Towards a Generic Autonomic Architecture for Legacy Resource Management

Radu Calinescu
Computing Laboratory, University of Oxford, Wolfson Building, Parks Road, Oxford OX1 3QD, UK

Abstract **Half a decade has passed since the objectives and benefits of autonomic computing were stated, yet even the latest system designs and deployments exhibit only limited and isolated elements of autonomic functionality. From an autonomic computing standpoint, all computing systems – old, new or under development – are legacy systems, and will continue to be so for some time to come. In this paper, we propose a generic architecture for developing fully-fledged autonomic systems out of legacy, non-autonomic components, and we investigate how existing technologies can be used to implement this architecture.**

I. Introduction

The vision of autonomic computing [1] set an unprecedented number of engineering and scientific challenges [2] directed at a single goal: the development of self-managing computing systems. Since the launch of the autonomic computing manifesto over five years ago, tremendous resources have been dedicated to solving these challenges, yet systems that "manage themselves according to an administrator's goals" [3] are far from ubiquitous.

The limited adoption of autonomic solutions is largely due to the fact that existing IT system components exhibit only restricted and isolated elements of autonomic functionality. Based on insights from the development of a commercial framework for the autonomic management of data centre resources [4] and on best practices presented in a separate paper [5], we propose a generic autonomic architecture for the development of autonomic systems out of non-autonomic components. This approach differs significantly from other autonomic frameworks that target the management of autonomic-enabled components, e.g., applications implemented to use the API of the autonomic framework explicitly [6].

The core component of our architecture is a *policy engine* that enforces a set of user-specified business policies. The policy engine can be configured to manage resources whose types are unknown at implementation time. This configuration is achieved by means of a model of the managed system, and allows the integration of the policy engine into systems comprising a heterogeneous mix of legacy resources. Again, this represents a major improvement over existing frameworks that are dedicated to the management of a specific type of resource [6, 7, 8]. Another novel feature of our generic autonomic architecture is the policy engine's ability to expose the collection of IT resources it manages as an atomic, higher-level resource. This enables the integration of an autonomic system as an individual resource into another instance of the same architecture, thus supporting the development of hierarchical systems-of-systems [9].

The remainder of the paper is organised as follows. Section II introduces our generic autonomic architecture and contrasts it with existing autonomic computing frameworks. Sections III through VII describe in detail the components of the architecture, and investigate the extent to which existing technologies can be used for their implementation. A preliminary specification of the managed system model used to configure the policy engine is proposed, and sample policies based on this model are presented in these sections. We then describe the requirements for the policy engine, and the criteria used to distinguish between its different realisations. Section VIII summarises our findings and the next steps of the project.

II. Autonomic Architecture for Legacy Resource Management

A large number of projects have investigated isolated aspects related to the development of autonomic computing systems out of non-autonomic components. Some of these projects addressed the standardisation of the policy information model, with the Policy Core Information Model (PCIM) [7, 8] representing the most prominent outcome of this work. Recent efforts such as Oasis' Web Services Distributed Management (WSDM) project were directed at the standardisation of the interfaces through which the manageability of a resource is made available to manageability consumers [9]. An integrated development environment for the implementation of WSDM-compliant interfaces is currently available from IBM [10].

In a different area, expression languages were proposed for the specification of policy conditions and actions, and used to implement a range of policies [11, 5, 12, 3]. In addition to the development of standards and technologies, complete autonomic computing solutions have been produced recently [4, 5, 6], typically for the management of specific systems, and with limited ability to function in different scenarios from those they were originally intended for.

The generic autonomic computing architecture depicted in Fig. 1 builds on all these recent developments, and generalises the author's previous work on policy-based resource management [5]. The policy engine at the core of the architecture is a generic module that can be configured to manage heterogeneous systems comprising components that vary from traditional computing resources such as servers and software applications to application servers, virtual machines and devices including load balancers, switches and PDAs.

The use of a generic policy engine across such a broad variety of domains requires that the engine configuration is done by means of a model of the system to be managed. This model has to specify the relevant system resources, alongside with their characteristics and relationships. A rich and expressive meta-model of managed systems is required to ensure that the manageability capabilities of all types of systems, whether small and simple or large and complex, can be specified as a model that the policy engine understands.

