

The Design of a New Policy Model to Support Ontology-Driven Reasoning for Autonomic Networking

John Strassner¹, José Neuman de Souza², Sven van der Meer¹, Steven Davy¹, Keara Barrett¹, Dave Raymer³, Srinu Samudrala³

¹Waterford Institute of Technology, Telecommunications Software & Systems Group, Waterford, Ireland
{jstrassner, vdmeer, sdavy, kbarrett}@tssg.org

²Federal University of Ceará, Brazil
neuman.souza@gmail.com

³Motorola Labs
{david.raymer, srini.samudrala}@motorola.com

Abstract The purpose of autonomic networking is to manage the business and technical complexity of networked components and systems. However, the lack of a common *lingua franca* makes it impossible to use vendor-specific network management data to ascertain the state of the network at any given time. Furthermore, the tools used to analyze management data, which include information and data models, ontologies, machine learning algorithms, and policy languages, are all different, and hence require different data in different formats. This paper describes a new version of the DEN-ng policy model, which is part of the FOCALe autonomic network architecture. This new policy model has been built using three guiding principles: (1) the policy model is rooted in information models, so that it can govern managed entities, (2) the model is expressly constructed to facilitate the generation of ontologies, so that reasoning about policies constructed from the model may be done, and (3) the model is expressly constructed so that a policy language can be developed from it.

Keywords: Autonomic Networking, FOCALe Autonomic Architecture, Next Generation Services, Ontology-Based Management, Policy Management, Semantic Reasoning.

I. INTRODUCTION

The business, technical, and even social aspects of systems have increased dramatically in complexity, requiring new technologies, paradigms and functionality to be introduced to cope with these challenges [1]. This increase in complexity has made it almost impossible for a human to manage the different operational scenarios that are possible in today's communication systems. While IP network management problems have been extensively documented [1][5][8][17][24], wireless systems present even more difficult problems. For example, wireless failures are usually not obtainable from a set of attributes – they must be *inferred*. Key performance and quality indicators (KPIs and KQIs) are calculated to provide a machine-interpretable view of system quality as perceived by the end user for a particular type of wireless system for a specific set of radio access technologies (RATs). However, current RATs use a set of non-compatible

standards and vendor-specific functionality. This is exacerbated by current trends, such as network convergence (which combine different types of wired and wireless networks), as well as future multi access mode devices [2] and cognitive networks [3], in which the type of network access can be dynamically defined. The vision of Seamless Mobility [4] is even more ambitious – the ability for the user to get and use data independent of access mode, device, and media.

Part of the allure of Policy-Based Network Management (PBNM) [5] was its simplicity in providing different services to different users while automating device, network and service management. However, most PBNM systems have been low-level systems that manage changes in commands for routers, switches, and firewalls. Hence, there is no link between business needs and the configuration of network resources and services. In addition, relatively new concepts, such as using context changes to determine which network services and resources to modify, are not present.

Our approach for solving this problem is realized as the FOCALe autonomic architecture [8], and is based on five key concepts. First, the use of a shared information model is required in order to harmonize the different data models that are used in Operational and Business Support Systems (OSSs and BSSs). Second, since information and data models are not capable of representing the detailed semantics required to reason about behavior, we augment our use of knowledge extracted from information and data models with ontologies. Third, we define a new, enhanced policy model that, while constructed as an information model, has been specifically designed to be able to generate ontologies for governing behavior. Fourth, we link this policy model to a new context model, so that policies can be written that adapt offered resources and services to sensed context changes. Finally, we outline how policies are used, together with machine learning and reasoning, to build a new adaptive control architecture.

The organization of this paper is as follows. Section 2 provides a brief introduction to autonomic computing and networking. Section 3 summarizes current PBNM approaches. Section 4 describes our new policy model in detail. Section 5 links this model to a context model. Section 6 describes initial progress that we have made in building an adaptive control architecture. Section 7 summarizes the paper.

II. AUTONOMIC NETWORKING OVERVIEW

This section discusses the difference between *autonomic computing* and *autonomic networking*.

A. Autonomic Computing

The purpose of autonomic computing is to manage complexity. The name was chosen to reflect the function of the autonomic nervous system in the human body. By transferring more manual functions to involuntary control, additional resources (human and otherwise) are made available to manage higher-level processes.

The fundamental management element of an autonomic computing architecture is a control loop, as defined in [6][7]. IBM's version is shown in Figure 1. The idea is to instrument a Managed Resource so that an Autonomic Manager can communicate with it. This is done using sensors that retrieve data, which is then analyzed to determine if any correction to the managed resource(s) being monitored is needed (e.g., to correct "non-optimal", "failed" or "error" states). If so, then those corrections are planned, and appropriate actions are executed using effectors that translate commands back to a form that the managed resource(s) can understand. The Autonomic Element embodies the control loop, and enables the autonomic manager to communicate with other types of autonomic and non-autonomic managers using its sensors and effectors.

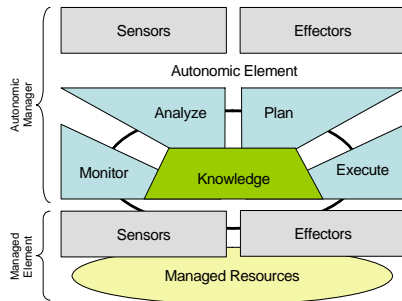


Figure 1. Autonomic Computing Control Loop

B. Autonomic Networking

The motivation behind autonomic networking is to identify those functions that can be done without human intervention to reduce the dependence on skilled resources for managing devices, networks, and networked applications. If the autonomic network can perform manual, time-consuming tasks (such as configuration management) on behalf of the network administrator, then that will free the system and the administrator to work together to perform higher-level cognitive functions, such as planning and optimization of the network.

One difference between autonomic computing and autonomic networking is that the latter must cope with and coordinate multiple control mechanisms (used by different networks), which the former usually doesn't consider. Motorola has defined a new management approach that is equally appropriate for legacy devices and applications as well as for next generation and cognitive networks. Figure 2 shows a simplified version of our FOCALe autonomic architecture [8], which was designed to support this need as follows. Multiple networks and network technologies require

multiple control planes that can use completely different mechanisms; this makes managing an end-to-end service difficult, since different management mechanisms must be coordinated. FOCALe addresses this through model-based translation. Second, in current environments, user needs and environmental conditions can change without warning. Therefore, the system, its environment, and the needs of its users must be continually analyzed with respect to business objectives. FOCALe uses inferencing to instruct the management plane to coordinate the (re)configuration of its control loops in order to protect the current business objectives of the organization.

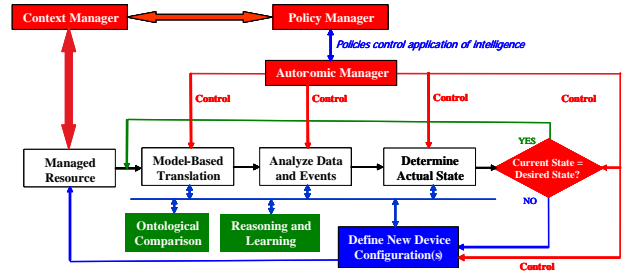


Figure 2. Simplified FOCALe Autonomic Architecture

The key to FOCALe's *adaptive control loops* is the interaction between the context manager, policy manager, and autonomic manager. Conceptually, the context manager detects changes in the network, or in user needs, or even in the business; these context changes in turn trigger a new set of policies to take over control of the autonomic system, which enables the services and resources provided by the autonomic system to *adapt* to these new needs given that appropriate policies are available for the new context. The autonomic manager uses these policies to govern each of the architectural components of the control loop, enabling the different control loop components to change the type of algorithm used, the type of function used, and even the type of data to use as a function of context. This is why policy management is so important to FOCALe.

