RICHARD ANTHONY
"Policy-Centric Integration and Dynamic Composition of Autonomic Computing Techniques"

Fourth International Conference on Autonomic Computing

Sponsored by

ICAC 2007

hp invent

IBM

FUJITSU
Intel
Microsoft Research
Raytheon
ACM
IEEE Computer Society

In cooperation with AAAI

11-15 June 2007
Jacksonville, Florida, USA

07/24

Session 1– Service-Level Agreements and Policies

Chair: Duncan Johnson Wolf, Enigmatec

Policy Verification and Validation Framework Based on Model Checking Approach
Shinji Kikuchi, Satoshi Tsuchiya, Motomitsu Adachi, and Tsuneo Katsuyama

Policy-Centric Integration and Dynamic Composition of Autonomic Computing Techniques
Richard John Anthony

SLA Decomposition: Translating Service Level Objectives to System Level Thresholds
Yuan Chen, Subu Iyer, Xue Liu, Dejan Milojicic, and Akhil Sahai

Policy-centric Integration and Dynamic Composition of Autonomic Computing Techniques

Richard John Anthony
Department of Computer Science, The University of Greenwich
Greenwich, London, SE10 9LS, United Kingdom
R.J.Anthony@gre.ac.uk

Abstract

This paper presents innovative work in the development of policy-based autonomic computing. The core of the work is a powerful and flexible policy-expression language AGILE, which facilitates run-time adaptable policy configuration of autonomic systems. AGILE also serves as an integrating platform for other self-management technologies including signal processing, automated trend analysis and utility functions. Each of these technologies has specific advantages and applicability to different types of dynamic adaptation.

The AGILE platform enables seamless interoperability of the different technologies to each perform various aspects of self-management within a single application. The various technologies are implemented as object components. Self-management behaviour is specified using the policy language semantics to bind the various components together as required. Since the policy semantics support run-time re-configuration, the self-management architecture is dynamically composable. Additional benefits include the standardisation of the application programmer interface, terminology and semantics, and only a single point of embedding is required.

1. Introduction

Autonomic computing has risen over recent years to become a key research area in computer science. This is largely because the functionality and scale of deployed software systems are following rapidly increasing trends. In many cases behavioural complexity now exceeds timely human comprehension and is thus a barrier to correct and optimised configuration of applications [1]. Impacts include longer development time, increased operational costs, reduced effectiveness, and higher occurrence of errors.

The field of autonomics is fundamentally concerned with the avoidance, removal or hiding of complexity. There are many different technologies being deployed to solve various problems in a variety of ways. Some of these solutions are very sophisticated, involving artificial neural networks, genetic algorithms etc, to achieve some degree of automatic learning. Other techniques can achieve context-aware behaviour despite their simplicity, for example utility functions. Some of the technologies offer greater potential 'behavioural intelligence' but are less generic, whilst others are more flexible and have wider applicability.

Each technology has different advantages and thus collectively they cover a very wide range of possibilities. This rich tool choice is driving the proliferation of autonomic solutions. However, current shortcomings in autonomics include the lack of a standardised development approach and a lack of interoperability across technologies. The various technologies are almost always used in isolation, the most appropriate single technology being chosen although many application domains have characteristics or requirements that straddle the provision of several technologies and are not ideally satisfied by a single technology.

On the other hand, developing systems with several different, incompatible technologies can actually represent an *increase* in complexity which is evidenced when it comes to deployment, versioning and upgrades etc. and thus represents risk. The author refers to this as 'complexity tail chasing', i.e. one dimension of the complexity problem is solved by adding complexity in a different dimension.

This paper presents AGILE, which serves as both a policy expression language *and* a framework that facilitates the integration and dynamic composition of several autonomic computing techniques and technologies. The combination of these technologies offers potentially far greater flexibility and powerful adaptation than when the technologies are used

separately. The framework represents a move towards a standard approach with the key benefit of interoperability across different autonomics techniques, which are combined together within a single deployment technology.

Policy-based computing has been chosen as the overarching technology for the framework because it has arguably the most flexibility and general applicability amongst the currently popular autonomics approaches; and thus offers the greatest potential to serve as a unification infrastructure for self-management. Specifically the following strengths are identified: 1. A policy can be used to express both structure and a sequence of logical steps to be followed, and is thus an ideal basis for composition. If the policy scheme supports dynamic self-adaptation, then the composition of behaviours can itself be dynamic. 2. In comparison with the other techniques, policies are relatively simple to develop and test, which translates into lower risk and cost. 3. The policy remains external to the embedded implementation mechanism, so even if a run-time static policy is used the system's behaviour can be subsequently modified without changing the code-base or its deployment. 4. The explicit decoupling between the business logic and the implementation mechanism makes policy schemes a feasible choice for deployment into legacy applications; with less effort than the other self-management techniques. 5. A policy can express high-level intentions, without concern for *how* they are achieved, and thus can be used to achieve dynamic adaptation by non-experts in self-management. 6. Policy schemes have low run-time resource requirements, which is an important consideration for mobile devices and embedded systems with limited resources. 7. In some applications policies can have re-usable aspects (for example a particular rule might be shared by several policies), reducing development time.

2. Supported self-management techniques

2.1 Templates and Policies

A comprehensive review of the various approaches to policy-based computing, with detailed comment on the current state of the art, has been provided in [2]. For completeness, a brief overview and update is provided here.

Templates: A template is the simplest type of 'policy', consisting only of configuration settings that are loaded at application initialisation. Internally there may be embedded policy logic comprising rules and actions etc, but the actual logic is fixed. Behaviour is thus fixed for any particular execution instance. Behaviour can be modified *between* executions but the extent of the modification is limited to changes to the parameters that constitute the template. [3] Embeds fixed rules into agents. Other examples are found in [4] and [5].

Statically configured policies: This category represents a more flexible approach than templates, because both the configuration parameters *and* the actual decision rules are held externally to the embedded mechanism. Although the behaviour can only be changed between executions, it is possible to update the actual logic of the policy as well as its initial parameterisation. A pilot study in [6], found that this style of policy-based configuration tended to have higher overall effectiveness in comparison to manual configuration. An example is provided by the Policy Description Language (PDL) [7], which is an event-driven programming language. Distributed Action Plans (DAP) are used to specify distributed network management tasks. DAPs are executed by policy agents which are specified in PDL [8].

The problem of identity delegation when policies are submitted by users is investigated in [9]. An intelligent delegation agent is used to determine the appropriate security authorisation for given policy.

