An Artificial Intelligence Perspective on Autonomic Computing Policies

Jeffrey O. Kephart and William E. Walsh
IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

***Abstract*— We introduce a unified framework that interrelates three different types of policies that will be used in autonomic computing systems: *Action*, *Goal*, and *Utility Function* policies. Our policy framework is based on concepts from artificial intelligence such as states, actions, and rational agents. We show how the framework can be used to support the use of all three types of policies within a single autonomic component or system, and use the framework to discuss the relative merits of each type.**

I. Introduction

In order to cope with the ever increasing complexity of managing distributed, heterogeneous computing systems, it has been proposed that the systems themselves should manage their own behavior in accordance with *high-level objectives from human administrators* [1]. This vision of self-managing systems is sometimes referred to as *autonomic computing*[2]. Given these high-level objectives, autonomic computing systems will autonomously perform many of the tasks that are now beginning to overwhelm human administrators, including configuration, optimization, healing, and protection.

Numerous and varied definitions of policy (e.g., [3], [4], [5], [6], to cite just a few) have been put forward in recent years. Common to most of them is the notion that policies are a form of guidance used to determine decisions and actions. Given the numerous disciplines that are brought together under the umbrella of autonomic computing, any single formal definition of policy is likely to be too restrictive to cover all of the many forms of behavioral guidance that will be required. Instead, we prefer to adopt a very broad, loose definition of *policy* for autonomic computing: a *policy* is any type of formal behavioral guide. Clearly, then, policies as we define them have a fundamentally important role to play in autonomic computing.

In this paper, we approach autonomic computing policies from the perspective of the artificial intelligence (AI) field.[1] AI is an appropriate core discipline from which to borrow concepts and techniques for autonomic computing, as automated decision making is its central focus. More specifically, one popular view of the field of AI—outlined by Russell and Norvig in their standard text on AI [7]—is that it is fundamentally concerned with the design of *rational agents*. A rational agent is any entity that perceives and acts upon its environment, selecting actions that, on the basis of information from sensors and built-in knowledge, are expected to maximize the agent's objective. Since this is precisely the behavior we demand of autonomic components, it is essential that they be designed as rational agents.

Russell and Norvig outline a series of progressively more sophisticated and flexible approaches to rational agent design. We review three here. A *reflex agent* uses if-then *action rules* that specify exactly what to do under the current condition. In this case, rational behavior is essentially compiled in by the designer, or somehow pre-computed. A *goal-based agent* exhibits rationality to the degree to which it can effectively determine which actions to take to achieve specified goals, allowing it greater flexibility than a reflex agent. A *utility-based agent* is rational to the extent that it chooses the actions to maximize its utility function, which allows a finer distinction between the desirability of different states than do goals.

As we demonstrate in this paper, these three notions of agenthood can fruitfully be codified into three policy types for autonomic computing. The **Action policies** that form the basis of reflex agents will be essential for early autonomic systems, and will likely continue to prove useful in more advanced systems. However, higher-level **Goal policies** and **Utility Function policies** are necessary to realize the vision of truly self-managing systems.

For expository simplicity, Russell and Norvig describe these three types of policies (although they do not refer to them as such) as being embodied in different types of agents. However, a sophisticated agent in a complex environment such as an autonomic computing system may potentially use a mixture of these different policy types for various types of decisions. Moreover, in an autonomic computing system, there will be a multiplicity of autonomous agents serving a variety of purposes, each potentially relying on different types of policies. In order to support multiple policy types within a single autonomic component or system, we propose a unified framework that interrelates Action, Goal, and Utility Function policies. The framework also helps to clarify the relative advantages and disadvantages of different types of policy, such as their susceptibility to mutual conflicts, and the situations in which each is most appropriate.

It is beyond the scope of this paper to explore the full range of technical issues involved in creating and deploying policies in a real system. Rather, we outline the relationship between policies in the unified framework and, using an autonomic

[1] In AI, there is a more specific definition of policy, described later. Except where noted, we use our specified definition of policy.

Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'04)
0-7695-2141-X/04 $ 20.00 © 2004 IEEE

data center scenario, we highlight some of the complexities of applying the different policy types at multiple levels in the data center system. In Section II, we define these three types of policies and outline a unified framework into which they all fit, including the relationships among the three types. In Section III, we discuss a prototype system in which we have implemented these types of policies and describe how they are used in practice, focusing in particular on policies for system performance management and resource allocation. We conclude in Section IV.

II. Unified Framework

The unified framework we outline for autonomic computing policies is based upon the notions of states and actions that are quite familiar in the computer science literature, particularly in the realm of AI. In general, one can characterize a system, or a system component, as being in a state S at a given moment in time. Typically, the state S can be described as a vector of attributes, each of which is either measured directly by a sensor, or perhaps inferred or synthesized from lower-level sensor measurements. A policy will directly or indirectly cause an action a to be taken, the result of which is that the system or component will make a deterministic or probabilistic transition to a new state σ.[2] This sequence of events is depicted in Figure 1.

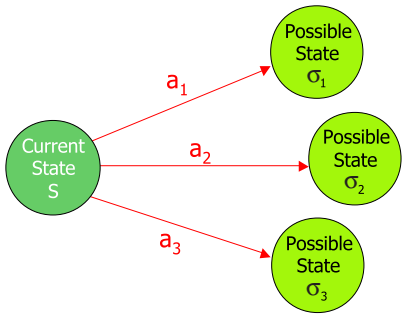

Fig. 1. States and actions: the basis for a unified framework for autonomic computing policy.

At least three types of policy will be useful for autonomic computing. We present them in order from the lowest to the highest level of behavioral specification.

