Reconfigurable Service-Oriented Architecture for Autonomic Computing

Radu Calinescu
Computing Laboratory, University of Oxford
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
Radu.Calinescu@comlab.ox.ac.uk

Abstract

Almost 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. In previous work, we identified several of the key challenges behind this delay in the adoption of autonomic solutions, and proposed a generic framework for the development of autonomic computing systems that overcomes these challenges. In this article, we describe how existing technologies and standards can be used to realise our autonomic computing framework, and present its implementation as a service-oriented architecture. We show how this implementation employs a combination of automated code generation, model-based and object-oriented development techniques to ensure that the framework can be used to add autonomic capabilities to systems whose characteristics are unknown until runtime. We then use our framework to develop two autonomic solutions for the allocation of server capacity to services of different priorities and variable workloads, thus illustrating its application in the context of a typical data-centre resource management problem.

Keywords: autonomic computing, self-* system, service-oriented architecture, model-driven development, reconfigurable system

1. Introduction

The onset of a *digital economy* led to revolutionary transformations to the way in which Information and Communication Technologies (ICT) are used to conduct business and research and to provide services in all sectors of the society [2, 3]. The ability to accomplish more, faster and on a broader scale through expert use of ICT is at the core of today's scientific discoveries, newly emerged services and everyday life. Due to unprecedented advances in ICT, business needs are attended to by ever more complex and feature-rich systems and systems of systems [4].

Autonomic computing represents a powerful approach to managing the spiralling ICT complexity brought by these developments, by reducing the level of expertise required from the end users of ICT systems, and leveraging the rich capabilities of complex ICT components. Formally launched less than a decade ago [5], autonomic computing proposes that the demanding tasks of configuring, optimising, repairing and protecting complex ICT systems are delegated to the systems themselves [6]. Based on a set of high-level objectives (or *policies*), autonomic systems are intended to "manage themselves according to an administrator's goals" [7].

Following several years of intense research, we now have a good understanding of what autonomic systems should look like [6, 7, 8, 9, 10] and what best practices to follow in building them [11, 12, 13, 14]. This significant progress is to a great extent a by-product of the effort that went into the development of successful autonomic solutions addressing specific management tasks in real-world applications [15, 16, 17, 18, 19, 20].

While these developments demonstrate the feasibility and advantages of the autonomic computing approach to complexity management, autonomic functionality is far from ubiquitous in today's ICT systems. In previous work, we used insights from the development of a commercial autonomic system for the management of data-centre resources [15] to identify key challenges in the development of autonomic systems [11], including:

- The lack of standardisation in ICT resource interfaces. Despite an increasing trend to add management interfaces to new ICT components and devices, and to make existing interfaces public, autonomic system development is hindered by the broad diversity of architectures and technologies these interfaces are based upon.
- The tendency to hardcode ICT resource metadata within the control component of the autonomic system. Management frameworks are often intended for handling particular types of resources, and the param-

eters of these resource types are hardcoded in the control element of the system. With careful design, complex systems consisting of supported resources can be successfully managed; however, adding in support for additional types of resources cannot be achieved in a cost-effective way.

- The high scalability expectations. As simple, small ICT systems are easy to manage by low-skilled human operators, autonomic solutions are required in areas where the systems to manage are complex and comprise large numbers of resources.

Based on best practices devised while investigating these challenges [11], we then proposed a generic autonomic framework for the effective development of autonomic solutions in [21, 22] and briefly described its implementation as a service-oriented architecture (SOA) in [1]. This article represents an extended version of [1]. As such, the article provides additional information about our generic autonomic framework and the way in which it addresses the challenges mentioned above. Also, the article provides a significantly enhanced description of the framework implementation and of the combination of automated code generation, model-based and object-oriented development techniques employed by this implementation. Finally, we present a new autonomic solution for the typical server capacity allocation problem from [1], thus additionally illustrating how the framework can be used to support utility-function autonomic computing policies (i.e., policies that require adjusting the configurable parameters of a system so as to maximise the value of a user-specified *utility function* [23]).

The remainder of the article is organised as follows. Section 2 provides an overview of the generic autonomic computing framework, and examines how existing standards, technologies and tools can be used for its practical realisation. Section 3 presents the implementation of the autonomic architecture proposed by our framework as a service-oriented architecture, a solution chosen in order to take advantage of web service technology benefits such as platform independence, loose coupling and security support [24]. In Section 4, a case study involving the allocation of server capacity to services of different priorities and variable workloads is used to illustrate the application of the framework. Section 5 reviews related work in the areas of autonomic systems development out of legacy resources, model-driven development of autonomic systems and autonomic computing expression languages. Finally, Section 6 summarises our results and discusses a number of further work directions.

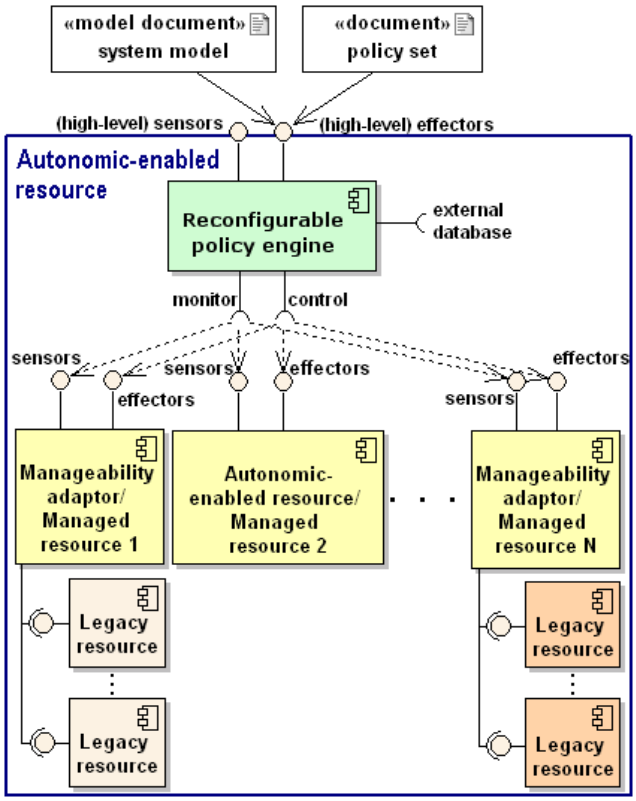

Figure 1. UML component diagram of the general-purpose autonomic architecture

2. Overview of the generic autonomic framework

Figure 1 depicts the general-purpose autonomic architecture used by our framework. Originally introduced in [21] and further developed in [22], this architecture builds on the recent developments mentioned in the introductory section, and extends the author's previous work on the policy-based management of data-centre resources [15].

The core component of the architecture is a reconfigurable policy engine that organises a heterogeneous collection of legacy ICT resources (i.e., resources not designed to support management by the policy engine) and autonomic-enabled resources into a self-managing system. In order to make autonomic solution development cost-effective, the policy engine can be configured to handle resources whose types are unknown during its implementation and deployment.

The rest of this section describes the components of the architecture and how they enable the runtime reconfiguration of the policy engine. Existing standards, technologies

and tools are suggested that can be employed to realise instances of these components.

2.1. Managed resources

The legacy ICT resources whose complexity can be managed through their integration into instances of the architecture include:

- physical and application servers, software applications;
- virtualisation environments and virtual machines;
- ICT devices such as switches and load balancers;
- factory automation equipment and robotic systems;
- household devices such as home safety and security devices.

The autonomic-enabled resources in the self-managing system are either typical ICT resources that were specifically designed to expose *sensors* and *effectors* interfaces allowing their direct inter-operation with the policy engine, or other instances of the architecture. As illustrated in Figure 1, the latter option is possible because the policy engine is exposing the entire system as an atomic ICT resource through *high-level sensors* and *high-level effectors*.

The high-level sensors expose to the outside world:

- The state of the policy engine itself, namely the current values of the engine parameters; for our implementation of the policy engine, these parameters are presented in Section 3.2.
- An overall view of the system state. Note that because the purpose of the high-level sensors is to facilitate the integration of the autonomic system as a component into a larger system, this view will typically—but not necessarily —represent a *summary* of the system state. Possible examples of such a summary include the average load of the servers within the system, the mean response time for a set of web applications or the failure rate of the system components. The precise nature of the system view presented by the high-level sensors is defined by special policies supplied to the policy engine, as described later in this section.

Likewise, the high-level effectors expose the configuration parameters of the engine (these parameters are described in Section 3.2), and any system-wide configuration parameters specified by the user-provided policy set.

2.2. Manageability adaptors

As recommended by IBM's architectural blueprint for autonomic computing [13], standardised adaptors are used to expose the *manageability* of all types of legacy ICT resources in a uniform way, through sensor and effector interfaces. These two types of interfaces enable the policy engine to access the state of the legacy resources and to configure their parameters, respectively—all without any modification to the managed resources. For efficiency reasons, the sensors should support both explicit reading of specific state information and, whenever possible, a state-change notification mechanism that the policy engine can subscribe to.

Note that the manageability adaptor interfaces for any instance of our architecture are fully defined by the system model used to configure the policy engine. This makes possible the use of model-based development techniques and tools for the semi-automatic generation of the manageability adaptors. By carefully selecting the technology used to "encode" the system model, off-the-shelf tools can be employed for this purpose—this is illustrated in Section 3, where XML system models are used for the configuration of the policy engine.

Another good approach to implementing the manageability adaptors is based on the OASIS Web Services Distributed Management (WSDM) standard. The Management Using Web Services (MUWS) component of WSDM [25] defines 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 *ManageabilityCharacteristics* built-in capability. The 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. An integrated development environment for the implementation of WSDM-compliant interfaces is currently available from IBM [26].

