う．ここで検査したい性質は，全てのロックがロックされていないという事前条件のもと，要素 p のロックに対してロック／アンロックの順に呼出しを行うことである．検査はリストの一つの要素に注目しその使われ方を追跡するという簡単なものであるが，モデル検査器でヒープを扱う必要が出てくる．

そこで開発中のモデル検査器 MKencha/ng [18] に対して，ヒープを扱うために Separation Logic [20] の部分言語をアサーション言語として導入する．扱うのは簡単なリスト構造である．

既存研究により Separation Logic の決定可能な部分言語とその決定手続きが示されている [4]．ヒープを扱う既存研究の多くは，関数毎に性質を仕様として与えるモジュラーな検査である．しかし既存のコードを検査するには，基本的に関数毎の仕様を必要としないモデル検査が適している．決定可能な論理があるならば，モデル検査でも利用できるはずである．述語抽象と抽象実行に基づく検査を行うモデル検査器では，論理式の充足性によって状態遷移を計算する．その遷移計算には，自動証明器の一種である SMT ソ

Model Checking State Changes of an Object in Heap
Motohiko Matsuda, 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.
Toshiyuki Maeda, 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.
Akinori Yonezawa, 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.

```
struct S { struct S *next; int lock; };
struct S *root;

void lock(struct S *p) {
    $assert(p->lock == 0); p->lock = 1;
}
void unlock(struct S *p) {
    $assert(p->lock != 0); p->lock = 0;
}
void f() {
    struct S *p = root;
    do {
        lock(p); g(p); unlock(p);
        p = p->next;
    } while (p != root);
}
```

図 1 動機付けとなるコード例

ルバー（Satisfiability Modulo Theories）[22] を使用する．そこで，既存研究の決定手続きを使ったヒープに関する充足性の計算を SMT ソルバーに導入することを考える．

しかし，既存の決定手続きはモデル検査で使用するには言語が強く制限され過ぎている．ただし，モデル検査器での SMT ソルバーの使われ方を見ると，検査の健全性を損なわないように状態遷移を行うにはその遷移計算は過大近似であればよい．その点を考慮して，ソルバーにヒープの扱いを導入する．

ヒープを含む状態遷移を SMT ソルバーで計算できても，リスト等の再帰的構造を扱うループが現れる問題がある．また，リストの要素を表すために限量子を含む式が現れる．それらを SMT ソルバーで解くのは難しい．そのため，ディレクティブにより簡単な変形を指示することにする．例えば，リストの要素とプログラムの変数と対応付ける変換を指示する．変形は再帰構造に対する展開とその逆の変形であり，メタに導出できる特定の推論規則を用いる．

以下では，ヒープに関する充足性計算を SMT ソルバーへ組込んだ，モデル検査器 MKencha/ng の設計と実装について述べる．その開発目標はできるだけ健全性のある検査を行うことである．特に，ループや再帰呼出しについて繰り返し回数を制限しない非有界なモデル検査を行っている．その代り，CEGAR [11] などを使った述語セットのリファインメント等は行なわない．

2 節で簡単にモデル検査と Separation Logic について紹介した後，設計と実装について述べる．3 節で決定手続きの SMT ソルバーへの導入，4 節でリストに対するループの変換ついて述べる．その後，5 節で問題点と今後の課題，6 節で関連研究，7 節でまとめを順に述べる．

2 基本事項

2.1 モデル検査の動作

ここでは，以降の節の理解に必要なモデル検査器の基本動作について述べる．開発しているモデル検査器は，SLAM [2]，BLAST [16] といったツールと同様，述語抽象と抽象実行に基づく [12] [15].

