実行履歴に基づくアクセス制御の形式モデルと検証

高田　喜朗[†a)]　　王　　静[††]　　関　　浩之[††]

A Formal Model and Its Verification of History-Based Access Control

Yoshiaki TAKATA[†a)], Jing WANG[††], and Hiroyuki SEKI[††]

あらまし　プログラムが望ましくない動作を行うことを防ぐ動的アクセス制御機構として，Java 仮想機械等に実装されているスタック検査が広く用いられている．しかし，スタック検査は，実行が完了したメソッド呼出しに関するセキュリティ情報が消失するという問題点をもつ．これを解決するため，実行履歴に基づく各種のアクセス制御機構が提案されている．本論文では，Abadi らのアクセス制御法を形式的に定義した HBAC プログラムモデルを提案し，HBAC プログラムに対するモデル検査法について考察する．具体的には以下の 2 点を行う．(1) HBAC プログラムに対する検証問題は，現実的な仮定のもとでは多項式時間可解であるが，一般の場合は決定性指数時間完全であることを示す．(2) HBAC プログラムの検証におけるいくつかの最適化法を提案する．試作検証ツールを使った簡単な実験により，現実的な時間と領域で HBAC プログラムの検証が可能であることを示す．

キーワード　言語ベースセキュリティ，スタック検査，実行履歴に基づくアクセス制御，モデル検査

1. まえがき

外部からダウンロードしたプログラムが望ましくない動作をするのを防ぐため，Java 仮想機械や共通言語ランタイム（CLR）などのプログラム実行系において，スタック検査（stack inspection）[15] という動的アクセス制御機構が導入され使用されている．これは，制御が特別なアクセス権検査文に到達したとき，呼出し制御スタック中の各メソッドが指定されたアクセス権を保持しているかどうか検査するという機構である．しかし，スタック検査では，実行が終了して呼出し制御スタックから削除されたメソッドに関する情報は扱えないため，セキュリティ保全の面から不十分であることが指摘されている．

この問題に対処するためにいくつかのアクセス制御モデルが提案されている [1], [14], [21]．これらのモデルに共通する機能は，メソッドの起動や資源へのアクセスなどに関する実行履歴を使用すること，かつ，メソッドの実行が完了してもそのメソッドを起動したという情報は必ずしも失われないこと，である．Schneider [21] は，実施可能なセキュリティポリシ（enforceable security policy）という概念を，イベント系列の集合に基づいて提案した．また，その表現方法として，セキュリティオートマトンを定義した．その後，Fong [14] はセキュリティオートマトンのいくつかの部分クラスを導入し，それらの表現能力を比較した．特に，有限状態セキュリティオートマトンの部分クラスである狭履歴オートマトン（shallow history automata，SHA）を定義し，表現能力に関して SHA はスタック検査と比較不能であることを示した．

これら以外のアプローチとして，Abadi と Fournet によるスタック検査の拡張 [1] がある．Abadi-Fournet のモデルでは，スタック検査と同様，アクセス制御の対象であるシステムはオブジェクト指向再帰プログラムであり，アクセス権の集合が静的に（実行前に）各メソッドに割り当てられる．一方，スタック検査とは違って，実行時アクセス権が処理系によって管理され，メソッド呼出しのたびに更新される．基本的には，実行時アクセス権はそれまでに実行されたすべてのメソッドに依存する．これは，スタック検査では実行が終了したメソッドに関する情報が消去されるのと対照

† 高知工科大学，香美市
Kochi University of Technology, Kami-shi, 782–8502 Japan
†† 奈良先端科学技術大学院大学，生駒市
Nara Institute of Science and Technology, Ikoma-shi, 630–0192 Japan
a) E-mail: takata.yoshiaki@kochi-tech.ac.jp

的である．Abadi らは，このモデルの C♯ 実行環境上の実装についても報告している．しかし，彼らのモデルに対する形式的検証法は，[2] を除いてほとんど研究されていない．

この論文では，Abadi-Fournet スタイルのアクセス制御に対する形式的モデルである History-Based Access Control（HBAC）を提案する(注1)．HBAC プログラムは，プログラム位置を頂点，制御フローを辺とする有向グラフである．本論文では，HBAC プログラムの構文と意味を形式的に定義した後，以下の 2 点を行う．

（1） HBAC プログラムに対する検証問題は，現実的な仮定のもとでは多項式時間可解であるが，一般の場合は決定性指数時間完全であることを示す．

（2） HBAC プログラムの検証におけるいくつかの最適化法を提案する．

本論文で提案する検証アルゴリズムは文脈自由言語の空判定に基づく．提案する最適化法は，不動点計算を使って，文脈自由文法の構築の際に不要な導出規則の構築を抑制するものである．試作検証ツールを使った簡単な実験により，現実的な時間と領域で HBAC プログラムの検証が可能であることを示す．

本論文の構成は以下のとおりである．本章の以降では関連研究について述べる．**2.** では HBAC プログラムの構文と意味を定義する．その後，上記（1），（2）をそれぞれ **3.**，**4.** で述べ，**5.** で最適化の効果を調べるための簡単な検証実験について述べる．最後に **6.** でまとめと今後の課題を述べる．

関 連 研 究

実行履歴に基づくアクセス制御に対する検証法がいくつか研究されている [2]〜[4], [9], [16].

Bartoletti ら [3], [4] は，プログラムモデルとして，正規言語で表された局所ポリシ付き値呼び λ 計算を提案している．このモデルでは，関数呼出しごとに，新しい（ただし呼び出された関数に静的に束縛された）局所ポリシが入れ子となって導入される．そして，このプログラムモデルに対するモデル検査問題を基本プロセス代数（basic process algebra，BPA）に対するモデル検査問題に帰着する方法が提案されている．[3], [4] で扱われているアクセス制御機構はセキュリティオートマトン [14], [21] の拡張であり，[1] や本論文のモデルと違って明示的な動的アクセス権検査文をもたない．

Erlingsson ら [9] は，与えられたセキュリティオートマトンを模倣するよう，アクセス制御対象プログラムに動的なアクセス制御文を挿入する方法を提案している．その後の研究 [16] では，書き換えられたプログラムがセキュリティポリシを満たすことを保証するような型システムが提案されている．このモデルも [14], [21] に基づくものであり，[1] や本論文のものとは異なる(注2)．また，モデル検査問題は扱われていない．

本論文と最も関連する既存研究として [2] がある．[2] のプログラムモデルは，明示的な動的アクセス権検査文及び HBAC と同様の grant/accept 機構をもつ．そして，Volpano-Smith スタイルの型システム [22] を提案し，型安全なプログラムが非干渉性（noninterference）を満たすことを証明している．アクセス制御の主目的の一つは望ましくない情報漏えいを防ぐことであるので，非干渉性を保証する技術は重要である．一方 [2] では，[9], [16] と同様に，モデル検査問題は扱われていない．また，上記の研究はいずれも，検証ツールを実装する上で重要な，検証問題の計算複雑さや最適化法について議論していない．

2. HBAC プログラム

Abadi ら [1] は，後述の grant/accept 機構によって実行時アクセス権を柔軟に制御する方法を提案している．また，C♯ 上での実装を行っている．一方，これらの機構やアクセス制御法の形式的な定義は与えていない．本論文では，[18], [20] でのスタック検査プログラムのモデル化を参考に，HBAC プログラムの構文と操作的意味論を新たに定義する．

HBAC プログラムは単純な制御フローグラフであり，頂点の種類は call，return，check の 3 通りである．グラフは一般に複数個のメソッドに分割される．各メソッドにはアクセス権の部分集合が割り当てられ，これをそのメソッドの静的アクセス権という．