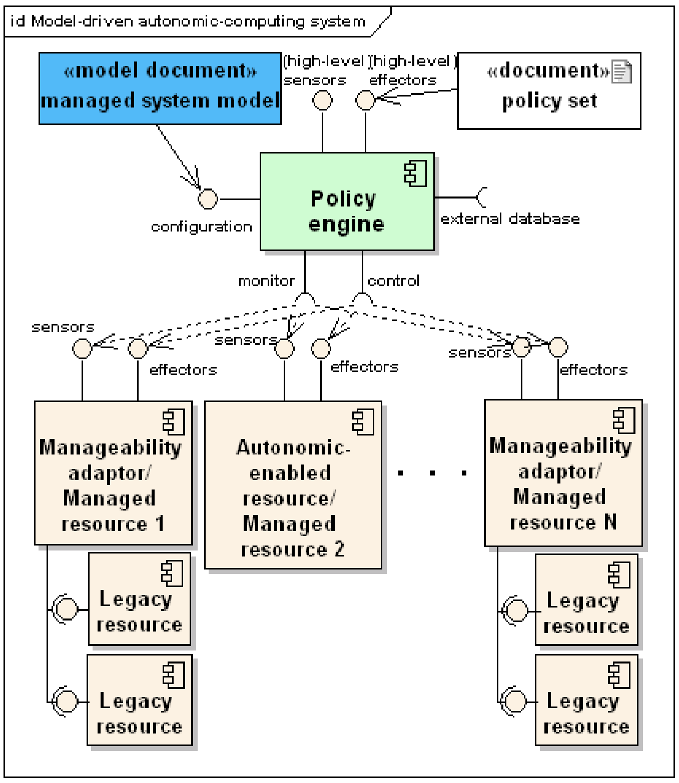


Fig. 1: The generic autonomic architecture for legacy resource management

A configured instance of the policy engine is capable of reading in and applying the set of policies on which to base its decisions in the management of the system. These policies are presented to the policy engine in a declarative language that makes references to the resources defined in the managed system model. They specify how the engine is required to monitor the resources, aggregate them into resource collections, report their state and act upon them to enforce higher-level business policies.

The managed system can include both legacy resources with no or limited autonomic capabilities, as well as autonomic-enabled resources. The legacy resources are exposed to the engine through a management-enabling layer with standard *sensors* (i.e., interfaces for gathering state information about resources) and *effectors* (i.e., resource control interfaces). The autonomic-enabled resources can be accessed directly by the engine. In particular, an implementation of the generic autonomic architecture can become a resource in a larger instance of the same architecture.

The next sections describe each part of the architecture in detail, specifying its required and desirable characteristics. Existing standards and technologies that can contribute to the realisation of the architecture are overviewed, together with their benefits and limitations.

III. Legacy Resources

The legacy resources that the generic autonomic architecture should support include a heterogeneous mix of "traditional" IT resources, recent types of IT resources, and devices. Typical traditional IT resources include:

- Physical servers and clusters of servers with their CPU, memory and disk resources, and the applications running on them. Starting/stopping, monitoring, reporting, allocating CPU, memory and disk service-level agreements to applications, and powering servers on and off in response to variations in load, failures, time of day/week and other business policies are among the activities covered by the autonomic management of a traditional IT system [5].
- Networks, collections of networks and the consumers that make use of them. Quality-of-service management, dynamic admission and provisioning control in the presence of variable demand, failures and changing business policies are the typical targets of a self-managed network [13].

Analogous but less traditional IT resources that will benefit from being part of a self-managed system include:

- Application servers with their web applications. Setting application service levels and access to the resources of the underlying hardware, monitoring, reporting and all functionality that is normally expected from a classical IT system will eventually be extended to these platforms.
- Virtualisation environments and the virtual machines they provide.

Some of the devices that will become increasingly present in autonomic systems are:

- Devices that are typical components of a standard IT system—printers, backup systems, switches, load balancers and power supplies. The latest models of all these devices exhibit interfaces that provide an ever increasing scope for automation.
- Common household devices—televisions, home cinemas, telephones, home security devices.

IV. Manageability Adaptors

Despite an increasing trend to add management interfaces to new computing components and devices, and to make existing ones public, achieving self-management in even small computing systems is hindered by the broad diversity of architectures and technologies these interfaces are based upon.

