Policies for Self-Managing Communities in Agile Organizations

Kevin Feeney, David Lewis, Vincent Wade

Abstract—Modern management practice is increasingly focused on human resource management as enterprises attempt to become more agile in response to ever changing customer requirements, competitive environments and technological innovation. This typically involves moving to flatter organizational structures with the devolution of decision-making authority to increasingly self-managed teams. It also requires flexible and complex flows of communication within and between teams. Collaborative and workflow IT system must therefore become increasingly flexible to support the frequent changes in communication flows that come from agile work practices based around project- and team-centered working. However, such IT systems are still largely managed using techniques designed for centralized IT administration. Even policy-based management, which aims to deliver flexibility in systems management, relies on highly centralized modeling of human and IT resources. In this paper we show how a recent innovation in community-based policy-based management allows the IT administration to be decentralized in a way that reflects and is highly responsive to the needs of devolved decision-making in agile organization.

Index Terms—Policy-based management, agile organizations, self-management

I. INTRODUCTION

Recent decades have seen a steady shift in organizational management theory away from the centralized planning and control techniques of Henry Ford and F.W. Taylor, which delivered efficient mass production in the early part of the 20th century [thompson]. Instead, as organizations grew larger and more complex, and as their success became more reliant on delivering services to customers, management techniques have focused increasingly on efficiently managing the human resources of firms. This involved the development of specialisms, a focus on team building, the study of business processes and their monitoring to deliver continual improvement in reducing costs, satisfying customers and innovating in services. Most recently, this trend has been addressed by the move to so-called agile organizations [dove]. These are organizations where flexibility is inherent, in permitting them to be more responsive to changing customer needs, competitive pressures and technological development. To achieve agility, organizations must distribute intelligence more widely and allow informed decisions to be made more frequently with responsibility devolved to those closest to the drivers of change, e.g. customer-facing staff. Human resource professionals often talk in this regard of the move to high-performance working, where decision-making responsibility is devolved to self-managed teams. The key to successful high-performance working is in the management of the complex flows of communication between teams. The timely and comprehensible communication of signals about changing goals, problems, constraints, threats, opportunities and solutions is the essential medium through which agile organizations empower and enable their members for high-performance working.

IT support for flexibility in organizations has largely centered on supporting business process engineering directly, e.g. through the use of script driven workflow engines that control the flow of information between different parts of a firm. This, however, still relies on a centralized model of a business process, and may thus be limited in the degree of decentralized decision-making it can support. Increasingly, organizations pursuing agility are drawing inspiration from technologies and associated organizational forms that have gained popularity on the Internet. Technologies such as instant messaging and web-based conferencing are now increasingly being integrated into enterprise collaboration offerings. More significantly, there seems to be a strong resonance between the decentralization of decision-making in agile organizations and the self-forming communities that are rapidly developing around collaborative content management technologies, such as platforms for discussion forums, blogs and wikis. These collaborative content generation technologies are also starting to be included in enterprise collaboration platforms. Such platforms allow enterprises to better support the rich communication flows between groups, which increasingly are geographically distributed and which often involve representation from customers and suppliers. Supporting agility in such an enterprise requires the free flow of information, knowledge and ideas across and organization, or even encompassing suppliers and customers. This requires a move away from communication flows restricted by workflow models, to a less constrained management of communication and collaboration. However, even where free flow of
information is sought there, is the need to restrict communication of certain knowledge. For instance, communication with suppliers and customers must not reveal commercially or competitively sensitive information. Equally, regulations in areas such as financial services [glba][sox] and health care [hipaa] require that the sharing of certain information be restricted in an enforceable and auditable manner.

However, the collaboration platforms currently used both in enterprises and for Internet communities offer only rudimentary mechanisms for managing the community itself and the flows of information within it. Community membership is typically mediated by user lists or, at best role-based models, which prove difficult to adapt to the frequent changes that organizations inevitably undergo. Such changes to an organization can involve: growth in membership, specialization into sub-groups, merging with other groups, splitting with sub-groups, cooperation with other communities on shared projects and finally dissolution. In addition, an organization may change the way it operates and makes decisions. Operational changes may involve adding new types of collaborative resources to those that organization shares and manages. Changes to decision-making mechanisms may include adopting new voting or polling techniques or making decisions conditional on organizational knowledge obtained at enforcement time, e.g. access permission may be granted based on the peer ranking received by a user rather than allocation to a static role. Such changes may, however, be difficult to model as complex organizations only have only a partial understanding of how they operate at any point in time, so administrative schemes must handle progressive capture of organizational knowledge, including operational rules.

