Policy-Based Management and Context Modelling Contributions for Supporting Services in Autonomic Systems

J. Martín Serrano[1]; Joan Serrat[1], John Strassner[2], Ray Carroll[3]

[1]Universitat Politècnica de Catalunya. Barcelona, Spain.
{jmserrano; serrat}@tsc.upc.edu
[2]Motorola Labs, Schaumburg, IL, USA.
{john.strassner@motorola.com}
[3]Telecommunications, Systems and Software Group,
Waterford Institute of Technology. Waterford, Ireland.
{rcarroll@tssg.org}

Abstract. Autonomic networking systems dynamically adapt the services and resources that they provide to meet the changing needs of users and/or in response to changing environmental conditions. This paper presents a novel policy-based, context-aware, service management framework for ensuring the efficient delivery and management of next generation services using autonomic computing principles. The novelty of this approach is its use of contextual information to drive policy-based changes that adapt network services and resources. Policies control the deployment and management of services and resources, as well as the software used to create, manage, and destroy these services and resources. Policies also control the distribution and deployment of the necessary components for fully managing the service lifecycle, and provide the efficiency and scalability necessary for supporting autonomic systems.

Keywords: Autonomic Systems, Autonomic Networking, Policy-Based Service Management, Context Information, Self-Management Technologies, Next Generation Services, Context-Aware Framework, Context Information Model.

1 Introduction

Over the last decade, simple advances in resources and services have been feasible as result of the ever-increasing power and growth of technologies. However, this drive for more functionality has dramatically increased the complexity of systems – so much so that it is now impossible for a human to visualise, much less manage, all of the different operational scenarios that are possible in today's complex systems. The stovepipe systems that are currently common in OSS and BSS design exemplify this – their desire to incorporate best of breed functionality prohibits the sharing and reuse of common data, and point out the inability of current management systems to address the increase in operational, system, and business complexity [1]. Operational and system complexity are spurred on by the exploitation of increases in technology to build more functionality. The price that has been paid is the increased complexity of system installation, maintenance, (re)configuration and tuning, complicating the administration and usage of the system. Business complexity is also increasing, with end-users wanting more functionality and more simplicity. This requires an increase in intelligence in the system, which defines the need for autonomic networking [2]. If autonomic computing, as described in [1][3] is to be realized, then the ***needs of the business must be able to drive the services and resources that the network provides***.

Autonomic systems were built to manage the increasing complexity of systems [3][4] as well as to dynamically respond to changes in the managed environment [1]. While many have proposed some variant of automatic code generation, this requires a detailed model of the system that is being reconfigured as well as the surrounding environment. More importantly, a system component can not be reconfigured if the system does not understand the functionality of the component. This means that the system requires ***self-knowledge*** of its component, and knowledge of its users and requirements. One of the most important forms of knowledge is ***context***. For a system to be able to generate code to modify itself, a model of *context* and its relation to not just the set of self-configuration, self-healing, and other operations, but in general to the management environment itself, is required. In addition, policy management is in theory able to control these functions and ensure that they are applied consistently.

There are important business and technical drivers encouraging the use of autonomic principles [3]. Our work in autonomic systems centers on using the combination of policy management and context information in a formal and efficient way for supporting and managing services. Formal indicates that the specification of context should be depicted through a representational language; efficient means that context are gathered and distributed in many layers (e.g., customer site as well as the core network), and hence requires *semantics* to relate them. This paper presents a novel policy-based service management framework for ensuring the efficient delivery and management of next generation services using autonomic principles. We present an architecture that uses an innovative *fusion* of semantic information that relates the current context to services and resources delivered by the network. Different self-functions, such as self-configuration, are controlled by policy-based management. From a service oriented architecture viewpoint, policies are used for distributing and deploying the necessary services, either atomically or through composition.

This paper provides an overview of our approach. In section II we provide an overview of policy based management, and how it relates to autonomic systems. Section III describes the functionality of the policy-based service management framework, including its corresponding functional blocks and their mutual relationships. It also introduces the “policy-based paradigm” to service management, which is a novel framework for supporting service management functionality in autonomic systems. Section IV presents the policy model that is currently implemented and deployed as our approach to providing next generation services. Section V presents the validation and results for our policy-based management system. Section VI presents the conclusions, and finally the acknowledgements and bibliography references are included.

2 Policy-Based Paradigm

Policy based management has been proven as a useful paradigm in the area of network management. In the last few years, several initiatives have used policy management approaches to tackle the problem of fast and customisable service delivery. These include OPES [5] and E-Services [6]. We go one step further and present an architecture that is intended to control the full service life cycle by means of policies; in addition, it takes into account the variation in context information, and relates those variations to changes in the services operation and performance. The synergy obtained from the autonomic systems and the policy-based paradigm is the knowledge platform, which is another innovative aspect of our work.

