An Analysis Method for the Improvement of Reliability and Performance in Policy-Based Management Systems

Naoto Maeda and Toshio Tonouchi

NEC Corporation, 1753 Shimonumabe, Nakahara-ku, Kawasaki 211-8666, Japan
n-maeda@bp.jp.nec.com, tonouchi@cw.jp.nec.com

Abstract. Policy-based management shows good promise for application to semi-automated distributed systems management. It is extremely difficult, however, to create policies for controlling the behavior of managed distributed systems that are sufficiently accurate to ensure good reliability. Further, when policy-based management technology is to be applied to actual systems, performance, in addition to reliability, also becomes an important consideration. In this paper, we propose a static analysis method for improving both the reliability and the performance of policy-based management systems. With this method, all sets of policies whose actions might possibly access the same target entity simultaneously are detected. Such sets of policies could cause unexpected trouble in managed systems if their policies were to be executed concurrently. Additionally the results of the static analysis can be used in the optimization of policy processing, and we have developed an experimental system for such optimization. The results of experimental use of this system show that an optimized system is as much as 1.47 times faster than a non-optimized system.

1 Introduction

Policy-based management shows good promise for application to semi-automated distributed systems management. It enables system managers to efficiently and flexibly manage complicated distributed systems, which are composed of a large number of servers and networks. This results in dramatic reductions in system management costs.

The reliability and performance in the policy-based management systems are essential issues when applying this kind of technology to actual systems. Flaws in a management system will degrade the reliability of the managed system, and poor performance may offset the advantage initially gained by using a policy-based technology: the ability to adjust rapidly to a changing situation.

Tool support is indispensable to managers who wish to create policies that are sufficiently correct to ensure reliability. Such tools check the properties of given policies, such as type-checking equipped with programming language compilers. Recently, methods for detecting and resolving *policy conflicts* have been studied actively[2,6,8,9,11]. Policy conflicts can be categorized into a number of different types, of which there are two major groupings: *modality conflicts* and *application specific conflicts*[9,11]. Modality conflicts can be detected by purely syntactic analysis using the semantics of policy specification languages[9]. Application specific conflicts, by way of contrast, are defined by application semantics as the name suggests. As a way of providing a generic way to cope with application specific conflicts, approaches using constraint-based rules have been proposed in [2,9].

In this paper, we propose a static analysis method for detecting all sets of policies whose actions might possibly access the same target entity at the same time. We call such sets of polices *suspicious policy sets*. The exact conditions for these sets will be explained in section 3.2. A suspicious policy set could cause unexpected trouble in a managed system if a policy run-time system were to execute[1] concurrently the policies included in it. From the viewpoint of [9,11], we may regard the target of analysis as application specific (O+, O+) conflicts, where "O+" is the abbreviation for "Positive Obligation Policy".

Our analysis method can also be used to optimize policy processing. This optimization is based on the detection of suspicious policy sets. Policy processing is optimized when policies may be executed concurrently so long as they do not comprise any combination of policies found in any one of the previously detected suspicious sets. This offers great advantages in efficiency over ordinary conservative policy processing, in which individual policies are all executed sequentially, and it is just as safe.

In order to confirm the feasibility of our approach, we have also developed an experimental system to measure its performance. As an experimental policy-based management system, we employ a slightly modified PONDER[4] framework, and as the system to be managed, we use the J2EE1.4 Application Server[14] provided by Sun Microsystems. As interfaces to monitor and control the system, we use Java Management Extension (JMX) interfaces[13]. Our experiments show that an optimized system is as much as 1.47 times faster than a non-optimized system.

The contributions of the work are as follows: (1) with our analysis method, it is possible to statically detect *suspicious policy sets*, i.e., those that might cause unexpected trouble in a managed system, if their policies were to be executed concurrently, thus ensuring improved reliability; and (2) it contributes to significant performance improvement in policy systems by making it possible to optimize policy processing. The effectiveness of this second contribution is shown in our experimental results.

