Towards Certifiable Autonomic Computing Systems Part I: A Consistent and Scalable System Design

Haffiz Shuaib and Richard John Anthony
Autonomics Research Group
School of Computing and Mathematical Sciences,The University of Greenwich.
Park Row, Greenwich, London SE10 9LS, UK
Email: haffiz.shuaib@yahoo.com, R.J.Anthony@gre.ac.uk

***Abstract*—Relative to currently deployed Information Technology (IT) systems, autonomic computing systems are expected to exhibit superior control/management behaviour and high adaptability, regardless of operational context. However, a means for measuring and certifying the self-management capabilities of these systems is lacking and as result, there is no way of assessing the trustworthiness of these systems. Two things are needed to begin to address the above. The first is a consistent structure for the autonomic computing system (ACS) and a consistent architecture for the autonomic computing manager (AM). The second is a set of metrics by which the operational characteristics of these systems are to be measured within the context of the targeted application domain.**

In this first part of a two-part paper, a biologically inspired architecture is proposed for the autonomic computing manager. The interfaces and messages by which this architecture communicates with objects within and those without are technically defined. Also discussed in this paper is the policy structure by which the autonomic manager is configured to sense contexts and effect changes in its managed environment. For the system framework, a tree structure together with its associated protocols is proposed, implemented and used as the basis for establishing administrative and security relationships between autonomic computing elements; for resolving management conflicts; for enforcing data integrity; for ensuring data availability and for providing mechanisms that aid system scalability, robustness and extensibility, while maintaining low system complexity. This framework is achieved using standards-based objects including the Lightweight Directory Access Protocol (LDAP), Policy Core Information Model (PCIM) and a significant number of Internet Engineering Task Force (IETF) Request For Comments (RFC) standard documents.

Keywords-Autonomic computing systems; Certification; Architecture; Intelligent Machine Design; LDAP;

I. Introduction

Information Technology (IT) systems are rapidly growing in complexity and are becoming more difficult to manage by the day. This growing complexity requires an increase in the number of expert human operators managing these systems. This in turn increases the cost associated with IT service management. Therefore, steadily replacing the human operator with machines that can carry out similar managerial functions is desirable. Apart from cost savings, this has the added benefit of allowing complex computing systems to evolve into even more complex ones with the associated value added service. The human operator will act as an overall guide to the system and should in no way constitute a technological bottleneck. However, such a computing platform must be verified and trusted before it is handed complex managerial duties. To accomplish this task, the internal components of the computing managers must be well understood, as well as their interactions with managed elements. The system of managers must be of low complexity, scalable, portable, secure and be able to efficiently and effectively accomplish the managerial tasks. Being able to assign a consistent measure of trust to these systems is also important. These are the challenges that need to be resolved by the autonomic computing research field.

Although this research field is about a decade old, the solutions to certifying autonomic computing systems (ACS), though urgently needed, have not been considered. Proposed solutions must tackle the certification problem from the twin angles of the architecture of the system and its component managers, as well as mechanisms for deriving quantitative and qualitative measures for the ACS. These solutions, when implemented and verified will lead to further acceptance of ACSs.

One of the difficulties associated with certification in this regard has to do with the inability to achieve a fair comparison between autonomic systems or elements from two or more vendors, as each may adopt a different system structure or element architecture. In order to address this difficulty, an architecture for autonomic systems and managers that enforces structure but is flexible is required. To that end, a three-layered architecture referred to as the Intelligent Machine Design (IMD) is co-opted and technically defined for autonomic computing managers (AMs). The general form of this architecture is based on observations of how humans or animals behave in terms of the way they perceive their immediate environment and effect changes as a result.

An autonomic computing system will typically consist of manager and managed elements. These elements must be able to co-exist and interact gracefully with one another within the system. However, versatile and standardized mechanisms that should aid proper management coordination within an autonomic computing system are nonexistent. Later in this paper, the requirements necessary for the above are identified, a system that relies solely on standardized protocols is proposed, and how this system meets each of the previously identified requirements is discussed.

This paper collates together the findings of a detailed research project whose roadmap can be found in [1] and more extensive details in [2].

The rest of this paper is organized as follows; In the next section, the state of the art as it relates to autonomic manager (AM) architecture is discussed. Also presented in this section, is an expression of the Intelligent Machine Design (IMD) architecture for AMs. Interfaces, event message types, valid configurations, policy object framework for the IMD are proposed and presented in Sections IV, V, VI and VII, respectively. A standard structure on which an ACS can be built upon is proposed and presented in Section IX. In Section VIII, the requirements for management coordination and efficient autonomic elements interactions in an ACS are set out. The solution to each of these requirements is presented in Section X. The conclusion follows in Section XI.

II. Autonomic Manager (AM) Architecture

An architectural standard is central to the process of the certification of an object. Any architecture that represents an autonomic

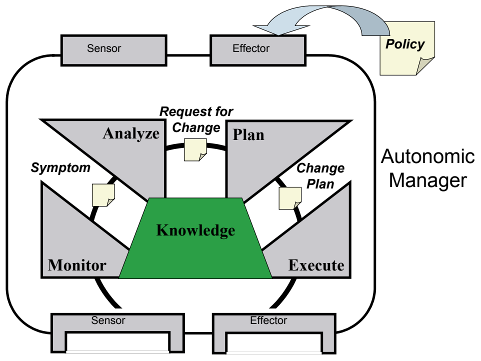

Fig. 1. IBM Autonomic MAPE Architecture [5]

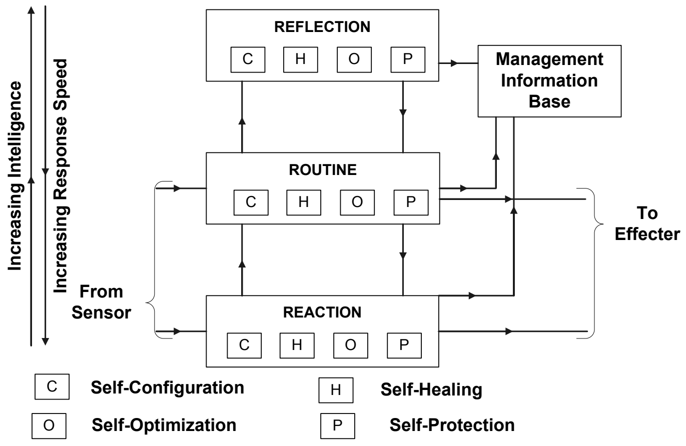

Fig. 2. An Autonomic Computing expression of the IMD

element e.g., an AM, must not be narrowly defined such that it precludes the ability for the element to evolve or cater for new use-cases. As noted in [3] and [4], the lack of an open standard is a challenge in the autonomic computing field. In this section, the most prevalent of all autonomic element architectures i.e., IBM's MAPE architecture is discussed. Drawbacks relating to this architecture are also discussed, and a certifiable alternative architecture with similar functionalities is presented.

A. *Related Work*

The IBM MAPE (Monitor, Analyze, Plan, Execute) architecture is a well known autonomic computing element architecture [5], and has been used as a reference for several autonomic computing systems. Systems that use the MAPE as a reference include; the web service host system proposed in [6], the self-adaptive service oriented system in [7] and the LOGO kit for data warehousing [8]. It has also been implemented in the Open Services Gateway initiative (OSGi) platform [9], applied to a Mobile Network Resource Management Architecture and several other projects [10][11].

The architecture consists of four main components which form a loop, as shown in Figure 1. The first of these components is the Monitor. Its main duty is to monitor the surrounding environment, including system resources. The output of this Monitor is used for making decisions at later stages of the loop. The second component i.e., the Analyze component, uses a number of algorithms to anticipate problems and possibly proffer solutions to these problems. The Planning component uses the information available to the autonomic system to choose which policies to execute. The Execution component, which is the fourth component, effects the most appropriate policy/policies chosen by the system. This executed policy may cause a change in the physical environment e.g., moving the arm of a robot, or simply pass instructions/information to another element, possibly an autonomic one. The input to the MAPE architecture comes from the sensory mechanism, while the effector mechanisms carry out the dictates of the machine.

While this architecture suffices for the purpose for which it was designed, it is ill-suited for certification purposes. This architecture has some limitations. For example, [12] considers it to be too narrowly defined to apply to some autonomic systems e.g., multi-agent systems. [13] points out that the loop in the MAPE architecture is vulnerable to failure, which in turn can precipitate the collapse of the management system all together. In addition to the above, there is no consensus as to whether the IBM MAPE architecture is a concrete architecture or a malleable concept. As a result, there are many different MAPE implementation permutations. A discussion of these divergent implementation views can be found in [1]. The current lack of a consistent architectural structure for the autonomic manager hampers the certification process.

Given that autonomic computing systems are biologically inspired, it follows that the manager architecture should also be similarly inspired, after all, AMs are supposed to steadily replace the human operator. This architecture must enforce structure without impeding innovation and it must allow for the separation of concerns i.e., the grouping of components with similar functionalities. This is the approach taken for the architecture presented in the next section.

B. *Intelligent Machine Design (IMD)*

The appeal of the Intelligent Machine Design (IMD) architecture [14] to autonomic computing systems is that it is closely related to the way intelligent biological systems work. The theory that underpins this architecture proceeds from the mechanisms by which animals and humans evaluate and effect changes in an environment, using their affect and cognitive abilities. Indeed, this architecture has been suggested as a generic framework on which, autonomic systems can be built upon [15]. While this architecture is mentioned in some autonomic computing literature, nothing concrete from a technical perspective has been achieved relative to IBM's architecture. Before describing this architecture in detail, it is necessary to specify what a Policy Rule is. A Policy Rule is the primary technical mechanism by which an AM effects changes in its managed environment given a specific context. A policy rule is made up of policy conditions, policy actions and other policy data that indicate how a policy condition is evaluated and how a policy action is to be executed. If the perceived state of the managed environment corresponds to the condition of a Policy rule, the AM executes the associated policy action accordingly. See Section VII for technical details on these Policy objects.

