Generic Support for Policy-Based Self-Adaptive Systems

Richard John Anthony
Department of Computer Science, University of Greenwich, UK
R.J.Anthony@gre.ac.uk

Abstract

This paper presents a policy definition language which forms part of a generic policy toolkit for autonomic computing systems in which the policies themselves can be modified dynamically and automatically. Targeted enhancements to the current state of practice include: policy self-adaptation where the policy itself is dynamically modified to match environmental conditions; improved support for non autonomics-expert developers; and facilitating easy deployment of adaptive policies into legacy code.

The policy definition language permits powerful expression of self-managing behaviours and facilitates a diverse policy behaviour space. Features include support for multiple versions of a given policy type, multiple configuration templates, and meta-policies to dynamically select between policy instances.

An example deployment scenario illustrates advanced functionality in the context of a multi-policy stock trading system which is sensitive to environmental volatility.

1. Introduction and background

Self-adaptive behaviour can be achieved by embedding a policy which itself is static at run-time, for example, it may just provide operational rules or parameterisation to set bounds of behaviour. In such approaches policy changes (for example to achieve long-term optimisation or to resolve rule conflicts), require open-loop adaptation in which inefficiencies or conflicts are identified and fixed manually, or are identified automatically but the solutions require human mediation.

There is a limit to the effectiveness of a system that has a fixed policy. In dynamic environments it is often necessary to not only adapt the controlled system, but also for the policy to self-modify its own behaviour to better achieve the system's goals. Fixed rules are likely to be sub-optimal over at least part of the system behaviour space. However, if behavioural trends are analysed, it may be possible to gradually tune the rules dynamically to better reflect the needs of the specific system, taking into account its current environment and context. If the policy configuration can also be persisted between executions, then longer-term adaptation could be achieved.

There are three distinct levels of sophistication found in current schemes: 1. In the simplest approach the policy rules are statically embedded. The template configuration is exposed and can be modified between executions. Examples are found in [1, 2]. [3] Embeds fixed rules into agents. [4] Provides an example where the policy mechanism is internally sophisticated, embedding utility functions which achieve dynamic self-configuration, although the actual policy configuration remains fixed. 2. Open-loop policy updates. An external entity (usually a human) identifies potential configuration optimisations, which can be applied *between* executions by modification of the template or direct manipulation of policy rules. An example of this configuration is IBM Research's Policy Management for Autonomic Computing (PMAC) [5], in which the policy mechanism is maintained externally to the run-time system and thus policy changes can be made without changing the application code. 3. Closed-loop adaptation, where the policy dynamically and automatically modifies its own rule-base or template settings during execution. A rule-based system for application configuration is described in [6], in which rules are statically assigned either high or low priority. In this system the dynamic adaptation is in the form of automated conflict resolution. Where there is a conflict between low priority rules, dynamic resolution is performed by using cost functions to select the most appropriate action.

A number of systems have employed short-term self-adaptation of policy (in which changes are volatile). Examples are found in [7], in which event-trigger conditions are dynamic, and [8] in which conflicts between the obligations of security policies are automatically detected and resolved at run-time.

Systems that have a wide potential behaviour space, with many dimensions of freedom, do not lend themselves well to governance by a single policy. In such cases it may be more appropriate to have a collection of policies and to use the most appropriate one for the given ambient conditions. A meta-policy can be used to make this selection. In [9] for example, administrators can view

Proceedings of the 17th International Conference on Database and Expert Systems Applications (DEXA'06)
0-7695-2641-1/06 $20.00 © 2006 IEEE
IEEE COMPUTER SOCIETY

and update externally visible security policies at run-time. A meta-policy is used to dynamically select between policy versions.

An earlier example of policy self-modification behaviour is provided in [10] in which internal thresholds and other configuration is changed dynamically to reflect environmental conditions. However, that was initial work in the author's current policy-library project, and the adaptation was limited to short-term changes that were not persisted between executions.

The remainder of this paper is concerned with future policy-based systems in which policies can modify their own behaviour as well as adapting the controlled system. From here on, the term 'policy' refers to a structured set of rules and actions that govern the behaviour of some aspect of an application's run-time behaviour. The implementation of such a policy system requires that several aspects can be dynamically tuned, including the initial configuration template settings and the actual parameterisation of each rule. The action carried out as a result of executing a rule is permitted to include policy-updating statements that change the way in which the same, or another, rule behaves in the future.