1 メソッド内でのプログラムの状態は，2 項組 $\langle n, C\rangle$ で表される．ただし，n は現在制御のある頂点（すなわちプログラム位置），C は実行時アクセス権と呼ばれるアクセス権の部分集合である．大域的なプログラムの状態は，呼出し制御スタック（以降，単にスタックという）で表される．スタックは，メソッド内での状態の有限列 $\langle n_1, C_1\rangle : \ldots : \langle n_k, C_k\rangle$ である．最も左

(注1)：本論文の 2〜3 章は国際会議発表 [23] に基づく．

(注2)：[10] では，[9] で用いられている Inlined Reference Monitor を使って Java スタック検査を実装する方法が提案されている．ただし，呼出し制御スタックを参照するなど本質的には Java スタック検査と同じであり，[14], [21] のモデルとは異なる．

の 2 項組がスタックトップを表す．

call 頂点はメソッド呼出しを表す．状態 $\langle n_1, C_1\rangle : \ldots : \langle n_k, C_k\rangle$ において，n_1 がメソッド f を呼び出す call 頂点であるとする．このとき，m を f の入口頂点，C' を C_1 と f の静的アクセス権の積集合として，2 項組 $\langle m, C'\rangle$ がスタックに追加される．すなわち，実行時アクセス権から f の静的アクセス権に含まれないものが除去される．

更に，call 頂点は二つのパラメータ，grant 権と accept 権をもつ．上の例で n_1 の grant 権・accept 権がそれぞれ P_G, P_A であるとすると，P_G はメソッド f を実行している間，実行時アクセス権に一時的に追加され，P_A は f から復帰する際に実行時アクセス権に追加される．つまり，grant 権はプログラム実行開始時点から現在までに失われたアクセス権の一部を一時的に回復し，accept 権は呼び出されたメソッドの実行中に失われたアクセス権の一部を回復する．

例えばスタック検査では，メソッド呼出しから復帰したら，呼出し中に失われたアクセス権はすべて回復される．これは，accept 権を最大に設定したのと等しい．一方，accept 権を空集合に設定すると，呼び出されたメソッドの実行中に失ったアクセス権は復帰後も失ったままとなる．これはスタック検査と最も対照的な場合である．このどちらの設定も可能であることが HBAC プログラムの特徴である．更に，accept 権をこれらの中間の値に設定することもできるので，例えばあるアクセス権 p_1 を失ったことはメソッドからの復帰と同時に忘却するが，p_2 を失ったことは記憶したまま以降の実行を続ける，というようにもできる．

grant 機構は，スタック検査での特権呼出し（Java の doPrivileged 文）を拡張したものにあたる．特権呼出しは，実行時アクセス権を一時的に呼出しもとの静的アクセス権に戻してから呼出し先を実行し，復帰後にもとの実行時アクセス権に戻す機構である．これは grant 権を最大に設定したのと等しい．特権呼出しでない通常の呼出しは，grant 権を空集合に設定したのと等しい．accept 権と同様に，grant 権もこれらの中間の値に設定できる．

Abadi らの提案では，Java の doPrivileged 文と同じ構文の Grant 文・Accept 文によって，grant 権・accept 権を指定するようになっていた．本論文では, [18], [20] で doPrivileged 文の作用の有無を call 頂点の属性として定義したのに倣い，grant 権・accept 権を call 頂点のパラメータとしてモデル化した．

check 頂点は，実行時アクセス権が指定されたアクセス権を含んでいるかどうか検査し，もし含んでいなければプログラムの実行を強制終了する．

簡単のため，本 HBAC モデルには例外処理機構を含めないことにするが，throw-catch 形の例外処理機構をモデルに含めることや，それに合わせて **4.** のモデル検査法を拡張することは, [19] と同様にして比較的容易に行えると考えられる．

形式的には，HBAC プログラムは 7 項組 $\pi = (NO, TG, CG, IS, IT, PRM, SP)$ で定義される有向グラフである．ただし，NO は頂点の有限集合，$TG, CG \subseteq NO \times NO$ はそれぞれ遷移辺，呼出し辺と呼ばれる有向辺の集合，$IS : NO \to \{call[P_G, P_A] \mid P_G, P_A \subseteq PRM\} \cup \{check[P] \mid P \subseteq PRM\} \cup \{return\}$ は頂点へのラベル付け関数，$IT \in NO$ はプログラムの開始点を表す初期頂点，PRM はアクセス権の有限集合，$SP : NO \to 2^{PRM}$ は頂点へのアクセス権の部分集合の割当である．

各頂点 $n \in NO$ は，IS によるラベル付けによって以下の 3 種類に分類される．

- $IS(n) = call[P_G, P_A]$, $P_G, P_A \subseteq PRM$．頂点 n はメソッド呼出しを表す．パラメータ P_G, P_A をそれぞれ grant 権，accept 権という．
- $IS(n) = return$．頂点 n はメソッドからの復帰を表す．
- $IS(n) = check[P]$, $P \subseteq PRM$．頂点 n は実行時アクセス権に対する検査を表す．実行時アクセス権が P を部分集合として含めば，実行は継続される．そうでなければ強制終了される．アクセス権 $p \in PRM$ に対し，$check[\{p\}]$ を $check[p]$ と略記する．

遷移辺（transfer edge，以降 tg と書く）はメソッド内の制御フローを表す．呼出し辺（call edge，以降 cg と書く）はメソッド呼出し元と呼出し先を結ぶ．読みやすさのため，$(n, n') \in TG$ であることを $n \overset{TG}{\to} n'$ と書く．同様に $(n, n') \in CG$ であることを $n \overset{CG}{\to} n'$ と書く．本論文中の図では，実線の矢印は cg を表し，破線の矢印は tg を表す．初期頂点は，図の外から入る矢印で表す．

［例 1］ 図 1 は頂点 n_0 を初期頂点とする HBAC プログラム π_1 である．n_0 から n_1 への tg は，n_0 の実行後，制御が n_1 に移動できることを表す．n_1 から n_4 への cg は，制御が n_1 に到達したとき，メソッド呼出しにより，n_4 に制御が移ることを表す．そして，制御が n_5 に到達したとき，n_1 に復帰する． □

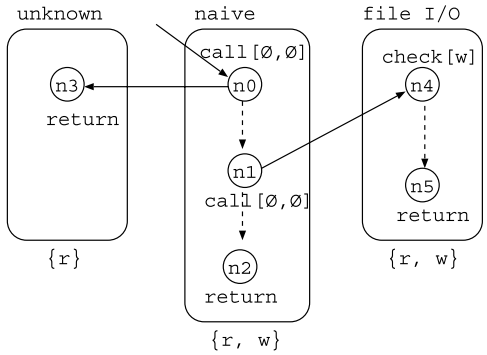

図 1 HBAC プログラム
Fig. 1 An HBAC program.

