

RBAC モデルの形式検証と修正支援

境 顕宏[†] 鵜林 尚靖[†] 中島 震^{††}

[†]九州工業大学情報工学部

^{††}国立情報学研究所

あらまし 情報システムにおいて機密情報を不正なアクセスから守る手法として、ロールベースアクセス制御 (RBAC: Role Based Access Control) が用いられている。RBAC 記述は組織の改革等により変化していくが、RBAC 記述の不用意な変更は、予期しない箇所に影響を及ぼす場合がある。本稿では、変更前後の RBAC を形式検証することで RBAC 変更の正しさを整形形式、整合性、妥当性の 3 つの観点から検査する方法を提案し、Prolog ベースの検査ツールを提供する。さらに、検査で矛盾が見つかった場合にそれを正しく修正する方法を提案する。

Formal Verification of RBAC Descriptions and Evolution Assistant

Akihiro Sakai[†] Naoyasu Ubayashi[†] Shin Nakajima^{††}

[†]Department of Artificial Intelligence, Kyushu Institute of Technology

^{††}National Institute of Informatics

Abstract RBAC (Role Based Access Control) is an approach to protecting the confidential information from the illegal accesses in a variety of information systems. The RBAC description, once designed, is sometimes modified due to a certain change in the organization. Unexpected defects might sneak in the RBAC description if it is modified in a careless manner. In this paper, we propose a formal definition of the RBAC description checking in terms of three viewpoints; the well-formedness, the consistency, and the validity. The proposed method is implemented in a Prolog-based tool for the automatic checking. We further propose a heuristic approach to finding feasible solutions to remedy the defects.

1 はじめに

情報システムにおいてアクセス制御を用い不正な利用から情報を守る技術が注目されている。ロールベースアクセス制御 (RBAC: Role Based Access Control)[1][4][5] とは、役割に基づいたアクセス制御のことで、ロールごとにパーミッションを割り当てる。ユーザはどのロールに所属しているかにより最終的にアクセス可能なオブジェクトが決まる。しかし、具体的な RBAC 記述が情報システム毎に異なる点、ユーザに直接パーミッションを割り当てない点、ロールの継承の概念により、具体的にどのユーザがどのパーミッションを持っているのかが分かりにくい場合がある。それゆえ現状では、設計

者の経験に頼って現実的な RBAC モデルの設計が行われている。さらに、設計した RBAC は上手く機能していたが、組織改革などで RBAC 記述を少し変更しただけで、思わぬ場所に影響を与え、意図どおりに RBAC が機能しなくなる可能性がある [3][7]。

本研究では、具体的に記述された RBAC が意図どおりに表現できているかを検査する手法を提案する。検査は、RBAC の記述自体に誤りがないかを調べる整形形式検査、記述した RBAC に矛盾がないかを調べる整合性検査、最後に組織内で満たしてほしい事項を性質として与え、それと照らし合わせた際に意図どおりであるかを調べる妥当性検査の 3

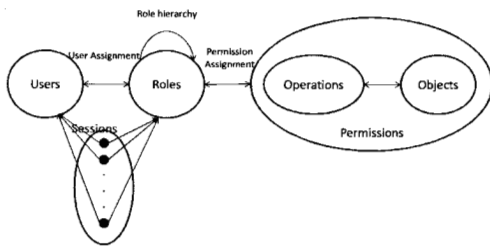


図 1: RBAC のモデル

種類に分類できる。システム管理者が RBAC を変更する前後でこの検査を行い、変更された RBAC が意図どおりに機能するかを判断することができる。また、検査の結果 RBAC が機能しなくなると判断した場合には、正しい記述に修正するための候補を出力することにより、意図どおりに機能する RBAC に修正するための支援を行う。

本稿では、2 節で既存の RBAC 設計に関する問題点を示し、3 節でそれを解決するための検査方法を示す。また、4 節では検査で矛盾が見つかった際の修正支援について述べる。さらに 5 節で議論を行い、6 節で関連研究について述べたあと、7 節で本稿を締めくくる。