2.3. System model

Information about the system under the control of the policy engine—including details about its parameters—is provided by a *system model* that is supplied to the engine at run time. This model represents a specification of all resources to be managed and of their relevant *properties*. Note that the parameters of a system resource (e.g., the CPU capacity of a server, or the name of a process running on this server) are termed "properties" throughout most of the article in order to match the terminology proposed by the

WSDM standard [25].

As the 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. To allow the use of appropriate operators in autonomic computing policies and to reduce the amount of work by the policy engine, the model should provide details about each resource property it defines, including:

- the data type of the property;
- whether the property is read-only or can be modified by the policy engine;
- if the property has a constant value or it changes over time;
- if the policy engine can request to be notified about changes in the property value.

Because the policy engine needs to handle new system models at runtime, two further requirements must be satisfied by these models. The first requirement is that all system models are instances of the same meta-model, and the second that they are expressed in a format that the engine can use to generate automatically any components it needs to inter-operate with the manageability adaptors (e.g., classes for new resource types or manageability adaptor proxies).

Several standards and technologies are good candidates for the representation of the system model:

- Microsoft's System Definition Model (SDM) is a meta-model used to create models of distributed systems [27] with a high degree of detail. The ongoing Dynamic Systems Initiative programme [28] 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 reconfigurable policy engine employed by our autonomic architecture.
- The WSDM/MUWS standard [25] uses the WS-Resource Metadata Descriptor framework to describe the metadata for a resource manageability endpoint. This allows the specification of the properties of resource state variables and parameters, as well as the definition of resource relationships and *operable collections* (i.e., set of resources with aggregated state and operations).
- The Service Modeling Language (SML) specification put together by a consortium of leading ICT companies [29] can be used to model complex ICT resources based on a philosophy similar to that underlying the design of our autonomic architecture. When SML is adopted as a W3C standard and an SML development toolset becomes available, the use of SML models for the configuration of the policy engine will become a compelling option.
- The Managed Resource Document (MRD) 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 [30].

Finally, in order to make the development of these system models and of the autonomic solutions they underpin cost-effective, their elements need to be drawn from resource definition repositories built around domain-specific ICT ontologies [11]. This enables the reuse and sharing of manageability adaptors and policies across autonomic solutions from the same application domain, therefore leveraging the advantages of ontology-based modelling in the realm of autonomic computing, as emphasised in [31] and demonstrated successfully by [32].

2.4. Autonomic computing policies

Our policy model (Figure 2) extends the policy paradigm in [15, 30, 33, 34] based on best practices proposed in [11]. The abstract $Policy$ type at the root of the policy class hierarchy comprises three elements that are common to all policy types:

- The *policy scope* specifies the resources to which the policy applies, and takes the form of a set of "resource group" expressions. Each such expression is specified as a filter applied to a resource type supported by the policy engine. For example, given a cluster of servers, resource group expressions can be used to select all processes running on these servers and whose name matches a regular expression *regex* (i.e., `process.name` $=\sim$ *regex*) and/or all servers whose CPU utilisation exceeds 75% (i.e., `server.cpuUtilisation > 75%`).
- A *policy value* specified as an arithmetic expression is associated with each policy, and in the presence of conflicting/competing policies, higher-priority policies are realised at the expense of lower priority ones. In its simplest form, a policy value is an integer number.
- The *policy condition* is a Boolean expression used to specify the circumstances in which the policy engine is required to perform an action. This expression can use as parameters properties of the system resources in the policy scope, or built-in system variables such as `time`. For example, the policy condition '`time.hour>=9 &`

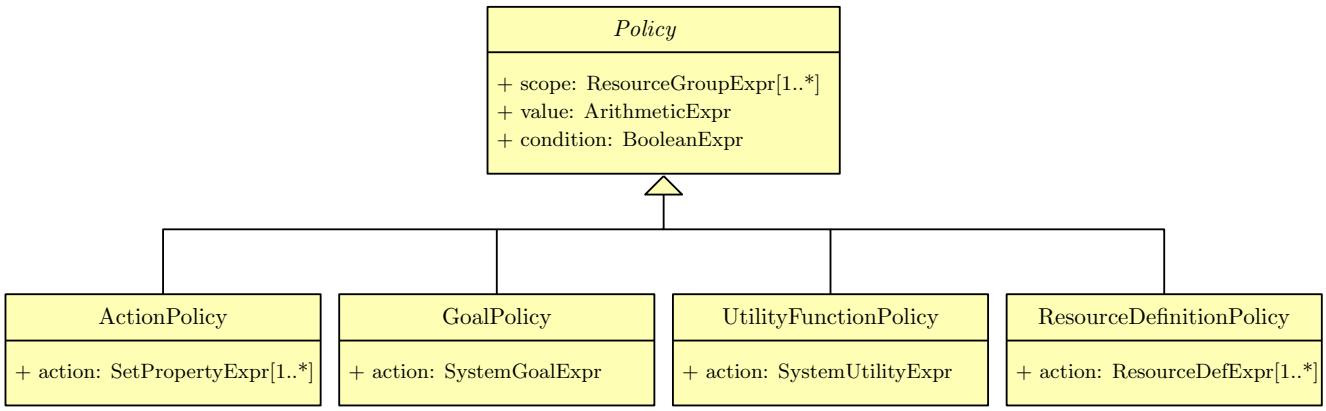

Figure 2. Policy model

`time.hour<=17`' specifies that the associated policy action should be performed between 9am and 5pm.

As illustrated in Figure 2, the abstract *Policy* class is specialised by four concrete classes of policies. These policy classes are associated with the *action*, *goal* and *utility function* policy types defined in [23, 35, 36], and with a *resource-definition* policy type that specifies how the policy engine should expose the managed resources through its high-level sensor and effector interfaces. The difference among these policy types is in the way in which they specify the fourth element of an autonomic computing policy, namely its *action*:

- The action element of an *action policy*[1] specifies new values for one or several properties of the resources within the scope of the policy. For this reason, such actions are encoded as sequences of assignment expressions of the form *`resource_property = expression`*.

- The action element of a *goal policy* is a Boolean expression that depends on the properties of the resources in the policy scope. Given a goal policy, the policy engine should adjust the modifiable properties of the resources in the policy scope in order to ensure that this Boolean expression evaluates to `true` at all times. For example, a goal policy may be specified that requests the policy engine to maintain the response time of all services in the policy scope below 1500ms—'`MAX(service, service.responseTime)<1500`'.[2]

- The action element of a *utility-function policy* specifies a "utility function" that associates a numerical value with each state of the resources in the policy scope (i.e., with the values of their properties). The policy engine is required to adjust the modifiable properties of these resources in order to bring them into a state that corresponds to the maximimum value of the utility function that is attainable. Utility-function policies are described in more detail in the context of the case study in Section 4.2.

- The action element of a *resource-definition policy* defines new types of resources that the policy engine is required to synthesise. The names and properties of these new resource types are fully specified by the action element of the resource-definition policy, and the policy engine is required to synthesise the software components for these resources dynamically. Presenting the semantics and implementation of resource-definition policies, and their role in the development of autonomic systems of systems is beyond the scope of this article—this information is available instead in a related publication by the author [58].

As described so far in this section, the four elements of a policy are specified in terms of expressions of appropriate type, and the ability to apply a rich set of operators and functions to the resource properties used in these expressions is key to supporting the types of policies in Figure 2. Accordingly, the policy language should include:

- an extensive set of operators for the manipulation of primitive types like the one provided by IBM's Autonomic Computing Expression Language [33];

[1] For historical reasons (in the early days of autonomic computing, action policies were the only type of autonomic computing policies), the term action is used to denote a component of autonomic computing policies, as well as a type of such policy. The meaning should be obvious from the context.

[2] Some of the techniques that the policy engine can employ to implement goal and utility policies are described in Section 4.2.

- regular expression and time operators similar to those implemented by Microsoft's Windows System Resource Manager [37];
- functions calculating average/minimum/maximum resource property values over a time interval and/or across a resource set like the built-in operators of the commercial policy engine in [15].

Additionally, a number of operators from areas such as formal specification [38] and formal quantitative analysis [39] are required to support or simplify the encoding of the four policy elements. These include *set comprehension* and *transitive closure* [38], for specifying the "resource group" expressions in the scope of policies; existential and universal quantification operators, to support the specification of policy conditions; and operators varying from ordinary assignments to choice, scheduling, linear programming and other optimisation operators for defining the actions of goal, utility-function and resource-definition policies.

2.5. Reconfigurable policy engine

The internal architecture of the reconfigurable policy engine (Figure 3) is dictated by the types of policies it implements and by its ability to handle ICT resources whose characteristics are supplied to the engine at runtime. A "coordinator" module is employing the components described below to implement the closed control loop of an autonomic system.

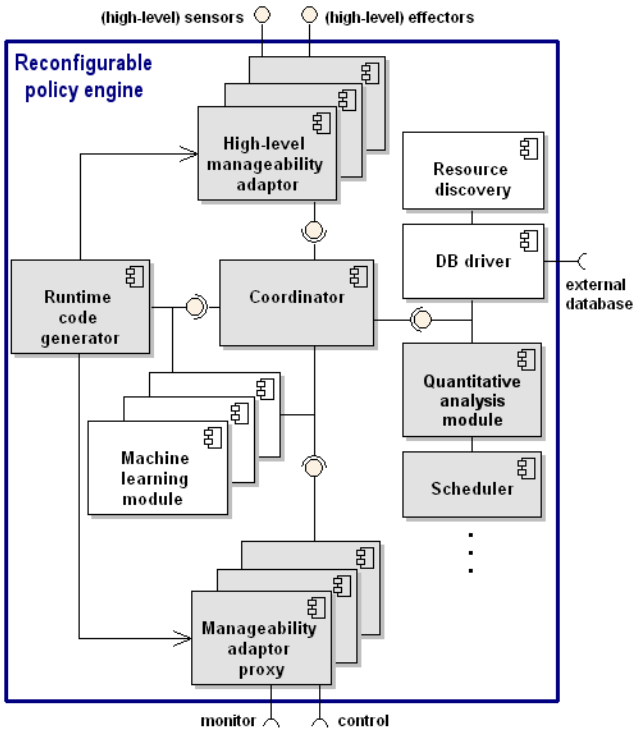

Figure 3. Architecture of the policy engine. The shaded components are implemented by the prototype described in Section 3.