The generic autonomic architecture requires that a standard interface is used to expose the manageability of all types of resources presented in the previous section in a uniform way. The manageability interface comprises:

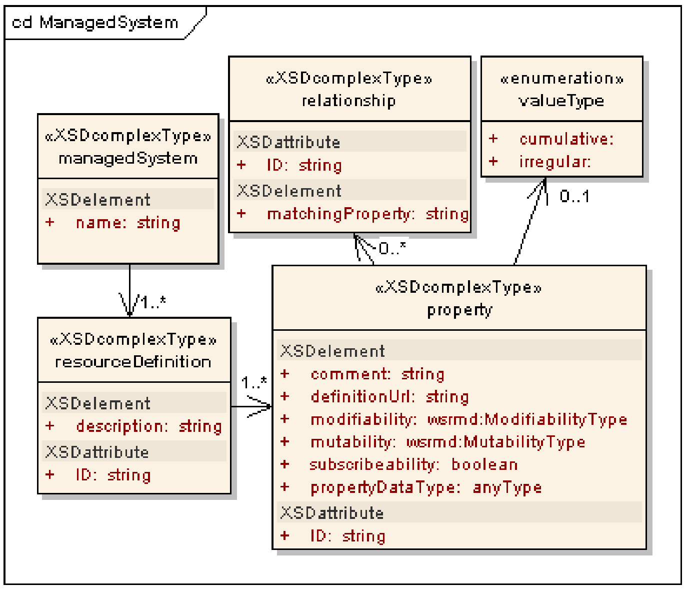
- Sensors for accessing the state of the managed resources. The sensors should support both explicit reading of specific state information, and a notification mechanism that the policy engine can use to subscribe and receive notifications of certain state changes.
- Effectors for configuring the resource parameters in line with the policies supplied to the policy engine.

The interface is solely responsible for the interoperability of a diverse spectrum of resources with the universal policy engine. To achieve this, the interface needs to be simple and flexible, and to associate only limited semantics to the state information and configuration parameters it exposes. Resource properties such as state variables and configuration parameters are uniquely labelled and strongly typed, but their roles and relationships are specified instead in the system model, as described in the next section.

A very good approach at defining a manageability interface standard that satisfies these requirements is represented by the Web Services Distributed Management (WSDM) standard. The Management Using Web Services (MUWS) component of WSDM [9] leverages web service technology benefits such as platform independence, loose coupling and security support to define a web service architecture enabling the management of generic distributed resources. The MUWS specification describes a standard way in which manageable resources can expose their capabilities, and defines a number of built-in capabilities that resources should provide (e.g., *ResourceId*, *Description* and *Version*). Resource-specific capabilities can be provided and listed as elements of the *Manageability-Characteristics* built-in capability. The WSDM/MUWS standard specifies ways for accessing resource capabilities by means of web services, and requires that a "resource properties document" XML schema is provided as a basic model of the managed resources. As a result, an implementation of the standard [10] provides a superset of the functionality required for a managed resource from our architecture.

V. Managed System Model

The system model used to configure the policy engine must specify all resources to be managed and all their relevant properties. As the policy engine can always be reconfigured using new versions of the model, resources and resource properties not referred to in the policies need not be specified. The model should also provide details about the characteristics of the resource properties, thus allowing the use of adequate operators in the policies and reducing the amount of work by the policy engine. Finally, to enable the reuse of model components and policies, standardised terminology and resource (property) definitions must be used in the model.



cd ManagedSystem
«XSDcomplexType» relationship
XSDattribute
+ ID: string
XSDelement
+ matchingProperty: string
«enumeration» valueType
+ cumulative:
+ irregular:
0..1
«XSDcomplexType» managedSystem
XSDelement
+ name: string
0..*
1..*
«XSDcomplexType» property
XSDelement
+ comment: string
+ definitionUrl: string
+ modifiability: wsrmd:ModifiabilityType
+ mutability: wsrmd:MutabilityType
+ subscribeability: boolean
+ propertyDataType: anyType
XSDattribute
+ ID: string
«XSDcomplexType» resourceDefinition
XSDelement
+ description: string
XSDattribute
+ ID: string
1..*