A policy has been defined in the sense of administrator-specified directives that manages and provides guidelines for the different network and service elements in [7]. In other words, a policy is a directive that is administratively specified to manage certain aspects of desirable or needed behaviour resulting from the interactions of user, applications and existing resources [8]. In this paper, we use the definition "Policy is a set of rules that are used to manage and control the changing and/or maintaining of the state of one or more managed objects" [4]. The inclusion of *state* is very important for autonomic systems, as state is the means by which we know if our goals have been achieved, and if the changes that are being made are helping or not.

The main benefits from using policies are improved scalability and flexibility for managing services. *Flexibility* is achieved by separating the policy from the implementation of the managed service, while *scalability* is improved by uniformly applying the same policy to large sets of devices and objects. Policy-based management emerged in the network management community and it is supported by standards organisations such as the IETF, DMTF, and TMF [9][10][11]. In next generation network (NGN) usage, the application of policies is being abstracted to facilitate the works of service customisation, creation, definition and management. Another benefit from using policies when managing services is their *simplicity*. This simplicity is achieved by means of two basic techniques: centralised configuration (e.g., each element does not have to be configured individually), and simplified abstraction (e.g., each device does not have to be explicitly and manually configured – rather, a set of policies is established that *governs* desired behaviour, and the system will translate this policy into device-specific commands and enforce its correct implementation.

The main objective of using policies for service management is the same of managing networks with policies: we want to automate management and do it using as high a level of abstraction as possible. The philosophy for managing a resource, a network or a service with a policy-based managed approach is that "IF" something happens "THEN" the management system is going to take an action. The main idea is to use generic policies that can be customised following user subscription; the parameters of the conditions and actions in the policies are different for each user, reflecting its personal characteristics and its desired context information. We use the policy-based paradigm to express the service life-cycle and subsequently manage its configuration in a dynamic manner. It is this characteristic which provides the necessary support and operations of autonomic systems. It should be noted that [4] and [11] propose an important extension and enhancement to the simpler definitions employed by the IETF and DMTF that is very attractive to autonomic systems. Specifically, the definition of policies is linked specific to state management. The following definitions are from [1] and will be used in our work:

"Policy is a set of rules that are used to manage and control the changing and/or maintaining of the state of one or more managed objects."

"A PolicyRule is an intelligent container. It contains data that define how the PolicyRule is used in a managed environment as well as a specification of behavior that dictates how the managed entities that it applies to will interact. The contained data is of four types: (1) data and metadata that define the semantics and behavior of the policy rule and the behavior that it imposes on the rest of the system, (2) a set of events that can be used to trigger evaluation of condition clause of a policy rule, (3) an aggregated set of policy conditions, and (4) an aggregated set of policy actions."

Policy management is best expressed using a language. However, there are multiple constituencies involved (e.g., business people, architects, programmers, and technicians). It is shown in Figure 1. While most of these constituencies would like to use some form of restricted natural language, this desire becomes much more important for the business and end users. This notion was codified as the Policy Continuum in [4][11]. Our approach is based on producing a formal language (with appropriate dialects matched to each constituency) for standardisation in near future.

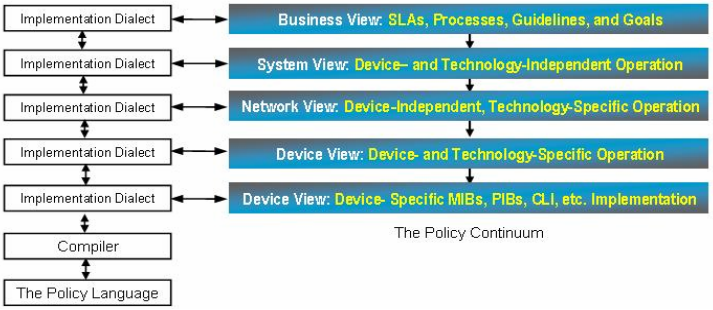

Figure 1. Mapping of Policy Languages to the Policy Continuum.

Our approach defines this set of languages in XML to ensure platform independence. It is also easy to understand and manage, and the large variety of off-the-shelf tools and freely available software provides powerful and cost-effective editing and processing capabilities. Each of the implementation dialects shown in Figure 1 is derived by successively removing vocabulary and grammar from the full policy language to make the dialect suitable for the appropriate level in the Policy Continuum. For some levels, vocabulary substitution using ontologies is used to enable more intuitive GUIs to be built, but this is beyond the scope of this paper.

The policies are used in the managing of various aspects of the services lifecycle, an important aspect of policy-based service management is the deployment of services throughout the programmable elements. For instance, when a service is going to be deployed over any type of network, decisions that have to be taken in order to determine in which network elements the service is going to be installed and/or supported by. This is most effectively done through the use of policies that map the user and his or her desired context to the capabilities of the set of networks that are going to support the service. Moreover, service invocation and execution can also be controlled by policies, which enable a flexible approach for customising one or more service *templates* to multiple users. Furthermore, the maintenance of the code realising the service, as well as the assurance of the service, can all be related using policies. Additionally when some variations in the service are sensed by the system, one or more policies can define what actions need to be taken to solve the problem.