$SP(n)$ は，実行開始前に頂点 n に割り当てられるアクセス権の部分集合（静的アクセス権）を表す．ここで，同じメソッド中の頂点は同じ静的アクセス権をもつと仮定する．すなわち，

$$n \overset{TG}{\rightarrow} n' \Rightarrow SP(n) = SP(n').$$

また，$IS(n) = call[P_G, P_A]$ である各 call 頂点 n について，$P_G \subseteq SP(n)$ かつ $P_A \subseteq SP(n)$ とする．

図 1 において，メソッドは長方形で囲まれた頂点群で表されている．長方形の近くに書かれた集合は，そのメソッド中の頂点の静的アクセス権を表す．例えば，$SP(n_0) = SP(n_1) = SP(n_2) = \{r, w\}$, $SP(n_3) = \{r\}$ である．

HBAC プログラム $\pi = (NO, TG, CG, IS, IT, PRM, SP)$ の記述長を $\|\pi\| = |NO| \cdot |PRM| + |TG| + |CG|$ と定義する．頂点 $n \in NO$ とアクセス権の部分集合 $C \subseteq PRM$ の組 $\langle n, C\rangle$ をスタックフレームと呼び，スタックフレームの有限系列をスタックと呼ぶ．π の大域的状態（configuration）はスタックで表される．スタックフレームの列 ξ_1, ξ_2 の連接を $\xi_1 : \xi_2$ で表す．

HBAC プログラムの意味論は，大域的状態の間の遷移関係 $\rightarrow$ で表される．$\rightarrow$ は以下の推論規則を満たす最小の関係である．

$$\frac{IS(n) = call[P_G, P_A],\ n \overset{CG}{\rightarrow} m}{\langle n, C\rangle : \xi \rightarrow \langle m, (C \cup P_G) \cap SP(m)\rangle : \langle n, C\rangle : \xi} \quad (1)$$

$$\frac{IS(m') = return,\ IS(n) = call[P_G, P_A],\ n \overset{TG}{\rightarrow} n'}{\langle m', C'\rangle : \langle n, C\rangle : \xi \rightarrow \langle n', C \cap (C' \cup P_A)\rangle : \xi} \quad (2)$$

表 1 実行時アクセス権の更新
Table 1 Modification of current permissions.

	method call	return
(general case)	$(C \cup P_G) \cap SP(m)$	$C \cap (C' \cup P_A)$
$P_G = SP(n)$	$SP(n) \cap SP(m)$	$C \cap (C' \cup P_A)$
$P_G = \emptyset$	$C \cap SP(m)$	$C \cap (C' \cup P_A)$
$P_A = SP(n)$	$(C \cup P_G) \cap SP(m)$	C
$P_A = \emptyset$	$(C \cup P_G) \cap SP(m)$	$C \cap C' (= C')$
$P_A = \emptyset, P_G = \emptyset$	$C \cap SP(m)$	$C \cap C' (= C')$

$$\frac{IS(n) = check[P],\ P \subseteq C,\ n \overset{TG}{\rightarrow} n'}{\langle n, C\rangle : \xi \rightarrow \langle n', C\rangle : \xi} \quad (3)$$

大域的状態 $\xi = \langle n_1, C_1\rangle : \ldots : \langle n_k, C_k\rangle$ において，スタックトップは $\langle n_1, C_1\rangle$ である．C_1 を ξ における実行時アクセス権と呼ぶ．

規則 (1) は，制御が $IS(n) = call[P_G, P_A]$ かつ $n \overset{CG}{\rightarrow} m$ である call 頂点 n にあるならば，スタックフレーム $\langle m, (C \cup P_G) \cap SP(m)\rangle$ がスタックに追加され得ることを表す．つまり，制御が call 頂点 n に到達したとき，メソッド呼出しによって制御が m に移動し，実行時アクセス権が $(C \cup P_G) \cap SP(m)$ に変わる．規則 (2) はメソッドからの復帰に関する規則である．$n \overset{TG}{\rightarrow} n'$ であると仮定する．もし現在のプログラム位置が call 頂点 n から呼び出されたメソッド中の return 頂点 m' ならば，次の状態でのプログラム位置は n' となることができる．また，実行時アクセス権は $C \cap (C' \cup P_A)$ となる．もし call 頂点 n から出る cg が一つもなければ，規則 (1) は n に適用できないため，制御は n から先に進めない．同様に，もし call 頂点 n から出る tg が一つもなければ，制御は呼び出されたメソッド中の return 頂点から先に進めない．このような行き止まりのあるプログラムは実用的には無意味だが，定義を簡潔にするため，tg や cg に構文的な制約は設けないことにする．

定義より，初期状態から到達可能な大域的状態に対して規則 (2) を適用するときは，必ず $C' \subseteq C \cup P_G$ である．表 1 は，いくつかの特別な場合について実行時アクセス権がどのように更新されるかを示している．

規則 (3) は，制御が $IS(n) = check[P]$ かつ $n \overset{TG}{\rightarrow} n'$ である check 頂点 n に到達し，かつ実行時アクセス権が P を含むならば，制御が n' に移動できることを表す．

π のトレースの集合を以下のように定義する．

$$[\![\pi]\!] = \{n_0 n_1 \ldots n_k \mid n_0 = IT,\ C_0 = SP(IT),\ \xi_0 = \varepsilon,\ \exists C_1, \ldots, C_k \subseteq PRM,$$

$$\exists \xi_1, \ldots, \xi_k \in (NO \times 2^{PRM})^*,$$
$$\langle n_i, C_i \rangle : \xi_i \to \langle n_{i+1}, C_{i+1} \rangle : \xi_{i+1}$$
$$\text{for } 0 \le i < k\}$$

ただし，ε は空系列を表す．

系列の集合 S に対し，S 中の系列の空ではない接頭辞すべてからなる集合を prefix(S) とする．

［例 2］ 図 1 の HBAC プログラム π_1 について再び考える．メソッド「unknown」が n_0 によって呼び出されたとき，$IS(n_0) = call[\emptyset, \emptyset]$ であるので，実行時アクセス権は $\{r, w\} \cap SP(n_3) = \{r, w\} \cap \{r\} = \{r\}$ に変化する（表 1 を参照）．$IS(n_4) = check[w]$ であり，かつ n_4 に到達したときの実行時アクセス権 $\{r\}$ は $\{w\}$ を含んでいないので，頂点 n_4 による検査は失敗する．したがって，

$$\langle n_0, \{r, w\} \rangle \to \langle n_3, \{r\} \rangle : \langle n_0, \{r, w\} \rangle \to \langle n_1, \{r\} \rangle$$
$$\to \langle n_4, \{r\} \rangle : \langle n_1, \{r\} \rangle \not\to \langle n_5, \{r\} \rangle : \langle n_1, \{r\} \rangle.$$
$$[\![\pi_1]\!] = \{n_0, n_0 n_3, n_0 n_3 n_1, n_0 n_3 n_1 n_4\}$$
$$= \text{prefix}(\{n_0 n_3 n_1 n_4\}).$$

この π_1 には複数の tg または複数の cg が出る頂点はなく（すなわち非決定性でなく），かつ閉路がないので，トレース集合は一つの系列 $n_0 n_3 n_1 n_4$ の接頭辞で表される． □

例 2 において，check 頂点 n_4 で実行時アクセス権が書込み権 w を含むかどうかを検査する理由は以下のとおりである．メソッド naive は，メソッド unknown の呼び出し後にメソッド file I/O を呼び出し，変数 *fname* で指されたファイルを削除するものとする．一方 unknown は，変数 *fname* の値を変更することで，重要なファイルを naive に削除させようと試みるとする．もし，file I/O が最初に $check[w]$ を実行するならば，unknown を実行したことにより実行時アクセス権から書込み権 w が失われているので，検査に合格せず，意図しないファイルの削除は行われない．しかしスタック検査では，unknown から naive に復帰した時点で unknown を呼び出したという情報が失われるので，このようなアクセス制御は実現できない．このような状況は，HBAC の利点が生かされる典型的な例である．

［例 3］ Chinese wall ポリシ [5] とは，利用者はすべての資源に対するアクセス権をもつが，一度いずれかの資源にアクセスすると，それ以外の資源へのアクセス権を失う（ある資源を選択すると，それと競合する他の資源は利用できなくなる）というアクセス制御ポリシのことである．図 2 の π_2 は，単純化した Chinese wall ポリシを表す HBAC プログラムである．n_0 が「serviceA」を呼び出すと，実行時アクセス権から p_B が失われる．したがって，その後 n_1 が「serviceB」を呼び出すと，n_5 での検査が失敗する．n_0 で「serviceB」を呼び出し，その後 n_1 で「serviceA」を呼び出した場合も同様である．実際，π_2 のトレース集合は以下のようになる．

