The Interoperability Challenge for Autonomic Computing

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
H.Shuaib@gre.ac.uk

Abstract - **Interoperability is an emerging need for autonomic computing systems, which 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. This 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. We advocate a service-based approach to interoperability and present a set of requirements for such an approach. We briefly present a universal interoperability service which automatically discovers and manages potential conflicts between manager components.**

Keywords - Autonomic systems, Interoperability, Services

I. INTRODUCTION

Autonomic Computing (AC) is increasingly popular, and has become a mainstream concept. Autonomic components will soon be commonplace and it is inevitable that there will be an increasing trend of co-existence amongst autonomic managers. As there are currently no universal standards for autonomic systems design, or for the provision of interoperability amongst managers, there can be no guarantees that separately-designed managers will operate harmoniously together. Almost all systems use multi-vendor software solutions and this implies that there will be a great variety of potential 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.

Any multi-manager scenario leads to potential conflicts. Direct conflicts occur where Autonomic Managers (AMs) attempt to manage the same explicit resource. Indirect conflicts arise when AMs control different resources, but the management effects of one have an undesirable impact on the management function of the other. This latter type of conflict is expected to be the most frequent and problematic, as there are such a wide variety of unpredictable ways in which such conflicts can occur. The effects of indirect conflict will also be less obvious to detect and harder to diagnose than the direct conflicts. The effects of conflicts can vary widely, including e.g., a cancellation effect of opposing managers, and serious performance or stability problems. The problem is illustrated with an example: consider a system with two AMs: a Power Manager (PM1) which shuts down servers that have been idle for a short time; and a Performance Manager (PM2) which attempts to maintain a pool of idle servers to ensure high responsiveness to high priority applications. Each service was developed and evaluated in isolation and both performed perfectly, however the respective vendors did not envisage that they would co-exist. Bringing a shutdown server back on line has a latency of several seconds, thus PM1's 'locally correct' behaviour defeats PM2's contribution. As each manager is unaware of the presence and behaviour of the other, the 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.

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 a work-in-progress service-based interoperability service which enables exploration of these requirements (section IV). Section V 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 VI. The interoperability service is evaluated in section VII and finally we conclude (section VIII).

II. BACKGROUND

A clear demonstration of the need for interoperability mechanisms is provided in [1] where two independently-developed autonomic managers were implemented. The first dealt with application resource management, specifically CPU usage optimization. The second, 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 their intended optimisations and potentially destabilising the system. We envisage widespread repetition of this problem until a universal approach to interoperability is implemented.

Early work has focussed on bespoke interoperability solutions for specific systems. [2] proposes a distributed management framework that seeks to achieve system-wide Quality of Service (QoS) goals. Autonomic controllers are added and removed from the system based on applications' QoS requirements. 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. Other works take a more direct approach to autonomic element interaction. For instance, in [3] the autonomic elements that enable the proposed data grid management system communicate directly with one another to ensure that management obligations are met. The relationship between each type of autonomic element is peer-to-peer – potentially leading to high interaction complexity. In contrast, [4] adopts a three-level hierarchical relationship to autonomic element interactions. Individual autonomic elements form the lowest level of the hierarchy. 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.

Several works deal with interoperability from the viewpoint of homogenous competing managers. [5] implements a two-level autonomic data management system that optimizes the managed system so jobs are not starved of resources. A global manager is tasked with allocation of physical resources to a number of virtual servers in an optimal and equitable manner. Local managers oversee each virtual server, using fuzzy logic to infer the expected resource requirements of the applications that run on the virtual servers. [6] 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 [7]. 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. [8] 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 more costly to develop and test than two independent managers.

The majority of 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 specific to the system in question; preventing wide applicability and reusability. Custom solutions are expensive to develop and are sensitive to changes in target systems, and thus generally restrictive and not future-proof. A significant issue is that they do not tackle the problem of unintended or unexpected interactions that can occur when independently developed AMs co-exist in a system.