Community Based Policy Management (CBPM) is a novel technology that allows the automated enforcement of an organization’s high level goals by mapping them into declarative policy rules [feeney05]. It differs from previous policy-based management approaches [strassner], in its unique and novel use of the community as the fundamental structural abstraction. It allows groups within an organization to form self-managing communities within precisely defined boundaries. Restricting the interconnection of different businesses and IT rules in this way minimizes the portion of an organization’s rule set that a community needs in order to self-manage, while clear provision of rules ensures the causes of conflicts between policies authored in different parts of an organization are immediately identified thereby quickening their resolution. This allows an organization’s structure to evolve over time as new communities with varying levels of autonomy are formed and new resources are delegated between them. In this paper we review the set of CBPM primitives and then provide a detailed case study showing how they may be used to support the definition of self-managing structure within an organization. This, together with CBPM’s demonstrated ability to integrate with existing IT policy management systems, provides controlled devolution of decision-making authority across an organization in support of agile working practices.

II. COMMUNITY-BASED POLICY MANAGEMENT

Community-Based Policy Management (CBPM) is a policy-based management scheme that aims to free organizations from the reliance on external authorities in determining the grouping abstractions used in authoring policies [feeney04]. For instance, modern role-based access control systems [sandhu][lupu], such as that used in IBM’s Workplace, typically require the services of external consultants to define roles and their policies, an approach that is expensive but also brittle in handling frequent organizational change [driver]. This is because the defined set of roles can be used freely across an organization such that the semantics of roles become embedded in the policy rules in an organization. This makes extensive changes to the policy set necessary whenever the semantics of roles are changed. Though this may not be an issue for highly centralized organizations with stable structures of authority, e.g. the military, it is not suitable for agile organizations. Agile organizations, by definition, view organizations structure and the roles played by people within them as provisional, subject to on-going evaluation and therefore to frequent change. Embedding role semantic in an organizations policy rule-set ossifies it to organization change that involves redefining the function of groups of employees.

CBPM uses the notion of community as the grouping abstraction with the aim of allowing groups within the organization itself to define communities to reflect the natural structure of the organization. Communities form a hierarchy with the root representing the entire organization, having authority over sub-communities, representing organizational units. Communities can be granted authority to access resources owned by the organization. As communities themselves are resources, authority can also be granted to spawn further sub-communities. Policies are bound to communities. Therefore, policies in a community control which actions on which resources can be undertaken by the members of that community. The community structure embodies the structure of the human organization it serves. The resources and actions that can be performed upon them are modeled in separate hierarchical tree structures, allowing abstraction of resources and of actions in order to simplify the authoring of policies.

A key part of the scheme is that the root community by default consists of all the members of its sub-communities and all the resources managed by all the sub-communities. The structure of the organization is therefore modeled through spawning of sub-communities to represent organizational units and the controlled delegation of authority through the community hierarchy. In the following section we describe how CBPM can be used to implement a self-managing organization, i.e. one where the authoring of resource management policies is tightly coupled to the structure of the organization and thus reactive to changes in that structure.
This is the key to making an organization self-managing.

<table>
<thead>
<tr>
<th>Community Record Management Service Primitives:</th>
</tr>
</thead>
<tbody>
<tr>
<td>The CBPMS Community Record Management Service (CRMS) supports 12 management primitives which allow an organization to define its policies through a dynamic evolutionary process. Resource authorities are delegated down the community hierarchy to provide an authority scope for each community – specifying what events the community can author policies for. The CRMS primitives can be divided up as follows:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>genesis</th>
<th>expel</th>
<th>Create / Destroy a root community</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spawn</td>
<td>Cull</td>
<td>Create / Destroy a sub-community of an existing community</td>
</tr>
<tr>
<td>Delegate</td>
<td>recall</td>
<td>Delegate authority to a sub-community or cancel a delegation</td>
</tr>
<tr>
<td>Policy</td>
<td>revoke</td>
<td>Specify a policy for a community or remove an existing one</td>
</tr>
<tr>
<td>Federate</td>
<td>withdraw</td>
<td>Join or leave a federation</td>
</tr>
<tr>
<td>Grant</td>
<td></td>
<td>Assign ownership of a resource to a community</td>
</tr>
<tr>
<td>gatekeeper</td>
<td></td>
<td>Define a membership rule for a community</td>
</tr>
</tbody>
</table>

**TABLE A: COMMUNITY MANAGEMENT PRIMITIVES**

III. SELF-MANAGEMENT

One of the main motivations behind the Community Based Model was to provide a policy based management to support community self-management. The model allows the members of particular communities to manage sections of the community-based model themselves – through the invocation of the primitives outlined in Table A. These enable self-management principally in the following ways:

- Authority to manage resources can be delegated across an organization in a controlled way that allows any community to manage its own structure and policies itself, within a precisely scoped domain of authority.
- Authority to invoke the CBPMS primitives is managed by the CBPMS itself. For example, we can author policies, which dictate the contexts in which a user can invoke the delegate primitive for a particular community. The CBPMS manages its own access control through the authoring of policies within it – just like any other managed resource.