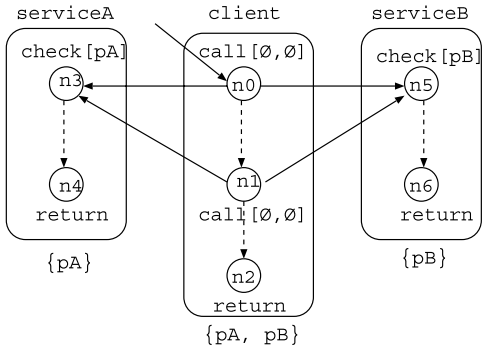

図 2 Chinese wall ポリシ
Fig. 2 Chinese wall policy.

$$[\![\pi_2]\!] = \text{prefix}(n_0 n_3 n_4 n_1 (n_3 n_4 n_2 + n_5)$$
$$+ n_0 n_5 n_6 n_1 (n_5 n_6 n_2 + n_3)).$$

ただし，prefix の引数を正規表現で表しており，$+$ は正規表現における和演算子である． □

3. HBAC プログラムのモデル検査

この章では，以下のように定義される検証問題（モデル検査問題ともいう）を議論する．

入力： HBAC プログラム $\pi = (NO, \ldots)$ 及び検証項目 $\psi \subseteq NO^*$．

出力： $[\![\pi]\!]$ 中のすべてのトレースが ψ を満たすか（すなわち $[\![\pi]\!] \subseteq \psi$ か）

［例 4］ 例 3 のプログラム π_2，及び検証項目 $\psi = (\Sigma - \{n_4\})^* + (\Sigma - \{n_6\})^*$ に関する検証問題について考える．ただし，$\Sigma = (n_0 + n_1 + \cdots + n_6)$ とする．例 3 で述べたように，頂点 n_4 と n_6 は単一のトレース上に両方現れることはない．したがって $[\![\pi_2]\!] \subseteq \psi$ が成り立つ． □

オートマトンや文法などの言語の表現 M に対し，M の記述長を $\|M\|$，M が表す言語を $L(M)$ とする．

［補題 1］ HBAC プログラム π に対し，$L(G) = [\![\pi]\!]$ かつ $\|G\| = O(\|\pi\| \cdot c^{|PRM|})$ $(c > 1)$ である文脈自由文法（cfg）G を構成できる．

（略証）G の非終端記号の集合を $(NO \times 2^{PRM}) \cup (NO \times 2^{PRM} \times 2^{PRM})$ とする．読みやすさのため，2 項組である非終端記号を $\langle n, C\rangle$，3 項組である非終端記号を $[n, C, C']$ と書く．2 項組である非終端記号 $\langle n, C\rangle \in NO \times 2^{PRM}$ からは，大域的状態 $\langle n, C\rangle$（すなわち $\langle n, C\rangle$ のみからなるスタック）から実行を開始したときのすべてのトレースが導出されるよう G を定義する．3 項組である非終端記号 $[n, C, C'] \in NO \times 2^{PRM} \times 2^{PRM}$ からは，大域的状態 $\langle n, C\rangle$ から実行を開始したときのトレースのうち，ある return 頂点 n' に関して大域的状態 $\langle n', C'\rangle$ で終了するようなものすべてが導出されるよう G を定義する．**2.** での定義より，大域的状態 $\langle n', C'\rangle$ からは遷移できない（**2.** の規則 (1)〜(3) はいずれも適用できない）ことに注意．

$[n, C, C']$ から導出される系列は，n から開始して，n が属するメソッドの終端である return 頂点に到達するまでのトレースを表す．途中の check 頂点で実行が中止されるようなトレースは含まない．n が属するメソッドが再帰呼出しされる場合を考慮して，「トレースの始点と同じ呼出しインスタンスにおける return 頂点に到達する」ことを，「長さ 1 のスタックから開始して長さ 1 のスタックで return 頂点に到達する」と表現している．例えば，下記の導出規則 (4) で，非終端記号 $[m, P_1, C']$ は，呼び出されたメソッドの開始頂点 m から実行を始め，その呼出しから復帰するまでのトレースを表している．このとき，メソッド終了時の実行時アクセス権 C' が，復帰後のトレースに影響する．なお，C' がどのような値になるかは G の構築時には分からないので，任意の C' に対して導出規則 (4)，(5) をもつよう G を定義する．return 頂点に関する導出規則の定義から，return 頂点に到達したときの実行時アクセス権が C' であるようなトレースのみ，$[n, C, C']$ から導出可能となる．

以下では，C, C', C'' を PRM の任意の部分集合とする．

各頂点 n について，G は導出規則 $\langle n, C\rangle \to n$ をもつ．各 call 頂点 n と $n \overset{CG}{\to} m$ である m について，$IS(n) = call[P_G, P_A]$ ならば，G は導出規則 $\langle n, C\rangle \to n\,\langle m, P_1\rangle$ をもつ．ただし $P_1 = (C \cup P_G) \cap SP(m)$ である．更に，$n \overset{TG}{\to} n'$ である各 n' に対し，G は以下の導出規則をもつ．

$$\langle n, C\rangle \to n[m, P_1, C']\langle n', P_2\rangle \tag{4}$$

$$[n, C, C''] \to n[m, P_1, C'][n', P_2, C''] \tag{5}$$

$$P_2 = C \cap (C' \cup P_A)$$

各 check 頂点 n と $n \overset{TG}{\to} n'$ である n' について，$IS(n) = check[P]$ かつ $P \subseteq C$ ならば，G は以下の規則をもつ．

$$\langle n, C\rangle \to n\langle n', C\rangle$$

$$[n, C, C'] \to n[n', C, C']$$

各 return 頂点 n について，G は規則 $[n, C, C] \to n$ をもつ．G の開始記号を $\langle IT, SP(IT)\rangle$ とする． □

［定理 1］ π を HBAC プログラム，M を有限オートマトンとする．π と $\psi = \overline{L(M)}$ に関する検証問題は決定性 $O(\|\pi\| \cdot c^{|PRM|} \cdot \|M\|^3)$ 時間 $(c > 1)$ で可解である．

（略証）補題 1 より，$L(G) = [\![\pi]\!]$ である cfg G を構築することができる．定理で述べられている検証問題は $L(G) \cap L(M) = \emptyset$ の判定と等価であり，これは $O(\|G\| \cdot \|M\|^3)$ 時間で決定可能である． □

［系 1］ 実行時アクセス権（定義より PRM の部分集合）のとり得る値が $O(\|\pi\|^c)$ 個ならば，π と $\psi = \overline{L(M)}$ に関する検証問題は，決定性 $O(\|\pi\|^{1+3c} \cdot \|M\|^3)$ 時間で可解である． □

$|PRM| = O(\log \|\pi\|)$ は，系 1 の前提の十分条件である．

系 1 の前提が現実的かどうかについて考える．現実のプログラムでは，管理が複雑になるのを防ぐため，メソッドに静的アクセス権を割り当てるためのポリシ記述（Java の java.policy ファイルに相当）や grant 権・accept 権の指定では，個々のファイル等の細かい単位ではなく，「ディレクトリ/tmp 中のすべてのファイルに対する書込み権」のような包括的なアクセス権が使われる．ポリシ記述や grant 権・accept 権に使われるアクセス権の集合を PRM_{plc} とすると，実行時アクセス権のとり得る値は PRM_{plc} の部分集合となる．$|PRM_{plc}|$ は $\|\pi\|$ に比べて十分小さいと考えられるので，ここで $|PRM_{plc}| = O(\log \|\pi\|)$ と仮定すると，実行時アクセス権のとり得る値の個数は $O(2^{|PRM_{plc}|}) = O(\|\pi\|^c)$ となる．したがって，現実的な環境では，系 1 の前提が満たされると考えられる．