This challenge has been recognised for some time, for example [9] defines a number of interfaces to aid autonomic element interactions. Several 'vision' papers [10], [11], [12] identify interoperability as a key challenge for future autonomic systems. [10] argues that 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. [10] also identifies the need for a function to translate the output of one element to the format understood by another. [11] 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. [12] 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

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 spec. is available for each AM.

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 near-future cloud computing facility 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 (and thus always ongoing).

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

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.
- Sufficiently trustworthy that compliant AM's are *certifiable* for safe co-existence – regardless of platform, vendor etc.

IV. AN INTEROPERABILITY SERVICE

This section presents an initial IS for exploration of the requirements identified above. 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 or suspend 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.

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 outlined in figure 1. A number of interfaces are specified, and form three groups:

IS-AM interaction is supported by two interfaces.

IAdvertise {*Advertise*, *Unregister*, *Heartbeat*} is used by AMs to signal joining (register), 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. *Unregister* is used by an AM to signal an orderly shutdown, and *Heartbeat* (normally invoked periodically) enables (when absent) the IS to detect when a manager crashes or leaves abruptly. In either case, the AM's management interests are unregistered 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*, *Resume*, *Throttle*} is used to receive directives 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 (part or all of its management activity). *Resume* reactivates a suspended AM. The IS uses *Throttle* to specify different rates of activity to potentially conflicting AMs to prevent certain oscillatory patterns developing.

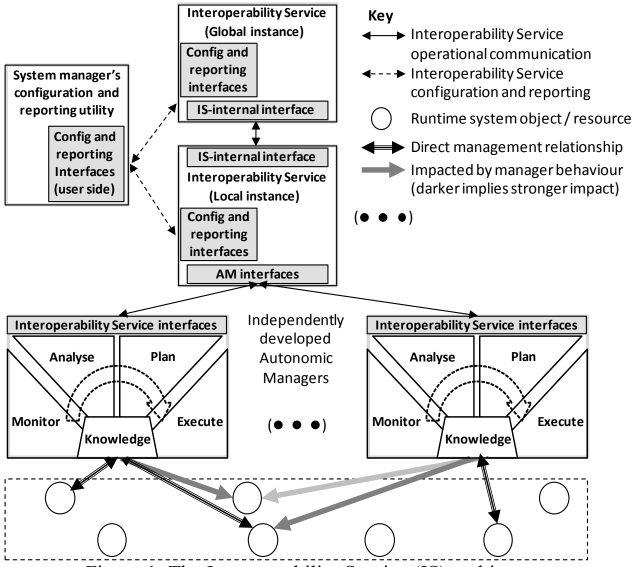

Interoperability Service (Global instance)
Config and reporting interfaces
IS-internal interface
System manager's configuration and reporting utility
Config and reporting Interfaces (user side)
IS-internal interface
Interoperability Service (Local instance)
Config and reporting interfaces
AM interfaces
Key
Interoperability Service operational communication
Interoperability Service configuration and reporting
Runtime system object / resource
Direct management relationship
Impacted by manager behaviour (darker implies stronger impact)
(● ● ●)
Interoperability Service interfaces
Analyse
Plan
Monitor
Execute
Knowledge
Independently developed Autonomic Managers
(● ● ●)

Figure 1. The Interoperability Service (IS) architecture.

IS-IS interaction is facilitated by a single interface.

ICommunicate {*Forward*, *Locate*, *Elect*, *SetISLevel*, *GetISLevel*} supports hierarchical operation. *Forward* is used to pass messages between local ISs which want to control global resources and the Global IS instance; 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 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* sets the IS level to be either Local or Coordinator. *GetISLevel* is used by each IS instance to determine its status during Locate and Elect events.

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. *SetMode* and *GetMode* allow configuration of the service to allow different levels of safety; 'Safe' requires that all of a particular AM's management activity is suspended when it is found to be involved in a conflict, whilst 'Permissive' allows partial suspension. *SetSensitivity* and *GetSensitivity* are used to configure the conflict detection sensitivity level. *StatusReport* collects status information and statistics for report generation and IS performance monitoring.

The IS architecture specification precisely 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 2 shows how the IS functionality is integrated with the various components of the system.