The IMD architecture is made up of three distinct layers i.e., Reaction (R1), Routine (R2) and the Reflection (R3) level (see Figure 2). Each layer is characterized by the following attributes; the amount of resources consumed, their ability to activate/inhibit the functionality of a connected layer and their ability to be activated or inhibited by another layer.

The lowest layer, the Reaction layer, is connected to the sensors (S) and effecters (E). When it receives a sensory stimulus, it responds relatively faster than the other two layers. The primary reason for this is that its internal mechanisms are simple, direct and hardwired i.e., it has an automatic response to incoming signals. Technically, the Reaction layer implements a single policy rule for all received input signals. However, if the input signal is such that this solitary policy rule does not suffice, control is handed over to the Routine layer (R2). The Reaction layer can also be inhibited/activated by the Routine layer. It consumes the least amount of resources.

The Routine (mid-level) layer is more learned and skilled when compared to the Reaction layer. It is expected to have access to the working memory or the Management Information Base (MIB), which contain a number of policy rules that are executed based on context, knowledge and self-awareness. As a result, it is comparatively slower than the Reaction level. Its activities can be activated or inhibited by the Reflection layer. Its input comes from both the sensory mechanism and the Reflection layer. Its output goes to the effecter mechanism and the Reflection layer. When the Routine level is unable to find a suitable policy rule for an immediate objective, due to ambiguities between two or more existing policy rules or the lack of a policy rule thereof, it hands control over to the human administrator or the Reflection layer.

While the Routine level's primary objective is to deal with expected situations whether learned or hardwired, the Reflection level, which is the highest level, helps the machine deal with deviations from the norm. The Reflection level is able to deal with abnormal situations, using a combination of learning technologies (e.g., Artificial Neural Networks, genetic algorithms), partial reasoning algorithms (e.g., Fuzzy Logic, Bayesian reasoning), the machine's knowledge base, context and self-awareness. Technically, the Reflection Layer's ultimate aim, as it relates to autonomic computing systems, is to create and validate new policies at runtime that will be used at the Routine level. If the system is able to adapt to an unexpected situation as a result of the new policy rule, then the rule is stored in the MIB. This new rule can be called upon if the situation is encountered in the future. Thus, making a formerly abnormal situation a routine one. The process of 'reasoning' out a new policy rule makes the Reflection layer the largest consumer of computing resources. This also means it has the slowest response time of all three layers. The Routine layer is the input source and output destination for the Reflection layer. The Reflection layer can inhibit/activate the processes of the Routine layer through new policy rule definitions.

C. The IMD and The Four Cardinal Self-Management Properties

Notice in Figure 2, that all three layers of the IMD are able to action each of the four 'cardinal' autonomic management properties (self- configuration, healing, optimisation and protection). To demonstrate how this works, consider an optimization scenario where limited resources must be allocated between competing requests. In this scenario, when the number of requests go beyond a certain threshold then the system is in danger of collapsing e.g., a sudden build up of service requests, leading to service queue overflow and thus violation of Service Level Agreements (SLAs). Assume that the number of requests currently being handled by the system is near that threshold. On sensing that the threshold is about to be reached, an autonomic manager (AM) implementing the IMD as its architecture engages its Reaction layer. The self-optimization component of the Reaction layer immediately forces the AM to stop any further allocation of resources to requests. There is little or no intelligence involved in this action. The Reaction layer then informs the Routine layer of this action. The Routine layer needs to effect an action such that the requests with higher priorities (based on the organizational goals) are met. To do this, the Routine layer looks to its policy rule database or MIB, to find the most appropriate rule whose condition fits the context. If an appropriate policy rule that optimizes the use of the limited resource is found, its associated action is executed. The execution of this policy rule's action overrides the lock placed on the managed system by the Reaction layer. Note that this Routine layer adopts a more intelligent and fine-grained approach to solving the optimization problem.

It may be that the Routine layer is unable to find a suitable policy rule in the policy repository for this specific context. In a case like this, control is handed over to the Reflection layer. Keep in mind that the lock implemented by the Reaction layer is still in effect at this time. The Reflection layer 'deliberates' on the best combination of requests that can be granted access to the managed resources, while still ensuring that the system is stable and organizational goals are met given the current context. The Reflection layer will be expected to implement a utility function or an artificial intelligence algorithm for this optimization process. As soon as a solution is computed a new policy rule is created and added to the policy repository and the Routine layer is informed of same. The Routine layer is now at liberty to effect the new policy rule. Again, as soon as the action associated with the new rule is executed, the previous lock placed by the Reaction layer on system resource allocation is removed. The same principle applies to the other three self-management properties.

Note that the terms 'the machine' and 'IMD' are synonymous and are used interchangeably through out this paper.

III. An Autonomic Application Example

An autonomic application example is used to illustrate and put into context some of the technical details presented in this work. For this purpose, an application called Path Finder (PF) in which robots are guided by AMs to and fro between a base and a target on a gridded map. The objective of this application is to have robots accomplish as many round trips as is possible between the base and the target within a considered time. The robots can be moved once on a clock tick in one of four directions on the map i.e., Top, Bottom, Left or Right square. To carry out this task, the AM must deduce, through its sensory mechanism (S), how many of the four squares constitute a valid next move for the robot, given its current position. Using its artificial intelligence algorithm, the AM decides which of the valid or available squares is best for the robot's next move. When the decision is made, the effecter mechanism (E) moves the robot, accordingly. In autonomic parlance, the robot is the *Managed Resource/Element* while the AM is the *Manager Element*.

IV. Machine Interfaces

Four distinct types of interfaces are proposed for the IMD in this work. These interfaces are shown in Figure 3 and are labeled *I-1* - *I-4*. Each interface is discussed in terms of the kind and structure of information it allows through.

A. The I-1 Interface

The first interface, *I-1* connects the Reaction and Routine layers to the sensory input (S) of the machine. Within this work, information that comes through the sensory interface i.e., *I-1* is referred to as a '*Context*'. While it would be expected that different autonomic applications would implement different *Context*s, it is necessary to describe this input information in a standardized way. The reason for this is that as long as an AM complies with the standard, it will always be able to interpret a *Context*, irrespective of the target application. The IETF standard, RFC 2252 [16] is the means by which the structure of a *Context* is described. RFC 2252 provides a standard basis for which attributes of different data types are defined. Multiple

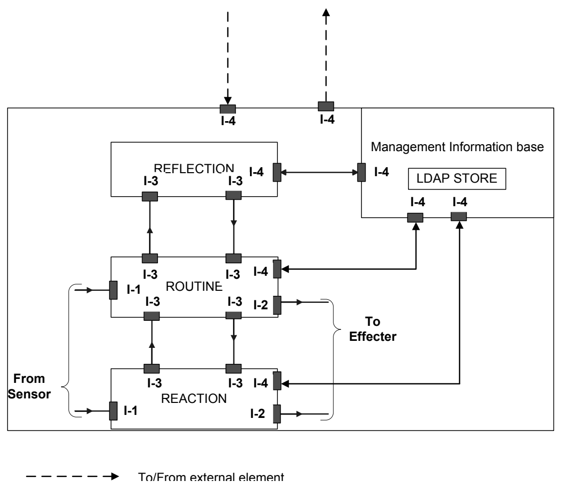

Fig. 3. Proposed interfaces for the IMD Architecture

```
objectclass ( 2.3.6.1.4.1.1.6863.6.1.909
NAME 'pfSensory' SUP Top STRUCTURAL
MUST (robotID $ topDirection $ bottomDirection $
rightDirection $ leftDirection) )
```

Fig. 4. RFC 2252 Compliant Object class for the PF *I-1*

attributes are grouped together under a structure called an object class. This RFC also mandates that all attributes and object classes are globally unique. This restriction ensures that no two applications will have the same *Context*. The object class in Figure 4 is an example of how the structure for the *Context* of the PF application described in Section III would look like.

In the figure, *2.3.6.1.4.1.1.6863.6.1.909* is a globally unique identifier for the object class called an Object ID (OID) and is assigned by the Internet Assigned Numbers Authority (IANA). Note that the OID in the figure above is fictitious. The name of the object class is *pfSensory* and it too must be globally unique. *SUP Top* means that the '*pfSensory*' class inherits the properties of another object class called *Top*. The *Top* object class is the parent class of all object classes defined in RFC 2252 and all sub-classes directly or indirectly inherit its properties. The *STRUCTURAL* Keyword simply means that *pfSensory* can be used as a stand-alone class. The *pfSensory* class consists of five mandatory attributes i.e., *robotID*, *topDirection*, *bottomDirection*, *rightDirection*, *leftDirection*. The *robotID* attribute is an integer value that contains the unique identifier of the robot. The other four attributes of this class i.e., *topDirection*, *bottomDirection*, *rightDirection* and *leftDirection* are Boolean values that are either True or False depending on whether a move to the Top, Bottom, Right or Left Square is valid, respectively. These attributes like the *pfSensory* object class must be defined in accordance with the rules set out in RFC 2252. The structure of these attributes will not be presented here. The interested reader should consult the RFC on how to go about this.

The *I-1* interface of AMs targeted at the *PF* application will only accept input information or *Context*s that are of the type *pfSensory*. With an information object class like this, multiple vendors can design AMs for this application without having to worry about compatibility issues. In addition, the contents of the *pfSensory* object class will have a globally consistent meaning, as it is compliant with the RFC 2252 standard.

```
objectclass ( 2.3.6.1.4.1.1.6863.6.1.910
NAME 'pfEffecter' SUP Top STRUCTURAL
MAY (topDirection $ bottomDirection
$ rightDirection $ leftDirection) )
```

Fig. 5. RFC 2252 Compliant Object class for the *I-2*

B. *The* I-2 *Interface*

Interface *I-2*, is used to instrument the physical environment or effect a change on the managed element. The instructions that lead to changes should be contained within the action code of the appropriate executing policy rule. The *I-2* interface on the other hand, sits between the AM and its effecter (E) mechanism, as shown in Figure 3. The information allowed on this interface must also be RFC 2252 compliant. An example of an object class for the I-2 interface for the *PF* is shown in Figure 5. This class consists of four optional boolean attributes viz; *topDirection*, *bottomDirection*, *rightDirection* and *leftDirection*. The ***MAY*** keyword in Figure 5 is what indicates that these attributes are optional. The AM creates an instance of this class and inserts the direction the robot is to be moved. This class instance is passed to the Effecter (E), which extracts the direction information and moves the robot in that direction accordingly. If more than one direction is specified in *pfEffecter* class instance, the effecter discards the information and the robot is not moved.