FOCALe is self-governing, in that the system senses changes in itself and its environment, and determines the effect of the changes on the currently active set of business policies. In general, those changes could either cause a new set of business policies to be activated, or endanger one or more goals of the currently active set of business policies. In the latter case, FOCALe reconfigures the system to ensure that the currently active set of business policies is not violated and observes the results [8].

FOCALe responds to both changing user needs as well as changing conditions in the business and in the network infrastructure through the use of a Policy Continuum [5], an exemplar of which is shown in Figure 3. The concept of the Policy Continuum is essential for next generation networks and services, as it enables the requirements for different OSS and BSS components to be translated among different groups of users and applications. The Policy Continuum implies a common set of translations between different policy rules, so that the PBNM system can embrace multiple constituencies and help them work together on OSSs and BSSs.

In particular, each view of the Policy Continuum is optimized for a different type of user that needs and/or uses slightly different information, even though they interact with the same managed entities. For example, the business user wants Service Level Agreement (SLA) information, and isn't interested in the type of queuing or other traffic conditioning functions that will be used to support the SLA. Conversely, the network administrator may want to develop Command Line Interface (CLI) or Simple Network Management Protocol (SNMP) commands to program the device, and may need to have a completely different representation of the policy in order to develop the appropriate command changes. Thus, the requirement is for policy to be treated as a continuum, where different policies take different forms and address the needs of different users. Note that unless there is an information model that can be used to relate these different forms of policy to each other, it becomes difficult (if not impossible) to define a set of mappings that transform the data between each type of policy in the continuum. This is one of the cornerstones of the DEN-ng policy model [5]. Specifically, it provides a layered set of policies with different levels of abstractions, and model mappings to translate between them.

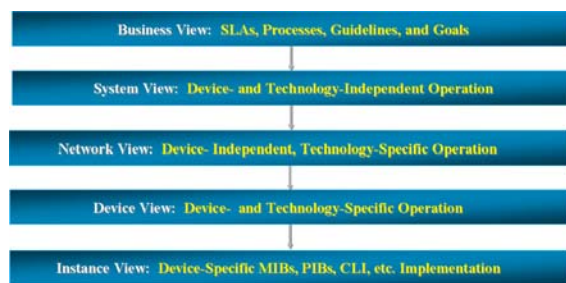


Figure 3. An Example of the Policy Continuum

III. CURRENT PBNM APPROACHES

Policy-based Network management (PBNM) [5] is a concept developed originally to reduce the administrative complexity of reconfiguring a device and/or a network to respond to the changing conditions of the business and the infrastructure. The manual process of reconfiguring the network is very difficult; two important sources of this difficulty are the challenge of enabling the business to determine the set of network services and resources to be provided at any given time, and because of the vast amount of heterogenous devices a network comprises [5]. PBNM aims to decrease this complexity and its associated cost by automating, to some degree, the reconfiguration process. This concept corresponds to the vision of self-governance innate in autonomic communications [9].

[11] provides a comprehensive, but slightly dated, survey of common approaches. One of the key points that is raised in [11] is that there is a marked separation between policy languages that are applied to specific domains. For example, the syntax, constructs used, and underlying model are very different between existing security and management policy languages. Three languages that claim to be independent of domain are briefly discussed below. Note the following shortcomings of all of these related works:

- None address the multiple stakeholders and views embodied in the Policy Continuum concept
- None combine information models and ontologies
- None provide a common *lingua franca* that enables different management data to be harmonized

The Policy Technologies group at the IBM T.J. Watson Research Center has developed the Policy Management for Autonomic Computing (PMAC) technology. PMAC is embedded within software applications, and is positioned as IBM's policy based autonomic management infrastructure. PMAC has a policy language called the Autonomic Computing Policy Language (ACPL), which in turn uses the Autonomic Computing Expression Language (ACEL) [19]. An "autonomic manager" tool can make decisions based on policies (business rules), created by the developer to make applications capable of self-managing and self-configuring [10]. However, PMAC only uses a condition-action tuple, which creates efficiency and predictability problems. For example, continuously checking every condition against every event is not going to be computationally efficient, nor will it produce temporally predictable firing of policy actions without prescriptive real-time requirements. Additionally, the predictability problems that may be introduced can open the door to a wide range of anomalous runtime behaviors.

Ponder, a policy language for distributed system management, has been developed as part of ongoing work in Imperial College, London. The language, which is declarative, strongly-typed and object-oriented, codifies six policy types, namely positive authorization, negative authorization, refrain, positive delegation, negative delegation and obligation. While these policy types allow for a rich policy set to manage the behavior of a domain, the language has some shortcomings. Both positive and negative authorizations are target based, which implies that a subject can't specify positive authorization policies to control its own behavior regarding interaction with a target. Refrain policies are approximately equal to subject based negative authorization policies, although the semantics of 'must refrain from' are not as strong as 'not authorized'. Moreover, refrain, positive authorization and negative authorization policies do not have an event clause, which can cause efficiency and predictability problems as discussed in relation to PMAC. Delegation policies, both positive and negative, allow the subject of an authorization policy to delegate temporarily access rights to a grantee. However, authorization policies are target based therefore it seems imprudent to allow the subject of an authorization policy to delegate the access right permission/revocation. Finally, the semantics of the Ponder policy language [24] itself have not been formally defined, making automated reasoning over policies, as required for autonomic networking, not possible through formal means.

The University of Maryland, Baltimore, has developed Rei to facilitate the definition of deontic logic based policies that are used to manage the behavior of agents operating in open distributed environments. More specifically, Rei defines constructs for right, prohibition, obligation and dispensation policies. However, unlike Ponder, Rei does not specify explicitly if a policy is target based or subject based, which can be limiting for policy conflict detection. OWL-Lite is used to encode the grammar of Rei; therefore individual instances of policies are defined as individuals/slots of the

defined classes and properties. Rei also uses OWL-Lite to reason over domain knowledge expressed in either RDF or OWL. To overcome the issue of defining policies that contain variables, which is inherited by implementing OWL to encode the grammar, Rei uses placeholders similar to those used in Prolog. This non-standardized extension, however, means that (DL Implementation Group) DIG reasoners and the REI engine are able to reason about the domain-specific knowledge, but not about all policy specifications. Finally, the most important critique of the Rei policy specification language is that the semantics of the language have not been formally defined. A natural language description (English) of the semantics has been provided but this is of little benefit for automated knowledge acquisition and inferencing [25][26].

IV. THE DESIGN OF THE NEW POLICY MODEL

The DEN-ng model is built on two important concepts: patterns and roles. DEN-ng is the *only* object-oriented information model that uses patterns [12][13].

A. The Importance of Using an Information Model

Since the aim of autonomic networking is to manage networked applications and systems, we need a means to represent the characteristics and behavior of system elements. ITU-T recommendations, along with IETF and other fora documents, describe current state data for managed objects, but do not describe how to manage the lifecycle aspects of managed objects. Hence, we turn to information models.

There are three main information models being used today: DEN-ng [5], the TMF SID [30] and the DMTF CIM [31] (other efforts are too sparse in their domain coverage to meet the needs of autonomies). The SID is partially derived from DEN-ng v3.5; unfortunately, the aims of the TMF diverged, and DEN-ng (the new version is 7.0) will now be submitted to the Autonomic Communications Forum. Parts of the original DEN-ng (v3.5) model that were defined in the SID are covered in the next section.

