The Design of a New Context-Aware Policy Model for Autonomic Networking

John Strassner¹, Srinidhi Samudrala¹, Greg Cox¹, Yan Liu¹, Michael Jiang¹, Jing Zhang¹, Sven van der Meer², Míchéal Ó Foghlú², Willie Donnelly²

¹Motorola Labs, Schaumburg, IL, USA
{john.strassner, srini.samudrala, greg.cox, yanliu, michael.jiang, j.zhang}@motorola.com
²TSSG, Waterford Institute of Technology, Ireland
{vdmeer, mofghlu, wdonnelly}@tssg.org

Abstract – This paper describes a new version of the DEN-ng context model, and now this model in conjunction with the DEN-ng policy model can be used for more effective and flexible context management. Both are part of the FOCALE autonomic network architecture. Context selects policies, which select roles that can be used, which in turn define allowed functionality for that particular context.

Keywords: Autonomic Architecture, Autonomic Networking, Context, Context Management, FOCALE, Ontology-Based Management, Policy Management, Semantic Reasoning.

I. Introduction

This paper extends our previous policy paper [1] to more thoroughly discuss our evolving model of context, and to present a newer, more powerful context model. Network convergence (which combines different types of wired and wireless networks), cognitive networks [2] (in which the type of network access can be dynamically defined), and Seamless Mobility [3] (the ability for the user to get and use data independent of access mode, device, and media) all require a rich and extensible context definition. This is driven by their common need to provide active context-aware services, in which offered services may change when context changes.

One of the fundamental principles of autonomic computing [4] [5] [6] is to be able to detect changes in the environment, as well as needs of the user or business, and change the system functionality provided in accordance with business objectives. However, most autonomic architectures to date do not adequately address the subject and needs of context.

The FOCALE autonomic architecture [7] has laid a basic groundwork for linking policy to context to the functionality provided by a device, component or system. FOCALE generates code to reconfigure functionality based on the DEN-ng information model [8]. This paper extends [1] by extending the context model described in [1] in flexibility and functionality, and showing how context-aware policies meet the needs of cognitive networks and Seamless Mobility.

Our approach for implementing context-aware services is realized as the FOCALE autonomic architecture, 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 context 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 context model to the policy model of [1], so that policies can be written that adapt offered resources and services to sensed context changes. Finally, we outline how context-aware policies are used, together with machine learning and reasoning, to build a new adaptive control architecture. This is illustrated throughout this paper using examples.

The organization of this paper is as follows. Section 2 provides a brief introduction to autonomic computing and networking. Section 3 describes the design of the newly revised DEN-ng context model. Section 4 summarizes the DEN-ng policy model from [1] and links these two models together. Section 5 shows how context-aware policies meet the needs of cognitive networks and Seamless Mobility applications. Section 6 presents related work, and 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 [5] [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](image)

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.

Figure 2 shows a simplified version of our FOCALE autonomic architecture [7], which was designed to support this need as follows.

![Figure 2. Simplified FOCALE Autonomic Architecture](image)

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, which uses a combination of models and ontologies to translate disparate sensed data to a common *lingua franca*. 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.

The key to the FOCALE *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.

A system built in accordance with 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.

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 [1] [5] [8]. 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. However, the interaction between the Policy Continuum and context-aware policies is beyond the scope of this paper.

III. THE DESIGN OF THE DEN-ng CONTEXT MODEL

The DEN-ng context model is designed to serve a wide variety of uses. We have focused on a modular and extensible design – modularity enables developers to use only those parts of the model that they need, and extensibility provides convenient places to refine the model for application-specific needs. The first, and most basic, decision is to define the base classes that will represent context. One of the most popular definitions of context is: “Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves.” [9]

It is interesting to note Dey’s rationale for his definition, compared to previous definitions: “We cannot enumerate which aspects of all situations are important, as this will change from situation to situation. For example, in some cases, the physical environment may be important, while in others it may be completely immaterial.” [9] While we agree with this logic, the above definition has some significant shortcomings when it is applied to autonomic computing or autonomic networking. The most significant problems are

- “characterize the situation” is too vague. We need to differentiate between measured and inferred data, since they have different relevancies and confidence
This is because context can consist of multiple distinct sets of related data and knowledge; therefore, in order to represent context in a flexible manner as possible, we have adopted the approach of using two different classes to represent context. Context, is used to represent a completely assembled representation of context. ContextData is an auxiliary class that is only used when context contains multiple distinct types of different data that need to be combined in order to determine the overall context of an entity. For example, when modeling a phone call which can involve handover between different technologies, we instantiate two classes that is bound to a particular technology. This enables us to better manage the phone call, since the underlying technologies are themselves fundamentally different.

