RC23687 (W0508-032) August 4, 2005
Computer Science

IBM Research Report

Dynamic Introduction of Attributes into Policies

Alla Segal, Murthy Devarakonda, Ian Whalley, David Chess
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Dynamic Introduction of Attributes into Policies

Alla Segal, Murthy Devarakonda, Ian Whalley, David Chess
IBM Thomas J. Watson Research Center
{segal,mdev,whalley,chess}@us.ibm.com

Abstract

Current techniques in the field of policy management rely on system designers and implementers 'baking' information about policies and the attributes that those policies can have and what those attributes mean into a system at design and build time. In the future, in which policy-driven dynamic techniques for systems management and system composition become widespread (initiatives in this area include Autonomic Computing [6,7] and On Demand Computing [8]), this design- and build-time approach will no longer be suitable—a dynamic system must, in order to be effective, have a dynamic understanding of the policies, and the attributes of those policies, which drive it.

We describe a step on the road to a policy-driven, dynamic IT world—specifically, we describe techniques by which (for example) a hardware upgrade or a small change in a user's requirements can have an effect on a system without requiring a change in the policies, or the policy infrastructure. Our technique permits new attributes to be defined within existing policies, without changes to the supporting policy evaluation and management code. As a concrete example, we further describe how the ability to introduce new attributes can be implemented as part of a policy-based storage management system.

1. Introduction

In current approaches to policy-based system management, such as for example [9,10], the attributes about which policies can be written are statically defined—defined when the policy models are created by the architect, and when policy management code is written by the developer. The values of these attributes change at run time, but the attributes that are available to the policies are static. This makes it difficult to adapt such systems to possible changes in managed resources or users' requirements.

Consider the following situation: company A bought policy-based storage management software a year ago to use with their storage systems (which they bought from supplier E), and is now adding a new storage system (purchased from supplier I) to their SAN. The new storage system exposes, to the policy management software, a set of metrics that is somewhat different from that exposed by the existing storage systems—and a version of the policy management software to support those metrics is not yet available. For example, while the supplier E's storage systems allowed specification of the average transfer rate (through an attribute called `transfer-rate`), the supplier I's storage system separates this single metric into two separate metrics—the read transfer rate and the write transfer rate (through attributes called `read-transfer-rate` and `write-transfer-rate`). Company A's QoS policies have to be redefined using the new metric, but their policy-based storage management software's user interface only allows specification of policies in terms of the `transfer-rate` attribute, and the old policy evaluation can only evaluate the policies based on this attribute.

A similar situation can occur even without the introduction of a new and different policy-managed resource. In many cases, it may not be possible at development time to identify all the variables that a particular enterprise may want to use in policies—constantly changing requirements may mean than it necessary for an enterprise to base its policies on a variable that wasn't considered important at the time of original development. For example, a designer of a policy-based storage manager may decide that most users would prefer to measure quality-of-service in terms of throughput and response time. Such a system would, for example, allow the user to define a policy that triggers re-allocation of storage whenever the throughput or response time falls within a certain range of throughput and response time values. The storage system may allow the determination of values of various other system properties such as read-transfer-rate, write-transfer-rate, serial number, etc. by way of low-level system commands, but the user is not able to use any but pre-defined attributes “throughput” and “response time” in the creation of policies. Allowing the use of every existing system property in a policy may not be practical, especially in cases where it is necessary to base policy decision on the values determined by applying more complicated functions to these properties. (Here and elsewhere in the paper, we’ll use the word “attribute” to refer to policy attributes and “properties” to refer to all features of a managed system that can be measured, determined via low level system commands or determined from the documentation whether or not they can be used in policies.)

In another case, the designer may have decided that users would want to determine the level of service to assign to requests based upon which company they come from; while the user may want to use another characteristic, such as server ID or application name. This case is especially likely to arise when a policy-based system such as, for example, storage or a DBMS is used in a larger system and the requester details vary depending on how storage or database management is integrated with the overall system. For instance, a utility management system submitting requests to a DBMS has its own set of requestor attributes different from the attributes considered at the time the database management system was designed.