- **Action Policies**. An Action policy dictates the action that should be taken whenever the system is in a given *current* state. Typically this takes the form of IF(*Condition*) THEN(*Action*), where *Condition* specifies either a specific state or a set of possible states that all satisfy the given *Condition*. Note that the state that will be reached by taking the given action is not specified explicitly. Presumably, the author knows which state will be reached upon taking the recommended action[3], and deems this state more desirable than states that would be reached via alternative actions. This is generally necessary to ensure that the system is exhibiting rational behavior. Related work on using Action policies for network and systems optimization includes [8], [9], [10], [11].
- **Goal Policies**. Rather than specifying exactly what to do in the *current* state S, Goal policies specify either a single *desired* state σ, or one or more criteria that characterize an entire set of desired states. Implicitly, any member of this set is equally acceptable—Goal policies cannot express fine distinctions in preference. The system is responsible for computing an action a (or possibly a sequence of actions or a workflow) that will cause the system to make a transition from the current state S to some desired state σ. Rather than relying on a human to explicitly encode rational behavior, as in Action policies, the system generates rational behavior itself from the Goal policy. This permits greater flexibility, and frees human policy-makers from the necessity of knowing low-level details of system function, at the cost of requiring reasonably sophisticated planning or modeling algorithms. Related work on using Goal policies for network and system optimization includes [12], [13], [14].
- **Utility Function Policies**. A Utility Function policy is an objective function that expresses the value of each possible state. Utility Function policies generalize Goal policies. Instead of performing a binary classification into desirable vs. undesirable states, they ascribe a real-valued scalar desirability to each state. Because the most desired state is not specified in advance, it is computed on a recurrent basis by selecting from the present collection of feasible states the one that has the highest utility. Utility Function policies provide more fine-grained and flexible specification of behavior than Goal and Action policies. In situations in which multiple Goal policies would *conflict* (i.e. they could not be simultaneously achieved), Utility Function Policies allow for unambiguous, rational decision making by specifying the appropriate tradeoff. On the other hand, Utility Functions policies can require policy authors to specify a multi-dimensional set of preferences, which may be difficult to elicit, and furthermore they require the use of modeling, optimization, and possibly other algorithms. Related work on using Utility Function policies for network and systems optimization includes [15], [16], [17], [18], [19], [20], [21].

It is instructive to compare our broad definition of policy to the standard definition used in AI, namely that a policy specifies a mapping from any state to the action that should be taken in that state [7]. In our terms, this notion of policy is a *set* of Action policies, in which the Conditions fully cover the state space, and in which each state is mapped to a unique action. Implicit in their definition, there are no policy conflicts, and the policy is a single coherent, consistent mapping from state to action. In order for an autonomic system or component to exhibit rational behavior, an Action *policy set* must cover

[2]For the sake of simplicity, we implicitly assume *deterministic* transitions in most of this paper.

[3]Or the probability distribution of states that could be reached, in the case of probabilistic state transitions.

every state in the relevant state space and provide a single unique action for each. Unfortunately, when even moderately large sets of Action policies are created manually by people, it is quite difficult to ensure that the resulting policy set satisfies this criterion. Recognizing then the fallibility of humans in specifying Action policies, we need additional mechanisms to help reduce the potential for conflicts, and to handle them when they do arise. We discuss this further in the next section.

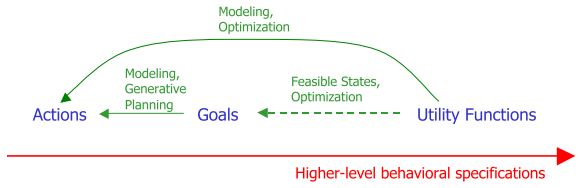

Fig. 2. Relationships between different types of policy.

Alternatively, coherent action can be automatically derived from the higher level forms of policy. Figure 2 illustrates the relationships between the three types of policy in a unified framework. Goal policies are translated into actions during system operation by any of a variety of methods including generative planning [7, chapter 11] for example. To generate a sequence of actions that achieves the desired goal, generative planning necessarily takes into account the results of performing an action. This in turn requires that the system have a model of itself that indicates how actions change the state of the system. In simple cases, translation of Goal policies into actions can be achieved purely via modeling without any planning, as will be illustrated in the next section.

Implementing Utility Function policies requires optimization algorithms. Because Utility Functions are a function of states, it might appear easy and natural to use optimization to directly identify the most desirable state as a Goal from which actions can then be derived via planning and/or modeling. However, one needs to identify the most valuable *feasible* state, and in general that identification requires system modeling. A likely byproduct of such modeling is knowledge of the low-level actions that result in a given state. Thus, once the most optimal state is identified, it is likely that the actions required to achieve that state are already known, so the intermediate step of establishing a goal state can be bypassed. Furthermore, deriving actions directly from the Utility Function policy allows the optimizer to take into account the potential costs of actions. Costs could include explicit monetary costs (such as the cost of power for running a server or the cost of obtaining computation from a data center), or opportunity costs (the cost, not explicitly modeled in the utility function, of not doing something else of value).

In many dynamic autonomic computing scenarios Utility Function policies would be optimized online to compute the best actions for the current state. There can also be situations in which it is feasible to use the utility function to compute an entire Action policy set offline, which is then interpreted by the running system. This approach is taken in the decision-theoretic planning literature by using Markov Decision Processes [22] to compute the actions that should be taken *every* state such that the optimal expected sum of (discounted) utility is obtained over the sequence of all states visited. A hybrid approach can also be taken, whereby the policy set is periodically recomputed online when conditions change sufficiently to make the old policy set suboptimal.

It is worth noting that Utility Function policies can be properly viewed as generalizations of Goal policies. Indeed, conceptually, a utility function can be defined by specifying a complete set of disjoint goals and assigning values to them. (This is not generally feasible when there is a large state space, and is impossible if the state space is continuous. In these cases, more compact functional expressions of utility functions are more practical.) On the other hand, although Action policies are computed by optimizing Utility Function policies, there is no meaningful sense in which Utility Function policies can be derived from Action policies because Action policies are defined over the *current* state space and Utility Function policies are defined over the *desired* state space.

Although not shown, there may also be self-loops in Fig. 2. For example, Utility Functions may be translated into other forms of Utility Functions for use in multiple levels of decision making. In this case, different Utility Functions correspond to the same value system translated into different state spaces. For instance, a Utility Function at one level could specify the relative value of different service levels—to be used for optimizing the performance of a stream of transactions in one part of the system—while a Utility Function at another level could specify the value for obtaining different amounts of computational resources—to be used for optimally allocating resources throughout the system. Generally, to derive some Utility Function B from Utility Function A, a procedure must compute, for each state in the space of B, the optimal value that could be obtained in the space of B. This requires optimization algorithms and a model of how available actions can transform the state space of B to the state space of A. The next section will demonstrate via a simple example how a service-level utility function by one autonomic component can be transformed into a resource-level function to be used by another autonomic component. One can easily imagine other examples in which Goal policies at one level can be transformed into Goal policies at another level.

III. Data Center Scenario

The previous section introduced and discussed (somewhat abstractly) three types of policies and a framework that encompassed them. In this section, we shall illustrate and compare the three types of policy more concretely via a data center scenario that is based on a prototype that we and our colleagues at IBM Research have implemented [23], [21].