述語抽象の抽象状態は，与えられた述語セット $\{P_i\}$ から正負のリテラル（P_i あるいは $\neg P_i$，それらをまとめて $\pm P_i$ と記述）を作りそれらの連言で表現する．

$$\varphi = \bigwedge \pm P_i$$

プログラムの各点で取り得る抽象状態のセット $\{\varphi_j\}$ は，論理的には選言で結合されていると解釈する．

状態遷移は，論理式の充足性によって計算する．抽象状態 φ_0 から抽象状態 φ_1 への遷移は次式が充足可能な場合に可能であるとする．

$$\mathcal{M} \models \varphi_0 \wedge C \wedge \varphi_1$$

ここで，C はプログラムの文を変換した制約式である．C には代入文などを等式に変換したものを用いる．充足性の検査であるので，自由変数には暗黙に限量子 $\exists$ を仮定する．

モデル検査器は，初期状態からプログラムのパスに従って状態遷移を繰り返し，プログラムの各点での抽象状態を求める．そして，各状態が与えられたアサーションを満たしているかを検査する．

2.2 Separation Logic

Separation Logic [20] は，Hoare 論理でヒープを扱うために拡張されたものである．ここで，その記法と意味を簡単に紹介する．図 2 に使用する Separation Logic の部分言語とその意味を示す．

emp は空のヒープを表す．$p \mapsto m$ は要素が一つのヒープを表す．これはポインタ p と指されるデータ

$\mathcal{H} \models \text{emp} \iff |\mathcal{H}| = \{\}$ （$\mathcal{H}$ のドメインが空）
$\mathcal{H} \models p{\mapsto}m \iff |\mathcal{H}| = \{p\}$ （$\mathcal{H}$ のドメインが p のみを含む）
$\mathcal{H} \models \sigma_0 * \sigma_1 \iff \mathcal{H}_0 \models \sigma_0$ かつ $\mathcal{H}_1 \models \sigma_1$ （ただし $\mathcal{H}_0$ と $\mathcal{H}_1$ は $\mathcal{H}$ のドメインを互いに交わりのないように分割）
$\mathcal{H} \models \sigma_0 \wedge \sigma_1 \iff \mathcal{H} \models \sigma_0$ かつ $\mathcal{H} \models \sigma_1$
$ls_{next}(p,q) \iff (p = q \wedge \text{emp}) \vee (p \neq q \wedge \exists m.(p{\mapsto}m * ls_{next}(m.next, q)))$

図 2 Separation Logic の部分論理の要素と意味

m の関係（point-to）である．ヒープは，以下のようにヒープの要素を並べて表記する．

$$p_0{\mapsto}m_0 * p_1{\mapsto}m_1 * p_2{\mapsto}m_2$$

$*$ は separating conjunction と呼ばれ，ポインタ群が互いに異なることを含意する．つまりこの場合の表記は，エイリアスのない三つのポインタからなるヒープを表現している．

さらに，$\wedge$ を使ってヒープの関係を示す．$\wedge$ は左辺／右辺のヒープが同じ要素からなることを意味する．例えば，エイリアスは $\wedge$ を使って表現できる．$(p_0{\mapsto}m_0 \wedge q_0{\mapsto}m_0) * p_1{\mapsto}m_1$ では，p_0 と q_0 は同じところを指していることを表す．

$ls_{next}(p,q)$ はリスト・セグメントを表す．$ls_{next}(p,q)$ は，$next$ フィールドで繋がれた p から q までの部分リスト（q を含まない）を表す．ここでリストの要素であるポインタはなんらかのデータを指していると解釈する．

決定手続きに直接関係しないので図 2 にないが，$cls_{next}(p)$ は循環リストを表す．$cls_{next}(p)$ は，p から $next$ フィールドで繋がれた空リストではないリスト全体を表す．

モデル検査では emp，$\mapsto$，$*$，$ls_{next}(p,q)$，$cls_{next}(p)$ と $\wedge$ を用いる．ここで特に，Separation Logic で任意のヒープを表す true は除外されている．