Also envisioned are systems that embed a *suite* of policies and use a meta-policy to dynamically select the most appropriate one for the ambient environmental or contextual conditions. In these systems it will be possible for several dimensions of adaptation to occur concurrently, from fine-grained policy-static control to medium-grained optimisation achieved through policy self-modification, to coarse-grained behavioural shifts achieved through automated switching between different policy instances.

2. A policy definition language

Several languages have been devised to permit specification of policy rules, including TPL [11] and Ponder [12]. The eXtensible Access Control Language (XACML) [13] includes a query protocol to examine policies and determine whether a particular access should be allowed. Some languages have additional features such as the automatic detection and resolution of rule conflicts, see for example [14].

The proposed language extends the state of the art in policy languages in several ways. It explicitly supports dynamic self-modification of policies over both short and long term through persisted configuration changes. Also supported are policy suites, in which a particular type of policy can have many differently-geared instances (for example cautious and aggressive versions). Suites of templates for a given policy-type allow different initial configurations to be used, depending on start-up circumstances. Meta-policies are also supported. These can be used to select amongst many policy-instances and templates at initiation time, and can also be used to automatically hot-swap between instances of the same type of policy should the environmental conditions or context change significantly. The language also incorporates features to support bounded behaviour, enhance stability and facilitate policy-object reuse. Policies, meta-policies and initial templates can be simply and unambiguously written using straightforward syntax and type-safe semantics.

The language is generic, in the sense that is capable of describing a very-wide space of policies for a very diverse set of application domains. This is achieved through using syntax and structure which is simple yet flexible and expressive. The language has a number of novel features:

- The language is object oriented. Different objects represent policies, rules, actions etc. The object approach facilitates re-use of behaviour, through reusing tested objects. For example a new policy can share some of the rules and actions of existing policies. This reduces (policy) development and testing effort and enhances reliability.
- The flexibility and powerfully expressive nature of the language stems from its hierarchical support for effectively three categories of policy: 1. Templates provide configuration parameters (in some schemes this is the extent of the 'policy'). These are used to initialise the other categories of policy - Normal Policies (NP) and Meta-Policies (MP). 2. NPs are those which contain the low-level autonomics business logic of applications (for example to achieve self-optimisation or self-protection). 3. MPs can be used to provide higher-level adaptation (typically by selecting the most appropriate NP for the prevailing circumstances, and/or selecting the most appropriate template with which to configure the policy). MPs can also be configured by a user-supplied template. A single policy script may contain all three categories of policy.
- MPs can either perform an initial configuration, or can operate continuously. In this latter mode, MPs support 'hot-swapping' between NP instances.
- The language reinforces the natural semantic differences between variables that are used to convey external information to the policy (environmental and contextual state) and those that are used to maintain internal policy state (such as counts, flags and thresholds). The values of External Variables (EV) represent the dynamic context in which the policy executes and therefore must be passed in (e.g. from sensors) *each* time the policy is fired. Thus EVs must not be modified by the policy, and there is no reason to persist the values of EVs, or to include them as part of a policy configuration template. The Internal Variables (IV) are part of the current configuration of the policy. As such it is important that IVs can be

IEEE COMPUTER SOCIETY

updated and their values persisted to enable longer-term adaptation. The separation of variables into two classes simplifies policy writing and debugging, and reinforces type-safety and semantic checking. For example, an EV may only occur on the RHS of an assignment, whereas an IV may be placed on either side.

- Template and run-time configuration of IVs supports specification of various attributes, such as value ranges (upper and lower limits) and increment / decrement amounts. For example when specifying a timer value in milliseconds, it may be desirable to set the increment amount to 100. In this way each time the policy increments the variable it actually adds 100 ms to the timer value. Such techniques can greatly simplify policy writing and reduce the occurrence of errors.
- The policy mechanisms are truly self-adaptive. At run-time the IVs can be dynamically updated. EVs change according to ambient conditions. The language structure makes it possible for rule execution to be influenced (ordering, omission or inclusion of specific rules) by the values of either IVs or EVs.
- To promote and enhance stability, a policy language object *ToleranceRangeCheck* (TRC) is provided to facilitate simple dynamically-configurable specification of dead-zones; which help to avoid oscillation. This language object replaces at least two rules and two threshold variables that would otherwise be needed to configure a dead-zone.
- Policy scripts are formatted in XML which enforces a standard general syntax and facilitates the deployment of policies in heterogeneous systems. The various objects of each type are grouped together in the script; i.e. the policy is not written in a procedural format as with for example pseudo-code and most other policy script-languages. The object format simplifies parsing and syntactic checking.
- A policy library implementation further complements the language by providing implicit support mechanisms such as long-term state persistence and library interface mechanisms that are easy to deploy into legacy code.

