Abstract

Self-managed systems need plans or goals to guide them during normal operation and allow them to react correctly to unexpected events. Policy-based technologies hold promise in providing these guidelines. Currently, most of the available tools for policy evaluation are domain-specific. While these tools are useful in designing a single specific system, they cannot provide a generic policy infrastructure that could be used across a variety of domains or standardized for intersystem communication or negotiations. In this paper, we present generic policy tools that could be used across a variety of domains. These policy tools and associated infrastructure are provided by the Autonomic Manager Toolkit (AMTK) developed in IBM Research. We present the AMTK policy evaluation system and describe how AMTK policy tools can be used to build the elements of self-managing systems.

1 Introduction

Self-managing or autonomic systems control their own operations. They choose actions appropriate in both common and uncommon situations, and they know what steps need to be taken under which circumstances. Policy-based technologies will allow an administrator to manage a complex system by setting high-level goals. They will provide tools for translation of these high-level goals into machine-readable instructions. These technologies will be vital to self-managing systems [2].

What is meant by 'a policy' in the context of self-managing systems? According to the PCIM [4] definition a policy is "a set of rules to administer, manage, and control access to network resources." In the context of autonomic systems, a policy is defined as a guideline provided to a computing element in order to affect its operations. For example, a sample storage allocation policy may instruct a computing element to "re-allocate storage if the response time exceeds 5ms". Translated into a machine-readable format, these policies can be used by a system to decide what to do in the described situations.

By themselves, policies are not dependent on the underlying structure of the application. They could be standardized and used by a number of systems, thus allowing high degree of flexibility, maintainability and manageability. Nevertheless, most of the current tools for policy infrastructure are domain and environment specific and their execution requires domain and environment specific knowledge. There is a need for an execution system that is flexible and usable across a variety of domains.

In this paper, we will present a generic infrastructure and components for policy execution. This policy execution system is part of the Autonomic Manager Toolkit (AMTK) developed within IBM Research. The objective of the AMTK is to supply components and infrastructure, which simplify the incorporation of autonomic functions into applications. It provides a set of building blocks for self-management, which includes monitoring, analysis, planning and execution components. AMTK design is consistent with a vision of autonomic computing presented in [2]. The AMTK policy execution framework is domain-independent and can be used for evaluating a wide range of policies. In this paper, we present the building
blocks for executing of policies supplied by the AMTK and outline how they can be used in design of self-managing systems. Other components and features of the toolkit will be described in separate reports. We begin with an overview of the proposed autonomic architecture and currently existing work on policy infrastructure, and then map the policy architecture to the proposed autonomic system architecture. Subsequently, we describe in detail the AMTK policy execution environment and give an example of its usage within a prototype self-managing system.

2 Policy Architecture for Autonomic Systems

A proposed architecture for an Autonomic System described in [2] views an Autonomic System as composed from the interacting autonomic elements. The autonomic elements are created from existing system elements by the addition of the autonomic management functions. Each autonomic element has two management tasks: they manage themselves and they manage their relationships with other elements through negotiated agreements. Policies provide the guidelines for autonomic element for performing these tasks.

![Diagram of Policy Architecture](image)

Figure 1

The implementation of policy infrastructure for the autonomic system is similar to that of any policy-based system and requires the implementation of several components. A typical policy-based system, based on the IDTF/DMTF policy architectural framework [12] consists of four components: the policy management tool, the policy repository, the policy decision point and the policy enforcement point. An administrator uses the policy management tool to define the policies that are to be enforced within the network. A device that can apply and execute the different policies is known as the policy enforcement point (PEP). The policy repository is used to store the policies generated by the management tool. In order to ensure interoperability across products from different vendors, information stored in the repository must correspond to an information model specified by the Policy Framework Working Group. A policy enforcement point uses an intermediary known as the policy decision point (PDP) to communicate with the repository. The PDP is responsible for interpreting the policies stored in the repository and communicating them to the PEP. The PEP or PDP may be in a single device or different physical device. When used in the context of Grid Services, Grid Protocols are used by the different components to communicate with each other.