As shown in Figure 3, the model data center is comprised of a number of *Application Environments* that provide application services to customers. Application Environments are logically separated from each other, each providing a distinct application

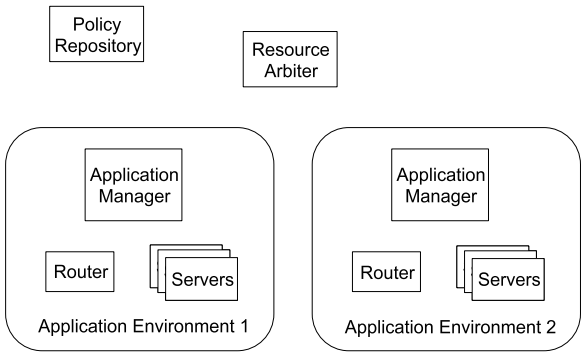

Fig. 3. A data center model.

service using a dedicated (but dynamically allocated) pool of resources. For simplicity of exposition, we assume that resources are identical servers. Each Application Environment is managed by an *Application Manager* (AM), which is responsible for implementing the policies obtained from a *Policy Repository*.

Each AM continually adjusts resource usage in accordance with its policies. When demand changes significantly, some AMs may not be able to adequately implement their policies with available resources. When this occurs, they can appeal to the *Resource Arbiter* for additional resources. The *Resource Arbiter* will consider such resource requests, and may under some circumstances even remove resources from one AM to give them to another if it deems this desirable.

Next, we illustrate in greater detail how policies of each type might drive the resource allocation behavior of the AMs and the Resource Arbiter, using this as a basis for comparing the relative merits of each type of policy.

A. Action policies

In this subsection, we consider three Action policy sets that might be used in our data center model: one to govern an AM's apportionment of its current resources among different transaction classes, a second to govern an AM's resource requests to the Resource Arbiter, and a third to govern the Resource Arbiter's allocation of resources across different AMs.

First, consider the following policy set, which specifies how an AM should apportion resources to Gold and Silver transaction classes:

G. IF ($\mathrm{RT_G} > 100$ msec) THEN (Increase $\mathrm{CPU_G}$ by 5%)

S. IF ($\mathrm{RT_S} > 200$ msec) THEN (Increase $\mathrm{CPU_S}$ by 5%) (1)

where $\mathrm{RT_G}$ and $\mathrm{RT_S}$ represent measured average response times for Gold and Silver transaction classes, respectively, and $\mathrm{CPU_G}$ and $\mathrm{CPU_S}$ represent the fraction of the CPU on each machine that is devoted to serving Gold and Silver class transactions, respectively. The policy pair is represented graphically in Figure 4.

The policy pair of Eq. (1) is quite plausible: the fraction of CPU devoted to each transaction class should be increased whenever the response time for that class exceeds a given threshold. Another part of each policy (not shown in Eq. (1)) would indicate the conditions under which it was to be applied; for instance there could be a periodic trigger that causes re-evaluation once every 30 seconds. Thus, if a class is still failing to make its objective after a first application of the pertinent policy, subsequent applications of the policy will result in further increases in the fraction of CPU devoted to that class, until finally the response time goal is met.

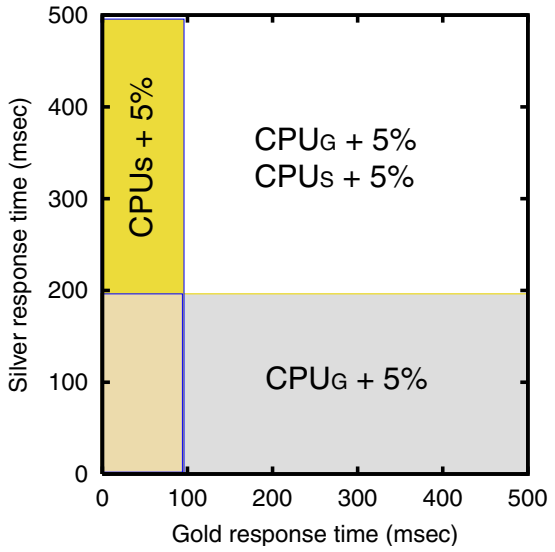

Fig. 4. Action policies for Gold and Silver transaction classes as a function of *current* measured response times for each class.

In terms of our framework, the space of possible current input states can be represented as a two-dimensional space $(\mathrm{RT}_G, \mathrm{RT}_S)$. The condition of each Action policy defines an entire subspace of input states that do and do not meet the condition. The condition of the Gold policy pertains to all states lying to the right of the vertical line at $\mathrm{RT}_G = 100$msec in Figure 4, while that of the Silver policy pertains to all states lying above the horizontal line at $\mathrm{RT}_S = 200$msec. These two intersecting lines create four regions. In the upper left region, only the Silver condition applies. In the lower right region, only the Gold condition applies. In both of these cases, the action to be taken is absolutely unambiguous. In the lower left region, neither condition applies. In this region, the policies do not explicitly state what action is to be taken, but a reasonable default policy is to take no action in this case. The system is satisfying both response time objectives, so no actions are warranted. Finally, in the upper right region, *both* conditions apply. If it is possible to increase both the Gold and the Silver CPU shares by 5%, then there are no conflicts, and both of these actions should be taken.

However, if the current state is in the upper right region and Gold and Silver are already sharing the entire CPU between them, then it is impossible to honor the Gold and Silver policies simultaneously. With the Gold and Silver policies in

conflict, the policy set as a whole is providing ambiguous, ill-defined guidance to the system. There are several ways to remove the ambiguity in the disputed upper right region of Fig. 4. One approach is to add meta-policies to disable some of the conflicting policies or triggering conditions [24], [25]. For instance, one could add the following rule to resolve the potential conflict:

$$\begin{aligned} &\text{IF ((ACTION}_1\text{: Increase CPU}_\text{G}\text{ by 5\%)} \\ &\quad \text{AND (ACTION}_2\text{: Increase CPU}_\text{S}\text{ by 5\%)} \\ &\quad \text{AND (CPU}_\text{G} + \text{CPU}_\text{S} > 90\%\text{))} \\ &\text{THEN(Ignore ACTION}_2\text{)} \end{aligned} \tag{2}$$

Here, we use "ACTION$_i$: *X*" to indicate that *X* is specified as an action by an active rule. We label the action by i.