2 Problem Statement

Figure 1 depicts an example of a problem that could possibly be caused by concurrently executing policies contained in a suspicious policy set. In order to quickly react to problems, it is highly possible that there are multiple managers responsible for creating and modifying policies. A management system is composed of a policy repository, Policy Enforcement Point (PEP), Policy Decision Point (PDP) and an event monitor[12]. A managed system consists of servers and network devices. We assume that managers register their carelessly created policies in the policy repository, and the policies are then deployed to the PEP and enabled. If (1) policies registered by different managers were to be executed simultaneously owing to the occurrence of specified events, (2) there were to be the same action target in the policies, and (3) actions to the target were to have side-effects, that is, the actions may change the value of attributes defined in the target,

[1] In this paper, "execute a policy" means "execute the actions in the action clause of a policy".

then the concurrent execution of the actions might possibly lead to problems. Although problems of this kind are considered under *Multiple Managers Conflict* in [11], this work does not present a way for detecting and resolving them.

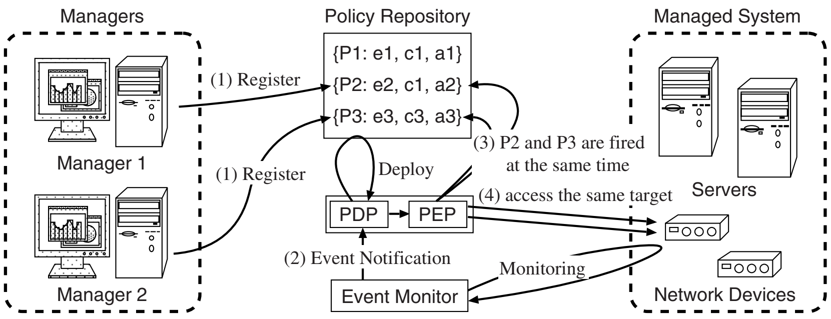

Fig. 1. Example of a problem caused by executing policies concurrently

The problems caused by the concurrent execution of policies included in a suspicious policy set are as follows: (1) if the policies were to be created without any considerations to race conditions of resources, threads executing such policies might possibly fall into deadlock; (2) if operations provided by a target were not to be implemented as thread-safe, the concurrent access to the target might possibly make states of the target inconsistent; (3) since a sequence of actions defined in a policy may be interleaved by another sequence of actions, the concurrent execution of them could cause transactional errors that might destroy the consistency of a managed system.

Our analysis introduced in the following section enables managers to ensure reliability by eliminating the potential for unexpected problems that might be caused by the concurrent execution of policies.

3 Analysis

In this section, we clarify what kind of policy specifications we assume for our analysis method and then explain the method in detail.

3.1 Target Policy Specifications

As targets of our analysis method, we assume Event-Condition-Action (ECA) policy specifications, such as PONDER[4]. An ECA policy is composed of an event clause, a condition clause and an action clause. The event clause shows events to be accepted. The condition clause is evaluated when a specified event occurs. If the condition holds, actions in the action clause will be executed.

The essential function of our method is to detect overlapped targets in policies. The accuracy of the detection depends on the characteristics of policy specifications. The most accurate information on a target is information on an instance that can be mapped to an actual device or a software entity composing a managed system. However, it is not pragmatic to expect the information on instances is available in policy definitions. Actual targets of actions are often decided at run-time, such as a target defined as the least loaded server in a managed system.

In contrast to the above case, if information on targets were not available in policy definitions, it would be impossible to apply our method to such definitions. An example of such an action clause is as follows:

```
ObjectName targetName= new ObjectName("name=logmanager,...") ;
Object[] params= makeParam("WARNING") ;
String[] signature= makeSignature(String.class.getName()) ;
mejb.invoke(targetName, "setLogLevel", params, signature) ;
```

In the above example, the action clause is written in Java programming language[1], which invokes `setLogLevel` provided by a managed entity in the J2EE[14] application server to change the log level to `WARNING` using JMX[13] interfaces. Information on the target is embedded in the parameter for the method. In general, it is impossible to analyze the exact value of parameters using program analysis techniques.

Therefore, we presume policy specifications in which targets are defined by *class* are applied to our method. The concept of class is the same as that defined in object-oriented languages, which define attributes and operations of a device or a software entity. In the network management domain, CIM[5] is the most promising model to define classes of managed entities. A rewrite of the above example using a class is as follows:

```
Target: Logmanager l ;
Action: l.setLogLevel("WARNING") ;
```

In the above example, the variable `l` belongs to class `Logmanger` and it is clear that the target of action `setLogLevel` is class `Logmanger`. While the equivalence of instances cannot be checked under our assumption, the equivalence of classes of targets can be determined.