Runtime code generator This component generates the necessary interfaces when the policy engine is configured to manage new types of resources or supplied with new "resource definition" policies. When a new system model is used to reconfigure the policy engine, *manageability adaptor proxies* are generated that allow the engine to interoperate with the manageability adaptors for the resource types specified in the system model. Likewise, when "resource definition" policies are set up that specify new ways in which the policy engine should expose the ICT resources it manages, *high-level manageability adaptors* need to be generated.

Manageability adaptor proxies These modules are thin interfaces allowing the policy engine to communicate with the autonomic-enabled resources and the manageability adaptors for the legacy resources in the system.

High-level manageability adaptors These elements are used to expose the system state and configuration in a format that allows its integration within another instance of the general-purpose autonomic architecture. The exposed system characteristics include the state and configuration of the policy engine itself (e.g., system model, policy set and monitoring period), as well as any characteristics of the managed resources that are specified by 'resource definition" policies implemented by the engine.

Scheduler This module is used to support the various operators appearing in policy actions for the goal and "utility function" policies handled by the policy engine, examples of which are provided in Section 4.

Resource discovery This component is used to locate the resources to be managed by the policy engine. The use of a technique such as the adaptive resource discovery described in [40] is recommended, although simpler approaches may be suitable for some use cases.

Database driver This module is used to maintain policy engine data such as historical resource property values in an external persistent storage.

Machine learning modules The ability to implement goal and "utility function" policies is key to the effective

management of complexity within an autonomic system. However, this requires the policy engine to possess in-depth knowledge about the behavioural characteristics of the managed system that should not (and often cannot) come from the system administrator. We are proposing that machine learning techniques [41] are employed by a set of policy engine modules to generate a behavioural (or *operational* [35]) model of the managed ICT resources based on sensor data and inside policy engine information. The usefulness of a *Modeler* component in autonomic systems that support utility functions is mentioned in [23], although the authors are not specific about the learning algorithms that such a component might use.

Quantitative analysis module This component enables the policy engine to take full advantage of a quantitative behavioural model that may be provided as part of the system model in Figure 1 or, in the future, built by its machine learning modules. The use of this module to support the implementation of a powerful class of utility-function policies represents the subject of a forthcoming paper [42].

3. Implementation

Two major choices influence the way in which an instance of the architecture in Figure 1 is realised: the technology used to represent the system model; and the technology chosen for the implementation of the policy engine components. This section describes how we made these choices for a prototype implementation of the architecture and gives prototype implementation details.

3.1. System model

For our prototype implementation, we chose to represent system models as plain XML documents that are instances of a pre-defined meta-model encoded as an XML schema. This choice that disregards some of the better suited modelling technologies discussed in Section 2.3 (e.g., [25, 27, 29]) was motivated by the availability of numerous "off-the-shelf" tools for the manipulation of XML documents and XML schemas that are largely lacking for the other technologies. In particular, by using existing XSLT engines and XML-based code generators we shortened the prototype development time and avoided the need to implement bespoke components for this functionality.

As illustrated by the UML class diagram in Figure 4, our meta-model specifies a managed system as a named set of resource definitions. Each resource definition (i.e., *resourceDefinition* in the UML diagram) comprises a unique identifier *ID*, a description and a set of resource properties with their characteristics. A resource property has a data type (i.e., *propertyDataType*), and is associated a unique

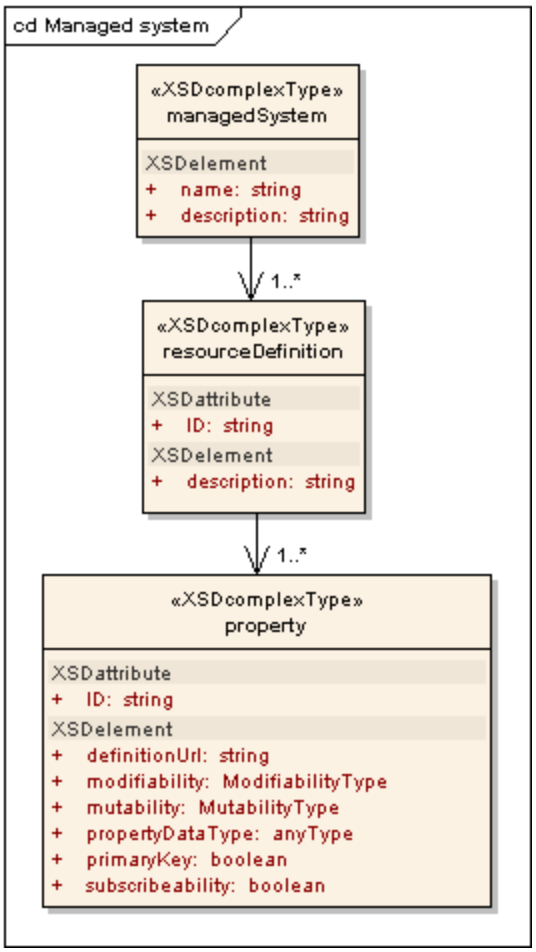

Figure 4. Meta-model of a managed system

ID and the metadata repository URL where its definition is available. Several other property characteristics are defined in the meta-model:

- *modifiability*—taken from the WS-ResourceMetadataDescriptor (WS-RMD) 1.0 specification [43], specifies if the property is "read-only" or "read-write";
- *mutability*—the WS-RMD MutabilityType [43] specifies if the property is "constant", "mutable" or "appendable";
- *primaryKey*—indicates whether the property is part of the property set used to identify a resource instance among all resource instances of the same type.
- *subscribeability*—specifies whether a client such as the policy engine can subscribe to receive notifications when the value of this property changes;

3.2. Policy engine

The generality of the autonomic architecture described in Section 2 allows the implementation of the reconfigurable policy engine using different technologies, e.g., as a software agent running on a data-centre server or a physical device incorporated into an industrial robotic system.

Our prototype policy engine and the manageability adaptors enabling its interoperation with legacy resources were implemented as web services in order to leverage the platform independence, loose coupling and security features of this technology. The runtime reconfiguration of the policy engine necessitated the extensive use of techniques available only in an object-oriented (OO) environment:

- Dynamic generation of data types (i.e., classes) was required to support new types of resources when the policy engine was reconfigured by means of a new system model.
- Runtime generation of web service proxies was required to enable the policy engine to interoperate with new, resource-specific manageability adaptors.
- *Reflection* (i.e., an object-oriented programming technique that allows the runtime discovery and creation of objects based on their metadata [44]) was heavily used to access the values of the resource properties, both to read their values once the policy engine obtained them from the manageability adaptors and to set new values for the modifiable properties.
- *Generic programming* (i.e., an OO programming technique enabling code to be written in terms of data types unknown until runtime [45]) was used to encode most of the functionality of manageability adaptors in a base abstract class, and to obtain resource-specific manageability adaptors by parameterising this abstract class with the dynamically generated resource data types.

Based on these requirements, J2EE and .NET were selected as candidate development environments for the prototype engine, with .NET being eventually preferred due to its better handling of dynamic proxy generation and slightly easier-to-use implementation of reflection.

In order to ensure that one instance of the policy engine can be configured to manage other policy engine instances as required by our framework, we started by modelling the policy engine as an instance of the system meta-model in Figure 4. The resulting model (depicted in Figure 5) defines the properties (i.e., the parameters) of the policy engine, namely:

1. The policy evaluation period, in seconds (i.e., 'period').

```
<?xml version="1.0" encoding="UTF-8"?>
<system ...>
 <name>Universal Policy Engine</name>

 <resource>
  <ID>policy engine</ID>

  <!-- WHEN to manage: period -->
  <property>
   <ID>period</ID>
   <propertyDataType>
    <xs:element name="period" type="pollingPeriod"/>
    <xs:simpleType name="pollingPeriod">
     <xs:restriction base="xs:unsignedInt"/>
    </xs:simpleType>
   </propertyDataType>
   <mutability>mutable</mutability>
   <modifiability>read-write</modifiability>
   <subscribeability>false</subscribeability>
   <primaryKey>false</primaryKey>
  </property>

  <!-- WHAT to manage: managed system model -->
  <property>
   <ID>system</ID>
   <propertyDataType> [53 lines]
   <mutability>mutable</mutability>
   <modifiability>read-write</modifiability>
   <subscribeability>false</subscribeability>
   <primaryKey>false</primaryKey>
  </property>

  <!-- HOW to manage: policy set -->
  <property>
   <ID>policySet</ID>
   <propertyDataType>
    <xs:element name="policySet" type="policySet"/>
    <xs:complexType name="policySet">
     <xs:sequence>
      <xs:element name="policy" type="autonomicComputingPolicy"
                  minOccurs="0" maxOccurs="unbounded"/>
     </xs:sequence>
    </xs:complexType>
    <xs:complexType name="autonomicComputingPolicy">
     <xs:sequence>
      <xs:element name="ID" type="xs:string"/>
      <xs:element name="scope" type="xs:string"/>
      <xs:element name="priority" type="xs:string"/>
      <xs:element name="condition" type="xs:string"/>
      <xs:element name="action" type="xs:string"/>
     </xs:sequence>
    </xs:complexType>
   </propertyDataType>
   <mutability>mutable</mutability>
   <modifiability>read-write</modifiability>
   <subscribeability>false</subscribeability>
   <primaryKey>false</primaryKey>
  </property>

  <!-- WHERE to manage: managed resource address(es) -->
  <property>
   <ID>resourceUrls</ID>
   <propertyDataType> [7 lines]
   <mutability>mutable</mutability>
   <modifiability>read-write</modifiability>
   <subscribeability>false</subscribeability>
   <primaryKey>false</primaryKey>
  </property>
 </resource>
</system>
```