Both of these types of self-management are readily supported by the CBPMS in exactly the same way that it supports the management of any other resource. We simply create a resource model of the CBPMS itself, where the communities and their components are nodes on the target tree and the various operations form the nodes of the action tree, as shown in figure 1.

The CBPMS resource tree structure captures some of the fundamental semantics of the resource being managed. Authority is distributed by delegating nodes from the resource trees, known as resource authorities. A community which possesses a resource authority for a node that is above a second node in the tree, will always have authority over the second node. We shall see the implications of this in the second step of our implementation of self-management.

Not all of the resource authorities that we can compose from the Cartesian product of the resource nodes and the action nodes are meaningful. For example, the CBPMS Actions (i.e. the Community Record Management Service Primitives: Grant, Genesis and Expel) are not associated with any particular community and thus the target field of resource authorities concerning invocations of these primitives will be ignored by the system. There is no difference in the policy decision request resource authorities (CBPMS, IT, genesis) and (CBPMS, Sales, genesis). However, the resource authorities (CBPMS, IT, policy) and (CBPMS, Sales, policy) are decidedly different. One conveys authority to define policy for the IT community, the other for the Sales community. In general, we expect that the entity which generates Policy Decision Requests and enforces the results will send meaningful combinations of targets and actions in resource authorities and won’t attempt to carry out meaningless actions just because a policy may have permitted them to do so. There are also some valid resource authorities which can be distributed to communities but should never arrive as part of a policy decision request. For example, the composite action nodes on the action tree in figure 2 are unlikely to correspond with a resource authority that might be generated by the entity that sends policy decision requests. However, they are useful in terms of allowing authority over sets of actions to be delegated in a single chunk rather than having to delegate each primitive individually.

Having created the resource model, we assign ultimate authority over the community to the community itself and the community is thereafter self-managed in both senses of the word. In order to understand fully how this process works while using it as our first detailed example of the CBPMS in operation, we will develop an example problem at length.

A. Problem Introduction

Problem: implement a self-management interface for a CBPMS system where the community structure is as shown in figure 2.
policies that is capable of providing an implementation of the purposes of this example, we will use the subset of the Ponder. This turns out to be very easy to do as many policy specification languages can be integrated into a service which implements the basic PRS service specification. For the purposes of this example, we will use the subset of the Ponder policy specification language [damianou] that deals with access control policy specification. We use Ponder here because of its expressiveness and easily understood syntax. In Ponder, an access control policy is specified as follows:

```plaintext
inst ( auth+ | auth- ) policyName {
  subject [<type>] domain-Scope-Expression ;
  target [<type>] domain-Scope-Expression ;
  action action-list ;
  [when constraint-Expression ; ]
}
```

We note here that the three fields `subject`, `target` and `action` are superfluous to our requirements as the subject is embodied in the community within which the policy rule is specified and the information contained in the target and action fields are incorporated into the resource authority associated with our policy specifications. Hence we can set the subject and target to either the same information as is contained in the resource authority or else to the root of Ponder’s domain structure “” and the action list as “*” meaning any action. Since the contents of these fields will be identical in all of our specifications, we will omit them. This leaves us with policy specifications which take the following form:

All managed resources are modeled as authority trees, each of which can be divided into an action tree and a target tree. A resource authority is a triple [R,T,A] where R is the resource model, T is a node on the resource’s target tree and A is a node on the resource’s action tree. Every delegation and policy is associated with a particular resource authority, which defines the scope of the delegation or policy. The authority trees for a community resource are shown below.

![Diagram of CBPMS Resource Tree for Self-Management Example](image)

Figure 2 Specification of CBPMS Resource Tree for Self-Management Example
Ponder’s policy specifications have the following semantics: the constraint-Expression is evaluated by the policy engine and if it evaluates to true then the return value is either auth+ or auth-, depending on which is specified. In order to integrate these Ponder policy specifications into our PRS we will need to provide some simple “wrapper” code around it. For the purpose of this example, we assume that we have a mechanism which translates the context information that is sent to the PRS into a form that Ponder can read and we assume that whatever information is required in order to evaluate the constraint expression is included in the context. So, for example, if a Ponder constraint expression states: \( \text{Time.between}("1600", "1800") \) and \((\text{Priority} > 2)\); we assume that the current time value and the priority value are both encoded in the context vector and are mapped to the relevant variables within Ponder by our wrapper code.

We assume that when our example starts there are no resources managed by the system and that there are no policies within it other than membership policies for each of the communities pictured. We also assume that the set of resource models is the null set initially. In short, we have just initialized our system and our first task is to use it to implement a self-management framework.