Figure 1 shows the components of policy IETF policy mapped into the autonomic elements. The functions of the Policy Decision Point and Policy Enforcement Point are both situated inside a specific autonomic element. AMTK policy execution environment provide the framework and the components for building both the Policy Decision Point (translation and validation) and the Policy Enforcement Point (policy execution). The functions of Policy Management Application are performed by a user interface that can be a specialized interface for the specific autonomic element or a system-wide UI. In this paper, we concentrate on the functions of the Policy Decision Point and the Policy Enforcement Point. The functions of Policy Management Applications and user interface issues are beyond the scope of this paper.
3 Policy Architecture for Autonomic Elements

Figure 2 shows the structure of an autonomic element and the role of policies in its operation. An autonomic element consists of the autonomic manager and the managed element [2]. A manager uses the monitor, analyze, plan and execute (MAPE) structure of the element using an inbuilt knowledge base (K). Sensors and effectors are used to get the information about the element and manage the element respectively. From the policy perspective, sensors are the values exposed by the element, and effectors can be used to control the behavior or provide inputs to the element. The autonomic manager and the element together form a composite autonomic element with its own sensors and effectors.

Policies are used in different phases of the autonomic management cycle as illustrated in the figure.

Policies can be used in various ways to simplify and automate the operation and management of autonomic computer systems. Policies apply to various parts of the MAPE structure as follows:

- **Monitor**: The monitoring component of MAPE is responsible for the monitoring of system states. Policies could be defined controlling the specific type of monitoring that needs to be performed under various conditions.
- **Analyze**: The analysis component is responsible for analyzing the monitored information in order to generate reports and to identify potential problems with the state of the system. Policies dictate the type of reports that are to be generated, and whether the current state of the system is in violation (or potential violation) of any specified policies in the system.
- **Plan**: The planning component determines the actions to be taken after the analysis phase. Policies dictate the type of actions that will need to be taken, including actions that can modify the configuration of the system. A policy may call for allocation of new resources, initiation of new tasks, or termination of existing tasks.
- **Execute**: In the execution component, a policy may define specific details for steps to take in order to carry out any actions or task identified by the planning component.

As we can see, the policies play part in every aspect of an autonomic manager. Going back to the IETF/DMTF framework shown on Figure 2, the autonomic manager is responsible for the implementation of Policy Decision Point and Policy Enforcement Point for the relevant policies. The planning phase of the autonomic element is for the most part responsible for the Policy Decision Point, whereas all phases of the autonomic element’s MAPE loop may have to implement the Policy Enforcement Points for the execution of relevant policies.

When an autonomic element starts, it gets its internal policies from policy repository. The external policies are delivered to the element via its external sensors. At the same time, an autonomic element subscribes to
changes to the policies. The “planning” phase of an autonomic element’s is responsible for translating new and updated policies. The planning component translates policies when it receives them from the repository during element’s initialization, and when the external or updated internal policies are delivered via the external sensors. The policies can be executed in response to events such as performance changes detected by the sensors or executed on demand. In the latter mode of executions, the autonomic manager can execute the policies periodically based on a specific time interval that can be defined in meta-policies. Each phase of an autonomic element’s operation is responsible for the execution of the relevant policies as needed. For example, the monitoring phase may execute the monitoring policies periodically or in response to some events – e.g. a sharp spike in the workload noticed by the sensors. The analysis phase of an element is responsible for executing the policies to determine the type of a report.

1.1 Policy Model
The AMTK uses a draft OGSA policy standard as its default policy model. This draft proposed a close mapping of Policy Core Information Model (PCIM) [3] to a policy XML schema. An Autonomic policy effort is defining a set of common policy definitions to be used in the context of Autonomic Computing within IBM. It is a refinement of the PCIM information model, where each policy is viewed as being composed from four types of constructs: pre-condition, measurable intent, scope, and business value. The autonomic policy workgroup will be defining some common types of pre-conditions, measurable intent, scope, and business value expressions. The AMTK and Policy Toolkit will track this emerging standard and will support it as it emerges.