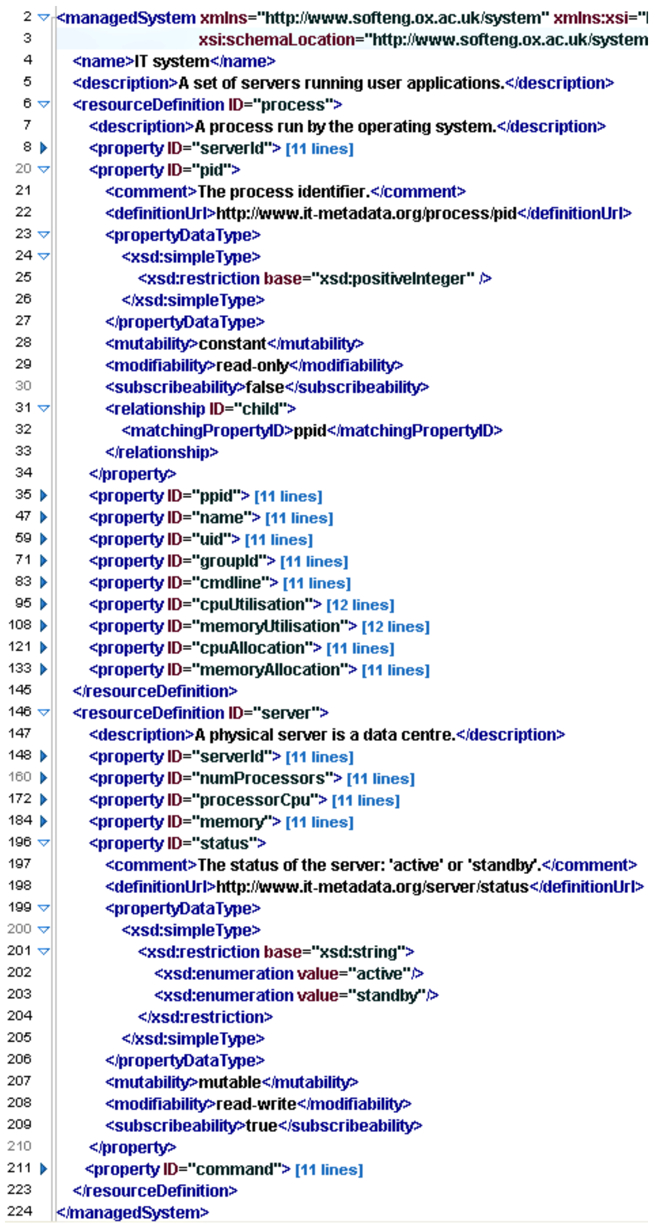
Fig. 2: Prototype meta-model of a managed system

The prototype *meta-model of a managed system* in Fig. 2 satisfies these requirements and is the preliminary result of a project to generalise the author's previous work on policy-based resource management [5]. The meta-model specifies a managed system as a named sequence of one or several resource definitions. Each resource definition (i.e., *resourceDefinition* in the UML diagram) comprises a unique identifier, a description and a set of resource properties with their characteristics. These properties should be drawn from a controlled metadata repository for the IT area of interest. Each property has a data type (*propertyDataType*), and is associated a unique ID and the URL within the metadata repository where its definition is located. The following property characteristics are exposed by the current version of the meta-model:

- *modifiability* – the *ModifiabilityType* from the *WSResourceMetadataDescriptor* (WS-RMD) 1.0 specification [14] is used to state if the value associated with this property is "read-only" or "read-write".
- *mutability* – the WS-RMD MutabilityType used specifies if the property is read-only or can be set. The possible values for this characteristic are "constant", "mutable" and "appendable".
- *subscribeability* – this element specifies if a client/agent such as the policy engine can subscribe to receive notifications when the value of this property changes.
- *valueType* – optionally, the model can specify for numerical properties if their value is cumulative (such the CPU utilization of a process over the process lifespan) or the property values follow no pattern.
- *relationship* – relationships between instances of a resource can optionally be specified as pairs comprising a unique ID and the ID of a "matching" property. Two resource instances are in the relationship if the current property of the first and the matching property of the second have the same value.

The sample model in Fig. 3 defines the processes and servers of an IT system. A policy engine configured to use this model can handle policies that refer to these two types of resources and their properties.