There are important approaches and projects dealing with Policy Based Service Management (PBSM). A survey of policy specification approaches has been provided [12]. The ANDROID Project [13] aims to prove the feasibility of providing a managed, scalable, programmable network infrastructure. A more service layer oriented approach is a proposed project [14]. The overall objective of the TEQUILA project [15] is to study, specify, implement and validate a set of service definition and traffic engineering tools to obtain quantitative end-to-end quality of service guarantees through careful planning, dimensioning and dynamic control of scalable and simple qualitative traffic management techniques within Internet services.

Another interesting proposal is the CONTEXT project [16], the objective is to provide a solution in the form of an architecture that is oriented for creating, deploying and managing context-aware services. This approach is well oriented under the scope of policy based systems, but was developed without considering the multiplicity of context information and the technology non-dependence. Generalised policy-based service management architecture for autonomic systems will articulate a functional process model and include process inter-relationships for an organisation. It shall encompass a methodology that identifies the necessary policies, practices, procedures, guidelines, standards, conventions, and rules needed to support the business and their process inter-relationship enabling the organisation to govern the application of policy management mechanisms. Hence, application of Policy-Based paradigm in the service management area seems to be a feasible alternative for meeting next generation service management goals.

3 Policy-Based Service Management Framework Approach

The PBSM framework that we present deals with the creation and modelling of services, including their context information, in order to provide service assurance and management functions. It does this by constructing a framework using autonomic elements for the management and execution of services. The framework uses programmable networks [17] implemented using autonomic elements to provide the suitable execution environment for the services [2].

The framework allows the use of appropriate APIs for deploying the services in mobile and IP domains. The context-policy model that we propose is flexible enough to accommodate the value of context information that needs to be evaluated, especially if the context changes [18]. Our context-policy model overcomes many of the approaches done in the area of policy-based service management. It is based on standards as much as possible and moreover, the policy model scales with the network or applications that use it. Our context-policy model is expected to be accessible from autonomic elements that can reside in heterogeneous networks and architectures. Storage and retrieval of this information is also important; we meet this requirement by being defined and implemented using an object-oriented philosophy.

The architecture deal with autonomic systems, for instance as an external management method such as a finite state machine, containing the desired state of the system. The autonomic system then compares the *current* state to the *desired* state and, if different, orchestrates any required changes (e.g., reconfiguration). The referred autonomic part is shown in Figure 2.

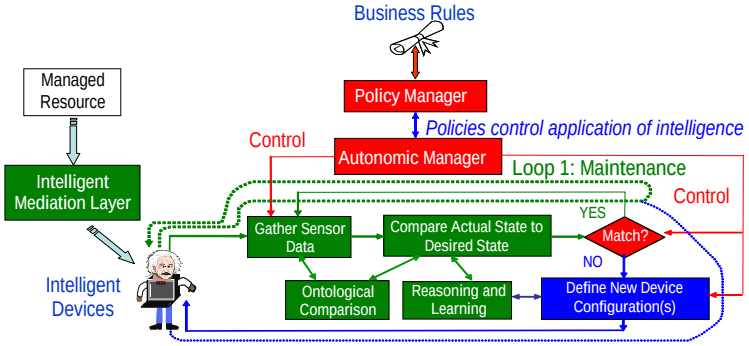

Figure 2. Behavioral Orchestration Using an Autonomic System and Policy Management.

The Intelligent Mediation Layer is a set of software that enables existing (i.e., legacy) and future (i.e., autonomic-enabled) entities to be governed by the autonomic system. Since network devices can have different programming languages and types of management information, we use DEN-ng to provide a Network *lingua franca* (i.e., a controlled vocabulary, along with a consistent set of meanings) for all management information. The "Gather Sensor Data" is responsible for harmonising data received from different sources. The "Ontological Comparison" describes *concepts* and relationships between objects defined in the DEN-ng information model, enabling the Autonomic Manager to *reason* about data received and *infer* meaning. For example, we can find out which customers are affected by a particular SNMP alarm, even though this information is not contained in the SNMP alarm, by looking at relationships in both the information model and relevant ontologies and deducing which customers are affected by the SNMP alarm. The autonomic system then determines whether the actual state of the managed entity is equal to its desired state. If it is, then the system keeps on monitoring the entity. If it is not, then the system defines the set of configuration changes that should be sent to the managed entity. These are translated by the Intelligent Middleware into vendor-specific commands for legacy as well as future devices. The Autonomic Manager provides the novel ability to change the different control functions (such as what data to gather and how to gather it) to best suit the needs of the changing environment. An important, but future, area of research is to add learning and reasoning algorithms to the system. Finally, the Policy Manager is both an interface to the outside world (GUI and/or scripted) as well as the translation of those requests to the autonomic system.

3.1. PBSM Framework Functional Block Description

The framework is composed of four main functional blocks as shown in Figure 3. The Code Distributor is intended to download the generated code to the appropriate code repositories and/or storage points (it could also be downloaded directly to entities that can reconfigure themselves using this mechanism; this is part of our future work). This installation process is a precondition for a service to be delivered. This is because the autonomic system functions by dynamically adapting the services and resources that they deliver, and one way of providing new or different functionality is through self-configuration. In particular, our approach uses a variant of the model driven architecture [19] to achieve this.

