ポリシ管理システムのための検査手法

A Check Technique for Policy-based Management Systems

前田 直人 †, 猪鹿倉 知広 †, 登内 敏夫 †
Naoto MAEDA, Tomohiro IGAKURA and Toshio TONOUCHI

†NEC インターネットシステム研究所
Internet Systems Research Laboratories, NEC Corporation
n-maeda@bp.jp.nec.com, t-igakura@bx.jp.nec.com, tonouchi@cw.jp.nec.com

ポリシ管理システムは，障害への迅速な対応，運用コスト削減，などを実現する管理手段として注目を集めている．管理システムの不具合は管理対象システムの信頼性を損なう要因となるため，ポリシ管理システムに高い信頼性が要求される．本稿では，ソフトウェア検証において広く利用されている，事前条件，事後条件，不変条件の記述をポリシ管理に導入した，ポリシ検査手法を提案する．提案手法は，これら条件記述を用いて，ポリシ記述の登録前に不正なポリシ記述を検出/排除し，実行時には，システムが予期せぬ振る舞いをしていないことを確認する．

1 はじめに

ポリシ管理システム [3] は，与えられたポリシ記述に従って，管理対象システムを半自動的に監視管理する管理システムであり，複雑化するシステム障害への迅速な対応や，増大する運用コストの削減を実現する管理手段として，注目を集めている．

一方，ポリシ管理システムは，管理対象システムの状態に応じて，複数のポリシ記述が並列に処理される複雑なシステムである．時々刻々と変化する管理対象システムの状態に対して，あるポリシが，常にそのポリシを記述した人間の意図通りに機能するか，あるいは，複数のポリシの作用の結果，意図せぬ問題を管理対象システムに引き起こすことはないか，などのポリシ記述の問題を，ツールの支援なしに検査することは困難である．

さらに，ポリシ管理システムを実際に運用する場面において，ポリシ記述者は複数存在する可能性がある．また，管理対象システムの管理者とポリシ記述者は異なるかもしれない．

このような理由から，ポリシ間の不整合や，管理対象システムの整合性を損なうような不適切なポリシ記述を，静的/動的に検出する手法が活発に研究されている [1, 2, 6, 7, 8].

本稿では，ソフトウェア検証において広く利用されている，事前条件，事後条件，不変条件の記述をポリシ管理に導入した，ポリシ検査手法を提案する．管理対象システムの状態を，CIM(Common Information Model)[5] のようなオブジェクト指向型の情報モデルとして取得できることを前提とする．各条件は，管理対象システムの状態に対する一階述語論理で記述し，ポリシ間の問題と，ポリシと管理対象システム間の問題を，論理式の充足可能性を判定することにより検出する．また，事後条件，および不変条件は，実行時に，ポリシ記述が想定通りの作用を管理対象に与えたか，管理対象システムは不正な状態に陥っていないか，を検査するためにも利用される．

2 ポリシ管理システムの仕様

2.1 条件記述

本手法では，if-then 型のポリシに対して，事前条件，事後条件を定義し，また，管理対象システムにおいて常に成立するべき条件を不変条件として定義する．事前条件は，ポリシの then 節に定義された操作をいつ管理対象システムに適用するか，を指定するための条件であり，ポリシの if 節に記述される発火条件に対応する．事後条件は，if-then 型ポリシの then 節に記述される管理対象への操作に対応し，ポリシに定義された操作直後の管理対象の状態が満たすべき条件を示す．不変条件は，ポリシの適用と関係なく，管理対象システムにおいて常に成立するべき性質を，システムの状態に対する条件として表現したものである．

なお，ポリシ管理システムは，時々刻々と変化する管理対象システムの状態を監視し，当該状態において発火条件が成立するポリシのうち一つ，あるい

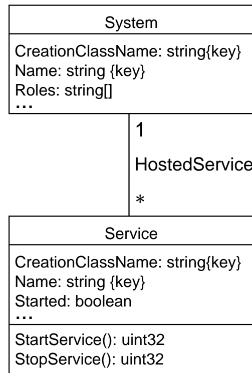

図 1: CIM の例: System と Service

は該当する全てのポリシの操作を管理対象システムに適用する．

管理対象の状態は，オブジェクト指向型の情報モデルによって表現されるものとし，特に本稿では CIM を仮定する．管理対象システムの各構成要素の状態は，ある特定の性質を持ったクラスのインスタンスにより表現され，各要素間の関係は Association と呼ばれる特殊なクラスのインスタンスにより表現される．図 1 に CIM クラスの例を示す．計算機のような管理対象を表す System クラスは，その管理対象が提供するサービスを表現する Service クラスに 1 対多の関係を持ち，その関係は Associaion である HostedService クラスのインスタンスにより保持される．