Microsoft's System Definition Model (SDM) is a meta-model used to create models of distributed systems [15] with a high degree of detail. The ongoing Dynamic Systems Initiative programme [16] intends to use these complex models as enabling elements in the development of manageable systems that exhibit elements of autonomic behaviour. Given its complexity, the SDM meta-model is less suited for use in conjunction with the generic policy engine employed by our generic architecture. The WSDM/MUWS standard [9] uses the WS-Resource Metadata Descriptor framework to describe the metadata for a resource manageability endpoint. This allows the specification of the properties of specific resource state variables and parameters, and the definition of resource relationships and operable collections. The Managed Resource Document used by version 1.1 of IBM's Policy Management for Autonomic Computing (PMAC) framework, and the combination of web services and autonomic computing standard specifications that version 1.2 of PMAC uses are further examples of managed system models [3].



```
<managedSystem xmlns="http://www.softeng.ox.ac.uk/system" xmlns:xsi="
                    xsi:schemaLocation="http://www.softeng.ox.ac.uk/system
  <name>IT system</name>
  <description>A set of servers running user applications.</description>
  <resourceDefinition ID="process">
    <description>A process run by the operating system.</description>
    <property ID="serverId"> [11 lines]
    <property ID="pid">
      <comment>The process identifier.</comment>
      <definitionUrl>http://www.it-metadata.org/process/pid</definitionUrl>
      <propertyDataType>
        <xsd:simpleType>
          <xsd:restriction base="xsd:positiveInteger" />
        </xsd:simpleType>
      </propertyDataType>
      <mutability>constant</mutability>
      <modifiability>read-only</modifiability>
      <subscribeability>false</subscribeability>
      <relationship ID="child">
        <matchingPropertyID>ppid</matchingPropertyID>
      </relationship>
    </property>
    <property ID="ppid"> [11 lines]
    <property ID="name"> [11 lines]
    <property ID="uid"> [11 lines]
    <property ID="groupId"> [11 lines]
    <property ID="cmdline"> [11 lines]
    <property ID="cpuUtilisation"> [12 lines]
    <property ID="memoryUtilisation"> [12 lines]
    <property ID="cpuAllocation"> [11 lines]
    <property ID="memoryAllocation"> [11 lines]
  </resourceDefinition>
  <resourceDefinition ID="server">
    <description>A physical server is a data centre.</description>
    <property ID="serverId"> [11 lines]
    <property ID="numProcessors"> [11 lines]
    <property ID="processorCpu"> [11 lines]
    <property ID="memory"> [11 lines]
    <property ID="status">
      <comment>The status of the server: 'active' or 'standby'.</comment>
      <definitionUrl>http://www.it-metadata.org/server/status</definitionUrl>
      <propertyDataType>
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:enumeration value="active"/>
            <xsd:enumeration value="standby"/>
          </xsd:restriction>
        </xsd:simpleType>
      </propertyDataType>
      <mutability>mutable</mutability>
      <modifiability>read-write</modifiability>
      <subscribeability>true</subscribeability>
    </property>
   <property ID="command"> [11 lines]
  </resourceDefinition>
</managedSystem>
```

Fig. 3: Basic model of an IT system

VI. Policy Set

A. Overview

Policies tell the policy engine how to manage the underlying system, and how to expose it to the outside world. Note that although the former role of policies is the only one considered by most autonomic computing frameworks, the latter role is equally important as it allows the architecture as a whole to become a managed component of a larger managed system. The policies employed by the autonomic architecture in Fig. 1 achieve these roles by specifying:

- How the modifiable properties of the resources (i.e., the resource configuration parameters) need to evolve as a function of the system state and of time.
- The exposed resources of the system, and their properties. As an example, consider the traditional IT system introduced in the previous section, whose resources are a set of servers and the processes running on them. A set of policies can specify that the policy engine exposes as high-level resources the applications running on the system, one property of this "application" resource being the number of servers on which the application is running.

Note that in a particular instance of the architecture, one or the other of these roles (but not both) can be missing.

The language used to express policies needs to be sufficiently flexible to support the use cases below.