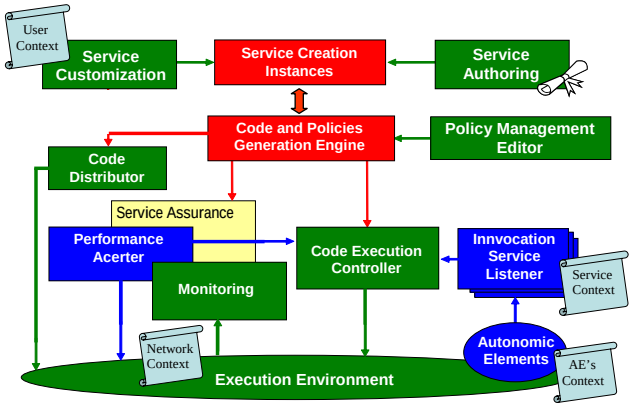

Figure 3. Policy-Based Service Management framework functional components.

Code repositories will be distributed throughout the network infrastructure. Therefore, the Distributor will need to keep track of the URIs where code was installed. The Code Execution Controller is the functional entity that will download the code corresponding to the desired service from one of the repositories to install it in the service execution environment and subsequently activate the service. The Invocation Service Listener captures any of the different types of triggers (e.g., an event, such as a user logging onto the network, or the change in state of a managed entity) that may cause this service to be activated. This component also is responsible for communicating these events to the autonomic elements that provide these services. Finally, the Service Assurance module continuously monitors the quality of service. The usual sequence of events would start with the detection of a trigger (perhaps caused by an explicit or implicit consumer request) by the Invocation Service Listener that detects a problem with the service; the service listener binds the entities shown in Figure 3 to the appropriate autonomic elements and informs the Code Execution Controller about this problem. The Code Execution Controller would compose and forward an activation request message to the appropriate programmable nodes in the service execution environment, which in turn would retrieve code from the repository and execute it, starting the service provisioning phase. As soon as the service has been started, Service Assurance module would keep track of its behavior.

3.2. PBSM Framework Components Description

The PBSM framework is composed of a set of components as shown in Figure 4. The key component is the *Policy Manager*, which supports and governs the functionality of all of the remaining modules *including* the Policy's Consumers. The *Authorisation Check Component* verifies that the user introducing the policy has the necessary access rights to perform this function. This is supported using the role-based access control model of DEN-ng. The *Policy Conflict Check* component is responsible for maintaining the consistency of all policies introduced into the system. E.g. each dialect of the policy language has its own vocabulary and grammar rules. The system will check a policy against these rules to ensure that the policy is well-formed. Decisions of when a policy must be enforced are closely linked with policy's Condition Evaluators, which are coordinated by the Decision Making Component.

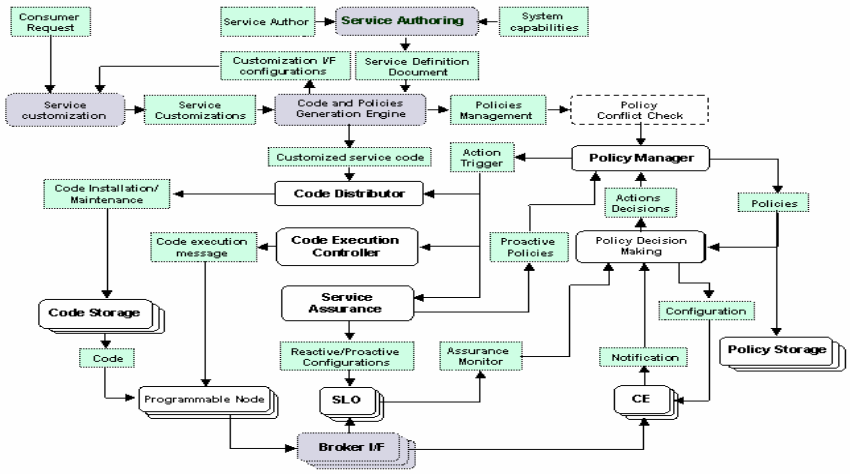

Figure 4. Policy-Based Service Management framework Components and Interfaces.

For example the Decision Making Component checks its list of business rules that must be enforced against the list of submitted policies, and notifies the administrator of any matches. It also provides valuable information for policy conflict detection and resolution. Finally, the *Policy Repository* (Database) gives support to all other components by storing Information Model objects that provide a picture of the system status and the Policies ruling the system at any time. Note that this makes use of the DEN-ng policy model [4], which is an ordered event-condition-action triplet. That is, WHEN an event occurs, IF a set of conditions are satisfied, THEN a set of actions are executed. Policy components are defined and stored as Information Model objects. Using the DEN-ng approach [4][11] the policy model is instantiated.