Figure 5. Policy engine model

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema ...>
 <xs:element name="policyEngine" type="policyEngine" />
 <xs:complexType name="policyEngine">
  <xs:sequence>
   <xs:element name="period" type="pollingPeriod" nillable="true" />
   <xs:element name="system" type="system" nillable="true" />
   <xs:element name="policySet" type="policySet" nillable="true" />
   <xs:element name="resourceUrls" type="urlSet" nillable="true" />
  </xs:sequence>
 </xs:complexType>

 <xs:simpleType name="pollingPeriod">
  <xs:restriction base="xs:unsignedInt"/>
 </xs:simpleType>

 <xs:complexType name="system">
  <!-- the type of this element is the system meta-model -->
  ...
 </xs:complexType>

 <xs:complexType name="policySet">
  <xs:sequence>
   <xs:element name="policy" type="autonomicComputingPolicy"
               minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>

 <xs:complexType name="autonomicComputingPolicy">
  <xs:sequence>
   <xs:element name="ID" type="xs:string"/>
   <xs:element name="scope" type="xs:string"/>
   <xs:element name="priority" type="xs:string"/>
   <xs:element name="condition" type="xs:string"/>
   <xs:element name="action" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>

 <xs:complexType name="urlSet">
  <xs:sequence>
   <xs:element name="address" type="xs:string"
               minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>
</xs:schema>
```

Figure 6. XML schema for a policy engine "resource"

2. The model of the managed system ('system'). Note that the 'propertyDataType' of this policy engine property (not shown in Figure 5 for the sake of conciseness) is the system meta-model from Figure 4, in its XML schema representation.

3. The set of policies to implement ('policySet'). Each such policy is an instance of a complex data type whose elements are described later in this section.

4. The locations of the resources to be managed ('resourceUrls'), which for the current version of the prototype are set explicitly (the use of a discovery technique [40] is intended for future versions).

A simple XSLT [46] (model) transformation that we implemented was used to generate the "policy engine" XML

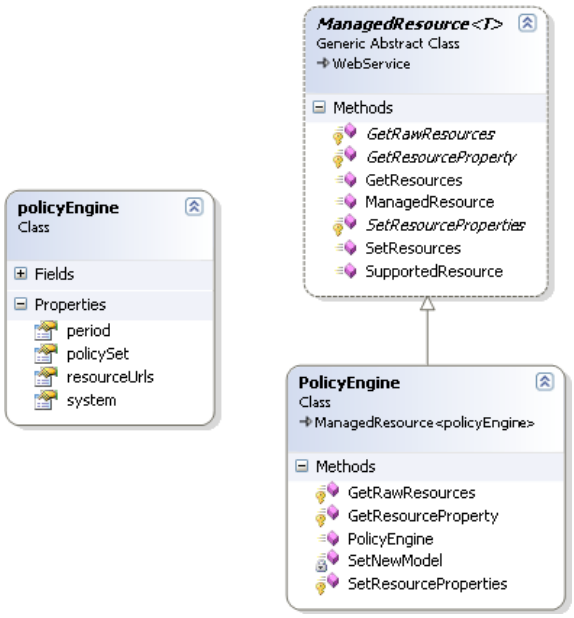

Figure 7. Policy engine resource (i.e., policyEngine) and manageability adaptor (i.e., PolicyEngine)—class diagram

schema in Figure 6 from the policy engine model, then a **policyEngine** C# class was generated automatically from this schema using the off-the-shelf XML Schema Definition (XSD) tool [47] (Figure 7).

Like for any other resource in our autonomic architecture from Figure 1, the parameters of the policy engine are accessed through a manageability adaptor. As shown in Figure 7, this adaptor (i.e., **PolicyEngine**) is a subclass of ***ManagedResource***$< T >$, the base class for all our manageability adaptors. The generic abstract class ***ManagedResource***$< T >$ comprises three web methods:

- SupportedResource returns the ID of the supported resource type.

- GetResources returns the list of all available resource instances. The method takes as argument a list of resource property IDs, and only the values of these properties are assessed and returned to the caller, thus preventing unnecessary resource property evaluation.

- SetResources takes as argument a list of resources of the supported type, and assigns any new values specified by the caller for the resource properties declared modifiable in the system model. The resources whose properties need to be modified are uniquely identified by the value of the resource properties marked as "primary key" components in the system model.

These web methods rely on resource-specific methods declared abstract in ***ManagedResource***$< T >$, and which any of its subclasses (including **PolicyEngine**) implements:

- *GetRawResources* builds a list of all available resource instances. The values of the resource properties need not be provided by this method.
- *GetResourceProperty* takes as arguments a resource instance and the ID of a resource property, and ensures that the property value is set in the resource object. The method is used by GetResources to fill in the required property values after obtaining a "raw" resource list from *GetRawResources*.
- *SetResourceProperties* takes a resource object and ensures that the modifiable properties of the corresponding real-world resource are assigned any new values specified in the resource object.

The web methods of our prototype, web service implementation of the policy engine correspond to the high-level sensors and effectors from the policy engine architecture in Figure 3. These methods can be used to read as well as to modify the engine parameters, ensuring that the parameters of the engine can be set by any type of software component that can be interfaced with a web service. For our case studies, we chose to implement a web-based administration tool that allows the remote configuration of the policy engine using a web browser (Figure 8), but this is by no means the only option available.

The four policy engine parameters that our administration tool reads and modifies using the web methods provided by the **PolicyEngine** manageability adaptor are described below. Note that these parameters are read by the tool whenever its front-end web page (shown in Figure 8) is loaded into a web browser, and modified when the administrator of the autonomic system uses the controls on this web page to explicitly operate a change in the engine parameters.

System model This parameter is an instance the XML system model described in Section 3.1. Changes to the 'system' property of the policy engine represent *reconfigurations of the engine for the management of new types of ICT resources*. This ability to specify the types of resources to be managed by the policy engine at runtime (and to change this specification as and when needed) represents a key feature of our autonomic computing architecture, and the reason why we term the policy engine a *reconfigurable* policy engine.[3]

[3]Clearly, other elements of an autonomic system implemented using our framework will undergo (re)configuration too, e.g., as a result of implementing the policies supplied to the policy engine. Such *self-configurations* are a defining characteristic of autonomic systems, and are discussed in detail elsewhere [5, 6, 7, 10, 12].

Internally, this operation involves:

- The automated generation of data types (i.e., C# classes) for the new types of ICT resources. The steps involved in the generation of these classes are those described above for the policy engine itself: first, the XSLT (model) transformation mentioned earlier in this section is applied to the newly supplied system model and an XML schema for the new resource types is obtained; then, the XSD tool [47] is employed to generate the necessary classes.
- The automated generation of proxies for the manageability adaptor web services associated with the new resource types. In the .NET framework, this amounts to generating a Web Service Description Language (WSDL) file and two "discovery" files for each type of manageability adaptor, and deploying these files into a subdirectory of the policy engine. Templates for each of these files are kept within the policy engine. This enables the engine to generate the manageability adaptor WSDL file for a new resource type by simply replacing a couple of placeholders in its template WSDL file with the identifier and the XML-encoded type for the new resource, respectively—both fields being available from the system model. As concerns the two "discovery" files, these are identical copies of the templates maintained within the policy engine.

Resource URLs This parameter is a space-separated list of URLs, each of which represents the address of a manageability adaptor for a set of resources to be managed by the policy engine. Changes to the resource URLs trigger the engine to contact the manageability adaptors at the specified addresses in order to establish the type of resources they expose. If these manageability adaptors exist and they support an ICT resource type defined in the system model used to configure the policy engine, then the policy engine will take into account all resources exposed through these manageability adaptors when implementing the user-supplied policies. Manageability adaptors associated with resource types unknown to the engine are ignored until such time as a system model defining these resource types has been provided to the policy engine.

Policy set This parameter is a space-separated list of policies that the system administrator can type directly into the web-based administration tool in Figure 8. Each of these policies consists of three of the policy components described in Section 2.4, i.e., scope, condition and action. The fourth policy component (i.e., policy value) is not supported by the current version of the policy engine.

Policy changes lead to a re-analysis of the policy set and to its parsing into an internal format that makes the period-

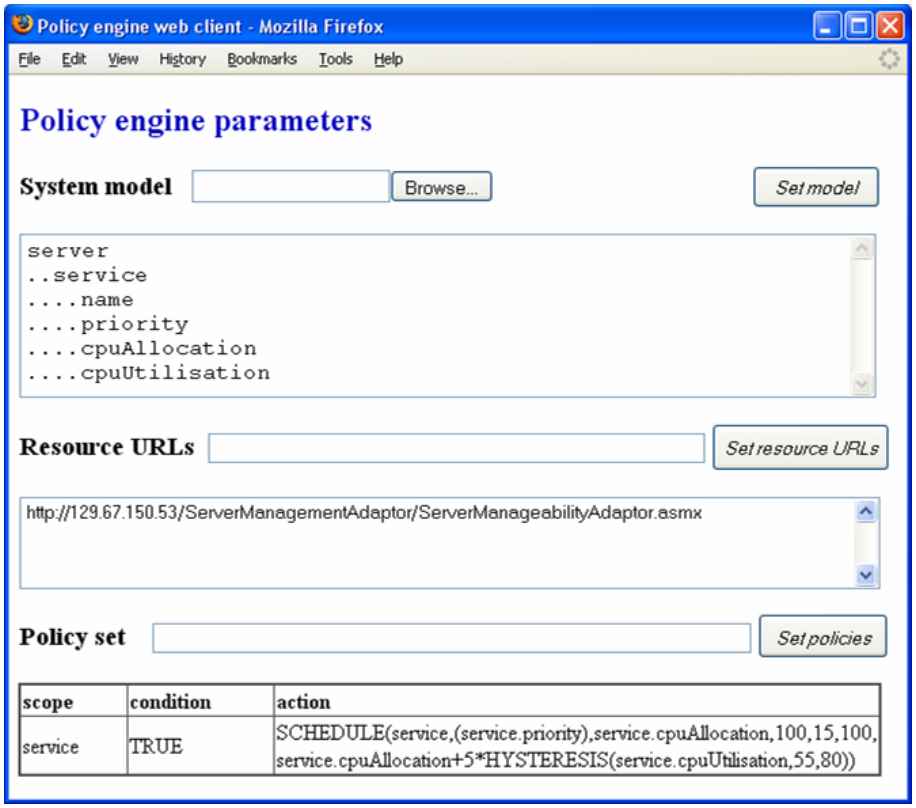

Figure 8. Snapshot of the web client used to configure the policy engine

ical evaluation of policies computationally efficient. Only policies referring to known types of resources and manipulating their properties in valid ways (e.g., a policy must not attempt to modify a "read-only" resource property) are accepted.

Period This parameter is an integer numerical value that represents the policy evaluation period, expressed in seconds.

The operators that the current version of the policy engine supports within the policy scope, value, condition and action expressions (cf. Figure 2) are the operators for the manipulation of primitive data types and only a few of the more sophisticated operators recommended in Section 2.4 (e.g., set comprehension and scheduling). Support for additional operators is added on a regular basis as new case studies are being explored.

Also, the current version of the policy engine comprises only a subset of the policy engine components presented in Section 2.5, namely the components that are shaded in Figure 3. These components were selected so as to speed up the completion of a prototype that could be used to assess the effectiveness of the framework, and to explore the feasibility of our approach in an area in which no research has been conducted so far, namely the runtime, model-based reconfiguration of autonomic computing policy engines.

4. Case study

This section presents two autonomic solutions for the allocation of server capacity to a set of services, one employing action policies and taken from [1] and the other one using utility-function policies. This choice of a case study was motivated by the importance that this real-world application has had since the release of server-level capacity control APIs such as [37, 48]. Additionally, our prior experience with data-centre resource management [15] helped significantly during the implementation of the two solutions, and in the interpretation of the case study results.

Note that effective autonomic solutions for case studies from other application domains were also developed using our generic autonomic framework and its SOA implementation presented in this article, including dynamic power management, adaptive control of cluster availability within data-centres, and dynamic generation of web content. All of these case studies are described in detail in [49].