**B. Step 1. Create the resource model**

The first step in the process is to build the resource model by constructing a target tree and an action tree, as shown in figure 2. These are partial-order models of the resource types and of the actions that can be performed upon them. The hierarchical nature of these trees aids in the searching of the policy set and provides levels of abstraction to ease the definition of policy rules. The communities who are viewed as being the subjects of the actions are modeled as nodes on the target tree. There are many possible ways in which we could design this resource model. The design of the resource tree should reflect the granularity of the policy decision requests that will be sent to the CBPMS for this particular resource, which reflects, in turn, the granularity of access control requests that are generated by the application which, in this case, is the CBPMS Policy Management Tool. From the point of view of the CBPMS all resources are simply a hierarchical model which will be used to compare the authority scope of policies and policy decision requests that arrive.

**C. Grant the CBPMS authority to the root community**

Having created the resource model of the CBPMS, we simply call the grant primitive to transfer ultimate authority over the CBPMS to the root community.

\[
\text{grant}(\text{Example.org, ra}) : \\
ra = (\text{CBPMS, All Actions, All Communities})
\]

Thereafter, access to the CBPMS can be managed by the system itself. Applications that wish to access the CBPMS primitives can generate appropriate policy decision requests and either allow or deny access according to the semantics of the policy decisions that the CBPMS will return. However, when we are dealing with the CBPMS resource itself, it is not sufficient to merely grant the resource authority to the community since we face a bootstrapping problem. Although the community possesses authority for the resource, there are no policies in existence for the resource and thus no way of permitting members of the community to access the resource. The resource authority is required before any policies are written, since it is possession of authority over the CBPMS resource itself which enables the authoring of policies about resources. However, a policy is required in order to turn the community’s authority into policies which allow the members of the community to use the resource.

The solution is simply to create a default policy in the recipient community whenever it is delegated authority to author policies – a policy that defines how members of the community may author policies, a meta-policy governing policy creation. In this case, we assume that the job of setting up a self-managed community structure has been allocated to an individual within the organization, who is identified as architect. Therefore, as soon as we have granted ultimate authority over the CBPMS resource to the root community, we invoke the policy primitive to establish this bootstrapping meta-policy. We assume that our PRS wrapper code maps a passed context variable representing the user’s id into the user.id variable in Ponder’s variable space.

\[
\text{policy(Example.org, p.initial)} : \\
p.initial = (\text{ra, ps}), \\
ra = (\text{CBPMS, Example.org, Community Actions}), \\
ps = (\text{auth+ InitialConstruction} \text{ when user.id = architect})
\]

This policy specifies that any user with id equal to architect can invoke the community policy primitives (policy, revoke, membership) for any target beneath Example.org in the target hierarchy. Since our target hierarchy is constructed to mirror the community hierarchy, this means that the architect is authorized to write policies for all communities in the hierarchy. Having invoked these two CBPMS primitives, the system bootstrapping is complete. These invocations were made without reference to any access control policies. We assume that the system provides a mechanism to initially bootstrap itself in such a manner before it starts to enforce access control policy on itself. However, after these two initial primitives have been invoked, we can entirely manage access to the system through the CBPMS itself.

**D. Step 3: Define global policies in root community**

After granting ultimate control over the CBPMS to the Example.org community itself, we now need to distribute the authority throughout the organization in order to enable communities beneath the root community to self-manage. At this stage we can observe that the root community will always retain authority to compel any community to carry out any action as a consequence of the way that the target tree is constructed with all other communities represented as nodes.
on the target tree beneath the root community. The root community can delegate self-management authority to any community in the hierarchy, but it can always define policies about any aspect of the CBPMS itself and they will apply to all members of the organization and have priority over any other policies that are defined elsewhere in the system. Thus, before delegating any authority we will define those policies at the root community level that we want to apply across the entire system. We can add policies to the root community at any time, but by doing so before we distribute authority we ensure that there will be no window of opportunity that might be used by a sub-community in order to define policies which might allow their members to take actions that we wish to forbid across the system.

Since this CBPMS is serving a single organization which is already present in the system, and the CBPMS primitives are designed to support community management spread across multiple organizations, the organization has no further need for the CBPMS Actions. Therefore, it defines in the root community a negative authorization rule with a condition list which will always evaluate to true. The system architect defines this community policy on behalf of the root community by invoking the CBPMS policy primitive:

```
policy(Example.org, p.forbidcbpms) :
  p.forbidcbpms = (ra, ps),
  ra=(CBPMS, All Communities, CBPMS Actions),
  ps=(inst auth= forbidCBPMSActions(when true)
```

Whenever an individual actor attempts to invoke this primitive with the community policy management tool, the tool will generate a policy request to the CBPMS, since the CBPMS is now managing itself. In this case the policy decision request will have the form:

```
pr=(ra,con),
  ra=(CBPMS, Example.org, policy),
  con=(user.id=architect, new_policy=p, ...)
```

The CBPMS will then invoke the decision function in order to generate a policy decision. In this case, the CBPMS will find that the root community for the resource authority in the policy decision request is the root community of the organization. Then it will search for a community policy rule which has a scope that incorporates the policy decision request’s resource authority. In our bootstrapping phase, we defined a policy labeled enableInitialConstruction with a resource authority = (CBPMS, Example.org, Community Policy Actions). This resource authority has greater authority scope than the policy in the policy decision request and thus it will be evaluated. According to the policy specification, if the user id in the passed context evaluates to architect, then the policy will return the policy decision auth+ to the community policy management tool causing it to invoke the primitive as the action is authorized, otherwise the policy will return a policy decision of none and thus it has not been authorized. Thus, in this example, the architect’s invocation of the policy primitive will be authorized by the system, giving us our first practical example of the CBPMS providing access control for itself.