3.2.1. Policy Manager (PM)

The Policy Manager (PM) has two main functions: 1) to govern the functionality of the system components, and 2) to check and incorporate new policies that arrive from a Policy Generation Engine. The specific fields required by these policies and its general structure are explained in the section "Policy Model". Once a new policy, or policies, arrives as an XML document from the Policy Generation Engine, the PM parses it into Java and sends it to the Policy Conflict Check Component, which checks for conflicts with other policies that are already loaded in the system. This provides the flexibility to a) reject the policy if it conflicts with an already deployed policy, b) replace an already deployed policy with this new policy (which of course involves locating and gracefully stopping policy execution of the previous instance), or c) note that there is an unresolved conflict and wait for a human administrator to resolve it.

The PM uses a sequential algorithm, aided by the atomic characteristics from the policies (described in the policy model), to decide which policies will be activated at any given moment, otherwise it will be stored in the repository for further processing. This is governed by several factors in the DEN-ng policy model [4]. Briefly put, two types of containment semantics are provided by DEN-ng for PolicyRules: grouping and nesting. Grouping is the simpler of the two methods. It has the semantics of an assembly of PolicyRules. The PolicyGroup establishes its own hierarchy for evaluation purposes, and all immediate children (whether PolicyGroups or PolicyRules) that are directly contained within a given PolicyGroup are treated as being at the same level of containment. Finally, the DEN-ng model provides a set of metadata attributes that control the semantics of a PolicyRule or PolicyGroup. The decisionStrategy attribute is used to determine whether a single or potentially a group of PolicyRules should be evaluated, and how it is done. The isMandatory attribute defines whether a rule must be executed, and hence should be used with care, since it requires the evaluation of the condition clause of a PolicyRule and the execution of the actions of a PolicyRule if the condition clause evaluates to TRUE.

3.2.2. Decision Making Component (DMC)

The DMC receives the policy conditions from the PM. When the DMC receives the conditions of a particular policy, it determines when this condition became true and notifies the PM when the condition is met. The PM then notifies the appropriate consumers of the policy that the policy actions are ready to be executed. During the evaluation process three main activities must be performed: obtain the values of the *Condition Objects*; evaluate the *Condition Requirements*; and apply the *Evaluation Method*. The DMC will use the components called Condition Evaluators (CEs) in order to support this evaluation process that deal with the autonomic elements.

3.2.3. Policy & Information Model Repository

The Policy & Information Model Repository is the logical place where policies and other needed management information will be stored and fetched when needed. The Repository will store information about the Policies loaded in the system, information about the network, the available components (CE & AC) and their capabilities (variables they can monitor and actions they can enforce), and other information entities (e.g. management information). This information is of great importance, for example when evaluating conditions, in order to determine whether the appropriate Condition Evaluators are already installed or not to monitor appropriate variables. The Policy Definition System will have to be aware of the different Action Consumer and Condition Evaluator components, as well as the different actions and conditions they are able to enforce and evaluate, in order to effectively use and activate Management Policies.

3.2.4. Policy Conflict Resolution (PCR)

The Policy Conflict Resolution component (PCR) has the responsibility of maintaining the overall consistency of the system. One example of inconsistency would be that two policies require opposite actions when the same conditions are met. The functionality of the PCR is to detect these situations and only accept new policies when they do not introduce inconsistencies into the system. The algorithms required to detect inconsistencies are not straightforward, and an active area of ongoing research. For this reason the implementation of this component is not yet done, and the component has been mentioned only for completeness.

3.2.5. Action Consumers (AC)

The Action Consumers (ACs) are the components for enforcing the policy actions after the request of the PM. The PM will send to the appropriate Action Consumer the action to be enforced and the initial parameters (if needed) to be used in order to do that. The AC will return the results of the enforcement, showing if the action has been successfully enforced. The ACs should be dynamically installed when they are needed. To support this feature, it is necessary that the code(s) of the ACs are stored in one (or more) Code Storage Points, playing the role of a Management Code Repository. An example of one API for these components is given below:

```
public class CodeInstallerInterface {
    public CodeInstallerInterface();
    public boolean InstallCode(String CodeId, URL[] URLList, int NoCopies, int Level, String[] PotentialExecPoints);
    public URL[] GetOptimalURLOfCode(String CodeId, InetAddress DINANodeIPAddr);
    public boolean RemoveCode(String CodeId);
}
```

3.2.6. Condition Evaluators (CE)

The CEs are software components that can be installed in different parts of the network or network components. They interact with the appropriate autonomic elements to get the values required by a given condition. The CEs perform monitoring and requirement evaluation activities, and can be responsible for evaluating requirements (apply Condition Requirements) if all the events or variables (Condition Objects) needed to evaluate the requirements are obtained. The Condition Object is a PolicyStatement [4] that has the generic form {variable operator value}. DEN-ng enables variables and values to be modeled as classes to facilitate reuse; runtime substitution (e.g., with a literal) is also supported. An example of one API for these components is given as follows:

```
public class userPositionController {
   public userPositionController ();
   public int userPosition(String userID, String X, String Y, String Z, String [] arg);
   public int userPosition(String userID, String X, String Y, String Z, String mapReference, String [] arg);
   public int sendMessageToService(String user ID, String X, String Y, String Z, String msg);
   public int sendMessageToService(String userID,  String X, String Y, String Z, String mapReference, String msg);
}
```

