An Interoperability Service for Autonomic Systems

Richard John Anthony
The University of Greenwich
Park Row, Greenwich
London SE10 9LS, UK
+44 (0) 208 331 8482
R.J.Anthony@gre.ac.uk

Mariusz Pelc
The University of Greenwich
Park Row, Greenwich
London SE10 9LS, UK
+44 (0) 208 331 8588
M.Pelc@gre.ac.uk

Haffiz Shuaib
The University of Greenwich
Park Row, Greenwich
London SE10 9LS, UK
+44 (0) 208 331 8588
Haffiz.Shuaib@yahoo.com

***Abstract* - Interoperability support is a key outstanding requirement for autonomic computing systems, and this need stems from the very success of these systems. Autonomic computing is increasingly popular; soon autonomic control components will be commonplace and present in almost every large or complex application. Interoperability between autonomic managers is an increasingly urgent concern, as the proliferation of autonomic systems inevitably leads to situations where multiple autonomic components coexist and interact either directly or indirectly within the same application or system. Problems can arise when numerous *independently* designed autonomic components interact, potentially destabilising systems. We advocate a service-based approach to interoperability and present a set of requirements for such an approach as well as a suitable architecture. A key component of this architecture is the Interoperability Service with which Autonomic Managers register their management interests and capabilities, using a management description language. The Interoperability Service automatically discovers and manages potential conflicts between manager components. Developers integrate Autonomic Managers with the Interoperability Service by importing its interfaces. This allows the Interoperability Service to automatically suspend and resume managers, or specific management functions as necessary, driven by the automated conflict detection. We illustrate the use of the Interoperability Service in a data-centre scenario in which independently developed power management and performance management autonomic components operate.**

Keywords - Autonomic systems; Interoperability; Services.

I. INTRODUCTION

Autonomic Computing (AC) has matured rapidly from a hot research topic to an accepted and valued technique for automating system management, in less than a decade. The main reason that the popularity of AC has grown so strongly in such a short timeframe is because it offers solutions to the problems caused by high complexity in systems. This complexity arises from large numbers of interacting components, typically with high functionality and with high operational speeds working in high throughput applications. The number of possible configurations and the different interactions and sequences of interactions, increases at an exponential combinatorial rate as the underlying behavioural richness of the systems and sub-components increases. This rapidly leads to systems whose behaviour is beyond a human manager's comprehension, certainly in terms of making real-time configuration decisions. Autonomic computing automates the management of one or more sub-components or resources, thus controlling certain elected characteristics of a system in a timely manner; increasing optimality and robustness and reducing errors. The sophistication of AC has also advanced at a spectacular rate. This is largely due to the reuse and extension of a wide range of reasoning and control concepts and techniques taken from established fields such as control theory and artificial intelligence.

The rapid evolution of AC has been driven by a main focus on the internal reasoning techniques, and a bias towards isolated development and deployment of Autonomic Managers (AM) which tend to have a very specific operational envelope; in order to demonstrate the robustness of the core techniques and thus to gain acceptance for the overall concept of AC.

However, the popularity of AC is driving expansion into ever more diverse application domains and increasing the variety of aspects of systems that can be automatically managed. This means that for future AMs, it is not safe to assume isolated management operation. In fact, it will be increasingly common for multiple AMs to coexist in any moderately sized computer system.

Almost all systems use multi-vendor software solutions and this implies that there will be potentially a variety of manager components existing, even for any one specific function of a system. For many systems, autonomic management will arrive incrementally; as new functionality is introduced, and through upgrades of non-managed components to new managed versions. In some cases the introduction of management capabilities will not be obvious – third party developers may deliver components with internal management that is not exposed at interfaces to other components.

Unplanned coexistence, or unexpected interactions could arise due to the highly dynamic nature of some systems in which configurations, and composition of components changes quickly. Automatic upgrades of individual components are another increasingly popular way by which systems behaviour changes over time, and not necessarily with the designer of a specific component having full

visibility of the whole system behaviour. Thus even a 'known' manager component could suddenly introduce new behaviour or potential conflict.

The possibility of coexistence and thus unplanned interactions or resource conflicts means that AMs will operate in environmental conditions not foreseeable by their designers. This means that an AM may pass behaviour tests 'in the lab' but still exhibit undesired behaviour when deployed.

This work extends our earlier work in [1]. We are interested in the challenge of interoperability for AMs, especially in the context of unplanned interactions, which can take many forms, but fall into two classes. *Direct* conflicts occur where two AMs attempt to manage the same explicit resource. *Indirect* conflicts arise when AMs control different resources, but the management effects of one have an impact on the management function of the other, or the combined effect of the two managers has an undesirable impact at system level.

The indirect conflicts are expected to be the most frequent and problematic, as there are such a wide variety of unpredictable ways in which such conflicts can occur. In addition, the effects of indirect conflict will be less obvious to detect and harder to diagnose than the direct conflicts. There will also be a range of severity of the effects of conflicts, from little consequence (such as a cancellation effect of opposing managers) whilst others could lead to serious performance or stability problems or even failure. The problem is illustrated with an example: Consider a system with two AMs: a Power Manager (PM1) shuts down servers that have been idle for a short time; and a Performance Manager (PM2) attempts to maintain a pool of idle servers to ensure high responsiveness to high priority applications. The two services were developed and evaluated in isolation and both performed perfectly; however the respective vendors did not envisage that they would co-exist. In current state of practice for AM development, interoperability is not a first-class concern, so each manager will be unaware of the other, i.e., it has no mechanism to detect and adapt to the presence and behaviour of the other. Bringing a shutdown server back on line has a latency of several seconds, thus when both AMs are co-resident PM1's 'locally correct' behaviour defeats PM2's contribution.

This problem can only be resolved if an external agent (such as a human system manager) can detect, diagnose, and identify a solution to the problem. This illustration is quite similar to the situation described in [2], see section II.

The general lack of interoperability support for AC is an urgent problem that could threaten its long-term success if not addressed in the near future. Custom solutions for interoperability may be necessary in some specific applications but in general this is a very expensive approach. In addition to the application-technical challenge, the interoperability solution itself becomes an additional component to keep up to date, as the AMs themselves, and the operating environment change over time. Some important issues arising from custom interoperability attempts are discussed in section II.

We advocate a universal solution for AM interoperability that is integrated into AMs at design time but which does not impose any limitations on the technology used to implement the management control functions and does not restrict or interfere with the way in which the autonomic management logic operates. We propose an Interoperability Service (IS) that monitors the various autonomic components present in a system. When a conflict of interest is detected the IS selectively suspends or shuts down the management function of autonomic components, based on a service description exchanged during the AM registration process (i.e., at run time). The IS has a hierarchical structure to ensure scalability and operates with a primarily local focus but also handles conflicts between non-local components where relevant. The proposed approach requires that at design time the developer identifies the resources that the manager will directly control, as well as those that could be indirectly affected. The approach has the main benefit of not requiring the developer to have any knowledge of other managers that may be present at run time. Compliance with such a scheme will be a step towards eventual 'certification' of AMs, which is important for long-term acceptance and growth of AC.

The contributions of this paper include: firstly we evaluate the nature and scope of the interoperability challenge for autonomic systems and identify a set of requirements for a universal solution (section III). We present the architecture of a service-based interoperability solution in section IV. Section IV, part C outlines a management description language which is intended for use by developers to ensure consistent description of AMs' management capabilities. Automatic detection of management conflicts is discussed in section IV, part D. Section V presents a work-in-progress implementation of the IS, and this is evaluated in section VI.

II. BACKGROUND

This section discusses the state-of-the-art in autonomic component interoperability. We also discuss some scenarios reported in the autonomic computing literature where either: purposeful interaction between several autonomic elements has been attempted to achieve a common goal; or where unexpected interactions or conflicts occurred between independent autonomic elements.

The potential significance of unwanted interaction between multiple autonomic elements was demonstrated in [2]. In this work, two autonomic managers were implemented. The first of these managers, the WebSphere Extended Deployment (WXD) dealt with application resource management, specifically in the area of CPU usage optimization. The second manager referred to as the Power

manager was responsible for modulating the operating frequency of the CPU to ensure that the power cap was not exceeded. It was shown that without a means to interact, both managers throttled and sped up the CPU without recourse to one another, thereby failing to achieve the said optimization the managers were expected to achieve, in terms of resource allocation and power utilization optimization, and potentially destabilising the system. We envisage widespread repetition of this problem until a universal approach to interoperability is implemented.