2 問題意識

本節では、具体例を用いて現状の RBAC 記述における問題点を整理する。

2.1 ソフトウェア開発現場における RBAC

RBAC[1] は現実社会における人の役割において可能な権限を割り当てることを目的としたモデルであり、ユーザがどのロールに属しているかによって可能なアクセス許可が決まる枠組みである。

RBAC のモデルは、Sandhu らにより RBAC96 モデル [5](図 1) として定式化された。このモデルにおいて Users はアクセスを行う実際のユーザを指し、Role で示した役割に所属する (User Assignment)。また、パーミッションとはどのオブジェクトへどんな操作が許可されるのかを示し、オブジェクトと操作の関係として定義されている。さらに、パーミッショ

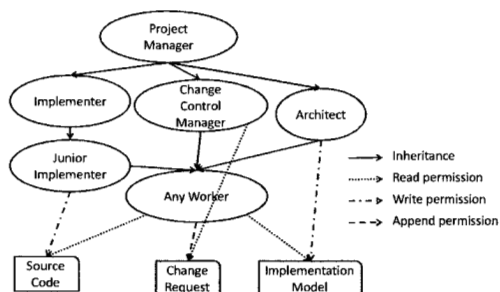


図 2: ソフトウェア開発現場における RBAC

ンの割り当てはロールに対して行われ (Permission Assignment)、その結果、どのロールがどのパーミッションを持つかが決められる。

ここで、ソフトウェア開発現場における例として図 2 のような RBAC を考える。今回は特にラショナル統一プロセス [9] の定義を用いた。成果物として、Source Code, Implementation Model, Change Request を使い、ロールとして Project Manager, Implementer, Junior Implementer, Architect, Change Control Manager, Any Worker を用意する。この RBAC では、操作として書き込み、読み込み、追記の 3 種類があり、操作を認めるパーミッションをそれぞれ Read Permission, Write Permission, Append Permission として図中に破線矢印で表している。例えば、Junior Implementer から Source Code へ繋がっている破線矢印は、Junior Implementer が Source Code への Write Permission を持っていることを意味する。

さらに NIST 標準の RBAC[4] では次のようなオブションが存在する。

ロール間の継承 ロール間の継承関係は、パーミッションの包含を定義することができる。例では、図中の実線の矢印で示しているように Architect が Any Worker を継承している。したがって Architect は Any Worker が持つ全てのパーミッションに加えて Implementation Model に対する書き込みができる。

Static Separation of Duties (SSoD) 同じユーザが同時に持つことができないロールの制約

を指す。例えば **Implementer** と **Architect** 間に SSoD 関係が存在していたとすると、**Implementer** に既に属しているユーザが **Architect** に属することはできない。

この例では、入社間もないプログラマに対しては、**Junior Implementer** というロールを付与し、**Implementer** と区別している。また、セキュリティ要求¹を以下のように定義する。

1. **Any Worker** は、**Change Request** に追記をすることで変更要求を行うことができる。
2. 例外として、**Junior Implementer** に対しては **Change Request** への追記を制限することができる。
3. **Project Manager** は、変更要求の確認と他の成果物への読み書きができる。

図2で示したRBACも、この要求を満たすように構築されており、プロジェクト内でRBACを運用する際は、常にこの要求をみたさなければならぬものとする。

2.2 RBACの変更

前節ではソフトウェア開発現場におけるRBACの例を示したが、組織内で同じRBACを使い続けることはあまりない。これは、組織改革等による役割の変化に合わせてRBACもそれに合う形に変更されるためである[7]。例では、**Junior Implementer** と **Implementer** を区別して扱っているが、若手を含めて実装を行う人は全て **Implementer** として扱うような組織変更があったとする。つまり、**Junior Implementer** を廃止することによりその両者の区別をなくす。ここでRBACに対して行う操作は次の3つである。