定数 c (> 1) と多項式 p に関して決定性 $O(c^{p(n)})$ 時

間で可解である決定問題のクラスを EXPTIME とする．以下の定理は，$|PRM| = O(\log \|\pi\|)$ という仮定が成り立たない場合，検証問題が EXPTIME 完全であることを述べている．

［定理 2］ π を HBAC プログラム，M を有限オートマトンとする．π と $\psi = \overline{L(M)}$ に関する検証問題は EXPTIME 完全である．

（略証）EXPTIME 困難性は，多項式領域限定交替 Turing 機械 [6] の所属問題からの帰着で示せる． □

4. モデル検査アルゴリズムの最適化

定理 1 の証明より，検証問題を解く以下のようなアルゴリズムが得られる．

［アルゴリズム 1］ 与えられた HBAC プログラム π と $\psi = \overline{L(M)}$ である有限オートマトン M に対し，以下の 3 ステップを順に実行する．

（1） 補題 1 の証明に沿って，$L(G) = [\![\pi]\!]$ である cfg G を構築する．

（2） $L(\widehat{G}) = L(G) \cap L(M)$ である cfg $\widehat{G}$ を構築する．

（3） $L(\widehat{G}) = \emptyset$ かどうか判定する． □

ステップ（1）で構築される G の大きさは $|PRM|$ に対して指数的である．しかし多くの場合，G には，G が表す言語に影響しないような不要な導出規則が多数含まれる．この章では，不要な導出規則を構築しないようにする方法をいくつか説明する．これにより，検証に必要な時間と領域を削減することができる．

4.1 基本アイデア

以下は，cfg 中の不要な導出規則を削除するよく知られたアルゴリズムである [17]．非終端記号 X から何らかの終端記号列が導出可能であるとき，X は生成的（generating）であるという．ある記号列 α, β が存在し，G の開始記号から $\alpha X \beta$ への導出が存在するとき，X は到達可能（reachable）であるという．導出規則 r が非生成的または非到達可能な記号を含むとき，r は不要であるという．不要な導出規則を削除するアルゴリズムは，生成的かつ到達可能な記号の集合 V を求めた後，V に含まれない記号を含む導出規則を削除する．このアルゴリズムは与えられた cfg 中の不要な導出規則を「削除」するが，ここで必要なのは，不要な導出規則を最初から「構築しない」ような方法である．補題 1 の証明中の G の定義から，以下の補題が得られる．

［補題 2］ π を HBAC プログラムとし，π に対してアルゴリズム 1 のステップ（1）で構築された cfg を G とする．任意の $n \in NO$ と $C, C' \subseteq PRM$ について，$C \not\subseteq SP(n)$ ならば，$\langle n, C\rangle$ 及び $[n, C, C']$ はどちらも到達可能でない．また，$C' \not\subseteq C$ ならば，$[n, C, C']$ は生成的でない． □

この補題より，$C \not\subseteq SP(n)$ または $C' \not\subseteq C$ であるような $\langle n, C\rangle$ または $[n, C, C']$ を含む導出規則は構築しなくてもよいことが分かる．しかし，依然多くの場合，導出規則の数は $|PRM|$ に対して指数的である．したがって，更なる最適化が必要である．

4.2 最適化 1：到達可能な記号からなる導出規則

以下のような探索アルゴリズムによって，到達可能な記号からなる導出規則だけを構築することができる．

［アルゴリズム 2］ 左辺が開始記号 $\langle IT, SP(IT)\rangle$ である導出規則をすべて構築する．次に，それらの導出規則の右辺に現れる非終端記号を左辺とする導出規則をすべて構築する．このことを，新しく現れる非終端記号がなくなるまで繰り返す． □

非終端記号の数が有限であるため，このアルゴリズムは必ず停止する．明らかに，導出規則 r が到達可能な記号のみからなるとき，かつそのときのみ，このアルゴリズムは r を構築する．もしある定数 c に対して以下の条件が成り立つとき，このアルゴリズムで構築される導出規則の数は $\|\pi\|$ に対して多項式的である．

（1） 開始点であるメソッド（すなわち IT が属するメソッド）中の任意の頂点 n，及び $n \overset{CG}{\to} m$ である任意の m について，$|SP(n) \cap SP(m)| < c$.

（2） $IS(n) = call[P_G, P_A]$ である各 call 頂点 n について，$|P_G| < c$.

例 3 の Chinese wall ポリシを表す HBAC プログラムはこの条件を満たす．

4.3 最適化 2：実行時アクセス権の事前計算

2 項組である非終端記号 $\langle n, C\rangle$ は，導出規則 $\langle n, C\rangle \to n$ が存在するので，必ず生成的である．一方，アルゴリズム 2 によって構築される導出規則には，到達可能だが生成的でない非終端記号 $[n, C, C']$ が含まれ得る．例えば仮に，$IS(n) = call[P_G, P_A]$ である n とある C に対し，$\langle n, C\rangle$ を左辺とする導出規則がこのアルゴリズムによって構築されるとする．このときこのアルゴリズムは，$n \overset{CG}{\to} m$ であるすべての m，$n \overset{TG}{\to} n'$ であるすべての n'，及びすべての $C' \subseteq P_1$ について，導出規則 $\langle n, C\rangle \to n\,[m, P_1, C']\langle n', P_2\rangle$（式 (4)）を構築する．ただし，$P_1 = (C \cup P_G) \cap SP(m)$，$P_2 = C \cap (C' \cup P_A)$ である．しかし，もし $IS(m) = return$

であれば，各 $C' \subseteq P_1$ に対する $2^{|P_1|}$ 個の $[m, P_1, C']$ のうち $[m, P_1, P_1]$ のみが生成的であり，それ以外の $[m, P_1, C']$ を含む導出規則はすべて不要である．

このような不要な導出規則の構築を防ぐため，まず集合 $X_{m,P_1} = \{C' \subseteq PRM \mid [m, P_1, C'] \text{ が生成的}\}$ を求め，$C' \in X_{m,P_1}$ である C' に対してのみ導出規則を構築するようにしたい．アルゴリズム 2 をこのように変更すれば，到達可能かつ生成的である記号のみからなる導出規則すべて，かつそれのみが構築される．

G の定義から，$X_{n,C}$ は以下の方程式の最小解である．

$$
X_{n,C} = \begin{cases} \{C\} & IS(n) = return \text{のとき}, \\ \emptyset & IS(n) = check[P],\ P \not\subseteq C \text{ のとき}, \\ \bigcup_{n' \in TG(n)} X_{n',C} & \\ \qquad IS(n) = check[P],\ P \subseteq C \text{ のとき}, & \\ \bigcup_{m \in CG(n)} \bigcup_{n' \in TG(n)} & \\ \quad \bigcup_{C' \in X_{m,(C \cup P_G) \cap SP(m)}} X_{n', C \cap (C' \cup P_A)} & \\ \qquad IS(n) = call[P_G, P_A] \text{ のとき}. \end{cases}
$$

ただし，$TG(n) = \{n' \mid n \overset{TG}{\rightarrow} n'\}$，$CG(n) = \{n' \mid n \overset{CG}{\rightarrow} n'\}$ とする．

もし HBAC プログラム π に $CG \cup TG$ 中の辺からなる閉路がなければ，与えられた n, C に対する $X_{n,C}$ の値を，上記の方程式に沿って再帰的に計算して求めることができる．つまり，上記の方程式を，2 引数 n, C をもつ関数の再帰的定義と考えればよい．しかし，π に閉路が含まれる場合，この再帰計算は停止しない可能性がある．すなわち，ある $X_{n,C}$ の計算中に再び $X_{n,C}$ の計算が呼び出され得る．そこで，そのような場合には，一時的に $X_{n,C} = 2^C$ と仮定して計算を続行するようにする．この計算によって得られる値は，$X_{n,C}$ を部分集合とするような（安全だが不要なものも含む）近似となる．なお，次の **4. 4** ではこの近似値を $X^*_{n,C}$ と表す．