事前条件，事後条件，不変条件は，管理対象の状態を表現した CIM のインスタンスに対して，一階述語論理で記述する．以下に記述例を示す:

$$\exists s \in \texttt{Service}, c.\texttt{Name} = \text{“}\texttt{mysql}\text{''} \wedge c.\texttt{Status} = \text{“}\texttt{OK}\text{''}$$

$x \in C$ という記述は，x が，管理対象システムの状態に含まれる C という名前を持つクラスのインスタンス集合の要素であることを示す．$x.p$ は p という名前の属性値参照を示す．上記の例は，MySQL サービスが少なくともひとつは起動していることを示す．

管理対象の状態に対する条件を記述するために，関係演算子 $=$, $\neq$, $>$, $\geq$, $<$, $\leq$ と，インスタンス間の関連を示す $connected(x,y,z)$ という述語を利用する．$connected(x,y,z)$ は，インスタンス x とインスタンス y が Association クラスのインスタンス z によって関連付けられていることを示す．

また，特定の性質を持ったインスタンス集合を以下のように記述する:

$$\{s \in \texttt{CommonDataBase} \mid s.\texttt{Name} = \text{“}\texttt{MySQL}\text{''}\}$$

上記の式は，`CommonDataBase` のインスタンスであって，`Name` という属性名が “`MySQL`” であるようなインスタンスの集合，を示す．以降，この種の記述を集合記述と呼ぶ．集合記述は，集合の要素数を返す関数 `card` の引数として用いられる．

2.2 記述例

記述例を図 2 に示す．事前条件の例では，`Shopping` という名前を持つアプリケーションの応答時間が 1 秒以上であるときに発火するポリシの条件を示す．`UnitOfWork` は，分散システムの各種メトリクスを表すためのクラスの一つであり，`LogicalElementPerformsUoW` という Association クラスにより，どのシステムの性能を示すものであるか関係付けられる．なお，$\geq_t$ は時間比較用の述語であり，`Status` $= 4$ は，正常に処理が完了した応答であることを示す．事後条件の例は，MySQL サーバが障害により停止した際に，そのサーバを起動させるポリシの作用を示す．不変条件の例は，負荷に応じてサービス提供サーバの台数を増減させるようなシステムにおいて，負荷が少ない場合に最低限確保しておくサーバの台数を指定している．

3 検査手法

本節では，ポリシ管理システムに登録する前にポリシ記述を検査する方法と，ポリシ管理システム実行時に，システムが予期せぬ振る舞いをしていないか確認する方法を説明する．

3.1 ポリシ記述の静的検査

まず，登録前にポリシ記述を検査する方法を述べる．検査は，ポリシ記述間の不整合検出と，管理対象システムに不適なポリシ記述の検出とから構成される．なお，述語 *connected*，および集合記述を含む論理式の検査は今後の課題であり，静的検査の対象外とする．

3.1.1 ポリシ記述間の不整合検出

本手法では,「同時に発火する可能性があり，かつ，同時に実行できない可能性がある」という条件を満たすポリシ記述の組合わせを不整合と見なし，そのような組合わせを検出する．

検出方法を説明する．与えられたポリシのペアを p_i と p_j，ポリシ p の事前条件を p^{pre}，事後条件を

事前条件の例: Shopping アプリケーション・システムの応答時間が 1 秒以上
$\exists w \in \texttt{UnitOfWork}, \exists s \in \texttt{ApplicationSystem}, \exists a \in \texttt{LogicalElementPerformsUoW},$
$w.\texttt{Status} = 4 \wedge s.\texttt{Name} = \text{“}\texttt{Shopping}\text{''} \wedge connected(w,s,a) \wedge w.\texttt{ElapsedTime} \geq_t 1000$

事後条件の例: MySQL サービスの状態が OK
$\exists s \in \texttt{Service}, c.\texttt{Name} = \text{“}\texttt{mysql}\text{''} \wedge c.\texttt{Status} = \text{“}\texttt{OK}\text{''}$