Dynamic introduction of new attributes can be seen as one, important way of achieving policy adaptation. Earlier work has considered policy adaptation by other means. Event calculus was used as a means of specifying adaptive policies to manage temporal events that encourage the system to adapt [11]. An adaptive policy management framework was proposed in the Ponder policy specification language for managing DiffServ networks [12]. The authors have defined policy adaptation as dynamically changing the parameters of a QoS policy, and enabling/disabling a policy from a set of pre-defined QoS policies at runtime. Policy transformation is another technique [13] which can be used to adapt a high-level policy to the specific low level policies and mechanisms (especially in a DiffServ network). None of these approaches discusses how a policy framework can become aware of new attributes to include them in policy definitions, and thus the work described in this paper defines a new dimension in policy adaptation.

In the remainder of this paper, we describe a model for the dynamic introduction of new attributes into policies. We present two approaches: the definition of new attributes via the user interface and the definition of new attributes using reusable components that represent individual attributes. We also show how both of these approaches can be combined for greater flexibility.

In our earlier work [1] we presented the policy management and rule execution architecture used in our prototype autonomic storage manager ALOMS-Tango [2]. In the described prototype, the attributes of policies such as response time and throughput were static. In this paper we expand this prototype to support dynamic attributes.

2. Dynamic attribute definition and discovery

2.1. Classification of Attributes

Before the dynamic introduction of attributes into policies is discussed, it is important to discuss the characteristics that define an attribute. In this section, we classify the attributes based on several important factors.

The first factor is the type of information represented by an attribute. An attribute can reflect a property of a managed system such as, for example, bandwidth or demand. These attributes are often used in quality-of-service policies. The evaluation of the policies depends on the ability to determine the attribute’s value. These attributes cannot be introduced into the system arbitrarily; they have to correspond to an existing system property (or a combination of such properties). We refer to these attributes as “system” attributes.

Alternatively, an attribute can represent a property of an input request. We call attributes of this type “request” attributes. An example of a request attribute is owner id from the service creation policy mentioned in [1]. This type of policy assigns a particular level of service based on the requestor’s characteristics. While request attributes can often be introduced to policies by a user simply typing in the attribute’s name, it may sometimes be desirable to define them prior to their use in policies. In addition to preventing errors, the prior definition of request attributes is necessary when the determination of the attribute’s value from the request is non-trivial or when additional information (e.g. external and internal attribute’s name, type of a policy the attribute will be used in) needs to be provided for a new policy attribute. Possible additional features of request attributes include: whether a particular attribute can be used as the only input characteristic of a request, whether an attribute is a required or optional characteristic of every request, whether one attribute shall only be used in combination with some of the other attributes, and other aspects of the relationships between multiple attributes.

Other examples of request attributes include a user name or a group name used in a security policy.

Another factor in attributes classification is the distinction between 'intrinsic' and 'derived'. An 'intrinsic' or 'base' attribute directly corresponds to a single system property. Intrinsic attributes can be defined by specifying the attribute's name and the name of the corresponding system (or requestor) property.

A 'derived' or 'computed' attribute is one whose value is obtained by combining a number of system or requestor properties. For example a user can define an attribute `demand-per-server` by dividing the value of system property `demand` by the `number-of-servers` assigned to the task, or define a new attribute `computer-id` by concatenating the two properties `serialNo` and `machineType`. The definition of a new derived attribute needs to specify 1) the names of system properties the attribute's value depends on 2) the instructions for determining the value based on these properties.

The last factor in attribute classification is "discoverability". This characteristic is used to distinguish between attributes that depend on the properties of the managed system or request that can be 'discovered' *at the time the policy is defined*, and those attributes the existence of which cannot be automatically determined until the time of policy execution. These are referred to as 'discoverable' and 'non-discoverable' attributes respectively.