3.2 Analysis Method

The analysis method detects all suspicious policy sets, which meet three conditions: (1) there is a shared target that appears in the action clause of policies contained in a suspicious policy set; (2) there are one or more actions that have side-effects on the shared target; (3) policies in a suspicious policy set might possibly be executed at the same time.

The method consists of two parts. One is the analysis for the action clause, corresponding to conditions 1 and 2, and the other is the analysis for the event clause and the condition clause, corresponding to condition 3. The former detects all sets of policies that are not assumed to be safe for the concurrent execution and the latter makes results of the former analysis more precise by dividing or removing the suspicious policy set containing policies that will not be executed at the same time.

The method is conservative, i.e. it detects all policy sets supposed to be unsafe, although the detected sets might possibly include the sets that are safe for the concurrent execution. Below, we will explain these two analyses.

Action Clause Analysis. In this analysis, all sets of policies meeting conditions 1 and 2 are detected. With the predicate logic, the conditions are formally defined as below:

C: set of all classes corresponding to managed entities in a managed system.
SP: set of policies (*Suspicious Policy set*).
targets(p): function that returns the set of all classes appearing in policy p.
actions$(p,\ st)$: function that returns the set of all actions appearing in policy p and defined in class st .
sideEffects(a): predicate that indicates the action a has side-effects.

$$\exists st \in C:\ \{\forall p \in SP : st \in targets(p)\} \wedge \{\exists p \in SP,\ \exists a \in actions(p, st) : sideEffects(a)\}$$

The variable st expresses the Shared Target of polices in a suspicious policy set. With this analysis, we detect all sets of the largest *SP* and the smallest *SP* for each class appearing in policies. The smallest *SP* is the set that contains only one policy whose actions have side-effects on the shared target. We regard the smallest set as a self conflict that a policy contained in the set should not be executed with itself concurrently, since it is possible that a policy may be executed twice at almost the same time if an event to be accepted by the policy were to be notified twice virtually simultaneously. Notice that the above logical expression is satisfied even in the case that there is only one action that has side-effects on the shared target in a suspicious policy set *SP*. In this case, while the race conditions of resources will not occur, the transactional errors mentioned in section 2 might possibly occur.

As mentioned before, whether targets are the same or not is determined by checking the name of classes. All sets of the smallest *SP* can be created by, for all policies, making a set containing only one policy whose actions have side-effects. How to obtain all sets of the largest *SP* is as below:

1) collect the class name of targets appearing in the action clause of all policies.
2) for each previously collected class name c, make a set of polices whose action clause contains the class name that is equal to c.
3) from the sets obtained above, remove all sets that contain only policies whose actions do not have side-effects on the shared target.

In order to decide whether an action has side-effects or not, all actions defined in classes must in advance be assigned one of 3 attributes: *Write*, *Read* and *Unknown*. *Write* is assigned to the actions that may change the target entity states, i.e. have side-effects. *Read* is assigned to the actions that do not have side-effects. Since the attributes are supposed to be assigned manually, *Unknown* is used for the actions that are not explicitly assigned an attribute. *Unknown* is treated as *Write* in this analysis.

These attributes can be included in class definitions or in other definitions separately from class definitions. For instance, using the JMX[13], which is the standard specification for monitoring and managing applications written in Java programming language, attributes of actions (or methods) can be obtained by invoking `MBeanOperationInfo.getImpact()`.

Event and Condition Clause Analysis. The problems mentioned in section 2 occur only when multiple threads execute actions of policies concurrently. There are two issues that determine whether policies will actually be executed concurrently. One is the difference between strategies that policy run-time systems employ to execute policies and the other is how to analyze the event and the condition clause. Below, we will explain both of these.

There are several strategies for executing policies. Whether policies are concurrently executed by a policy run-time system depends on strategies. We categorized the strategies into three types:

- Conservative Strategy: All the policies executions are serialized. Although the problems mentioned in section 2 will not occur, it involves deterioration of policy processing performance. In section 4, we will introduce an application using the action clause analysis to improve the performance of systems employing the strategy.
- Serialized Event Strategy: The execution of policies for incoming events is suspended until all executions of policies triggered by the previous event have been completed. Policies triggered by the same event will be executed concurrently. With this strategy, we can detect the sets of policies that will not be executed concurrently with the analysis for the event clause and the condition clause.
- Concurrent Strategy: Policies are executed concurrently. Therefore, managers have to take into account the concurrent processing issues when writing policies. The analysis for the event clause will not make sense, since all kinds of events may possibly occur all the time. The analysis for the condition clause, however, will work effectively. For instance, a policy in which only the temporal condition 10:00-17:00 holds will not be executed with one in which the temporal condition 18:00-21:00 holds'D

Thus the effectiveness of the analysis for the event clause and the condition clause depends on strategies.