Another method entails modifying the individual policies to eliminate overlapping conditions [26]. Suppose for example that one wishes the Gold class to receive extra CPU in the disputed region. One could add an additional clause to the condition of the silver policy that restricts its application to the upper left region whenever resources are sufficiently scarce, e.g.,

$$\begin{aligned} &\text{IF ((RT}_\text{S} > 200\text{msec)} \\ &\quad \text{AND ((RT}_\text{G} < 200\text{msec) OR (CPU}_\text{G} + \text{CPU}_\text{S} < 90\%\text{)))} \\ &\text{THEN (Increase CPU}_\text{S}\text{ by 5\%)} \end{aligned} \tag{3}$$

A third method is to leave the individual policies alone, but to permit policies to override one another in the case of conflict. One approach is to associate priorities with each policy with the understanding that, among those policies whose conditions are satisfied simultaneously, only the one with the highest priority will be executed [27]. Using priorities, one could achieve the same effect as in Eq. (3) merely by assigning to the Gold policy and Silver policies priorities of 10 and 5, respectively (with higher numbers indicating higher priorities). A more elegant and flexible approach is to express policies in logic that handles conflicts, and define additional policies that explicitly describe conditions under which overriding occurs [24], [28], [29], e.g., "Gold policy always has priority over Silver policy".

Since it is inherently difficult for humans to compose policy sets that map each and every condition into a single unique action, it is essential for systems that employ Action policies to provide methods for detecting and resolving conflicts among policies. In some cases, static analysis of a set of policies may be adequate to detect conflicts [26], [27]. For example, if is known *a priori* that the Gold and Silver classes together share the entire CPU, then one can determine statically that the Silver and Gold policies of Eq. (1) are in conflict whenever both of their conditions are satisfied, and the system can prompt the policy author to either modify or prioritize the rules to remove the ambiguity. In a more complex system in which CPU may be consumed by several other processes not explicitly mentioned in the policies, static analysis can only suggest the possibility of a conflict, and it is only during the running operation of the system that a definite conflict can be detected [30]. In this case, a possible solution is to establish which policy should take precedence in the event of a conflict. However, this is not a fully general solution for at least two reasons. First, one might want to base the choice on the nature of the conflict itself. For example, in terms of Fig. 4, the choice between the Gold and Silver policies might depend on the exact location within the upper right hand rectangle, with preference given to the class that is furthest from its response time target. Second, and more importantly, one might want to compromise between the proposed actions, rather than selecting one over the other. For example, suppose that Gold and Silver are both failing, and are together consuming 94% of the available CPU. One might wish to allocate just 4% more CPU to Gold, and the remaining 2% CPU to Silver.

Now, consider a second policy set that guides the AM's resource requests to the Resource Arbiter, for example:

$$\begin{aligned} &\text{IF (RT}_\text{G} > 100\text{msec OR RT}_\text{S} > 200\text{msec)} \\ &\quad \text{AND (CPU}_\text{G}\text{ + CPU}_\text{S} > 98\%\text{)} \\ &\text{THEN (Request 1 server)} \end{aligned} \tag{4}$$

This policy could be invoked repeatedly until the system achieves its response time goals, or is rebuffed once too often by the Resource Arbiter. Note that the terms regarding the CPU threshold in policy 4 need to be commensurate with the CPU thresholds in policies 2 and 3. In general, the policy writer is forced to think very carefully about how each newly added rule may interact with the ones that are already present in the rule set.

Finally, consider a third policy set used by the Arbiter to adjudicate among resource allocation requests from three different Application Managers, e.g.,

$$\begin{aligned} &\textbf{Arb1.}\ \text{IF (AM}_\text{A}\text{ requests } n \text{ servers) THEN Grant} \\ &\qquad \text{: Priority = 10} \\ &\textbf{Arb2.}\ \text{IF (AM}_\text{B}\text{ requests } n \text{ servers) THEN Grant} \\ &\qquad \text{: Priority = 8} \\ &\textbf{Arb3.}\ \text{IF (AM}_\text{C}\text{ requests } n \text{ servers) THEN Grant} \\ &\qquad \text{: Priority = 5} \end{aligned} \tag{5}$$

Note that the priority-based scheme does not permit compromises among the various requests, which might well be desirable in this situation. One way to achieve partial granting of requests is to write enumerate a set of rules—one for each possible triple of requests. However, this solution would be quite unwieldy, and would not generalize if more Application Managers joined the system.

B. Goal policies

Here we illustrate how the data center's behavior might be guided by Goal policies. First, the policy set used by an AM to govern apportionment of resources between the Silver and Gold transaction classes would take the form of performance targets, e.g.,

$$\begin{aligned} &\textbf{G.}\ \text{RT}_\text{G} \leq 100\text{msec} \\ &\textbf{S.}\ \text{RT}_\text{S} \leq 200\text{msec} \end{aligned} \tag{6}$$

(Strictly speaking, only the goals themselves appear in (6); the full policy would be for the system to *act so as to achieve these goals*. In what follows, we shall not make fine distinctions between Goal policies and the goals on which they are based.) The policy set of (6) is illustrated graphically in Fig. 5.

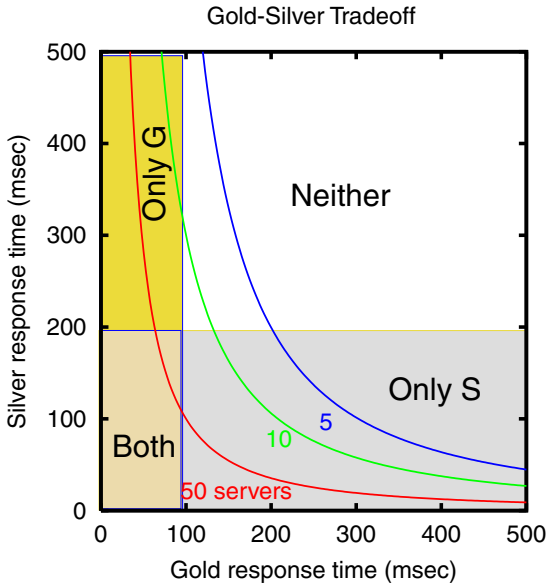

Fig. 5. Goal policies for Gold and Silver transaction classes as a function of *desired* measured response times for each class. Response-time states to the left of the vertical line at 100 msec satisfy the Gold criterion; those below the horizontal line at 200 msec satisfy the Silver criterion. Labels within each of the four regions formed by the intersection of these two thresholds indicate which policies are satisfied within that region. Hyperbolic curves represent boundary of feasible regions for a fixed workload and for various assumptions about total resource; the feasible region lies above and to the right of these boundaries.