The invocation of the policy primitive will cause the policy p to be added to the community policy set of the CBPMS, which means that whenever a policy request arrives which is within the scope of the resource authority (i.e. whenever an access request arrives for any of the CBPMS primitives), the system will return a negative authorization decision and the Policy enforcer will deny access. Since this policy is defined in the root community, it applies to all members of the organization and no policies defined elsewhere in the system can override it, even in cases where a resource authority within the relevant scope is delegated to another community. Similarly, the architect goes on to create a second policy for the root community, this time forbidding the creation and use of the primitives that concern themselves with federations, again for the same reason as before.

```
policy(Example.org, p.forbidfederal) :
  p.forbidfederal = (ra, ps),
  ra=(CBPMS, All Communities, Federal Actions),
  ps=(inst auth= forbidFederalActions(when true)
```

### E. Step 4: Delegate self-management authority

Having established our system wide rules in the root community, we go on to delegate self-management authority to the communities in the system. These primitives will all be authorized by the CBPMS when invoked by the architect as a result of the initial policy defined above, thus we will not mention the generation of policy decision requests for these primitives in the discussion:

```
delegate(Example.org, ra, Accounts)
  ra=(CBPMS, Accounts, community actions)
delegate(Example.org, ra, IT)
  ra=(CBPMS, IT, community actions)
delegate(Example.org, ra, Team1)
  ra=(CBPMS, Team1, community actions)
delegate(Example.org, ra, Team2)
  ra=(CBPMS, Team2, community actions)
delegate(Example.org, ra, Marketing)
  ra=(CBPMS, Marketing, community actions)
delegate(Example.org, ra, Sales)
  ra=(CBPMS, Sales, community actions)
```

We note that the order of these invocations is important, as a community cannot delegate authority to a child community unless it has the authority itself. Thus, for example, we need to delegate the authority for the IT community to it before it delegates authority to Team1 in that community. Having carried out these delegations, each community has authority to define policies for itself and its children which will apply to the members of the community. However, in order to exercise this authority, we will need to add policies about policy creation to each of these communities, in order to bootstrap the system as before. We also note that the authority that has been delegated to each of the communities includes the authority to take federal actions (Extend, Federate, Withdraw) even
though a requirement of the organization is that no community should invoke these primitives. This is not a problem, since we have already defined a policy at the root community level forbidding all members of the organization, in all circumstances from invoking these primitives, thus, even if any community does define a policy authorizing its members to invoke one of these primitives, it will be over-ridden by the policy in the root community, indeed as an applicable policy will be found in the root community, communities further down the hierarchy will not even be searched.

F. Step 5: Define policies for self-management

Although all of the communities now possess authority over the resource representing the community structure itself, there are no policies defined for them which map this authority to the individual members of the communities. Therefore, before we can complete the self-management configuration, we need to establish these policies. We view this as the community deciding what rules must be followed in order for the community to make a decision – a sort of meta-decision making. The best way to do this largely depends on how the communities within the organization are structured. We introduce here three different possibilities – which we call the hierarchical, the technocratic and the democratic.

In a typical organization it is possible to identify many different hierarchies [moffett]. In addition to the functional hierarchy upon which we have based our community model, a typical organization, such as a corporation, has a decision-making hierarchy. If this decision making hierarchy is well tailored to the needs of the organization it should, to a great extent, mirror the functional hierarchy. Thus, the individual or group who is at the top of the decision making hierarchy makes decisions on behalf of the entire organization – which is the organization’s root community in our model. Similarly, the individuals or groups that are beneath the top level of the decision making hierarchy should make decisions on behalf of the major functional divisions within the organization as depicted in our community hierarchy. In order to implement this we need to allocate decision making over each community to the appropriate person or group in the decision making hierarchy. There are two ways that we could achieve this. If the decision making entity is an individual as, for example, a CEO might be with respect to certain organizations, we can introduce here three different possibilities – which we call the hierarchical, the technocratic and the democratic.