4 Policy Model

The policies are structured hierarchically, in terms of Policy Sets, which can be either PolicyRules or Policy Groups. The Policy Groups can contain PolicyRules and/or other Policy Groups. This is enabled through the use of the composite pattern for defining a PolicySet, and is shown in Figure 5. That is, a PolicySet is defined as either a PolicyGroup or a PolicyRule. The aggregation HasPolicySets means that a PolicyGroup can contain zero or more PolicySets, which in turn means that a PolicyGroup can contain a PolicyGroup and/or a PolicyRule. In this way, hierarchies of PolicyGroups can be defined.

The order of execution of PolicyRules and PolicyGroups depends on the structure of the hierarchy (e.g., grouped and/or nested) and is controlled by the set of metadata attributes defined in Section 3.2.1. The high level description of policies follows the following format:

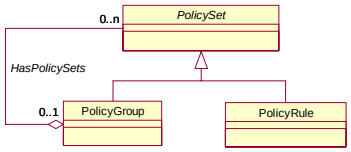

Figure 5. The Composite Pattern Applied to the Defintion of PolicySets

WHEN an event_clause is received that triggers a condition_clause evaluation
IF a condition_clause evaluates to TRUE, subject to the evaluation strategy
THEN execute one or more actions, subject to the rule execution strategy

The above has the following semantics. Policies are not evaluated until an event that triggers their evaluation is processed. Note that the event does not have to be actively sent to the autonomic system by a network component (e.g. as in an SNMP alarm) – passive events, such as the passing of time, can also be included. The Service Management Policies control just the service lifecycle, never the logic of the service. In this way, Service Management policies are used by the PBSM components of the system to define the Code Distribution and Code Maintenance of the service as well as the Service Invocation, Service Execution and Service Assurance processes. Coming from the general requirements associated to the Service Management layer, we have defined five types of policies covering the service lifecycle. These five policy types are structured around an information model whose most representative part is shown in Figure 6.

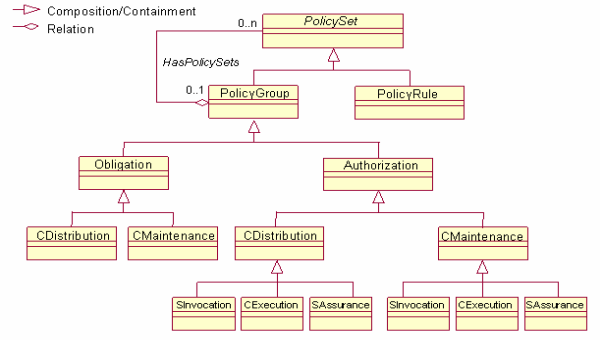

Figure 6. Policy Information Model Hierarchy

We have extended the above model by adding the new concept *atomicity* of a PolicyGroup or PolicyRule. If the first PolicyRule or PolicyGroup is labeled as Atomic, then the second PolicyRule or PolicyGroup will be activated only if the first one has been previously enforced. If it is not then the second Policy can be activated simultaneously with the first one. Note that if a PolicyGroup is labelled as Atomic, then all the Policies contained have to be previously enforced before *any* Policies of the second PolicyGroup can be activated. The different Policy Groups inside a Policy Set or inside other Policy Groups are also ordered with a sequence number. The first Policy Group in the sequence will be processed first, then the next, and so forth.

4.1. Service Code Distribution

This step takes place immediately after the service creation and customisation. It consists of storing the service code in specific storage points. Policies controlling this phase are *CDistribution Policies*. The mechanism controlling the code distribution will determine in which storage points the code is to be stored. The enforcement will be carried out by the Code Distribution Action Consumer. A high level example of this type of policies is presented:

"**If** (customised service B code is received)
then (configure distribution of service B code and optimum Storage Point selection parameters)"

4.2. Service code maintenance

Once the code is distributed, it must be maintained in order to support updates and new versions. For this task, we have the *CMaintenance Policies*. These policies control the maintenance activities carried out by the system on the code of specific services. A typical trigger for these policies could be the creation of a new code version or the usage of a service by the consumer. The actions include code removal, update and redistribution. These policies will be enforced by the Code Distribution Action Consumer. Three high level examples of this type of policies are shown here:

"**If** (new version of customised service B code)
then (remove old code version of service B from Storage Points) & (distribute new service B code)"

"**If** (customised service B code expiration date has been reached)
then (deactivate execution polices for service B) & (remove code of service B from Storage Points)"

"**If** (The invocation's number for service B is very high)
then (distribute more service B code replicas to new Storage Points)"

4.3. Service invocation

The service invocation is controlled by *SInvocation Policies.* The Service Invocation Condition Evaluator detects specific triggers produced by the service consumers. These triggers also contain the necessary information that the policy is going to evaluate in order to determine the associated actions such as addressing a specific code repository and/or sending the code to specific execution environments in the network. The policy enforcement takes place in the Code Execution Controller Action Consumer. A high level example of this type of policies is presented:

"**If** (invocation event X is received)
then (customised service B must be downloaded to the specific IP address) "

4.4. Code execution

CExecution Policies will drive how the service code is executed. This means that the decision about where to execute the service code is based on one or more factors (e.g., using performance data monitored from different network nodes, or based on one or more context parameters, such as location or user identity). Service Assurance Action Consumers are entrusted to evaluate such network conditions and the

enforcement will be the responsibility of the Code Execution Controller Action Consumer. A high level example of this type of policies is presented:

"**If** (invocation event X is received)
then (customised service B must be executed)"

4.5. Service assurance

This phase is under the control of *SAssurance Policies*, *which* are intended to specify the system behaviour under service quality violations. Rule conditions are evaluated by the Service Assurance Condition Evaluator. These policies include preventive or proactive actions, which will be enforced by the Service Assurance Action Consumer. Two high level examples of this type of policies are presented:

"**If** (customised service B is running)
then (configure assurance parameters for service B) & (configure local assurance variables)"

"**If** (level=2)&(parameterA>X) **then** (Action M)"
"**If** (level=2)&(parameterB>Y) **then** (Action N)"
"**If** (level=2)&(parameterC<Z) **then** (level=1)&(Action K)"

The proposed PBSM framework does not assume a 'static' information model (i.e., a particular, well defined vocabulary that does not change) for expressing policies. In contrast, our proposed framework can process policies that can be defined dynamically (e.g., new variable classes can be defined at run-time). The essence of our framework approach is to associate the information expressing policy structure, conditions and actions with information coming from the external environment. Specifically, the externally provided information can either match pre-defined schema elements or, more importantly, can extend these schema elements. The extension requires machine-based reasoning to determine the semantics and relationships between the new data and the previously modelled data. This is new work, an overview of which is in [1], and is beyond the scope of this paper. Information consistency and completeness is guaranteed by a policy-definition system, which is assumed to reside outside the proposed framework (the service creation and customisation systems in the Context system).

By supporting dynamically defined policies, we achieve the flexibility of policy-based management. We think that this feature is indeed a requirement of the design of the overall Context system (for achieving rapid context-aware service introduction and automated provisioning) and that it is supported by our approach.

5 Validation and Results

Many applications that are currently being developed that require (or would benefit from) autonomic computing systems follow the premise of optimising the support for user-oriented services. Our framework is uniquely positioned to do this task efficiently and effectively, since it is *contextually aware* of changes to the environment and/or user. More important, it provides a means for reacting to context changes in a predictable and scalable manner through policy based management. Notably, this avoids requiring the use of skilled resources for simple, manually intensive operations. The automation that these kinds of architectures provide is critical for meeting the requirements of next generation services.

As an example application of this PBSM framework, we have been using a set of scenarios in which the main factor is the non-intervention of specialised network managers. Once the service has been created, the main idea is that the user interacts with the system in a direct way, and the network operator now has a much simpler

and reduced role – to function mainly as a simple authoring entity, and/or an entity that allows the operation. Only occasionally will the network operator have to intervene and define new policies to take direct hard control of the system.

Let's imagine services that need to be deployed on the fly (e.g., services that are pre-defined but that will be deployed when the user signs in to the system or after a certain period of time). Such services can be functions that execute when the user logs in, a video conference or the downloading of multimedia content that starts on demand. Furthermore the services need to be assigned the appropriate quality of service, depending on the type of user as well as user-centric context-sensitive attributes (e.g., this is a user that has just purchased a new service, or this is an established gold user). These scenarios inherently require technological support based on policy that allows them to react in a specific manner based on the specific context.

Using Policies to manage a communication system requires more computer resources than systems that are not policy-based. Figure 7 shows the CPU usage (%) and Memory usage (MBytes) versus the number of policies associated to a given service. We can observe that the CPU usage decreases with the replication of the service invocation rate. This means that the CPU usage decreases when the number of services (users of policy) increases, while the Memory usage increases due to the number of services (policies used to support services) being used. This occurs in all systems (even those without a policy-based paradigm). This feature is important when we try to design a scalable system with CPU limitations. In this scenario, the advantages in using a policy-based management system outweigh the benefit of requiring less computer resources, as provided by those that do not use policy.

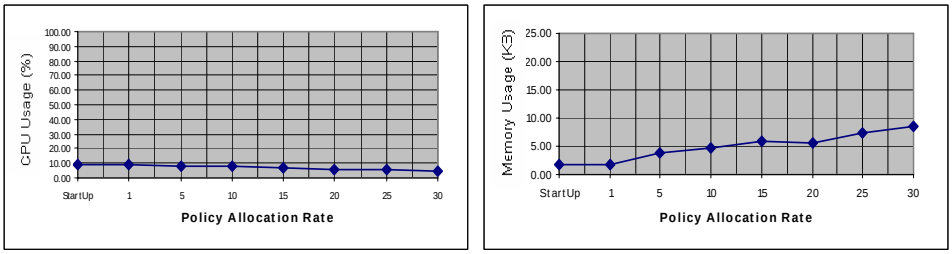

Figure 7**.** CPU-Memory usage vs. Policy Allocation Rate.