Note that, in contrast to the policy set of Eq. (1), the state space in question is defined by the *desired* response times for Gold and Silver transactions, as opposed to the *current* response times. The two policies in Eq. (1) each define a set of goal states that are acceptable; the set of acceptable goal states for the policy set is given by the intersection—the shaded lower left hand rectangle of Fig. (5).

The policies contain no hints about *how* the goals are to be attained; instead, the system must employ other mechanisms in order to translate these goals into actions. Many such mechanisms are conceivable, but one can reason about them in general terms as follows. Denote the response time vector (T_G, T_S) of the two transaction classes by $\vec{T}$. In general, there will be some functional dependency $\vec{T}(\vec{\lambda}, \vec{C})$ of the response times upon the request rates $\vec{\lambda} = (\lambda_G, \lambda_S)$ and the apportionment of resource among the various classes $\vec{C} = (\mathrm{CPU}_G, \mathrm{CPU}_S)$.

Treating the quantity of CPU resource in absolute (rather than fractional terms, as in the previous subsection), suppose that the total amount of CPU resource available for the Gold and Silver classes is a constant CPU capacity C_0. Then the set of feasible states are those defined by $\vec{T}(\vec{\lambda}, \vec{C})$, for all $\vec{C}$ satisfying the constraint $C_G + C_S \leq C_0$, evaluated at the current value of the demand vector $\vec{\lambda}$. In general, this will define a region of state space with a nonlinear boundary. For example, suppose that the system can be modeled as a simple M/M/1 queue, and that the response time for each class is proportional to the amount of CPU devoted to it. Then

$$T_i = \frac{1}{\alpha C_i - \lambda_i}, \tag{7}$$

where α is some constant of proportionality that relates service time to CPU resource. The resulting feasible regions are bounded by hyperbolas, as shown in Fig. 5 for three different choices of C_0.

There are several cases to consider. First, suppose there is enough resource to satisfy both the Gold and the Silver Goal policies. Then the regions of desired and feasible states overlap. In Fig. 5, this occurs when $C_0 = 50$ servers—note the roughly triangular region of overlap between the desired states in the lower left rectangle and the feasible states bounded by the hyperbola. *Any* state in this triangle of intersection is equally acceptable. Then the appropriate action is to set $\vec{C}$ to any value such that $\vec{T}$ lies within the desired region.

However, if there is insufficient resource to satisfy both goals, but enough to satisfy either the Gold or the Silver goal alone, then there is a policy conflict. In Fig. 5, this occurs when $C_0 = 10$ servers—the hyperbolically bounded feasible region overlaps with the upper left rectangle (representing states that satisfy only the Gold policy) and with the lower right rectangle (representing states that satisfy only the Silver policy), but not with the lower left rectangle (representing states satisfying both the Silver and Gold policies).

A common approach to meeting quality of service goals in network environments is to restrict requests whose required goal cannot be met [12], [13]. However, data centers are typically obliged to service all requests (at least up to a certain request rate threshold). An alternative approach to resolving the policy conflict is to assign priorities to the policies, as with action policies. However, this conflict resolution strategy is not quite as clear-cut as one might expect. A naive approach is to successively give up on lower priority goals until one finally achieves intersection between the feasible and the desirable states. However, for some values of $\vec{\lambda}$ and C_0, it may be the case that the Silver goal can be achieved, but the Gold goal cannot. In Figure 6, this occurs when there are fewer than five servers. Should the AM then meet the Silver goal, but choose a state such that the deviation from the Gold response time threshold is minimized? Or should most of the resource be given to Gold to get it as close as possible to the desired response time goal, even if this means driving the Silver response time to a very unacceptable level? The possibilities get even more complex when one considers more than two transaction classes. Finally, if neither goal can be satisfied individually, then the policy set offers no guidance for how the system should behave. There is no way to distinguish any of the feasible states from one another. All are equally undesirable, because all represent failure, and goal policies

express only a binary success or failure, not degrees of failure.

As we discussed in the case of Action policies, if the current resource is insufficient to satisfy the goals, the AM can appeal to the Resource Arbiter for more resource. In the previous subsection, this function required a separate policy set. For Goal policies, this is not strictly necessary. A capacity planning engine armed with a good system model might be able to achieve this purely from the Goal policies of Eq. (6). From knowledge of the current value of λ and the function $\vec{T}(\vec{\lambda}, \vec{C})$, it could compute a value of $\vec{C}^*$ required to achieve overlap between the feasible and desired state and request from the Resource Arbiter enough additional servers to bring the total resource to $C_G^* + C_S^*$. In practice, there might be additional policies that establish further conditions that govern when the AM would request additional resources.

In response to a request for resource from the AM, the Arbiter would also ask the other AMs how many servers they need to meet their Goal policies before determining the global reallocation of servers. The Arbiter might employ the Action policies of Policy Set (5). As noted in the previous subsection, this particular policy set has some protection against resource allocation conflicts that occur when the total number of available servers is inadequate to satisfy the total expressed resource needs of the AMs. However, the situation may be more subtle. Suppose for example that there are several AMs. If the highest priority AM requests all of the resources, the other AMs would get nothing. This may not always reflect what is really desired. An alternative is for the Arbiter to itself use a Goal policy, e.g.,

$$\text{Ensure that at least } 3/4 \text{ of the AMs are satisfied.} \quad (8)$$

Since this policy set contains just a single goal, there is no danger of conflict in this case. Such a Goal policy could be reasonable if the data center is configured with sufficient servers to ensure it can be met. However, if Goal policy 8 cannot be satisfied, there is no guidance as to what the Arbiter should do.

One can envisage much more complex Arbiter policies that take into account other information, such as the state or the importance of each Application Environment; the mechanisms required to execute them may need to be rather sophisticated.

C. Utility Function policies

Now we illustrate how the data center's behavior might be guided by Utility Function policies. With a Utility Function policy, the system acts so as to optimize a given utility function, for example by setting control parameters or allocating resources appropriately. In the remainder of this section, we focus on the utility functions themselves, and do not make fine distinctions between Utility Function policies and the utility functions on which they are based.