In a typical organization it is possible to identify many different hierarchies [moffett]. In addition to the functional hierarchy upon which we have based our community model, a typical organization, such as a corporation, has a decision-making hierarchy. If this decision making hierarchy is well tailored to the needs of the organization it should, to a great extent, mirror the functional hierarchy. Thus, the individual or group who is at the top of the decision making hierarchy makes decisions on behalf of the entire organization – which is the organization’s root community in our model. Similarly, the individuals or groups that are beneath the top level of the decision making hierarchy should make decisions on behalf of the major functional divisions within the organization as depicted in our community hierarchy. In order to implement this we need to allocate decision making over each community to the appropriate person or group in the decision making hierarchy. There are two ways that we could achieve this. If the decision making entity is an individual as, for example, a CEO might be with respect to certain organizations, we can simply write a policy that authorizes this individual to define policies for the community. Thus for example, we could invoke the policy primitive as follows:

\[
\text{policy}(\text{Example.org,p1}) \ :
\text{p1} = (\text{r1,p1}),
\text{r1} = (\text{CBPMS, Example.org, policy}),
\text{p1} = (\text{auth+ enablePolicyDefinition (when user.id = CEO;)});
\]

\[
\text{policy}(\text{IT,p2}) : 
\text{p2} = (\text{r2,p2}),
\text{r2} = (\text{CBPMS, IT, policy}),
\text{p2} = (\text{auth+ enablePolicyDefinition (when user.id = CTO;)});
\]

These rules would have the effect of authorizing the CEO to define policy on behalf of the entire organization – which would apply to all members of the organization, and the CTO to define policy on behalf of the IT community, which would apply to members of the IT community.

However, it is rarely the case in practice that a single individual is at liberty to define arbitrary policy on behalf of an entire organization or even an entire division. Most organizations are structured in such a way that the individuals at the top of the organizational hierarchy are executive officers, rather than policy makers and the individuals are often constrained in their actions by groups, such as boards of directors, which define the boundaries within which they may operate. Thus, it is generally more useful to create a new decision making community as a sub-community of the community which it will be making decisions on behalf of and delegate the required authority to it.

Another good reason for relying on communities for situations like this, rather than using conditions in policy specifications to identify subsets of the community members to which policies apply, concerns the efficiency of the policy search algorithm. A community policy which uses a constraint in a policy specification will have to be evaluated for each and every policy decision request to that community that is within scope in order to establish whether it is applicable or not. By contrast, if we create a new community and specify the policy rule in that community, we will only need to evaluate the policy specification for members of that new community, which will always be less or equal than the membership of the parent community and hence we will reduce the number of policy specifications that need to be evaluated for many requests. The payoff is that we will need to evaluate the membership policy specification of the new community for each new request. Thus, if there is one single policy in a community that applies to a subset of the community members, there is no net advantage of forming a new sub-community. However, as soon as there is more than one such community policy, we gain an immediate gain in efficiency of our policy search algorithm overall. This is a good rule of thumb to apply when deciding whether to specify a subset of users with a policy specification constraint (if the policy language allows it) or to form a new sub-community.

G. Control Communities

In the case of this example, we can form decision making sub-communities, which we term control communities (as they control their parent community) for each community in the hierarchy and define membership policies for these sub-communities which limit their membership to the appropriate members of the organization. In order to do this, we need to create the decision making sub-communities and delegate to them the appropriate resources for the communities which they will make decisions on behalf of.

\[
\text{spawn}(\text{Example.org, board})
\]

\[
\text{delegate}(\text{board},
\quad (\text{CBPMS, Example.org, Community Actions}))
\]

\[
\text{spawn (IT, Technical Committee)}
\]

\[
\text{delegate}(\text{Technical Committee,}
\]

\[
\]
And so on for each community in the hierarchy that contains a decision making sub-community within it. This gives us the community hierarchy shown in figure 3. This is a typical structure of a hierarchical organization. Each functional unit has an associated decision making group whose authority is limited to the functional unit over which it presides. However, we face the same problem as before – we need to establish policies, which translate the control community authority into policies that apply to individuals. In some cases we can simply define policy rules which authorize any member of the decision making group to take an action on behalf of the community. So, for example, we might define a policy which authorizes any board member to invoke the policy and revoke primitives when a system emergency has been detected. However, in general, when we are dealing with the CBPMS resource, these control communities will require some sort of collective approval before primitives are invoked. For example, the organization might require a majority vote of board members before changing policy across the whole organization.

![Figure 3. Community Structure for Self-Management](image)

**H. Collective Decision Making**

A detailed discussion of collective decision making mechanisms and the various voting and approval systems in existence is beyond the scope of this paper. However, there is a large volume of published material on the matter [ferscha][nurmi83][nurmi87] and a wide variety of systems available which implement various collective decision making strategies such as the Dodgson rule, the majority rule, plurality voting, Borda counting [ferscha], approval voting [brahms], the Nanson rule the Hare rule, the Coomb rule [ferscha] and others. Whichever particular system, strategy or set of strategies is used is not important from the point of view of the CBPMS. All that is of concern to us is that a collective decision making system be available to the community and that we can engineer it to obey certain semantics required to integrate it into the CBMPS.