Next, we consider the analysis for the event clause. The event clause shows events to be accepted. It contains a single event or an expression of composite events. Composite events are combined by logical operators or operators that specify an order of event occurrences[3].

In the event clause analysis, we focus on the events that may directly trigger an execution of policies. Since the event clause analysis is mainly used for systems employing the serialized event strategy, whether polices can possibly be executed simultaneously can be determined by checking whether the policies have the same *direct trigger event*. Here, we will explain the direct trigger events in detail. In the case of a single event and a composite event combined by the “OR” operator, direct trigger events are all events a policy accepts, since the occurrence of these events might directly involve an execution of the policy. In the case of the composite event “$e_1 \rightarrow e_2$”, which means that the event e_2 occurs after the event e_1, the direct trigger event is e_2. In the case of the “AND” operator, “e_1 AND e_2” is interpreted as “$e_1 \rightarrow e_2$ OR $e_2 \rightarrow e_1$”, so the direct trigger events are both e_1 and e_2. In the other case, if we could make an automaton from the event expression we would be able to obtain direct trigger events of policies. Such an automaton may have a start state, final states, nodes to express states of the acceptance of events and transitional labels corresponding to events. Events corresponding to labels to the final states of an automaton can be regarded as direct trigger events. Thus, the policies that contain the same direct trigger event might possibly be executed concurrently.

These policy sets can be detected by a method similar to the action clause analysis. By adapting the action clause analysis to result sets detected with this analysis for the event clause, we can make suspicious policy sets more precise.

Next, we will consider the condition clause. There are two widely adopted conditions. One is the temporal constraint used for specifying the duration a policy should be enabled, such as 10:00-17:00. The other is to check whether a specified condition for a managed system holds by retrieving states of managed entites when an event occurs.

By analyzing conditions for time constraints, policies that will not be executed at the same time can be detected (only if there are no undefined variables in the condition clause). Consider an example: $\{P_1, P_2, P_3\}$ is one of suspicious policy sets detected with the action clause analysis. Then by analyzing the conditions of P_1, P_2 and P_3, we presume to obtain the result that neither P_1 and P_2 nor P_2 and P_3 will be executed at the same time. We can make suspicious policy sets obtained with the action clause analysis more precise as follows:

$$\begin{gathered}\text{We know } P_1 \text{ will not be executed with } P_2 \\ \{P_1, P_2, P_3\} \rightarrow \{P_1, P_3\}, \{P_2, P_3\} \\ \text{We know } P_2 \text{ will not be executed with } P_3 \\ \{P_1, P_3\}, \{P_2, P_3\} \rightarrow \{P_1, P_3\}, \{P_2\}, \{P_3\}\end{gathered}$$

Since $\{P_2\}$, $\{P_3\}$ can be eliminated, we obtain the result set $\{P_1, P_3\}$. Thus we can refine results of the action clause analysis with the condition clause analysis.

In the case of conditions that check states of managed entities, it is almost impossible to determine whether the conditions will not hold at the same time. Consider a condition "`x.CPU_LOAD > 90`" and another condition "`x.CPU_LOAD < 30`". If the variable `x` is always bound to the same target, these conditions will not hold at the same time. In most cases, however, the variable `x` might possibly be bound to different target entities. Therefore, we do not deal with this kind of condition in this paper.

Thus, we have explained our analysis method that detects all suspicious policy sets. The analysis for the action clause detects all sets of policies that should not be executed concurrently, and the analysis for the event clause and the condition clause make the sets more precise using information on whether policies are actually executed concurrently.

4 Optimization for Policy Processing Using Analysis

Here, we introduce the optimization of policy processing based on the detection of suspicious policy sets and explain the implementation for the optimization.

4.1 Basic Idea

The conservative strategy mentioned in section 3.2 is highly advantageous over the concurrent strategy, in that managers are freed from complicated concurrent processing issues when writing policies. However, this strategy has a problem in terms of performance.