First, we consider how utility functions can be used to apportion resource between the Gold and Silver transaction classes. Figure 6 shows two utility functions (measured in a monetary unit such as dollars): one for average response time of Gold transactions, and the other for the average response time of Silver transactions. The utility functions can be thought of as softened versions of the individual goals of Eq. (6). For Gold utility, the utility peaks at response times of just less than 100 msec, dropping significantly as the response time rises above 100 msec. The Silver utility curve has a similar shape, with a threshold at 200 msec, but a more gradual transition from maximum to minimum utility. These two utility functions can be mathematically combined into a single utility function in any number of ways. The two-dimensional utility function of Fig. 7 is obtained simply by summing the individual Gold and Silver utility functions U_G and U_S, e.g.

$$U(\text{RT}_G, \text{RT}_S) = U_G(\text{RT}_G) + U_S(\text{RT}_S). \quad (9)$$

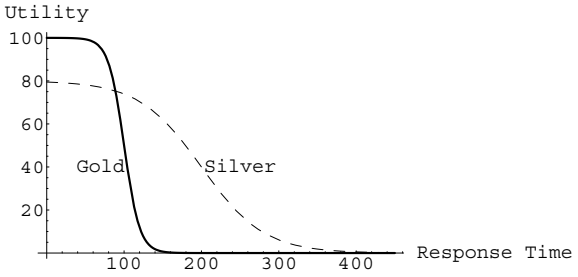

Fig. 6. Utility functions providing the utility in monetary units as a function of response time. The solid curve shows the utility for response time of Gold transactions and the dashed curve shows the utility for response time of Silver transactions.

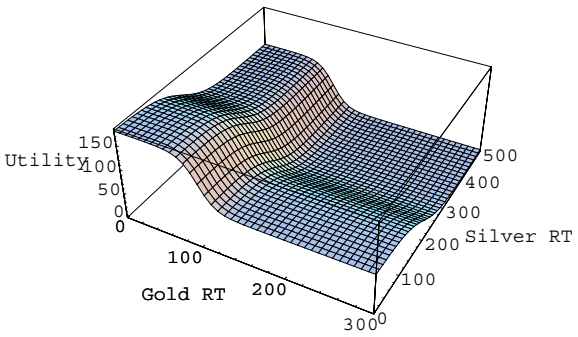

Fig. 7. A single utility function, providing the utility in monetary units as a function of joint average response time for Gold and Silver transactions.

The Goal policies of the previous section established a binary classification of all states into either "desirable" or "undesirable". The objective was then to place the system into any state that was both desirable and feasible—something that is not possible if these sets do not intersect. With Utility Function policies, each state has a real value rather than just a Boolean one, and the objective is to place the system into the feasible state with the highest utility. Thus there is never a failure. Also, note that a single utility function maps all states of interest into a single unique value, and as such only one is needed. Therefore there cannot be any conflicts.

It may however be the case that the highest attainable utility is deemed unacceptably low. This may be indicated by adding

an additional Goal policy of the form

$$\text{The utility should be at least } 50 \tag{10}$$

If the utility falls below 50, the AM could redress the problem by appealing to the Resource Arbiter for more resources. Rather than specifying a fixed number of servers, the AM could compute a *resource-level* utility function indicating the value as a function of the number of additional servers and send this function to the Resource Arbiter. To determine the resource-level utility function, each AM must compute, for each number r of servers, the maximal utility it could obtain if it had r servers and optimized its service-level utility function. Using the notation from Section III-B, the utility function for CPU capacity C_r obtained from r servers is

$$U(C_r) = \max_{\vec{C}|C_G+C_S \leq C_r} U(\vec{T}(\vec{\lambda}, \vec{C})) \tag{11}$$

To compute the optimal service-level utility that can be obtained from a particular resource level, each AM must possess a performance model that maps resource level and client demand into performance.

The Arbiter would query other AMs for their resource-level utility functions, and apply its policy to compute a best allocation. A typical policy for the Arbiter might be:

$$\text{Maximize sum of utilities of all AMs} \tag{12}$$

Although this optimization could be expensive (and may need to be approximated), it is conceptually simple. There are no conflicts that need be be resolved. Instead, the complexity is placed inside the AMs.

Although Utility Function policies have the benefit of being naturally conflict free, they impose an added burden of precisely specifying numeric values over the entire state space. It is well established that humans have difficulty specifying utility functions, and much work has been done to understand and ameliorate the difficulty [31], [32], [33]. In our data center domain, there are at least three challenges associated with specifying a utility function. One is to determine the appropriate shape of the function. Another challenge is to put multiple metrics on a common value scale. For the sake of exposition, this paper has explored a single simple performance metric only. In a real system, one would employ utility functions expressing preferences for a variety of performance metrics, as well as metrics describing availability, security, and any other service attributes of interest. Combining utility functions for individual metrics into a common multi-dimensional utility function requires (say via summation) that the individual utility function share a common value scale; some monetary unit would be a reasonable choice. A third challenge is to evaluate the relative value of multiple Application Environments.

At first blush, it would seem that Action and Goal policies are easier and more natural for humans to think about and specify. We acknowledge this to be true in relatively simple cases, but as we have shown, policy conflicts can complicate this process significantly. Indeed, the problem of evaluating the relative importance of different metrics and of different Application Environments becomes manifest in even moderately complex systems. While Utility Function policies place the burden on humans to be more precise about these relative tradeoffs, they have the advantage of forcing humans to consider these issues. In contrast, Action and Goal policies allow these issues to be swept under the rug, which can result in conflicts. Determining the appropriate shape for Utility Functions can be challenging, but often one can simplify this problem by starting out with what are effectively smoothed out goals, as with the Utility Function policies of Figure 6.

D. Mixing Policy Types

Observe that different components can use different types of policies. In Section III-B we showed an example in which an Application Manager used Goal policies and the Arbiter used an Action policy. As another example, the Arbiter can use a Utility Function policy even if the AMs are using other types of policies. For instance, if the AMs are using either Action or Goal policies for requesting more servers (as in sections III-A or III-B), the AM might use the following Utility Function policy:

$$\text{Maximize num(AMs) whose resource requests are granted} \tag{13}$$

However, policy types cannot be mixed arbitrarily. For example, the Arbiter's policy has to be consistent with the form of the resource requests, which depend on the type of policy used by the AM issuing the request. Policies (8) and (13) would not be meaningful if the AMs expressed the resource requests in terms of resource utility functions, as in Eq. (11). Likewise, the Arbiter could not use a Utility Function policy such as (12) if the AMs issued requests for a specific quantity of resource, as in section III-B. These examples hint at the careful relationship that must be maintained between policies used by various elements in an autonomic computing system and the protocols for interaction between the elements.