The TMF has had a five-year liaison relationship with the DMTF, trying to align the DMTF CIM with the TMF SID. This work has been hampered by three important problems: (1) the lack of a common metamodel used by the SID and the CIM, (2) the type of modeling done, and (3) the lack of the use of patterns. The CIM uses its own proprietary metamodel, not that of UML [32]. Since a metamodel defines the constructs used to build a model, such as classes, attributes and relationships, this means that the concepts used to build models (e.g., classes, attributes, and relationships) in a UML model vs. the CIM model are defined differently. The CIM is in reality a data model, as it contains technology-specific concepts, such as keys and weak relationships, which are not technology neutral. Information models by definition require technology neutrality, or else coherency between the same knowledge represented in different forms in different data models will be lost. Finally, patterns play a critical role in model-driven design. The CIM does not use patterns. Successful transformation between models requires that the original model to contain enough details and semantics to completely guide software tools through the process. By incorporating the semantics through the use of patterns, instead of manual insertion, we gain multiple benefits:

- Design of an architecture is less time consuming
- Resulting model is easier to learn, since it reuses familiar concepts
- Resulting model reflects the collective wisdom of all contributors, and
- Tools can work with the patterns through the transformation, ultimately pulling implementation code from a library written by experts and inserting it into the final application and/or system

DEN-ng also uses roles [14]. Roles make a design inherently scalable by abstracting individual users, devices, and services into roles that can be played by various managed entities. DEN-ng is unique, in that roles are not limited to just people; rather, they may include roles representing resources, services, products, locations, and other managed entities of interest. In [15], both the entity requesting access as well as the target entity being accessed has roles; in addition, the application of policy is done using roles, both to represent the subject and the target of policy. Note that this paper uses the old DEN-ng policy model.

B. The Original DEN-ng Policy Model

We start with the original DEN-ng policy model, as it has been standardized in several fora and proven to be an excellent model. Figure 4 shows the root of the old DEN-ng policy model.

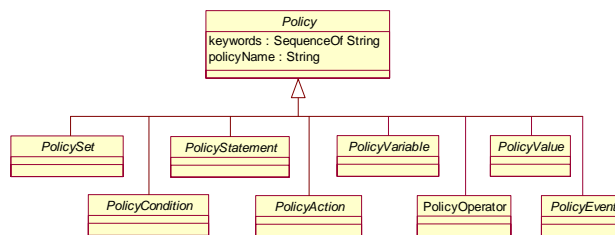


Figure 4. Root of the “Old” DEN-ng Policy Model

In this design, the Policy class is the root of the class hierarchy that realizes the Policy model. As such, it defines common attributes, methods and relationships that all policy subclasses use and take part in. The PolicySet subclass defines two types of collection classes, PolicyRule and PolicyGroup. The PolicyRule class collects PolicyEvents, PolicyConditions, and PolicyActions, while the PolicyGroup class collects PolicyRules and PolicyGroups. Two important and powerful features of this arrangement are that a PolicySet defines a common decision strategy and a common set of PolicyRoles to be used by the PolicyGroups and the PolicyRules that inherit from it.

The PolicyRule class, which is a subclass of the PolicySet class, defines the “event-condition-action” semantics that form a DEN-ng policy rule. The semantics of this rule are that the rule is evaluated when an event occurs. If the condition clause is satisfied, then the pass-action clause will be executed (otherwise, the optional fail-action clause will be executed). The PolicyEvent, PolicyCondition, and PolicyAction classes are aggregated by the PolicyRule class. (The actual DEN-ng model is more complex; for example, different policy rule components can be selectively enabled and disabled, or put into a special test mode. The new DEN-

ng model has these additional features. This paper will ignore these and similar features, as they are not germane.)

The PolicyStatement class models the triplet {variable, operator, value} that is used by both the PolicyCondition and PolicyAction classes. Note that the semantics are reflected in the types of operators that are allowed to be used in each case. For conditions, we want the semantics of "variable relates to value", where "relates to" is usually the match operator, but could also be other applicable operators (e.g., a comparison operator). For actions, we want the semantics of "set variable to value". Here, the only operator allowed is the set operator. Both of these semantics are enforced using OCL. The ability to use the same basic form for writing PolicyConditions and PolicyActions greatly simplifies the design and implementation of a policy management system by enabling such statements to be automatically generated.

Figure 5 shows the old PolicyRule model in more detail. The PolicyRule aggregates a set of one or more PolicyEvents, PolicyConditions, and PolicyActions. Note that each of the three aggregations shown in Figure 5 has a cardinality of 1..n on the component side – this means that a PolicyRule must aggregate at least one or more PolicyEvent, PolicyCondition, and PolicyAction in order to be considered a correct instantiation of a PolicyRule.

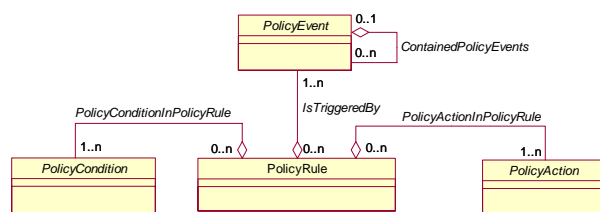


Figure 5. The “Old” DEN-ng PolicyRule class

C. Rationale for Building a New Policy Model

The original DEN-ng policy model has worked well, as documented in [5]. However, the old DEN-ng model was never intended to be used in conjunction with a set of ontologies to *infer* and *understand* data. Information models are meant to represent known facts, not for discovering new knowledge. Unfortunately, autonomic networks require both the discovery (and subsequent classification) of new knowledge as well as the updating of existing knowledge for several reasons. First, network devices are inherently extremely complicated. It is impossible to build a complete model of (for example) a router, because in order to do that, one would have to model its static properties (e.g., an interface) as well as its dynamic properties (e.g., instantiating a VPN requires use of shared resources) and behavior (e.g., modeling of particular bugs and side effects). Hence, what is needed is to construct a model that approximates the static properties of the router, and use a *different* tool to model its dynamic and behavioral properties.

One might ask why UML is not able to model the dynamic and behavioral properties of a managed element, like a router. There are several reasons, all of which are beyond the scope of this paper. Suffice it to say that in order to model dynamic properties and behavior, we need a set of mechanisms beyond what UML currently provides. The deficiencies in UML are particularly felt in the areas of

meaningful mechanisms for modeling actions and the semantics of said actions, semantic and ontological imprecision as well as poor support for model instance management and manipulation.

The problem is that networks are made up of heterogeneous devices, each with its own vendor-specific concepts and implementation dependencies. Hence, incompatibilities between different means of representing and processing information, along with defining the meaning of said information, arise. These issues result in a phenomenon called cognitive dissonance. Cognitive dissonance was first introduced in 1957, by Dr. Leo Festinger, then a member of the Department of Psychology at Stanford University. Simplistically, cognitive dissonance results in a situation where two supportable, held beliefs are in opposition to each other. In terms of UML models, this most often results when considering instances of models due to the lack of precision in the specification of relationships in the presence of inheritance hierarchies. Furthermore, UML does not contain the constructs necessary to support the definition of knowledge or reasoning about knowledge, which requires formal semantic definitions that enable unambiguous representations and organization of the representations that facilitate the calculation of semantic similarity construction [16]. As another example, UML does not have the mechanisms required to establish semantic relationships between information (e.g., synonyms and antonyms). For example, there is no ability in UML to represent explicit semantics (i.e., the definitions used in UML are implicit). In addition, relationships, such as “is similar to”, that are needed to relate different vocabularies (such as commands from different vendors) to each other, along with tense, modals, and episodic sequences, cannot be defined.

Thus, when different terms need to be equated, existing standards-based models and vendor programming commands, such as SNMP and vendor-specific command-line interfaces, express knowledge differently. End-to-end management requires different commands from different vendors to be equated. This is, conceptually, a graph-matching problem, where one concept in one model (which might map to one or more model elements) must be equated to another concept in the other model. Most network management devices and applications do not share a common vocabulary, which necessitates a mapping between these two sub-graphs. The different semantics of each model require semantic similarity mappings to resolve these problems [16]. For example, it is common to find a single command from one vendor map to multiple commands from another vendor.