With our analysis, we aim to improve the performance of policy processing systems that employ the conservative strategy, retaining the advantage of the conservative strategy. We will explain this idea using Figure 2.

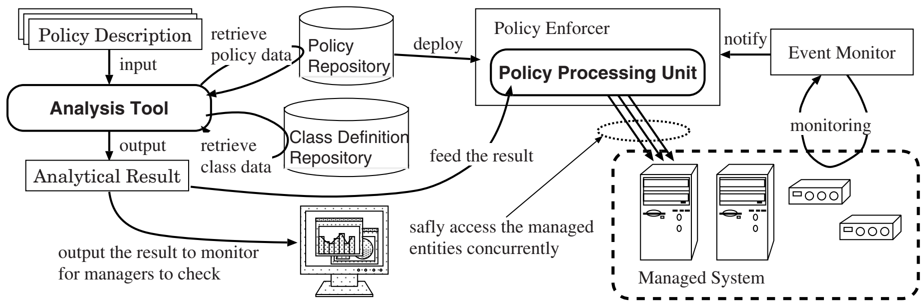

Fig. 2. Overview of the System for Policy Processing Optimization

At first, a manager applies the action clause analysis to new policy descriptions to be deployed into a policy enforcer and to deployed policies which can be retrieved from a policy repository. Then, the analytical result that indicates suspicious policy sets is reflected to a configuration of a policy processing unit in the policy enforcer. The policy processing unit controls the executions of actions and concurrently executes policies so long as they do not comprise any combination of policies found in any of the suspicious policy sets shown in the analytical result.

4.2 Implementation

We have implemented an experimental system using the Ponder[4] framework developed at Imperial College and the J2EE application server[14] provided by Sun Microsystems. The implementation of the experimental system can be divided into two parts, the policy analysis and the run-time execution control.

Policy Analysis. Figure 3 shows the policy analysis part of the implementation. The policies written in the Ponder policy specification language are compiled into the Java classfiles and stored in an LDAP server called *Domain Storage*. The analysis component in the figure applies the action clause analysis to policies stored in the LDAP server and outputs the result into a file, which will be fed to the policy processing unit. In order to check side-effects of actions, the analysis tool retrieves information on classes of the targets and on attributes of the actions from the J2EE application server via the JMX interfaces.

While targets are expressed in *Domain Notation*[9] in the Ponder framework , we treat the domain name for targets as the name to be mapped to managed entities in a J2EE application server. For instance, a target class "`/J2EE/logmanager`" is mapped to the corresponding managed entity "`Logmanager`" in a server. The managed entities in the J2EE applications are modeled in [15].

Run-time Execution Control. The run-time execution control is a policy run-time system based on the Ponder framework, which is intended for use in the management

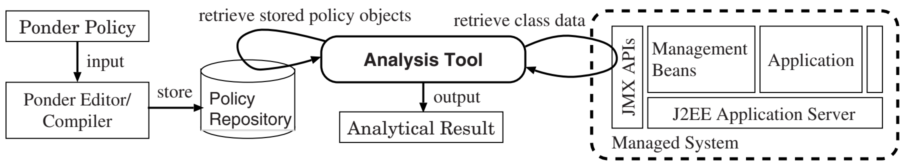

Fig. 3. Policy Analysis Part

of J2EE applications. While the framework employs the concurrent strategy, we have modified the implementation of PONDER so as to execute policies sequentially, using a waiting queue into which policies to be executed are put. This was a minor modification and we have modified less than a hundred lines of the original source code in interpreting the action clause and executing the actions defined in the clause. We have also added a few new classes for the optimization.

Figure 4 shows the internal mechanism of the run-time execution control. There are a policy enforcer that accepts events notified by a event monitor and a managed system. The policy enforcer contains a policy processing unit that controls executions of the action clause of policies with a waiting queue and a set named *active policy set*. We will explain the mechanism using the example depicted in the figure.