In FOCALCE, we implement the above notion in a reusable manner by providing a generic mechanism for associating detailed models and ontologies with content. Consider the concept of location. Instead of defining a “locatedAt” attribute in the Context class (which would necessarily have a fixed meaning associated with it), we can have the Context class reference a set of location classes that provide more detailed information and semantics for what location “means” to the Context of this particular ManagedEntity. This also allows the semantics of location to change as a function of context.

Figure 4 shows how this approach works in practice. First, we define a ContextData to represent the concept of location. We link the ContextData class to a set of location classes, which describe the location in a reusable way. The physical characteristics of the location are defined in this set of Location classes, while the semantics of the location are defined in the ContextData classes. Note that location is more than just latitude and longitude, but also includes height, proximity, co-location with other entities, and orientation. This genericity is why a set of classes (and appropriate units of measurement in 2-D or 3-D space) are required for defining location, as opposed to a simple attribute.
enables reusable context models to be created by attaching application-specific ContextData Domain Specific Ontology data to ContextData Upper Ontology information. This separation of common vs. domain-specific semantics reduces the burden of context processing for each individual application, and ensures simple re-purposing through its ability to dynamically connect or disconnect different ontologies to or from the upper ontology in order to meet the needs of resource-constrained devices, such as a cell phone.

This latter is an important design consideration. For example, when a user leaves his home to drive to work, our system can swap out the “home context” ontology with a “car context” ontology that reflects the different devices and their capabilities that the user now has access to. Similarly, when the user arrives at work, the “car context” ontology is swapped out with the “work context” ontology. Each context model is reusable, and from a processing point-of-view, each model only needs to be loaded when the context changes to make it relevant. This streamlined processing is critical for applications such as those embedded in cell phones, since they do not have a lot of freely available processing power.

Our Upper Ontology consists of five top-level types of information: ManagedEntity, Location, Time, Activity, and PersonOrGroup. Each of these information types is detailed in the DEN-ng information model. This is similar to [10]; notable differences are (1) the DEN-ng ManagedEntity class is more generic than the ComputationalEntity of [10], (2) the DEN-ng PersonOrGroup is more generic than the Person of [10], and (3) time is missing in [10]. While the first four types of ContextData can all be valid for a particular time, the purpose of the fifth (Time) class is to model information that is purely temporal in nature.

We provide a consistent and coherent set of knowledge through the use of the DEN-ng comprehensive information. This enables us to build a model-driven system, where changes to the model can be directly translated into changes in code. Thus, we are able to generate code to reconfigure the system in response to a context change. This is a fundamental advantage of our approach compared to other similar approaches, like that in [10].

DEN-ng is best viewed as a toolbox that contains the elements to model application-specific behavior using models [8]. The DEN-ng model can be viewed as containing sub-models of ManagedEntities, Context and ContextData that can be associated with each other to represent the behavior and characteristics that an application requires. ContextData typically consists of separate data found in separate portions of the DEN-ng hierarchy, such as location, identity and state of people, groups and computational and physical objects. Hence, a Context will in general aggregate multiple ContextData objects, as shown in Figure 6 below.

Both the Context and ContextData classes use the composite pattern for flexibility and extensibility. The ContextAtomic and ContextDataAtomic classes represent context that can be modeled as a single, stand-alone object. In contrast, the ContextComposite and ContextDataComposite classes represent context objects that are composite in nature (e.g., made up of multiple distinct Context or ContextData objects that can each be separately managed). This works because a ContextComposite class aggregates Context, and both ContextAtomic and ContextComposite inherit from Context. (This same line of reasoning applies to ContextData). Note one difference between Context and ContextData: the former has an additional relationship, called RelatedContexts. This recursive association enables different Contexts that are, in some way, related to this particular Context, to be defined.

The design shown in Figure 3 enables hierarchies of context information to be related to other hierarchies of context information. Each context node (a Context object) can have a set of ContextData objects that provide further detail describing the characteristics and behavior of that node. For example, the Context object “Communication” could have the following ContextData objects associated with it: PSTN (to model the characteristics of fixed telephone lines), CellularDevice (to model the characteristics of mobile phones and PDAs), ComputerDevice (to model the characteristics of laptops and desktops) and VisualAudioDevice (to model the characteristics of a television with Internet capability). Each of these four classes of device uses different types of media and provides different types of communication experiences. This model would be useful if the user had access to one or more of these devices at any particular time; in this case, the model defines how the user can communicate, along with modeling additional capabilities, such as when handover (e.g., between WiFi and cellular) is allowed.