1.2 Policy Execution

1.2.1 Rule Engine Based Policy Execution
Policies are generally not represented as executable code, and must be converted to an executable representation such as a rule language. This conversion involves understanding the semantics of the policy, the mapping of logic in the policy to the constructs of the underlying code, and the mapping of terms used in policy to the underlying data such as database entries, application objects or resource models which describe the devices to which the policy is applied. A policy executed in a different environment may require different mappings; while the logic expressed in the policy may be the same, the data, class objects or device resource model may be different. Generic rule engines such as production rule systems (including OPS5, CLIPS, flex [8], Java Script engines, backward and/or forward chaining inference rule engines [9], fuzzy rule engines, and SQL engines) can be used as the underlying execution system for policy, depending on the data requirements of the policy and the level of reasoning required. For examples, if a policy execution episode requires a large amount of data that does not change often, a Rete rule engine [5] provides maximum efficiency since the Rete rule engine examines only the changed data. The AMTK provides a set of interfaces for all key policy engine components, such as policy, policy evaluation, rule engine and translator, and the resource mapping [Table 1], so developers can implement their own components as needed if the default implementations do not satisfy their requirements.

A tabular-based system offers a fast, efficient, simple and easy to implement policy system and is appropriate for many policy-based applications. For some applications, such as those which require sophisticated executing strategies (forward and/or backward chaining or finding the best combinations of results…etc) and those which require reasoning (to determine the next policy rule to execute), it may be more applicable to use a rule engine based system. The AMTK provides interfaces for, and default implementations of, a rule based policy execution mechanism, that support the direct invocation of class methods as sensors and effectors using a pluggable rule engine. For certain applications of policy, it may be difficult to model the sensor data as a set of entries in a table, especially applications with data that are run-time dependent on each other and requires run-time data bindings. Direct invocation of methods by policy rules during rule execution simplifies policy processing and execution considerably, and the various type of rule engines to execute the resulting rules are readily available, such as those offered by ABLE [11]. The Resource Mapping provides the links between the expressions in the policy and the underlying class methods. This design allows interchangeable policy and resource mapping definitions, and pluggable translators and rule engines, in order to handle different context and environments with the same policy.
Figure 4 outlines a typical policy execution episode. Policy execution begins by input of a policy document (which may have been retrieved from a Policy Repository) – typically during the planning phase of an autonomic element’s MAPE loop. The policy is then translated to an executable format by using an appropriate translator from the Translator Repository, using a set of resource mapping definitions expressed in XML. The output of the translator is an executable ruleset for the selected rule engine or execution mechanism. The ruleset can then be published to the execution engine, or the ruleset can be simply stored in the Executable Ruleset Repository for subsequent use. In addition, executable rulesets can be pre-processed or cached, and fetched at runtime to achieve better performance and variability. The examples in section 6 illustrate this concept.

The AMTK policy execution system provides both event-driven and on-demand operation. Sensor information can be passed to the policy evaluation system as an event, and this operation mode represents a majority of applications, which require a sensor to post an event for any changes of state. However, there are applications which have on-demand sensors, or whose sensors will best be implemented as on-demand sensors. AMTK allows both event-driven operation and direct invocation of the policy evaluator via application control to meet the needs of both classes of applications.

1.2.2 HyperCube Evaluator System

In many applications of policies, policy evaluation consists simply of finding a set of policies that match some conditions and taking the actions upon them. In these cases, developers may want to use a more efficient evaluation mechanism. The hypercube evaluation system provides such a library, and may be appropriate for developers who do not need inference capabilities provided by rule-engines.

Consider, a simple policy statement:

\[ P1: C1 \land C2 \rightarrow A1 \]  
where  
\[ C1 \text{ is } (1.0\text{sec} < \text{ResponseTime} < 3.0\text{sec}), \text{ C2 is } (70\% < \text{CPU Utilization} < 90\%) \]

Figure 5 represents this graphically. The box represents the conditional expression, which is \( C1 \land C2 \). Point \( x = (1.3\text{sec}, 82\%) \) satisfies the conditional expression since it satisfies both conditions. Point \( y = (2\text{sec}, 50\%) \) fails to satisfy the policy rule since it fails to satisfy both conditions. Therefore, action A1 is performed for point x but not performed for point y.
Each policy rule defines an n-dimensional box in n-dimensional space, where n is the total number of variables used in the policy definition. Given an n-dimensional point that represents the current measurements for each dimension, the problem becomes identifying all boxes that contain the point (or the box with the highest priority that contains the point). Boxes can be overlapping, nested, share an edge, etc. Figure 6 shows an example in 2-dimensions where point y is contained in three boxes (i.e. point y satisfies three policy rules) and point x is contained in four boxes (i.e. point x satisfies four policy rules). The set of all boxes containing the point may be found, or the box with the highest priority.