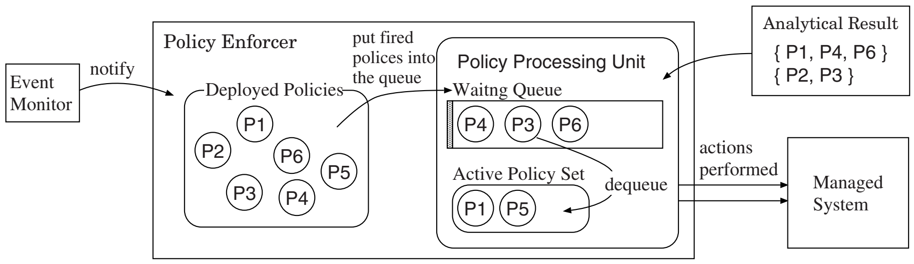

Fig. 4. Run-time Execution Control Part

The analytical result is fed to the policy processing unit beforehand, which shows that neither P1, P4 and P6 nor P2 and P3 should be executed concurrently[2]. When an event occurs and a policy is fired, the policy will be put into the waiting queue. The policy processing unit dequeues a policy in the FIFO manner and put it into the active policy set. The policy will remain in the set until the execution for it has been completed. If the analytical result shows that a policy to be dequeued conflicts with any of the policies in the active policy set, it will be skipped and the next policy will be dequeued. In the figure, P1 and P5 are in the active policy set and P4 in the waiting queue conflicts with

[2] The conflicts between the same policy are omitted for simplicity.

P1 and P6 as shown in the result. Thus, P4 will be skipped and P3 will be dequeued to be concurrently executed with P1 and P5 using the Java threads.

Thus, the optimized policy processing executes policies efficiently and safely using the analytical results. Using the implementation, the efficiency of optimized policy processing over the sequential processing is presented in the following section.

4.3 Experiments

We have conducted experiments for comparing performance of the sequential policy processing named the conservative strategy and the optimized policy processing. The results show the optimized one is as much as 1.47 times faster than the sequential one under the experimental environment.

We employ two PCs(CPU: Pentium4 3.0 GHz, Memory: 1.0GBytesOS: WidnowsXP Pro). The implementation based on the latest version of the Ponder Toolkit(11 March 2003) is located at one PC. On the other PC, the J2EE1.4 Application Server Platform Edition 8 is located. The PCs are connected by a 100base-T switch.

A total of 48 polices are deployed in the implementation. The definitions of the policies are the same except the name of policy. The policy definition is as follows:

```
inst oblig /Policy/${PolicyName}{
  on EventForExperiment() ;
  subject /PMAs/PMA;
  target t= /J2EE/logmanager;
  do t.setLogLevel("","SEVERE") -> t.setLogLevel("","WARNING");
}
```

The action clause of the policy means that the operation "`setLogLevel`" of the managed entity "logmanager" has to be invoked twice sequentially. The operation is used for changing the grain of data to be logged.

We prepare an artificial analytical result that is only written for controlling the behavior of the optimized processing for the experiment. The result consists of 8 sets of a suspicious policy set that contains 6 name of policies that should not be executed concurrently. The name of a policy appears in the result exactly once, that is, a policy is assumed to conflict with the other 5 policies and itself.

We put the 48 polices into the waiting queue randomly at first, then measured the time to complete 100 iterations of the process that (1) make a copy of the original waiting queue and (2) process all policies in the copy. The measurement was conducted 3 times to check the variance of results. The results are shown in Table 1. The time described in the table is the average of the 100 iterations of the process. The result shows the optimized processing is as much as 1.47 times faster than the sequential one.

5 Related Work

Our analysis method is developed for detecting all sets of polices that should not be executed concurrently. This type of problem between policies is classified as *Multiple Managers Conflict* in [11], although how to detect them is not presented.

Table 1. Experimental Result

	First	Second	Third	Average
Sequential processing	951ms	944ms	944ms	946ms
Optimized processing	643ms	645ms	643ms	643ms

The way of the detection and the resolution for *Modality Conflict* is proposed in [9]. It detects sets of polices of which subjects, targets and actions are overlapped. However, it cannot detect the polices that should not be executed concurrently, since it is not necessary for actions to be overlapped, although attributes of actions should be taken into account.

In order to cope with the application specific conflicts, approaches of using constraints on polices are proposed in [2,9]. In particular [2] focuses on conflicts of actions and presents formal semantics and notation to detect and resolve such conflicts. Although they may allow managers to write constraints for the concurrent processing issues as mentioned in section 2, how to implement an interpreter for these constraints is not presented. We have focused on the concurrent processing issues and presented analysis specific to them in detail, taking into account the strategies of the policy processing. In addition to improve the reliability of the system, we have shown the analysis can be used for improving policy processing performance.