There are several examples of bespoke interoperability solutions for specific systems. A distributed management framework that seeks to achieve system-wide Quality of Service (QoS) goals for autonomic/self-managing systems was proposed in [3]. In this work, autonomic controllers were added and removed from the system based on the demands of the application QoS requirements. Here, the controllers communicate indirectly with one another using the system variables repository. If a controller were to fail, other controllers reading this repository take over the responsibilities of the failed controller, to ensure that QoS objectives are met. Other research works take a more direct approach to autonomic element interaction. For instance, in [4] the autonomic elements that enable the proposed data grid management system communicate directly with one another to ensure that management obligations are met. This paper defines four types of autonomic element including a data scheduler, data replication service provider, client and server file system providers. The relationship between each type of autonomic element is peer-to-peer. In contrast, [5] adopts a three-level hierarchical relationship to autonomic element interactions. The hierarchy is such that it is made up of a single device at its lowest level. Multiple devices are grouped into servers and servers are further grouped into clusters. The autonomic element at each level interacts with the autonomic elements above and below it to achieve autonomic power and performance management. [6] proposes a two-level autonomic data management system that optimizes the managed system so that jobs are not starved of resources. Physical servers each support multiple virtual servers. Local autonomic controllers manage each virtual server. These controllers use fuzzy logic rules to determine the expected amount of resources needed by the applications that run on the virtual servers. A global manager is tasked with allocation of physical resources to the virtual servers in an optimal and equitable manner. [7] implements a mechanism similar to that proposed in [6], in that virtualization on each physical server is used to optimize system usage and power consumption. The difference is that in [7] the local controllers manage each physical server as opposed to the virtual machine (VM) in [6]. A higher-level autonomic manager interacts with the local controllers to switch on or off the physical servers to ensure that Service Level Agreements (SLAs) are met, while also lowering power consumption. In [8] a combination of database replication and the avoidance of 'hot-spots' (devices with above-average operating temperature) is used to improve the performance of the managed system. Here, the autonomic system consists of two types of element. The responsibility of the first autonomic element i.e., the application scheduler is the creation and destruction of replicas of a database to assure high-availability. The other autonomic element, the resource manager, interacts with the scheduler to provide physical computational resources to the applications based on the SLAs. In addition to other responsibilities, the resource manager uses a model of past operations to move jobs from equipment operating at a higher temperature onto equipment with lower operating temperature. [9] describes an experiment to separate out the Monitoring and Analysis stages of the MAPE loop into distinct autonomic elements, with designed-in interactions between them. Monitoring capabilities are implemented in a node called an agent, with the analysis aspect implemented in a node called a broker. Information received from the environment are processed by the agents and forwarded to the broker where it is further analyzed. One or more agents feed information to a specific broker. An example of bespoke designed-in interaction between autonomic elements is provided in [10]. Three types of autonomic elements work hierarchically to provide scalable management, differentiated in terms of their operating timescale and scope of responsibility. This example serves to differentiate interaction between components which is achieved here, from the concept of interoperability which has stricter requirements. The fact that the various elements are part of a single coherent service with designed-in support for interaction means that the full challenge of interoperability is not encountered in this situation.

[11] illustrates the complexity of combining multiple management domains into a single controller. In this work a joint QoS and Energy manager is developed using a design-time oriented approach tuned for a specific environment and is thus highly sensitive to its operating conditions. This tight integration approach is not generalisable and the resulting combined manager would appear to be much more costly to develop and test than two independent managers.

The majority of the work to date has targeted planned interoperability between designed-for-collaboration AMs working towards a common goal. This is a valuable step towards AM interoperability, although these solutions generally lack a formal definition of the interfaces or where defined, these interfaces are highly specific to the system in question, thus preventing wide applicability and reusability.

Custom solutions are expensive to develop and are sensitive to changes in the target systems, thus they are generally restrictive and not future proof. A significant issue is that they do not tackle the specific problem of unintended or unexpected interactions that can occur when independently developed AMs co-exist in a system. However, the wider problem of standardised and system independent interoperability in autonomic systems has been

considered in several works. For instance, [12] defines a number of interfaces {Monitoring and test, Lifecycle, Policy, Negotiation and binding} to aid autonomic element interactions. Together these interface definitions enable the following properties:

- A means to establish appropriate administrative relationships.
- A means to monitor an autonomic element.
- A means to instruct these elements from an external source.
- A means to determine the current state of an autonomic element e.g., start, stop etc.
- A means to export and import policies to and from an autonomic element.
- A means to grant and request service to and from another autonomic element.
- A means to provide interaction integrity.

Multi-agent systems have some similarities to multiple independent-AM systems. However the interoperability problem is different because a multi-agent system is usually a coherent application and thus designed and tested specifically with the intention of multiple, similar, known-at-design-time agents; whereas in the independent-AM case incremental addition of new or upgraded AMs introduces unplanned interactions (i.e., unplanned at the time the various AMs were designed and tested).

Several 'vision' papers [13], [14], [15] identify interoperability as a key challenge for future autonomic systems. [13] argues that the mechanisms that define interoperability between autonomic elements must be reusable to limit complexities i.e., it must be generic enough to capture all communications across the board but also prevent bloatedeness. A standard means must exist for exchanging contexts between communicating elements to allow one autonomic element to understand the basis for the action of another autonomic element. [13] also identifies the need for a function to translate the output of one element to the format understood by another. [14] identifies some necessary components for autonomic element interaction, including: a name service registry for autonomic elements; a system interaction broker and a negotiator. An interface specification must also take cognizance of hierarchy amongst autonomic elements. [15] observes that a strict and specified communication behaviour should be enforced, to prevent interoperating autonomic elements from communicating through undocumented or backdoor interfaces.

III. INTEROPERABILITY ISSUES AND REQUIREMENTS

This section highlights the technical challenges of providing interoperability between AMs, and analyses the requirements for a universal solution. The state-of-the-art in achieving interoperability in autonomic systems has been discussed in section II and is predominantly focussed on custom and system-specific (or application-specific) solutions. This demonstrates the plausibility of AM interoperability and provides important starting points towards our goal of universal interoperability.

We posit that interoperability support (or lack of it) will become a make-or-break issue for future autonomic systems which inevitably contain multiple AM components. Bespoke or application-specific approaches to interoperability only offer a temporary respite at best, as they suffer a number of significant limitations which include:

1. Lack of flexibility and ability to scale - it is unrealistic to keep adding signals and functionality to deal with each possible interaction between any combination of AM's.

2. Having many isolated pools of interoperability is too complex. AC became popular fundamentally as a means of controlling, or hiding, complexity. It is undesirable from maintainability and stability perspectives to actually add excessive complexity in the process of solving the complexity problem.

3. It is not technically feasible to achieve close-coupled interoperability (i.e., where specific actions in one AM react to, or complement those of another) unless the source code and detailed functional specification is available for each AM involved. Without standardised interfaces this will always be a major challenge.

4. It will not be cost effective or timely. The cost and complexity of a bespoke solution spirals exponentially as the number of interacting AM's increase (consider a cloud computing facility or data centre with multi-vendor management software systems and with autonomic management embedded into platforms, operating software, application software and also infrastructure such as power management and cooling systems – this is a complexity and stability storm just waiting to happen).

5. Re-development of managers to facilitate specific interoperability, and especially to deal with conflicts that arise unexpectedly, is reactive and incremental (thus always ongoing).

6. It is not possible to know the nature of AMs not yet built, or to predict exactly if/where/when conflict will materialise in advance of adding a particular AM into a running system.

7. The incremental re-development approach cannot be applied on-line (in the medium term) as current technology is not sufficiently sophisticated, although for the longer term it may be possible since work is underway in several projects to develop self-evolvable systems.

In summary, the biggest single challenge to universal interoperability of autonomic systems is that it is not possible (at time of design, development or deployment of a particular AM) to predict all future autonomic services that could be added to a particular system, or even to predict upgrades that could be made to known services.

A. Requirements of a Universal IS

The issues highlighted above strongly suggest that it is necessary to deal with interoperability proactively by developing managers that are interoperability-enabled from the outset. We propose a service-based approach to interoperability, in which an Interoperability Service (IS) is responsible for detecting possible conflicts of management interest, and granting or withholding management rights to specific AMs as appropriate. In this way the IS performs all of the active interoperability management, and AMs only participate passively by providing information and following control commands from the IS. The IS interacts with AMs via a special interface which they must support.

We identify a number of requirements for a universal IS solution:

- Be application-domain independent and system independent.
- Able to represent AMs' management interests in a standard way that facilitates accurate conflict detection. This includes recognising resources which are not directly managed, but are nevertheless impacted by the behaviour of the manager.
- Have variable conflict-detection sensitivity which is run-time configurable to suit specific system requirements.
- Have a hierarchical architecture so as to deal with both local and global conflicts, and conflicts that occur across different levels in a complex system.
- Be proactive and automated; these are mandatory qualities for sustainable systems containing dynamic combinations of AM's with potentially complex interaction patterns.
- Able to automatically suspend and resume AM management activity on the basis of conflict detection and resolution.
- Support independently developed and tested AMs which in the presence of other AMs are susceptible to conflicts that they cannot locally detect or handle.
- Be sufficiently trustworthy that compliant AM's are *certifiable* for safe co-existence – regardless of platform, vendor etc.

Two diverse candidate architectural approaches were considered: The first is fully distributed, with localised conflict detection logic embedded in each autonomic manager. This approach requires that each manager exchanges standardised management description information with other managers on a peer-peer basis. Each participant would compare their own management interests with those of its discovered peers. On discovery of a conflict, a negotiation phase would determine which manager has the authority to manage the contested resource. This approach has the benefit of a standardised conflict detection mechanism, embedded in the form of a library, but has the disadvantages of extensive replication of functionality, the need for the negotiation phase, and potential scalability limitations.