2.3 ヒープ表現と抽象実行

モデル検査器 MKencha/ng では，ヒープ表現と述語論理の式をペアで扱う．状態表現は，

$$(\mathcal{M}, \mathcal{H}) \models (\varphi \wedge \sigma)$$

である．φ は述語論理でリテラルの連言からなる式，σ は Separation Logic の式である．連言（$\wedge$）が用いられるが述語論理のものではない．これによって状態 $(\varphi \wedge \sigma)$ が抽象実行の状態空間に含まれることを表す．状態表現だけでなくアサーションなどでも，ヒープ表現は正項かつ述語論理式に対して連言でしか現れない．以下では，文字 φ をリテラルの連言，文字 σ をヒープ表現，文字 ψ を述語論理の式に当てることにする．

状態遷移については，$(\varphi_0 \wedge \sigma_0)$ から $(\varphi_1 \wedge \sigma_1)$ への遷移を調べる場合，

$$(\mathcal{M}, \mathcal{H}) \models (\varphi_0 \wedge C \wedge \varphi_1) \wedge (\sigma_0 \wedge \sigma_1)$$

の充足性を調べる．一方，アサーションについては，$(\varphi_0 \wedge \sigma_0)$ の状態で $(\sigma_1 \wedge \psi)$ が恒真であることを調べるには，

$$(\mathcal{M}, \mathcal{H}) \models ((\varphi_0 \wedge C) \vee \neg\psi) \wedge (\sigma_0 \wedge \sigma_1)$$

の非充足性を調べる．ここで，C はプログラムの文を変換した制約式である．ヒープに関する含意は，ヒープが等価であると解釈している．

状態遷移もアサーションもヒープ表現と述語論理の充足性を調べるが，SMT ソルバーで充足性を解く前に，まず，述語論理の式に現れるポインタ参照の値の解決を行う．ヒープ表現からポインタ参照の指す値を取出し述語論理の式中のポインタ参照を置換して簡約化する．その後，SMT ソルバーを使ってヒープ表現と述語論理の充足性を検査する．

SMT ソルバーによる充足性の検査では，まず，ヒープの充足性に必要な制約 C_{heap} を生成する．そして次に，その制約を述語論理の式に付け加えて述語論理の充足性を調べる．つまり，$(\mathcal{M}, \mathcal{H}) \models (\psi \wedge \sigma)$ の充足性は，$\mathcal{M} \models \psi \wedge C_{heap}$ と $\mathcal{H} \models \sigma$ の充足性の検査になる．

3 ソルバー

3.1 SMT ソルバー

モデル検査では，論理式の充足性の検査には SMT ソルバーが使用される．SMT ソルバー（Satisfiability Modulo Theories）[22] は自動定理証明器の一種であり，複数の論理の決定手続きを組合わせたものを指すものとする．一般のプログラム検証では恒真性を用いるが，モデル検査では充足性によって遷移の計算を行う．特にモデル検査では，論理が完全でない場合，検査に健全性を持たせるため遷移計算は過大近似（overapproximation）になっている．SMT ソルバーには実際より広い充足性を持つ（完全性はあり健全性はない）ものを用いるのは許される．

ベースとなる SMT ソルバーは，命題論理に対する SAT ソルバー，等式に対する Congruence Closure，整数制約に対する Cooper アルゴリズム，を組み合わせる．組み合わせには Nelson-Oppen の組合わせアルゴリズムを用いる [9] [13] [17].

ベースとなる SMT ソルバーにヒープに関する制約生成を組み込むが，ヒープに関する制約生成は等式の制約を生成するのでベースに使う SMT ソルバーに制限はない．

3.2 決定手続きの組込み

図 3 に Berdine らによる決定手続き [4] で用いられる展開式を示す（[4] では UnrollCollape と呼ばれる推論規則として提示されている）．σ は任意のヒープ表現を表す．彼らは証明論とモデル論の二つの決定手続きを与えているが，そのうちモデル側の決定手続きを利用する．