An attribute is *discoverable* if 1) it depends on one or more properties the existence of which can be discovered programmatically (for example by issuing low-level system commands) at the time the policy is defined; and 2) the values of these properties can be determined programmatically at the time the policy is evaluated. These properties can be static (such as a system's model or serial number) or dynamic (such as throughput or response time). While theoretically the request attributes can be discoverable (by for example querying the transport for what kind of information is supported in request's syntax), most often they are not. While most system attributes are also discoverable, it is possible to imagine a situation when they are not. For example a particular system property may not be listed by low-level system commands, but its existence may be documented and its value can be queried at the time policy is executed.

When a discoverable attribute is defined by a user, a list of the available system properties can be provided to the user to facilitate the process of attribute's definition. The user can create an attribute based on these properties. When a non-discoverable attribute is defined by a user, the user will have to type in the name of the attribute and, when appropriate, the name of non-discoverable property the attribute will depend on. In some cases, a system may prohibit the definition of non-discoverable system attributes.

Both discoverable and non-discoverable attributes can be intrinsic or derived.

2.2. User-interface based attribute definition

The most straightforward approach to the definition of a new attribute involves the user interface (which, in CIM [3] terminology, is part of a policy-management application).

Using a user interface, new attributes are defined by the user (typically, the user will be a system administrator)—the system will, presumably, come with some attributes predefined, but will allow a user to create new ones.

In order to create a new system attribute, the user is required to specify both the name of the new attribute, and the mechanism by which the value of the new attribute is to be determined. The latter may involve selecting (or specifying – if the attribute is non-discoverable) a system property or properties upon which this attribute depends and defining a formula by which the new attribute's value may be determined.

In order to specify the new attribute with reference to pre-existing managed system's properties, the user will need to have access to a list of the properties of the policy-managed systems that can be used in policies. In order to support this requirement, the underlying resource management components must be able to provide lists of their properties. In cases when the 'external' name of the new attribute is different from the one used internally, the latter will have to be provided. When appropriate, the range of allowed values can be specified as well, as well as other parameters.

For example, in order to define a new derived attribute `average-transfer–rate`, the value of which is determined as a simple average of the two discoverable properties `read-transfer-rate` and `write-transfer-rate`, the user might:

1. Select the UI function “Define New Attribute”;
2. Type in both internal and “screen” names for the new attribute:
    a. Attribute name: `average-transfer–rate`
    b. Screen name: Average transfer rate
3. Select `read-transfer-rate` and `write-transfer-rate` from a list of available metrics;
4. Define, in an implementation-dependent manner, the formula to be used to derive the value of the new attribute. In this case, that formula is (`read-transfer-rate`+`write-transfer-rate`)/2;
5. Save the new attribute.

The new attribute definition can then be saved in the policy repository as a special ‘attribute definition policy’ or it can be stored separately in ‘attribute definition repository’. The formula by which the attribute’s value is determined can be encoded in any format, for example, by defining an XML format for expressions or as an ABLE [4] rule. In the case of non-discoverable request attributes the definition can involve simply specifying the name, specifying internal and external name as well as other characteristics such as the attribute’s relative priority and its relation to other attributes. The internal name of an attribute shall be used to determine the attribute’s value from the request that triggers policy evaluation.

Once an attribute is defined, it can be used in policy creation. When evaluating a policy containing new attribute(s), the policy processing code that determines the attribute’s value first checks if knows how to determine the attribute’s value directly. If it does not, the attribute definition is located to determine the names of required system properties. The value of each of these properties is obtained by calling appropriate sensors and the attribute’s value is determined by applying the specified formula (or by evaluating an ‘attribute definition policy’ as mentioned above).

2.3. Attribute beans

The user interface approach works best in simple cases where the value of an attribute is easily determined through simple predefined operations done on managed system’s properties. In cases where the attribute value is determined by applying a fairly complex formula, a different approach is appropriate.

This approach supports the dynamic discovery of the attributes and dynamic determination of their values during policy evaluation. To facilitate dynamic introduction of a new attribute into the system, we use Java beans and reflection technologies. We introduce the ‘attribute bean’—a JavaBean™ [5] that contains methods that allow construction of the new attribute from discoverable properties of the system.