The analysis assumes the action clause is written in the typed languages. As proposed in [10], it is possible to assign type to the targets in the action clause which is written in non-typed languages, by mapping the targets to the management model, such as CIM[5] or the model of J2EE[15].

The idea of assigning attributes to operations for checking side-effects has been commonly used in distributed systems. For instance, the distributed object system “Orca” uses the attributed method of the distributed objects for keeping the consistency of replicas of objects[7]. We have applied this idea to our analysis.

6 Summary and Future Work

In this paper, we have presented an analysis method for improving both reliability and performance in policy-based management systems. It detects all set of policies whose actions might possibly access the same target entity simultaneously. This information is vital to managers, who naturally wish to ensure reliability by eliminating the potential for unexpected problems that might be caused by the concurrent execution of combinations of policies contained in any one of such *suspicious policy sets*. The same information can also be used to optimize policy processing, making it possible to execute concurrently all policy combinations not included in any detected set.

Experimental testing of our analysis method shows that it can be used to execute policies more efficiently than can be done with the conservative, sequential-execution approach, and that it can do so just as safely. Results further indicate that an optimized system is as much as 1.47 times faster than a conservative system.

In our analysis, the equivalence of targets is checked at the class level, not at the instance level. It will be a main cause of the false detection of the analysis. In order to determine whether or not this approach is both accurate and effective, we intend to continue our work by applying our analysis to use-case scenarios.

In this paper, we assume that there is one policy engine to execute policies in a policy-based management system. In the case of multiple engines, we think our method is still useful for managers to create policies, since using the method they can know whether they should consider the concurrent processing issues or not. Improvement of our method taking into account the multiple engines is also future work.

Acknowledgements. This work is supported by the Ministry of Public Management, Home Affairs, Posts and Telecommunications.

References

1. Arnold, K. and Gosling, J.: *The Java Programming Language, Second Edition*, Addison-Wesley (1998).
2. Chomicki, J., Lobo, J. and Naqvi, S.: Conflict resolusion using logic programming, IEEE Trans. on Knowledge and Data Engineering, Vol.15, pp.245–250 (2003).
3. Damianou, N.: A Policy Framework for Management of Distributed Systems, PhD Thesis, Imperial College, London, Feb (2002).
4. Damianou, N., Dulay, N., Lupu, E. and Sloman, M.: The Ponder Policy Specification Language, In Proc. of Policy2001, Jan (2001).
5. DMTF: Common Information Model Spec.v2.2, June (1999).
6. Dunlop, N., Indulska, J. and Raymond, K.: Methods for Conflict Resolution in Policy-Based Management Systems, In Proc. of EDOC2003, Sep (2003).
7. Hassen, B.S., Athanasiu, I. and Bal, H.E.: A Flexible Operation Execution Model for Shared Distributed Objects, In Proc. of OOPSLA '96, pp.30–50,(1996).
8. Fu, Z., Wu, S. F., Huang, H., Loh, K and Gong, F.: IPSec/VPN Security Policy: Correctness, Conflict Detection and Resolution, In Proc. of Policy2001, Jan (2001).
9. Lupu, E. and Sloman, M.: Conflicts in Policy-Based Distributed System Management, IEEE Trans. on SE, Vol.25, No.6, Nov (1999).
10. Lymberopoulos, L., Lupu, E. and Sloman, M: Using CIM to Realize Policy Validation within the Ponder Framework, DMTF 2003 Global Management Conference, Jun (2003).
11. Moffett, J. and Sloman, M.: Policy Conflict Analysis in Distributed System Management, Journal of Organizational Computing, Vol.4, No.1 (1994).
12. Moore, B., Ellesson, E., Strassner, J. and Westerinen A.: Policy Core Information Model - Version 1 Specification, IETF, RFC 3060, Feb (2001).
13. Sun Microsystems Inc: Java Management Extensions Instrumentation and Agent Spec.v1.2, Oct (2002).
14. Sun Microsystems Inc: Java2 Platform, Enterprise Edition Specification, v1.4 Final Release, Nov (2003).
15. Sun Microsystems Inc: Java2 Platform, Enterprise Edition Management Specification, Final Release v1.0, June (2002).