B. Resource group specification

Policies are about resources of the managed system and their properties. Therefore, the policy language needs to allow the specification of the set of resources to which the policies apply. Specifying the scope of policies typically organises system resources into groups that are regarded as a single entity from the standpoint of a policy or set of policies. Resources grouped together for this purpose can be exposed as a higher-level resource by the policy engine. To illustrate this with an example, consider the IT system defined in Section V. The XML fragment below shows how the transitive closure of the child process relationship applied to all processes whose name is '`httpd`' can be used to group the processes of an Apache web server and all their descendents:

```
<resourceGroup ID="Apache">
  <includes resource="process">
    child*(name=="httpd")
  </includes>
<resourceGroup>
```

C. Higher-level resource definition

Policies can specify higher-level resources that the policy engine exposes to the outside world, e.g., to present system administrators with a summary of the state of the managed system or to enable its integration into a larger managed system. The example below instructs the policy engine to expose an 'application' as a higher-level resource:

```
<resourceDefinition ID="application">
  <description>
    A software application.
  </description>
  <property ID="name"> [...]
  <property ID="numServers"> [...]
</resourceDefinition>
<exportedResourcePolicy type="application">
  <policyScope>
    <resourceGroup ID="Apache"/>
  </policyScope>
  <policyCondition>TRUE</policyCondition>
  <policyAction>
    <property>
      <name>name</name>
      <value>Apache web server</value>
    </property>
    <property>
      <name>numServers</name>
      <value>COUNT(p:process|p.serverId)</value>
    </property>
  </policyAction>
</exportedResourcePolicy>
```

D. Resource configuration

Policies specify the desired value of modifiable resource properties as a function of the state of the managed system and of time. The following sample policy illustrates how between 8:00 and 18:00 the processes in a resource group are allocated 80% of the CPU power of their servers:

```
<resourceConfigurationPolicy>
  <policyScope>
    <resourceGroup ID="Apache"/>
  </policyScope>
  <policyValue>100</policyValue>
  <policyCondition>Hour IN 8..18</policyCondition>
  <policyAction applyTo="EACH(process.pid)">
    <property>
      <name>groupId</name>
      <value>1</value>
    </property>
  </policyAction>
  <policyAction applyTo="EACH(process.serverId)">
    <property>
      <name>cpuAllocation</name>
      <value>80%</value>
    </property>
  </policyAction>
</resourceConfigurationPolicy>
```

Other policies can be used to define how these resources should be managed outside this time interval, or policies with a higher policy value can enforce different actions between 8:00 and 18:00 on certain week days.

E. Resource scheduling

In resource scheduling, system capacity specified by resource properties are allocated to resource groups. Similar to other policies, this involves setting the value of specific resource properties. For instance, in our basic IT system scheduling policies could be used to specify how the server CPU and memory is to be partitioned among software applications. This may involve setting the "cpuAllocation" property of processes to allocate CPU to running groups of processes, and/or using the "command" and "state" properties of servers to start/stop applications and power on/off servers, respectively [5].

F. Workflow

Each configuration policy is a simple, one-step workflow. More complex workflows are often needed in which a sequence of actions is performed, with well-defined delays and state validations between successive actions in the sequence. Although this behaviour could potentially be simulated using a number of configuration policies and supporting additional resource properties, this approach would unnecessarily complicate the implementation of the manageability layer, the system model and the policies themselves. The use of BPEL workflows [17] represents a significantly more effective approach to expressing and handling workflow policies.

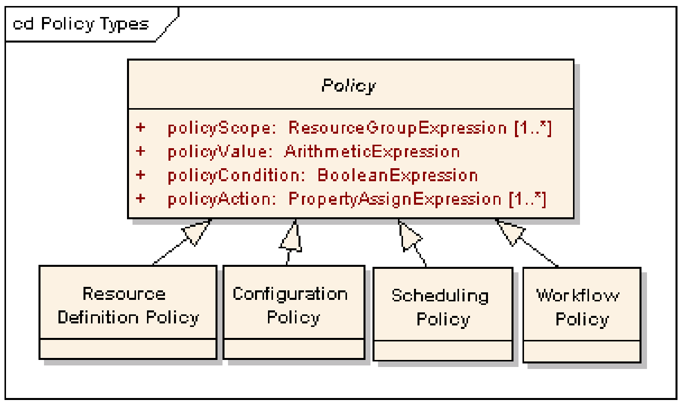

cd Policy Types
Policy
+ policyScope: ResourceGroupExpression [1..*]
+ policyValue: ArithmeticExpression
+ policyCondition: BooleanExpression
+ policyAction: PropertyAssignExpression [1..*]
Resource Definition Policy
Configuration Policy
Scheduling Policy
Workflow Policy