その決定手続きは，ヒープのリストセグメント表記（ls_{next}）に対して，長さ 0 と 2 の有限個の要素からなるモデルだけを考慮すれば決定手続きになるというものである．図 3 の展開式は，ls_{next} を長さ 0 と長さ 2 に場合分けしている．この展開を繰り返すと ls_{next} が一つずつ置換えられるので，生成される制約はヒープの前提条件が point-to 関係のみだけになる．この新しい前提条件は，正確なヒープの要素が与えられていると見なして良いことを意味する．

$$\models e_0 = e_1 \wedge \sigma \supset \sigma' \text{ かつ}$$
$$\models e_0 \neq e_1 \wedge (e_0 \mapsto x * x.next \mapsto e_1 * \sigma) \supset \sigma'$$
$$\Longleftrightarrow \models ls_{next}(e_0, e_1) * \sigma \supset \sigma'$$

図 3 決定手続きによる展開式

ここで，決定手続きの SMT ソルバーへの組込みについて注意点がある．決定手続きを，実際には決定手続きではなく制約条件の生成とみなしている点である．モデル検査器に必要なソルバーは過大近似であれば良く，決定手続きである必要はない．ソルバー内部では Nelson-Oppen 組合わせによって複数の決定手続きを組合わせている．しかし Nelson-Oppen 組合わせの正しさは，interpolation 定理に依存している．一方，リスト等は推移閉包を含み一階述語論理の範囲でなく，一階の等号について interpolation 定理の成立は自明ではない．だだし，決定手続きに現れる展開式を制約式として利用するのであれば，ソルバーは過大近似になるので問題はなくなる．また，生成される制約は等号／不等号で表現されている．

Berdine らの提案するヒープに対する決定手続きは言語の制約が強く，ヒープに許されるのは変数と等号だけになっている．しかし，この展開式は変数でなくても使える．

3.3 ヒープ表現の充足性検査

Separation Logic の含意は，前提と結論の二つのヒープのドメインが同一であることである．Berdine らによる決定手続きでは，前提から ls_{next} を除く変形を行った後，前提と結論のヒープのドメインの比較を行う．彼らの場合，言語が制限されており変数間の等号しか現れないので，順に検査できるとしている．

MKencha/ng では結論側の ls_{next} も前提側と同じ ls_{next} に由来する．そのため実際には，結論側の ls_{next} も展開回数の上限も分かるので，ヒープのドメインの比較を少し単純化できる．ヒープのドメインの比較は，結論側の ls_{next} も展開し等式の制約への変換している．展開回数の上限が分かっているので，ヒープ表現から全てのポインタ群を取り出しそれぞれが一致する制約式を作ることができる．

二つのポインタの集合 $\{p_i\}$ と $\{q_j\}$ の比較には全

ての組合わせを作る必要がある．そのため，全ての順列 $perm$ を生成して比較する：

$$\bigvee_{perm} \bigwedge_{i} (p_i = q_{perm(i)})$$

すべてのポインタに関してそれぞれ制約式を生成する．ここで，全ての組合わせを生成するのは制約式が大きくなりそうであるが，実際に現れる項については前提と結論で同一の変数の出現が多い．それらについては制約を生成する必要はないので，組合わせ数はそれほど大きくならない．

3.4 述語の選択による状態の削減

既に述べたように，MKencha/ng は CEGAR [11] などを使った述語セットのリファインメントを行わない．その代わり，抽象状態表現に使う述語セットの中から，関数内の状態に関係のない述語を関数呼出しの時点で分離する．それによって実際に計算する必要がある状態数を削減する．Hoare 論理と Separation Logic にはフレーム・ルールがあり，述語の選択はそれに相当する．

これは，リファインメント自体に実行時間がかかることと，検査毎にリファインメントを繰返すのは無駄という判断である．また，BLAST などではリファインメントのために関数のインライン展開やループのアンロールを行なっている．このため，再帰関数やループを固定回数で打ち切っているので，健全性がなくなっている．

リファインメントの目的は，できるだけ小さな述語セットの抽出にあるが，実用上はその必要はない．述語の充足性を解く問題は組合わせ問題であり，SMT ソルバーが用いるアルゴリズムの多くは指数的（それ以上）の振る舞いをする．しかし，一般的なプログラムであれば関数に関係する述語数は 10 程度だと考えられる．その程度の数ならば実用時間で問題を解くことができる．