Consider a simple collective decision making system which supports several different types of decision making strategy. For example our system might support the following strategies, major vote – where the winner is the majority of votes cast after a certain amount of time, at least x votes for and at most y votes against where a binary decision is deemed passed if the formula is met for a specified x and y after a certain amount of time, x approvals, where a proposal is deemed passed if it receives x proposals within a certain amount of time. In order to integrate policies which require collective decision making procedures to have been completed before an action is authorized, we need to be able to express the information in the policy specification language. Exactly the best way to do this depends upon the flexibility of the policy specification language. However, in general, it amounts to a similar process. We create an authorization policy for the relevant resource authority and attach a condition to the policy specification which signifies “only authorized after decision making strategy x has approved it”. This condition could be as simple as a check to see if a certificate issued by the decision making system is present in the context of the policy request, leaving the work of populating this context to other parts of the system, or it could be integrated into the policy specification language. So, for example, we could define a Ponder constraint that looked something like the following in order to associate certain actions on certain targets with required decision making procedures:

```
when inst auth+ { when vote(majority, context.policy-request) }
```

to specify a majority decision is required or:

```
when inst auth+ { when vote(approval(2), context.policy-request) }
```

to specify that the action requires two approvals before it will be authorized. Being able to specify voting rules within policy specifications in such a manner is dependant on the policy specification language being expressive enough to implement the following. When a resource authority arrives that is within the scope of this policy, the evaluation of the condition will invoke a decision making process that will be associated with the policy decision request (derived from the context). Only after this decision making process has completed will the action be authorized or not depending on the result of the decision making procedure.

We will assume that we have such a facility available within the policy specification language and PRS. Thus, to return to the discussion of the control communities exercising their authority, we can now specify the following policies for the board control community:

```
policy(board, p) :
p  = {ra, ps},  
  rai=(CBPMS, Example.org, Community Policy Actions),  
  ps=(auth+ majorityVote  
   {when vote(majority, context.policy-request )})
```

This policy enables the board to take any community policy action on any target community beneath the Example.org community in the community hierarchy after a majority vote of the board. Whenever a board member proposes an action with the community policy management tool, the tool will send a
policy request to the CBPMS which will cause this policy to be evaluated and a voting procedure to be initiated. If the voting procedure is successful, the action will be authorized. We can now apply a similar policy to the control community’s of each community in the original hierarchy enabling each community to be managed by majority vote, or we could apply different voting rules to different communities as required. Indeed, if we want to implement an entirely democratic decision making process we can eliminate the control communities throughout the hierarchy and apply these policies to the original communities themselves – thereby making the entire organization democratic rather than hierarchical.

However, one immediate problem which arises is that, although the board of directors, or the entire membership, may have authority to define the organization’s policies, they are unlikely to always the individuals who actually use the policy management tool in order to author the policies. Instead, it is likely that, in many situations, specialists who are familiar with the tool will translate the high level policy decisions of the decision makers into specific community actions and specific policy definitions. We can easily incorporate this into our model by creating further control communities to control the first layer of control communities and populating these communities with the technical specialists who will implement the decisions of the control community within the system itself. We refer to this configuration as a technocracy, particularly when the technical sub-communities control the community representing the functional unit rather than the control community for the unit.

The final step is to remove the policies that we put in place to allow the architect to build the system leaving the system entirely self-managed by the communities within it:

\[
\text{revolve(Example.org, p.initial)}
\]

I. Mandates and Limited Autonomy

We have illustrated how to construct a self-managed CBPMS system, where the CBPMS controls access to its own primitive operations and the communities within it control their own decision making, whether that be through a democratic, hierarchical or technocratic paradigm. However, due to the nature of the target tree, the root community has authority over all of the communities beneath it and it can decide to define a policy for any sub-community in the system, regardless of the wishes of that sub-community. This allows the organization to be structured in such a way that each community acts with strictly bounded authority. In the case where authority is bounded in such a manner over the community resource itself, we refer to this as giving a community limited autonomy. Its parent communities can define whatever policies they want and the community is bound to follow these policies. However, the community can also define its own policies and invoke other community primitives to create sub-communities of itself and delegate authority to them and so on. Thus each community can exercise total control over its sub-communities, by specifying a restrictive policy set for the community, or can allow the community a great deal of autonomy, by allowing it to decide what policies it will define for its own invocations of the CBPMS primitives.