The second approach is central service based. This approach is based around an interoperability service which keeps details of all autonomic managers present and maintains a mapping of the resources they manage and their scope of operation and management. Autonomic managers register with the service via a standard interface (much like a name service) and provide details of their management capabilities using a standardised description language. The interoperability service contains the logic to detect conflicts and when necessary send a signal to one of the involved managers to stop its management activity. This approach can be highly scalable and robust if the service is itself distributed and operates hierarchically with a dynamically elected global instance.

We have adopted the second approach because it is scalable, generalisable, has low component-interaction complexity and has the advantage of not requiring further negotiation once a conflict has been detected.

IV. INTEROPERABILITY SERVICE

This section presents the architecture of an IS to facilitate exploration of the requirements identified above, and thus investigate the feasibility of a universal IS. By 'universal' it is meant that the architecture promotes a CORBA-like view of autonomic systems development, in which it is intended that any two autonomic managers that comply with the architecture specification will be guaranteed to co-exist in a system, without undesirable interactions leading to instability.

The IS maintains a database of all registered AMs along with a mapping of the resources they manage and their scope of operation and management. AMs register with the service via a standard interface and provide details of their management capabilities using a standardised description language. The IS detects potential conflicts and sends appropriate signals to one or more AMs to e.g., stop, suspend or restrict their management activity. The strengths of this approach are that it is scalable, generalisable, has low component-interaction complexity and because conflict management is handled within the IS, the AMs are not involved in negotiation with peers.

The service has a hierarchical structure for scalability, enabling conflict detection at both global level (such as system-wide security management) and local level (such as platform-wide, or VM-wide, resource management) with respect to a particular AM. Additional levels can be added, with a communication infrastructure resembling that of a typical hierarchical service such as DNS.

It is important that conflict-detection is performed at the correct level. For example, an autonomic VM scheduler only has a potential conflict with an autonomic memory manager, if they are both operating on the same processor unit.

Figure 1 shows the system-level view. The IS comprises a number of service instances distributed throughout a

system. Each instance of the IS provides service to a local group of AMs, resolving conflicts that occur at the local level. One of these instances is dynamically elected to serve as the global instance, and deals with resource conflicts at system level.

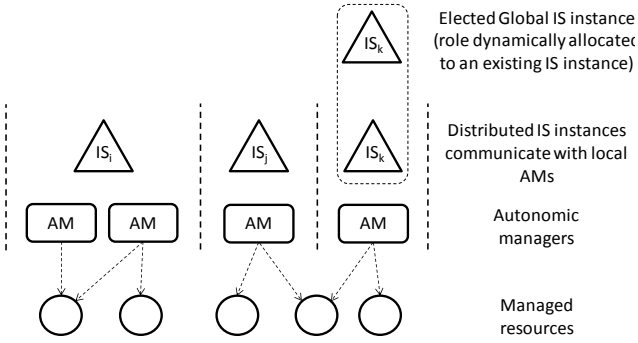


Figure 1. System-level view

The architecture is formed around a number of regular interfaces and a communication protocol which define the interaction between the components of the system, as shown in Figure 2.

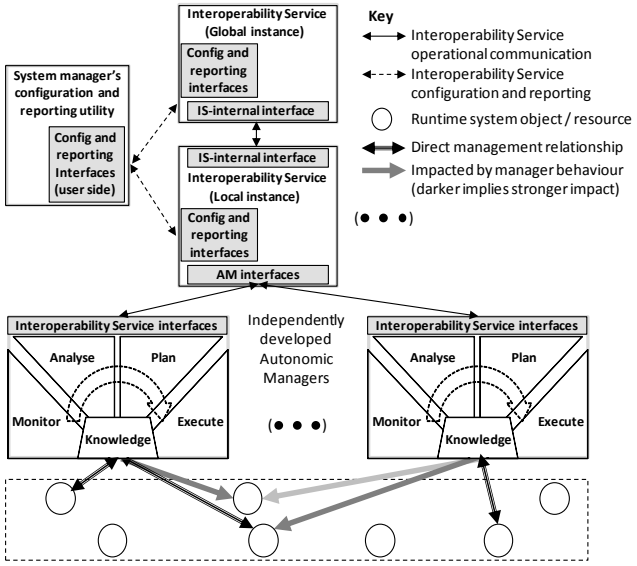


Figure 2. The Interoperability Service (IS) architecture, showing interface details.

A. Interoperability Service Interfaces

A number of interfaces are specified, and form three groups:

1. IS-AM interaction is supported by two interfaces.
   IAdvertise {*Advertise*, *Unregister*, *Heartbeat*} is used by AMs to signal joining (registering), leaving and heartbeat messages to the IS. *Advertise* is accompanied by a list of resources that the AM either wishes to manage directly, or that the developer has identified might be impacted by the manager's behaviour. This has the effect of registering the management interests of the AM with the IS. *Unregister* is used by an AM to signal an orderly shutdown, and *Heartbeat* (invoked periodically under normal conditions) enables (when absent) the IS to detect when a manager crashes or leaves abruptly. In either case, the AM's management interests are unregistered from the IS and the conflict detection analysis is triggered, so that any AMs which were suspended but are no longer in conflict with the system can be resumed.
   IInteroperate {*Run*, *Stop*, *Suspend*, *Restrict*, *Resume*, *Throttle*} is used to receive directives sent from the IS. The AM developer uses the IS API to map these directives onto the AM-internal behaviour. *Run* is accompanied by a sub-list of the requested resources that the AM can manage, so partial conflicts can be handled without suspending the entire manager. *Stop* shuts down the AM. *Suspend* backgrounds the AM (the AM developer determines the actual AM-internal semantics). *Restrict* is used to partially suspend an AM where potential conflict is discovered for a subset but not all of its management activities and is only used when the IS is configured to operate in the SAFE_COEXISTENCE mode (see later). *Resume* reactivates a suspended AM. *Throttle* provides for a more-sophisticated adjustment of AM behaviour in which the IS can specify different rates of management activity to potentially conflicting AMs to prevent certain oscillatory patterns developing.
2. IS-IS interaction is facilitated by a single interface.
   ICommunicate {*Forward*, *Locate*, *Elect*, *SetISLevel*, *GetISLevel*} supports hierarchical operation, necessary in large or complex systems when AMs operate at different levels within a system and may be involved in local or system-wide conflicts. *Forward* is used to pass messages between the Global IS instance and local ISs which want to control or impact on global-level resources (e.g., communication between low and high level scheduling managers); this is the basis of system-wide and cross-level conflict detection. The remaining functions support the hierarchical IS structure itself including leader election for robustness. *Locate* returns the ID of the current service coordinator IS instance (which also performs the role of global conflict detection). *Elect* initiates an election if no coordinator instance is found. *SetISLevel* is used to set the IS level status to be either Local or Coordinator. *GetISLevel* is used by each IS instance to determine its status during *Locate* and *Elect* events.
3. The IS provides an external management interface.
   IConfigure {*SetMode*, *GetMode*, *SetSensitivity*, *GetSensitivity*, *StatusReport*} is a configuration and reporting interface which allows external system management utilities to perform system-specific configuration and generate status reports and statistics. *SetMode* and *GetMode* allow run-time configuration of the service to allow different levels of safety; 'SAFETY_CRITICAL' requires that all of a particular AM's management activity is suspended when it is

found to be involved in a conflict, whilst 'SAFE_COEXISTENCE' allows partial suspension of AM functionality, such that only non-conflicting management activities continue. The IS is initialised to SAFETY_CRITICAL mode. *SetSensitivity* and *GetSensitivity* are used to configure the conflict detection sensitivity level (see section IV, part D) and to dynamically adjust this if necessary. *StatusReport* collects status information and statistics for report generation and IS performance monitoring.

The IS architecture specification defines the interfaces, and with its accompanying communication protocol, defines the message formats and sequences that form the inter-component communication. It also specifies the semantics of this communication. Figure 3 shows how the IS functionality is integrated with the various components of the system.

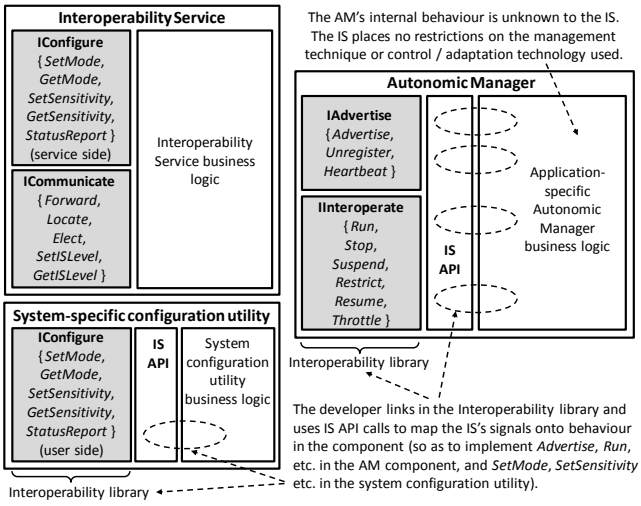


Figure 3. Internal architecture of the system components and the integration of the IS interfaces with these components.