4 リスト要素に対するループとディレクティブ

4.1 ヒープ表現のコード例

1 節で示した図 1 のコード例に戻る．検査したい性質は，要素 p に含まれるロックをロック／アンロックの順に行うことである．関数呼出しの事前条件は，全てのロックがロックされていないという仮定である．

$$cls_{next}(root) \wedge \forall p.(p \in cls_{next}(root) \supset p\text{->lock} = 0)$$

ここでは，アサーションにヒープのドメインを集合のように扱う記述を使った．

`$roll_cls`$_{next}$`(root,p)` と `$unroll_cls`$_{next}$`(root,p)`：

$cls_{next}(root)$

$\Longleftrightarrow ls_{next}(root, p) * p \mapsto m * ls_{next}(m.next, root)$

`$unfold_ls`$_{next}$`(p)`：

$ls_{next}(p, q)$

$\Longleftrightarrow (p = q \wedge \mathrm{emp}) \vee (p \neq q \wedge p \mapsto m * ls_{next}(m.next, q))$

図 4 ディレクティブとヒープ表現の変換

```
void f(struct S *root) {
    struct S *p = root;
    do {
        $unroll_cls_next(root,p);
        lock(p); g(p); unlock(p);
        $unfold_ls_next(p->next);
        p = p->next;
        $roll_cls_next(root,p);
    } while (p != root);
}
```

図 5 ディレクティブの挿入されたコード

4.2 状態表現に限量子を含む論理式の扱い

リスト要素に関する制約には限量子を含む論理式が現れるが，限量子が実質的な意味を持つのはループ中で参照する時のみである．抽象実行の状態遷移を求める場合，ループ以外の部分では限量子は実際には前提と結論で同一の部分式の一部として現れる．そのような限量子の出現は SMT ソルバーで自明に解決される．

ループの部分ではディレクティブによって展開を行う．ディレクティブの定義の正しさは SMT ソルバーで解決するのではなく，メタな論理で正しいとされているものを使用する．そのため，限定された展開だけを利用できるようにしている．

```
void f(struct S *root) {
    struct S *p = root;
    do {
        $assume( ls_next(root, p) * p↦m * ls_next(m.next, root)
                 ∧(p->lock = 0) );
        lock(p); g(p); unlock(p);
        $unfold_ls_next(p->next);
        p = p->next;
        $assert( ls_next(root, p) * p↦m * ls_next(m.next, root)
                 ∧(p->lock = 0) );
    } while (p != root);
}
```

図 6　ディレクティブ展開後のコード

4.3　リスト要素に対するループの展開

MKencha/ng では，リスト表現に関する論理式の変換をディレクティブに頼っている．図 4 にディレクティブとその変換を示す．これらディレクティブによる変換は等価な変換である（ls_{next} と cls_{next} の定義である）．unroll ディレクティブが指示する展開は，循環リストを一点 p で分割して三つの部分に分割している．

roll/unroll ディレクティブは，リスト要素に関する述語とループに使用するプログラム変数の関連を指定する．unroll は cls_{next} を展開するが，そのためにループ本体を実行する際の事前条件を挿入する．roll は展開された cls_{next} を元に戻すが，そのための条件をアサーションの形で挿入する．

unfold ディレクティブは，ls_{next} を一段展開する指示である．ディレクティブが指示する展開は，ls_{next} の再帰的な定義そのままである．

図 5 にディレクティブの使用例を示す．図 6 にディレクティブのために挿入されたアサーションを示す．ディレクティブを使用するための条件がアサーション文（$assume と$assert）として挿入されている．