C. *The* I-3 *Interface*

The next interface, *I-3* is used for communication between layers of the IMD. Communication between layers is accomplished using a Machine Event Message (MEM). The exact technical details of the MEM are presented Section V. Recall from Section II-B, that a higher layer can modulate the response of a lower layer to an input stimulus or *Context*. This is accomplished through the *I-3* interface. Suffice it to say that this interface will only accept information that complies with the defined structure of the MEM.

D. *The* I-4 *Interface*

Before describing the *I-4* interface, it is instructive to briefly discuss the Lightweight Directory Access Protocol (LDAP) shown in Figure 3. In an autonomic computing system (ACS) and indeed any system, there is a need to have the ability to store and retrieve data information relating to management activities. Particularly, for ACSs, one needs to be able to store information relating to functional components within the autonomic domain e.g., managers, managed elements, policy objects, operational states of active elements, activity logs etc. All of the above will require a management information base (MIB). In this project, the Lightweight Directory Access Protocol (LDAP) defined in RFC 4510 [17] is the mechanism by which the MIB is realized. The LDAP is both a data storage/retrieval protocol as well as a communication protocol. As a data storage /retrieval protocol, it acts as a front-end to file storage systems that conform to the .X500 directory services. As a communication protocol, it runs atop the TCP/IP protocol stack. This provides an efficient, robust and secure link between any two autonomic elements.

The machine, therefore, uses its *I-4* interface to communicate with its LDAP compliant working memory or MIB and to communicate with external components, including a system-wide LDAP store, where available. As a result, only LDAP compliant operations are allowed on this interface. These operations are divided into three groups as delineated below;

1) **Interrogation operations**: Search and Compare.

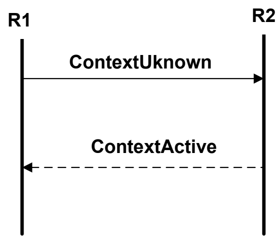

Fig. 6. Machine Event Message Exchange I

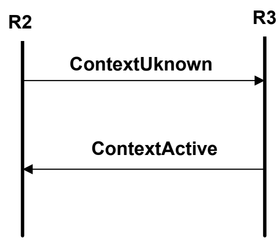

Fig. 7. Machine Event Message Exchange II

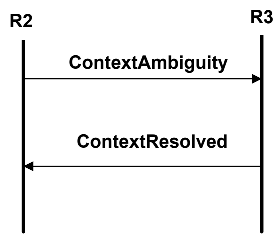

Fig. 8. Machine Event Message Exchange III

2) **Update operations**: Add, Delete, Modify and Rename.
3) **Authentication and control operations**: Bind, Unbind and Abandon.

Authentication and control operations are used to set up and tear down administrative and security relationships between autonomic elements. The retrieval of relevant information is based on the functionalities exposed by the *Interrogation operations*. Finally, the *Update operations* is used to carry out functions for which they are appropriately named.

V. Machine Event Messages

Messages exchanged between layers are called machine event messages (MEM) and are four-tuple objects that take the general form;

$$MEM = \begin{cases} eventType \\ eventID \\ policyRuleDNList \\ Context \end{cases}$$

The first component, *eventType*, of a MEM identifies what type of message a layer is signalling. Four types of message events are identified for this machine and they are;

$$eventType = \begin{cases} contextUknown \\ contextAmbiguity \\ contextActive \\ contextResolved \end{cases}$$

The purpose of each of these four event types are described later. The *eventID* is a unique integer identifier for the MEM and the '*policyRuleDNList*' is a list that contains the identities or Distinguished Names (*DNs*) of policy rules. *Context* is the information retrieved from the *I-1* interface (see Section IV-A). The valid message exchange process between layers of the machine is described using Figures 6, 7 and 8.

Recall from Section II-B that (1) when the Reaction Layer (R1) is unable to deal with an incoming *Context* or signal, it hands control over to the Routine layer (R2) and (2) that R2 can regulate R1's response to incoming *Context*. The process is accomplished using the message sequence chart shown in Figure 6. If the R1 layer can handle an incoming *Context*, it simply executes the action contained within its singular policy rule, otherwise, it creates an MEM. The event message will have a unique integer value inserted into the *eventID* field. The *eventType* of the MEM is set to *contextUnknown*. The incoming *Context* is inserted into the *Context* field of this MEM. The *policyRuleDNList* is left empty. R1 hands control over to R2 by sending this newly created MEM to the R2 layer through the connecting *I-3* interface (see Section IV-C). On receipt of the MEM, the R2 layer retrieves the *Context* and uses this to search for the most suitable policy rule from its policy repository i.e., LDAP store. If found, the dictates of the action associated with the policy rule is passed to the effecter for execution. Note that the R2 layer will reject a MEM from R1 that contains an eventType other than *contextUnknown*. In order to modulate the response of the R1 layer to incoming signals, the R2 layer replaces the policy rule currently active in the R1 layer. To do this, R2 creates an MEM with the *evenType* set to *contextActive* and the name of the new active policy rule is inserted into the *policyRuleDNList*. As soon as the R1 layer receives this event message, it replaces its current active rule with that contained in the received *policyRuleDNList*.

On receiving a *Context* directly from the Sensory input (S) through the *I-1* interface or from a MEM created by R1, R2 may be unable to find a policy rule in the repository that matches the received *Context*. As discussed in Section II-B, in order to resolve this problem R2 must engage R3. To this end, R2 creates a MEM with its *eventType* set to *contextUknown* and the context field set to the received *Context*. Let this newly created MEM be called MEM1. This event message is transmitted to R3 through the I-3 interface. R3, using its implemented artificial intelligence algorithm, will attempt to create a new policy rule for this unknown *Context*. If successful, the new policy rule is written to the repository through R3's *I-4* interface. A new MEM with *eventType* set to *contextActive* is created and the distinguished name (DN) of the new policy rule is added to the *policyRuleDNList*. Let this MEM be called MEM2. Since MEM2 is a response to MEM1, both will share a similar *eventID* field value. When R2 receives MEM2 , it checks the *eventID* field to make sure it is a response to a previously sent MEM. If it is not, MEM2 is discarded and no action is taken for that *Context*. If the *eventID* field is a match to the *eventID* of a previously sent MEM, the policy rule in the *policyRuleDNList* is extracted and its associated action executed by the effecter (E) for that *Context*. If this *Context* is encountered in the future, there will be no need for R2 to reengage R3, as a matching policy rule has already been created previously. The message sequence for the above is depicted in Figure 7.

Consider a scenario where R2 finds that two or more policy rules in the repository match a particular *Context*. This uncertainty must be resolved by R3 (see Section II-B), this interaction is illustrated in Figure 8. In an instance like this, R2 creates MEM3 with

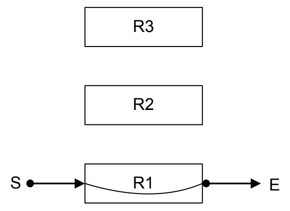

Fig. 9. Configuration I ($R1 \leftrightarrows \emptyset \leftrightarrows \emptyset$)

eventType set to *ContextAmbiguity* and inserts the *Context* into the Context field of MME3. The DN of all rules that match the *Context* are added to the *policyRuleDNList* of MEM3. When R3 receives MEM3, it attempts to resolve the policy rule conflict. If it does, it generates MEM4, sets the *eventType* to *ContextResolved* and inserts the *eventID* of MEM3 into that of MEM4. The preferred policy rule in MEM3's *policyRuleDNList* is added to the *policyRuleDNList* of MEM4. MEM4 is sent to R2. On receipt, R2 will attempt to match the eventID of MEM4 to that of MEM3. If they are not a match, MEM4 is discarded. Otherwise, R2 executes the action contained within the policy rule in MEM4's *policyRuleDNList*. The interactions shown in Figures 7 and 8 are the means by which R3 regulates the activities of R2.

VI. Machine Configurations

There may be instances where a targeted autonomic application domain can do without the functionality of any one of the three layers of the IMD, for example, the R1 layer may or may not be needed. In another application example, the R3 layer may not be needed, if the targeted application does not require an artificial intelligence algorithm to determine behaviour for new or unexpected situations. As a result, the structure of the IMD lends itself to several layer configurations, five to be precise, depending on the autonomic application. These five configurations are governed by two rules.

1) **Rule 1**: A configuration must have at least an R1 or R2 layer. Observe from Figure 2 that only the Reaction (R1) and Routine (R2) layers have access to the sensor and effecter mechanisms. These are the means by which an IMD-compliant AM perceives and effects changes on the managed system. Without at least one of these two layers, the AM is ineffective.

2) **Rule 2**: This rule has to do with the presence of the R3 layer. If the R3 layer is present, then the R2 layer must also be present. Recall from Sections II-B and V, that the R3 layer resolves conflicts if two or more policy rules match a particular *Context*. Only R2 is able to detect rule conflicts, as R1 only implements a single policy rule.

Based on these two rules, the five valid machine configurations of the IMD and their allowed event message sequences are presented in Sections VI-A-VI-E.