Fig. 4: Policies encountered in a generic autonomic architecture

G. Summary

Fig. 4 summarises the types of policies described in this section, illustrating how policy components are formulated in terms of expressions that depend on the resource properties of the managed system. These expressions vary in complexity from the very simple to the sophisticated, and the effectiveness of the policies supported by a realisation of the architecture is dependent on the power of its underlying expression language. Several autonomic computing expression languages have been proposed in the recent years. The language used by the policy-based resource allocation framework in [5] enables the specification of policies for resource monitoring and management in a data centre through the use of combinations of arithmetic and logic operators, pre-defined functions that can be applied to resource properties and built-in variables. While this works well for the system that the framework is targeting, the use of system-specific pre-defined variables such as PercCpuUtilServer (i.e., the percentage of CPU that an application is using on a given server) and AbsCpuHeadroomServer (i.e., the amount of CPU unused on a given server) is not generic enough for our system. However, the built-in variables used by the system-specific approach in [5] suggest the type of operators that would be needed in a realisation of the generic autonomic architecture.

The Windows System Resource Manager [4] uses regular expressions, logical and string operators, and built-in time variables to specify the process-matching criteria that define the WSRM policy scope, as well as the policy conditions and actions. The Autonomic Computing Expression Language (ACEL) [12] used by IBM's PMAC framework [3] supports a wide variety of primitive types (e.g., Boolean, several integer and float types, and String), and a selection of complex data types—Calendar, Composite and Collection. The standard operators are employed to combine resource properties and constants of these types into expressions. The extensive operator set in ACEL covers most of the use cases envisaged by the architecture described in this paper, although some very useful (albeit more complex) operators such as set comprehension and transitive closure are not supported.

VII. Policy Engine

The core component of the autonomic architecture implements a set of policies by monitoring and controlling the sensors and effectors of the managed resources, respectively. The "high level" resources of the managed system are exposed through the (high-level) sensors interface, enabling the inclusion of the system into another instance of the same architecture, a key requirement for the design of manageable systems of systems [1]. As indicated in Fig. 1, the engine is expected to make use of an external database for storing its internal state, e.g., the managed system model, the active policies and historical resource property values. To keep the architecture generic, we do not propose any particular way in which the policy engine should learn about the actual set of resources it is responsible for. Possible options include direct configuration, the use of a discovery technique [25] or a combination of the two.

Internally, the engine comprises modules for evaluating the expressions in the four policy components, an internal clock for time-based expressions, and an implementation-dependent set of schedulers, linear programming solvers and other optimisers, workflow engines, etc. An internal cache can optionally be used in addition to the external database for the rapid retrieval of state information. To keep the architecture generic, we are not going to propose a particular way in which the policy engine should be informed about the actual set of resources it is responsible for. Possible options include a static configuration by means of the policy set itself, the use of a discovery technique [19] or a combination of the two.

Given the generality of its specification, the engine can be implemented using a number of very different technologies, including standalone software applications/agents, a web services, or hardware appliances. As the field progresses and agreement is reached on a standard specification for the universal policy engine, its largely interchangeable implementations will differ in:

- The presence or absence of certain areas of functionality. The management of certain legacy resources may not require the use of scheduling and/or workflow policies. In this case, the use of a fast, off-the-shelf hardware appliance that does not support these parts of the specification could be ideal.
- The "quality" (i.e., the complexity and effectiveness) of the algorithms and heuristics involved. For instance, some implementations may use suboptimal, fast scheduling heuristics, while others may provide optimal decision making but a longer response time. Each of these implementations may be suitable for use in some systems but not in others.
- The total cost of ownership (TCO). Open-source and proprietary implementations of the engine will inevitably come with different TCO and TCO breakdowns. An open-source solution may involve no initial expenditure but significant effort to integrate and configure. Conversely, commercial implementations will require a major initial investment but offer the guarantee of a high-quality documentation and support over a long period of time.

VIII. Conclusions

Starting from a policy-based management framework targeted at data-centre resources [4, 5] and building on recent advances in autonomic computing [2, 3, 13, 16, 19, 23], we proposed a generic autonomic architecture and a universal policy engine for autonomic solution development. Our policy engine can be configured to monitor and control a wide variety of systems comprising heterogeneous mixes of legacy resources. The policy engine is configured by presenting it with a model of the system to be managed, i.e., a formal specification of the legacy resources in the system and of their relevant properties.

The components of the generic autonomic architecture were defined, and their requirements were discussed in the paper. Existing technologies that could be used to build these components were briefly analysed, and possible approaches to implementing the architecture were outlined.