このアサーションは静的に生成されるのではなく，ループ実行前の抽象実行状態を使って生成される．ループ中に現れる二つのアサーションの内容は同一である．

4.4　リストの再参照の検出

リストの操作中に改めてルートからリストをたどるようなコードの実行はエラーを通知すべきである．リストが展開された状況でディレクティブを利用しようとする場合，必要な実行状態を持たないのでディレクティブの展開が失敗を通知する．例えば図 1 のコード例では，本体の処理を行う関数呼出し $g()$ の中で同じリストをルート $root$ からたどることはできない．この場合，リストをたどるには循環リストを展開する必要があるが既にヒープの状態は展開されてしまっている．

5　問題点と今後の課題

5.1　リスト表現の問題

モデル検査の利点は，述語セットを与えるだけで，ユーザーが正確なインバリアントを与える必要がない点である．与えた述語セットの正負リテラルの組合わせで状態が表現されるので，もしインバリアントがあればそれで表現される．しかし，ヒープに関しては操作／参照されるヒープ要素の全てについて正確に表現する必要がある．結果的に，述語セットではなく正確な（弱い）インバリアントを与えることが必要になっている．

さらに，MKencha/ng では，リスト表現に関してループに対する論理式の変換をディレクティブに頼っている．ループ変数と論理変数との対応を取る必要が理由である．現在，このディレクティブはリストを参照するループ全てに与える必要がある．ディレクティ

ブは簡単なパターンマッチによって付加することも考えられるが，その点は今後評価する必要がある．

5.2 エイリアス

カーネル・モジュールなどではある構造体を複数のリストにつないで管理することがある．ポインタのエイリアスは $\wedge$ を使って，$p_0 \mapsto m \wedge p_1 \mapsto n$ といった形で記述することができる．しかし，複数のリストが要素を共有していることを同じように表現しようとするのは，うまく行かない．

二つの異なるルート $root_a$，$root_b$ を持つリストを $\{cls_{next}(root_a) \wedge cls_{next}(root_b)\}$ と表現するものとする．一方のリストをたどっている時に，他方の制約を満たすことを確認することは難しい．両方のリストを同時に展開して制約を検査するようなことは，MKencha/ng の仕組みでは不可能である．ただし，どちらのリストの制約も同一の制約であれば記述できる可能性がある．今後の検討課題である．

5.3 ヒープ表現の直接操作

既に述べたように，実行によるヒープの変化は直接ヒープ表現を操作している．この操作は抽象実行のインタープリタで行っており，操作を行ってから SMT ソルバーで状態遷移を求めるようになっている．これは，述語論理による状態変化がリテラルの正負の変更で表現されることに比較すると，スマートではない実装である．ヒープ操作をもっとうまく表現する論理を捜す必要があると考えている．

ポインタに対する演算は現れないものと仮定している．特に MKencha/ng では，ヒープ操作をインタープリタで行うとき，ポインタを参照する式を字面上で比較している．つまり，$c = 4$ という制約があっても $p + 4$ と $p + c$ は同じポインタとは判断されない．ただし，もしポインタ演算が現れたとしても，一致するポインタがヒープ表現に存在しないことになりポインタ参照は失敗するので健全性はある．

6 関連研究

ツール Smallfoot/SpaceInvader [6] [23] はモデル検査器ではなく自動証明系に分類される．モデル検査との違いは，事前／事後条件を与えて検査が関数単位で行なわれる点である．一方，モデル検査では全実行パスを検査する．MKencha/ng は関数の入口で述語の選択による状態数の削減を行なうが，それと同様のことを Smallfoot/SpaceInvader ではフレーム公理として使用している．SpaceInvader では選言の項を過大近似する演算子を定義し，それによって選言の項を簡略化することでシェープ解析とその高速化を行う．モデル検査器はリテラルの連言を個々の状態とし，それを選言で並べたもので可能な状態群を表現する．モデル検査器は，各状態をほぼ独立に扱うので，状態の選言に関して自明な変換以外の簡略化は行わない．