The software developer retains flexibility with respect to the internal design and behaviour of the business logic of AM components and system configuration utilities. The architecture specification does not restrict the management approach, internal structure or control / adaptation techniques used within an AM component. However, the AM developer must integrate the API calls into the manager such that the control behaviour meets the IS specification (i.e., to interpret the directives {stop, suspend etc.} so that the AM's behaviour adheres to the respective IS semantics). Where an AM manages multiple resources the developer can choose to implement *Restrict* such that it is effective at the level of the AM itself, or only on the management activity that has been notified as being in conflict. In contrast *Suspend* always acts at the level of the entire AM. Similarly, the developer can decide the AM-internal semantics of *Suspend* and *Restrict* so as to isolate the management output (effecter output) of the manager whilst still running the monitor, analyse and plan parts if desired. This approach facilitates the IS' regulatory control over the AM when conflicts occur, whilst enabling 'warm' start-ups of components when conflicts are resolved.

B. The IS AM-state model

The IS maintains an instance of a state model for each locally registered AM (see Figure 4). The information held in these models drives the IS conflict management behaviour and is the basis on which AMs' management rights are governed.

An AM is discovered when it registers its management interests with its local IS instance. If there are no other AMs registered the new AM is granted management rights for the resources requested and signalled that it can run. If other AMs are already registered, the IS evaluates whether or not there is a possible conflict of interest, and if so signals the AM to either Stop (in which case the AM must attempt re-registration at a later time driven by some external event) or Suspend (in which case the IS will automatically signal the AM that it can resume, i.e., manage, once the conflict has been resolved).

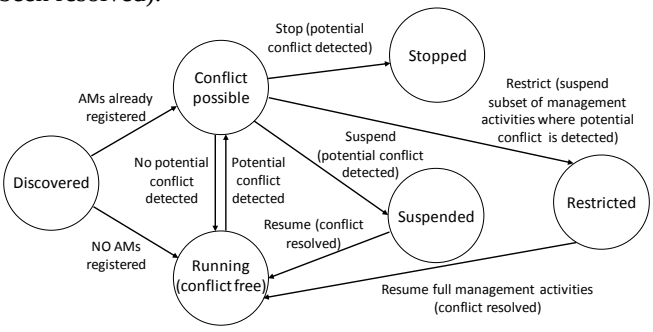


Figure 4. State diagram held by an IS instance, for each locally registered AM.

C. A Management Description Language

We discuss the need for a standard description of AMs' management interests, and briefly introduce our current language which is extensible to accommodate improvements in our understanding of ways actual and potential conflicts arise.

The IS facilitates interoperability (in the most limited case: safe coexistence) amongst (unknown in advance) AMs which have been developed independently of each other, and thus do not directly support interoperability amongst themselves.

The overall goal is to maximise the management freedom of AMs whilst at the same time ensuring that the system remains stable. To fulfil its main role, the IS must also:

- Detect AMs and learn their characteristics (via AM registration);
- Identify situations where conflicts can potentially occur, determine the consequences and the level of risk, and

achieve a system-specific balance when taking decisions to resolve conflicts by restricting, suspending or stopping AMs' management activities;
- Automatically enable the not-in-conflict subset of management activities for *restricted* AMs;
- Automatically resume suspended AMs when conflicts are resolved (e.g., on the basis or re-evaluating potential conflict status when other AMs leave the system);
- Enable cooperation between AMs. For example to share learnt knowledge concerning system state, volatility etc.

To perform these functions, the IS needs certain information detailing each AMs' management domain and specific resources of interest. This information must use a standard language format, and a fixed vocabulary of key terms so that automated searching for overlaps of interest can be performed effectively. The information will be provided at run time by the AM via the IS API (the information is provided ultimately by the AM developer).

Conflicts can arise in several ways. Direct conflicts occur where multiple AMs attempt to manage the same resource or object. However conflicts can be indirect (and less obvious) because a manager's activity may impact resources other than those directly managed. Categories of this include *cross-application* conflicts, for example increasing a specific application's use of a particular resource such as network bandwidth reduces the availability of bandwidth available to other applications. Another category of indirect conflicts are *cross-resource* conflicts, for example increasing processor speed to maximise throughput increases direct power usage and may also increase power requirements for cooling systems (which may have their own autonomic management systems). Some system characteristics such as security policy, power usage, server provisioning strategy etc. may be managed at both the system-wide level, and locally at the level of individual computing node or cluster. This can lead to conflicts between global and local managers, resulting in parts of the system being out-of step with global policy, and/or inefficient behaviour.

Clearly, it is difficult to identify every possible case of indirect conflict with certainty, and the *extent* of management impact in such cases is also highly variable. Therefore the description information provided by AMs must be sufficient to derive a similarity measure between their management effects. The language needs to contain appropriate categories to express areas of management concern in a structured way, i.e., from high-level domain in which the manager operates down to specific resources that are managed, and also to express characteristics including the management scope (global or local) and specificity (e.g., organisation specific, application specific).

Given these requirements, the standard management description should include:

Category. Mandatory. The highest-level and most generic descriptor used to identify the AM's domain of interest. Terms include:
{*Power general*, *Performance general*, *Security general*, ...}

Zone. Mandatory. A second level, more specific sub-category enabling developers to differentiate between specific management functions. Terms include:
{*Power system*, *Power platform*, *Power cooling* ... *Performance system*, *Performance CPU*, *Performance disk*, *Scheduling*, *VM management*, ... }

Impact. Mandatory. A numerical indicator Impact Factor (IF), (where $0 < IF \leq 1$), is defined to express the strength of the management influence. A directly controlled resource or parameter is assigned the value 1. A value close to 0 indicates that the particular AM has a weak influence on the resource whilst values close to 1 indicate that the resource is closely impacted by changes to one that is directly managed by the AM. For example an AM directly controlling CPU speed (IF = 1) has a strong indirect influence on VM performance ($IF \approx 0.8$). Term: { *ImpactFactor*(value) }

Scope. Mandatory. Whether the manager has local or global impact. Terms: { *Local*, *Global* }

Specificity. Optional. The extent of manager operation. Terms include: { *System-wide*, *Application-wide*, *Platform-wide*, *Process-wide*, *User-specific*, ... }

Trigger. Optional. This facilitates expression of temporal aspects such as periodicity or operating timescale, as well as specific events that invoke the management activity. Such characteristics can potentially be used to detect combinations of AMs at risk of causing of instability in the form of oscillation or control divergence for example. Terms include: { *Period*(value), *Event*(name) , ... }

Parameter. Optional. Identification of specific context parameters that are of interest to the AM. Term:
{ *Name*(value) }

Envelope. Optional. Expression of range of control freedom for a given named Parameter. This can potentially help to avoid false positive detections of conflict, when managers operate in the same domain but have non-overlapping envelopes of operation. Terms include:
{ *Name*(range, value) }

Where provided, the Envelope term allows more precise determination of the risk of conflict in cases where a pair of AMs both declare an envelope value for a specific parameter. Where an AM does not declare an envelope value for any given Parameter the full state space of values is assumed.

D. Conflict Detection

The architecture specification does not mandate the actual conflict detection technique to be used; this is an

implementation decision and will be based on the level of sophistication required in a particular system.

In our exploratory work conflict detection is based on calculating a numerical measure of similarity between the management interests of a pair of AMs, and comparing this measure with a sensitivity threshold level. A newly registering AM's management description is compared with those of the already registered AMs.

The technique is described below and an example implementation is outlined in section V.

The architecture specification defines a dynamically configurable conflict sensitivity threshold ($0 < Thresh_C \leq 1$) which is used to tune the conflict detection sensitivity (via *SetSensitivity*, on **IConfigure**). A potential conflict is detected if the similarity match measure *Match* of a pair of AMs exceeds $Thresh_C$. The sensitivity level is configured by the facility manager via a control console application (or tuning of this parameter could be automated), and can be changed at run time as necessary. This enables safety critical systems (for example) to operate pessimistically with very low tolerance to potential manager conflicts, whereas in domains where only efficiency (for example) is at stake, the system can operate more optimistically, with a higher tolerance which can lead to benefits of having a greater number of AMs working simultaneously (bearing in mind that a 'potential conflict' may not be realised).

V. IMPLEMENTATION

This section describes a work-in-progress implementation which employs a subset of the extensible architecture's characteristics for demonstration of the core behaviour. Here we focus on the operation of the service at a local level, since it is intuitive to expect that many conflicts between autonomic managers will be localised due to decisions concerning local resources, or configurations of local services.

The IS maintains a table which contains the identity and state of each registered AM, and a second table which keeps track of each AM's directly managed and indirectly impacted resources (see figure 5). Information in this table comprises: AM_ID (a value allocated to the AM by the IS during the discovery process); General area of management function (a 'category' term from the management description language); Sub-classifier of management function (a 'zone' term from the management description language); Managed parameter name ACItem_ID (the optional 'parameter' term from the management description language); Conflict status and Impact Factor for the related resource; and Scope (a 'scope' term from the management description language). Figure 5 also shows the communication that takes place between an AM and the IS. MAdvertise, MRelease and MHeartbeat are messages sent from the AM via actions on the IAdvertise interface. MACK / MNACK are Acknowledge / Not Acknowledge responses to management requests accompanying MAdvertise. This works as follows: the AM tries to register (Advertise) its management interests one by one and the IS replies with MNACK messages if any are in conflict with the rest of the system, MACK otherwise. MSuspend, MResume, MRun, MStop and MThrottle are directives sent by the IS via the IInteroperate interface.