Policy schemes supporting open-loop adaptation: The policy or its support mechanisms indirectly support adaptation of the actual policy. This support is in terms of *identifying*, or *facilitating* the identification of, inefficiencies in, or conflicts between, policies. The policy behaviour remains fixed during the current execution instance. The user is notified and manually updates the policy between executions. IBM Research's Policy Management for Autonomic Computing (PMAC) [10] provides an automated policy management and deployment application to assist with the manual policy updates. Policy Schedule Advisor [11] is a utility that assists in the refining of a policy schedule to ensure efficient execution on the PMAC middleware. The Unity policy environment 'Policyscape' [12] provides a set of 'templates' that can be used as building blocks to create more-complex policies and supports automated policy creation.

Policy schemes supporting closed-loop self-adaptation: The policy or its support mechanisms can automatically adapt the policy's own behaviour to suit contextual or environmental circumstances. The dynamic adaptation is achieved in a variety of ways and to a wide variety of extents. Ponder [13] is a sophisticated policy specification language with a feature-rich grammar that approaches the complexity of a conventional programming language [14]. Ponder is adaptive in the sense that meta-policies define semantic constraints on the regular policies. A security policy implementation described in [15] uses a meta-

policy to dynamically select between security policy versions, but policy updates are performed manually by administrators (although this can occur during run-time). Further examples of short-term adaptation are found in [16] in which event-trigger conditions are dynamic, and [17] in which conflicts between the obligations of security policies are automatically detected and resolved at run-time by dynamically removing conflicting obligations under certain circumstances. Similarly, the language described in [18] supports automatic detection and resolution of rule conflicts. The AGILE policy expression language has been purposefully designed to be highly flexible in terms of dynamic self-reconfiguration to facilitate context aware behaviour in a wide variety of application domains. It achieves dynamic self-adaptation in several ways which are discussed in section 3.

2.2 Data aggregation and signal processing

Typically in dynamic systems the current context cannot be completely determined solely from instantaneous values from environmental sources such as sensors. More likely, it is necessary to consider characteristics of the recent sequence of samples, such as mean values and counts of events such as the number of spikes or other anomalies detected over a certain time frame. Data aggregation and signal processing techniques, collectively referred to from this point on as Signal Processing (SP), provide a temporal association on top of discrete samples of environmental conditions.

2.3 Trend Analysis

Within the information supplied directly by an environmental input, or resulting from SP, there can be various trends which can be identified. Typical patterns that are of interest in autonomic systems include: rising / falling trends, stable values (showing no significant change over several samples), and step changes (abrupt and significant changes in value). By deploying an automatic Trend Analyzer (TA), a policy or other technology can base decisions on a more-complete view of the current system behaviour. Most significantly, the analysis of current or recent trends enables prediction of the short-term future. The importance of TA and prediction to self-management are discussed in depth in [19].

2.4. Utility functions

A Utility Function (UF) provides a means of choosing from several options, each expressed as a series of weighted terms. The term values are supplied by environment sensors and the weights are set (possibly dynamically) to reflect the application's interpretation of utility. The products of each term and its associated weight are combined to determine the *instantaneous* merit of each option. This technique allows context-sensitive decisions to be made dynamically with quite low implementation complexity. The versatility of UFs is illustrated by a brief review of several diverse examples: An emergent graph colouring algorithm [20] uses a UF to escape from illegal configurations which occur when all of a given node's neighbours are themselves legally coloured but there is no spare colour available for the node to colour itself without a clash. The UF is used to choose those neighbour(s) that must be forced to change colour such that the extent of the knock-on effect disturbance is minimised. In [21] a UF is used to dynamically select between context providers based on their various QoS parameters in an adaptive middleware that supports context-aware applications. UFs can be used as the basis for trading in simulated economic systems. The UF determines the price for which a particular agent is prepared to offer a service, or the price an agent will pay for a service, taking contextual factors into account. See for example [22]. Various different UFs can be used concurrently within the same system to solve quite challenging problems. For example [23] applies UFs to the optimisation of computing-resource allocation in data centres. UFs are used in two ways: individual autonomic elements apply service-level functions to optimise application resource usage; whilst at a higher level a system-wide arbiter uses UFs configured by application managers to allocate resources across the various application environments.

2.5 Integration of techniques

Each technique described above has some specific relative strengths over the others. Thus in combination the technologies have the potential to yield far greater benefit, in terms of flexibility, applicability and the extent of dynamic adaptation achievable, than when used in isolation. For example the SP and TA provide a temporal aspect that UFs and policies do not naturally have. Some of the benefits of combining specific techniques are discussed:

Policies and UFs. Policies can express logic, structure and sequence. Utility functions are a simple yet powerful way of making complex decisions based on the combination of several factors, each weighted to reflect current importance. Utility functions can choose between a large number of options whilst rules in policies tend to have a boolean outcome. So a policy is useful to express overall business logic, and a utility

function offers a means of making context-aware optimal choices at specific decision points. In combination a policy could dynamically configure a UF (by setting the various weight values), and then invoke it. Alternatively, a UF could be used within a meta-policy to determine the instantaneously most beneficial business policy to use, based on a number of environmental parameters. An example of embedding UFs into a policy-based system is found in [24].

Policies and SP. SP treats discrete input values as a related stream, generating useful attributes such as mean, mode, or median, and identifying features such as noise levels within the data. This can greatly simplify the amount of work to be done within the policy itself, and facilitates more 'intelligent' decisions.

Policies and TA. The predictive capability introduced by TA enables a policy to be proactive, rather than purely reactive if working with instantaneous values or the output of a SP.

SP and UFs. The ability of an UF to make context-aware decisions and select between a number of options is greatly enhanced if the attributes provided by a SP are available to it, rather than just the raw data inputs. For example, in a multi-sensor system it would be possible for a UF to select the most reliable sensor, identify anomalies in readings, and even to detect a faulty sensor by taking into account mean values, noise levels, and fluctuations in the values provided by each sensor.

TA and UFs. An UF's ability to select the best option from a choice of many is dependent on the amount and quality of information available to it. TA enables the UF to take into account short-term predictions. For example two resources might have the same *instantaneous* capacity, but choosing between them is simplified if it is known that one has a rising trend in load level whilst the other has a falling load trend.

3. The AGILE policy expression language

The advanced concepts for policy-based autonomics presented in [2] have been implemented in a sophisticated policy scheme named AGILE; comprising a feature-rich policy expression language, a powerful implementation library and a simple-to-use API. This policy scheme boasts a number of advanced and novel features, some key examples of which are described below using policy script excerpts for illustration. The discussion relates to the features of AGILE version 1.2.

1. Templates are used in three different ways. Firstly to provide initial policy configuration; an example is shown later in this section. Secondly to 'reset' a self-adapted policy back to a known state. Thirdly, templates can be dynamically created through a policy-state persist mechanism, such that the new template serves as a 'checkpoint' of current adapted state and can be used to 'warm start' new policy instances.