モジュラーな検査を行う Smallfoot [5] では，フレーム・ルールにおける不変なフレーム部分，つまり，実際にコードが操作するヒープ以外の部分，の推論はパターンマッチ等の簡単な方法では発見できないので，重要になると述べている．一方，モデル検査では，ヒープの大域的な状態を受け取り抽象実行を進めるので，その部分で困難はない．

Botinčan [8] らも Smallfoot/SpaceInvader に同様のシステムを提案している．また，Distefano [14] らは Java 言語に対して同様のシステムを提案している．

BLAST の Lazy Shape Analysis [7] では，Lazy Abstration Refinement [16] による述語セットのリファインメントをシェープ解析に拡張している．そのため，シェープ解析のサマリーノードを示す述語，リストの後続リンクを示す述語，フィールドの値に関する述語を追加し，Lazy Abstration Refinement を行っている．MKencha/ng の開発は，ツールが毎度リファインメントを行うのは不要だという点からスタートしている．リファインメントは述語セットを最小にすることが目的であるが，実用上は小さいセットなら問題ない．その確認が MKencha/ng の開発の目的でもある．特に，我々がモデル検査で検査したい性質はロックなどの大域データに関わる．その場合，関係する実行パスが長くなり時間がかかることが予想される．述語セットの選択は重要な課題であるが，ツールとしては与えられたセットで速く検査するものも必要だという立場である．Lazy Shape Analysis ではヒープの構造を仮定せずに実行する必

要があるため保守的なエイリアス解析を使用しているが，MKencha/ng ではエイリアスはヒープ表現に反映されるので正確である．

SLAM [1] [2] [3] ツールでは，ポインタで指されたロックに関する検査を行うのに最初はロックが保持されていないことを暗黙に仮定している．その後，ロックを指すポインタを追跡してアンロックされることを検査する．Windows ではドライバ・ルーチンの呼出し規則で，ドライバから制御が離れる時は全てのロックを保持していないことを要求しているはずであり，暗黙の仮定は正当であり効果的な検査法になっている．MKencha/ng の用途では，暗黙の仮定はできないので明示的に記述することを選んでいる．

MKencha/ng ではヒープ表現に Separation Logic の記述を借りているが，重要な働きをしているのはリスト・セグメントである．一階述語論理でない述語に決定手続きを与えて検証を行うのは Suzuki 他の研究など古くから行われている [21]．特に，リスト・セグメントについては Burstall の研究や Nelson 他の研究で使用されている [10] [19]．

7 まとめ

MKencha/ng では，ヒープに関する決定手続きを SMT ソルバーに導入し，ヒープ中に置かれたロック等に関する性質に対してもモデル検査が適用可能であることを示した．また，ヒープの separating conjunction ついても述語論理の場合と同様に，関数に関係する述語セットに絞ることで SMT ソルバーに渡す要素数を削減した．

モデル検査器は与えられた述語セットを使って基本的に全パスに対して抽象実行を行う．抽象実行は状態が述語抽象で表現されていることを除けばプログラマに直感的であり，ツールとして利用し易いはずである．例えば，アサーションが満たせなかった場合，実行パスを遡って状態のバックトレースを確認するなど，普通のプログラムのデバッグと同じアプローチを採ることができる．

ここで述べた構造のモデル検査器は完成したものではなく，ヒープ構造のバリエーションに従って新しい決定手続きを SMT ソルバーに付け加えて行く必要がある．カーネル・モジュールで必要になるヒープ構造への対応を順次追加して予定である．

謝辞 本研究の一部は，科学技術振興機構 戦略的創造研究推進事業（CREST）（領域名：実用化を目指した組込みシステム用ディペンダブル・オペレーティングシステム）技術課題：「ディペンダブルシステムソフトウェア構築技術に関する研究」による．

参 考 文 献

[1] Ball, T., Majumdar, R., Millstein, T., and Rajamani, S. K.: Automatic Predicate Abstraction of C Programs. In Proc. of Conf. on Programming Language Design and Implementation (PLDI) (2001).

[2] Ball, T. and Rajamani, S. K.: The SLAM Project: Debugging System Software via Static Analysis. In Proc. of Conf. on Principles of Programming Languages (POPL) (2002).