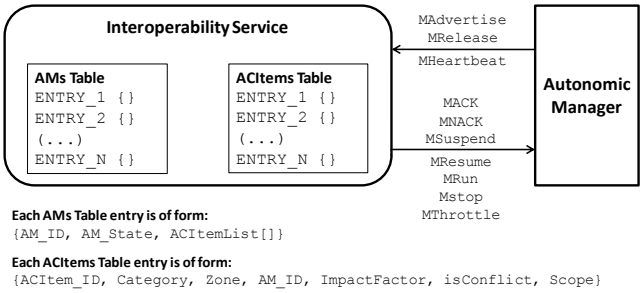


Figure 5. The IS' internal data tables, and overview of the AM-IS communication protocol.

For initial exploration we use a conflict detection technique based on a numerical similarity measure of AMs' management interests. Conflict detection activity is triggered by events that change the population or configuration of the AMs; such as the registration of a newly-discovered AM, or the departure of an AM from the system.

For a pair of AMs $\{AM_i, AM_j\}$ the similarity measure $Match_{ij}$ is derived from the management descriptions of the AMs as follows:

Let N_i = name of the specific managed resource (specified by the Parameter term in the management description),

C_i = management category,

Z_i = management zone,

IF_i = impact factor (of AM_i on the resource identified by $\{N_i, C_i, Z_i\}$),

S_N, S_C, S_Z = similarity indicator of management description terms Name, Category and Zone respectively for the pair of AMs.

$$Match_{ij} = \frac{S_N + S_C + S_Z + IF}{4}$$

where:

$$S_N = \begin{cases} 1 & \text{when } N_i = N_j \\ 0 & \text{when } N_i \neq N_j \end{cases},$$

$$S_C = \begin{cases} 1 & \text{when } C_i = C_j \\ 0 & \text{when } C_i \neq C_j \end{cases},$$

$$S_Z = \begin{cases} 1 & \text{when } Z_i = Z_j \\ 0 & \text{when } Z_i \neq Z_j \end{cases},$$

$$IF = \frac{IF_i + IF_j}{2}.$$

IF values are normalised, i.e., $IF_i, IF_j \in (0,1]$, thus the resulting similarity measure will always be a normalised value $Match_{ij} \in (0,1]$.

A newly registering AM's management interests are compared with details of each already registered AM, at the local IS instance in most cases. This is performed independently for each resource pair combination; so if AM_i and AM_j are registered with declared management interests in *m* and *n* resources respectively, and AM_k attempts to register *p* resource management interests, then *mp* + *np* similarity measures are generated.

A potential conflict is detected if for any pair of AMs {i,j}, $Match_{ij}$ exceeds the conflict sensitivity threshold ($Thresh_C$).

When evaluating the scalability of the approach it is important to consider: 1. conflict detection occurs predominantly at the level of the local IS instance; only in cases where an AM's resource description has global scope does the conflict detection get invoked at the global level; 2. conflict detection is only performed when events that affect the AM population occur (e.g., AMs arriving, leaving); and 3. whilst we do not limit the number of AMs registered at a local IS instance, we expect this number to be of order 10, or perhaps 100 rather than much bigger values, for realistic systems.

The dynamically configurable operating mode of the IS determines what action is taken once a potential conflict has been detected. If the IS mode is SAFETY_CRITICAL, AM_k will be *suspended* (i.e., management activities are inhibited at the level of the AM itself). In SAFE_COEXISTENCE mode AM_k will be *restricted*, (i.e., management activities are inhibited at the level of specific resources managed by a particular AM; it is allowed to perform its normal management operations for the not-in-conflict subset of its management domain). The actual semantics for restricted AM-internal operations are to some extent implementation specific. In some cases it will be desirable to enable the *monitoring* aspect to operate as normal (to prevent discontinuity in monitoring traces etc., and to facilitate warm restarts of restricted operations), but in all cases the *effecter* is switched off, i.e., the manager can monitor its environment but cannot change anything.

The current implementation uses policy-based management logic within AMs; and is based on Agile++ [16], [17]. Agile++ has language components including Rules, Variables and Actions. Under typical normal behaviour, a Rule will be evaluated to determine which Action needs to be performed, using Environment Variables to reflect external inputs to the Rule and Output Variables to signal the result of an Action. *Restricted* mode has been implemented for conflicting operations such that the AM still evaluates its control policy and executes Actions within, as normal. However, Output Variables are disabled (value forced to NULL) so that the Action can continue to make internal updates (such as for external-state tracking) but cannot actually effect the external system state.

As an alternative to using the IAdvertise interface for AMs to register their management interests, the implementation supports the encoding of the Management Description Language in XML format. An example configuration file is shown in Figure 6.

```
<!-- Autonomic Manager Configuration Specification Language -->
<MetaData>
  <ConfigAuthor Name="Mariusz Pelc" Organisation="UoG" />
  <TimeStamp Time="12:00" Date="20/12/2010" />
    <AMDescription>
      <AM ID="AM1">
        <ACItems>
          <ACItem ID="Performance" Scope="Local">
            <Category>Performance General</Category>
            <Zone>CPU Performance</Zone>
            <ImpactFactor>1.0</ImpactFactor>
          </ACItem>
          <ACItem ID="Power" Scope="Local">
            <Category>Performance General</Category>
            <Zone>System Performance</Zone>
            <ImpactFactor>0.5</ImpactFactor>
          </ACItem>
        </ACItems>
      </AM>
    </AMDescription>
</MetaData>
```

Figure 6. XML representation of the Management Description Language

A. Wider Architectural Perspective

The IS implementation forms part of a wider project to develop a full component model and middleware for autonomic computing which has been ongoing at Greenwich for several years, see for example [18], [19]. Full details of this are out of scope for this paper, but in brief, this is a policy-based system in which services including communication manager, context manager, repository manager and now the IS are optionally policy supervised. The middleware supports policy-based application-specific components which can have dynamic (run-time) policy upgrades and which have in-built fault recovery. For example if a new policy is loaded but its required context information is not available from the context manager then an automatic roll-back to a previously working policy is performed. Architectural support for low-resourced embedded platforms is also included.

B. Evaluation Application Scenario

Data centre management is a popular application domain for AC; due in part to the high configuration complexity that arises from the scale of operation, and also because with such large amounts of resources deployed the potential efficiency savings are very high. AC currently targets several key aspects of data centres, including power management to reduce running costs, and scheduling to improve resource efficiency. We demonstrate the operation and benefit of the IS in a data centre scenario in which two independently developed AMs coexist (managing power usage, and processor scheduling, respectively); their management operations potentially conflicting.

The scenario: The scheduling manager (AM1) has a main goal of maximising throughput by keeping all resources utilised where possible. The power manager (AM2) is designed to minimise power usage by slowing down processor speed or by shutting down entire processor units where possible. We assume that, in the absence of other managers, each of these services has been extensively evaluated and found to improve overall performance.

The co-existence of these AMs creates a high potential for conflict. For example AM2 will attempt to shutdown an underutilised resource as soon as load level starts to fall, whilst AM1 will attempt to bring unused resources into play as soon as load levels increase (or a backlog develops). Depending on the sequence of load level changes it is possible that oscillation will build up between the actions of these two managers.

Operation: During its initialisation each AM registers with the IS. The management capabilities of each AM are described using the standard language and categories described earlier.

AM1 directly controls a parameter *performance* within the general management category *performance general*, and specific sub-zone *CPU performance*; and indirectly influences a parameter *power* within the general category *performance general*, and sub-zone *system performance*.

AM2 directly controls a parameter *power* within the general category *power general*, and the specific zone of interest *system power*; and indirectly influences a parameter *performance* within the general category *performance general*, and the specific zone of interest *CPU performance*.

```
a) AddACItem     ("Performance", "Performance General",
                  "CPU Performance", "1.0", "Local");
   AddACItem     ("Power", "Performance General",
                  "System Performance", "0.5", "Local");
   RegisterAsAM  ();

b) AddACItem     ("Power", "Power General",
                  "System Power","1.0","Local");
   AddACItem     ("Performance", "Performance General",
                  "System Performance", "0.5", "Local");
   RegisterAsAM  ();

c) bool AddACItem(char *ParameterName, char *Category,
                  char *Zone, char *Impactfactor, char *Scope);
```

Figure 7. API calls to register AM's management interests.

The API calls to perform the manager registration with the IS are shown in Figure 7a (for AM1), and 7b (for AM2), where AddACItem means 'Add autonomically controlled item'; its template is shown in Figure 7c.

VI. EVALUATION

As mentioned in section V, part A this work forms part of a larger project to develop a full component model and middleware for autonomic computing. We use the existing infrastructure as a testbed to evaluate the IS in a realistic system setting.