In addition, autonomics requires traceability of configuration changes made. Hence, it is important to be able to reason about changes in past and present configurations, and what needs to be done in the future. If network management is going to be able to be made easier to understand by non-experts, then a natural language interface, or a restricted form of this interface, must be built that includes concepts such as modals and other modifiers (e.g., mood and tense). This also includes episodic sequences, which are used to formalize the set of actions performed in a configuration. Interaction diagrams fall short of this, because of their lack of semantics and their inability to specify irregular patterns.

These limitations also make it impossible to do basic functions, such as define the set of simple network management protocol (SNMP) monitoring commands that can determine if a command line interface (CLI) configuration command was successful or not. Even though UML has defined extension mechanisms (profiles, tagged data, and stereotypes), those mechanisms are insufficient to guarantee interoperability, because they require too much additional custom processing that is not specified in the standard. In [17] and [18], initial work on the fusion of information models and ontologies is described; this will be extended in future work to include the fusion of machine learning and reasoning mechanisms. This fusion is critical – models and ontologies represent facts and extended semantic meaning describing the facts; reasoning enables hypothesis generation using those facts and meanings; learning accelerates the incorporation of knowledge. It is only through this knowledge fusion approach that autonomic systems can relate diverse data and understand the current state of the system, and hence any reconfiguration that is required.

Figure 6 shows the concept of a “specification” class, which is one of the fundamental patterns of DEN-ng. The purpose of a DEN-ng specification class is to define all of the invariant attributes, methods, relationships, constraints, and other model elements for a concept, so that different elements can be instantiated with those invariant characteristics and behavior intact. Figure 6 shows that the decisionStrategy attribute is defined to be common to all types of PolicyRules. Hence, we can define completely different types of policy rules – utility functions, goal policies [33] and of course event-condition-action policies – and all of these different types of policies will use the same decisionStrategy attribute.

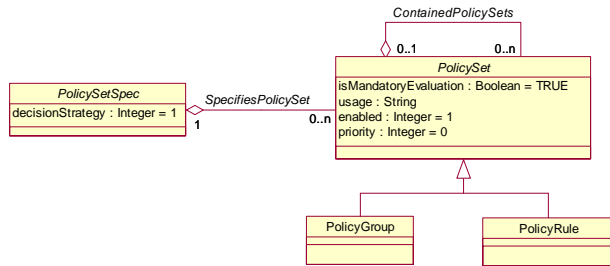


Figure 6. PolicySetSpecification Class

Specification classes can be subclassed, just as any other class. Figure 7 shows the PolicyRuleSpecification class.

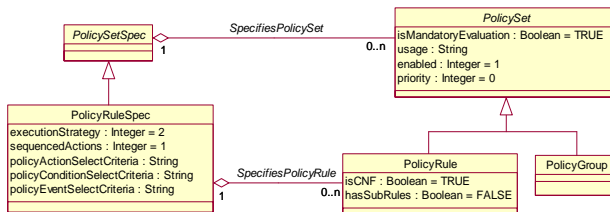


Figure 7. PolicyRuleSpecification Class

Similar to Figure 6, the PolicyRuleSpec class defines the invariant model elements that *all* PolicyRule types will use. The use of the specification pattern is very powerful, as it

ensures that multiple people and groups will use the *same* core elements of a concept.

D. The New DEN-ng Policy Model

The policies currently used in our project follow a standard event-condition-action model: we monitor events generated when new system data is available, and use these events to trigger the evaluation of the conditions of one or more policy rules. If the conditions are matched, then one or more policy actions are executed. In our prototype system, we currently have two layers of policy control – the top layer takes events from the wireless system and analyzes these to perform a causal analysis, which will classify the reason for the KPI or KQI violation. This then serves as an event for the lower-level policy layer, whose purpose is to determine how best to fix the problem automatically (or instruct humans what needs to be done). The benefit of having two layers is to provide flexibility in writing policies for the different tasks of classifying and fixing problems in the network.

The new model uses all of the power of the existing model, such as patterns and roles. Therefore, we will concentrate on just the structural changes to the model that are necessary to make it easier to generate ontologies.

Figure 8 shows the root of the new DEN-ng policy model (note that this is different from the DEN-ng policy model described in [5]). A PolicyConcept is an abstract class, and is the root of the new DEN-ng Policy model. As such, it defines common attributes, methods and relationships that all policy subclasses use and take part in.

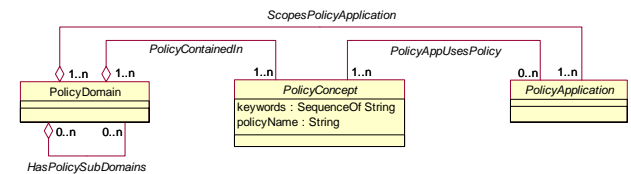


Figure 8. Root of the New DEN-ng Policy Model

This class is named PolicyConcept because it does not define the characteristics and behavior of a Policy Rule (or any of its components); rather, from a classification theory point-of-view, it defines a new part of the overall DEN-ng model that is concerned with modeling generic concepts related to policy. Note that this also helps in generating ontologies from the model, in that the name makes clear that this is not a Policy Rule or a component of a Policy Rule. This is in direct contrast to other models (e.g., the CIM, whose root class is called “Policy”); we have found that this generates confusion, since “policy” and “policy rule” are often used interchangeably).

A PolicyDomain is a collection of entities and services that are administered in a coordinated fashion using a set of policies. The policies are used to control the set of services and entities according to a common methodology, such as a finite state machine. The HasPolicySubDomains aggregation enables a PolicyDomain to contain other PolicyDomains.

A PolicyApplication defines concepts and components that can use policy entities to manage the behavior and configuration of other entities. A PolicyApplication, as a minimum, consists of entities to provide policy decision-making capabilities, as well as entities to enforce and execute policies. It also includes entities to coordinate management

and usage aspects of policy as well as entities to enable policy components to scale to large distributed systems. This class is used as a convenient place for defining relationships to managed entities that it governs as well as managed entities that provide policy management functionality. It has three principal subclasses: PolicyBroker, PolicyController, and PolicyControllerComponent. The principal role of a PolicyApplication is to define generic relationships that its subclasses can participate in.

A PolicyController is a fundamental building block of a DEN-ng based policy management system. It represents both a set of core functionality for implementing policy as well as a unit of distribution in a distributed implementation. A PolicyController is an entity that is either manufactured or is constructed by integrating the functionality of different PolicyControllerComponents. PolicyControllerComponents are modular units of functionality that are used to construct policy management applications. Large systems will employ multiple policy management systems for various reasons, such as geographic distribution, separate management of different domains, and others. A PolicyBroker is used to coordinate and control how different PolicyServers interact with each other. In this regard, it has two different functions. The first function is to ensure that conflicts between different policy rules don't exist when different Policy Servers are asked to work together. The second is to coordinate the application of different policies in different Policy Servers.

The PolicyContainedIn association defines the set of Policies that reside in a particular PolicyDomain (remember that PolicyConcept is the superclass for the classes that together constitute the definition and representation of a Policy Rule and its components).

PolicyApplications are related to a given PolicyDomain through the ScopesPolicyApplication aggregation. When this is combined with role-based access control [19][20][21][22], different types of PolicyApplications, as well as different types of Policies that a PolicyApplication uses, can have their access and usage limited to roles that different users takes on.