Each policy rule is represented by a box – not a circle, or any other shape – that will always have boundaries orthogonal to the axis. Note that many policy rules will not have all dimensions explicitly defined; many policy rules will have don’t cares for several of the dimensions. In those cases, the box for that policy rule will have an edge in that dimension that covers the entire range of values. For example, if a policy doesn’t care about the response time, then the range for that policy in the response time dimension is [0, +∞]. Likewise, if the condition on a dimension is simply: ≤, ≥, > or < the edge of the box extends to the natural end (or beginning) of the dimension.

The = operator defines a point in the dimension, represented by a range with equal start and end values, such as [40, 40]. Strings and enumerations are assumed to have an ordering.

The hypercube policy evaluator performs an efficient matching of policies by determining the region of the space that corresponds to the conditions triggering policy evaluations. The different regions in the hyper-space are mapped to a search tree, and efficient tree-traversal techniques are used to find out the set of hypercube spaces, and the policies that match them. The hypercube space representation of policies also provides the background for checking the consistency, coverage, and conflict resolution aspects of policies.

2 Policy Use in Autonomic Manager

The architecture of policy execution in AMTK allows an autonomic manager to dynamically configure various policy execution components. Executable rulesets, execution engine, and mapping information can be retrieved and loaded during runtime to satisfy the execution context and environment. For example, an autonomic manager can manage storage allocation for different systems using a single policy; and each managed system contains different sensors, and/or environment information. The same autonomic manager and the same policy can be used for all managed systems, without creating a separate policy for each managed system, by loading a resource mapping specification appropriate for the managed system at runtime. In addition, an autonomic manager may have multiple policy execution engines, with different properties and missions. For example, an autonomic manager manages storage allocation for an application and negotiates with a storage provider for additional storage resources if application’s demand for storage exceeds capacity. A possible design for such an autonomic manager is to have a policy for managing the storage allocation of an application and to have a separate policy for the negotiations with the storage provider. The nature of the policy used in managing storage allocation is different from the one used in
negotiation with the storage provider. Typically, an AMTK HyperCube evaluation system is best used to manage the storage allocation. The parameters for managing such a system are relatively simple and can be expressed using simple primitive data types and expressions, but this type of policy require fast access and execution. Another instance of the policy execution, possibly with an inference engine, may be more suitable for negotiations that require reasoning on environmental data, current and previous performance data and so on. The seamless interaction of various instances of the policy engines can be achieved via the use of a central knowledge base, which is also a basic component of the AMTK, but is not the subject of this paper.

Figure 7 shows a basic conceptual view of how policy based functions are incorporated into Autonomic Elements (AE). Authoring and policy services (figure 3,4) at design time to construct policies and a run time infrastructure which allows pluggable components (figure 4) to be dynamically loaded into the AEs are provided by the AMTK. An AE can switch to another ruleset and/or mapping and/or execution mechanism depending on the environment or/and the logic of the policy. This architecture provides AE with a much higher level of adaptive behaviors, and allows the AEs to be well situated in a heterogeneous environment, and maximize the variability and flexibility provided by the use of policies.

While the picture shows three autonomic elements each using a different policy evaluator, all evaluators can and are likely to be used by the same autonomic manager.

3 Example: Policy based Autonomic Storage Management

The AMTK policy engine was used in a policy-based autonomic storage manager prototype whose manager architecture was described in [10]. The prototype's policies use a set of service classes that specify expected values for a particular set of attributes, such as response time, throughput, a ratio of random and sequential access, and maximum size of an allocated data object. A "Gold" service class, for example, can be defined as a response time of no more than 5 ms, a throughput of at least 100Mb per second, all sequential access, and maximum size of 200 GB.