```
<system>
  <name>server</name>
  <resource>
    <ID>service</ID>
    <property>
      <ID>name</ID>
      <propertyDataType>
         <xs:simpleType name="serviceName">
           <xs:restriction base="xs:string"/>
         </xs:simpleType>
      </propertyDataType>
      <mutability>constant</mutability>
      <modifiability>read-only</modifiability>
      <subscribeability>false</subscribeability>
      <primaryKey>true</primaryKey>
    </property>
    <property>
      <ID>priority</ID>
      ...
    </property>
    <property>
      <ID>cpuAllocation</ID>
      <propertyDataType>
        <xs:simpleType name="serviceCpuAllocation">
          <xs:restriction base="xs:int">
            <xs:minExclusive value="0"/>
            <xs:maxInclusive value="100"/>
          </xs:restriction>
        </xs:simpleType>
      </propertyDataType>
      <mutability>mutable</mutability>
      <modifiability>read-write</modifiability>
      <subscribeability>false</subscribeability>
      <primaryKey>false</primaryKey>
    </property>
    <property>
      <ID>cpuUtilisation</ID>
      ...
    </property>
  </resource>
</system>
```

Figure 9. XML model of the managed system

4.1. Server capacity allocation using action policies

In order to test the SOA implementation of our autonomic computing framework, we configured a running instance of the policy engine from Section 3 to allocate the CPU capacity of a server to a set of services of different priority, and subjected to variable workloads. The only resource defined in the server model (Figure 9) was `service` with four properties: a unique `name`, an integer `priority`, the percentage of the server CPU allocated to the service (`cpuAllocation`) and the amount of CPU utilised by the service, expressed as a percentage of its CPU allocation (`cpuUtilisation`).

The policy depicted in Figure 8 allocates a percentage of the CPU capacity of the server to each 'service' resource, as selected by the policy scope. The 'TRUE' policy condition requires that the policy action is applied at all times (i.e., in line with the policy evaluation period of the engine). The policy action is specified by means of an expression that uses the SCHEDULE(R, $ordering$, $property$, $capacity$, min, max, $optimal$) operator that

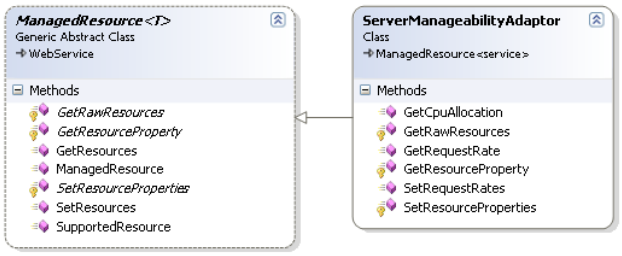

Figure 10. The server manageability adaptor

- sorts the resources in R in non-increasing order of the comparable expressions in $ordering$;
- in the sorted order, sets the specified resource $property$ to a value never smaller than min or larger than max, and as close to $optimal$ as possible;
- ensures that the overall sum of all $property$ values does not exceed the available $capacity$.

Accordingly, the policy action

$$\text{SCHEDULE}(service, \langle service.priority \rangle, service.cpuAllocation, 100, 15, 100, service.cpuAllocation + 5 * \text{HYSTERESIS}(service.cpuUtilisation, 55, 80))$$

in Figure 8 will set the $cpuAllocation$ property of all services to a value between 15% and 100%, subject to the overall CPU allocation staying within the 100% available capacity. Optimally, $cpuAllocation$ should be left unchanged if $55 \leq cpuUtilisation \leq 85$;[4] decreased by 5(%) if $cpuUtilisation < 55$;[5] and increased by 5(%) if $cpuUtilisation > 85$.[6] Note that this adjustment is performed repetitively, with a period given by the policy evaluation period parameter of the policy engine.

Like the policy engine itself, the manageability adaptor used to interface the engine with the server was implemented as a sub-class of ***ManagedResource***$< T >$— Figure 10.

The policy engine was then configured to manage remotely a server simulator running a high-priority 'premier' service and a lower-priority 'standard' service. The two services handled simulated user requests with exponentially-distributed inter-arrival time and normally-distributed processing time. Figure 11 shows the change in the system

[4]The HYSTERESIS($val, lower, upper$) operator used to achieve this behaviour returns -1, 0 or 1 if $val < lower$, $lower \leq val \leq upper$ or $upper < val$, respectively.

[5]The current CPU allocation is underutilised in this case, so it is decreased to avoid waste of CPU capacity.

[6]In this case, the service is utilising almost all CPU allocated to it, running the risk of becoming under-provisioned.

parameters when the request inter-arrival time of the two services was varied to simulate different workloads, and the policy engine was configured to implement the policy described earlier in this section; the system behaviour over the time intervals a to h is described below:

a. Both services are lightly loaded (5000μs request inter-arrival time) and have the minimum amount of CPU allocated (i.e., 15% each).

b. The load increases for the standard service, and its allocated CPU is increased by the policy engine accordingly.

c. For a brief period of time, the standard service uses its allocated CPU completely; no requests timeout though as its CPU allocation is increased swiftly.

d. The premium service workload starts to increase, and the policy engine increases its CPU allocation. Accordingly, the standard service starts to get less CPU.

e. As the workload for the premium service peaks and the policy engine schedules additional CPU capacity for this service, the standard service is allocated insufficient CPU and some of its client requests time out.[7]

f. The inter-arrival time for the premium service increases, and some of the CPU capacity allocated to it during the previous time interval is re-deployed by the policy engine to the standard service. No more requests time out.

g. Under constant workload, the CPU allocation is mostly stable.

h. To explore the role of the hysteresis, we replaced the hysteresis term in the policy action with HYSTERESIS$(service.cpuUtilisation, 80, 80)$, thus eliminating the hysteresis. This led to significant oscillations in the CPU capacity allocated to the services. The reinstatement of the original policy after this time interval brings the system back into a stable state.

The policy evaluation period was set to 3 seconds for this experiment, so that the system could self-adapt to the rapid variation in the workload of the two services. This allowed us to measure the CPU overhead of the policy engine, which was under 1% with the engine service running on a 1.8 GHz Windows XP machine. In a real scenario, such variations in the request inter-arrival time are likely to happen over longer intervals of time, and the system would successfully self-configure with far less frequent policy evaluations.

Note also that since the policy engine service is implemented as a managed resource, its policy evaluation period can be adjusted by another policy engine instance, so that it stays in step with the rate of change in the request inter-arrival time—a scenario that we are in the process of experimenting with.

[7] Requests time out after spending T=5s in a service request queue.

4.2. Server capacity allocation using utility-function policies

We showed in the previous section that our framework can be used successfully to develop a realistic autonomic solution. However, the cost-effectiveness of this solution is limited by its usage of action policies designed by a system administrator with in-depth knowledge about the system resources. In this section, we describe how the same self-management capability can be realised by means of utility-function policies that can be designed by someone aware of the high-level business goals of the system but who has limited knowledge about its internal operation.

To implement utility-function policies, the policy engine needs an understanding of the *behaviour* of the system and its resources. Given a resource, we define its state $\mathbf{s}$ as the vector whose elements are the read-only properties of the resource, and its configuration $\mathbf{c}$ as the vector comprising its modifiable (i.e., read-write and write-only) properties. Let S and C be the value domains for $\mathbf{s}$ and $\mathbf{c}$, respectively.[8] A behavioural model of the resource is a function

$$behaviouralModel : S \times C \rightarrow S, \qquad (1)$$

such that for any current resource state $\mathbf{s} \in S$ and for any resource configuration $\mathbf{c} \in C$, $behaviouralModel(\mathbf{s}, \mathbf{c})$ represents the future state of the resource if its configuration is set to $\mathbf{c}$.