不変条件の例: Pet Store サービスには 2 台以上のコンピュータが割り当てられている
$\texttt{card}\{ c \in \texttt{ComputerSystem} | \exists a \in \texttt{HostedService}, \exists s \in \texttt{J2eeDeployedObject},$
$connected(c,s,a) \wedge s.\texttt{DeploymentDescriptor} = \text{“}\texttt{petstore}\text{''}\} \geq 2$

図 2: 条件の記述例

p^{post}，すべての不変条件を $\wedge$ 結合した論理式を inv と記述する．ポリシ間の不整合は，基本的に，以下の手順で検査する:

1. **同時発火検査**
$p_i^{pre} \wedge p_j^{pre} \wedge inv$ が充足可能であれば，ポリシ p_i と p_j が同時に発火する可能性があると判断する．

2. **実行可能性検査**
同時に発火可能なポリシの組合わせについて，次に，$p_i^{post} \wedge p_j^{post} \wedge inv$ が充足可能か調査する．充足不可能な場合，ポリシ P_i と P_j の操作は同時に実行できないと判定され，ポリシ間の不整合として報告される．

さらに，論理式中に登場する束縛変数に対応づけられたインスタンスが同一であるか否かを判定する制約式を追加する．以下の例を用いて説明する:

$$P_i \equiv \exists s \in \texttt{S}, s.\texttt{p1} = \text{“}\texttt{mysql}\text{''} \wedge s.\texttt{p2} = \text{“}\texttt{OK}\text{''}$$
$$P_j \equiv \exists s \in \texttt{S}, s.\texttt{p1} = \text{“}\texttt{mysql}\text{''} \wedge s.\texttt{p2} = \text{“}\texttt{NG}\text{''}$$

P_i と P_j を，それぞれ異なるポリシの事前条件とする．この時，$P_i \wedge P_j$ は，S というクラスの属性のうち p1 が同じ値であり，p2 は異なる値を取るインスタンスがそれぞれ存在すれば充足可能となる．特に制約を加えなければ，このような組合わせの事前条件は常に充足可能，すなわち同時に発火すると判断される．そこで，S.p1 に同じ値をもつインスタンスは同一であるという制約を導入する．すると，P_i と P_j を同時に充足することが不可能になる．

ここでは CIM で定義されている Key 属性 [5] を利用して，このような制約を追加する．Key 属性を持つクラスのサブクラスは Key 属性を新たに定義することはできない．Key 属性は，Key 属性を定義したクラス，およびそのサブクラスの中でインスタンスを一意に示す値を持つ．Key 属性が複数ある場合には，組合わせた値が唯一となる．このように，CIM の定義によると，Key 属性によりインスタンスを識別することが可能である．

以下に，Key 属性を用いてインスタンス等価性に関する制約を追加する方法を示す:

1. $P_i \wedge P_j \wedge inv$ を冠頭標準形に変換し，この式を L とする．

2. 論理式に登場するクラスのうち，当該クラスのインスタンスに対応づけられた束縛変数の個数が 2 以上あるようなクラス C と，当該クラスに対応する束縛変数のリストと，をまとめた表を作成する．

3-A 同時発火検査の場合:
得られた表の各 C に対して次の処理を行う．まず，C に対応する束縛変数から可能なすべての組合わせを作成する．次に，各組み合わせについて，以下に示す条件を L に追加する．なお，組み合わせた変数名をそれぞれ x と y と記述し，$keys$ はクラス名から当該クラスに定義された Key 属性名の集合を返す関数とする．

$$(\bigwedge_{p \in keys(C)} x.p = y.p) \leftrightarrow x = y$$

3-B 実行可能性検査の場合:
関数 $keys$ の代わりに関数 $keys2$ を利用する点を除いて，同時発火検査の場合と同様に，以下に示す条件を L に追加する．$keys2$ は当該クラスに定義された Key 属性のうち，L の中で変数 x と y の両方から参照される Key 属性名のみを要素とする集合を返す関数とする．また，keys2 の返り

値が空集合の場合，$(\bigwedge_{p\in keys2(C,x,y)} x.p = y.p)$ は真とする．

$$(\bigwedge_{p\in keys2(C,x,y)} x.p = y.p) \leftrightarrow x = y$$

以上の手続きにより得られた論理式を用いて充足可能性を判定することで，ポリシの不整合を検出する．

実行可能性検査では，インスタンスが同じか否かを与えられた条件から判断できない場合に，同じインスタンスであると見なすように，等価性の制約を与えている．これは，同時に実行できない可能性があるポリシの組合わせをより広く検出する為である．