A. Machine Configuration I

In the first valid configuration (shown in Figure 9), R2 and R3 are not in commission, meaning that R1 is the only active layer, giving rise to the $R1 \leftrightarrows \emptyset \leftrightarrows \emptyset$ configuration. Where $\leftrightarrows$ represents the bidirectional *I-3* interfaces that connect the layers (see Figure 3). R1 receives input from the sensory object (S). This input forms the current *Context*. If the conditions associated with the singular policy rule in R1 match this *Context*, R1 passes the associated policy actions to the effecter object (E) for execution. If not, control is passed to the human operator of the application.

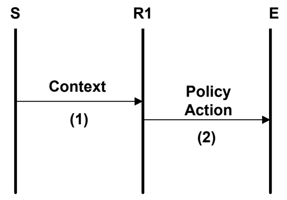

Fig. 10. Message Sequence Chart (Config. I)

The possible message sequence for the machine when the human operator is not involved is shown in Figure 10and represented by Expression (1).

$$(1) \mapsto (2) \tag{1}$$

The symbol $\mapsto$ in Expression (1) indicates the execution sequence from the sensing of a *Context* to the effecting of a change in the managed environment.

B. Machine Configuration II

Configuration II (Figure 11) i.e., $R1 \leftrightarrows R2 \leftrightarrows \emptyset$, assumes that the R3 layer is not needed for the targeted application domain. There are two possible message sequences for this configuration. These are shown in Expressions (2) and (3) and depicted in Figure 12.

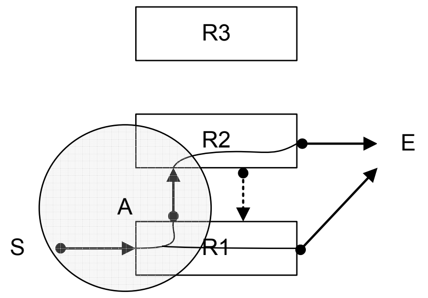

Fig. 11. Configuration II ($R1 \leftrightarrows R2 \leftrightarrows \emptyset$)

$$(1) \mapsto (2) \tag{2}$$

$$(1) \mapsto (3) \mapsto (4) \tag{3}$$

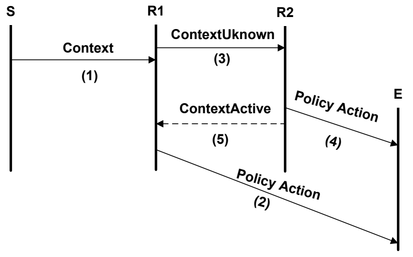

Fig. 12. Message Sequence Chart (Config. II)

Notice that Expression (2) corresponds to Expression (1) in Section VI-A. This indicates that Configuration II is an extension of Configuration I. If possible, the portion circled and labeled '**A**' in Figure 11 should be implemented as a single self-contained function. As is shown later, this function is reusable when a machine with this configuration needs to be extended. The dotted arrows shown in

Figures 11 and 12 represent instances where R2 changes the policy rule implemented in R1.

C. Machine Configuration III

All three layers of the machine in Configuration III are active as shown in Figure 13. This configuration allows for three different message sequence depending on the *Context* (see Expressions (4), (5), (6) and Figure 14).

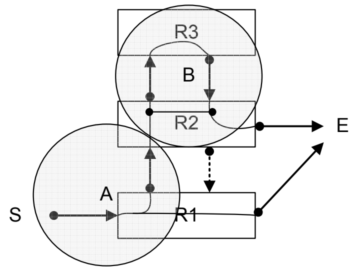

Fig. 13. Configuration III ($R1 \leftrightarrows R2 \leftrightarrows R3$)

$$(1) \mapsto (2) \tag{4}$$

$$(1) \mapsto (3) \mapsto (4) \tag{5}$$

$$(1) \mapsto (3) \mapsto (6) \mapsto (7) \mapsto (4) \tag{6}$$

Observe that the message sequence represented by Expressions (4) and (5) are also present in the Expressions of Configuration II. Again, this is a concrete expression of the fact that Configuration III simply an extension of Configuration II.

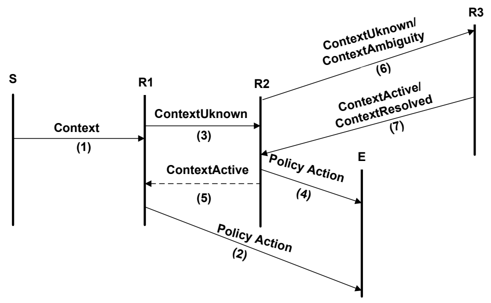

Fig. 14. Message Sequence Chart (Config. III)

Still on the theme of extensibility, notice that Figure 13 also has a portion circled '**A**', as did Figure 11 of Configuration II. If for any reason a machine with Configuration II needs to be extended to Configuration III, the self-contained function that implements the circled portion '**A**' of Figures 11 and 13 need not be rewritten, as it can be used as is. In a similar vain, the portion circled and labeled '**B**' in Figure 13 should also be implemented in a single self-contained function, as it can be reused when a need arises to transition from Configuration III to Configuration V.

D. Machine Configuration IV

In configuration IV, only the R2 layer is active as shown in Figure 15. This means that as soon as a *Context* is sensed, the matching policy rule is found and its associated policy action is passed to the Effecter (E). The only message sequence allowed for this configuration is straight forward as laid out in Figure 16 and Expression (7).

$$(3) \mapsto (4) \tag{7}$$

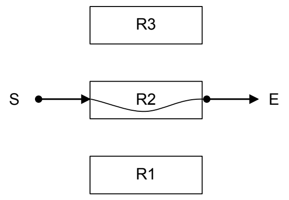

Fig. 15. Configuration IV ($\emptyset \leftrightarrows R2 \leftrightarrows \emptyset$)

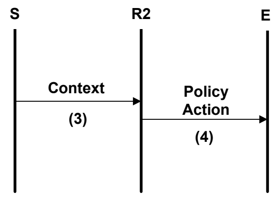

Fig. 16. Message Sequence Chart (Config. IV)

E. Machine Configuration V

From Figures 17 and 18 and Expressions (8) and (9), it is clear that Configuration IV is a subset of Configuration V.

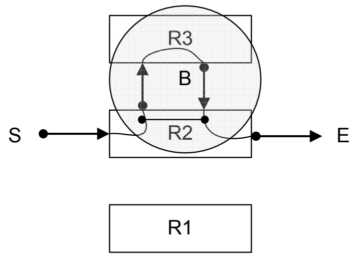

Fig. 17. Configuration V ($\emptyset \leftrightarrows R2 \leftrightarrows R3$)

$$(3) \mapsto (4) \tag{8}$$

$$(3) \mapsto (6) \mapsto (7) \mapsto (4) \tag{9}$$

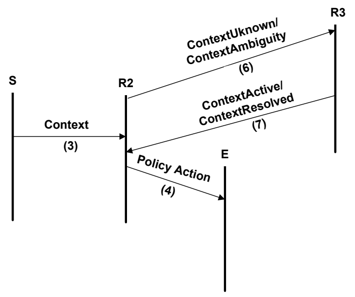

Fig. 18. Message Sequence Chart (Config. V)

This configuration is for application instances where the R1 layer may not be needed. Therefore, like Configuration IV, there is no need to attempt to change the policy rule implemented in R1. The circled portion labeled '**B**' in Figure 13 of Configuration III is also present in Figure 17. As discussed in Section VI-C, implementing the portion labeled '**B**' as a self-contained function eases the process of extensibility, if required at some future time.

VII. Machine Policy Object Framework

In Section II-B, policy rules, conditions and actions for the AM were introduced but without their proper technical structures. In this section, these policy objects are discussed from a technical perspective. The Policy Core Information Model (PCIM) in RFC 3060 [18] and its update defined in RFC 3460 [19] is the primary framework by which an IMD- compliant AM and by extension an ACS are able to process a received *Context*, select the best action as a result and aid some system management functions.

The main appeal of the PCIM framework is that it is well established, in that it is standardized and used in a number of computing management related fields, for example the management of computer networks. Apart from the fact that this framework outlines the structures by which policy objects are defined, it also enforces type safety, which allows for ease of parsing and automation. And this it does without restricting problem-specific applicability. The PCIM framework is able to achieve all of the above by defining the processes and associated schemas by which already defined policy object classes are to be encapsulated, extended or reused. The PCIM defines a number of policy objects but only four of these i.e., policy rules, policy conditions, policy actions and policy role collection are relevant to this work. Each of these policy objects implements an RFC 2252 compliant object class and consists of a number of similarly compliant object attributes. The object classes together with their associated attributes govern how specific policy objects are interpreted when read and/or executed. The classes and attributes of the four relevant policy objects are discussed further in the subsections that follow.

Note that this section is not meant to be an exhaustive discussion of the PCIM framework and its dependencies, as this information spans more than 10 Internet Engineering Task Force (IETF) RFCs or standards. However, it is detailed enough to support the core ideas discussed, leaving out extraneous information. In addition, when a policy object is introduced, the containing RFC is mentioned along side.