The purpose of the ContextDataDetails association class is to define the particular semantics of how ContextData relates to Context. This enables different types of ContextData, each modeling a specific aspect of an overall Context, to be aggregated together with their own semantics.

The ContextDataFact and ContextDataInference classes shown in Figure 3 are used to represent additional data that is either known a priori or can be inferred from other knowledge about a given ContextData. For example, a given access point’s signal strength can vary over time – this can be observed by a sensor and a decision made as to whether the access point is stable enough to support mission-critical data.
communication over an encrypted link or not. Similar capabilities exist for the Context class – it has ContextFact and ContextInference subclasses. In both cases, these Fact and Inference classes complete the Context (or ContextData) model, and represent conclusions from examining the associated DEN-ng data attached to them (e.g., the location information attached to the ContextData class in Figure 4).

Figure 7 enhances the above basic model, in order to provide as much flexibility as possible. It represents a set of extensions that we will briefly describe in this paper (others exist, but are beyond the scope of this paper).

<table>
<thead>
<tr>
<th>Figure 7. DEN-ng Core Context Model Extensions</th>
</tr>
</thead>
</table>

The first thing we want to do is provide the ability to model additional semantics that are beyond the ability of UML to model. ContextSemantics represents data and/or knowledge that describes, but does not contribute to or impact, the behavioral aspects of the Context that this ManagedEntity is associated with. A similar class (ContextDataSemantics) is constructed for the ContextData hierarchy. The ContextSemantics and ContextDataSemantics classes represent a convenient point for fusing information from ontologies with data from information and data models. That is, in our implementation, these represent points where we can associate code generated from the model with code generated from the ontologies. The most obvious and important use of these code points is to enable detailed modeled data to be used to reason with. They also present convenient points for either augmenting context information (e.g., tagging it with metadata to enhance information retrieval) and/or using context data to perform (for example) a set of services. Finally, these two semantics classes enable the application to declare what it needs to complete its view of context, as opposed to merely obtaining context information. However, these latter examples are each complex processes and beyond the scope of this paper.

Identity enables the system to unambiguously refer to an object instance. The identifier has to be unique in the namespace that the object instance exists in.

Time is a common attribute of context. For example, time can determine when something is allowed to be used, or when use is no longer permitted. Since time can be handled differently for different types of contexts, we define two association classes (ContextDataAffectedByTimeDetails and ContextAffectedByTimeDetails) to represent the semantics of the corresponding associations ContextDataAffectedByTime and ContextAffectedByTime, respectively.

[8] defines a policy as “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.” In FOCAL, context is related to the state of an entity. In particular, changes in context trigger state transitions that adjust the behavior the entity in accordance with the changes in the environment that it exists in. Accordingly, both Context and ContextDetails can have a particular state; this enables the state of context information to influence which set of policy rules should be applicable or invoked.

State is complex, as it can vary from object type to object type. For example, for a person, it can refer to the activity that the person is currently doing, physiological factors such as whether the person is sick or tired, whether the person is busy or not, different physical conditions, and other factors. In contrast, the state of a network device includes its operational status, its power state, and other attributes that can be queried. Note that as context changes, the state associated with context changes, which causes policy to change. This is conceptually represented in Figure 8 below.

<table>
<thead>
<tr>
<th>Figure 8. Conceptual Context-Policy Relationship</th>
</tr>
</thead>
</table>

For example, imagine a user is switching between a business profile and an entertainment profile. The former uses a special service provider that offers encrypted, highly secure communication for business use, while the latter uses a completely different set of service providers (one for local and a different one for long distance calling) as well as links to social networks. In this situation, the policies defined as being usable for that context state will in general be different; some policies may be the same (e.g., rules about which devices can be used), some policies may be completely different (e.g., rules about which services can be used), and some policies may be modified (e.g., rules that govern communication).

[11] defines three types of context. Passive context is when an application presents new or updated context data to a user, but does not adapt its functionality or behavior as a result of context changes. Active context is when an application automatically adapts its functionality to discovered changes in context, by changing the application’s behavior. Spontaneous context is when an application must be able to cope with the uncertainty and vagueness of acquired context information. DEN-ng supports all three uses of context; the type of algorithms and amount of reasoning that must be done is a function of the specific needs of the application and, more importantly, whether the context is passive, active, or spontaneous. Figure 9 shows conceptually how context is used to affect policy.