Handling multiple policy types within the same component is also possible, but much more complicated. Here we highlight some of the salient issues. Overall, Utility Function and Goal policies tend to be more naturally compatible with one another than either is with Action policies. The reason is simple: both are expressed in terms of the space of desired states rather than the current state. We anticipate that one common pattern for combining Utility Function and Goal policies within a single component will involve using a Goal policy as a constraint on the optimization problem defined by the Utility Function policy.

For example, consider an Application Manager with a Policy Set consisting of two policies: the Utility Function shown in Figure 7, plus the Goal policy $\text{RT}_\text{G} \leq 200$msec. Expressed as a constraint to the optimization procedure, the Goal policy would simply exclude the region of state space lying above $\text{RT}_\text{G} = 200$msec. As another example, consider the communication between this AM and the Resource Arbiter. The hard constraint on RT_G might translate into a hard constraint on the minimum number of servers that the AM

must obtain in order to satisfy its Policy Set. In other words, the AM could submit the constraint "Obtain at least 1 server" along with its utility function. If the Resource Arbiter uses a Utility Function policy like (12), and it is willing to accept this constraint by dynamically (and temporarily) adding it to its own Policy Set, then the Resource Arbiter's Policy Set is likewise a mixture of Utility Function and Goal policies.

Of course, since Goal policies introduce constraints, and thus eliminate regions of state space, Policy Sets that mix Utility Function and Goal policies possess the same problems as pure Goal Policy Sets. Chief among these problems are the possibility of failure to satisfy the Policy Set (which necessitates other policies that govern behavior in the event of failure), and the possibility of conflict if there are multiple constraints that are not mutually satisfiable (which is really just a special case of Policy Set failure).

Combining Action policies with Utility Function policies or Goal policies is more problematic. If the optimizer that operates upon the utility function tries to control variables that appear in the Action clause of an Action policy, conflicts are bound to occur. Unfortunately, the control variables available to the optimizer (e.g. the CPU devoted to each class, in our simple example) are typically not mentioned explicitly in the utility function. Therefore, pure syntactic checking of the Policy Set alone cannot reveal potential conflicts. If it is somehow determined that there is a potential conflict (through detailed knowledge of the control variables available to the optimizer, for example), then the various options for conflict resolution discussed in Section III-A may be explored, such as assigning priorities to the policies or writing more general metapolicies.

A possible way to minimize the risk of conflicts between Action and Utility Function policies employed within the same component is to ensure that they are applied to very different classes of service attributes, such as performance and availability or security. However, this is not a surefire solution because the control variables that underlie performance, availability and security are not necessarily mutually exclusive. For example, consider a utility function for performance. The optimizer that takes the utility function as its objective may have as one of its control variables the number of servers allocated to each transaction class. An Action policy may govern the availability characteristics for the Gold class by controlling (among other things) the number of servers that are allocated to it. Once again, conflict is possible.

Another way to avoid conflict is to ensure that the optimizer cannot control any variable mentioned in the Action clause of an Action policy. This may work, although it is still possible that indirect interactions among variables will cause trouble. For example, certain actions (such as installation of new software) may temporarily make a resource (along with any associated control variables) temporarily unavailable. If the optimizer is not aware of this outage, and it relies on some of these control variables, it may produce inappropriate results.

IV. Conclusion

We have presented a unified view of different types of autonomic computing policy from an AI perspective. States and actions are the key concepts that relate Action, Goal, and Utility Function policies. In accordance with the AI principle of rational agents, the purpose of a policy is to provide guidance for an autonomic system to choose actions that move it into desirable states. Still, only if the policies are complete and coherent can rationality be realized in the system and correct behavior be achieved.

To this end, policies must cover the entire state space and provide unambiguous guidance to the system. Because this cannot generally be achieved by considering individual policies in isolation, Policy *Sets* are the right way to think about policy. Unfortunately, people typically think of policies individually, rather than as Policy Sets, because they have difficulty comprehending and reasoning about Policy Sets in systems of realistic complexity. As seen in section III-A, even small Policy Sets can exhibit subtle complexities. This points to a need for mechanisms to enforce the use of Policy Sets, and to support their creation and management.

Possessing a unified view of policy gives us the power to mix the different types of policy within a system, so long as we ensure that unambiguous action is guaranteed across the entire state space. Our data center scenario illustrated some inherent advantages and disadvantages of each of the different types of policy.

A nice feature of Action policies is that they do not require an operational model of the system in order to be used. In real systems, performance models will be considerably more complex than the simple M/M/1 queuing model given in Eq. 7. It will be hard to obtain accurate models of real, complex systems, and by and large they do not exist today. For this reason, we expect Action policies to predominate in early autonomic computing systems. One might argue that Action policies are also appealing in their simplicity, but this is debatable. Action policies require the user to know about the space of control variables as well as the state space, which typically requires a large amount of domain expertise. Furthermore, even in the simple examples of this paper, we found it necessary to reconceive our two simple initial Action rules—which seem quite reasonable on first inspection—to ensure that the AM would avoid conflict. Although our example is idealized, the revised rule (3) is somewhat complex, and requires a bit of thought for a human to verify its correctness. Priority schemes are another approach, but in either case it seems clear that even moderately larger policy sets would become quite unwieldy.

As system models and planners become better and more widely available, Goal based policies will become more feasible and more prevalent. Goal policies have an advantage over Action policies because they give an autonomic system the flexibility to choose the best actions under the current conditions. Furthermore, Goal policies reduce the burden on the user because they are expressed only in terms of the state space, leaving to an automated mechanism like an optimizer or

planner the task of dealing with the space of control variables (such as CPU allocation in our simple example). Thus the user need only deal with a smaller and easier set of variables. However, Goal policies are dependent upon system models and planning algorithms, and moreover they are still susceptible to conflict. When not all goals can be realized, the Goal policies alone provide no guidance. Additional policies and mechanisms may be needed to resolve the impasse, all of which can complicate the resulting Policy Set.

Utility Function policies are quite appealing for advanced autonomic computing systems because they inherently avoid conflicts. Optimizing a utility function is conceptually more straightforward than having to patch a Policy Set with additional policies, priorities, and various conflict resolution mechanisms, as is necessary for Action and Goal policies. However, as with Goal policies, Utility Function policies rely on the existence of potentially complex system models and optimization algorithms. Furthermore, since it is difficult for humans to specify them, Utility Function policies will not be very usable until we have good interfaces and algorithms for eliciting them. It may take a some years of solid research effort to address all of these issues, but ultimately the inherent advantages of Utility Function policies may make them the predominant form of policy.