3.1.2 管理対象に不適なポリシ記述の検出

事後条件は，ポリシに定義された操作を適用直後に，管理対象システムの状態について成立するべき条件が記述されている．一方，不変条件は，ポリシ適用前も適用後も，常に管理対象システムにおいて成立すべき性質を示す．したがって，不変条件と同時に成立しない事後条件をもつポリシは，管理対象システムに不正な作用を持つ可能性がある．

そのような不正なポリシを検出するために，与えられたポリシの事後条件と不変条件の連接とを $\wedge$ 結合し，さらに，節 3.1.1 で述べたインスタンスの等価性制約（3-B）を $\wedge$ 結合した論理式の充足可能性を判定する．充足不可能な事後条件を持つポリシは，管理対象システムに不正な作用を持つと判断する．

本検査で検出されるポリシの事後条件は，先の事後条件間の不整合検出でも同様に発見される．しかし，ポリシと事後条件間の不整合と事後条件間の不整合のどちらに起因するのか判断できない．したがって，静的検査では，まず事後条件と不変条件の不整合検出を行い，不適切なポリシを取り除いた後，事後条件間の不整合検出を行う．

3.2 実行時検査

次に，ポリシ管理システム実行中に行う検査について説明する．

ポリシの事後条件は，当該ポリシのアクション実行後に管理対象システムにおいて満たされるべき性質を示す．ポリシ管理システムは，ポリシの操作を管理対象に適用後に，当該ポリシの操作に対応した事後条件が管理対象システムの状態について成立するかを判定することにより，ポリシの作用がポリシ記述者の意図通りであったことを確認する．

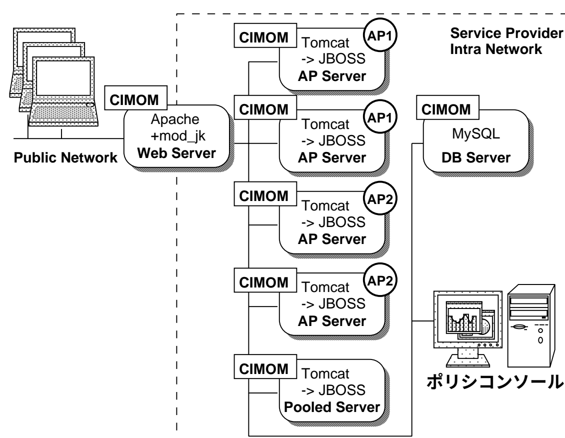

図 3: 実験用 Web アプリケーションの構成

また，不変条件は管理対象システムの状態が変化しても常に成立するべき性質を示す．実行中，ポリシ管理システムは，定期的に不変条件の成立を検査することにより，管理対象システムが想定した安全な状態にあることを確認する．

4 事例: Web3 層システムの管理

我々は，提案手法を実装した独自のポリシ型管理システムを開発し，クラスタ化した Web3 層システムを管理対象として実験を行った．本節では，事例に基づく手法の説明と，その実験結果を報告する．

4.1 実験シナリオと管理対象システムの構成

実験シナリオでは，複数のサービスを提供するサービスプロバイダのシステムを想定する．資源をより効率的に利用するために，サービス提供サーバの負荷に応じて，サーバの台数を増減させる．

図 3 に実験用の管理対象システムの構成を示す．典型的な Web3 層システムであり，各サーバの OS には RedHat9.0，クラスタ化したアプリケーション・サーバには jboss4.0.2 を採用した．各サーバは Switching Hub 経由で 100Mbps Ethernet で接続されている．また，管理対象システムの状態を CIM 準拠の形式で提供する CIMOM(CIM Object Manager) として OpenPegsus2.5.0 を，Linux 用の CIM プロバイダとして SBLIM パッケージを採用した．

4.2 静的検査

実験には，定理証明ツール Simplify1.5.4 利用した．Simplify[4] は一階述語論理 P を与えられると，P の否定の充足可能性を判定し，充足不可能な場合に P が正しいと結論づける．そこで，本検査では $P_i \wedge P_j$ の充足可能性を判定するために，Simplify に $\neg(P_i \wedge P_j)$ が正しいかを判定させる．Simplify が与えられた論理を正しいと判定した場合，否定をはずした論理式，すなわち $P_i \wedge P_j$ が充足不可能であることを意味する．