1. **Junior Implementer** ロールに属している全てのユーザ割り当てを削除する。
2. 割り当てが削除されたユーザに対して新たに **Implementer** ロールを割り当てる。

¹RBACを運用する企業や組織でRBACを満たすべき性質をまとめたもの。

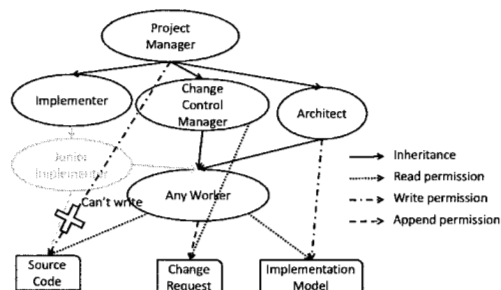


図3: RBAC 変更により要求を満たさなくなった例

3. RBAC から **Junior Implementer** ロールを取り除く。

この操作を行ったあとのRBACの様子を図3に表す。薄く表示された破線はロールの削除に伴って失われたパーミッション割り当て、薄い実線は失われたロール間継承を表している。

ロールの削除による影響は削除したロールに留まらず、RBAC全体に様々な影響を及ぼす。例えば、**Implementer** は削除前は **Any Worker** を継承していたが、削除後には **Any Worker** を継承していないロールになっている。また、削除後の **Implementer** は **Source Code** への **Write Permission** を失っている。さらに、**Project Manager** は **Implementer** の変化の影響を受けるため、**Project Manager** もまた **Source Code** への **Write Permission** を失う。

このようにRBACに対する少しの変更が他の様々なロールに影響を与える中で事前に組織内で定められたセキュリティ要求を満たさなくなってしまう可能性がある。例題のセキュリティ要求では、**Project Manager** は変更要求の確認と他の成果物への読み書きが定められていたが、先ほど述べた通り、**Project Manager** は **Source Code** への書き込みができない。つまり **Junior Implementer** ロールの削除はセキュリティ要求に反する変更だとわかる。

2.3 本研究の目的

前節では、セキュリティ要求に反するRBAC変更の例を示したが、RBAC変更を行ったときに

それがセキュリティ要求通りであるかを調べるのは難しい。これは次の理由からである。

- RBAC に対する変更操作は継承関係によって他のロールやそのロールを付与された人に対して様々な影響を及ぼす。前節では、Junior Implementor ロールの削除により他のロールに対して影響を与える様子を示した。
- RBAC は継承や SSoD の概念をもつことや、一人が複数のロールにつくことが可能であるため、どのユーザがどのオブジェクトにどの操作が可能なのか分かりにくい場合がある。そのため、RBAC 変更時はロールだけではなく、各ユーザが受けるパーミッション変化を追跡する必要がある。

本論文ではこの問題を解決するため RBAC 変更時にそれが RBAC が満たすべき性質を満たしているか、さらに事前に定義したセキュリティ要求を満たしているかを自動検証することで変更が妥当であるかを機械的に判断する方法を提案する。変更が妥当であれば修正の必要がないが、変更の問題があった際は反例を示すことで RBAC 管理者に変更操作が与える影響を伝えることができる。つまり、RBAC 管理者が変更による影響をみながら変更の問題があればそれを正しく修正し、意図通りの RBAC 変更を可能にすることが本研究の目的である。

3 RBAC の検査手法

この節では RBAC 変更時の問題に対する形式検証の手法について述べる。

3.1 検査の概要

本研究が提案している RBAC 検査の流れは図 4 のようになっている。まず RBAC 管理者によって RBAC の変更がおこなわれる。変更を行ったら次にその変更が正しいかを検査器 (1) に渡す。ここで検査器に渡す入力、設計した RBAC と満たしてほしいセキュリティ要求であり、出力は正しい場合はその旨を出力し、正しくなかった場合は反例を出力する。