Now, let's examine the changes in subclassing. The first area is the definition of a Policy Rule itself. In the "old" DEN-ng model, the purpose of the PolicyRule class was to model a policy rule in its entirety. This means that a PolicyRule must contain all concepts that are required for a PolicyRule to be instantiated. This, from a pure semantic viewpoint, is incorrect. For example, the concepts of policy subject, policy target, and policy continuum level (not shown in the old policy model; suffice it to say that they were all directly related to PolicyRule in some fashion) have nothing to do with the representation of a PolicyRule. This is all that is being defined at this level of the hierarchy – the structural representation of a Policy Rule. Hence, we separate the structural representation of a Policy Rule from other semantic aspects of that Policy Rule. This means that, while it is still OK to subclass PolicyRule, PolicyEvent, PolicyCondition, and PolicyAction directly from PolicyConcept, we need to ensure that the new PolicyRule class is restricted to codifying the event-condition-action structure of a policy rule.

Before we do that, it is important to remember that there are many different types of Policy Rules. No policy model to date has actually tried to represent more than one basic type of policy (from a structural point-of-view). We want the new

DEN-ng model to be able to represent all types of relevant policy rules. Hence, we need to be able to again separate the specific structural representation of a type of policy rule from the label "Policy Rule". As a simple example, if we define a PolicyRule to be of the form event-condition-action, then how do we represent a policy rule from the DMTF or IETF, which only uses simple condition-action representations? (The reason that we want to be able to do this is to be able to map between different policy rule representations that different systems may use; however, this is beyond the scope of this paper.)

Hence, we define the ECAPolicyRule class to define a particular type of PolicyRule - one that has an {Event, Condition, Action} structure, as shown in Figure 9.

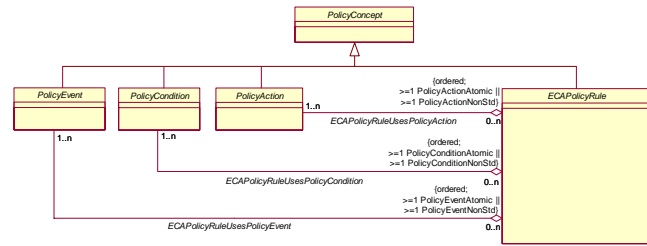


Figure 9. The ECAPolicyRule Class

In keeping with the original DEN-ng model as defined by [5], this class represents an intelligent container that gathers metadata and PolicyEvents, PolicyConditions, and PolicyActions. As such, it doesn't have an inherent relationship with PolicySubject, PolicyTarget, or any particular level of the Policy Continuum; these all represent bound semantics for a particular use of an ECAPolicyRule. Note that, in Figure 11, we will introduce superclasses of PolicyEvent, PolicyCondition, PolicyAction, and ECAPolicyRule, called PolicyRuleComponentStructure (for the first three) and PolicyRuleStructure (for the last). This enables different types of policy rule component and policy rule structures to be represented. The metadata referred to above is shown in Figure 10.

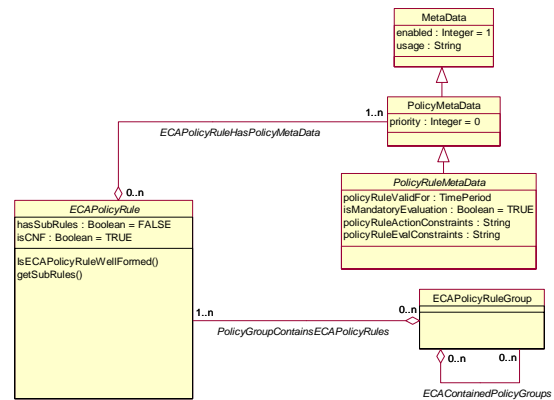


Figure 10. PolicyMetadata and PolicyGroup Classes

The PolicyMetadata class defines metadata that applies to different types of Policies, such as (but not limited to) ECAPolicies. This decouples common metadata that different Policy representation systems need from the actual realization of the Policy. Hence, ECA- and non-ECA Policies can both use the metadata contained in this class. It also decouples the

representation and structure of a particular type of policy (e.g., an ECAPolicy) from the metadata. This is critical for properly constructing ontologies from policy models.

Figure 10 also shows another change. In the old DEN-ng model, the ContainedPolicySets aggregation was used to form groups of PolicySets (i.e., PolicyRules and/or PolicyGroups). In the new DEN-ng model, this is done via two explicit aggregations, since the PolicySet class no longer exists. The ContainedPolicyGroups aggregation is used solely to nest PolicyGroups. In contrast, the PolicyGroupContainsECAPolicyRules aggregation is used to define the set of ECAPolicyRules that are contained in a particular PolicyGroup. The separation of the (old) single aggregation into these (new) two aggregations makes the containment semantics explicit – this type of change is precisely what is needed in order to construct our ontology.

The above discussion has shown that there is no longer any value in keeping the PolicySet class. This makes sense from the ontology point-of-view: since there is a distinct semantic difference between a PolicyGroup (which is just a simple container) and an ECAPolicyRule (which has a fixed structural semantic form), the old concept of a PolicySet (which was to define a common container) is no longer applicable. While both the new PolicyGroup and ECAPolicyRule classes are containers, they contain very different things and have very different semantics.

The next part of the puzzle is to define the concepts of PolicySubject and PolicyTarget. A PolicySubject is a set of entities that represent the authority imposing policy. The PolicySubject can make policy decision and information requests, and it can direct policies to be enforced at a set of PolicyTargets. A PolicyTarget is a set of entities that a set of policies will be applied to. The objective of applying policy is to either maintain the current state of the PolicyTarget, or to transition the PolicyTarget to a new state.

In the old DEN-ng model, these were directly associated with a PolicyRule. In the new DEN-ng policy model, we could associate them with a new subclass of the ECAPolicyRule. However, this has the unfortunate effect of binding the usage of PolicySubject and PolicyTarget to an ECAPolicyRule. Instead, we make the modifications shown in Figure 11.

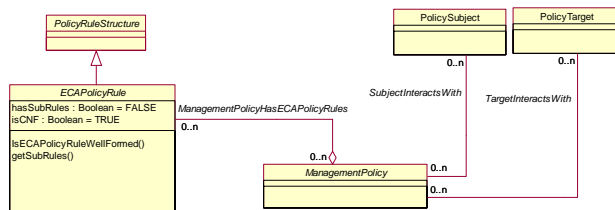


Figure 11. Improved ECAPolicyRule Design

The PolicyRule Structure class is used to define the concept of representing the structure of a policy rule independent of any other semantics. The ManagementPolicy class is used to realize deontic actions (i.e., obligations and permissions) that will be used to make management decisions. DEN-ng adds the notion of delegations as well. Note that in the old DEN-ng design, the semantics of deontic policies could only be defined as subclasses of the PolicyRule class.

This is a weak definition of deontic logic, since it is essentially saying that the concept of deontic logic is derived from the structural representation of a policy rule combined with associations to policy subjects and policy targets. With our new approach, we can define the concept of deontic logic separate from how it is represented, thereby allowing different structural forms of policy rules to be able to be used to represent deontic logic. Furthermore, we reinforce the concept that management is a deontic process by defining the ManagementPolicy abstract class to be the superclass of all deontic policy rules. Finally, since the associations between ManagementPolicy and both PolicySubject as well as PolicyTarget are completely optional (since their multiplicity is 0..n – 0..n), we are free to define deontic management as a function of PolicySubject and/or PolicyTarget. The flexibility described here does not exist in any other policy model to date.