The main functions of an attribute bean are to provide the names of managed resource properties that are required to determine the attribute value, and to determine the attribute value based on the values of these properties. A sample interface, that such beans are required to implement, is shown in Fig. 1. This `AttributeBeanInterface` contains various methods to allow for future tools that may help in the creation of attribute beans.

```
public interface AttributeBeanInterface {
    // get name of the attribute to facilitate discovery
    public String getAttributeName();
    public void setAttributeName(String name);
    // the name under which the attribute is known to the user
    public String getAttributeUIName();
    public void setAttributeUIName(String uiName);
    public void setMetricNames(String[] metricNames);
    // names of the metrics required to compute
    // the value of the bean
    public String[] getMetricNames();
    // in an alternative embodiment, the metric
    public Object getAttributeValue(HashMap metricValues);
}
```

Figure 1. A sample AttributeBeanInterface

An attribute bean may also be used to convert the user-visible name of an attribute to the one used in a request, or to determine its relationship to other attributes. It is possible to keep this information inside attribute beans as well, as shown in Fig. 2.

```
public interface AttributeBeanInterface {
    // get name of the attribute to facilitate discovery
    public String getAttributeName();
    public void setAttributeName(String name);
    // the name under which the attribute  is known to the user
    public String getAttributeUIName();
    public void setAttributeUIName(String uiName);
    // set names of attributes upon which this attribute depends
    public void setAlsoSpecify(String[] attributeNames);
    // get the names of the attribute(s) that
    // also have to be specified on the rule
    public String[] getAlsoSpecify();
    // set attribute’s priority
    public void setPriority(int priority);
    // get attribute’s priority
    public int getPriority();
    // indicate if this is a required attribute
    public void setRequired(boolean required);
    // is it a required attribute?
    public boolean getRequired();
    public Object getValue();
}
```

Figure 2. Another example of a sample AttributeBeanInterface

The attribute beans may be introduced into the system in a number of ways. For example, a naming convention that associates an attribute bean's name with the attribute name can be used. In this case, a user creating a policy can simply type in an attribute name and have the attribute bean located during the policy evaluation or indicate that an attribute bean will be provided. The location of the bean can optionally be specified as well. Alternatively, a configuration file with a list of supported attribute beans can be used by both the user interface and the policy execution environment or all beans can be located by searching for all classes that are located in a specific location (in, for instance, a filesystem or a particular URL) and follow the attribute bean's naming convention. In cases where the user interface is designed to discover the attribute beans, no additional information needs to be specified by the user; the user need only use the already defined attributes in the policies.

2.4. A hybrid approach

Sometimes it may be desirable to use both user-interface based attribute definition and bean-based attribute definition.

In this case, the user interface-based definition can be used for the attributes created by using simple computations on a few available system properties whereas the attribute beans can be used for complex computations or those combining a large number of managed resource properties.

In this case, the policy execution environment will have to first have to check if an attribute referenced in the policy was defined by the user, and then try to locate a corresponding attribute bean.

It is also possible to define some information for the new attribute via the user interface and use attribute beans to provide additional information.

3. A worked example — policy-based storage management

3.1. Autonomic Storage Manager Prototype

As an example of the possible implementation of this capability, we'll use the policy based autonomic storage management prototype, ALOMS-Tango, described in [2].

ALOMS-Tango allows administrators to define classes of service for storage in terms of performance and space metrics, set up alerts to be generated if the actual performance of the allocated storage comes within a given fraction of violating the requirements of the class of service, and visualize the configuration of the storage system for the allocated storage and identify performance bottlenecks. For the purposes of this paper, we are mostly interested in two main components of the prototype: the ALOMS-Tango Management Unit (ATMU) (which is responsible for resource provisioning and re-provisioning, collection of configuration information and performance metrics, policy management, and user interface support) and the sensors that the ATMU uses to collect the information about the storage system.

The user interface manager component of the ATMU allows the administrator to define service classes, to define policies and to visualize the configuration of the storage.