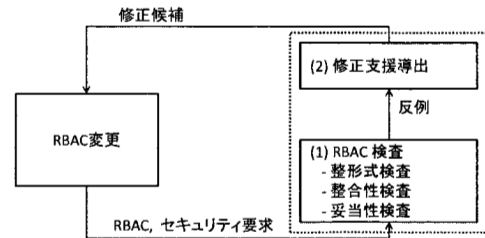


図 4: 検査の流れ

検査器で反例が見つかった場合は、設計した RBAC と満たしてほしいセキュリティ要求を検査器で見つかった反例とともに推論器 (2) に渡す。推論器では矛盾がないように RBAC を修正するための推論を行った後、修正候補を出力する。RBAC 管理者は反例や修正候補を確認しつつ、RBAC を修正したうえで再度検査を行う。

3.2 検査の定式化

この節では RBAC 修正に対して行うべき検査の内容を整理することで、検査法を定式化する。検査はの整形式検査、整合性検査、妥当性検査の 3 段階に分けて行い、順を追って説明する。

3.2.1 整形式検査

入力した RBAC が RBAC 標準定義に沿っているかを調べる検査を整形式検査と呼ぶ。整形式検査で行う検査は次の項目からなっている。

RBAC 構成要素の検査 RBAC の各構成要素が定義を満たすかを検査する。この検査は第 2 節で述べた RBAC モデルに沿っているかを調べることにより行う。例えば、パーミッションが操作とオブジェクトの関係の 2 項関係になっているか等はこの検査に当てはまる。

継承関係の検査 継承関係の定義を満たすかを検査する。RBAC では継承関係は半順序集合でなければならない。

3.2.2 整合性検査

次に整合性検査について説明する。整合性検査は、RBAC の定義に当てはまっているが記述内容に矛盾があるものを調べる。以下に例を示す。

- SSoD に矛盾するユーザ割り当てを検出する。
- 意味をなさない RBAC 記述を検出する。構文上は問題ないが、実際に運用する RBAC としてはふさわしくない場合がある。例えば、同じロール間に SSoD 関係が存在しないか、パーミッションが 1 つも割り当てられていないオブジェクトがないか²をここで検査する。ただし、この検査は対象とする組織によって排除すべき RBAC 記述の意味が変わる場合がある。例えば、Architect と Implementer に SSoD 関係が成り立っているとする。本論文の例の場合、Project Manager のロールを持つことと、Architect と Implementer を兼任することは同じ意味であるので、結果的に SSoD に違反するユーザ割り当てになる。しかしこの例では、Project Manager のセキュリティ要件上、成果物への書き込み権がなければならぬ為、継承関係により SSoD 違反になることを意図した上で Project Manager へのユーザ割り当てをおこなっている。

3.2.3 妥当性検査

最後に妥当性検査について説明する。妥当性検査は、整合性はあるがそれが RBAC 管理者の意図通りでないものを検出する。第 2 節では、はじめに組織内で満たすべきセキュリティ要件を定義した上で、RBAC に対して変更を行い、変更の結果削除したロールの影響が Project Manager に伝播し、セキュリティ要件に反する例を説明した。つまり、特定の組織内で定義された要件があり、検査の際はその要件と照らし合わせて検査を行う必要がある。従って整形形式検査、整合性検査では変更前後の RBAC モデルが入力であったのに対し、妥当性検査ではセキュリティ要件を入力として追加する必要がある。

²これに関しては 5.1 節で述べる。

3.3 Prolog による形式化

この節では、前節で定義した各種検査をどのように行うかについて述べる。本研究で対象としている検査は RBAC の静的な構造に基づいた 3 種類の検査である。また、RBAC モデルは集合により与えられており、その性質を保った上で検査を行う必要がある。従って本研究では集合をリストとして表現した上で検査内容を一階述語論理で与え、Prolog を用いて検査する。