A. Policy Rule

A policy rule object is the means by which a condition or set of conditions is/are associated with an action or set of actions. According to RFC 3060/3460, it is not necessary for a policy rule to have an associated condition or action. However, in this work, all policy rules have conditions and corresponding actions. A policy rule is realized through the object class called *PolicyRule*. This class consists of 10 optional attributes. Only seven of these attributes are relevant to this work and they are discussed here. The first two attributes of the *PolicyRule* class to be dealt with are the *ConditionList* and the *ActionList*. The *ConditionList* contains the unique identities of the conditions that are to be evaluated when the policy rule is invoked. The *ActionList*, likewise, contains the unique identifiers of actions that would be executed if all the associated conditions evaluate to true. An attribute called the *ConditionListType* determines how the conditions of a policy rule are to be evaluated. This attribute specifies two types of condition evaluation procedures, namely; Disjunctive Normal Form (DNF) and Conjunctive Normal Form (CNF). In order to describe how the *DNF* and *CNF* apply to condition evaluation, it is necessary to state here that one or more conditions can be assembled under a single group number and a policy rule may be associated with more than one group of conditions. Assume that an instance of the policy rule class exists, such that it is made up of three groups of conditions. The first, i.e., Group 1 contains conditions C1 and C2, Group 2 contains C3 and C4 and Group 3 contains C5 and C6. If the *ConditionListType* is set to *DNF* (which is the default), the conditions are evaluated thus;

$$(C1 \text{ AND } C2) \text{ OR } (C3 \text{ AND } C4) \text{ OR } (C5 \text{ AND } C6)$$

If *ConditionListType* = *CNF*, then

$$(C1 \text{ OR } C2) \text{ AND } (C3 \text{ OR } C4) \text{ AND } (C5 \text{ OR } C6)$$

The *PolicyRule* class has an attribute called *Enabled*. This attribute can take one of three values i.e., *enabled*, *disabled* and *enabledForDebug*. If this attribute is set to *enabled* and if the associated conditions evaluate to true, the actions are executed. If it is set to *disabled*, the conditions are not evaluated and actions not executed. Lastly, if it is set to *enabledForDebug*, the conditions are evaluated but the actions are not executed.

pcimRuleSequencedActions attribute contains a list of integers that indicate the relative execution order of the policy actions associated with a *PolicyRule*. The values in this list are obtained from the *ActionOrder* attribute of the associated policy actions (see Section VII-C). The *Mandatory* attribute of the *PolicyRule* object class specifies the order, in which the policy actions associated with a policy rule are to be executed or interpreted. The allowed values for the *Mandatory* attribute are *mandatory*, *recommended* and *dontCare*. If *Mandatory* = *mandatory*, then the action order must be enforced, otherwise none of the actions should be executed. if *Mandatory* = *recommended*, the machine will attempt to execute the actions based on their order. If this fails, any other order may be attempted. If *Mandatory* = *dontCare*, the actions are executed in any order on the first try. In PCIM, managed elements can be grouped under a single named role. The name of this role can be added to a policy rule's *policyRoles* attribute. Every time this policy rule executes an action, the action impacts all managed elements pointed to by the content of its *policyRoles* attribute. The last attribute of the *PolicyRule* class discussed here is the *PolicyRuleName*. This attribute should ordinarily uniquely identify an instance of a policy rule object.

To store a defined policy rule instance in an LDAP store or MIB (see Section IV-D), the rule instance must follow the schema structure *pcelsRuleInstance* defined in RFCs 4104 [20] and 3703 [21]. These RFCs list this schema's globally unique identifier.

B. Policy Condition

A policy condition is defined by its object class called *PolicyCondition*. This is an abstract class that cannot be instantiated directly. It consists of four sub-classes, i.e., *PolicyTimePeriodCondition* , *SimplePolicyCondition*, *CompoundPolicyCondition* and *VendorPolicyCondition*. All of these sub-classes, save the last one are standardized. The *VendorPolicyCondition* is of most relevance to this work. This sub-class was created to allow for the definition of domain specific conditions that can be associated with policy rules. In other words, an instance of the ***PolicyCondition*** object can be applied to a vendor specific device through the *VendorPolicyCondition* sub-class.

According to RFC 4104 and 3703, creating a vendor specific condition and associating it with a policy rule is a four-step process.

1) First, the vendor must define the structure and interpretation of the input signal from the device. The defined structure of the signal is based on RFC 2252.
2) A schema for the vendor specific policy condition and its associated attributes must be defined according RFC 2252.
3) The defined vendor specific condition must then be coupled with an instance of what is called a policy rule association class.
4) The unique identifier of the instance of the association class is added to the policy rule's *ConditionList* attribute described in Section VII-A.

The PF application discussed in Section III is used to illustrate this four-step process. Step 1 has already been dealt with for the PF application in Section IV-A.

```
objectclass ( 2.3.6.1.4.1.1.6863.6.1.911
NAME 'pfCondition'
SUP pcimConditionVendorAuxClass AUXILIARY
MUST (isActive $ isValidMove $ topDirection
$ bottomDirection $ rightDirection $ leftDirection) )
```

Fig. 19. RFC 2252 Compliant PF Vendor Specific condition class

The second step in this process relates to defining the actual vendor policy condition. The schema for the vendor specific condition for the PF application is shown in Figure 19. From the figure it can be seen that the *pfCondition* class is derived from the *pcimCondition-VendorAuxClass* class, which is defined in RFC 3703. This vendor class has 7 attributes. The *isActive* attribute checks that the robot has been instantiated and is currently active. This attribute is always set to True. Attribute *isValidMove* verifies that the robot has not been moved on the current clock tick. Recall that a robot is moved at most once at the tick of the clock. The *isValidMove* attribute is also always set to True. The *topDirection*, *bottomDirection*, *rightDirection* and *leftDirection* attributes are similar to those discussed in Section IV-A. The *AUXILIARY* keyword indicates that the *pfCondition* class is not a stand-alone class and must be coupled with another class, which must be *STRUCTURAL* (see Section IV-A). In other words, its identity is drawn from the Structural class. This is the basis for Step 3 discussed later.

Assume that an AM maintains two variables for each robot i.e., *rActive* and *rMoved*. The *rActive* variable indicates the active state of the robot and the variable *rMoved* is either True or False depending on whether the robot has been moved in the current clock tick. Let *iSensory* be an instance of the *pfSensory* class (see Section IV-A) containing current information regarding a robot and its valid positions for the next move. An example condition evaluation code within an AM is shown in Figure 20.

```
isActive == rActive AND isValidMove == rMoved
AND (topDirection == iSensory.topDirection OR
bottomDirection == iSensory.bottomDirection OR
rightDirection == iSensory.rightDirection OR
leftDirection == iSensory. leftDirection)
```

Fig. 20. Condition evaluation code example

If the condition evaluates to True then all policy rules with matching conditions are equally applicable to the extant *Context*.

In the third step, the AUXILIARY *pfCondition* class must be coupled with a *STRUCTURAL* class called *pcelsConditionAssociation* class (as explained above). *pcelsConditionAssociation* is defined in RFC 4104 and its schema is shown in Figure 21.

```
objectclass ( 1.3.6.1.1.9.1.9
NAME 'pcelsConditionAssociation'
SUP pcimRuleConditionAssociation STRUCTURAL
MUST ( pcimConditionGroupNumber
$ pcimConditionNegated )
MAY ( pcimConditionName $ pcimConditionDN ) )
```

Fig. 21. RFC 2252 Compliant pcelsConditionAssociation class

The attribute *pcimConditionGroupNumber* is the group number to which the condition belongs. The *pcimConditionNegated* attribute indicates whether the condition should be negated before it is evaluated. A condition may have a condition name assigned to the attribute *pcimConditionName*, hence the use of the MAY keyword. Apart from the condition name, a DN or Distinguished Name may also identify the defined condition. The DN is assigned to the attribute *pcimConditionDN*. Coupling an *AUXILIARY* class to a *STRUC-TURAL* class is necessary because an *AUXILIARY* class cannot be instantiated directly. When a *STRUCTURAL* class is instantiated, the attached *AUXILIARY* class is also instantiated but indirectly. By coupling the defined *pfCondition* to the *pcelsConditionAssociation* class, the attributes of the former are included with the attributes of the latter. A machine reading this condition instance sees only the *pcelsConditionAssociation* class and not the *pfCondition*. However, due to coupling the machine is able to read the attributes of the *pfCondition* object class.

In the fourth and final step, the DN of the instance of the coupled *pcelsConditionAssociation* is added to the *ConditionList* attribute of an instance of a *PolicyRule* class.

C. Policy Action

The creation of a vendor-specific policy action object class follows the same four-step process used to create and associate a policy condition to a policy rule. The first step in this case, is to create an object class by which information going to the effecter through the *I-2* interface must be an instance of. This was done for the PF application in Section IV-B. In the next step, the PF vendor policy action *AUXILIARY* object class has to be specified. An example is shown in Figure 22.

```
objectclass ( 2.3.6.1.4.1.1.6863.6.1.912
NAME 'pfAction'
SUP pcimActionAuxClass AUXILIARY
MUST functionID )
```

Fig. 22. 2252 Compliant PF Vendor Specific action class

It consists of a compulsory solitary attribute called *functionID* and the class is derived from the *pcimActionAuxClass* defined in RFC 3703. This *functionID* attribute is of data type string and it points to the function that creates an instance of the *pfEffecter* object class defined in Section IV-B. Once created, the instance of the *pfEffecter* class is passed to the effecter (E), which then moves the robot in the indicated direction.

For the third step, the *AUXILIARY pfAction* class is attached to the *STRUCTURAL pcelsActionAssociation* defined in RFC 4104. The compulsory *pcimActionOrder* attribute shown in Figure 23 indicates the relative execution order of a policy action, in a policy rule

```
objectclass ( 1.3.6.1.1.6.1.10
NAME 'pcelsActionAssociation'
SUP pcimRuleActionAssociation STRUCTURAL
MUST ( pcimActionOrder )
MAY ( pcimActionName $ pcimActionDN ) )
```

Fig. 23. RFC 2252 Compliant pcelsActionAssociation class

that consists of more than one policy action. The optional attributes *pcimActionName* and *pcimActionDN* hold the name of the action and the action's Distinguished Name (DN), respectively.

In the final step, the DN of the coupled instance of the *pcelsActionAssociation* is added to the *ActionList* attribute of an instance of the policy rule object. When all the conditions associated with a policy rule evaluate to true, the associated policy action is retrieved and the content of its *functionID* attribute is passed to the effecter for execution.

D. Policy Role Collection

Unlike the other three policy objects discussed previously, the Policy Role Collection object is simply an administrative unit that groups a number of related managed elements on which a single rule is applicable. Note that a named Policy Role Collection instance is associated to a specific policy rule using the *policyRoles* attribute of that rule (see Section VII-A). In other words, the *policyRoles* attribute contains the name of an instance of a Role Collection. If the conditions associated with a policy rule evaluate to true, then the associated policy actions are applied to all managed elements pointed to by the *policyRoles* attribute of the rule. The Policy Role Collection object is schematically represented by the object class *pcelsRoleCollection* defined in RFC 4104.