4. 4　最適化 3：実行時アクセス権の正確な計算

以下のような反復アルゴリズムによって，$X_{n,C}$ の値を正確に計算することができる．

［アルゴリズム 3］　各 $X_{n,C}$ を変数とみなし，$V = \{X_{n,C} \mid n \in NO \text{ かつ } C \subseteq PRM\}$ と定義する．まず V 中の全変数を空集合に初期化する．現在の V 中の各変数の値を使って，各 n, C に対する **4. 3** の方程式の右辺の値を計算し，それを $X_{n,C}$ に代入する．変数の値が変化しなくなるまでこのことを繰り返す．□

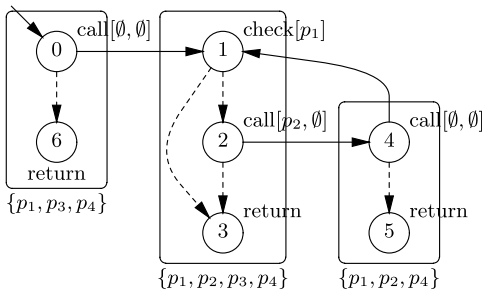

図 3　HBAC プログラムの例
Fig. 3　Sample HBAC program.

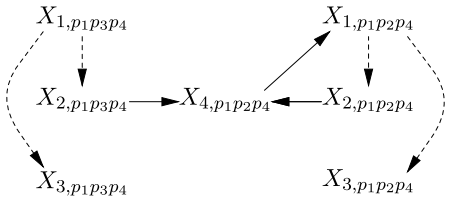

図 4　$X_{1,\{p_1,p_3,p_4\}}$ に対する依存グラフの初期状態
Fig. 4　Initial dependency graph for $X_{1,\{p_1,p_3,p_4\}}$.

$X_{n,C}$ の値域が有限であり，かつ方程式の右辺が単調（和集合演算のみで構成されている）なので，このアルゴリズムによって方程式の最小解を求めることができる．しかし，V 中のすべての変数を管理するのはコストが大きすぎるので，不必要な変数に関する計算は行わないようにしたい．また，ある $X_{n,C}$ の値が変化したとき，それに影響を受ける変数 $X_{n',C'}$ を効率良く見つけられるようにしたい．

上記の要求を満たすため，以下のようなアルゴリズムを用いる．

［アルゴリズム 4］　与えられたある n_0, C_0 に対する X_{n_0,C_0} の値の計算を目的とする．

（1）　X_{n_0,C_0} を開始点として，V（のある部分集合）中の変数間の依存関係を表す有向グラフを構築する．このグラフを依存グラフと呼ぶ．例えば，図 3 の HBAC プログラムに対して，$X_{1,\{p_1,p_3,p_4\}}$ の値を計算したい場合，すなわち $n_0 = 1, C_0 = \{p_1, p_3, p_4\}$ にこのアルゴリズムを適用した場合，このステップ（1）で図 4 のような依存グラフが構築される．

依存グラフは，もとの HBAC プログラムの頂点を頂点と実行時アクセス権の組に置き換えたような構造をしている．例えば，頂点 $X_{2,\{p_1,p_3,p_4\}}$ から $X_{4,\{p_1,p_2,p_4\}}$ への cg が存在するが，これは，「実行時アクセス権

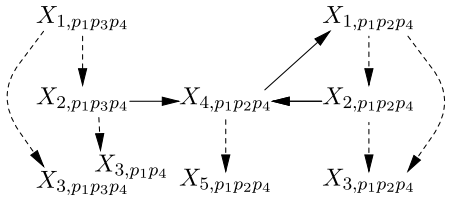

図 5　$X_{1,\{p_1,p_3,p_4\}}$ に対する最終的な依存グラフ
Fig. 5　Final dependency graph for $X_{1,\{p_1,p_3,p_4\}}$.

$\{p_1, p_3, p_4\}$ で頂点 2 に制御があるとき，頂点 4 を呼び出すことができ，それを実行すると実行時アクセス権は $\{p_1, p_2, p_4\}$ に更新される」ことを表している．

依存グラフにおいて $X_{n,C}$ から $X_{n',C'}$ への cg または tg が存在する場合，$X_{n,C}$ を左辺とする方程式の右辺に $X_{n',C'}$ が現れる．すなわち，依存グラフは V の要素間の依存関係を表している．

依存グラフは，指定された開始頂点から探索を開始して，もとの HBAC プログラム中の辺に対応する辺を順次加えていくことで構築できる．ただし，この時点では，呼出し先から復帰したときの実行時アクセス権が分からないため，call 頂点から出る tg は無視する．

依存グラフ中の各変数 $X_{n,C}$ の値を，$IS(n) = return$ のときは $\{C\}$，それ以外のときは空に初期化する．

以降，もとの HBAC プログラムの return 頂点に対応する依存グラフ中の頂点をやはり return 頂点と呼ぶ．call 頂点，check 頂点についても同様とする．

（2） 以下のように，return 頂点である変数の値を他の頂点に伝搬させていく．

(2a)　L を依存グラフ中の辺のリストとし，return 頂点に入る辺すべてからなるリストを L の初期値とする．

(2b)　L から 1 個辺を取り出し，これを e とする．

e が $X_{n,C}$ から $X_{n',C'}$ への tg ならば，$X_{n,C}$ に $X_{n',C'}$ の値を加える（$X_{n,C}$ の値を $X_{n,C} \cup X_{n',C'}$ に更新する）．この際に $X_{n,C}$ の値が変化したときは，$X_{n,C}$ に入る辺すべてを L に追加する．

e が $X_{n,C}$ から $X_{m,C''}$ への cg ならば，$n \overset{TG}{\rightarrow} n'$ である各 n' と各 $C' \in X_{m,C''}$ に対して $X_{n,C}$ から $X_{n',C\cap(C'\cup P_A)}$ への tg を加えることで，依存グラフを拡張する．ただし，P_A は n の accept 権である．もし $X_{n',C\cap(C'\cup P_A)}$ がそれまでに依存グラフ中に現れていなければ，更に $X_{n',C\cap(C'\cup P_A)}$ を開始点として，ステップ（1）と同様にして依存グラフを拡張する．

既に存在している頂点に入る辺，または return 頂点に入る辺が依存グラフに加えられたときは，その辺を L に追加する．

(2c)　L が空になるまでステップ 2b を繰り返す．

□

上記図 3, 図 4 の例では，最終的に図 5 のような依存グラフが得られ，$X_{1,\{p_1,p_3,p_4\}} = \{\{p_1, p_4\}, \{p_1, p_3, p_4\}\}$ という計算結果が得られる．一方，**4.3** のアルゴリズムは，$X^*_{4,\{p_1,p_2,p_4\}} = 2^{\{p_1,p_2,p_4\}}$ であるため，$X^*_{1,\{p_1,p_3,p_4\}} = 2^{\{p_1,p_4\}} \cup \{\{p_1, p_3, p_4\}\}$ という近似値を返す．

5. 実　　験

前章の最適化の効果を調べるため，検証ツールを実装し，以下の二つの例について検証時間を測定した．

（a） Chinese wall ポリシ

例 3 の HBAC プログラム π_2 を以下のように拡張し，これを $\pi_c(k)$ と呼ぶ．$\pi_c(k)$ は，π_2 中のメソッド serviceA, serviceB を serviceA の k 個のコピー $\text{service}_1, \ldots, \text{service}_k$ で置き換えたものである．すなわち，$\pi_c(k)$ はこれら k 個のメソッドとメソッド client からなる．メソッド client は，π_2 のものに，頂点 n_1 から n_1 自身への tg を加えたものである．client の静的アクセス権は $\{p_1, p_2, \ldots, p_k\}$，各 service_i $(1 \le i \le k)$ の静的アクセス権は $\{p_i\}$ である．