The policy language comprises several object types, the semantics of which are described in turn:

ExternalVariable: Representation of environmental or contextual conditions. Passed in to the policy at the point of policy evaluation.
InternalVariable: Used internally by a policy (typically counters, flags and thresholds).
Each class of variable can be of three basic types: Long, Boolean and PolicyName, and strong validation is performed (for example assignment requires similar types for the LHS and RHS variables). PolicyName variables are only used in MPs.
Template: A set of configuration statements that apply to a particular policy-type. The configuration for each variable can include attributes such as maximum and minimum values.
ReturnValue: Numerical return codes are mapped onto named values for use within the policy script.
Rule: A statement that can evaluate to either true or false. Rules have separate Actions for the *evaluate true* and *evaluate false* cases.
ToleranceRangeCheck: A specialized rule used to implement fuzzy variable comparison and dead-zones.
Action: A grouped sequence of activities that occur when a rule or TRC evaluates to either true or false.
Policy: A sequence of Rules and TRCs. An MP is a special policy that can be used to dynamically select the current NP and template configuration.
PolicySuite: A collection of policies of the same type, i.e. concerned with the same aspect of business logic. Dynamic selection between NPs within a suite can be mediated by an MP, based on environmental and contextual influences and recent behaviour history. For example if the current adaptation is too slow for the ambient conditions, a 'cautious' policy might need to be replaced by a more 'aggressive' policy.

The language grammar is formalised using EBNF notation:

Non Terminals:

E	ExternalVariable	**I**	InternalVariable	**A**	Action
T	Template	**N**	returNvalue	**R**	Rule
C	ToleranceRangeCheck	**S**	PolicySuite	**P**	Policy

Terminals:
number: constants used in rules and assignments, and numerical return codes.
'true' and 'false': when assigning or comparing boolean variables, and in rules.
'Null': used in Rules and TRCs when no action is required in either branch.
'EQ', 'NE', 'GT', 'LT', 'GE', 'LE': Operator values, used in Rules.

Attributes:

PolicySuite:	Name
Policy:	Name, Type {MetaPolicy, NormalPolicy}
Rule:	Name, LHS, Operator, RHS, ActionIfTrue, ElseAction
ToleranceRangeCheck:	Name, CheckVariable, CompareAgainstVariable, ToleranceRangePercentSpecifier, ActionIfInZone, ActionIfOutsideZone
Action:	Name
ReturnValue:	Name, Value
ExternalVariable:	Name, Type {Long, Boolean, PolicyName}
InternalVariable:	Name, Type {Long, Boolean, PolicyName}, InitialValue, MinValueValid {true, false}, MinValue, MaxValueValid {true, false}, MaxValue, IncrementAmount
Template:	Name

Production rules:
E: true | false | number **I**: true | false | number **P**: [T] (R | C)+
T: Assignment+ **N**: number **S**: P+
A: {Assignment | Increment | Decrement}+ [R | C | N]
R: if ((E | I) Operator (E | I)) then (A | Null) else (A | Null)
C: if ((E | I) in-range-of (E | I) where-range-specified-by (E | I)) then (A | Null) else (A | Null)
Operator: {EQ, NE, GT, LT, GE, LE}
Assignment: I = (I | E | number | true | false)
Increment: $I = I + I_{attribute_Increment_Value}$
Decrement: $I = I - I_{attribute_Increment_Value}$

The Action production rules ensure that an Action can comprise many Assignment, Increment or Decrement

IEEE COMPUTER SOCIETY

statements, in any order, but can contain a maximum of one of either a Rule, TRC or ReturnValue, which if present, must be the last action statement. A Rule or TRC can conditionally invoke further Actions.