[3] Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S. K., and Ustuner, A.: Thorough Static Analysis of Device Drivers. In Proc. of Eurosys Conf. (2006).

[4] Berdine, J., Calcagno, C., and O'Hearn, P.: A Decidable Fragment of Separation Logic. In Proc. of Foundations of Software Technology and Theoretical Computer Science (FSTTCS), LNCS 3328 (2005).

[5] Berdine, J., Calcagno, C., and O'Hearn, P.: Symbolic Execution with Separation Logic. In Proc. of ASIAN Symp. on Programming Languages and Systems (APLAS), LNCS 3780 (2005).

[6] Berdine, J., Calcagno, C., and O'Hearn, P.: Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In Proc. of Intl. Symp. on Formal Methods for Components and Objects (FMCO), LNCS 4111 (2006).

[7] Beyer, D., Henzinger, T. A., and Théoduloz, G.: Lazy Shape Analysis. In Proc. of Intl. Conf. on Computer Aided Verification (CAV), LNCS 4144 (2006).

[8] Botinčan, M., Parkinson, M., and Schulte, W.: Separation Logic Verification of C Programs with an SMT Solver. In Electronic Notes Theoretical Computer Science, Vol. 254 (2009).

[9] Bradley, A. R. and Manna, Z.: *The Calculus of Computation – Decision Procedures with Applications to Verification*. Springer Verlag (2007).

[10] Burstall, R.: Some Techniques for Proving Correctness of Programs which Alter Data Structures. In Meltzer, B., and Michie, D. (Eds.), Machine Intelligence 7, Edinburgh University Press, pp. 23–50 (1972).

[11] Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H.: Counterexample-Guided Abstraction Refinement. In Proc. of Intl. Conf. on Computer Aided

Verification (CAV), LNCS 1855 (2000).

[12] Cousot, P. and Cousot, R.: Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In Proc. of Conf. on Principles of Programming Languages (POPL) (1977).

[13] Detlefs, D., Nelson, G., and Saxe, J. B.: Simplify: a Theorem Prover for Program Checking. Jour. ACM (JACM), Vol. 52, No. 3, pp. 365–473 (2005).

[14] Distefano, D. and Parkinson, M. J.: jStar: Towards Practical Verification for Java. In Proc. of Conf. on Object-Oriented Programming Systems Languages and Applications (OOPSLA) (2008).

[15] Graf, S. and Saidi, H.: Construction of Abstract State Graphs with PVS. In Proc. of Intl. Conf. on Computer Aided Verification (CAV), LNCS 1254 (1997).

[16] Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G.: Lazy Abstraction. In Proc. of Conf. on Principles of Programming Languages (POPL) (2002).

[17] Kroening, D. and Strichman, O.: *Decision Procedures – An Algorithmic Point of View*. Springer Verlag (2008).

[18] Matsuda, M., Maeda, T., and Yonezawa, A.: Towards Design and Implementation of Model Checker for System Software. In Proc. of Intl. Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD) (2009).

[19] Nelson, G.: Verifying Reachability Invariants of Linked Structures. In Proc. of Conf. on Principles of Programming Languages (POPL) (1983).

[20] Reynolds, J. C.: Separation Logic: A Logic for Shared Mutable Data Structures. In Proc. of Logic in Computer Science (LICS) (2002).

[21] Suzuki, N. and Jefferson, D.: Verification Decidability of Presburger Array Programs. Jour. ACM (JACM), Vol. 27, No. 1, pp. 191–205 (1980).

[22] Tinelli, C.: A DPLL-Based Calculus for Ground Satisfiability Modulo Theories. In Proc. of European Conference on Logics in Artificial Intelligence (JELIA), LNCS 2424 (2002).

[23] Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., and O'Hearn, P.: Scalable Shape Analysis for Systems Code. In Proc. of Intl. Conf. on Computer Aided Verification (CAV), LNCS 5123 (2008).