節 3.1.1 に示した MySQL に関する条件 P_i と P_j を利用して，Simplify を利用した充足判定方法を説明する．Simplify は，S 式で記述された整数に対する条件を解析する．そこで，論理式に含まれる定数文字列はハッシュ値などを用いて整数に変換する．さらに，論理式中に登場する属性値参照は，クラス名と属性名を結合した名前を持つ関数呼び出しに変換する．先の事後条件を判定するために Simplify に与える式は以下の通り:

```
(NOT
  (EXISTS (s t)
    (AND
      (AND (EQ (S_p1 s) 1)
           (EQ (S_p2 s) 2))
      (AND (EQ (S_p1 t) 1)
           (EQ (S_p2 t) 3))
      (IFF
        (EQ (S_p1 s) (S_p1 t))
        (EQ s t)))))
```

この論理式は Simplify により valid と判定され，前述の 2 つの条件が充足不可能であることがわかる．

なお，充足判定を行う論理式に全称記号が含まれると，多くの場合，Simplify は充足判定に失敗する．現時点では，本検査が適用可能な対象は限られており，検査能力の向上は今後の課題である．

4.3 実行時の検査

ポリシ記述には，事後条件と，事後条件が成立しなかった場合の対応（ポリシ再適用，管理者に通知，など）が含まれる．事後条件は，ポリシの操作適用後，指定した時間後に評価され，成立しなかった場合，ポリシ記述に定義された対応処理が実行される．

実験用に次の 2 種類のポリシ記述を作成した: (1) サービス提供サーバ全体の負荷が一定の値を越えたら，予備サーバよりサーバを確保し，当該サービスをデプロイして，サービス提供サーバの台数を増やす，(2) サービスの各種データを記憶する MySQL サーバが停止していたら再起動させる．

ポリシ記述 (1) の事後条件は以下の通り:

$$\forall \mathtt{o} \in \mathtt{MonitoredSet},\ \mathtt{o.PctTotalCPUTime} < 60$$

ポリシ管理システムは，ポリシ記述を登録すると，ポリシの管理対象を特定し，監視を行う．MonitoredSet は，その管理対象（監視対象）の状態を示す CIM インスタンスの集合を表す．ポリシ (1) の場合，管理対象は，サービス提供サーバの各 Linux_OperatingSystem インスタンスとなる．上記条件は，サーバ追加の効果により CPU 負荷が 60% 未満になることを示す．

ポリシ記述 (2) の事後条件は，図 2 の事後条件と同様である．

次に，不変条件の検査を説明する．我々のポリシ管理システムは，登録された不変条件が，管理対象システムの現在の状態について成立するか否かを，一定間隔で検査し，違反が見つかった場合，管理者に通知する．

評価のため，実験環境下で，もっともインスタンス数が多い "プロセス" を利用して，以下に示す不変条件を作成した．本条件は「データベース・サーバ以外で MySQL が動作していないこと」を示す．なお，本条件が真になる環境下で，3 回評価時間を計測した．

$$\begin{aligned}&\forall \mathtt{p} \in \mathtt{UnixProcess}, \exists \mathtt{o} \in \mathtt{OperatingSystem},\\ &\quad \exists \mathtt{a} \in \mathtt{OSProcess}, \mathtt{p.Name} = \text{``}\mathtt{mysqld}\text{''} \rightarrow\\ &\qquad (\mathtt{o.Name} = \text{``}\mathtt{dbsrv}\text{''} \wedge \mathtt{connected}(\mathtt{p},\mathtt{o},\mathtt{a}))\end{aligned}$$

条件評価のために取得されたインスタンス数は，OperatingSystem が 7 個，UnixProcess と OSProcess はそれぞれ 536 個である．3 回の評価で個数は変化しなかった．前記インスタンス・データの取得時間は 9.07 秒かかり，条件の評価時間は 2.10 秒であった．時間は 3 回の平均である．現在の Pegasus の実装では複数クラスのインスタンス情報をまとめて取得することができないため，情報取得に時間がかかる．また，現在の実装では論理式を効率的に評価するよう考慮されていない．CIMOM および，評価方法を改善することで，評価時間を短縮することは可能である．現在の実装であっても，1 時間に一度程度の間隔での評価ならば実用に耐えうると考える．