Similarly, Figure 12 abstracts the different types of PolicyRuleComponents by introducing a new superclass. Since different types of Policy Rules have different structural components, the PolicyRuleComponentStructure class is used to represent the different types of Policy Rule Components that can be used in a PolicyRule. Notable subclasses of this class include PolicyEvent, PolicyCondition, and PolicyAction (which are used together to define an ECAPolicyRule). For example, goal policies can define their own subclasses under PolicyRuleComponentStructure; this enables them to interact with other types of policy rules. The ability to represent and model interactions between different types of policy rules is unique.

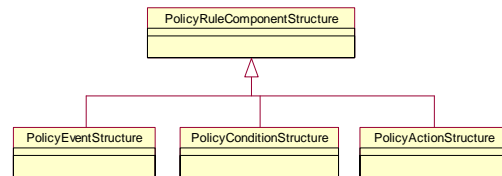


Figure 12. Improved PolicyRuleComponent Design

Similar to the above, we define the PolicyEventStructure, PolicyConditionStructure, and PolicyActionStructure classes as the superclasses of the PolicyEvent, PolicyCondition, and PolicyAction hierarchies. This enables us to use the same pattern to construct each of these hierarchies; while this pattern is very flexible, it can also accommodate extensions.

The pattern is shown in Figure 13 below. Each of the xxxStructure classes has two subclasses – an xxxGroup and an xxx class (so, for Figure 13, the PolicyEventStructure class has the PolicyEventGroup and the PolicyEvent subclasses). Each xxxGroup class has two aggregations, a recursive one for forming hierarchies of xxxGroups, and one that contains xxx classes. Each of the xxx classes has three subclasses, an xxxComposite to form containers of xxx instances, an xxxAtomic to represent stand-alone xxx instances that have “standard” attributes defined in this model, and an xxxNonStd, which is a generic extension mechanism for representing xxx instances that have not been modeled with the attributes specified in this model. For example, the PolicyEventNonStd class defines the eventEncoding and eventData properties for defining the format and content of a

vendor-specific event, and the eventResponse property for providing a standard result.

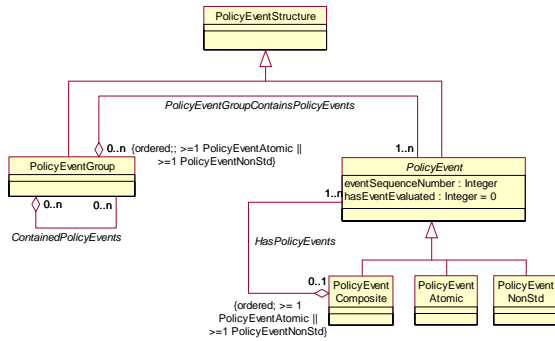


Figure 13. PolicyEventStructure Diagram

All important relationships that are not specific to one of these subclasses are defined to use the xxxStructure class. This enables the developer to add a new xxxStructure subclass that can either replace or augment the standard capabilities provided by the existing xxxStructure hierarchy.

Figure 14 shows how metadata is extended to work with all subclasses of PolicyRuleComponentStructure, just as it does for all subclasses of PolicyRuleStructure.

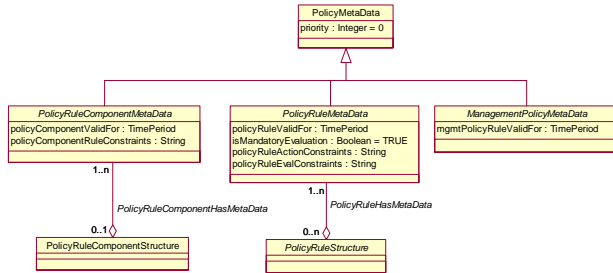


Figure 14. PolicyRuleComponent Metadata

The final part of the redesign of the policy model is the modeling of deontic policy rules. Figure 15 shows the insertion of a new class, PolicyCategory, as the parent of ManagementPolicy and child of PolicyConcept.

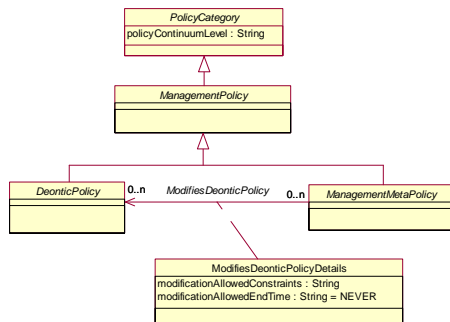


Figure 15. Making ManagementPolicy More Extensible

This is used to define how a Policy is used (e.g., what type of Policy this instance is). Important subclasses of PolicyCategory include management policies, application-specific policies, (e.g., backup, storage, query), and governance policies (management of policies). Two subclasses are defined: DeonticPolicy and

ManagementMetaPolicy. The former models obligation, permission, and related concepts, while the latter represents policies that describe how to use ManagementPolicies.

This version of DEN-ng defines four types of deontic policies – authorization (may), obligation (must), prohibition (may not), and exemption (need not), and two types of metapolicies (delegation and revocation). Each of these has two subclasses (not shown), one for a subject-based version and another for a target-based version. This provides a number of significant benefits that are not currently available in other policy models. First, the literature takes the simplistic view of defining subject-based obligation and target-based authorization (e.g., Ponder [24]). In addition to providing target-based obligation and subject-based authorization, this model also enables the definition of an authorization policy that is jointly dependent on its subject and target. Second, in a distributed system, the concepts of delegating functionality and revoking that functionality are very important. We explicitly represent delegation and revocation as separate concepts so that they can be related to deontic policy rules. However, there must be a PolicyRule in place for them to act on – hence, they are classified as metapolicies. Third, by explicitly differentiating between subject-, target-, and jointly-dependent policies, we make the semantics associated with different types of policy rules explicit, and hence easier to both define as well as detect. This considerably simplifies policy rule conflict detection and resolution. Along these lines, by formally defining subject- and target-based forms of deontic policies, we can more explicitly model complex orchestration that is needed in next generation and autonomic systems.

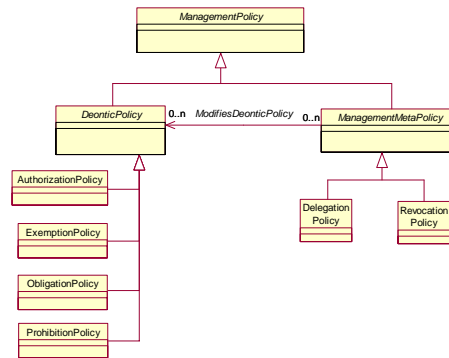


Figure 16. Realizing Deontic Policy Rules

Authorization embodies the concept of “is permitted to” or “is allowed to”. Deontic logicians assign the concept “may” to authorization. A subject-based authorization policy is a policy that is enforced by a subject, and defines the actions that a subject is permitted to perform on one or more targets. Conceptually, subject-based authorization policies are designed to define actions that can be performed on the target by the subject that will not adversely affect the subject. In contrast, target-based authorization policies are enforced by the target, and define the actions that a target permits a subject to perform on the target.

Prohibition defines the set of actions that an entity is forbidden to execute on another entity. Deontic logicians assign the concept “may not” to authorization. Hence, subject-based prohibition policies are enforced by the subject,

and define actions that the subject is forbidden to perform on a target, because performing such actions would jeopardize the subject. Target-based prohibition policies are enforced by the target, and define the actions that a target forbids a subject to execute on that target.

Obligation defines the set of actions that one entity must perform on another entity. Deontic logicians assign the concept “must” to authorization. Subject-based obligation policies are enforced by the subject, and define the set of actions that a subject must do on a target. Target-based obligation policies are enforced by the target, and define the set of actions that the subject must do on a target.

Exemption is, literally, immunity or release from an obligation. Deontic logicians assign the concept “need not” to authorization. Subject-based exemption policies are enforced by the subject, and define the set of actions that the subject need not perform on the target. Target-based exemption policies are enforced by the target, and define the set of actions that the subject need not perform on the target.