In practice, the policy engine works with an approximation of the behavioural model that consists of a set of discrete values of the $behaviouralModel$ in (1)—an approach that works well with the continuous behavioural models that are typical to most real-world systems. As a further simplification, any state and configuration components that play no role in the resource behaviour (e.g., the `name` and `priority` properties of the `service` resource in our system) are disregarded in the behavioural model approximation that the policy engine operates with.

There are multiple ways in which the policy engine can acquire the behavioural model required to support utility-function policies. The two extreme ones are to have this model supplied by the resource itself and to have the model generated automatically by the machine learning modules within the policy engine (see Figure 3). An intermediate option is to have an initial behavioural model supplied to the policy engine, and further refined by its machine learning modules. Our prototype policy engine does not include the machine learning modules, hence the required

[8] Note that S and C are fully specified in the system model.

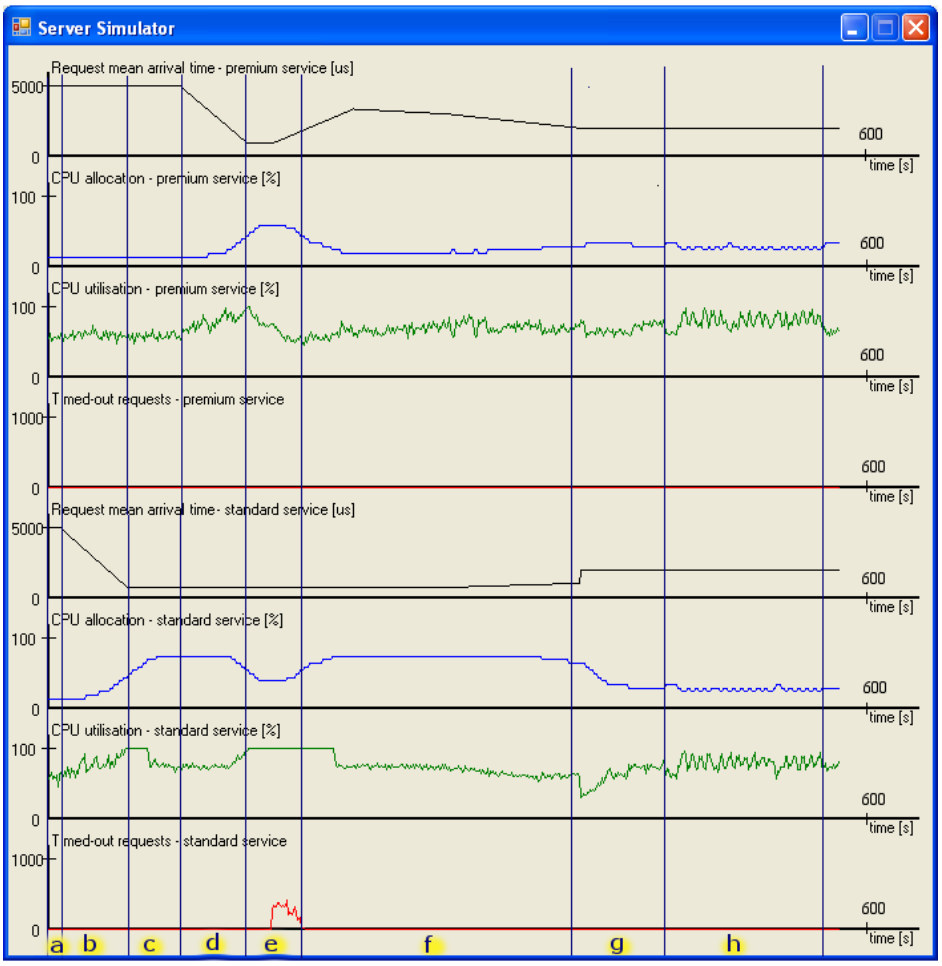

Figure 11. Snapshot of a typical server simulation experiment

behavioural model is provided by the manageability adaptor for the `service` resource. This behavioural model (Figure 12) describes how the response time of a service varies with the request inter-arrival time and the percentage of server CPU allocated to the service, and was obtained from multiple runs of the server simulator in which the average service response time was recorded for 920 equidistant points covering the entire (`interArrivalTime`, `cpuAllocation`) value domain.

To use utility-function policies in our autonomic solution, we added several new `service` properties to the system model devised in the previous section (Figure 13):

- `responseTime`, the service response time, measured in milliseconds and averaged over the past one-second time interval;
- `interArrivalTime`, the mean request inter-arrival time;
- `behaviouralModel`, an approximation of the service behavioural model.

We then defined a utility function that models the business gain associated with running $n > 0$ services with different levels of service:

$$utility(R) = \sum_{r \in R} r.priority * \min(1000, \max(0, 2000 - r.responseTime)), \quad (2)$$

where R is the set of `service` resources. Figure 14 depicts the utility function for a server running a "premium" service with priority 100 and a "standard" service with priority 10.

The policy implemented by the autonomic system was defined by means of the MAXIMIZE($R, utility, property, capacity, min, max, model$) operator that uses the information about the system behaviour encoded in $model$ to set the value of the specified resource $property$ for all resources in R such as to:

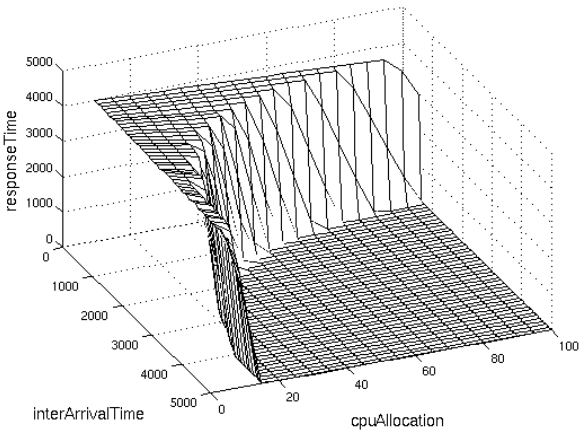

Figure 12. Service behavioural model

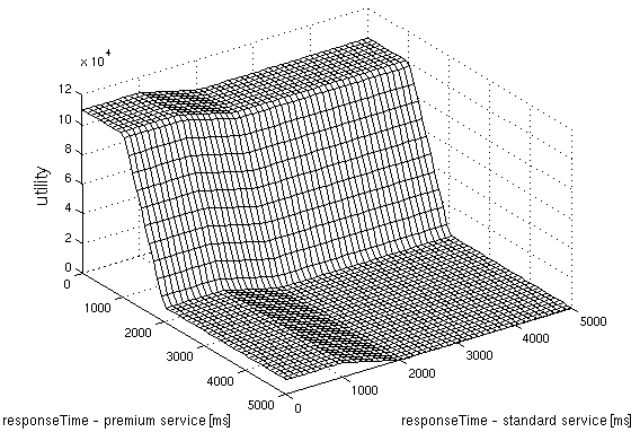

Figure 14. Utility function

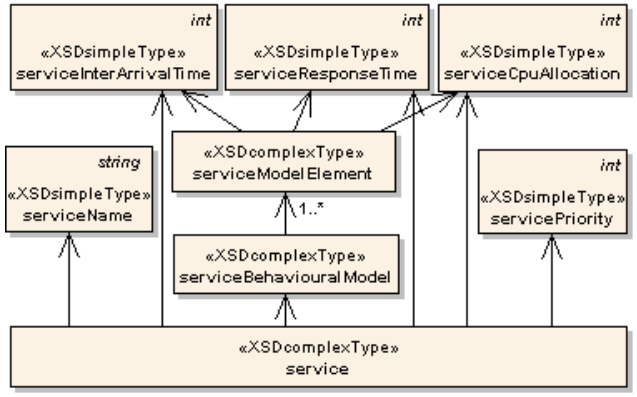

Figure 13. Service model for Section 4.2

- maximize the value of the $utility$ function;
- ensure that the value of $property$ stays between min and max, and that the sum of the $property$ values across all resources in R does not exceed the available $capacity$.

The arguments of MAXIMIZE were specified as shown in Table 1, in order to supply the policy engine with the definition of the utility function, and to link the `responseTime`, `interArrivalTime` and `cpuAllocation` properties of a `service` resource to the components of its `behaviouralModel` property. Each time it evaluates the utility-function policy, the policy engine uses this information to select the elements from the behavioural model that are in the proximity of the current state of the system; the Euclidean metric is used for this calculation. The new configuration for the system is then chosen as the one associated with the selected element that maximizes the value of the utility function.

Note that the policy engine could be required to synthesise the behavioural model itself by specifying the $model$ argument of MAXIMIZE as "$service.responseTime(service.interArrivalTime, service.cpuAllocation)$", so as to indicate only that the service response time depends on the request inter-arrival time and the CPU allocation for the service. This syntax will be used when machine learning support is added to the policy engine prototype.

Figure 15 illustrates a typical experiment in which the utility-function policy described in this section was used to manage the allocation of CPU to the same two services as in Section 4.1. The experimental results resemble those obtained when an action policy was used (Figure 11), therefore confirming the effectiveness of our approach to developing autonomic solutions that use utility-function policies in conjunction with a behavioural model of the managed resources. The few differences between the two sets of experimental results indicate that the autonomic solution that uses utility-function policies is actually superior to the solution based on action policies, as shown by these differences across the time intervals a to e in Figure 15:

a. Shortly after the utility-function policy is supplied to the policy engine, the CPU allocation is decreased to the minimum level that can ensure the optimal level of service. When the action policy was used, CPU variations of such magnitude required multiple policy evaluations.

b. The CPU allocated to the standard service increases in line with its workload.

c. The CPU allocation for the premium service also increases, but the response time of both services can still be maintained at values that maximize the utility function.

Table 1. Arguments of the MAXIMIZE operator for Section 4.2.

argument	value
R	$service$
$utility$	SUM$(service.priority * \text{MIN}(1000, \text{MAX}(0, 2000 - service.responseTime)))$
$property$	$service.cpuAllocation$
$capacity$	100
min	15
max	100
$model$	$service.responseTime(service.interArrivalTime, service.cpuAllocation) =$ $service.behaviouralModel.responseTime($ $service.behaviouralModel.interArrivalTime, service.behaviouralModel.cpuAllocation)$

d. The amount of CPU required to satisfy the increased demand for the premium service leaves insufficient CPU capacity for the standard service to make any contribution to the utility function, hence it is allocated the minimum amount of CPU (15%). When the action policy was used, all CPU capacity not given to the premium service was allocated to the standard service even if the standard service was of no use to the business. In contrast, the utility-function policy allocates additional CPU to the standard service only when enough capacity is available to bring this service into a region of operation in which it can contribute to the utility function.

e. The response time for the standard service is recovering slowly, as it takes time to drain the request queue built during the previous time interval. The use of an enhanced behavioural model that takes into account the length of the service request queue should speed up this recovery.

f. The CPU allocations for the two services are constant over long periods of time. With action policies, this could be achieved only by explicitly including a hysteresis construct in the policy specification.

Note that in order to outperform solutions based on action policies (as demonstrated by our case study), utility-function policies need to employ "adequately specified" utility functions. From our experience with developing policy-based autonomic solutions for data-centre resource management, devising effective utility functions for medium-sized applications requires in-depth knowledge of the application domain and careful validation before deployment within a production system, but is a task that can be completed successfully by an experienced system administrator. When optimal utility functions are sought, multiple (and possibly conflicting) system objectives need to be captured by these functions and/or large-scale, complex systems are involved in the intended autonomic applications, devising the utility functions is much more difficult. The development of techniques for the construction of such utility functions represents an active research area in autonomic computing.