In addition to the IS, three additional system services are provided to create a run-time environment in which the behaviour of the IS and AMs can be evaluated, these are: Communication Manager; ContextManager and RepositoryManager. In addition, a couple of services were fabricated to provide mock context values for two system parameters which are needed as inputs in the run-time execution of various control policies used in the experiments. The EfficiencyProvider component generates the 'Efficiency' parameter, and likewise the LoadProvider component generates the 'Load' system parameter.

The services are integrated into a middleware component (available in the form of shared library for Linux) with API interface enabling communication, context and repository management, conflict resolving and policy evaluation.

Two IS-compliant AMs (AM1, AM2) have been developed to evaluate and demonstrate the behaviour of the Interoperability Service. AM1 and AM2 target popular management domains within cloud / grid computing, typical of autonomic control systems currently deployed in data centre systems for example. The whole application (including the AMs) thus comprises of 8 services. Figure 8 provides a snapshot of the system in operation during scenario 5 (see below), showing clockwise from top left: Communication Manager, Context Manager, Interoperability Service, AM2, AM1, and the Repository Manager.

The management domains of AM1, AM2 respectively are: processor scheduling (with the goal of maximising throughput by keeping resources utilised where possible), and power management (with the goal of minimising power usage). This is a realistic situation in which the direct management activities are well differentiated, but in which there is an indirect conflict as discussed in section IV, part C.

The AMs are designed so as to be representative of *independently developed* components operating in a data-centre system, i.e., the AMs include no direct support for co-existence or interoperability amongst themselves. The evaluation is performed in 5 scenarios. The first four scenarios show the behaviour of the IS when operating in SAFETY-CRITICAL mode under a range of different resource management circumstances. The fifth scenario shows how the IS responds to AM conflicts when the IS is operating in SAFE-COEXISTENCE mode.

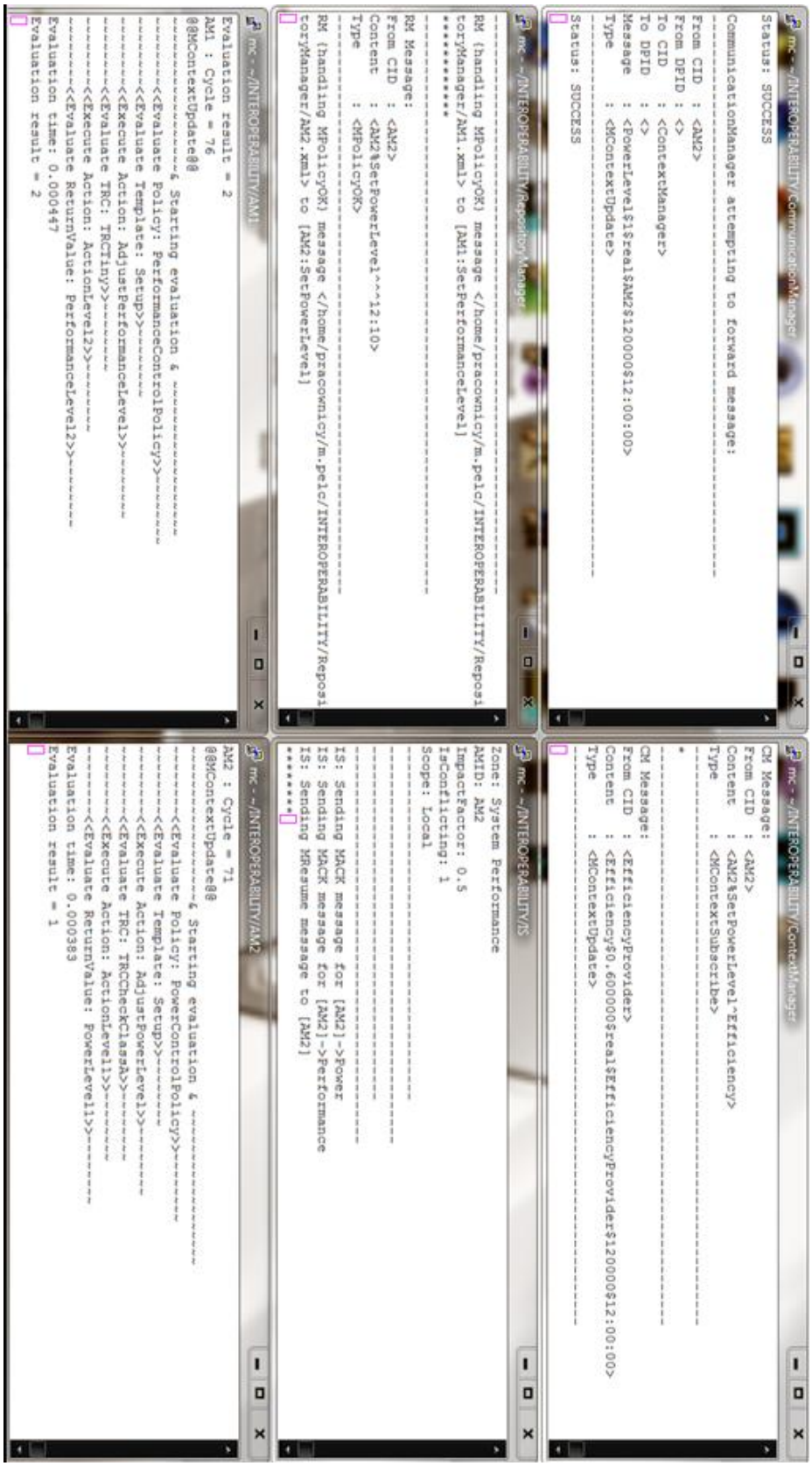


Figure 8. The system in operation during the evaluation.

Scenario 1 illustrates the standalone manager case, and is included for completeness. Each manager registers separately in the system in the absence of the other. $Thresh_C$ = 0.6. AM1 requests management rights for CPU performance, and also notifies a potential impact on system power. As there are no other AMs present, the IS grants AM1 permission to manage unimpeded. Similarly, for AM2 (in the absence of AM1) the IS grants rights to manage system power level and also to have an indirect impact on system performance.

Scenario 2 illustrates the case where a potential conflict is detected between a pair of managers (IS operating in SAFETY-CRITICAL mode). AM1 registers with the IS and is granted rights to manage the resources it has requested. AM2 then registers whilst AM1 is still present. $Thresh_C$ = 0.6. The IS performs conflict detection analysis, based on the AMs' announced Impact Factors (IFs) for each requested managed item. This determines whether AM2 can be granted the requested management rights: *Power* directly managed (IF=1.0), and *Performance* potentially affected indirectly (IF=0.5). The match levels are determined using the algorithm presented in section V. In this case a conflict is detected; arising from AM1's direct management of performance and AM2's indirect impact on performance, giving a match value greater than the threshold. This can be seen in the diagnostic trace in figure 9.

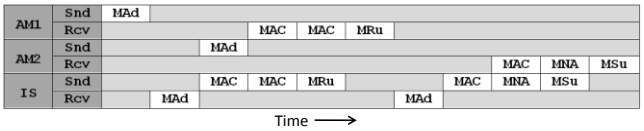
```
IS: Handling Advertise Message:
IS: Conflict Detection [AM2->Power]::[AM1->Performance]
IS: Match Level=0.25, Threshold=0.6
IS Decision: No Conflict Detected
IS: Conflict Detection [AM2->Power]::[AM1->Power]
IS: Match Level=0.4375, Threshold=0.6
IS Decision: No Conflict Detected

IS: Conflict Detection [AM2->Performance]::[AM1->Performance]
IS: Match Level=0.6875, Threshold=0.6
IS Decision: Conflict Detected
IS: Conflict Detection [AM2->Performance]::[AM1->Power]
IS: Match Level=0.625, Threshold=0.6
IS Decision: Conflict Detected

IS: Sending MACK message for [AM2]->Power
IS: Sending MNACK message for [AM2]->Performance
IS: Sending MSuspend message to [AM2]
```

Figure 9. A potential conflict is detected.

Figure 9 shows a diagnostic trace of the IS conflict detection process, in which the advertised management interests of AM2 are compared for all relevant AMs. In this specific case AM1 is already managing a system performance characteristic (specifically CPU performance), when AM2 registers, requesting to manage system power, but also announcing a potential impact on system performance. The IS does not detect a direct conflict with the power management, but the conflict match level for system performance exceeds the current $Thresh_C$ (0.6). The IS suspends the newly registering manager to prevent possible instability (this manager will be automatically resumed if AM1 leaves the system and there are no other conflicts with other AMs registered in the meantime). Figure 10 shows the resulting message sequence.




```
Key:  Snd - Sent Message          MNA - MNACK          MRu - MRun
      Rcv - Received Message      MRl - MRelease       MSp - MStop
      MAd - MAdvertise Message    MRe - MResume
      MAC - MACK Message          MSu - MSuspend
```

Figure 10. Message sequence for scenario 2.