2. Dynamic self-adaptation of policies. The policy can change its own behaviour through the use of indirect addressing at the policy-script level, so that the actual variables that are compared in a rule, or the action to be followed on the outcome of a particular rule, can be changed dynamically. For example, variables of type *PolicyObject* can be declared:

```
<IVariable Name="ActionName" Type="PolicyObject"/>
<IVariable Name="FirstVAR" Type="PolicyObject"/>
<IVariable Name="SecondVAR" Type="PolicyObject"/>
```

Their values can point to any policy objects, using a template to provide initial configuration for example:

```
<Template Name="T1">
  <Assign Variable="ActionName" Value="RuleABC"/>
  <Assign Variable="FirstVAR" Value="Var1"/>
  <Assign Variable="SecondVAR" Value="Var2"/>
</Template>
```

With this particular configuration the following rule R1 compares Var1 and Var2, and if equal executes RuleABC (because 'ActionName' points to it):

```
<Rule  Name="R1"
       LHS="FirstVAR" Op="EQ" RHS="SecondVAR"
       ActionIfTrue="ActionName"
       ElseAction="null"/>
```

This logic can be re-configured dynamically to suit execution context. For example, a rule elsewhere in the policy might execute action block A1:

```
<Action Name="A1">
  <Assign LHS="ActionName" RHS="RuleXYZ"/>
  <Assign LHS="FirstVAR" RHS="Var3"/>
  <Assign LHS="SecondVAR" RHS="Var4"/>
</Action>
```

This changes the behaviour of rule R1, when next evaluated (as 'ActionName' now points to RuleXYZ).

Rules (and some other types of objects) each have a Boolean guard property whose value can be changed dynamically. If the guard is 'false' the rule is skipped. This provides a simple means of changing rule execution sequences dynamically, and allows rules to operate either in the *event+condition→action* mode (the guard provides the condition), or by leaving the guard at its default true state, rules operate on the simpler *event→action* basis. For example the following action A2 evaluates rule R1 and then R2 by default:

```
<Action Name="A2">
  <EvaluateRule Rule="R1"/>
  <EvaluateRule Rule="R2"/>
</Action>
```

The assign statement below causes R1 to be skipped, so that action A2 (in the future) will evaluate R2 only:

```
<Assign LHS="R1:Guard" RHS="false"/>
```

3. The policy language is object oriented, enabling re-use of variables, rules and other components. For example, several actions may reference the same rule.

4. Policies can 'yield', i.e. transfer control, to another policy. This facilitates dynamically switching the active business policy based on appropriateness to run-time context. Control can be handed to a specific policy, (such as Policy1 in the following example):

```
<Yield Policy="Policy1"/>
```

Policy selection can be dynamic, to support context-sensitive decisions. The target policy name is held in a *PolicyObject* type variable, which is declared as:

```
<IVariable Name="PolicyName" Type="PolicyObject"/>
```

The new policy is chosen by the current policy logic. Subsequently, the current policy yields to whatever policy is referenced by the *PolicyName* variable:

```
<Assign LHS="PolicyName" RHS="Policy2"/>
  ...
<Yield Policy="PolicyName"/>
```

Policies are switched 'hot'; i.e. all objects retain their state during the policy swap. In the typical case where policies share some objects such as rules and variables, the new policy picks up where the other left off and the switch is transparent to the controlled system.

5. The language directly supports self-stabilising behaviour. The tolerance-range-check object (TRC) encapsulates dead-zone logic and a three-way decision fork within a single policy component. A key use of the TRC is to reduce oscillation. Consider a system in which it is necessary for the system state to track a *dynamic* goal, whilst minimising oscillation due to lag and overshoot. A typical TRC configuration is shown:

```
<TRC Name="DZ"
  Check="CurrentValue" Compare="GoalState"
  Tolerance="AcceptableDeviation"
  ActionInZone="null"     ActionLower="ActIncrease"
  ActionHigher="ActDecrease"/>
```

The *Check*, *Compare* and *Tolerance* parameters represent respectively: current system state; goal state; and acceptable deviation from goal state (i.e. deadzone). Separate actions can be specified for each outcome with respect to the *Check* variable, i.e. *in*, *lower*, or *higher* than the dead-zone. Figure 1 depicts the use of a TRC to enhance stability.

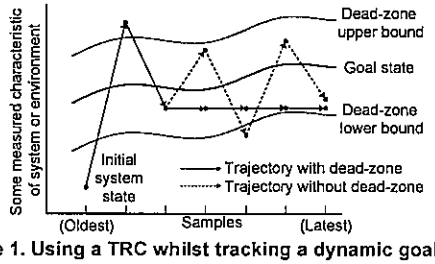

Figure 1. Using a TRC whilst tracking a dynamic goal state

As with rules, TRCs have a Boolean guard contributing to dynamic orchestration of evaluation sequences. Also like rules, TRCs can use indirect addressing for variable and action parameters. Thus the policy can dynamically reconfigure the TRC; for example sensitivity can be adjusted by changing the tolerance parameter to suit context.

6. Run-time diagnostics are provided; a text log is written after each policy invocation showing the sequence of logical steps taken (an example is provided in section 4.3).

7. Policies, once loaded via the API, retain state between invocations. This enables a temporal relationship to be established at the level of the policy, for example to allow reasoning about a sequence of events that span several invocations (see the examples presented in sections 4 and 5).

4. The integration framework

At the highest level, the framework is seen by systems developers as an extensive policy-based computing toolkit. The fundamental interaction with the framework occurs through the preparation of AGILE policy scripts, and through the integration of the implementation library with application code, by using the AGILE API.

However, the framework also integrates several autonomic computing technologies in a modular fashion within a single deployment architecture. The potential benefits of the combinatorial use of these different techniques have been discussed in section 2.5 above. Additional benefits of the framework include the standardisation of the application programmer interface, terminology and semantics, and only a single point of embedding is required.

Section 3 above has described the policy expression language aspect of AGILE. The remainder of this section focuses on the other technologies and their integration within the framework. Each technology is implemented as a modular component which can be connected to other components to form whatever logical structure is required.