はじめに、RBAC の各要素を Prolog に変換する。以下に Prolog コードの例を示す。

```
user(user_id,user_name). %Users (1)
role(role_id,role_name). %Roles (1)
file(file_id,file_name). %Objects
operation(op_id,operation).
    %Operations Ex: operation(o1,write)
permission(permission_id,
    file_id,op_id) % Permissions (2)
user_assigns([(user_id1,role_id2),
    (user_id2,role_id5), ...]).
    %Set of UA (3)
permission_assigns([
    (permission_id,role_id),
    ...]). %Set of PA (4)
```

Users, Roles, Objects, Operations は関係を持たないのでそのまま (1) の user_id, role_id のようにアトムとして扱う。また、Permissions の定義は $Permissions = 2^{Operations \times Objects}$ であるので、(2) のように Permissions に対応する Operations と Objects の組と一緒に節で表す。ロールへのユーザ割り当ての定義は $UA = Users \times Roles$ の 2 項関係であり、Prolog では (3) のように表記する。これは user_id1 が role_id2 に属し、user_id2 が role_id5 に属していることを意味する。同様に Permission 割り当て ($PA = Permissions \times Roles$) は (4) のように表記する。

上記のコードにより RBAC の構成要素を集合を用いて Prolog で書けたが、上記の変換だけでは整形形式を満たさない場合が存在する。例えば、次のような循環を含む Prolog コードが書ける。

```
parents([(child,father),(father,
    grandfather),(grandfather,child)]).
```

この問題を解決するために、次のように継承関係が半順序集合を満たすかを検査するコードを与

える。つまり、このコードはParents内の任意の2つのロールに対して反射律、反対称律が成り立ち、任意の3ロールに推移律が成り立つことを確認することを確認している。

次に整合性検査を行う。整合性検査ではSSoDに違反するユーザ割り当てや同一ロール間でのSSoDのチェック等、それぞれに対して検査できるような述語を用意しておく。例としてSoDに違反するユーザ割り当てを検出する述語を示す。このコードではconflicts述語の中で定義されたSSoD関係の集合がロールへのユーザ割り当てに対して矛盾を含んでいないかを確認することにより検査を行っている。

```
conflicts((r1,r2),(r2,r5), ...).
%Set of SSoD Relation
check_ssod(Role1,Role2,User,UA,Conflicts) :-
    role(Role1,_), role(Role2,_),
    user(User,_),
    member((Role1,Role2),Conflicts),
    member((User,Role1),UA),
    member((User,Role2),UA).
check_ssod(Role1,Role2,User) :-
    conflicts(Conflicts), user_assigns(UA),
    check_ssod(Role1,Role2,User,UA,Conflicts).
```

最後に妥当性検査を行う。妥当性検査は、セキュリティ要求を与えた上で矛盾しないことを確かめる必要があるため、セキュリティ要求を一階述語論理として与えられるようにあらかじめ決められた形で与える必要がある。したがって、RBAC管理者がセキュリティ要求を表現するための記述言語を用意する。これを以下インターフェイスと呼ぶ。

- ロールRがあるパーミッションPを持つ。
- ロールRがあるパーミッションPを持たない。
- ユーザAがロールRに属している。
- ユーザAがロールRに属していない。

このインターフェイスを用いて定義したセキュリティ要求のコードを示す。この例はロールr1がパーミッションp1を持っていないなければならないことを意味している。

```
policy(r1,p1,positive):-permission(p1, _,_).
```

4 修正支援

検査器による3つの検査で反例が出た場合、推論器は反例をもとに修正するための候補を導出する。本節ではその導出の流れについて説明する。

アクセス制御ポリシーを考える際には、常にPrinciple of Least Privilege (PLP)[1]を意識する必要がある。すなわち、最小かつ不可欠な権限だけを与えて、余計なアクセス権限を与えないようにしなければならない。本研究の推論器では、パーミッションを中心に考えることでPLPに沿った修正案を提示する方針を採用する。