5. Related work

The autonomic infrastructure proposed in [50] is retrofitting autonomic functionality onto legacy systems by using *sensors* to collect resource data, *gauges* to interpret these data and *controllers* to decide the "adaptations" to be enforced on the managed systems through *effectors*. This infrastructure was successfully used to monitor, analyse and control legacy systems in applications such as spam detection, instant messaging quality-of-service management and load balancing for geographical information systems [51]. Our generic autonomic framework addresses several key areas that are not supported by the approach in [50, 51]. By using a system model for the configuration of its policy engine, our architecture can be used for the autonomic management of heterogeneous types of resources. Moreover, our managed system can include resources beyond the software components handled by the infrastructure in [50].

In [52], the authors define an autonomic architecture meta-model that extends IBM's autonomic computing blueprint [13], and use a model-driven process to partly automate the generation of instances of this meta-model. Each instance is a special-purpose *organic computing system* that can handle the use cases defined by the model used for its generation. Our general-purpose autonomic architecture eliminates the need for the 19-step generation process described in [52] by using a policy engine that can be dynamically reconfigured to handle any use cases encoded within its system model and policy set.

A number of other projects have investigated isolated aspects related to the development of autonomic systems out of non-autonomic components. Some of these projects addressed the standardisation of the policy information model,

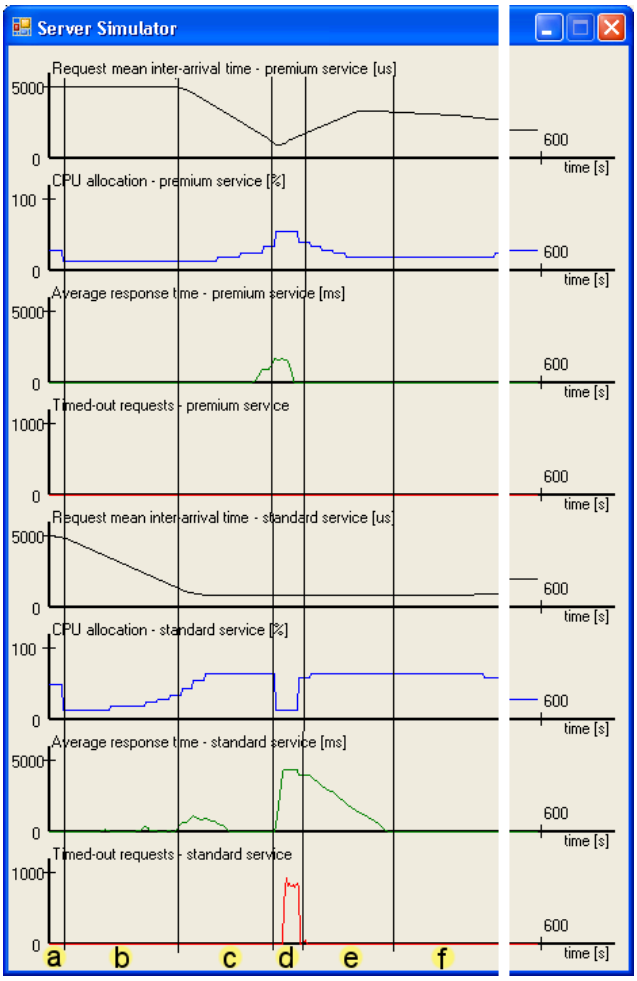

Figure 15. Utility-function results

with the Policy Core Information Model [34] 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 other applications [25]. An integrated development environment for the implementation of WSDM-compliant interfaces is currently available from IBM [26].

In a different area, expression languages were proposed for the specification of policy conditions and actions, and used to implement a range of policies [30, 33, 53, 54]. In addition to the development of standards and technologies, complete autonomic computing solutions have been produced recently [15, 37, 48], typically for the management of specific systems, and with limited ability to function in different scenarios from those they were originally intended for.

6. Conclusion

We described the SOA-based implementation of a generic framework intended to simplify significantly the development of autonomic systems, and thus to establish autonomic computing as a cost-effective approach to handling the spiralling complexity of today's computer systems. The ability to dynamically reconfigure the policy engine employed by the framework ensures that it can be used to build self-managing systems out of legacy and autonomic-enabled ICT resources whose characteristics are unknown until runtime, all without any modification to these resources or the policy engine.

Experimental work was carried out to validate the effectiveness of the SOA implementation of our autonomic computing framework. In this article, we presented a case study involving the development of two autonomic solutions for the allocation of server capacity to services of different priorities and varying workloads. The experimental results showed that our general-purpose framework could perform the planned management task successfully, and similarly to a dedicated commercial system for data-centre resource management [15, 55]. However, unlike the commercial resource-management system, our novel approach has the unique ability to handle resources whose types are unknown at implementation and deployment time, therefore enabling the cost-effective development of autonomic solutions across a broad variety of application domains—additional case studies from other application domains are described in [49].

The experimental results from the case studies presented in this article and in [49] suggest that the overheads associated with the evaluation of autonomic computing policies for realistic applications involving small ICT systems are acceptable. Thus, the realisation of relatively sophisticated utility-function policies far outweighs the observed utilisation of 1-2% of the CPU capacity and of a negligible amount of the memory of a low-end server. Furthermore, notice that this server need not be part of the managed system if self-management capabilites are added to a production ICT system that might be sensitive to such overheads: given its implementation as a web service, the policy engine can be deployed on a dedicated server. The only components to be deployed on the production system when this approach is used are the low-footprint manageability adaptors.

Clearly, the overhead levels mentioned above are characteristic of applications involving small to medium-sized ICT systems similar to those considered in our cases studies. Further work is planned to assess the scalability of the framework to large and very large ICT systems like those encountered in today's data centres.

Ongoing work is also dedicated to augmenting the SOA implementation by adding the policy engine components

and functionality specified by our framework but which are not supported by its current version (cf. Figure 3). In particular, we are looking at ways to integrate machine learning [41] into the prototype, along the lines of the work described in [56, 57]. Another research topic requiring investigation is the automated synthesis of effective autonomic computing policies, for instance for the scenario in which one instance of the policy engine is tasked with managing another instance of the same architecture, as described in the article.

Finally, devising "good" utility-function policies for complex ICT systems and avoiding conflicts within sets of such policies represent open research questions for the autonomic computing community. It is hoped that the availability of generic development frameworks such as the one described in this article will help address these questions by re-directing much of the effort involved in developing an autonomic system away from the implementation of its components and towards the design and analysis of its autonomic computing policies.

Acknowledgement This work was partly supported by the UK Engineering and Physical Sciences Research Council grant EP/F001096/1.

References

[1] R. Calinescu, "Implementation of a generic autonomic framework," in *Fourth International Conference on Autonomic and Autonomous Systems (ICAS 2008)*, D. Greenwood M. Grottke, H. Lutfiyya and M. Popescu, Eds., March 2008, pp. 124–129.

[2] T. Lenard and D. Britton, *The Digital Economy Factbook*. The Progress and Freedom Foundation, 2006.

[3] D. Tapscott, *Digital Economy: Promise and Peril in the Age of Networked Intelligence*. McGraw-Hill, 1997.

[4] Y. Bar-Yam, M. A. Allison, R. Batdorf, H. Chen, H. Generazio, H. Singh and S. Tucker, "The characteristics and emerging behaviors of system-of-systems," New England Complex Systems Institute, Tech. Rep., January 2004.

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

[6] R. Murch, *Autonomic Computing*. IBM Press, 2004.

[7] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer Journal*, vol. 36, no. 1, pp. 41–50, January 2003.

[8] S. Dobson, S. Denazis, A. Fernndez, D. Gaiti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt and F. Zambonelli, "A survey of autonomic communications," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 1, no. 2, pp. 223–259, December 2006.

[9] M. Hinchey and R. Sterritt, "Self-managing software," *Computer*, vol. 39, no. 2, pp. 107–109, February 2006.

[10] M. Parashar and S. Hariri, *Autonomic Computing: Concepts, Infrastructure & Applications*. CRC Press, 2006.

[11] R. Calinescu, "Challenges and best practices in policy-based autonomic architectures," in *Proc. 3rd IEEE Intl. Symp. Dependable, Autonomic and Secure Computing*, 2007, pp. 65–74.