Delegation defines the ability for a sender to confer some function or privilege, to a receiver. Subject-based delegation policies are enforced by the subject, and apply a subject-based policy to a receiver. Similarly, a target-based delegation policy is enforced by the target, and applies a target-based policy to a receiver.

Revocation policies are used to retract functionality that was previously delegated. Subject-based revocation policies are enforced by the subject, and retract a subject-based policy from a receiver. Target-based revocation policies are enforced by the target, and retract a target-based policy from a receiver.

E. Associating Models with Ontologies

Earlier, we stated that UML lacks rich logic functionality to enable it to be used to represent behavioral semantics. If we can find a way to augment the facts represented by the information and data models, then we will be able to orchestrate behavior.

Our approach is to combine information models with ontologies. Conceptually, they form two graphs that are joined together as a single multi-graph, where facts in the information model serve as indices into concepts in the ontologies. This is a complex topic, and due to space limitations, will be deferred to a separate paper.

F. Building a New Policy Language

We are building a new policy language that will be derived from this policy model. It will consist of a number of dialects, where each dialect supports one or more layers of the Policy Continuum. Since this policy model can be used to generate ontologies, our new policy language will also contain syntactic and semantic links to concepts in our generated ontology. This is a complex topic, and due to space limitations, will be deferred to a separate paper.

V. CONTEXT AWARENESS

The DEN-ng context model is critical for ubiquitous computing, as it relates context changes to policy changes. It is shown in Figure 17 below. The *SelectsPolicies* aggregation defines a given set of Policies that should be loaded based on the current context. Hence, as context changes, policy can

change accordingly, enabling our system to adapt to changing demands. The *PolicyResultAffectsContext* association enables policy results to influence Context. For example, if a policy execution fails, not only did the desired state change not occur, but the context may have changed as well. The selected working set of Policies uses the *GovernsManagedEntityRoles* aggregation to define the appropriate roles of the ManagedEntities that are influenced by this Context; each *ManagedEntityRole* defines functionality of the ManagedEntity that can take on that role. In this way, policy indirectly (through the use of roles) controls the functionality of the system, again as a function of context. Both *ManagedEntityRoles* and *ManagementInfo* (management data describing the state of the ManagedEntity) are then linked to both Policy and Context by the four aggregations shown. Specifically, Policy is used to define which management information will be collected and examined; this management information affects policy decisions, as well as selecting which policies should be used at any given time. Once the management information is defined, then the two associations *MgmtInfoAltersContext* and *ContextDependsOnMgmtInfo* codify these dependencies (e.g., context defines the management information to monitor, and the values of these management data affect context, respectively).

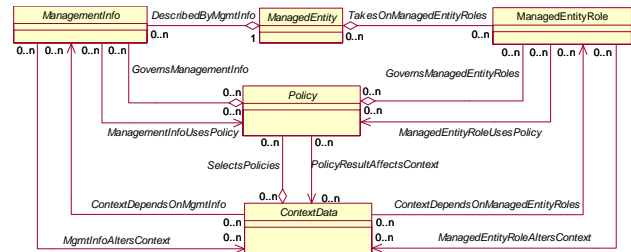


Figure 17. Simplified DEN-ng Context Model

VI. ADAPTIVE CONTROL ARCHITECTURE

Our autonomic architecture, FOCAL, has as one of its main goals the ability to adapt network services and resources to changing user needs, environmental conditions, and business goals. FOCAL accomplishes this goal through the use of multiple types of adaptive control loops (Figure 2 shows two for simplicity) in order to provide better, more flexible management. In network management, correcting a problem may involve more actions affecting more entities than the original managed entity in which the problem was noticed.

The use of two different control loops, one for maintenance operations and one for reconfiguration operations, is fundamental to overcoming the limitations of using a single static control loop having fixed functionality. Since FOCAL is designed to adapt its functionality as a function of context, the loop controlling the reconfiguration process must be able to have its functionality adapted to suit the vendor-specific needs of the different devices being adapted. For example, suppose that a particular type of management data is being collected. Referring to Figure 2, this will have the following effects:

- Set the sensors to retrieve a particular type of data
- Load appropriate data templates in the model-based translation function to enable disparate data management definitions to be analyzed
- Define a particular algorithm that uses pre-defined data structures to analyze sensed data
- Define a particular algorithm to enable these data to be analyzed, so that the current state of the managed entity to which the management data belong can be determined
- Define a particular algorithm to compare the computed current state to the desired state, and from that information, create an orchestrated set of changes to implement the state changes.

FOCALE performs these tasks by first, establishing the context; second, using that context to select a set of policies that govern the functionality of the system; third, using those policies (via the autonomic manager) to control each of the functions of the multiple control loops shown in Figure 2.

When context changes, the above ensures that the functionality of the appropriate control loops are changed to match the new nature of what is being governed. Similarly, if the system is performing a goal and needs to temporarily change its focus (e.g., examine sensor data from a different data source and correlate those data with the data that it is currently examining), the above processes can be used to again adapt the control loop.

Another important reason to use multiple control loops is to protect the set of business goals and objectives of the users as well as the network operators. The implementation of these objectives is different and sometimes in conflict – having a single control loop to protect these objectives is simply not feasible.

The reconfiguration process uses dynamic code generation based on models and ontologies [1][27][28][29]. Dynamic code generation is important, because the main goal of any autonomic system is to adapt to change while managing the associated complexity of doing so. Adaptation in turn requires the ability to reconfigure managed elements to perform different tasks.

The models are used to populate the state machines that in turn specify the operation of each entity that the autonomic system is governing. The management information that the autonomic system is monitoring signals any context changes, which in turn adjusts the set of policies that are being used to govern the system, which in turn supplies new information to the state machines. The goal of the reconfiguration process is specified by the state machines, which defines the (re)configuration commands required.

VII. RELATED WORK

In the original presentation of this paper, further questions were asked specifically about variants of CIM, such as Quartermaster [34], and specifically, why define another policy language instead of using either Ponder, Rei, or CIM-SPL. None of these approaches address 3 important requirements of autonomic systems: (1) multiple stakeholders and views embodied in the Policy Continuum concept, (2) the combination of models and ontologies, and (3) a common *lingua franca* that enables different management data to be

mapped to a common representation for further analysis. This section makes this claim again, and adds additional issues.

[34] takes a unique approach by proposing a model that enables resources to be composed into new resources by defining policies to govern how resources are specified and how they can be combined to build new resources. Each resource has an associated construction policy, which is “...modeled as an aggregate of one or more constraints that are defined using one or more attributes in policy.” This type of policy is itself completely different than other policies – it uses policy to define a constraint satisfaction problem, and hence, the policy itself does not have traditional condition-action pairs. It is impossible to use this language in a manner comparable to the DEN-ng approach, because its language elements and representation are completely different.

Rei is a policy language, defined using OWL-Lite, that allows policies to be specified as constraints over allowable and obligated actions on resources in the environment. The Rei engine reasons over Rei policies and domain knowledge in RDF and OWL to provide answers about the current permissions and obligations of an entity, which are used to guide the entity's behavior. Thus, Rei is limited to defining and performing deontic behavioral checks, and is *not* a general-purpose language like DEN-ng. Specifically, its lack of the use of an information model means that it cannot be used as described in this paper. In addition, the ontology that Rei uses are not formally defined. This makes it difficult to ascertain the intended semantics of Rei policy elements and indeed, of Rei policies themselves. A further problem is that the Rei ontology classes do not all have OWL restrictions; those that do are arguably under-specified. As a simple example, it is possible to define a Policy that contains no policy rules! Hence, Rei cannot be used in this approach.