The performance study was made using a PC with the following characteristics: Processor Intel Pentium 4 at 2.00 GHz. AT/At Compatible with 785 KB of RAM and supported by Microsoft Windows 2000 5.00.2195 with Service Pack 4; JRE 1.4.2_06 from Sun. and the AppPerfect DevSuite 5.0.1 - AppPerfect Java Profiler [20]

6 Conclusions

The main benefits from using policies for managing services are improved scalability and flexibility for the management of the systems; this also simplifies the management tasks that need to be performed. These scalability and simplification improvements are obtained by providing higher level abstractions to the administrators, and using policies to coordinate and automate tasks.

Policies make automation easier, as well as enable service management tasks to take into account any customisation of the service, made by either the consumer or the Service Provider. We have used the DEN-ng policy model due to its use of patterns, roles, and additional functionality not present in other models (e.g., its support of finite state machines). The extension of service management functionality to act *on*

demand is an important property supported by using policy-based service management systems. We extend the current state-of-the-art in this area in two important ways. 1) We use the novel combination of policy management and context to describe changes to service management functionality and, more importantly, how to respond to these changes. 2) We have built a componentised, inherently scalable architecture for managing services. This architecture has then been linked into a larger autonomic architecture that uses the combination of information models, data models, ontologies, and machine-based learning and reasoning to achieve autonomic behaviour. Significantly, the Autonomic Manger of this architecture, and hence each of the components that it governs, are controlled by our Policy Manager.

The policy-based service management framework approach proposed emerges as a tool to solve some of the autonomic systems requirements in the line of the services management. This framework approach has been conceived to provide an innovative managing tool for next generation services and interact with autonomic systems.

Acknowledgments

This work is being extended and refers partially to the activities done in the framework of the IST-CONTEXT project, a two and a half years research and development project, partially funded by the European Commission. Thank you very much also to all the collaborators that helped in the realisation of this paper.

References

[1] Strassner, J. and Kephart, J., “Autonomic Networks and Systems: Theory and Practice”, NOMS 2006 Tutorial, April 2006.
[2] Strassner, J., “Seamless Mobility – A Compelling Blend of Ubiquitous Computing and Autonomic Computing”, in Dagstuhl Workshop on Autonomic Networking, January 2006.
[3] Kephart, J.O. and Chess, D.M., “The Vision of Autonomic Computing”, IEEE Computer, January 2003. http://research.ibm.com/autonomic/research/papers/
[4] Strassner, J., “Policy Based Network Management”, Morgan Kaufman, ISBN 1-55860-859-1
[5] G. Tomlinson, R. Chen, M. Hoffman, R. Penno, “A Model for Open Pluggable Edge Services”, draft-tomlinson-opes-model-00.txt, http://www.ietf-opes.org
[6] Giacomo Piccinelli, Cesare Stefanelli, Michal Morciniec. “Policy-based Management for E-Services Delivery” HP-OVUA 2001.
[7] Westerinen, A.; Schnizlein, J.; Strassner, J. “Terminology for Policy-Based Management”. IETF Request for Comments (RFC 3198). November 2001.
[8] Verma D. (2000) “Policy Based Networking” 1ª ed. New Riders. ISBN: 1-57870-226-7 Macmillan Technical Publishing USA.
[9] Moore, E.; Elleson, J. Strassner, A. “Policy Core Information Model-Version 1 Specification”. IETF Request for comments (RFC 3060), February 2001.
[10] Moore, E.; “Policy Core Information Model-Extensions”. IETF Request for comments (RFC 3460), January 2003.
[11] TMF, “The Shared Information and Data Model – Common Business Entity Definitions: Policy”, GB922 Addendum 1-POL, July 2003.
[12] Damianou, N.; Bandara, A.; Sloman, M.; Lupu E. “A Survey of Policy Specification Approaches, http://www.doc.ic.ac.uk/~mss/MSSPubs.html
[13] ANDROID Project. http://www.cs.ucl.ac.uk/research/android
[14] Jun-Jang Jeng; Chang, H; Jen-Yao Chung; “A Policy Framework for Business Activity Management”. E-Commerce, IEEE International Conference. June 2003.
[15] IST-TEQUILA Project. http://www.ist-tequila.org
[16] IST-CONTEXT project. http://context.upc.es
[17] Raz, and Y. Shavitt, "An Active Network Approach for Efficient Network Management", IWAN'99, July 1999, Berlin, Germany, LNCS 1653, pp. 220 –231.
[18] Henricksen. K., et al. “Modelling Context Information in Pervasive Computing System”. Proceedings of Pervasive 2002, LNCS 2414, pp. 167-160, 2002.
[19] Please see http://www.omg.org/mda
[20] AppPerfect DevSuite 5.0.1-AppPerfect Java Profiler. http://www.appperfect.com/