4.1 Signal processing

The AGILE library automatically performs SP. This has been included to vastly increase the flexibility and expressive power of AGILE policies without any effort on the part of either the policy writer or the application developer. The SP logic is attached to each numeric variable and silently monitors successive sample values. This enables reasoning within policies to incorporate temporal characteristics of an input stream, such as mean value, largest value, etc. The various

temporal characteristics are represented as properties of their respective variable, and represented in policy scripts in the form object:property. Thee properties can be directly incorporated into policy logic. For example the following rule, R3, tests whether the exponentially smoothed mean of variable Var1 is greater than the maximum observed value of variable Var2.

```
<Rule Name="R3"
  LHS="Var1:ExpMean02" Op="GT"
  RHS="Var2:MaxValueSeen" ActionIfTrue="SomeAction"
  ElseAction="OtherAction"/>
```

AGILE supports self-adaptation at several levels. The various ways that dynamic configuration can be achieved at the level of policies has been discussed in section 3. However the SP component is also capable of automatic internal self-adaptation in two important ways:

Automatic determination of noise level. The 'noise level' implies the natural level of deviation between values in the input stream which does not indicate an interesting event. Noise level is measured as double the mean of absolute deviations between successive samples, omitting samples where a spike (or other characteristic recognisable by the TA – see later) begins or ends, which would significantly affect the accuracy of the approach. The mean is determined by exponential smoothing with a smoothing constant α = 0.1 (the weight associated with the latest deviation). The resulting value is assigned to the noise level property of the appropriate variable (i.e. for variable Var1 this is `Var1:NoiseLevel`). This property is used directly by the TA during the process of identifying the beginning of a spike or other feature within the input pattern. As with other properties, `NoiseLevel` can also be directly manipulated within a policy or can form part of the input to a utility function.

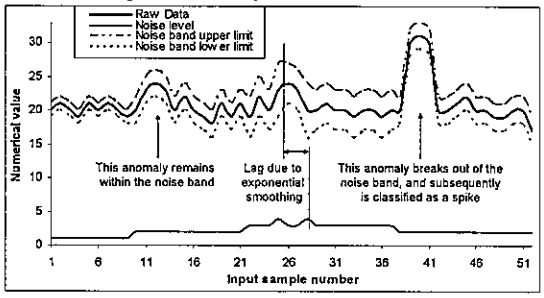

Figure 2. Automatic tracking of noise level within the SP

Figure 2 illustrates the operation of the automatic noise level determination facility. The upper and lower noise-band limits are shown to illustrate how the noise level property can be utilised by further stages, such as the TA. Noise level tracking enables the anomalies to the left and centre of the figure to be classified as uninteresting, whilst the anomaly to the right is identified as a spike.

Automatic determination of step characteristics. A signal that emerges from the noise band is initially identified as a spike. However, it is important to be able to determine at what point the spike becomes a step, i.e. the change in behaviour is considered permanent in the short time frame and the system should respond to it. A system needs to be able to perform this identification otherwise it must treat all spikes and steps alike, which can lead to inefficient and unstable oscillatory behaviour if spikes are frequent. For example in resource management, steps in an application's demand level require a reallocation of resources. However, if the pattern that is treated as a step is in fact a spike (i.e. a short-lived fluctuation) the work of reallocating resource is wasted. If such spikes occur frequently and are not detected as such the system is at best inefficient and possibly unstable. This problem is discussed in some detail in [25].

The SP component uses a per-variable configuration property `StepMinimumWidth` to set the lower threshold value for the number of samples that constitute a step. The property has a default value of 5 and is not changed automatically; i.e. the algorithm described below treats this value as an absolute lower limit. However it can be assigned a value from within the policy script, e.g. (for variable Var1):

```
<Assign Variable="Var1:StepMinimumWidth" Value="3"/>
```

An anomaly that ends before this count is reached is classified as a spike.

The automatic adjustment algorithm takes into account the step duty-cycle (the proportion of data samples that constitute steps). The duty-cycle is computed by exponentially smoothing observed step-widths (with smoothing constant α = 0.2). The same smoothing is applied to inter-step gaps.

If the mean duty cycle is greater than 33% i.e. one third of the time or more is spent in (non-continuous) steps, then the system is in danger of oscillating. The problem is more significant when the actual step-widths are short; yielding less benefit from each change of behaviour. Thus a second level of reasoning is applied, comparing the exponentially smoothed mean step-width value (which by its nature puts greater emphasis on recent values) against the minimum acceptable step-width value. When both problem indicators are present (the duty cycle is high and also the mean step-width is marginal, i.e. there are a high number of steps close to the minimum width), the SP component automatically increases the minimum step width (so that the classification is tightened, and marginal-width steps are now classified as spikes). This is achieved by incrementing an offset value which is added to the `StepMinimumWidth` property. This offset

is likewise decremented (but cannot drop below zero) when the system is more stable; i.e. a duty cycle of less than 17% (1 in 6). Further adjustments can be made until the system reaches a stable state. To ensure that these adjustments themselves do not cause instability, the system does not make adjustments more frequently than once every three steps it encounters, to ensure that the effects of previous changes have a chance to take effect.

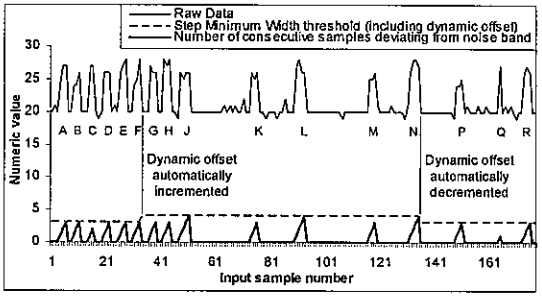

Figure 3. Automatic adaptation of step classification threshold

Figure 3 illustrates this dynamic adaptation behaviour. The example shown is taken directly from one of the implementation library's programatic correctness tests of adaptive behaviour. For the relevent variable, the `StepMinimumWidth` property was initialised to the value three in the test policy script, and the dynamic offset was internally set to zero in the library. In combination these values determine the subsequent classification of anomalies. A number of closely bunched anomalies (labelled A – J) threaten stability. The mechanism detects (after anomaly F) that a sequence of marginal-width steps is occuring in close succession, and responds by temporarilly incrementing the threshold number of consecutive samples within a spike needed for it to be re-classified as a step. The TA subsequently classifies anomalies of up to three samples wide as spikes (effectively downgrading the short steps (G, H, K and M) and re-gaining stability. When the duty-cycle between recognised steps has fallen back to a sustainable level the threshold is decremented. Anomalies C and Q are less than 3 samples wide so are always classified as spikes. Anomalies J, L and N have sufficient width to be still classed as steps after the threshold has been raised.

Together these two dynamic internal adaptations support self-stabilisation and optimization, and operate independently on each variable defined in a policy.