The SelectsPoliciesToActivate aggregation defines a set of Policies that should be loaded and activated based on the current context. Hence, as context changes, policy can change accordingly, enabling our system to adapt to changing...
Note that this selection is an “intelligent decision”, in that the selection process depends on other components that are part of a particular context. Another “intelligent decision” is the PolicyResultAffectsContext association, which enables policy results to influence Context via the ContextControllerComponent, the application that manages 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.

![Figure 9. Conceptual DEN-ng Context-Aware Policy Model](image)

The selected working set of Policies uses the ContextAwarePolicyEnablesManagedEntityRoles association to define and enable the appropriate ManagedEntityRoles that are influenced by this Context; each ManagedEntityRole defines functionality that the ManagedEntity can use. In this way, policy indirectly (through the use of roles) controls the functionality of the system, again as a function of context. Similarly, ContextAwarePolicyEnablesMgmtInfo defines the set of management data that is useful for this Context; ManagementInfoAffectsContext represents feedback from these management data regarding the execution of the policy rule. Once the management information is defined, then the two associations MgmtInfoAffectsContext and ManagedEntityRoleAffectsContext codify these dependencies (e.g., context defines the management information to monitor, and the values of these management data affect context, respectively).

Finally, the ContextControllerComponent defines its own set of ManagedEntityRoles and ManagementInfo to use to monitor the environment; feedback from the ManagedEntities identified by their roles and specific measurements (in the form of ManagementInfo) are used by the ContextControllerComponent to operate its own finite state machine (FSM). This FSM is used to orchestrate the actions of the ContextControllerComponent, including which ManagedEntities it should determine the context for, how a particular element of context should be analyzed, and what procedures for determining its context should be used.

IV. DEN-NG CONTEXT-AWARE POLICY MODEL

The DEN-ng policies currently used in our project have followed a standard event-condition-action model. However, as we integrate machine learning and reasoning, we need to use other types of policy representations, such as goal policies and utility function policies [5] [12]. This necessitates a flexible policy hierarchy.

Another reason for using a flexible policy hierarchy is that in our prototype system, we currently use 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 high-level problems found. 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 do). 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.

Figure 10 shows the root of the new DEN-ng policy model. PolicyConcept 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 10. Root of the New DEN-ng Policy Model](image)

A PolicyDomain is a collection of entities and services that are administered using a common methodology. In FOCALE, this methodology involves using policies and finite state machines to orchestrate behavior. The HasPolicySubDomains aggregation enables a PolicyDomain to contain other PolicyDomains.

The PolicyContainedIn association defines the set of Policies that reside in a particular PolicyDomain. PolicyApplications are related to a given PolicyDomain through the ScopesPolicyApplication aggregation. When this is combined with role-based access control [13][14], 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 take on. Finally, the PolicyAppUsesPolicy association defines the set of Policies that are used by a particular PolicyApplication.

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 other managed entities, and is shown in Figure 11. A PolicyApplication controls the state of the system using policies. Control is implemented using a management model such as a finite state machine and, as a minimum, one or more entities to provide policy decision-making capabilities, as well as one or more entities to enforce and execute policies. It also includes entities to provide related functions, such as verify that the executed policies operated as expected, administer policies, coordinate management and usage aspects of policy as well as entities to enable policy components to scale to large distributed systems.

A ContextApplication provides Resources and Services for acquiring, assessing the relevance of, managing, updating,
and controlling the context of a set of entities. These functions are provided collectively or individually by one or more ContextApplication and ContextApplicationComponent subclasses. ContextApplications (and their respective ContextApplicationComponents) communicate with one or more PolicyApplications (and their respective PolicyApplicationComponents) in order to ensure that for a given change in context, the appropriate set of policies are activated (and deactivated).

Both PolicyApplications and ContextApplications (and their respective components) use the same pattern for realizing distributed management and communication. The following applies for both Context and Policy management; for brevity, only the former will be described. A ContextController represents both a set of core functionality for implementing context as well as a unit of distribution of context. It is either manufactured or constructed by integrating the functionality of different ContextControllerComponents. These are modular units of functionality that are used to construct context management applications. Large systems will employ multiple context management systems for various reasons, such as geographic distribution, separate management of different domains, and others. A ContextBroker is used to coordinate and control how different ContextEngines interact with each other. In this regard, it has two different functions. The first function is to ensure that conflicts between different context definitions don't exist when different Context Engines are asked to work together. The second is to coordinate the meaning and inference of different context information.