推論器はパーミッションが不足していることによりセキュリティ要求に対する反例が見つかれば、セキュリティ要求に合うように、permission.assignsのリストに追加を行うことでパーミッション割り当てを推論する。例えば図5は第2節の例題に対する推論結果を表している。Project ManagerがSource Codeへの書き込みができないことがセキュリティ要求に対する反例であったため、破線(a)のようにpermission.assignsにImplementerがSource Codeに書き込みを追加する。同様に他の修正候補として他の候補(b)(c)(d)(e)と共に出力する。

さらに、修正候補出力する際はヒューリスティック値を用いて順位づけを行う。これは、修正候補の中からRBAC管理者が何を選べばよいのかを判断する際に用いることでより正しい修正を行うことを目的としている。

したがって修正支援では、余計なパーミッションが与えられたり、逆に必要なパーミッションが失われているかを考慮する。例えば、修正候補の中から(d)を選んだとすると、Junior Implementer削除前と比べてAny WorkerがSource Codeに新たに書き込みができるようになるため、セキュリティ上好ましいとは言えない。一方で、修正候補の中から(b)を選んだとすると、削除前と比べてImplementerがSource Codeへ書き込みを行うことが不可能になってしまう。

以上を踏まえ、修正候補の中から解を絞り込むにあたっては、修正後に他のユーザやロールに与える影響を考慮する。具体的には、はじめに各々の修正候補に対して次の2つを求める。

- 修正後に他ロールで新たに加わったパーミッ

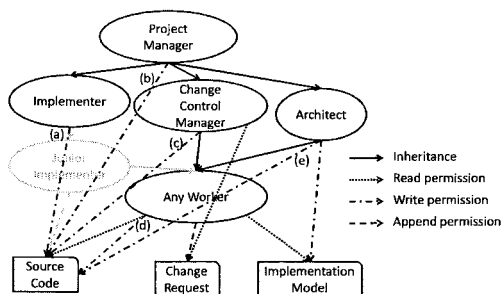


図 5: 修正候補の導出

シヨンの数.

- 修正後に他ロールで失われたパーミッションの数.

次に、これらが最も少なくなる修正候補を最終的に選ぶ。つまり例題では、図 5 の 5 つから (a) が最適解として選ばれる。

5 議論

5.1 変更による矛盾の導出

前節の実験で行った 3 種類の検査による矛盾の導出に関する考察を行う。整形形式検証に関しては RBAC 定義と一対一に対応するため高い信頼性のもとも機械的に行うことができる。

次に、整合性検査に関しては SSoD のように検査内容が予め明確なものもあるが、検査法に工夫を要する項目も存在する。例えば、今回用いた例題において、Implementer は RBAC の変更後に Source Code の書き込みができなかったが、変更後は RBAC 上の全てのロールに Source Code の書き込みは不可能であった。このように、全てのリソースに対して書き込みが行えないケースも意味のないロールの割り当て方と捉えれば整合性検査の段階で矛盾を導出することが可能であった。本研究では検査法に工夫を要する項目に対する整合性検査を用意してはなかった。しかしこの点に関しては今後、記述実験等を繰り返すことによって拡張可能であると考えている。

また、組織においてセキュリティポリシーというものは用語の使われ方から内容まで様々であり、

その内容を直接使って RBAC の妥当性検証を行うことは難しい問題があった。これを解決するため本研究では、RBAC 変更前後で満たすべき性質を一階述語論理へと変換可能なインターフェイス導入し、RBAC 管理者がセキュリティ要件をインターフェイス用いて与えることでセキュリティポリシーから妥当性検査へと結びつけた。