検証項目 ψ を

$$(N_1 \cup N_c)^* + (N_2 \cup N_c)^* + \cdots + (N_k \cup N_c)^*$$

と定義する．ただし，N_c は client 中の頂点の集合，N_i は service_i 中の頂点の集合を表す．すなわち ψ は，与えられた HBAC プログラムのトレースがいずれも $\text{service}_1, \ldots \text{service}_k$ 中の異なる二つ以上のメソッドを通過しないことを表している．

client は，（正当か不正かを問わず）任意の利用者のプロセスを表している．利用者は本来，client の中身を自由に定義し，client 中の頂点から他のメソッドの入り口頂点への cg を任意に置くことができる．「不正利用者の攻撃を防げるかどうか調べる」という観点から考えると，本来は「どのように client を定義しても検証項目 ψ が満たされる」かどうかを調べるべきだが，検証においてはプログラムを固定する必要があるため，ここでは client の中身を「2 回以上の任意の回数，$\text{service}_1, \ldots, \text{service}_k$ のうち任意のメソッドを呼

び出す」という一般的なものに固定して検証を行っている．

（b） オンライン銀行システム

ここでは，[20] で例として用いられているスタック検査付きプログラムに対して検証を行う．このプログラムと等価な HBAC プログラムを $\pi_o(k)$ とする．このプログラムは，k 個の銀行からなるオンライン銀行システムの一部をモデル化したものである（図 6）．各銀行は顧客に対し，預金引き出しを行うメソッドを提供する．spender は，正当な利用者のプロセスを表すメソッドであり，静的アクセス権として $\{d_1, \ldots, d_k\}$ をもつ．clyde は不正利用者のプロセスを表すメソッドであり，静的アクセス権は空である．両者は，第 i 銀行のメソッド debit_i $(1 \le i \le k)$ を呼び出すことができる．各 debit_i は，呼出しもとの利用者がアクセス権 d_i をもっているか検査し，その後，特権モードで read_i と write_i を呼び出す．

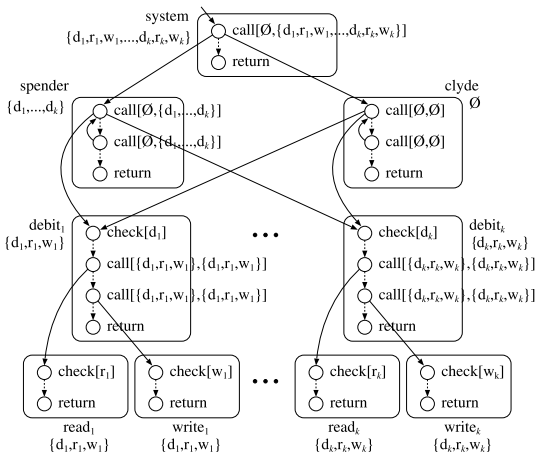

図 6 オンライン銀行システムの例
Fig. 6 Online banking system.

検証項目 ψ は，[20] と同じく，ψ の否定が $\overline{\psi} = \Sigma^* N_{\mathrm{clyde}} \Sigma^* N_{\mathrm{rw}} \Sigma^*$ となるよう定義する．ただし，Σ は全頂点の集合，N_{clyde} はメソッド clyde 中の頂点の集合，N_{rw} はメソッド read_i 及び write_i $(1 \le i \le k)$ 中の頂点をすべて集めた集合を表す．すなわち ψ は，与えられた HBAC プログラムにおいて制御が clyde を通過したら，その後 read_i にも write_i にも到達しないことを表している．

この例題も本来は「どのように clyde を定義しても検証項目 ψ が満たされる」かどうかを調べるべきだが，ここでも一例に固定して検証を行っている．

実験結果を表 2 に示す．HBAC プログラムの大きさは，現実のプログラムから内部計算を捨象し，メソッド呼出し・復帰とアクセス権検査のみを残したときの大きさに相当する．G はアルゴリズム 1 のステップ（1）で構築される cfg を表す．M は $\psi = \overline{L(M)}$ であるような正規文法を表す．上記の a)，b) とも，ψ は k に依存して決まるので，M も k に依存して決まる．図 7 は $\pi_c(k)$ 及び $\pi_o(k)$ の検証に掛かった計算時間を示している．

最適化を行わなければ，$\pi_o(k)$ については $k \ge 3$ のとき，$\pi_c(k)$ については $k \ge 10$ のとき，メモリ不足のため検証できなかった．これは，G の導出規則数が k に対して指数的となり，G を格納するためのメモリが不足したためである．$\pi_c(k)$ は **4. 2** で述べた条件を

表 2 例プログラムに対する検証結果
Table 2 Verification profiles of sample programs.

		$\pi_c(k)$						$\pi_o(k)$			
k		5	10	20	40	60	80	5	10	15	20
the number of permissions		5	10	20	40	60	80	15	30	45	60
the size of the	$\lvert NO\rvert$	13	23	43	83	123	163	48	88	128	168
HBAC program	$\lvert TG\rvert + \lvert CG\rvert$	18	33	63	123	183	243	54	99	144	189
the number	base*	2173									
of the rules	1**	163	473	1543	5483	11823		1870			
of G	1+2**	86	221	641	2081	4321	7361	184	1316	33200	
	1+2+3**	86	221	641	2081	4321	7361	142	277	412	547
$\lVert M\rVert$		124	389	1369	5129	11289	19849	212	392	572	752
computation	base	0.138									
time***(sec)	1	0.004	0.033	0.316	4.96	30.4		0.086			
	1+2	0.003	0.022	0.272	4.74	29.9	115	0.003	0.066	1.49	
	1+2+3	0.003	0.022	0.270	4.74	29.8	115	0.002	0.006	0.013	0.021
verification result		true	true	true	true	true	true	true	true	true	true

* base is Algorithm 1 modified based on Lemma 2 (Section 4. 1).
** optimizations 1 to 3 are described in sections 4. 2, 4. 3, and 4. 4, respectively.
*** Java VM build 1.5.0_07, on Mac OS X 10.4.10 (PowerPC G5, 2 GHz, 2 GB RAM).

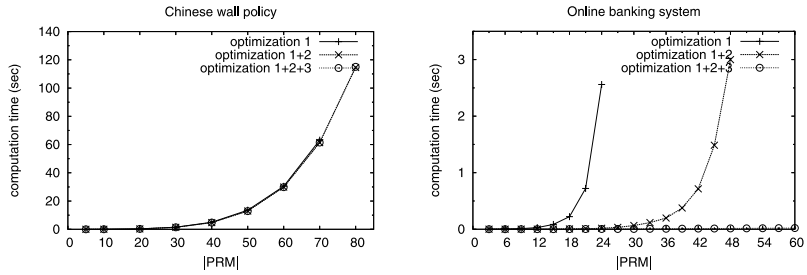

図 7　$\pi_c(k)$, $\pi_o(k)$ に対する検証時間
Fig. 7　Verification time for $\pi_c(k)$ and $\pi_o(k)$.