Work is underway to validate the proposed system meta-model and the types of policies supported by the universal policy engine in data-centre resource management scenarios similar to those addressed by the commercial framework in [4]. The project is currently investigating the best way to use policies to define the high-level resources exposed by the policy engine so that an instance of the architecture can be integrated as a managed resource into a system of systems [18]. In the future, this work will continue in conjunction with the development of an IT metadata repository from which the models used to configure the policy engine will draw their resource property definitions. In the longer term, this should allow the definition of reusable policies and policy templates that will ease the adoption of the architecture.

References

[1] IBM Corporation: Autonomic computing: IBM's perspective on the state of information technology, October 2001.

[2] M. Parashar and S. Hariri. Autonomic computing: An overview. In: Unconventional Programming Paradigms, volume 3566 of LNCS, pages 257–269, 2005.

[3] J.O. Kephart and D.M. Chess. The vision of autonomic computing. IEEE Computer Journal, 36(1):41–50, January 2003.

[4] R. Calinescu and J.M. D. Hill. System providing methodology for policy-based resource allocation, July 2004. United States Patent Application no. 10/710322.

[5] R. Calinescu. Challenges and Best Practices in Policy-Based Autonomic Architectures. In: Proc. 3rd IEEE International Symposium on Dependable, Autonomic and Secure Computing, Columbia, MD, USA, September 2007, pages 65–74.

[6] IBM Corporation. Policy Management for Autonomic Computing, version 1.2, 2005. http://dl.alphaworks.ibm.com/technologies/pmac/PMAC12sdd.pdf.

[7] Microsoft Corporation. Windows System Resource Manager (WSRM) White Paper, August 2003.

[8] Sun Microsystems. Inc. Sun^{TM} Grid Compute Utility—Reference guide, June 2006. http://www.sun.com/service/sungrid/SunGridUG.pdf.

[9] Y. Bar-Yam et al. The characteristics and emerging behaviors of system-of-systems. Technical report, New England Complex Systems Institute, January 2004.

[10] J. Strassner, B. Moore, E. Ellesson and A. Westerinen, Policy Core Information Model—version 1 specification, February 2001. IETF RFC 3060, http://www.ietf.org/rfc/rfc3060.txt.

[11] B. Moore. Policy Core Information Model (PCIM) extensions, January 2003. IETF RFC 3460, http://www.ietf.org/rfc/rfc3460.txt.

[12] B. Murray et al. Web Services Distributed Management: MUWS primer, February 2006. OASIS WSDM Committee Draft, http://www.oasisopen.org/committees/download.php/17000/wsdm-1.0-muws-primer-cd-01.doc.

[13] IBM Corporation. Autonomic integrated development environment, April 2006. http://www.alphaworks.ibm.com/tech/aide.

[14] N. Damianou et al. The Ponder policy specification language. In: Policies for Distributed Systems and Networks, volume 1995 of LNCS, pages 18–38, Bristol, UK, 2001.

[15] D. Agrawal et al. Autonomic Computing Expression Language 1.2: User's Guide, 2005. http://www-128.ibm.com/developerworks/edu/acdw-ac-acel-i.html.

[16] A. Bandara et al. Policy refinement for diffserv quality of service management. IEEE eTransactions on Network and Service Management, 3(2):2–13, 2006.

[17] OASIS. Web Services Resource Metadata 1.0, November 2006.

[18] Microsoft Corporation. System definition model overview, April 2004. http://download.microsoft.com/download/b/3/8/b38239c7-2766-4632-b13-33cf08fad522/sdmwp.doc.

[19] Microsoft Corporation. Microsoft Dynamic Systems Initiative Overview, March 2005. http://download.microsoft.com/download/8/7/8/8783b65ed619-46d7-a8d-b4f13a97eeb0/DSIoverview.doc.

[20] M.B. Juric et al. Business Process Execution Language for Web Services. Packt Publishing, 2004.

[21] R. Harbird et al. Adaptive resource discovery for ubiquitous computing. In: Proc. 2nd workshop on middleware for pervasive and ad-hoc computing, volume 77 of ACM Intl. Conference Proceeding Series, pages 155–160, Toronto, Canada, October 2004.

[22] D. Hornby and K. Pepple. Consolidation in the Data Center. Sun Blueprints. Sun Microsystems Press, 2003.

[23] Murch, R.: Autonomic Computing. IBM Press, 2004.