3. A case example

A multi-policy stock trading scenario is used as a vehicle to illustrate the flexibility afforded by the use of MPs to perform higher-level configuration choices; and the ability to dynamically switch between policy instances. The language's support for these features is demonstrated.

The stock trading system is representative of many real-world problems that have highly complex behaviour and have dissimilar sensitivities to several sources of environmental volatility. The system has many dimensions of freedom and a very wide behaviour space. Tracking the fluctuations in stock prices, and making trading decisions (buy, sell, hold), is subject to influences which include: recent and longer-term trends in price behaviour, trading volumes, the *rate of change* in traded volumes and the *rate of change* in price. It is not desirable to closely track such a system over the entire behaviour range with a single policy, because the system is non-linear in its sensitivity to the various environmental parameters. For example, bolder decisions are typically made when the rate of price change is low because there is less risk. Conversely, when the price is less stable the policy must make more cautious decisions to reflect the greater risk. In such a scenario, the use of a single policy could lead to significant sub-optimality across a wide spectrum of behaviours. Also, a policy that could cope with all conditions would itself be a source of considerable complexity and thus risk. One way to resolve this problem is to divide the application behaviour space into several zones, as shown in Figure 1. For the purpose of simple illustration the example uses only two zones per dimension of behavioural freedom. The actual number of zones required is a function of the extent of non-linearity in a particular application domain.

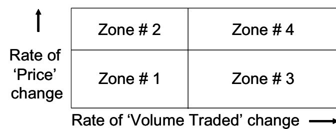

Figure 1. Segmentation of application behaviour space along two dimensions of freedom, creating four zones.

A policy is devised for each zone. Each policy is thus tuned specifically for optimal operation over a subset of the application behaviour space. This facilitates a possible solution to non-linear sensitivity to environmental conditions. Following the price-rate-change example; certain rules that work well when the stock price changes gradually might be totally inappropriate in more-volatile conditions when sudden fluctuations are encountered. The zoning yields numerous individually-simple policies (relatively) in place of one large unwieldy policy. For example, if the zones are chosen appropriately it might be possible to approximate a complex non-linear global relationship with a series of simpler, (possibly) linear rules.

The four policies have different configurations but are of the same *type* because they both address the same business logic decision (although they arrive at their decisions differently).

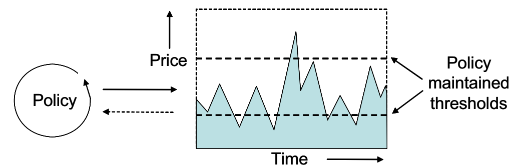

Figure 2. A policy operating within its zone.

Each policy is configured such that its operational envelope (that over which it guarantees safe and desirable behaviour) maps closely onto its zone. A self-adaptive policy may adjust its own thresholds over time, as depicted in figure 2, or these might be supplied externally. Individual policies need not be aware of the delimitation of their zones (which might even be dynamically variable). An MP is employed to select between the different business policies. The MP must monitor the behaviour of the target system and determine which policy should be employed at any given moment. This behaviour is depicted in figure 3. The policy tools described in this paper support dynamic switching between policies.

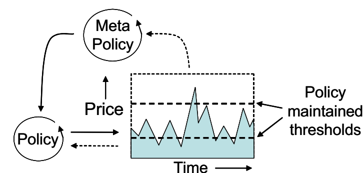

Figure 3. The meta-policy monitors the target system and selects an appropriate policy for the ambient conditions.

However, there is a risk of instability if simple cut-off points are used to determine the policy selection. If the behaviour of the monitored system is close to a zone boundary it is possible that small changes in target system behaviour could lead to oscillatory switching to and fro between policies, causing extra work for the system. To avoid this, the policy tools directly support stability through the use of dead-zones, as illustrated in figure 4.

IEEE COMPUTER SOCIETY

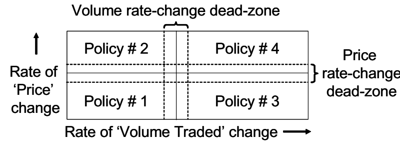

Figure 4. Policy zone boundaries are wrapped by dead-zones to avoid oscillatory switching by the meta-policy when the behaviour of the monitored system is 'marginal'.