Context aware systems require many different types of Policy Rules. The DEN-ng model is unique, in that it models different types of policy rules. The PolicyRuleStructure class is the superclass of different types of policy rules; this paper concentrates on one type, known as an ECA (event-condition-action) policy, and is shown in Figure 12. The PolicyRuleStructure class uses 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.

The PolicyMetaData class defines metadata that applies to different types of Policies, such as (but not limited to) ECA-Policies. This decouples common metadata that different Policy representation systems need from the actual realization of the Policy, enabling both ECA- and non-ECA-Policies. Figure 13 shows the design of a ManagementPolicy. This policy integrates the representation of a policy rule with a PolicySubject and a PolicyTarget. For this paper, we are only concerned with ECA-PolicyRules.
that will be used to make management decisions. DEN-ng adds the notion of delegations as well. We separate the definition of deontic logic 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 and 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.

PolicyEvents, PolicyConditions, and PolicyActions all share a common design pattern, an exemplar of which is shown in Figure 15 below. Each of the xxxStructure classes has two subclasses – an xxxGroup and an xxx class (e.g., the PolicyEventStructure class has two subclasses: PolicyEventGroup and PolicyEvent). 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.

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

Figure 17 shows the insertion of a new class, PolicyCategory, for defining 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). DeonticPolicy models obligation, permission, and related concepts; ManagementMetaPolicy models policies that describe how to use ManagementPolicies.

We define four types of deontic policies – authorization (may), obligation (must), prohibition (may not), and exemption (need not), and two types of meta-policies (delegation and revocation), as shown in Figure 18. 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 and only defines subject-based obligation and target-based authorization (e.g., Ponder [15]). In addition to providing target-based obligation and subject-based authorization, our 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 and revoking functionality are very important. We explicitly represent delegation and revocation as separate concepts so that they can be related to deontic policy rules. 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.
Prohibition defines the set of actions that an entity is forbidden to execute on another entity. Subject-based prohibition policies are enforced by the subject, and define actions that the subject is forbidden to perform on a target. 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. 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. 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.

V. MEETING THE NEEDS OF CONTEXT AWARENESS

One of the main goals of our FOCALE autonomic architecture is to adapt network services and resources to changing user needs, environmental conditions, and business goals. FOCALE 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.

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 FOCALE 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.

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][6][17][18]. 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.

VI. RELATED WORK

There are three main information models being used today: DEN-ng [8], the TMF SID [19] and the DMTF CIM [20] (other efforts are too sparse in their domain coverage to meet the needs of autonomics). However, only DEN-ng has defined a context model, and the DEN-ng information model is more thorough and extensible than [19] and [20].

[10] describes the development of a software architecture to support the building of context-aware applications by defining a conceptual framework. While the framework simplifies the task of acquiring and delivering context to applications, it doesn’t provide an information model to visualize or define how context is used, nor does it describe how policy is used.

[21] defines a context service to let applications query and register for particular context information using a first-order logic to model context. This limits applications to using only context data from the context providers defined in the registry.

describing concepts for context-aware applications. However, it is designed to support context negotiation, not general purpose analysis, and doesn’t use information models at all.

There are several proposed extensions to the Composite Capabilities/Prefereces Profile (CC/PP) [22] and User Agent Profile (UAProf) [23] standards. Both of these mention context, but are in reality focused on describing device capabilities and user profiles. [24] is typical of these extensions – it extends the vocabulary of CC/PP, and provides some nice modeling of the processes of acquiring context, but doesn’t offer either a concrete model or ontology.

VII. SUMMARY

This paper has described a novel context-aware 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 model has been restructured to make it more amenable to both generating as well as integrating ontological information, and is based on ongoing experimentation using different applications that focus on (1) building reusable, extensible context models, (2) supporting dynamically changing context knowledge bases, and (3) supporting learning and reasoning about context awareness.

The FOCALE 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 FOCALE architecture.

We provide a consistent and coherent set of knowledge through the use of the DEN-ng information model. This enables us to build a model-driven system, where changes to the model can be directly translated into changes in code. These model-driven code points enable us to fuse information from ontologies with data from the information model by identifying points where we can associate code generated from the model with code generated from the ontologies. Thus, we are able to generate code to reconfigure the system in response to a context change.

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 FOCALE 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. 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