The policy management and rule execution architecture of the prototype was described in [1]. It has three subcomponents: a policy agent, a translator, and a rule engine. The policy agent retrieves relevant policies from a policy repository (in an XML schema) and uses the translator to convert them into a form that is suitable for the rule engine. During policy evaluation, the rule engine references certain variables and functions in evaluating the condition part of a rule, and (when necessary) in carrying out the action part.

The ATMU obtains the values of performance metrics via sensors.

3.2. Incorporating dynamic attributes into storage policy templates

In order to illustrate how the ability to add dynamic attributes will affect storage policy templates we'll consider the service class template and the alert policy template described in [1].

With only pre-defined attributes, the service class definition was defined in [2] as

```
Service Class "Gold"
Maximum size = 100Gbytes
Throughput = 20 Mbytes/Sec
Response Time = 5ms/4K block
Seq/rand access % = 100%
```

Out of the specified service class attributes only throughput and response time are discoverable metrics that are used to monitor performance as in the following alert policy:

```
   Generate an alert, if
[throughput] for [a container]
falls below [95%] of the value
specified in its service class
definition, in a 10-minute
period.
```

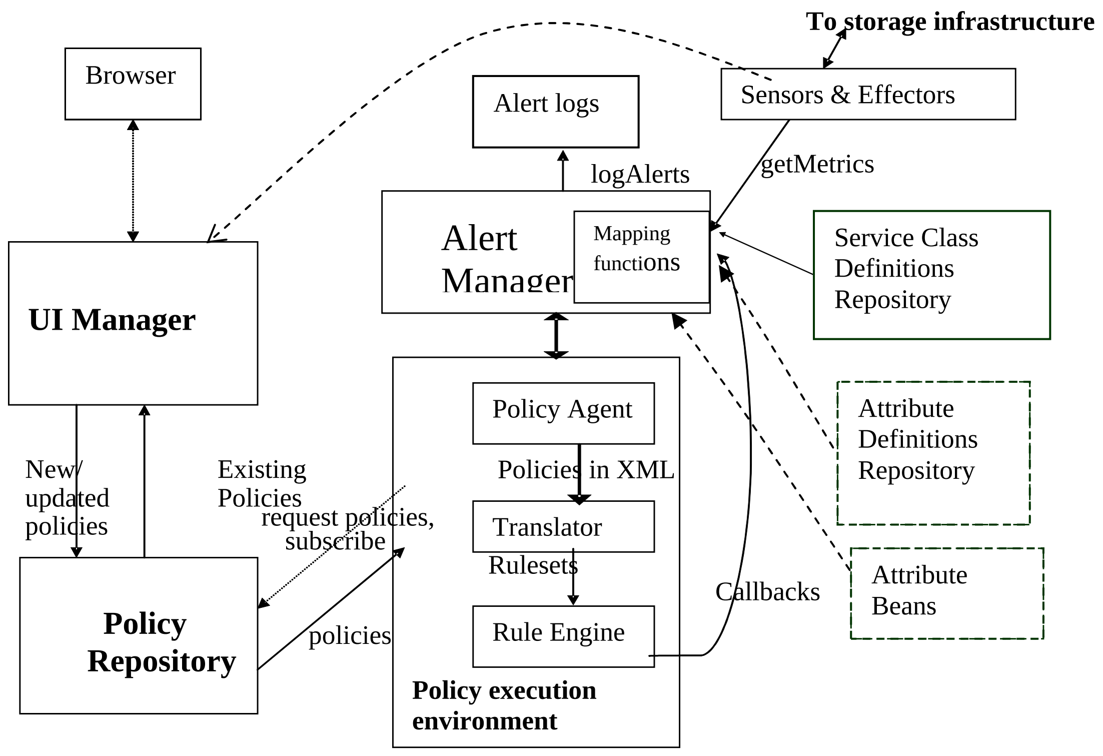

Figure 3 Policy management and rule execution architecture with support for the dynamic introduction of attributes

If for whatever reason, an enterprise wants to use a performance metric different from the throughput and the response time, it would need the ability to introduce new attributes into (and remove old attributes from) both the service class definition and the alert policy. Moreover, it would make sense to allow the use of a new attribute in an alert policy only after it has been incorporated into the service class definition.