We describe this type of policy arrangement, where a parent community defines the policies of its sub-communities, as a mandate. So, for example, the IT community can decide to specify a policy for the Team 1 community forbidding it from spawning further communities, but otherwise it will allow the community to define its own policies. Thus, after the IT community, through its control community, has come to a decision to do so, it will invoke the following primitive:

\[
\begin{align*}
\text{policy(\textbf{Team1}, p.mandate)} : \\
p.mandate &= (ra, ps), \\
ra &= (\text{CBPMS, Team1, spawn}), \\
ps &= \text{auth-} \{\text{when true}\}
\end{align*}
\]

However, there is a problem with this arrangement. The Team1 community has authority to revoke policies within its own community policy set and can thus simply remove the mandate that has been specified by its parent. To solve this problem the IT community simply specifies a rule in its own policy set which prevents the mandated policy from being removed.

\[
\begin{align*}
\text{policy(\textbf{IT}, p.guard)} : \\
p.guard &= (ra, ps), \\
ra &= (\text{CBPMS, Team1, revoke}), \\
ps &= \text{auth-} \{\text{when policy = p.mandate}\}
\end{align*}
\]

Since the IT community is higher in the hierarchy of authority than the Team 1 community, this rule will always have precedence over any rules that might be defined by the Team 1 community. Thus, the mandated policy, p.mandate, can not be revoked until the IT community decides to revoke this guard policy, p.guard, or the root community defines a policy which overrides that policy in turn. The Team1 community is bound to obey it. In general, the invocation of these two primitives in such a manner is what we mean when we say that a community defines a mandate for a community further down the functional hierarchy.

J. Centralized Organization

We refer to this model of the hierarchy of authority as a centralized organization. The root resource authority for the CBPMS resource is possessed by the root community and all the other communities are beneath it in the target tree. Hence it can always decide, through its decision making mechanisms to define policies on behalf of any other community. In general, this is the authority model that many organizations, such as corporations, use in practice. While the high level decision makers may delegate self-management authority and local autonomy to lower level units in the organization, they reserve the right to impose rules directly on any area of the organization’s operations. For example, if a decision maker lower down the organization makes a decision that the CEO disagrees with, it is generally the case that the organizational
model will allow the CEO to override the decision.

K. Autonomous Communities – hybrid organization

However, there are several ways in which we could change this feature of the system in order to limit the authority of each community to the community itself and allow sub-communities total autonomy from their parent communities. The easiest way to do this is by simply attaching a condition to the policy rules that authorize the use of community actions. Thus, instead of defining the following policy for the Example.org community:

\[
\text{policy(board, p) : p = (ra, ps),} \\
\text{ra = (CBPMS, Example.org, Community Policy Actions),} \\
\text{ps = (auth+ majorityVote}} \\
\text{ {when vote(majority, context.policy-request)}}
\]

It is defined as:

\[
\text{policy(board, p) :} \\
\text{ra = (CBPMS, Example.org, Community Policy Actions),} \\
\text{ps = (auth+ majorityVote}} \\
\text{ {when context.policy-request.target = Example.org && vote(majority, context.policy-request)}} \\
\text{ p = (ra, ps)}
\]

Thus limiting the authorization of the Example.org decision makers to invoke the community primitives on the Example.org community itself. Similarly, any community can grant its sub-communities full autonomy by limiting the target of community policy actions to the community itself. However, what can be given can be taken away, and a community that defines this autonomy clause in its policy set is free to add another policy which authorizes it to take actions on communities further down the target tree. The autonomy provided by this solution can be revoked at any time by the community that defined it.

I. IMPLEMENTATION AND EVALUATION

The implementation of a CBPM System (CBPMS) is structured as in figure 4. In this paper we focus on how the primitives offered by the Community Record Management Service (see Table A) can be used to enable self-management within an organization. This involves interaction with the system from the Policy Management Tool, a front end consisting of web-based forms in our current implementation. It also involves the Policy Reasoning Service (PRS) which operates within the Community Policy Decision Service, evaluating policy decision requests from the policy decision consumer.

Initial validation have been through integration of CBPMS with a collaborative web portal for the open source community [feeney04], a location-aware instant messaging application in a ubiquitous computing scenario [lewis04][feeney05] and providing dynamic spectrum access policies to a cognitive radio systems [lewis06].

II. CONCLUSION

This work presents an initial step to enabling the policy-based self-management of IT resources to be undertaken by units across an organization through the controlled delegation of authority. This is not only important for agile organizations where team autonomy and innovation are needed, but also in federated organizations that rely on regulated access to shared resources. This is a relatively unexplored aspect of self-management in IT.

ACKNOWLEDGMENT

The authors would like to thanks Declan O’Sullivan and John Keeney for their valuable assistance in preparing this paper.

REFERENCES


driver] Driver, E., IBM Delivers on its vision of Role-based workplaces, Forrester, 12 April 2006

[feeney04] K. Feeney, D. Lewis, V. Wade Policy Based Management for Internet Communities, , in Proc of IEEE 5th International Workshop on Policies for Distributed Systems and Networks (Policy 2004), June 8th-9th 2004, IBM Thomas J Watson Research Center, New York, USA, pp 23-34


[sox] Sarbanes–Oxley Act in the US, to be found at: http://www.legalarchiver.org/soa.htm