There will be situations where it remains desirable to manually set the noise level and/or spike/step crossover point, instead of automatically adjusting them. Therfore these features are disabled by default and must be enabled by assigning the value 'true' to the appropriate property on specific variables. This can be achieved in a template, as shown below, or dynamically from within an action block:

```
<Assign Variable="Var1:AutoNoiseLevel"
        Value="true"/>
<Assign Variable="Var1:AutoSpikeWidth"
        Value="true"/>
```

4.2 Trend analysis

This component provides more-complex temporal analysis than the SP. The TA automatically classifies the higher-level patterns within an input stream, as shown in figure 4.

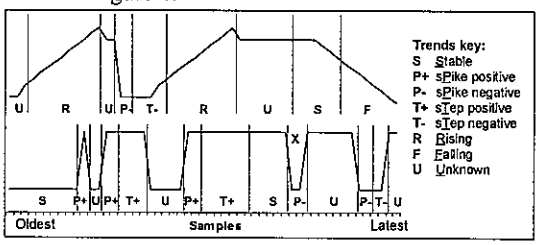

Figure 4. Trends identified by the trend analyzer

The TA differentiates between positive and negative spikes, and similarly between positive and negative steps. The recognition of 'stable', 'rising' or 'falling' trend requires a sequence of consecutive samples lying within the noise band. The three states are distinguished by examining the difference between values offset by 10 samples. Once the 'stable' trend has been detected, the mean signal value becomes the new norm from which deviation is measured. Hence the sudden drop (at point X) in the centre-right of the lower trace is identified as a negative spike.

The current trend for a specific variable is accessible within policies via the *Trend* property. For example the rule R4, "if variable X has a rising trend do Action_Y" is expressed as:

```
<Rule Name="R4" LHS="X:Trend" Op="EQ" RHS="Rising"
  ActionIfTrue="Action_Y" ElseAction="null"/>
```

By implementing TA in the library, as a policy-composable component, the analysis can be performed on all variables, including external variables that provide environment sensing information, and internal variables that are used as counts, thresholds etc. No implementation is required within the actual policy script or within the host application; all the work is done silently within the implementation library.

4.3 Utility functions

The utility function (UF) component implements summative UFs with a policy-specified number of input terms per option line. The level of importance attributed to each input term is represented by the

weight value associated with the term. The weights can be fixed or dynamic, in the latter case enabling an application to determine the relative importance of each input term based on current context. Term and weight values can be provided by any of: environmental variables passed into the policy; internal variables such as threshold values and counts of events; properties generated by a SP component; or numeric constants.

A single option i having N terms, is defined as:

$$U_i = (W_{i1} * T_{i1}) + (W_{i2} * T_{i2}) + \ldots + (W_{iN} * T_{iN})$$

where $W_{i1} \ldots W_{iN}$ are weights, $T_{i1} \ldots T_{iN}$ are input terms and U_i is the resultant 'utility' attributed to the option.

Consider an application that has access to several sensors that each monitor the same aspect of the environment, thus providing a degree of redundancy. The sensors might measure physical attributes of a system such as pressure, temperature etc, or could equally be information feeds coming concerning stock prices or other financial information. Selection amongst them should be made on a *qualitative* basis; i.e. the selection is not concerned with the discrete data values produced by the sensors, rather it is concerned with characteristics of their output signals that are indicative of stability, trustability and reliability. As there are multiple qualitative measures for each signal, the problem is ideally suited for solving with an UF.

To demonstrate the use of the policy framework in this scenario, an evaluator tool was devised to facilitate the insertion of controlled extents of noise and spikes onto signals; thus enabling simulation of a range of realistic environments.

Two signals are passed into the policy logic as variables. The SP attached to each variable generates two quality measures (`SpikeInterval` and `NoiseLevel`). These values are fed as inputs into an UF; i.e. the quality of signal i is represented by the tuple $\{T_{i1}, T_{i2}\}$ where T_{i1} = `SpikeInterval`, and T_{i2} = `NoiseLevel`.

The evaluator tool passes the weight values into the policy as variables W1 and W2. The results presented below were generated with the weights set as follows:

$W_{11} = W_{21} = 1$ (represented as W1 in the policy),

$W_{12} = W_{22} = -12$ (represented as W2 in the policy).

These weights were chosen to balance out the numerical bias in the actual properties generated by the SP; effectively weighting the two characteristics as being equally important. The weight value attributed to the `NoiseLevel` property is negative because *lower* values of noise equates to *higher* quality.

The UF determines which signal is 'best' by computing the utilities: $U_{SIGNAL1}$ and $U_{SIGNAL2}$. The policy is shown below:

```
<EnvironmentVariables>
  <EVariable Name="S1" Type="long"/>
  <EVariable Name="S2" Type="long"/>
  <EVariable Name="W1" Type="long"/>
  <EVariable Name="W2" Type="long"/>
</EnvironmentVariables>
<Templates>
  <Template Name="T1">
    <Assign Variable="S1:AutoNoiseLevel"
            Value="true"/>
    <Assign Variable="S2:AutoNoiseLevel"
            Value="true"/>
  </Template>
</Templates>
<ReturnValues>
  <ReturnValue Name="R_Signal1" Value="1"/>
  <ReturnValue Name="R_Signal2" Value="2"/>
</ReturnValues>
<UtilityFunctions>
  <UF Name="UF1" Terms="2">
    <Option Action="R_Signal1" T1="S1:SpikeInterval"
            W1="W1" T2="S1:NoiseLevel" W2="W2"/>
    <Option Action="R_Signal2" T1="S2:SpikeInterval"
            W1="W1" T2="S2:NoiseLevel" W2="W2"/>
  </UF>
</UtilityFunctions>
<Policies>
  <Policy Name="Policy1" PolicyType="NormalPolicy">
    <Load Template="T1"/>
    <Execute Action="UF1"/>
  </Policy>
</Policies>
```

The policy script combines three autonomics technologies (policy, SP and UF) yet is itself very short and simple. This is fundamentally because key features such as SP and extensive run-time validation are embedded in the library and operate automatically.

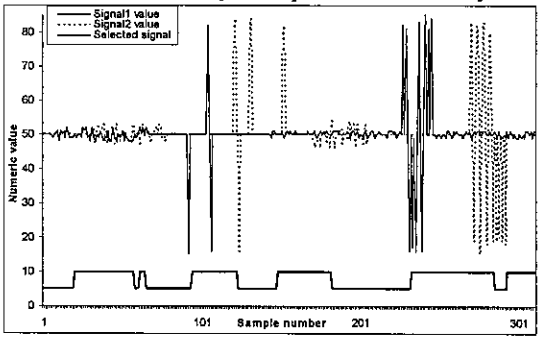

Figure 5. Dynamic, qualitative signal selection

Figure 5 shows a sample of the selection behaviour of the UF. The 'selected signal' trace indicates which signal is attributed the highest 'utility': low indicates signal1, high indicates signal2. The system parameter that the sensors were measuring was fixed at the value 50 throughout the experiment to aid clarity, as it is the quality of the signals that is important, not their specific instantaneous readings.