The MP takes no action while the system is in a dead-zone. A significant shift in behaviour is needed to cross the dead-zone in one go. Localised oscillatory behaviour in the target system is not mirrored in the behaviour of the MP.

Dead-zones are identified by two parameters: the centre-point, and the width of the zone (expressed as a percentage deviation from the centre-point). These parameters are maintained within the policy as IVs so that they can be dynamically modified.

The overall behaviour exhibits three dimensions of adaptability: 1. Low-level changes enforced by the operation of a particular business policy; 2. Automatic meta-policy switching between business policies to ensure that the *CurrentPolicy* always remains within its optimal operational envelope; 3. Dynamic adjustments to configuration parameters, including those that define the position and width of the dead-zones. The dead-zones ensure stability despite the high extent of adaptability. Each of these dimensions of adaptability has the potential to operate at the level of adapting the target system, as well as at the level of modifying the policy system itself (i.e. the controlling system).

Figure 5 presents the XML policy script for the stock-trading example illustrating how the language's simple yet powerful syntax and object-oriented semantics support the expression of sophisticated policy logic. Due to limited space, the example focuses on the meta-policy and its dynamic selection of the active business policies. The actual details of the four business policies are not shown in the illustration.

The example illustrates several novel features of the language. Multiple NP business-logic policies (named Policy1 – 4) are used. Each of these policies makes the same type of business decisions (buy / sell stock etc.), however, each is specifically tuned for operation over a specific zone of application behaviour space. An MP (named Meta_Policy) is responsible for dynamically selecting which of the business policies should be executed at any given moment, based on the current behavioural zone of the system which the MP determines from the values of two EVs. The current business policy selection is identified by the special variable *CurrentPolicy*. A template is used to configure the MP upon initiation, setting the *CurrentPolicy* variable to point to Policy1. Two dead-zones are implemented through the use of TRC objects. The deadzones are initially configured by the MP template and prevent excessive swapping between business policy instances when the target system's behaviour loiters close to a boundary between zones.

```
!-- Policy Definition XML file: Policy Language version 1.0 -->
<!-- Application: Multi-Policy Stock-Trading illustration -->
<PolicyConfiguration> <!-- PolicyTypeName= Stock-Trading Policy -->
  <EnvironmentVariables>
    <Variable Name="E_iCurrentPriceRateChange" Type="long"/>
    <Variable Name="E_iCurrentVolumeRateChange" Type="long"/>
    <!-- Details of EnvironmentVariables used by NormalPolicies not shown -->
  </EnvironmentVariables>
  <InternalVariables>
    <Variable Name="I_iPriceRateChangeDeadZone" Type="long"/>
    <Variable Name="I_iVolumeRateChangeDeadZone" Type="long"/>
    <Variable Name="I_iDeadZoneTolerancePercent" Type="long"/>
    <!-- Details of InternalVariables used by NormalPolicies not shown -->
  </InternalVariables>
  <Templates>
    <Template Name="MetaTemplate">
      <Assign Variable="I_pCurrentPolicy" InitialValue="Policy1"/>
      <Assign Variable="I_iPriceRateChangeDeadZone" InitialValue="60"/>
      <Assign Variable="I_iVolumeRateChangeDeadZone" InitialValue="50"/>
      <Assign Variable="I_iDeadZoneTolerancePercent" InitialValue="10"/>
    </Template>
    <!-- Details of templates used by NormalPolicies not shown -->
  </Templates>
  <ReturnValues> <!-- Details of ReturnValues not shown --> </ReturnValues>
  <Actions>
    <Action Name="A_DeterminePolicyBasedOnPriceRateChange">
      <Evaluate Rule="R_CurrentPriceRateChange"/>
    </Action>
    <Action Name="A_DeterminePolicyBasedOnVolumeRateChange">
      <Evaluate Rule="R_CurrentVolumeRateChange"/>
    </Action>
    <Action Name="A_SelectLowPriceRateChangePolicy"> <!-- . . . --></Action>
    <Action Name="A_SelectHighPriceRateChangePolicy"> <!-- . . . --></Action>
    <Action Name="A_SelectLowVolumeRateChangePolicy"> <!-- . . . --></Action>
    <Action Name="A_SelectHighVolumeRateChangePolicy"> <!-- . . . --></Action>
    <!-- Details of Actions used by NormalPolicies not shown -->
  </Actions>
  <Rules>
    <Rule Name="R_CurrentPriceRateChange" LHS="E_iCurrentPriceRateChange"
      Operator="LT" RHS="I_iPriceRateChangeDeadZone"
      ActionIfTrue="A_SelectLowPriceRateChangePolicy"
      ElseAction="A_SelectHighPriceRateChangePolicy"/>
    <Rule Name="R_CurrentVolumeRateChange"
      LHS="E_iCurrentVolumeRateChange" Operator="LT"
      RHS="I_iVolumeRateChangeDeadZone"
      ActionIfTrue="A_SelectLowVolumeRateChangePolicy"
      ElseAction="A_SelectHighVolumeRateChangePolicy"/>
    <!-- Details of Rules used by NormalPolicies not shown -->
  </Rules>
  <ToleranceRangeChecks>
    <ToleranceRangeCheck Name="C_PriceRateChangeDeadZone"
      CheckVariable="E_iCurrentPriceRateChange"
      CompareAgainstVariable="I_iPriceRateChangeDeadZone"
      ToleranceRangePercentSpecifier="I_iDeadZoneTolerancePercent"
      ActionIfInZone="Null"
      ActionIfOutsideZone="A_DeterminePolicyBasedOnPriceRateChange"/>
    <ToleranceRangeCheck Name="C_VolumeRateChangeDeadZone"
      CheckVariable="E_iCurrentVolumeRateChange"
      CompareAgainstVariable="I_iVolumeRateChangeDeadZone"
      ToleranceRangePercentSpecifier="I_iDeadZoneTolerancePercent"
      ActionIfInZone="Null"
      ActionIfOutsideZone="A_DeterminePolicyBasedOnVolumeRateChange"/>
  </ToleranceRangeChecks>
  <Policies>
    <Policy Name="Meta_Policy" PolicyType="MetaPolicy">
      <Initialise CurrentTemplate="MetaTemplate"/>
      <Evaluate ToleranceRangeCheck="C_PriceRateChangeDeadZone"/>
      <Evaluate ToleranceRangeCheck="C_VolumeRateChangeDeadZone"/>
    </Policy>
    <Policy Name="Policy1" PolicyType="NormalPolicy"> <!-- . . . --> </Policy>
    <Policy Name="Policy2" PolicyType="NormalPolicy"> <!-- . . . --> </Policy>
    <Policy Name="Policy3" PolicyType="NormalPolicy"> <!-- . . . --> </Policy>
    <Policy Name="Policy4" PolicyType="NormalPolicy"> <!-- . . . --> </Policy>
  </Policies>
</PolicyConfiguration>
```