The prototype uses the on-demand model of policy execution. The policies are evaluated in either a periodic or a request-response mode. This mode of policy execution was chosen because of its simplicity, and because it allowed greater control over policy execution within the storage prototype. The model of generating an event whenever there is a change in performance attributes values was not appropriate because of sensor limitations. The difficulties involved in tracking the values of multiple constantly changing attributes far outweighed any advantage that could have been obtained. It would have been possible to use the event driven model in case of periodic policies by a timer event and, in case of request-response policies, by generating an event every time the request is received, but this approach seemed unnecessarily more complicated than the one chosen.
3.1 Alert Policies

The alert policies allow a storage administrator to specify that an alert should be generated and an appropriate party contacted when specific performance conditions are not met. An example of an alert policy would be:

Generate an alert if throughput for a PMDO PMDO1 falls below 95% of expected value during 5 minutes

where 'PMDO' represents a specific allocated data object ("Policy Managed Data Object") such as, for example a DB2 tablespace, managed by (allocated within) one of the provisioning classes, 'throughput' is one of the service class attributes, and expected value is obtained from the definition of the service class managing PMDO1 at the time the alert was issued. The difficulty in representing this type of policy in a generic manner lies in the necessity of specifying a number of parameters - throughput, PMDO1, time interval and the relationship between them, in a rule and the need to determine the expected value of an attribute at runtime. If a table-driven model were to be used, it would be possible to specify, values for the parameters - observedAttribute=throughput, PMDO=PMDO1, interval=5minutes; but it is difficult to express this association in a table. It may also be difficult to specify the fact that the expected value (e.g. 95% of expectedValue("PMDO1", "throughput") ) needs to be determined at runtime from the definition of the appropriate service class. In representing this type of policy using the AMTK policy execution environment, we have chosen to use a callback method. The policy is written as an OSGA based rule in XML [4], and an associated resource mapping file, using the default AMTK translator, is translated into the following simple executable rule which will be executed by the AMTK default rule engine:

if ( observedValue("PMDO1", "throughput", "minutes", 5) < 95% of expectedValue("PMDO1", "throughput") ) then createAlert("PMDO1", "observedValue")

where the policy generated by the user interface contains the generic information about the comparison and the values of the variables: PMDO1, throughput, etc., while the details of objects, actual methods to invoke, and parameters are stored inside a mapping file used by the policy engine. This approach allows a clear and clean separation between the policy specification and the implementation details. If the implementation is changed, neither the user interface nor the policy specification needs to change.

3.1.1 Service Creation Policies

An example of service creation polices specify that data objects allocated for a specific user or application should be assigned a specific service class.

if userId is "1002" and serverId="2356" then use service class "gold"
if customerId is "ABC co" then use service class "silver"

These policies are evaluated in response to a storage allocation request from a user with specific attributes, in order to determine the value of the service class. Only a subset of possible attributes can be specified on a particular request, but all policies need to be executed to determine which one is applicable. These policies are fairly simple to represent in any policy execution model. The only difficulty lies in the large number of possible user attributes and the desirability of possible changes in the allowed attributes list. To allow greater flexibility, these policies are also represented with callbacks. For example, the aforementioned policy will result in the following executable rule:

if valueOfAttribute("userId") == "1002" && valueOfAttribute("2356") then setServiceClass("gold")

The main advantage of this approach is the improved flexibility: there is no need for the generic policy engine to know which requestor attributes are possible, and the attributes not used in a specific request do not need to be initialized. The applications policy management framework can initialize only the attributes specified on a particular request in a simple hash containing specified attributes as name-value pairs, and
valueOfAttribute(attributeName) methods simply return an invalid value if the specified attributeName is not found.

4 Conclusion
We have presented an approach to the policy infrastructure and components for policy execution for self-managing systems. We presented this approach in the context of the autonomic system architecture [2] and IETF policy infrastructure [4]. We believe this approach and the various policy execution tools in AMTK satisfy the needs of self-managing systems for policy tooling.

Design changes and further additions to the AMTK policy execution engine are currently under development. We expect that there will be additional policy evaluation engines or mechanisms included in the AMTK as new requirements and autonomic functions are identified. One of the goals of the AMTK is to provide the foundation and infrastructure, so that additional components can easily be included. We welcome researchers and practitioners in various fields to try the AMTK [1], to develop sets of requirements, and to provide feedback on their experiences, to guide further improvements to, and extensions of, the AMTK.

5 Acknowledgements
Thanks to John Rofrano, Lee Rafalow, Robert Moore, David Kaminsky and Jeff Kephart for their contributions

6 References

Java is a trademark of Sun Corp, IBM is a trademark of IBM Corp, AT&T is a trademark of AT&T Corp.