AGILE automatically generates a run-time trace each time a policy is evaluated. These traces are retrieved as a text string, via a method on the API. An abridged run-time trace for the signal selection policy is shown, corresponding to a single evaluation

instance. Only properties updated in current evaluation instance are listed in the trace.

```
Property: Obj=S1 Prop=MinValueSeen Val=15
Property: Obj=S1 Prop=ExpMean02 Val=43
Property: Obj=S1 Prop=NoiseLevel Val=1
Property: Obj=S1 Prop=Trend Val=SpikeNegative
Property: Obj=S2 Prop=ExpMean02 Val=49
Property: Obj=S2 Prop=SpikeInterval Val=8
Property: Obj=S2 Prop=NoiseLevel Val=2
Property: Obj=S2 Prop=Trend Val=Unknown
Policy: Policy1   Type: NormalPolicy
    Load Template - Skipped (already loaded)
Action: UF1
UtilityFunction: UF1  Guard was True: Evaluated
    Terms:2   Options:2
    Variable: W1    Value: 1
    Variable: W2    Value: -12
    Utility values {15:R_Signal1}{-16:R_Signal2}
    Selected Action:R_Signal1
Action: R_Signal1
ReturnValue: R_Signal1
Property: Obj=S1 Prop=PreviousValue Val=15
Property: Obj=S2 Prop=PreviousValue Val=50
```

The results (a sample are shown in figure 5) suggest that the properties generated by the SP are sufficiently reflective of the signal characteristics that the UF manages to choose the cleanest signal under a range of noise-level and spike occurrence conditions.

4.4 Integration and interoperability

The example application described in section 4.3 illustrates integration of three key components; policy, SP and UF. The integration is more-clearly shown in a composition diagram.

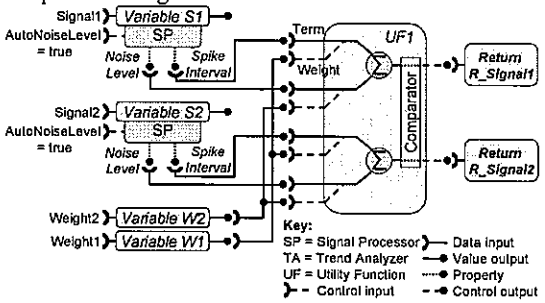

Figure 6. Composition of component behaviours

The representation of composition in figure 6 highlights the function of, and the interconnection between, components. The interfaces are standardized so that the various components can be connected together in a variety of ways to achieve a very flexible range of behaviours with a relatively small number of component types. The policy writer simply has to compose the desired behaviour using policy logic to bind the components as required.

In the application logic shown, the actual values of variables S1 and S2 are not examined; instead qualitative properties produced by the SP are fed directly into the UF. The SP and TA are actually wired to *each* numeric variable, but are only shown in cases where they are used, to aid clarity.

To illustrate the flexibility of AGILE's modular integration approach, consider a version of the adaptation example provided in section 4.3, but now with stricter requirements on the selection of signals, such that it is not sufficient to examine current properties of the signals themselves (`SpikeInterval` and `Noiselevel`), but that it is necessary to examine *trends* in these qualities. This more-sophisticated scenario is facilitated by cascading Variable components (and thus SP and TA), as shown in figure 7 (only the revised logic for one property is fully shown).

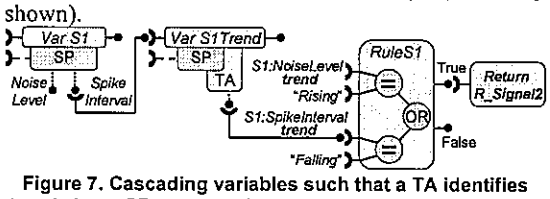

Figure 7. Cascading variables such that a TA identifies trends in an SP-generated property. A rule then evaluates the trend.

The TA output is non-numeric so a rule is used to check for adverse trends. Key additions to the policy script, extending the example from section 4.3, are shown below. Only the handling of *signal1* is shown due to limited space.

```
<InternalVariables>
  <IVariable Name="S1Spike" Type="long"/>
  <IVariable Name="S1Noise" Type="long"/>
  ...
<Action Name="A1">
  <Assign LHS="S1Spike" RHS="S1:SpikeInterval"/>
  <Assign LHS="S1Noise" RHS="S1:NoiseLevel"/>
  ...
  <EvaluateRule Rule="RuleS1"/>
  <EvaluateRule Rule="RuleS2"/>
  <EvaluateUF UF="UF1"/>
</Action>
<Rule Name="RuleS1"
      LHS="S1Spike:Trend" Op="EQ" RHS="Falling"
      LOp="OR" LH2="S1Noise:Trend" Op2="EQ"
      RH2="Rising" ActionIfTrue="R_Signal2"
      ElseAction="null"/>
  ...
<Policy Name="Policy1" PolicyType="NormalPolicy">
  <Load Template="T1"/>
  <Execute Action="A1"/>
</Policy>
```

The policy invokes Action-block A1 instead of directly invoking UF1. This enables a sequence of actions to be specified: assign the appropriate SP-generated properties to the new variables; invoke the new rules; and if neither rule fires 'true' (and thus returns a decision result), UF1 is invoked as before.

5. Dynamic composition

Sections 4.3 and 4.4 illustrate the ease with which the various components can be bound together; the composition being expressed in the policy script itself.

Some applications operate in a wide variety of

contexts and their environments can exhibit variable, and /or high volatility. Therefore there can be limits to the correctness and optimality of a static policy under certain circumstances.

However, due to the inherently dynamic reconfigurability of AGILE (explained in section 3) it is straightforward to write policies that are self-adaptive; *including the actual composition of behaviours*, i.e. the way in which the various components are connected together and interact.

A few of the very many scenarios in which this is useful include: Dynamically changing the relative priority of rules or other actions (a common requirement in security policies, but also important in optimization and healing scenarios); Automatically swapping between a conservative policy or rule (suitable for turbulent environmental conditions), and a more aggressive variant which can bring higher dividends in more-stable conditions; and Automatically detecting, and adjusting sensitivity to, differing levels of noise and other characteristics of the environment signals.

To demonstrate dynamic self-adaptation at the policy level, an extension to the application scenario presented in sections 4.3 and 4.4 is described. Consider a problem that can occur: if the selection logic flips between signals and back again in a short time frame the managing system is itself producing oscillatory behaviour which is highly undesirable from stability and efficiency viewpoints.