VIII. Requirements for Management Coordination in ACSs

As a testament to the need for the ability to coordinate and manage autonomic element interactions, autonomic computing literature is replete with instances or scenarios where several autonomic elements must interact to achieve a common goal. In [22], the autonomic managers communicate indirectly with one another using the system variables repository. If a manager were to fail, other managers reading this repository take over the responsibilities of the failed one. Other research works take a more direct approach to autonomic manager interaction. In [23] and [24], the communication between managers is peer-to-peer, while [25], [26], [27], [28] and [29] adopt a hierarchical system for manager interactions. These works either lack a formal definition of the mechanisms by which these autonomic managers interact, or where defined, these mechanisms were highly specific to the system in question, thus preventing wide applicability and reusability.

Notwithstanding the lack of a formal framework that addresses issues relating to autonomic element interoperability, attempts have been made to specify certain requirements that should be met if interoperability is to be made possible. For example, [30] argues that the mechanisms that define interoperability between autonomic elements must be reusable to limit complexities i.e., it must be generic enough to capture all communications across the board. [3] mentions the need for a name service registry for autonomic elements, a system interaction broker and a negotiator as necessary components for autonomic element interaction. Also required is a need for standardized communication interfaces between autonomic elements to ensure interactions are well documented and secure [31], [12]. Based on some of the information contained in these works, the following eight requirements are proposed for effective management coordination and element interaction in ACSs;

1) **Administrative relationships**: A means to establish proper administrative relationships should exist. This way the sphere of influence of autonomic managers is clearly defined. This requirement is necessary to solve problems associated with operational conflicts. Also included within this requirement, is the need to define clear procedures for security relationships between elements in an ACS.
2) **Conflict Resolution Mechanism**: A conflict resolution mechanism must exist if two or more managers are able to simultaneously effect changes on the same resource.
3) **Monitoring Autonomic Elements**: A means must exist to query the internal state of an autonomic element. This is taken for granted when an AM might inquire as to the current state of an ME (e.g., start, stop and resume). Nevertheless, it may be necessary for an AM determine whether another AM is in a suitable operational state to allow for element interaction.
4) **Grant and Request Services**: For Requirement 3 to be possible, a mechanism for requesting and granting services must exist. For instance, an AM might need to understand the context in which a peer AM took an action. Requesting contextual information is within the remit of this requirement.
5) **Remote Policy Object Communication**: Following from Requirement 5, queries and associated responses must be transparent, regardless of the relative physical location of the AMs and MEs. In this case, an appropriate standardized communication protocol must exist to satisfy this requirement.
6) **Policy Object sharing**: If two or more AMs implement the same policy rule or if two or more MEs are instructed using the same policies, then an administrative mechanism (e.g., a well defined policy repository) for policy sharing should exist.
7) **A Policy Rule Selection Mechanism**: A structure to support the selection of the best policy for a given context should be available to a multi-policy system.
8) **Low complexity and Reusability**: Finally, the framework must be reusable across a broad spectrum of autonomic application domains without increasing its complexity.

The ways in which these requirements are met by the technical proposals in this work are presented in Section X.

IX. A System Architecture for ACSs

It is necessary to describe how the structure and build of an autonomic computing system (ACS) is approached in this work. The mechanisms discussed herein contribute to the solutions for the management coordination requirements set out in the last section.

In addition to the LDAP being the basis for an MIB (see Section IV-D), it is also the structure on which autonomic computing elements within an administrative domain are brought together to form an ACS. The mechanisms of the LDAP that provide for the core structure of the system, that ensure data integrity, security and availability are discussed here.

It may be noteworthy to state here that as the LDAP is an IETF standard, there are several implementations available. However, for this project, the *openLDAP* platform is the preferred choice. The reason for this is that in addition to being able to run on multiple operating systems, it is free, open source and implements a non-propriety license.

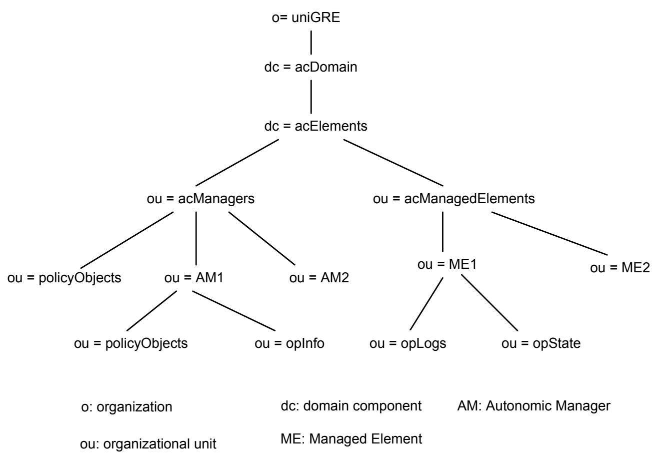

Fig. 24. Example DIT for an Autonomic Computing System

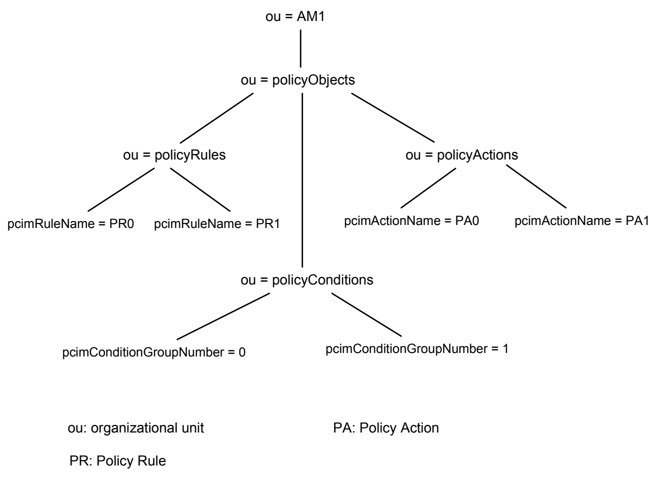

Fig. 25. AM1 Policy Object Branch

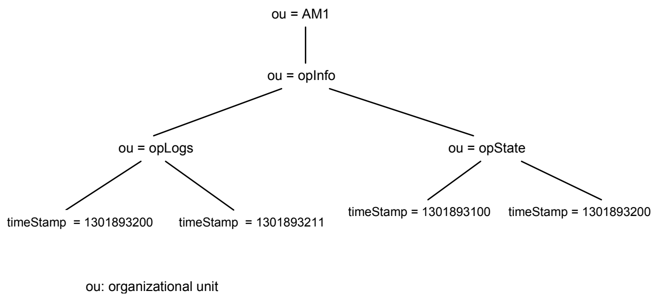

Fig. 26. AM1 Operational Information (opInfo) Branch

A. Core Structure

The LDAP stores data in a structure referred to as the Directory Information Tree (DIT). An example DIT implemented in this project is shown in Figure 24, with branches in Figures 25 and 26. At the root of the tree is *o=uniGre*. The *uniGre* (or University of Greenwich) component of this root entry is the name of the organization that owns the directory. The *o* component of this root entry is the name of the object class that defines the rules governing the naming of organizations in a DIT. This object class is defined in RFC 2256 [32]. Put more succinctly, the entry 'o=uniGre' means that the name of the organization is *uniGre* and this name conforms to the *o* object class.

At the next level of the DIT is the name of the domain within the organization i.e., *acDomain*, which stands for autonomic computing domain. The domain name conforms to the domain component (*dc*) object class defined in RFC 2247 [33]. The *acDomain* has a single branch called *acElements*. The *acElements* domain is split between autonomic computing managers and managed elements. These two branches conform to the organizational unit (*ou*) object class defined in RFC 2256. The *ou* object is a container that holds a number of other object classes. All managed devices in the organization are placed under the *acManagedElements* organizational unit. Each managed device e.g., ME1, ME2 contain a branch for storing operational logs (*opLogs*) and operational state (*opState*).

In a similar manner, all autonomic computing managers e.g., AM1, AM2 in the organization are placed under the *acManagers* branch of the *acElements* domain. From Figure 25, it can be seen that each manager has its own repository of policy objects that are applicable to the specific manager. A unit for storing operational information is also present.

As noted previously the *policyObjects* organizational unit consists of policy rules, conditions and actions. In Figure 25, under the policyRules branch are two rules named *PR0* and *PR1*. Each rule is identified by its *pcimRuleName* attribute. Recall from Section VII-A, that this is an attribute of the *pcelsRuleInstance* object class. Based on the above it can be inferred that all policy rules within the *policyObjects* branch of an autonomic manager must conform to the *pcelsRuleInstance* schema definition. Each policy condition under the *policyConditions* branch of Figure 25 must conform to the *pcimRuleConditionAssociation* object class definition, as its *pcimConditionGroupNumber* attribute is the basis for which policy conditions are identified (see Section VII-B). The same is true for the *policyAction* branch, only this time, any entry under this organizational unit must conform to the *pcimRuleActionAssociation* schema, as its *pcimActionName* attribute is the means by which policy actions are stored and accessed (see Section VII-C). Observe from Figure 24, that the *acManagers ou* also has a branch for policy objects i.e., policy rules, conditions and actions that have manager-wide applicability. In other words, all managers in the domain can share the policy objects under this branch.

Each manager also has a unit for entries relating to operational information (*opInfo*). In the DIT implemented in this project the *opInfo* consists of the state of the machine (opState) at points in time, as well as information relating to changes made to devices by the AM (*opLogs*). In Figure 26, the entries in operational logs (opLogs) and operational state (*opState*) organizational units are both identified by their individual UNIX time stamp (or Posix time).