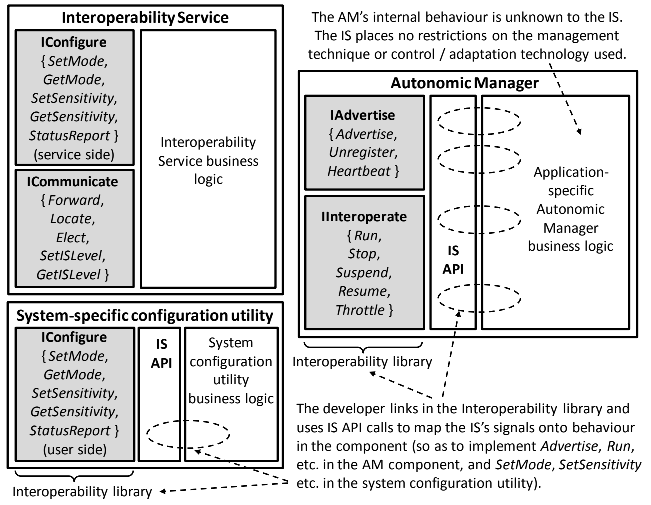

Interoperability Service
IConfigure { SetMode, GetMode, SetSensitivity, GetSensitivity, StatusReport } (service side)
ICommunicate { Forward, Locate, Elect, SetISLevel, GetISLevel }
Interoperability Service business logic
The AM's internal behaviour is unknown to the IS. The IS places no restrictions on the management technique or control / adaptation technology used.
Autonomic Manager
IAdvertise { Advertise, Unregister, Heartbeat }
IInteroperate { Run, Stop, Suspend, Resume, Throttle }
IS API
Application-specific Autonomic Manager business logic
Interoperability library
System-specific configuration utility
IConfigure { SetMode, GetMode, SetSensitivity, GetSensitivity, StatusReport } (user side)
IS API
System configuration utility business logic
Interoperability library
The developer links in the Interoperability library and uses IS API calls to map the IS's signals onto behaviour in the component (so as to implement Advertise, Run, etc. in the AM component, and SetMode, SetSensitivity etc. in the system configuration utility).

Figure 2. Internal architecture of the system components and the integration of the IS interfaces with these components.

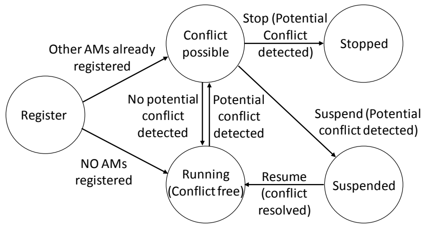

Register
Other AMs already registered
Conflict possible
Stop (Potential Conflict detected)
Stopped
No potential conflict detected
Potential conflict detected
Suspend (Potential conflict detected)
NO AMs registered
Running (Conflict free)
Resume (conflict resolved)
Suspended

Figure 3. State diagram held by IS, for each registered AM.

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. The AM developer must integrate the API calls into the manager such that the control behaviour meets the IS specification. Where an AM manages multiple resources the developer can choose to implement Suspend 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. Similarly, the developer can decide the AM-internal semantics of Suspend 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.

An instance of a state model is maintained for each registered AM (see figure 3). The information held in these models drives the IS' conflict management behaviour and is the basis on which AMs' management rights are governed. During AM registration, if no other AMs are 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 signal the AM that it can Resume, i.e., manage, once the conflict has been resolved).

V. 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 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; requiring that the IS must also:

- Detect AMs and learn their characteristics (via registration);
- Identify potential conflict, determine the consequences and the level of risk, and achieve a system-specific balance when taking decisions to resolve conflicts by suspending or stopping AMs' management activities;
- Automatically resume suspended AMs when conflicts are resolved (e.g., 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. It will be 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 interests and 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 (see figures 4 and 5 for an example):

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 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 ≈ 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. 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. Terms include:

{*Period*(value), *Event*(name) , ... }

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

Envelope. Optional. Expresses range of, and/or the number of dimensions of, control freedom. 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) }

VI. CONFLICT DETECTION

For the initial exploration we use a conflict detection technique based on pair-wise fuzzy similarity measures of AMs' management interests. This uses a summation of weighted terms, derived from AMs' management descriptions (see sections V and VII). Conflict detection activity is triggered by events such as the registration of a newly-discovered AM, or the departure of an AM from the system. The items that comprise the management description form a vector. Weights are allocated to the items to signify relative importance.