Self-stabilisation behaviour is now added to prevent switching between signals too frequently. There are two key requirements to make this work: firstly, a means by which the policy logic can detect that it is producing unstable behaviour; and secondly, a means by which the policy logic can re-configure itself to restore stability.

The first requirement is satisfied by using a property `DecisionChangeInterval` which is implemented on decision-making objects (Rule, TRC, UF, TA). This is a policy-accessible exponentially-smoothed measure of the number of similar decision outputs between instances where the decision is changed. Thus a policy can monitor its own performance and the overall stability of the controlled environment. A low value indicates that the policy is over-reactive and needs to de-tune itself, or in extreme cases that the policy configuration is not well-suited to the environmental conditions; this can be used to drive a self-adaptation change of the policy logic.

The second requirement can be satisfied in several ways. The example below demonstrates the use of the special '*PolicyObject*' type of variable, which provides indirect addressing. Indirect addressing is used to dynamically change the relationships between components; in this case to maintain stability when the policy detects that it itself is flipping between two decision states.

Taking the previous policy as a starting point, there are several implementation changes: 1. The outputs of the UF are indirectly mapped onto return values (which form the output of the policy itself) using variables of a special type. 2. An alternative decision maker is provided to operate in extreme conditions when the current UF is unstable (to keep the demonstration simple this 'alternative' is in fact an action-block that provides constant values, but could easily be another active component such as Rule, UF, TRC etc.). 3. A new stability-enforcing rule is added to detect unstable behaviour. The rule manipulates the indirect addressing to reconfigure the policy logic. This has the effect of automatically switching between the UF and the alternative decision maker, see figure 8. The rule continues to monitor the UF's decision behaviour. Once this has stabilised, the rule reverts the policy logic back to its original configuration.

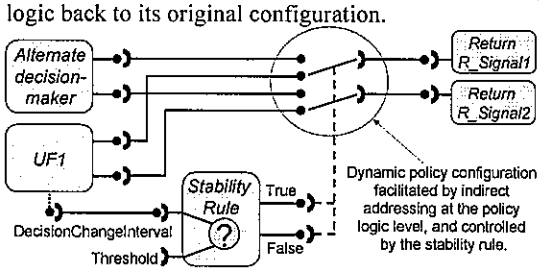

Figure 8. Self-adaptation of policy logic

The additions to policy script are shown below (key aspects are shown bold):

```
<InternalVariables>
  <IVariable Name="U1_ACT1" Type="PolicyObject"/>
  <IVariable Name="U1_ACT2" Type="PolicyObject"/>
</InternalVariables>
<Action Name="A1">
      ...
  <EvaluateRule Rule="RuleS1"/>
  <EvaluateRule Rule="RuleS2"/>
  <EvaluateRule Rule="StabilityRule"/>
  <EvaluateUF UF="UF1"/>
</Action>
<Action Name="A_Stable">
  <Assign LHS="U1_ACT1" RHS="R_Signal1"/>
  <Assign LHS="U1_ACT2" RHS="R_Signal2"/>
</Action>
<Action Name="A_UnStable">
  <Assign LHS="U1_ACT1" RHS="R_Signal1"/>
  <Assign LHS="U1_ACT2" RHS="R_Signal1"/>
</Action>
<Rule Name="StabilityRule"
  LHS="UF1:DecisionChangeInterval" Op="GE" RHS="5"
  ActionIfTrue="A_Stable" ElseAction="A_UnStable"/>
<UF Name="UF1" Terms="2">
  <Option Action="U1_ACT1" T1="S1:SpikeInterval"
          W1="W1" T2="S1:NoiseLevel" W2="W2"/>
  <Option Action="U1_ACT2" T1="S2:SpikeInterval"
          W1="W1" T2="S2:NoiseLevel" W2="W2"/>
</UF>
<Policy Name="Policy1" PolicyType="NormalPolicy">
  <Load Template="T1"/>
```

```
  <Execute Action="A1"/>
</Policy>
```

To evaluate the effectiveness of the self-adaptation, the original and dynamic policies were compared under extreme conditions. Very high noise levels were injected into each input signal, such that the original policy tended to flip erratically in its selection of the best signal. The results are shown in figures 10 and 11.

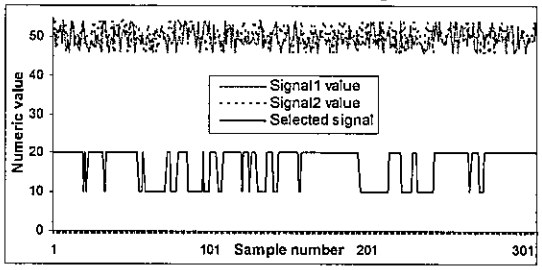

Figure 9. Instability when signals have high levels of noise

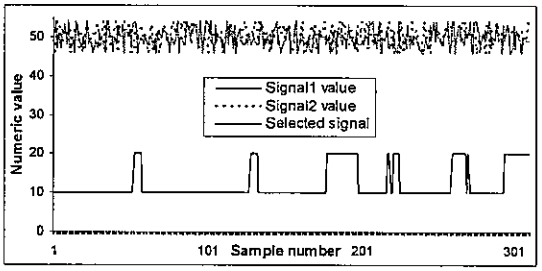

Figure 10. Stabilised behaviour through dynamic policy adaptation

The difference in the signals' quality is marginal at any moment and thus it is undesirable to flip between them. As can be seen from the 'selected signal' trace (low indicating signal1 selected, high indicating signal2), the dynamic policy exhibits significantly more-stable behaviour. The original policy flipped 38 times over a run of 310 samples (figure 9), whereas the dynamic policy achieved only 28 flips with a `DecisionChangeInterval` threshold value of 3, and only 15 times with a threshold of 5 (figure 10).

The adaptation in this example is based on dynamically swapping the UF decision maker for an alternative component, better suited to context. However, if the context was radically different it might be more appropriate to change the entire policy, and not just the component interaction within. This would be achieved by replacing the content of the A_UnStable Action block with a 'yield' action, such as:

```
<Yield Policy="AlternativePolicy"/>
```

This section has rounded off the discussion of the integration framework by presenting an example application in which a policy monitors and adjusts its *own* configuration dynamically to suit its execution context. This is achieved in addition to managing the controlled system. The input sensors were simulated but the real policy library and policy were used.

All of the policy resources discussed in this paper (including the AGILE library, policy expression language semantics documentation, API documentation, sample applications and policy scripts) are available for download at the AGILE support website [26].

6. Conclusion

The longer-term advancement of autonomic computing and its acceptance by the wider software development community depends on the provision of powerful and versatile tools that facilitate the creation of flexible management systems.