Entries or information in a DIT are identified by their distinguished names or DN. For instance, the unique identifier for policy rule PR1 within the organizational structure is;

```
DN: pcimRuleName=PR1, ou=policyRules,
ou=policyObjects, ou=AM1, ou=acManagers,
dc=acElements, dc=acDomain
```

Fig. 27. Unique identifier for PR1

and the identifier for the operational state information of AM1 at (Posix) time 1301893100 is;

```
DN: timeStamp=1301893100, ou=opLogs,
ou=opInfo, ou=AM1, ou=acManagers,
dc=acElements, dc=acDomain
```

Fig. 28. Example identifier for AM opState information

B. Data Integrity and Security

In a directory that acts as a back-end for LDAP, data integrity is enforced through the twin mechanisms of object classes and attributes. Object classes are used to group attributes that apply to a specific component, and attributes contain data values for this component. The definition of the object class of a component indicates those attributes that are mandatory and those that are optional. The structures of the contained attributes in turn, specify the data types that values can take. For instance, values can be restricted to Integer, String or Boolean types etc., thus ensuring type safety. The syntax of attributes also indicates how attributes are to be compared during a search or compare operation. For example, an attribute that specifies a string value might indicate case sensitivity during a search etc. In addition, the attribute definition also indicates if it is multi-valued or single-valued. In Section VII, it was shown that object classes can include the attributes of other classes through inheritance or the use of auxiliary classes. This mechanism enables reusability and extensibility. As mentioned previously, all object classes and defined attributes must have globally unique identifiers assigned by IANA (see Section IV-A). All of these mechanisms help enforce entry integrity in the DIT. All related object classes and attributes are written to a schema file and the LDAP server is pointed to it. The syntax for object class and attribute definition are contained in RFC 2252.

Concerning data security, the LDAP allows for granting, restricting or denying access to any branch, attribute or distinguished name (DN) on a DIT using a user name/password mechanism. For instance, the domain controller of the DIT shown in Figure 24 might restrict access of one AM to a handful of managed element on the one hand, and grant full access to all managed element to another AM, on the other. In another scenario, the domain controller might give read access to its policy object entries to an AM but deny write access.

Apart from data integrity on the DIT, there is also a need to ensure integrity of the information exchanged between autonomic elements or between an autonomic element and an LDAP server located remotely. Recall from Section IV-D, that the interface between an autonomic element and its MIB or an external component i.e., *I-4*, only allows for LDAP-type transactions. Since LDAP uses TCP as its transport layer, it is able to ensure communication security by leveraging the Secure Sockets Layer (SSL) of the transport layer. If remote communication security is required, TCP port 636 is used, otherwise port 389 is used.

C. Data Availability through Referrals and Replication

LDAP allows branches of a DIT to be spread over several servers located at different physical locations. Regardless of the physical location of branches, the LDAP client e.g., an AM, still views the DIT as a consistent whole. The above is enabled by a mechanism known as Referrals. In a DIT where referral is implemented, rather than entries containing values, they would contain addresses to where the required data is housed. The object class for a referral entry is defined in RFC 3296 [34]. The LDAP client or server may resolve the referral. If the LDAP server is configured with *Chaining*, then the server gets the data from another server using the address contained in the instance of the referral object. Without *Chaining*, the referral is returned to the client and it is up to the client to issue a query based on the referral. Obviously, *Chaining* is the preferred method as it allows the whole process to be transparent to the client.

Whole copies of the DIT are allowed to be placed on multiple servers using a technique known as replication. Two types of replication configuration exist. The first, Master-Slave replication allows the Slave server to be updated by the Master. Client accessing the copy of the DIT on the Slave are only given read access. Write and update operations must be done on the Master server which then updates the Slave server after a defined period. If the server is configured without *Chaining* and a client attempts to write to the copy of the DIT on the Slave, the Slave server returns a referral for the Master server to the client. If configured with *Chaining*, the Slave server handles the update transparently. The second replication configuration is called Master-Master, which allows for reading, writing and updating on any of the LDAP servers. Changes to the DIT propagated to other servers later.

Both of these methods enable data availability, improve performance and ensure reliability. For instance, data can be placed closer to the consuming client through replication or referrals, thus reducing network overhead. In addition to the above, a backup of the DIT or branches of the DIT is always maintained through replication and referrals, respectively. Another benefit of these two mechanisms is that the DIT or parts thereof can be moved around possibly for scalability reasons without the need to change the client codes.

X. Mechanisms for Achieving Management Coordination in ACSs

The technical details of the IMD and ACS presented in Sections IV, V, VI, VII and IX provide the mechanisms by which the management coordination requirements set out in Section VIII are achieved. This section describes how each of these mechanisms or a combination of mechanisms is used to meet each requirement.

1) **Establishing administrative relationships:** The DIT structure shown in Figure 24 of Section IX-A is the basis for which administrative and security relationships are formed. In order to participate in an autonomic computing domain (*acDomain*), an AM must attempt to bind itself to the DIT of that domain. It does this by issuing an LDAP bind command through its I-4 interface to the *acDomain* (see Section IV-D). To place this joining request, the AM must have been configured with the right credentials i.e., username and password for the *acDomain* (see Section IX-B). If the bind request is successful, the *acDomain* creates a branch on the DIT for the new AM. Recall that the exact physical location of this new AM branch is irrelevant (see Section IX-C). As soon as the AM becomes aware of its new branch, it proceeds to set up its policy objects, operation and state information sub-branches. An ME is added to the domain much in the same way as an AM. All successful bind requests are recorded by the *acDomain*. This way, it is aware of all active objects within its sphere of influence. If an autonomic element no longer wishes to be part of the DIT, the element informs the domain controller of same.
2) **Resolving management conflict:** Management conflict can be resolved in two ways, once areas of potential conflicts have been identified. The first mechanism is known as hard resolution mechanism. Here, two or more AMs that may negatively interfere with one another are prevented from executing policy rules that point to the same *Policy Role Collection* Object (see Section VII-D). The soft resolution mechanism, which is the

second method, allows two or more AMs to use policy rules that point to the same *Policy Role Collection* Object but during periods where there is a risk of conflict, the ACS domain manager disables the policy rules. This is done by setting the *Enabled* attribute of the policy rule to *disabled* (see Section VII-A). Outside of the conflict risk period, the policy rule is enabled.

3) **Monitoring Autonomic Elements:** An autonomic element is able to persist current and previous state information in its *opState* branch on the DIT. It is also able to log its operational activities in the *opLogs* branch. *opLogs* and *opState* are depicted in Figure 26 for AM1 and in Figure 24 for ME1. If the state of an autonomic element needs to be verified, then it is simply a matter of querying its *opState* branch. This query will be based on the estimated entry time of the *opState* entry of interest. If the *acDomian* controller or a peer-AM requires information on why a managerial action was taken by an AM, a similar query with the time estimate is performed on the *opLogs* branch of the AM.
4) **Support for granting and requesting Services:** The *acDomain* controller supports and grants services to autonomic elements also using the DIT structure. For instance, assuming the proper administrative relationships have been established, an AM can query the *acDomian* for information relating to the available managed elements. Based on the retrieved information, the AM can then proceed to create its own policy objects for managing these elements. Requesting a bind to a DIT is also an example of support for services. Many other services specific to an *acDomain* can be defined, requested for and granted using the LDAP Interrogation, update and authentication and control operations (see Section IV-D).
5) **Reliable remote policy object communication:** Since LDAP relies on the TCP for network transport functionalities; an AM through its interfaces is able to reliably communicate policy actions or instructions to an ME and receive sensory information from the same ME. Keep in mind that TCP provides reliable ordered byte stream delivery to a network end device. SSL in TCP can also be used to provide security for autonomic elements when managerial transactions are carried out over a network (see Section IX-B).
6) **Policy object sharing:** An *acDomain* can define policy objects that are globally available to all AMs in its domain. For instance, in this project policy sharing is achieved by placing these common policy objects in the *policyObject* organizational unit of the *acManagers* branch (see Figure 24). Of course, the policy objects defined in the branch of an AM can be utilized by other AMs, if need be, assuming the right security relationships have been established. In the above example, one AM might be totally dependent on another AM 's policy object branch for its policy rules, conditions and actions. This may be a mechanism for enforcing hierarchy in a group of AMs. Recall from Sections VII-B and VII-C, that policy conditions and actions are associated with policy rules using their DNs. This mechanism allows two or more policy rules to reuse the same condition(s) or action(s), if necessary.
7) **A Policy rule Selection Mechanism:** The means to select the best policy rule for a particular *Context* is unique to the targeted application. Nevertheless, support for this exists in this work. Consider a scenario where two or more policy rules are applicable to a *Context*. An AM that conforms to the IMD architecture simply uses the *contextAmbiguity* and *contextResolved* message events defined in Section V to resolve the uncertainty.
8) **Low complexity and Reusability:** In this work, there are several levels of extensibility, which support low complexity and reusability. The reliance on standard based objects e.g., LDAP and its associated RFCs allow autonomic elements designed by different vendors to interact efficiently with one another, thus engendering low implementation complexity. If the *acDomain* becomes too large, it can be split into more manageable chunks without impacting on the structural integrity of the implemented DIT (see Section IX-C). This makes the *acDomian* scalable and by extension of low complexity. The security mechanisms utilized in this work are also well established. The policy object classes presented can also be extended in a structured manner to include attributes of vendor specific objects (see Sections VII-B and VII-C). In Section VI, suggestions were made regarding the implementation of functions that are self-contained and therefore reusable when attempts are made to extend the layer configuration of an AM.

XI. Conclusion

The technological reach of autonomic computing systems (ACS) is currently stymied by the lack of standardized certification procedures for these systems. This is made more difficult still by the lack of a consistent architecture for autonomic elements and systems and a deficit in standard metrics by which performances of these systems once built can be measured. In this paper, a structure based on already standardized protocols for the ACSs was proposed along with a flexible but consistent architecture for the autonomic manager (AM). Standard metrics are dealt with in the second part of this two-part paper.