A dynamically configurable conflict threshold ($0 < \text{ThreshC} \leq 1$) is used to tune the conflict detection sensitivity (via SetSensitivity, on IConfigure). A potential conflict is detected if the similarity measure of a pair of vectors exceeds ThreshC. It is intended that the sensitivity level is configured by the facility manager, via a control console application (or automated), and can be changed at run time as necessary. This enables safety critical systems to operate with very low tolerance to potential conflicts, whereas in domains where only e.g., efficiency is at stake, a higher tolerance can lead to benefits of having more AMs working simultaneously (bearing in mind that a 'potential conflict' may not be realised).

VII. EVALUATION

We demonstrate the operation and benefit of the IS in a data centre scenario in which two independently developed AMs coexist. A scheduling manager (AM1) has a main goal of maximising throughput by keeping all resources utilised where possible. A power manager (AM2) is designed to minimise power usage by slowing down processor speed or by shutting down entire processor units where possible. 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 4. API calls to register AMs' management interests.

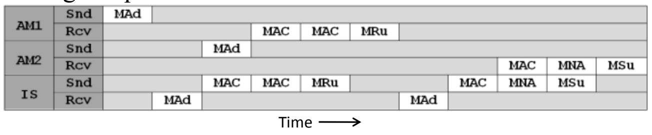
The API calls for manager registration are shown in Figure 4a (for AM1), and 4b (for AM2), where AddACItem means 'Add autonomically controlled item'; its template is shown in figure 4c. Figure 5 shows the XML equivalent representation for AM1.

```
<!-- 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 5. XML representation of the Management Description Language.

Scenario 1: Each manager registers separately in the system in the absence of the other. ThreshC = 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.

Scenario2: AM1 registers and is granted rights to manage the resources it requested. AM2 then registers whilst AM1 is still present. ThreshC=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). An indirect conflict is detected: AM1 already manages 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 weighted conflict level for system performance (found to be 0.6875) exceeds the current ThreshC (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 6 shows the resulting message sequence.



Key: Snd - Send message MNA - MNACK MAC - MACK MRu – Mrun
Rcv - Received message Mad – MAdvertise MSu - MSuspend

Figure 6. Message sequence for scenario 2.

Scenario 3: As scenario 2, but with ThreshC = 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 resulting message sequence is shown in Figure 7. 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.

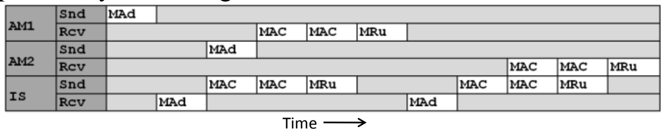


Figure 7. Message sequence for scenario 3.

VIII. CONCLUSION

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 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 future AM's that may exist in the target system, and thus supports agility i.e., configuration changes, expansion and upgrades.

IX. REFERENCES

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

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

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

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

[5] Xu J., Zhao M., Fortes J., Carpenter R. and Yousif M. On the use of fuzzy modeling in virtualized data center management. *4th Intl. Conf. on Autonomic Computing* (Jacksonville, FL, USA, June 2007). ICAC '07. IEEE, 25-34.

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

[7] 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. 1000 islands: Integrated capacity and workload management for the next generation data center. *5th Intl. Conf. on Autonomic Computing* (Chicago, IL, USA, 2008). ICAC '08. IEEE, 172–181.

[8] 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.

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

[10] Kennedy C. 2010. Decentralised metacognition in context-aware autonomic systems: some key challenges. In *Proc. 24th American Institute of Aeronautics and Astronautics* (AIAA) *Workshop on Metacognition for Robust Social Systems* (Atlanta, Georgia,) AAAI-10, AIAA. 34-41.

[11] Salehie M. and Tahvildari L. Autonomic computing: Emerging trends and open problems. *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.

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