5.2 修正支援候補の導出アルゴリズム

この節では修正支援に関する考察を行う。今回は、検査によって矛盾が導出された場合、パーミッション割り当てを追加または削除することにより修正の支援を行った。パーミッション割り当てによる推論の場合、リストを操作した後で矛盾が起きないかを再検査すると共に、他のロールやユーザーに与える影響を計算した上でヒューリスティック値を求めた。この方法により修正後、他のロールに対し余計なパーミッションが与えられる可能性を減らし、PLP を満たす修正候補を得ることができた。一方で、パーミッション割り当て以外での修正の支援も考えられる。例えば継承関係の追加や削除による支援や、その両方を用いた支援等である。計算量に関しては、変更を行った箇所付近に着目して探索の範囲を狭めることで今後は規模が大きい性能評価をしつつ、実用性のある時間内で修正候補を推論するアルゴリズムを考える予定である。

6 関連研究

本研究では、RBAC 変更時の検証について提案したが、これに関連した研究としては次のようなものがある。

Schaad ら [6] は、仕様記述言語 Alloy[2] を用いて RBAC96 のメタモデルやその拡張を Alloy で解析できることを示した。Alloy は一階述語論理をベースにした仕様記述言語で主に静的な構造検証に適した仕様記述言語である。

また Zao ら [8] は、RBAC96 のスキーマを Alloy で記述し検証した。彼らはこの論文の中でアクセス制御システムの仕様記述として Alloy が有効であることを示した。

本研究ではRBACに対する静的な検証というアプローチではこれらの研究に近いところがある。しかし、これらの研究では主にRBAC記述の構造検査の段階にとどまっている。これに対し本稿ではRBACの検査を整形形式検査、整合性検査、妥当性検査と3つに分類することで、RBACの変更操作に対し、セキュリティ要件を踏まえた妥当性までを検査可能にした。また本研究では、矛盾する反例から要件を満たすRBACを構成するための推論方法を提案し、修正支援を行っている。

7 おわりに

本論文ではRBACに変更を加えた際に検査を行うことで、変更後のRBACが意図どおりに機能するかを確かめる手法を提案した。さらに、意図どおりに動かないと判定された場合は、意図どおりに動くような修正候補を出力することで、正しいRBAC記述に戻す支援を行う手法を提案した。

今後は現実的なRBACへの適用実験を行うことでの性能評価や、それを通して整合性検査における検査項目の充実化が挙げられる。また、変更が重なって起きた場合に履歴情報を使って修正候補を選ぶ等、複数の修正候補から最適解を選ぶアルゴリズムの改良などが課題である。

参考文献

- [1] Ferraiolo, D., Kuhn, D., Chandramouli, R., *Role-Based Access Control, 2nd Edition*, Artech House, 2007.
- [2] Jackson, D., *Software Abstractions: Logic, Language, And Analysis*, The MIT Press, 2006.
- [3] Pistoia, M., Fink, J., Flynn, J., Yahav, E., "When Role Models Have Flaws: Static Validation of Enterprise Security Policies" *Proc. 29th International Conference on Software Engineering (ICSE)*, pp.478-488, 2007.
- [4] *Role-Based Access Control*, ANSI INCITS 359-2004, American National Standard for Information Technology, 2004.
- [5] Sandhu, R., Coyne, E., Feinstein, H., Youman, C., "Role-Based Access Control Models", *IEEE Computer*, vol.29, no.2, pp.38-47, 1996.
- [6] Schaad, A., Moffett, J., "A lightweight approach to specification and analysis of role-based access control extensions", In *Proc. 7th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pp.13-22, 2002.
- [7] Schaad, A., Moffett, J., Jacob, J., "The Role-Based Access Control System of a European Bank: A Case Study and Discussion", *Proc. 6th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pp.3-9, 2001.
- [8] Zao, J., Wee, H., Chu, J., Jackson, D., "RBAC Schema Verification Using Lightweight Formal Model and Constraint Analysis", MIT, Cambridge, 2002.
- [9] フィリップ・クルーシュテン, ラショナル統一プロセス入門, ピアソンエデュケーション, 2001.