Concerning the architecture for the AM, it was shown in this paper and elsewhere that due to implementation inconsistencies, the MAPE architecture was not well suited for certification purposes. As a result, an architecture that is flexible but guides the implementation more clearly than MAPE does was required. Biological animals, including humans use the same basic physiological structure to sense, process and effect changes in their immediate environment. Technically, this structure can be called a 'standard'. Incidentally, an architecture i.e., the Intelligent Machine Design (IMD) based on biological 'standard' had already been proposed. However, it lacked specific technical details to make it viable for ACSs or any other computing system for that matter. The first task in the process of expressing the three-layered IMD architecture for use in autonomic computing was to have it imbued with the four cardinal self-management properties i.e., self-configuration, self-healing, self-optimization and self-protecting. Each layer will implement these properties, albeit with differing levels of intelligence, computational complexities and execution speed, depending on both the architectural rules and on the application's requirements. The second task required the definition of four interfaces i.e., *I-1*, *I-2*, *I-3* and *I-4* for the IMD. Along with the definition of these interfaces, were the descriptions of the structure of the information communicated on each. Based on the make up of the IMD, five possible configurations, including their allowable message sequence charts were derived and presented. These configurations are one of the vehicles by which architectural flexibility is achieved. For the final task, the Policy Core Information Model (PCIM) framework was proposed as the basis for which policy rules, conditions, actions and repository are defined.

The Directory Information Tree (DIT) of the Lightweight Directory Access Protocol (LDAP) was proposed as the structure on which the

ACS is built. To support scalability in the system, branches of the DIT can be distributed over a number of physical locations and hardware, without affecting the consistency of the tree as viewed by elements. Apart from aiding scalability, distributing the branches of the tree also helps with data availability, as data can be transparently placed close to where it is consumed. This branch distribution is realized through a mechanism known as LDAP Referrals. Data availability can also be achieved by storing copies of whole or parts of the DIT in multiple locations. These copies are made consistent with the original using the LDAP server Replication mechanism. Replicating the DIT also achieves system robustness, as copies can be used as backups, if the main DIT becomes corrupted or unavailable.

Using the technical components of the DIT and the IMD, a number of issues relating to efficient management coordination and element interactions are resolved. These issues include but are not limited to; establishing security and administrative relationships, management conflict resolutions, autonomic element monitoring, support for extensibility and reusability across the system.

The proposals in this paper are foundational steps towards standardization of autonomic components, with a longer term goal of achieving certification of autonomic systems, which in turn is key to the long term acceptance and sustainability of the autonomic computing paradigm.

REFERENCES

[1] H. Shuaib, R. J. Anthony, and M. Pelc, “A framework for certifying autonomic computing systems,” *The Seventh International Conference on Autonomic and Autonomous Systems: ICAS 2011*, pp. 122–127, May 2011.

[2] Autonomic Research Group, University of Greenwich *http://cms1.gre.ac.uk/research/autonomics/tech.html. Latest Access: December 20th, 2012*.

[3] M. Salehie and L. Tahvildari, “Autonomic computing: Emerging trends and open problems,” *DEAS’05. Workshop on the Design and Evolution of Autonomic Application Software*, vol. 30, pp. 1–7, 2005.

[4] R. Sterritt, “Autonomic computing,” *Innovations System Software Engineering (2005), Springer-Verlag*, vol. 1, no. 1, pp. 79–88, 2005.

[5] IBM, “An architectural blueprint for autonomic computing,” *IBM Whitepaper*, June 2006.

[6] C. Reich, K. Bubendorfer, and R. Buyya, “An autonomic peer-to-peer architecture for hosting stateful web services,” *CCGRID ’08. 8th IEEE International Symposium on Cluster Computing and the Grid*, 2008.

[7] C. Dorn, D. Schall, and S. Dustdar, “A model and algorithm for self-adaptation in service-oriented systems,” *ECOWS ’09. Seventh IEEE European Conference on Web Services*, pp. 161 – 170, 2009.

[8] V. Nicolicin-Georgescu, H. B. R. Lehn, and V. Benatier, “An ontology-based autonomic system for improving data warehouses by cache allocation management,” *Knowledge and Experience Management Workshop*, September 2009.

[9] J. Ferreira, J. Leitao, and L. Rodrigues, “A-osgi: A framework to support the construction of autonomic osgi-based applications,” *Technical Report RT/33/2009*, May 2009.

[10] F. Mei, Y. Liu, H. Kang, and S. Zhang, “Policy-based autonomic mobile network resource management architecture,” *ISNNS 10. Proceedings of the Second International Symposium on Networking and Network Security*, April 2010.

[11] B. Pickering, S. Robert, S. Mnoret, and E. Mengusoglu, “Model-driven management of complex systems,” *MoDELS’08. Proceedings of the 3rd International Workshop on Models@Runtime , Toulouse, France*, pp. 277 – 286, October 2008.

[12] R. Quitadamo and F. Zambonelli, “Autonomic communication services: a new challenge for software agents,” *Journal of Autonomous Agents and Multi-Agent Systems*, vol. 17, no. 3, pp. 457–475, 2008.

[13] B. A. Caprarescu and D. Petcu, “A self-organizing feedback loop for autonomic computing,” *IEEE CS’09. In Proceedings of the Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, vol. 126–131, 2009.

[14] D. A. Norman, A. Ortony, and D. M. Russell, “Affect and machine design: Lessons for the development of autonomous machines,” *IBM Systems Journal*, vol. 42, no. 1, pp. 38 – 44, 2003.

[15] R. Sterritt, M. Parashar, H. Tianfield, and R. Unland, “A concise introduction to autonomic computing,” *Elsevier Journal on Advanced Engineering Informatics,*, pp. 181–187, 2005.

[16] M. Wahl, A. Coulbeck, T. Howes, S. Kille, Critical Angle Inc., Netscape Communications Corp., and Isode Limited, “Lightweight directory access protocol (v3): Attribute syntax definitions (RFC 2252),” December 1997.

[17] K. Zeilenga and OpenLDAP Foundation, “Lightweight directory access protocol (LDAP): Technical specification road map (RFC 4510),” June 2006.

[18] B. Moore, E. Ellesson, LongBoard-Inc., J. Strassner, A. Westerinen, and Cisco-Systems, “Policy core information model – version 1 specification (RFC 3060),” February 2001.

[19] B. Moore and IBM, “Policy core information model (PCIM) extensions (RFC 3460),” January 2003.

[20] M. Pana, MetaSolv, A. Reyes, Computer Architecture, UPC, A. Barba, D. Moron, Technical University of Catalonia, M.Brunner, and NEC., “Policy core extension lightweight directory access protocol schema (PCELS) (RFC 4104),” June 2005.

[21] J. Strassner, Intelliden Corporation, B. Moore, IBM Corporation, R. Moats, Lemur Networks, Inc., and E. Ellesson, “Policy core lightweight directory access protocol (LDAP) schema (RFC 3703).”

[22] M. Wang, N. Kandasamyt, A. Guezl, and M. Kam, “Adaptive performance control of computing systems via distributed cooperative control: Application to power management in computing clusters,” *ICAC ’06. IEEE International Conference on Autonomic Computing*, pp. 165–174, 2006.

[23] M. Zhao, J. Xu, and R. J. Figueiredo, “Towards autonomic grid data management with virtualized distributed file systems,” *ICAC ’06. IEEE International Conference on Autonomic Computing*, pp. 209–218, 2006.

[24] S. Ghanbari, G. Soundararajan, J. Chen, and C. Amza, “Adaptive learning of metric correlations for temperature-aware database provisioning,” *ICAC ’07. IEEE International Conference on Autonomic Computing*, 2007.

[25] B. Khargharia, S. Hariri, and M. S. Yousif, “Autonomic power and performance management for computing systems,” *ICAC ’06. IEEE International Conference on Computing*, pp. 145–154, 2006.

[26] J. Xu, M. Zhao, J. Fortes, and R. Carpenter, “On the use of fuzzy modeling in virtualized data center management,” *ICAC ’07. IEEE International Conference on Autonomic Computing*, 2007.

[27] R. Wang, D. M. Kusic, and N. Kandasamy, “A distributed control framework for performance management of virtualized computing environments,” *ICAC ’10. IEEE International Conference on Autonomic Computing*, pp. 89–98, 2010.

[28] M. Kutare, G. Eisenhauer, and C. Wang, “Monalytics: Online monitoring and analytics for managing large scale data centers,” *ICAC ’10. IEEE International Conference on Autonomic Computing*, pp. 141–150, 2010.

[29] X. Zhu, D. Young, B. J. Watson, Z. Wang, J. Rolia, S. Singhal, B. McKee, C. Hyser, D. Gmach, R. Gardner, T. Christian, and L. Cherkasova, “1000 islands: Integrated capacity and workload management for the next generation data center,” *ICAC ’08. IEEE International Conference on Autonomic Computing*, pp. 172–181, 2008.

[30] C. Kennedy, “Decentralised metacognition in context-aware autonomic systems: some key challenges,” *In American Institute of Aeronautics and Astronautics (AIAA) AAAI-10 Workshop on Metacognition for Robust Social Systems, Atlanta, Georgia*, 2010.

[31] S. R. White, J. E. Hanson, I. Whalley, D. M. Chess, , and J. O. Kephart, “An architectural approach to autonomic computing,” *ICAC ’04. IEEE International Conference on Autonomic Computing*, 2004.

[32] M. Wahl and Critical Angle Inc., “A summary of the x.500(96) user schema for use with ldapv3 (RFC 2256),” December 1997.

[33] S. Kille, Isode Ltd., M. Wahl, Critical Angle Inc., A. Grimstad, R. Huber, and S. Sataluri, “Using domains in ldap/x.500 distinguished names (RFC 2247),” January 1998.

[34] K. Zeilenga and OpenLDAP Foundation, “Named subordinate references in lightweight directory access protocol (LDAP) directories (RFC 3296),” July 2002.