Scenario 3: As scenario 2, but with $Thresh_C$ = 0.8, i.e., the IS is less sensitive to potential conflicts (this configuration may be better suited to non-critical systems where some potential for conflict may be acceptable, i.e., the tradeoff between safety and management flexibility is shifted). The new diagnostic behaviour trace and the resulting message sequence are shown in Figure 11 and Figure 12 respectively. In this case no conflicts are detected and the newly arriving AM2 is granted rights to manage system power level, and to have an impact on system performance, thus potentially interacting with AM1.

```
IS: Handling Advertise Message:
IS: Conflict Detection [AM2->Power]::[AM1->Performance]
IS: Match Level=0.25, Threshold=0.8
IS Decision: No Conflict Detected
IS: Conflict Detection [AM2->Power]::[AM1->Power]
IS: Match Level=0.4375, Threshold=0.8
IS Decision: No Conflict Detected

IS: Conflict Detection [AM2->Performance]::[AM1->Performance]
IS: Match Level=0.6875, Threshold=0.8
IS Decision: No Conflict Detected
IS: Conflict Detection [AM2->Performance]::[AM1->Power]
IS: Match Level=0.625, Threshold=0.8
IS Decision: No Conflict Detected

IS: Sending MACK message for [AM2]->Power
IS: Sending MACK message for [AM2]->Performance
IS: Sending MRun message to [AM2]
```

Figure 11. IS conflict detection analysis in which the conflict match level is below the conflict threshold.

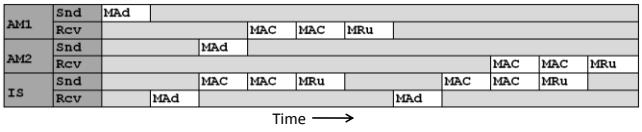


Figure 12. Message sequence for scenario 3.

Scenario 4 Illustrates the case where AMs are replicated and the IS must ensure that only a single instance is active at any time (note that the IS does not know that the two managers are identical, it bases its decisions only on the AMs' management descriptions). Manager AM1 registers and begins managing its advertised resource. A second instance of the *same manager type as AM1*, AM3, requests management rights from the IS. $Thresh_C$ = 0.6. The conflict detection procedure is not executed when AM1 registers as there are no other AMs registered with the IS. Thus AM1 is granted management rights for both resources requested. The registration of AM3, advertising a direct management interest in *Performance* and an indirect impact on *Power*, triggers conflict detection analysis, as shown in Figure 13.

In this case, conflicts are detected for both of the requested resources, so as a result, AM3 is suspended. At a later time, AM1 performs an orderly shutdown sending an

MRelease message to the IS, invoking the UnregisterAM function at the IS. This has 3 effects: 1. an *MStop* message is sent to AM1 (see Figure 14); 2. the IS unregisters all AM1's management interests; 3. conflict detection analysis is again triggered, now with the goal of detecting situations where previous conflicts have now been resolved. Any suspended AM's that are no longer in conflict with active managers are now resumed. In this case AM3 is the only suspended AM, and in the absence of any conflicts with active AMs it is automatically resumed and granted its requested management rights (see Figure 15).

```
IS: Handling Advertise Message:
IS: Conflict Detection [AM3->Performance]::[AM1->Performance]
IS: Match Level=1, Threshold=0.6
IS Decision: Conflict Detected
IS: Conflict Detection [AM3->Performance]::[AM1->Power]
IS: Match Level=0.4375, Threshold=0.6
IS Decision: No Conflict Detected

IS: Conflict Detection [AM3->Power]::[AM1->Performance]
IS: Match Level=0.4375, Threshold=0.6
IS Decision: No Conflict Detected
IS: Conflict Detection [AM3->Power]::[AM1->Power]
IS: Match Level=0.875, Threshold=0.6
IS Decision: Conflict Detected

IS: Sending MNACK message for [AM3]->Performance
IS: Sending MNACK message for [AM3]->Power
IS: Sending MSuspend message to [AM3]
```

Figure 13. Conflict detection analysis finds potential conflicts of interest between two instances of the same AM type.

```
IS: Handling Release Message:
IS: Sending MStop message to [AM1]
```

Figure 14. IS receives *MRelease*, responds with *MStop*.

```
List of Suspended AMs:
-----------------------------------------------
AM Name: AM3
AM State: SUSPENDED
-----------------------------------------------
IS: Sending MACK message for [AM3]->Performance
IS: Sending MACK message for [AM3]->Power
IS: Sending MResume message to [AM3]
```

Figure 15. IS resumes the AM3 Manager

Figure 15 illustrates the IS's behaviour on receipt of an *MRelease* message, which implies that an AM has left the system and thus one or more previously detected conflict conditions may have been removed. First the state model is searched for any AMs in the SUSPENDED state. The management interests of these are re-examined against those of the remaining RUNNING state AMs (conflict detection analysis is triggered again). Any suspended AMs which are now conflict-free are resumed (AM3 in this case). Figure 16 shows the entire message sequence for scenario 4.

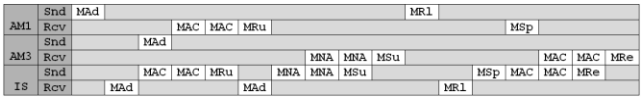


Figure 16. Message sequence for scenario 4.

In addition to illustrating the prevention of conflicts of directly overlapping management interest; scenario 4 also shows how the IS architectural approach facilitates and manages redundant replication of autonomic manager processes for robustness within a system. Only one AM is given management rights for a particular resource at any time, but whenever an AM leaves the system the set of running and suspended AMs is automatically re-evaluated for changes in conflict status. Suspended replicas are resumed when determined conflict-free, and can start 'warm' because the AM's developer can choose to implement '*suspend*' as only shutting down the execute stage of the MAPE loop.

Scenario 5 is the equivalent of scenario 2, except that in this case the IS operates in SAFE-COEXISTENCE mode. AM1 registers its management interests with the IS, followed by AM2. $Thresh_C$ = 0.6. The two Autonomic Managers attempt to control respectively, Performance (direct control with IF=1.0) and Power (indirect control with IF=0.5) for AM1 and Power (direct control, IF=1.0) and Performance (indirect, IF=0.5) for AM2.

As there are no other AMs running when AM1 registers it is granted full management rights, as shown in figure 17.

```
IS: Handling Advertise Message:
IS: Sending MACK message for [AM1]->Performance
IS: Sending MACK message for [AM1]->Power
IS: Sending MRun message to [AM1]
```

Figure 17. IS issues full rights to the AM1 Manager

When AM2 registers its management interest the IS checks for a conflict with all other registered managers. As a result the IS allows AM2 to control Power but **restricts** controlling Performance and sends an MRestrict message to AM2 as the diagnostic trace in figure 18 shows.

```
IS: Handling Advertise Message:
IS: Conflict Detection [AM2->Power]::[AM1->Performance]
IS: Match Level=0.25, Threshold=0.6
IS Decision: No Conflict Detected
IS: Conflict Detection [AM2->Power]::[AM1->Power]
IS: Match Level=0.3875, Threshold=0.6
IS Decision: No Conflict Detected
IS: Conflict Detection [AM2->Performance]::[AM1->Performance]
IS: Match Level=0.6875, Threshold=0.6
IS Decision: Conflict Detected
IS: Conflict Detection [AM2->Performance]::[AM1->Power]
IS: Match Level=0.575, Threshold=0.6
IS Decision: No Conflict Detected
IS: Sending MACK message for [AM2]->Power
IS: Sending MNACK message for [AM2]->Performance
IS: Sending MRestrict message to [AM2]
```

Figure 18. A potential conflict is detected; AM2 is restricted.

In the Restricted mode AM2 evaluates its policy as normal but the Performance output variable is set to NULL, i.e., AM2 cannot actually effect the system performance whilst restricted in this management aspect. AM2 manages power normally, as this aspect was not restricted.

Later, AM1 Unregisters with the IS, this again triggers conflict check operation. AM2 is no longer in conflict, so is now granted permission to control all items of interest, as shown in the trace in figure 19.

```
IS: Handling Release Message:
delete AMDesc: AM1
IS: Sending MStop message to [AM1]
List of Restricted AMs:
-----------------------------------------------
AM Name: AM2
-----------------------------------------------
ACItem Name: Power
Category: Power General
Zone: System Power
AMID: AM2
```

```
ImpactFactor: 1.0
IsConflicting: 0
Scope: Local
----------------------------------------
----------------------------------------
ACItem Name: Performance
Category: Performance General
Zone: System Performance
AMID: AM2
ImpactFactor: 0.5
IsConflicting: 1
Scope: Local
----------------------------------------
------------------------------------------------
IS: Sending MACK message for [AM2]->Power
IS: Sending MACK message for [AM2]->Performance
IS: Sending MResume message to [AM2]
```

Figure 19. Diagnostic trace showing IS behaviour during release of AM1 and subsequent granting of full rights to AM2

Figure 20 shows the entire message sequence for scenario 5.

A. Evaluation summary

The evaluation aspect of this paper is mainly concerned with demonstration of the IS concept. Our implementation is not necessarily optimised for processing performance. We have focussed the evaluation on exploring the behaviour of the system in a set of base cases which represent realistic types of conflicts (direct and indirect) between AMs.