CIM-SPL [35] is a policy language specification for the CIM environment. This causes two problems. First, since CIM is not an information model, and is not UML compliant, the associated semantics of CIM-SPL are not the same as that for DEN-ng. Second, the CIM model has a set of model flaws (such as the definition of no less than nine recursive associations at its root class – this means that (1) *all* CIM classes must inherit these associations, regardless of whether they are applicable or not, and (2) this is wrong from a classification point-of-view; as a simple example, three identity associations are defined – how can a chassis (as an example) make use of these?). More importantly, CIM-SPL (as CIM) does *not* define an event as part of its policy construct. This means that no conclusions or guarantees can be made about *when* a policy rule will be triggered. In addition, it does not meet the three criteria of autonomic computing (policy continuum support, use of models and ontologies, and a *lingua franca*).

VIII. SUMMARY

This paper has described a novel policy model that has been designed for next generation networked applications and services, such as those described in Motorola's Seamless Mobility scenarios. This new policy model is based on the original DEN-ng policy model, but changes it to make it more semantically accurate. This in turn enables it to be used

with other knowledge engineering tools, such as ontologies and machine learning, to reason about policies.

The FOCAL architecture is based on context-aware policy management. Changes in context change the set of policies that are being used; this in turn changes the allowed functionality that can be used in the system. Hence, changes in user needs and environmental conditions can be adjusted for by the FOCAL architecture.

Future work will concentrate on automatic ontology generation, including updating the ontology to conform to changes in the underlying information and policy models. We will build a new policy language that has the ability to include syntactic and semantic links to both information model objects as well as ontology concepts. We will then apply these to our FOCAL test bed and use them in various Seamless Mobility experiments to measure how useful ontologies and a new policy language can be.

ACKNOWLEDGMENTS

This research activity is part of continuing joint research between Motorola Labs and WIT. We'd like to acknowledge Greg Cox, Walter Johnson from Motorola Labs and Sven van der Meer, Brendan Jennings, Mícheál Ó Foghlú, and Willie Donnelly from WIT. In addition, this activity is partially co-funded by the Science Foundation Ireland (SFI) under the Autonomic Management of Communications Networks and Services programme (grant no. 04/IN3I404C).

REFERENCES

- [1] J. Strassner "Autonomic Networks and Systems: Theory and Practice", IM 2007 Tutorial, April 2006.
- [2] F. Ovesjö, E. Dahlman, T. Ojanperä, A. Toskala, A. Klein, "FRAMES Multiple Access Mode 2 – Wideband CDMA", PIMRC 1997
- [3] J. Mitola, "Cognitive Radio Architecture: The Engineering Foundations of Radio XML", Wiley-Interscience, ISBN 0471742449
- [4] <http://www.motorola.com/content.jsp?globalObjectId=6611-9309>
- [5] J. Strassner, "Policy Based Network Management", Morgan Kaufman, ISBN 1-55860-859-1
- [6] J.O. Kephart, and D.M. Chess, "The Vision of Autonomic Computing", IEEE Computer, January 2003. <http://research.ibm.com/autonomic/research/papers/>
- [7] IBM, "An Architectural Blueprint for Autonomic Computing", v7, June 2005
- [8] J. Strassner, E. Lehtihet, N. Agoulmine, "FOCALE – A Novel Autonomic Computing Architecture: extended version", accepted by ITSSA journal, 2007.
- [9] C. Prehofer, and C. Bettstetter, "Self-Organization in Communication Networks: Principles and Paradigms", IEEE Communication Magazine, July 2005.
- [10] <http://www.alphaworks.ibm.com/tech/pmac>
- [11] N. Damianou, A. Bandara, M. Sloman, E.C. Lupu "A Survey of Policy Specification Approaches", Department of Computing, Imperial College of Science Technology and Medicine, London, 2002.
- [12] <http://hillside.net/patterns/>
- [13] M. Fowler, "Analysis Patterns – Reusable Object Models", ISBN 0-201-89542-0
- [14] <http://citeseer.ist.psu.edu/355724.html>
- [15] J. Strassner, and J. Fu, "Policy-Based Enforcement of Ubiquitous Role-Based Access Control", paper accepted for Autonomic and Trusted Computing conference, 2007.
- [16] A. Wong, P. Ray, N. Parameswaran, J. Strassner, "Ontology mapping for the interoperability problem in network management", Journal on Selected Areas in Communications, Oct. 2005, Vol. 23, Issue 10, page(s): 2058- 2068.
- [17] J. Strassner, "Knowledge Management Issues for Autonomic Systems", TAKMA 2005 conference
- [18] J. Strassner, and B. Menich, "Philosophy and Methodology for Knowledge Discovery in Autonomic Computing Systems, PMKD 2005 Conference
- [19] <http://dl.alphaworks.ibm.com/technologies/pmac/acpl.pdf>
- [20] D. Ferraiolo, J. Barkley, D. Kuhn, "A Role-Based Access Control Model and Reference Implementation Within a Corporate Intranet", ACM Transactions on Information and System Security (TISSEC), Volume 2 , Issue 1, Feb. 1999, pages 34-64.
- [21] D. Ferraiolo, J. Cugini and D. Kuhn, "Role based access control: Features and motivations." In Proceedings of the 11th Annual Conference on Computer Security Applications. IEEE Computer Society Press, Los Alamitos, CA, 241-248, 1995
- [22] R. Sandhu, E. Coyne, H. Feinstein, C. Youman, "Role-Based Access Control Models", IEEE Computer 29(2): 38-47, IEEE Press, 1996.
- [23] R. Sandhu, D. Ferraiolo, D. Kuhn, "The NIST Model for Role Based Access Control: Towards a Unified Standard," Postscript PDF Proceedings, 5th ACM Workshop on Role Based Access Control, July 26-27, 2000 - first public draft of proposal for an RBAC standard.
- [24] N. Damianou, N. Dulay, E.C. Lupu, M. and Sloman, "The Ponder Policy Specification Language", LNCS Proceedings, IEEE 2nd International Workshop on Policies for Distributed Systems and Networks, 2001, pages 18-38
- [25] L. Kagal, T. Finin and A. Joshi, "A Policy Language for a Pervasive Computing Environment", Proceedings IEEE 4th International Workshop on Policies for Distributed Systems and Networks, June 2003
- [26] A. Toninelli, R. Montanari, L. Kagal, O. Lassila, "A Semantic Context-Aware Access Control Framework for Secure Collaborations in Pervasive Computing Environments", Proceedings 5th International Semantic Web Conference (ISWC), November 2006, pages 473-486
- [27] J. Strassner, D. Raymer, E. Lehtihet, S. Van der Meer, "End-to-end Model-Driven Policy Based Network Management", In: Policy 2006 Conference
- [28] J. Strassner and D. Raymer, "Implementing Next Generation Services Using Policy-Based Management and Autonomic Computing Principles", NOMS 2006
- [29] J. Strassner, "Seamless Mobility – A Compelling Blend of Ubiquitous Computing and Autonomic Computing", in Dagstuhl Workshop on Autonomic Networking, Jan 06
- [30] <http://www.tmf.org/browse.aspx?catID=1684>
- [31] http://www.dmtf.org/standards/cim/cim_schema_v214/
- [32] <http://www.omg.org/cgi-bin/doc?ptc/2004-10-05>
- [33] J. Kephart, and W. Walsh, "An AI Perspective on Autonomic Computing Policies", Policy 2004.
- [34] A. Sahai, V. Singhal, V. Machiraju, R. Joshi, "Automated Policy-based Resource Construction in Utility Computing Environments", NOMS 2004
- [35] DMTF, "CIM Simplified Policy Language", DSP0231, Version 1.0.0a, January 10, 2007