5 関連研究

ポリシ記述間の不整合はポリシ衝突 (Policy Conflicts) と呼ばれ，適用ドメインに非依存の *Modality Conflict* と，ドメインに依存する *Application Specific Conflict* の 2 種類に分類される [6]．Modality Conflict は，例えば,「A は B をして良い」という許可ポリシと「A は B をしてはならない」という禁止ポリシ間で発生するような論理的矛盾を示す．一方，Application Specific Conflict の場合，どのようなポリシ記述の組合わせを衝突と見なすかは，適用ドメイン毎に定義される．そこで，与えられたポリシ記述の集合全体，あるいは組合わせを対象に，あらかじめ定義した衝突規則を満たすかどうかを検査することで，衝突を検出する方法が提案されている [2, 7, 8]．この方法は，問題がある組合わせをあらかじめ与えておく必要があるため，不十分な衝突ルール記述による問題の見逃しが課題となる．我々の手法では，述語論理の充足可能性により不具合を判断する．これにより，ルールを与えることなしに，Modality Conflict より複雑な問題を検出できる．

文献 [1] では，与えられたポリシ記述が既に登録されているポリシ記述と衝突しないか，あるいは，登録されているポリシ記述の集合は適用対象に対して網羅的か，などの問題を，我々の手法同様に，述語論理の充足可能性により検出する．しかし，ここで扱われる問題は，時刻やパラメタの大小関係のみであり，本稿で対象とするオブジェクト指向型の情報モデルを利用した場合の検査方法については考慮されていない．

6 まとめと今後の課題

本稿では，ポリシ記述に対応する事前/事後条件と，管理対象システムに対応する不変条件とを用いて，ポリシ登録時に不正なポリシ記述を検出/排除し，実行時にはシステムが望ましい状態を保っていることを確認する検査方法を述べた．

本手法では，管理対象クラスの各属性の値が守るべき条件を不変条件として与えることにより，システムを異常な状態に導くような不正な値の組み合わせを検出し，障害を防止することができる．また，ポリシ記述に事後条件を与えることにより，実行時に，ポリシ記述が意図通りの作用を管理対象に与えたことを確認できる．これら実行時の検査は，ポリシ管理システムを安定して動作させるために重要である．

また，障害を未然に防止するには，まず，不正なポリシ記述を検出/排除することが肝要である．本稿では，充足可能性により不正なポリシ記述を検出する方法を示した．しかし，提案方法は，検査可能な論理式の種類が限られており，我々が実行時検査に利用している多くの条件式を検査できない．ポリシ記述の静的検査手法の改善が今後の課題である．

謝辞

本研究は、総務省からの委託研究の成果である。

参考文献

[1] D. Agrawal, J. Giles, K.W. Lee, K. Voruganti and J. Lobo: Policy Ratification, in *Proc. of IEEE Policy'05*, Jun., 2005.

[2] M. Charalambides, P. Flegkas, A. K. Bandara, E. C. Lupu, A. Russo, N. Dulay, M. Sloman and J. Rubio-Loyala: Policy Conflict Analysis for Quality of Service Management, in *Proc. of IEEE Policy'05*, Jun., 2005.

[3] N. Damianou, N. Dulay, E. Lupu, M. Sloman and T. Tonouchi: Tools for Domain-based Policy Management of Distributed Systems, in *Proc. of NOMS2002*, Apr., 2002.

[4] D. Detlefs, G. Nelson and J. B. Saxe: Simplify: A Theorem Prover for Program Checking, *Journal of the ACM*, vol. 52, no. 3, pp. 365-473, May, 2005.

[5] DMTF:*Common Information Model Infrastructure Specification Ver.2.3 Final*, Oct., 2005.

[6] E. C. Lupu and M. Sloman: Conflicts in Policy-Based Distributed System Management, *IEEE Trans. on Software Engineering*, vol. 25, no. 6, Nov., 1999.

[7] N. Maeda and T. Tonouchi: An Analysis Method for the Improvement of Reliability and Performance in Policy-Based Management Systems, in *Proc. of DSOM2004*, LNCS3278, 2004, pp. 147-158.

[8] C. Shankar, A. Ranganathan and R. Campbell: An ECA-P Policy-based Framework for Managing Ubiquitous Computing Environments, in *Proc. of Mobiquitous2005*, Jul., 2005.