The evaluation was based on a number of ways in which a pair of AMs may overlap in their management activities. These scenarios were chosen so as to reflect a wide range of possibilities.

The case results show how the IS controls the management rights of AMs dynamically, using the management similarity-measure based conflict detection. We have demonstrated the variable safety-sensitivity of the service, using the configurable sensitivity threshold combined with the choice of two safety levels (SAFTEY-CRITICAL and SAFE-COEXISTENCE).

The processing overhead of conflict detection does not increase significantly when larger populations of AMs exist, because conflict detection is only triggered when the AM population changes (e.g., a new AM is registered), and the existing AMs are only compared against the arriving AM (not against each other). The conflict detection always considers AMs on a pair-wise basis, so functional complexity remains the same regardless of the number of AMs present.

VII. CONCLUSION

In this paper, we have outlined the case for greater research effort in the area of interoperability of autonomic managers. We have discussed why bespoke and custom solutions will not work in the long term and argued for a universal standard for interoperability. In line with this we have identified requirements for a service-based approach.

We are working towards standards and services for universal interoperability in autonomic systems. In particular we are targeting the under-addressed challenge of interoperability and co-existence in not-planned circumstances, i.e., for AMs that are developed independently and brought together when systems are built from a number of separate components, and also when existing systems or components are upgraded.

This work is timely and important because the likelihood of conflicts will escalate as autonomic computing continues to increase in popularity, and AMs are deployed in an ever-wider array of components with ever-richer functionality.

We have presented initial work towards a service-based automatic and proactive interoperability service, being integrated into autonomic components and making them ‘interoperability ready’ in advance of their deployment. Our approach enables AMs to be developed independently, requiring that the developer uses a management description language to describe the component’s management characteristics. This approach has the main advantage of not requiring an AM developer to have knowledge of other AM’s that may exist in the target system, and thus supports agility i.e., configuration changes, expansion and upgrades.

The technique has been developed with generalisation as a main goal. In the same way that it is not possible when developing an AM to perceive all the possible other AMs and their management capabilities that could coexist; it is also not possible when developing an IS to predict all of the application domains and behaviours of future AMs. Therefore we have ensured that the language used to describe management capabilities is extensible, and can be represented using a standard format (XML). The architecture defines the interfaces and communication between the key management components of the system but leaves open the implementation decisions for the IS-internal business logic so it can be tailored to a system’s needs.

The demonstration-of-concept implementation has focussed initially on ‘safe coexistence’ as a mandatory foundational step towards universal AM interoperability. Further work focuses on more-sophisticated techniques for the conflict detection, and further refinement of the management description language on which the conflict detection is based.

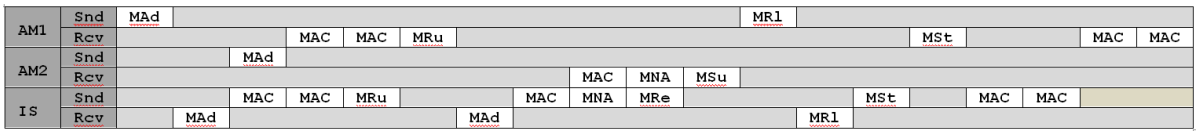


Figure 20. Message sequence for scenario 5.

VIII. REFERENCES

[1] Anthony R, Pelc M, and Shuaib H, The Interoperability Challenge for Autonomic Computing, The Third International Conference on Emerging Network Intelligence (EMERGING 2011), Lisbon, Portugal, November 20-25, 2011, pp. 13-19, IARIA, ISBN 978-1-61208-174-8.

[2] Kephart J. O., Chan H., Das R., Levine D. W., Tesauro G., Rawson F., and Lefurgy C. 2007. Coordinating multiple autonomic managers to achieve specified power-performance tradeoffs. In *Proc. 4th Intl. Conf. on Autonomic Computing* (Jacksonville, FL, USA, June 2007). ICAC'07. IEEE, 1–9.

[3] Wang M., Kandasamyt N., Guezl A., and Kam M. 2006. Adaptive performance control of computing systems via distributed cooperative control: Application to power management in computing clusters. In *Proc. 3rd Intl. Conf. on Autonomic Computing* (Dublin, Ireland, June 2006). ICAC'06. IEEE, 165–174.

[4] Zhao M., Xu J., and Figueiredo R. J. 2006. Towards autonomic grid data management with virtualized distributed file systems. In *Proc. 3rd Intl. Conf. on Autonomic Computing* (Dublin, Ireland, June 2006). ICAC'06. IEEE, 209–218.

[5] Khargharia B., Hariri S., and Yousif M. S. 2006. Autonomic power and performance management for computing systems. In *Proc. 3rd Intl. Conf. on Autonomic Computing* (Dublin, Ireland, June 2006). ICAC'06. IEEE, 145–154.

[6] Xu J., Zhao M., Fortes J., and Carpenter R. 2007. On the use of fuzzy modeling in virtualized data center management. Autonomic Computing, 2007. In *Proc. 4th Intl. Conf. on Autonomic Computing* (Jacksonville, FL, USA, June 2007). ICAC '07. IEEE Computer Society.

[7] Wang R., Kusic D. M., and Kandasamy N. 2010. A distributed control framework for performance management of virtualized computing environments. In *Proc. 7th Intl. Conf. on Autonomic Computing* (Washington DC, USA, June 2010). ICAC'10. IEEE, 89–98.

[8] Ghanbari S., Soundararajan G., Chen J., and Amza C. 2007. Adaptive learning of metric correlations for temperature-aware database provisioning. In *Proc. 4th Intl. Conf. on Autonomic Computing* (Jacksonville, FL, USA, June 2007). ICAC'07. IEEE.

[9] Kutare M., Eisenhauer G., and C. Wang. 2010. Monalytics: Online monitoring and analytics for managing large scale data centers. In *Proc. 7th Intl. Conf. on Autonomic Computing* (Washington DC, USA, June 2010). ICAC'10. IEEE, 141–150.

[10] Zhu X., Young D., Watson B. J., Wang Z., Rolia J., Singhal S., McKee B., Hyser C., Gmach D., Gardner R., Christian T., and Cherkasova L. 2008. 1000 islands: Integrated capacity and workload management for the next generation data center. In *Proc. 5th Intl. Conf. on Autonomic Computing* (Chicago, IL, USA, 2008). ICAC '08. IEEE, 172–181.

[11] Poussot-Vassal C., Tanelli M., and Lovera M. 2010. A Control-Theoretic Approach for the Combined Management of Quality-of-Service and Energy in Service Centres. In *Run-time Models for self-managing Systems and Applications*. Ardagna D and Zhang L, Eds). Springer Basel AG. 73-96.

[12] White S. R., Hanson J. E., Whalley I., Chess D. M., and Kephart J. O. 2004. An architectural approach to autonomic computing. In *Proc. 1st Intl. Conf. on Autonomic Computing* (New York, NY, USA, May 2004). ICAC'04. IEEE. 2-9.

[13] Kennedy C. 2010. Decentralised metacognition in context-aware autonomic systems: some key challenges. In *Proc. American Institute of Aeronautics and Astronautics* (AIAA) *Workshop on Metacognition for Robust Social Systems* (Atlanta, Georgia,) AAAI-10, AIAA. 34-41.

[14] Salehie M. and Tahvildari L. 2005. Autonomic computing: Emerging trends and open problems. In *Proc. Workshop on the Design and Evolution of Autonomic Application Software* (New York, NY, USA, 2005). DEAS'05. ACM Special Interest Group on Software Engineering. 30. 1–7.

[15] Quitadamo R. and Zambonelli F. 2008. Autonomic communication services: a new challenge for software agents. *SpringerLink Journal of Autonomous Agents and Multi-Agent Systems*. 17, 3 (2008), 457–475.

[16] Anthony, R. J. Policy-based autonomic computing with integral support for self-stabilisation, *International Journal of Autonomic Computing*, Vol. 1, No. 1, pp.1-33. ISSN (Online): 1741-8577, ISSN (Print): 1741-8569, 2009, Inderscience.

[17] P. Ward, M. Pelc, J. Hawthorne, and R. Anthony, Embedding Dynamic Behaviour into a Self-configuring Software System, *In Proc. 5th Intl Conf. on Autonomic and Trusted Computing* (ATC 2008), Oslo, Norway, Lecture Notes in Computer Science (LNCS 5060/2008), ISBN 978-3-540-69294-2, pp373-387, June 23-25, 2008, Springer-Verlag.

[18] Anthony R. J., Pelc M., Ward P., and Hawthorne J. 2009. A Software Architecture supporting Run-Time Configuration and Self-Management. *Communications of SIWN*. 7 (May. 2009), SIWN. 103-112.

[19] Pelc M., Anthony R. J., Ward P., and Hawthorne J. 2009. Practical Implementation of a Middleware and Software Component Architecture supporting Reconfigurability of Real-Time Embedded Systems. In *Proc. 7th IEEE/IFIP Intl. Conf. on Embedded and Ubiquitous Computing* (Vancouver, Canada, 2009). EUC'09. IEEE, 394-402.