満たすため，最適化 1 のみでも十分 G のサイズが小さくなる．また，最適化 1＋2 で不要な導出規則をすべて除去できるため(注3)，最適化 1＋2 も 1＋2＋3 も構築される cfg は同一である(注4)．一方，$\pi_o(k)$ については，最適化 1 及び 2 だけでは G の導出規則数は k に対して依然指数的であり，比較的小さい k に対しても検証が行えなかった．すべての最適化を行った場合，$\pi_c(k)$, $\pi_o(k)$ とも，G の導出規則数及び計算時間は k に対して多項式的となった．特に，$\pi_o(k)$ の検証に掛かる時間は k に対して線形であった．[20] で述べられているように，通常のネットワークアプリケーションでのアクセス権の数はたかだか数十程度と思われるので，上記の結果から，提案した検証法は実際のプログラムに対して現実的な計算量で適用可能と考えられる．

6. むすび

本論文では，実行履歴に基づく動的アクセス制御機構のモデルである HBAC プログラムを提案した．そして，HBAC プログラムの検証問題が可解であることを示した．この検証問題は一般には決定性指数時間完全だが，いくつかの最適化法を検証ツールに実装した結果，例題プログラムを現実的な時間で検証することができた．

なお，本論文の例 2 や Abadi らの論文 [1] では，HBAC プログラムの表現能力がスタック検査付きプログラムより強いことを直観的に述べているが，厳密な証明は筆者らの別の発表 [24] で行われている．

提案したプログラムモデルはプッシュダウンシステム（pushdown systems，PDS）と呼ばれる無限状態システムと深く関連している．実際，HBAC プログラムの振舞いは，HBAC プログラムの大きさに対して指数個のスタック記号をもつような PDS でモデル化することができる．PDS に対する LTL 及び CTL* モデル検査 [7] の可解性及び計算複雑さについては，Esparza らの研究 [11], [12] が詳しい．PDS のためのモデル検査器の実装については [13] で述べられている．

本論文での検証問題及びモデル検査アルゴリズムは有限長のトレースに基づいて定義されているが，ω 文脈自由文法 [8] を使って，提案したアルゴリズムを無限長トレース及び LTL に拡張することが可能である．また，この場合の時間計算量は，[11] のアルゴリズムを HBAC プログラム（の振舞いを表す PDS）に適用した場合より若干改善される．具体的には，前者が $|Q_M|^3$ に比例するのに対し，後者は $|Q_M|^2|\Delta_M|$ に比例する．ただし，$|Q_M|$ は検証項目の否定を表す Büchi オートマトン M の状態数，$|\Delta_M|$ は M の遷移数である．なお，**4.** で述べた最適化法は，提案したモデル検査法を適用する場合だけでなく，PDS 用モデル検査器を使って HBAC プログラムの検証を行う場合でも有効である．

今後の課題として，セキュリティオートマトンの様々な部分クラス [14], [21] と，HBAC プログラムとの表現能力の比較などがある．

(注3)：プログラム中に n_1 に関する自己ループからなる閉路が存在するが，cfg G の構築中に $X_{n_1,C}$ の計算が必要になることはなく，必要な各 $X_{n,C}$ の値は閉路に影響されずに計算できる．
(注4)：ただし，G を生成するアルゴリズムの違いから，計算時間は全く同一にはなっていない．また，どちらが速いかも，ガーベジコレクション等の影響から，k の値によってまちまちである．

文　献

[1] M. Abadi and C. Fournet, “Access control based on execution history,” Network & Distributed System Security Symp., pp.107–121, 2003.
[2] A. Banerjee and D.A. Naumann, “History-based access control and secure information flow,” CASSIS04, LNCS 3362, pp.27–48, 2004.
[3] M. Bartoletti, P. Degano, and G.L. Ferrari, “Enforcing secure service composition,” IEEE 18th CSFW, pp.211–223, 2005.

[4] M. Bartoletti, P. Degano, and G.L. Ferrari, "History-based access control with local policies," 8th FOSSACS, LNCS 3441, pp.316–332, 2005.
[5] D.F.C. Brewer and M.J. Nash, "The Chinese wall security policy," IEEE Symp. on Security & Privacy, pp.206–214, 1989.
[6] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer, "Alternation," J. Assoc. Comput. Mach., vol.28, pp.114–133, 1981.
[7] E.M. Clarke, Jr., O. Grumberg, and D. Peled, Model Checking, MIT Press, 2000.
[8] R.S. Cohen and A.Y. Gold, "Theory of ω-languages. I: Characterizations of ω-context-free languages," J. Comput. Syst. Sci., vol.15, pp.169–184, 1977.
[9] Ú. Erlingsson and F.B. Schneider, "SASI enforcement of security policies: A retrospective," New Security Paradigms Workshop, pp.87–95, 1999.
[10] Ú. Erlingsson and F.B. Schneider, "IRM enforcement of Java stack inspection," IEEE Symp. on Security & Privacy, pp.246–255, 2000.
[11] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon, "Efficient algorithms for model-checking pushdown systems," CAV2000, LNCS 1855, pp.232–247, 2000.
[12] J. Esparza, A. Kučera, and S. Schwoon, "Model-checking LTL with regular variations for pushdown systems," TACS01, LNCS 2215, pp.316–339, 2001.
[13] J. Esparza and S. Schwoon, "A BDD-based model checker for recursive programs," CAV2001, LNCS 2102, pp.324–336, 2001.
[14] P.W. Fong, "Access control by tracking shallow execution history," IEEE Symp. on Security & Privacy, pp.43–55, 2004.
[15] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers, "Going beyond the sandbox: An overview of the new security architecture in the Java^{TM} development kit 1.2," USENIX Symp. on Internet Technologies and Systems, pp.103–112, 1997.
[16] K.W. Hamlen, G. Morrisett, and F.B. Schneider, "Certified in-lined reference monitoring on .NET," Technical Report TR2005-2003, Cornell University Computing and Information Science, 2005.
[17] J.E. Hopcroft, R. Motwani, and J.D. Ullman, Introduction to automata theory, languages, and computation, Second ed., Chapter 7, Addison Wesley, 2001.
[18] T. Jensen, D. le Métayer, and T. Thorn, "Verification of control flow based security properties," IEEE Symp. on Security & Privacy, pp.89–103, 1999.
[19] S. Kuninobu, N. Nitta, Y. Takata, and H. Seki, "Policy controlled system and its model checking," IEICE Trans. Inf. & Syst., vol.E88-D, no.7, pp.1685–1695, July 2005.
[20] N. Nitta, Y. Takata, and H. Seki, "An efficient security verification method for programs with stack inspection," 8th ACM Computer & Communications Security, pp.68–77, July 2001.
[21] F.B. Schneider, "Enforceable security policies," ACM Trans. Information & System Security, vol.3, no.1, pp.30–50, 2000.
[22] D. Volpano and G. Smith, "A type-based approach to program security," TAPSOFT'97, LNCS 1214, pp.607–621, 1997.
[23] J. Wang, Y. Takata, and H. Seki, "HBAC: A model for history-based access control and its model checking," 11th ESORICS, LNCS 4189, pp.263–278, 2006.
[24] H. Seki and Y. Takata, "Comparison of the expressive power of language-based access controls," 第 5 回ディペンダブルシステムワークショップ DSW 2007 論文集, pp.69–73, 2007.

(平成 19 年 7 月 13 日受付, 11 月 1 日再受付)

高田　喜朗　(正員)

平 4 阪大・基礎工・情報卒. 平 9 同大大学院博士後期課程了. 同年奈良先端大・情報助手. 平 19 高知工科大・工・情報講師, 現在に至る. 博士 (工学). ソフトウェアの設計・検証法に関する研究に従事.

王　　静

平 19 奈良先端大・情報・博士後期課程了. 同年電子科技大 (中国)・計算機科学工程学院講師, 現在に至る. 博士 (工学). 計算機セキュリティと形式的検証に関する研究に従事.

関　　浩之　(正員)

昭 57 阪大・基礎工・情報卒. 昭 62 同大大学院博士後期課程了. 同年同大・基礎工・情報助手. 同講師, 同助教授を経て, 平 6 奈良先端大・情報助教授, 平 8 同教授, 現在に至る. 工博. 形式言語理論, ソフトウェア基礎理論に関する研究に従事.