The other policy templates mentioned in [1] are service-change and service-creation policy templates. Service-change policies specify the conditions that trigger re-allocation of storage. Since most of these conditions depend on the same performance metrics as alert policies, the process of introducing of new attributes into these policies is identical to that of alert policies.

Service-creation policy templates specify that data objects allocated for a specific user should be assigned a specific service class. Each user has properties such as a user id, a server id or a company id. These are non-discoverable attributes as they represent the characteristics of an input request. While these attributes are not affected by changes in the managed resource, the ability to introduce them into policies would allow an enterprise a greater ability to customize the system to their own needs and thus will result in greater usability of the system. In the next section, we will illustrate the use of an UI-based approach to introduce these attributes into policies.

3.3. Changes to policy management and rule execution architecture

In the policy management framework described in the previous section, the ability to introduce new attributes mostly affects the UI manager component, as well as the system components implementing mapping functions responsible for determining the attribute's value at policy evaluation time.

The UI-based definition of new discoverable attributes is based on the list of all available storage metrics provided by the sensors. The user creates new attributes by combining one or more metrics from the list using simple arithmetic operations. In case where more complex operations are desired, the user can specify that a corresponding attribute bean will be provided during policy evaluation. A more complicated solution would require the user interface to have access to the list of attributes defined as attribute beans.

Our sensors use low-level system commands and are capable of returning all available static system metrics and all dynamic metrics measured for a specific logical or physical volume. This makes it easy to both present the list of all metrics to the user and to determine the attribute's value by simply extracting the values of relevant metrics from sensor's output XML. In systems with sensors that can only measure a specific property, the layer that is capable of determining all available properties would need to be developed.

In case of request attributes, the user specifies the attribute UI name, the internal name and relative priority for conflict resolution. At present, only service creation policies have the potential to cause conflicts that can be resolved by assigning priority to attributes. An example of such a conflict is the existence of two policies that assign different service classes for different requester attributes such as 'assign service class "gold" to all requests from company "Chase" and 'assign service class "silver" to all requests from server "a001"'. A request that includes both of these parameters – `company="Chase" & server="a001"` – will cause conflict that can be resolved by assigning a higher priority to the attribute "company". Since neither alert policies nor re-provisioning policies have potential for the similar conflicts, the priority field is not needed for discoverable attributes in this particular system.

Once an attribute has been defined, it is saved in the attribute repository and is available for use in policies. In our case, the discoverable attributes become part of service class definitions and quality-of-service-related policies, whereas non-discoverable attributes are used in service creation policies. For example, in an alert policy mentioned above, the metric of storage performance can be any dynamically defined attribute.

In case of a richer set of policy templates, it may be necessary to group attributes by template. For example, if different sets of requestor attributes are applicable to service creation policies and to security policies, it will be necessary to specify which attributes are supported by which set of policies.

When the rules are evaluated, the responsibility for determining the newly defined discoverable attributes' values falls to callback functions. For example the following rule might result from the translation of an alert policy that indicates that an alert shall be issued if the value of the new attribute `aveTransferRate` falls below 95% of the value specified in the corresponding service class:

```
   if (observedValue("pmdo1",
aveTransferRate, "minutes", 10)
< 95% of expectedValue("pmdo1",
aveTransferRate)) then
(createAlert("logEntry", "pmdo1
average transfer rate is below
95% of specified value"))
```

The callback responsible for evaluating `observedValue` must locate the `aveTransferRate` attribute's definition and determine its current value. This callback is one of the mapping functions located inside the AlertManager. In order to do this, the Alert Manager first checks if the attribute corresponds to a property measurable by the sensors. If not, the Alert Manager attempts to locate the attribute's definition from the attributes repository and, if this fails, tries to instantiate the `AveTransferRateAttributeBean` class. From the attribute's definition the Alert Manager follows the process outlined above for determining this attribute's value using sensor output.