Figure 5. The stock-trading multi-policy XML script

One example of object re-use is demonstrated in the form of a variable (*DeadZoneTolerancePercent*) which is shared between the two TRC objects. The IVs (such as the dead-zone specifiers), and EVs (in this case the price

IEEE COMPUTER SOCIETY

rate-change and volume rate-change variables) are clearly separated.

4. Conclusion

A policy definition language has been presented. The language facilitates a very diverse policy behaviour space through both hierarchical and recursive uses of language elements. The object-oriented nature of the language enables highly expressive policy logic using a simple and consistent syntax. In particular it promotes reuse of policy objects. Reusing policy objects represents significant savings in the time and cost associated with policy development and testing.

The object oriented approach allows attributes to be assigned to the various objects (rules, actions, variables etc). The attributes are treated semantically in the same way as IVs and thus facilitate flexible run-time configuration, beyond simply changing the values of variables. For example, the size of a deadzone, or the upper-value-limit for a variable can be changed dynamically. The object attributes are also persisted in the same way as the IVs, as they form part of the current configuration state of a policy.

Policy configuration state is persisted in well-formed XML script, which promotes interoperation in heterogeneous environments. Innovations include support for multiple policy versions of a given policy type, multiple configuration templates, and MPs to dynamically select between policy instances and templates. This use of MPs represents a meta-state transition in the evolution of policy-based computing; bringing far greater flexibility and a hierarchical aspect that helps control complexity. A large monolithic policy is replaced with a suite of simpler, more-highly-tuned, policies with limited operational envelopes and selection between these is controlled by a higher-layer policy which may also be self-modifying.