This paper has presented AGILE, which comprises a policy expression language and a framework to support the integration of several techniques popular in the development of autonomic systems.

The work goes some way towards providing a generic toolkit for autonomic computing; this is the overall goal.

The approach taken uses policies not only to express business logic, but also to provide composition structure so that several different autonomics techniques can interoperate. This represents a flexible way to implement a very wide variety of autonomic systems and AGILE has a powerful application programmer interface (API) to support and simplify implementation and deployment.

Mostly the various mechanisms embedded in the implementation library operate silently and automatically deal with run-time validation and certain internal self adaptations. This enables policies to provide powerful expression of behaviour whilst being quite short and simple to understand semantically.

The policy language has several novel features which contribute to pushing forward the state of the art in policy based computing; these were discussed in section 3.

The framework itself provides two further contributions: First, the novel use of the policy language as a means of integrating the various other technologies in a modular fashion. Second, the fact that the policy language supports dynamic reconfiguration of the policies themselves means that the actual composition of, and interaction between, the various components can be adapted at run-time to increase contextual scope.

Human involvement with autonomic systems will gradually and inevitably move to higher levels of

abstraction. Low-level interactions will be replaced by control systems that monitor their *own* effectiveness and adapt themselves, as well as the components they manage. AGILE aspires towards this, with its self-adaptive signal-processing logic and the various mechanisms it supports for dynamic adaptation of policies, and dynamic composition of the various components.

7. Further work

Policy visualization support is needed to lever AGILE into the mainstream practitioner domain. The intention is to develop tools that allow a policy to be developed through drag-and-drop of the various component-types and connectors.

8. References

[1] R. Barrett, P. Maglio, E. Kandogan and J. Bailey, "Usable Autonomic Computing Systems: the Administrator's Perspective", *First Intl. Conf. Autonomic Computing (ICAC)*, New York, USA, May 2004, pp. 18-25, IEEE.

[2] R. Anthony, A Policy-Definition Language and Prototype Implementation Library for Policy-based Autonomic Systems, *3rd International Conference on Autonomic Computing (ICAC)*, IEEE, Dublin, June 2006, pp. 265-276.

[3] R. Basra, K. Lu, G. Rzevski and P. Skobelev, Resolving Scheduling Issues of the London Underground Using a Multi-agent System, *2nd Intl. Conf. on Industrial Applications of Holonic and Multi-Agent Systems (HoloMAS)*, Copenhagen, Denmark, LNAI 3593, Springer-Verlag, 2005, pp. 188-196.

[4] 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.

[5] F. Cuomo and C. Martello, "Ultra Wide Band WLANs: A Self-Configuring Resource Control Scheme for Accessing UWB Hot-Spots with QoS Guarantees, *Mobile Networks and Applications*, 10, Springer Science and Business Media Inc, 2005, pp. 727-739.

[6] E. Kandogan, C. Campbell, P. Khooshabeh, J. Bailey and P. Maglio, Policy-based Management of an E-commerce Business Simulation: An Experimental Study, *3rd International Conference on Autonomic Computing (ICAC)*, IEEE, Dublin, June 2006, pp. 33-42.

[7] J. Lobo, R. Bhatia and S. Naqvi, A policy description language, *Proc. AAAI*, Orlando, USA, 1999, pp. 291-298.

[8] M. Kohli and J. Lobo, Realizing Network Control Policies Using Distributed Action Plans, *Journal of Network and Systems Management*, 11 (3), Plenum Publishing Corporation, September 2003, pp. 305-327.

[9] R. Gupta, S. Roy and M. Bhide, Identity Delegation in Policy Based Systems, *3rd International Conference on Autonomic Computing (ICAC)*, IEEE, Dublin, June 2006, pp. 283-284.

[10] IBM Research, Policy technologies. *http://www.research.ibm.com/policytechnologies/.*

[11] R. Lotlikar, R. Vatsavai, M. Mohania and S. Chakravarthy, Policy Schedule Advisor for Performance Management, *proc. of the 2nd Intl. Conf. on Autonomic Computing (ICAC)*, IEEE, Seattle, 2005, pp. 183-192.

[12] O. Ronen and R. Allen, Autonomic Policy Creation with Singlestep Unity, *proc. of the 2nd Intl. Conf. on Autonomic Computing (ICAC)*, IEEE, Seattle, 2005, pp. 353-355.

[13] 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.

[14] Ponder Specification, available at: *http://www-dse.doc.ic.ac.uk/Research/policies/ponder/PonderSpec.pdf*

[15] 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.

[16] L. Lymberopoulos, E. Lupu, 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.

[17] R. Ananthanarayanan, M. Mohania and A. Gupta, Management of Conflicting Obligations in Self-Protecting Policy-Based Systems, *proc. 2nd Intl. Conf. on Autonomic Computing (ICAC)*, IEEE, Seattle, 2005, pp. 274-285.

[18] 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.

[19] E. Thereska, M. Abd-El-Malek, J. Wylie, D. Narayanan and G. Granger, Informed data distribution selection in a self-predicting storage system, *3rd Intl. Conf. on Autonomic Computing (ICAC)*, IEEE, Dublin, June 2006, pp. 283-284.

[20] R. Anthony, Emergent Graph Colouring, Engineering Emergence for Autonomic Systems (EEAS), First Annual International Workshop, at the third International Conference on Autonomic Computing (ICAC), Dublin, Ireland, 12-16 June, 2006, pp. 4-13.

[21] M. Huebscher, J. McCann, An adaptive middleware framework for context-aware applications, Personal Ubiquitous Computing (2006) 10, Springer-Verlag, pp.10-20.

[22] A. Wilhite, Self-Organizing Production and Exchange, Computational Economics, 21, 2003, Kluwer Academic Publishers, pp. 107-123.

[23] W. Walsh, G. Tesauro, J. Kephart, and R. Das, Utility Functions in Autonomic Systems, Proc. 1st Intl. Conf. Autonomic Computing (ICAC), IEEE, New York, May 2004, pp. 70-77.

[24] V. Kumar, B. Cooper and K. Schwan, Distributed Stream Management using Utility-Driven Self-Adaptive Middleware, *proc. of the 2nd Intl. Conf. on Autonomic Computing (ICAC)*, IEEE, Seattle, 2005, pp. 3-14.

[25] J. Chen, G. Soundararajan and C. Amza, Autonomic Provisioning of Backend Databases in Dynamic Content Web Servers, *3rd International Conference on Autonomic Computing (ICAC)*, IEEE, Dublin, June 2006, pp. 231-242.

[26] AGILE support website: *http://www.PolicyAutonomics.Net*