ACKNOWLEDGMENTS

The authors thank Rajarshi Das and Gerald Tesauro for sharing their insights on utility functions, policy, and artificial intelligence, and for numerous helpful comments on this paper.

REFERENCES

[1] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–52, 2003.

[2] "Autonomic computing: IBM's perspective on the state of information technology," http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf%.

[3] M. Bearden, S. Garg, and W. jyh Lee, "Integrating goal specification in policy-based management," in *2nd International Workshop on Policies for Distributed Systems and Networks*, 2001.

[4] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The Ponder policy specification language," in *2nd International Workshop on Policies for Distributed Systems and Networks*, 2001.

[5] J. Strassner, "How policy empowers business-driven device management," in *3rd International Workshop on Policies for Distributed Systems and Networks*, 2002.

[6] A. Westerinen, J. Schnizlein, J. Strassner, M. Scherling, B. Quinn, S. Herzog, A. Huynh, M. Carlson, J. Perry, and S. Waldbusser, "Terminology for policy-based management," University of Wolverhamptons School of Computing and Information Technology, http://www.scit.wlv.ac.uk/rfc/, Tech. Rep. RFC 3198, 2001.

[7] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Prentice Hall, 2003.

[8] C. Efstratiou, A. Friday, N. Davies, and K. Cheverst, "Utilising the event calculus for policy driven adaptation on mobile systems," in *3rd International Workshop on Policies for Distributed Systems and Networks*, 2002, pp. 13–24.

[9] H. Lutfiyya, G. Molenkamp, M. Katchabaw, and M. Bauer, "Issues in managing soft QoS requirements in distributed systems using a policy-based framework," in *2nd International Workshop on Policies for Distributed Systems and Networks*, 2001.

[10] L. Lymberopoulos, E. Lupu, and M. Sloman, "An adaptive policy based management framework for differentiated services networks," in *3rd International Workshop on Policies for Distributed Systems and Networks*, 2002, pp. 147–158.

[11] A. Ponnappan, L. Yang, and R. Pillai.R, "A policy based QoS management system for the IntServ/DiffServ based internet," in *3rd International Workshop on Policies for Distributed Systems and Networks*, 2002, pp. 159–168.

[12] S. Wang, D. Xuan, R. Bettati, and W. Zhao, "Providing absolute differentiated services for real-time applications in static-priority scheduling networks," in *IEEE Infocom*, 2001.

[13] J. Yoon and R. Bettati, "A three-pass establishment protocol for real-time multiparty communication," Texas A&M University, Tech. Rep. 97-006, 1997.

[14] A. Chandra, W. Gong, and P. Shenoy, "Dynamic resource allocation for shared data centers using online measurements," in *International Workshop on Quality of Service*, 2003, pp. 381–400.

[15] J. S. Chase, D. C. Anderson, P. N. Thakar, and A. M. Vahdat, "Managing energy and server resources in hosting centers," in *18th Symposium on Operating Systems Principles*, 2001.

[16] T. Kelly, "Utility-directed allocation," in *First Workshop on Algorithms and Architectures for Self-Managing Systems*, 2003.

[17] R. Rajkumar, C. Lee, J. P. Lehoczky, and D. P. Siewiorek, "Practical solutions for QoS-based resource allocation problems," in *IEEE Real-Time Systems Symposium*, 1998, pp. 296–306.

[18] S. Lalis, C. Nikolaou, D. Papadakis, and M. Marazakis, "Market-driven service allocation in a QoS-capbable environment," in *First International Conference on Information and Computation Economies*, 1998.

[19] P. Thomas, D. Teneketzis, and J. K. MacKie-Mason, "A market-based approach to optimal resource allocation in integrated-services connection-oriented networks," *Operations Research*, vol. 50, no. 4, 2002.

[20] H. Yamaki, M. P. Wellman, and T. Ishida, "A market-based approach to allocating QoS for multimedia applications," in *Second International Conference on Multi-Agent Systems*, 1996, pp. 385–392.

[21] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das, "Utility functions in autonomic systems," in *International Conference on Autonomic Computing*, 2004.

[22] C. Boutilier, T. Dean, and S. Hanks, "Decision-theoretic planning: Structural assumptions and computational leverage," *Journal of Artificial Intelligence Research*, vol. 11, pp. 1–94, 1999.

[23] D. Chess, A. Segal, I. Whalley, and S. White, "Unity: Experiences with a prototype autonomic computing system," in *International Conference on Autonomic Computing*, 2004.

[24] H. V. Jagadish, A. O. Mendelzon, and I. S. Mumick, "Managing conflicts between rules," in *Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1996, pp. 192–201.

[25] J. Chomicki, J. Lobo, and S. Naqvi, "A logic programming approach to conflict eesolution in policy management," in *Seventh International Conference on Principles of Knowledge Representation and Reasoning*, 2002, pp. 121–132.

[26] M. Koch, L. V. Mancini, and F. Parisi-Presicce, "Conflict detection and resolution in access control policy specifications," in *5th International Conference on Foundations of Software Sicence and Computation Structures*, ser. LNCS. Springer, 2002, no. 2303.

[27] E. C. Lupu and M. Sloman, "Conflicts in policy-based distributed systems management," *IEEE Transactions on Software Engineering*, vol. 25, no. 6, 1999.

[28] B. N. Grosof, Y. Labrou, and H. Y. Chan, "A declarative approach to business rules in contracts: courteous logic programs in XML," in *ACM Conference on Electronic Commerce*, 1999, pp. 68–77.

[29] R. Agrawal, R. Cohrane, and B. Lindsay, "On maintaining priorities in a production rule system," in *17th International Conference on Very Large Data Bases*, 1991, pp. 479–487.

[30] N. Dunlop, J. Indulska, and K. Raymond, "Dynamic conflict detection in policy-based management systems," in *Sixth International Enterprixe Distributed Computing Conference*, 2002, pp. 15–26.

[31] C. Boutilier, "A POMPDP formulation of preference elicitation problems," in *Eighteenth National Conference on Artificial Intelligence*, 2002, pp. 239–246.

[32] V. Iyengar, J. Lee, and M. Campbell, "Evaluating multiple attribute items using queries," in *3rd ACM Conference on Electronic Commerce*, 2001, pp. 144–153.

[33] R. L. Keeney and H. Raiffa, *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. Cambridge University Press, 1993.