It would also have been possible to encode the attribute definition as a special group of policies. An example of an attribute definition policy would be an action-only policy that specifies that the attribute `aveTransferRate` is equal to the average of two system properties read-transfer-rate and write-transfer-rate. In this case, the determination of the attribute's value would have triggered the execution of a particular set of rules that would have resulted in this attribute's value.

Fig. 3 shows the policy management and rule execution architecture modified to allow the dynamic introduction of attributes through the user interface and attribute beans.

4. Conclusion

In this paper we have presented the case for the dynamic introduction of attributes into policies. We introduced the concept of 'attribute bean' and further described the system that allows dynamic definition of attributes via user interface and discovery of attribute beans at run-time, as well as the determination of new attributes' values at run-time without requiring any change to the existing policy management code.

In terms of the CIM policy management framework [3], the ability to define new attributes has to be supported by the policy management tool, while the ability to execute policies that use dynamic attributes and to determine discovered attributes' values has to be added to policy decision and policy enforcement points.

In order to facilitate the dynamic discovery of supported metrics system properties and their use in policy attribute definitions, the standards for both metric names and the sensors' output need to be developed. Similarly, the ability of a future system's policy decision code to evaluate policies that use new attributes depends on the development of a standard format for attribute definitions produced by policy management tool, and standard interfaces for the attribute beans created by a programmer.

It is our belief that the ability to introduce attributes into policies at run-time will be very important in the emerging areas of autonomic computing and e-business on demand.

5. References

[1] Murthy Devarakonda, Alla Segal and David Chess "A Toolkit-Based Approach to Policy Managed Storage", Proc. of 4th IEEE Intl Workshop on Policies for Distributed Systems and Networks, June 2003.

[2] Murthy Devarakonda, David Chess, Ian Whalley, Alla Segal, Pawan Goyal, Aamer Sachedina, Keri Romanufa, Ed Lassettre, William Tezlaff, and Bill Arnold "Policy-based autonomic storage allocation", Proc. of 14th IFIP/IEEE Intl. Workshop on Distributed Systems: Operations and Management, DSOM 2003, Marcus Brunner and Alexander Keller (Eds.), Published by Springer-Verlag as Lecture Notes on Computer Science 2867.

[3] DMTF, "CIM Core Policy Model white paper," DSP0108, February 2001, http://www.dmtf.org/education/whitepapers.php.

[4] Joseph P. Bigus, Jennifer Bigus, "Constructing Intelligent Agents Using Java, "Second Edition, John Wiley & Sons, Inc., 2001.

[5] Cay S. Horstmann, Gary Cornell "Core Java. Volume II-Advanced Features", Sun Microsystems Press Java Series, 2000.

[6] Paul Horn, "Autonomic Computing: IBM's Perspective on The State of Information Technology", IBM Corporation, http://www.research.ibm.com/autonomic/manifesto

[7] Jeffrey O. Kephart and David M. Chess, "The vision of Autonomic Computing," Computer Magazine, IEEE, Jan 2003.

[8] IBM Corp, "Living in an On Demand World", October 2002, on the Web at http://t1d.www3.cacheibm.com/ebusiness/doc/content/pdf/whitepaper.pdf

[9] M. Kaczmarski, T. Jiang, D.A. Pease, "Beyond backup toward storage management", IBM Systems Journal, Vol 42, No 2, 2003

[10] Dinesh Verma, "Simplifying Network Administration using Policy based Management", IEEE Network Magazine, March 2002

[11] Chrisos Efstratiou, Adrian Friday, Nigel Davies, Keith Cheverst, "Utilising the Event Calculus for Policy Driven Adaptation on Mobile Systems," Proc. of 3rd IEEE Intl Workshop on Policies for Distributed Systems and Networks, June 2002.

[12] Leonidas Lymberopoulos, Emil Lupu, Morris Sloman, "An Adaptive Policy Based Management Framework for Differentiated Services Networks," Proc. of 3rd IEEE Intl Workshop on Policies for Distributed Systems and Networks, June 2002.

[13] Dinesh C. Verma, "Policy-Based Networking: Architecture and Algorithms," New Riders Publishing, 2001.

[tm] JavaBean is a trademark of Sun Corporation