[12] M. C. Huebscher and J. A. McCann, "A survey of autonomic computing—degrees, models, and applications," *ACM Comput. Surv.*, vol. 40, no. 3, pp. 1–28, 2008.

[13] IBM Corporation, "An architectural blueprint for autonomic computing," 2004, http://www-03.ibm.com/autonomic/pdfs/ACBP2_2004-10-04.pdf.

[14] R. Sterritt and M. Hinchey, "Biologically-inspired concepts for self-management of complexity," in *Proc. 11th IEEE Intl. Conf. Engineering of Complex Computer Systems*, 2006, pp. 163–168.

[15] R. Calinescu and J. Hill, "System providing methodology for policy-based resource allocation," July 2005, US Patent Application 20050149940.

[16] M. Devarakonda, D. Chess, I. Whalley, A. Segal, P. Goyal, A. Sachedina, K. Romanufa, E. Lassettre, W. Tetzlaff and B. Arnold, "Policy-based autonomic storage allocation," in *Self-Managing Distributed Systems*, ser. LNCS, vol. 2867. Springer, 2004, pp. 143–154.

[17] S. Ghanbari, G. Soundararajan, J. Chen and C. Amza, "Adaptive learning of metric correlations for temperature-aware database provisioning," in *Proc. 4th IEEE Intl. Conf. Autonomic Computing*, June 2007.

[18] C. Lefurgy, X. Wang and M. Ware, "Server-level power control," in *Proc. 4th IEEE Intl. Conf. Autonomic Computing*, June 2007.

[19] W.-S. Li, D. C. Zilio, V. S. Batra, M. Subramanian, C. Zuzarte and I. Narang, "Load balancing for multi-tiered database systems through autonomic placement

of materialized views," in *Proc. 22nd IEEE Intl. Conf. Data Engineering*, April 2006.

[20] R. Sterritt, M. Hinchey, C. Rouff, J. Rash and W. Truszkowski, "Sustainable and autonomic space exploration missions," in *Proc. 2nd IEEE Intl. Conf. Space Mission Challenges for Information Technology*, 2006, pp. 59–66.

[21] R. Calinescu, "Model-driven autonomic architecture," in *Proc. 4th IEEE Intl. Conf. Autonomic Computing*, 2007.

[22] R. Calinescu, "Towards a generic autonomic architecture for legacy resource management," in *Innovations and Advanced Techniques in Systems, Computing Sciences and Software Engineering*, K. Elleithy, Ed. Springer, 2008, pp. 410–415.

[23] W. Walsh, G. Tesauro, J. O. Kephart and R. Das, "Utility functions in autonomic systems," in *Proc. 1st Intl. Conf. Autonomic Computing*, 2004, pp. 70–77.

[24] O. Zimmermann, M. Tomlinson and S. Peuser, *Perspectives on Web Services: Applying SOAP, WSDL and UDDI to Real-World Projects*. Springer, 2005.

[25] B. Murray, K. Wilson and M. Ellison, "Web Services Distributed Management: MUWS primer," February 2006, oASIS WSDM Committee Draft, http://www.oasis-open.org/committees/download.php/17000/wsdm-1.0-muws-primer-cd-01.doc.

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

[27] Microsoft Corporation, "System Definition Model overview," April 2004, http://download.microsoft.com/download/b/3/8/b38239c7-2766-4632-9b13-33cf08fad522/sdmwp.doc.

[28] Microsoft Corporation, "Microsoft Dynamic Systems Initiative Overview," March 2005, http://download.microsoft.com/download/8/7/8/8783b65e-d619-46d7-aa8d-b4f13a97eeb0/DSIoverview.doc.

[29] J. Arwe, J. Boucher, P. Dublish, Z. Eckert, D. Ehnebuske, J. Hass, S. Jerman *et al.*, "Service Modeling Language, version 1.0," March 2007, http://www.w3.org/ Submission/2007/SUBM-sml-20070321.

[30] IBM Corporation, "Policy Management for Autonomic Computing, version 1.2," 2005, http://dl.alphaworks.ibm.com/technologies/pmac/PMAC12_sdd.pdf.

[31] L. Stojanovic, J. Schneider, A. Maedche, S. Libischer, R. Studer, T. Lumpp, A. Abecker, G. Breiter, and J. Dinger, "The role of ontologies in autonomic computing systems," *IBM Systems Journal*, vol. 43, no. 3, pp. 598–616, 2004.

[32] K. Breitman and M. Perazolo, "Using formal ontology representation and alignment strategies to enhance resource integration in multi vendor autonomic environments," in *Proceedings of the 4th IEEE International Workshop on Engineering of Autonomic and Autonomous Systems*, Tucson, AZ USA, 2007, pp. 117–126.

[33] D. Agrawal, J. Giles, K.-W. Lee and J. Lobo, "Autonomic Computing Expression Language (ACEL) 1.2: User's Guide," 2005, http://www-128.ibm.com/developerworks/edu/ac-dw-ac-acel-i.html.

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

[35] J. O. Kephart and W. E. Walsh, "An artificial intelligence perspective on autonomic computing policies," in *Proc. 5th IEEE Intl. Workshop on Policies for Distributed Systems and Networks*, 2004.

[36] S. White, J. E. Hanson, I. Whalley, D. M. Chess, and J. O. Kephart, "An architectural approach to autonomic computing," in *Proc. 1st IEEE Intl. Conf. Autonomic Computing*. IEEE Computer Society, 2004, pp. 2–9.

[37] Microsoft Corporation, "Windows System Resource Manager (WSRM) White Paper," August 2003, http://download.microsoft.com/download/a/7/a/a7a06462-1d80-4386-9505-91cca1e61940/WSRM%20Command-Line%20Interface.doc.

[38] J. Woodcock and J. Davies, *Using Z. Specification, Refinement and Proof*. Prentice Hall, 1996.

[39] M. Kwiatkowska, "Quantitative verification: Models, techniques and tools," in *Proc. 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM Press, September 2007, pp. 449–458.

[40] R. Harbird, S. Hailes and C. Mascolo, "Adaptive resource discovery for ubiquitous computing," in *Proc. 2nd Workshop Middleware for Pervasive and Ad-hoc Computing*, ser. ACM Intl. Conference Proceeding Series, vol. 77, October 2004, pp. 155–160.

[41] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2007.

[42] R. Calinescu and M. Kwiatkowska, "Using quantitative analysis to implement autonomic IT systems," in *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, May 2009, to appear.

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

[44] J. M. Sobel and D. P. Friedman, "An introduction to reflection-oriented programming," in *In Proceedings of Reflection96*, 1996.

[45] R. Garcia, J. Jarvi, A. Lumsdaine, J. G. Siek and J. Willcock, "A comparative study of language support for generic programming," *ACM SIGPLAN Notices*, vol. 38, no. 11, pp. 115–134, November 2003.

[46] "SAXON – The XSLT and XQuery Processor," http://saxon.sourceforge.net/.

[47] Microsoft Corporation, "Xml schema definition tool (xsd.exe)," 2007, http://msdn2.microsoft.com/en-us/library/x6c1kb0s(VS.80).aspx.

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

[49] R. Calinescu, "General-purpose autonomic computing," in *Autonomic Computing and Networking*, M. Denko *et al.*, Eds. Springer, June 2009.

[50] J. Parekh, G. Kaiser, P. Gross and G. Valetto, "Retrofitting autonomic capabilities onto legacy systems," *Cluster Computing*, vol. 9, no. 2, pp. 141–159, April 2006.

[51] G. Kaiser, J. Parekh, P. Gross and G. Valetto, "Kinesthetics extreme: An external infrastructure for monitoring distributed legacy systems," in *Proc. of the 5th Annual Intl. Active Middleware Workshop*, June 2003.

[52] H. Kasinger and B. Bauer, "Towards a model-driven software engineering methodology for organic computing systems," in *Proc. 4th Intl. Conf. Computational Intelligence*, July 2005, pp. 141–146.

[53] R. Anthony, "A policy-definition language and prototype implementation library for policy-based autonomic systems," in *Proceedings of the 4th IEEE International Conference on Autonomic Computing*, Dublin, Ireland, June 2006, pp. 265–276.

[54] N. Damianou, N. Dulay , E. Lupu and M. Sloman, "The Ponder policy specification language," in *Policies for Distributed Systems and Networks*, ser. LNCS, vol. 1995, Bristol, UK, 2001, pp. 18–38.

[55] B. McColl, "Intelligent, policy-driven orchestration of sensors and effectors across the data center in real-time," Sychron Inc, White paper, April 2004, http://hosteddocs.ittoolbox.com/BM042304.pdf.

[56] G. Tesauro, "Reinforcement learning in autonomic computing: A manifesto and case studies," *IEEE Internet Computing*, vol. 11, no. 1, pp. 22–30, January 2007.

[57] T. Lau, D. Oblinger, L. Bergman, V. Castelli and C. Anderson, "Learning procedures for autonomic computing," in *Proc. Workshop on AI and Autonomic Computing: Developing a Research Agenda for Self-Managing Computer Systems*, August 2003, http://tlau.org/research/papers/autonomic-ijcai2003.pdf.

[58] R. Calinescu, "Resource-definition policies for autonomic computing," in *Fifth International Conference on Autonomic and Autonomous Systems (ICAS 2009)*, IEEEComputer Society Press, April 2009, pp. 111–116.