The language, together with its library implementation, is intended to facilitate adaptive-policy deployment in designed-in circumstances, as well as retro-fitting self-management into legacy code. This is a very important issue because there are a great many applications in current use that are in urgent need of self-management, but complete re-development is ruled out due to costs and operational logistics.

Whilst it is accepted that the policy tools proposed in this paper are not yet fully mature, their inbuilt scalability facilitates a developmental bridge allowing self-management to be embedded in a piecemeal fashion. For example a simple, single policy and template can be created initially, providing limited adaptability, but quick to deploy. It is possible to subsequently expand to several policies and / or templates, and to introduce a meta-policy to mediate dynamically.

An example deployment scenario has been presented, providing an illustration of how meta-policy mediated policy hot-swapping can facilitate highly-optimised, hierarchical self-management using multiple, individually non-complex, policies.

References

[1] K. Phanse, L. DaSilva, and S. Midkiff, "Design and demonstration of policy-based management in a multi-hop ad hoc network", *Ad Hoc Networks*, 3(2005), Elsevier B.V., 2005, pp. 389-401.
[2] E. Terzi, A. Vakali, and L. Angelis, "A simulated annealing approach for multimedia data placement", *The Journal of Systems and Software*, 73, Elsevier Inc, 2004, pp. 467-480.
[3] R. Basra, K. Lu, G. Rzevski, and P. Skobelev, "Resolving Scheduling Issues of the London Underground Using a Multi-agent System", *2nd International Conference on Industrial Applications of Holonic and Multi-Agent Systems (HoloMAS)*, Copenhagen, Denmark, LNAI 3593, Springer-Verlag, 2005, pp. 188-196.
[4] V. Kumar, B. Cooper, and K. Schwan, "Distributed Stream Management using Utility-Driven Self-Adaptive Middleware", *proceedings of the 2nd International Conference on Autonomic Computing* (*ICAC*), IEEE, Seattle, 2005, pp. 3-14.
[5] IBM Research, Policy technologies. *http://www.research.ibm.com/policytechnologies/*.
[6] H. Liu, and M. Parashar, "A Framework for Rule-Based Management of Parallel Scientific Applications", *proceedings of the 2nd International Conference on Autonomic Computing* (*ICAC*), IEEE, Seattle, 2005, pp. 360-361.
[7] L. Lymberopoulos, E. Lupu, and M. Sloman, "An adaptive policy based management framework for differentiated services networks". Workshop on policies for distributed systems and networks, California, 2002, pp. 147-158.
[8] R. Ananthanarayanan, M. Mohania, and A. Gupta, "Management of Conflicting Obligations in Self-Protecting Policy-Based Systems", *proceedings of the 2nd International Conference on Autonomic Computing* (*ICAC*), IEEE, Seattle, 2005, pp. 274-285.
[9] J. Tan, and S. Poslad, "Dynamic security reconfiguration for the semantic web", *Engineering Applications of Artificial Intelligence*, 17(2004), Elsevier Ltd, 2004, pp. 783-797.
[10] R. Anthony, "Self-Configuration in Parallel Processing", *3rd International Workshop on Self-Adaptable and Autonomic Computing Systems - SAACS '05* (*DEXA 2005*), IEEE, Copenhagen, Denmark, August 2005, pp. 175-180.
[11] A. Herzberg, Y. Mass, J. Michaeli, and Y. Ravid, "Access control meets public key infrastructure, Or: assigning roles to strangers", *Symposium on Security and Privacy*, IEEE, California, USA, May 2000, pp. 2-14.
[12] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The Ponder policy specification language", In: M. Sloman, J. Lobo, E. Lupu (Eds), *Policies for Distributed Systems and Networks*, Springer, Berlin, 2001, pp. 18-38.
[13] XACML standard. Available at *http://www.oasis-open.org/comittees/xacml*.
[14] J. Chomicki, and J. Lobo, "Monitors for history-based policies", In: M. Sloman, J. Lobo, E. Lupu (Eds), *Policies for Distributed Systems and Networks*, Springer, Berlin, 2001, pp. 57-72.

IEEE COMPUTER SOCIETY