General-Purpose Autonomic Computing

Radu Calinescu

Abstract The success of mainstream computing is largely due to the widespread availability of general-purpose architectures and of generic approaches that can be used to solve real-world problems cost-effectively and across a broad range of application domains. In this chapter, we propose that a similar generic framework is used to make the development of autonomic solutions cost effective, and to establish autonomic computing as a major approach to managing the complexity of today's large-scale systems and systems of systems. To demonstrate the feasibility of *general-purpose autonomic computing*, we introduce a generic autonomic computing framework comprising a policy-based autonomic architecture and a novel four-step method for the effective development of self-managing systems. A prototype implementation of the reconfigurable policy engine at the core of our architecture is then used to develop autonomic solutions for case studies from several application domains. Looking into the future, we describe a methodology for the engineering of self-managing systems that extends and generalises our autonomic computing framework further.

1 Introduction

The last decade has brought 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 [26]. The ability to accomplish more, faster and on a broader scale through expert use of ever more complex ICT systems is at the core of today's scientific discoveries, newly emerged services and everyday life. Autonomic computing represents an effective approach to managing the spiralling complexity of these systems by delegating their configuration, optimisation, repair and protection to the systems themselves [15, 21].

Radu Calinescu
Computing Laboratory, University of Oxford, UK, e-mail: Radu.Calinescu@comlab.ox.ac.uk

The research efforts of the past few years have generated a wealth of knowledge on what autonomic systems should look like [9, 13, 21, 31, 34] and what best practices to follow in building them [4, 16, 41, 43]. This 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 [8, 25, 27, 40, 42]. While these developments demonstrate the feasibility of the autonomic computing approach to complexity management, the current use of bespoke and domain-specific architectures, and of dedicated models and policies limits significantly the cost-effectiveness and reusability of today's autonomic solutions.

These limitations resemble the problems encountered in the early days of mainstream computing, and overcome successfully through the use of general-purpose architectures and generic approaches for the development of real-world applications across multiple application domains. We therefore propose that an equally generic framework is used to make the development of self-managing systems cost effective, and to drive standardisation, component reuse and user adoption in the realm of autonomic computing. Given that policy-based autonomic computing represents the most advanced approach to developing self-managing systems of practical utility, we describe below the criteria that a policy-based autonomic computing framework needs to satisfy in order to qualify as "general purpose":

C1 **Support for the whole range of software, hardware and data components encountered in real-world ICT systems.** To enable the development of effective autonomic systems for real-world applications, the framework should support the organisation of heterogeneous collections of existing and future ICT components into self-managing systems. Both components specifically designed for inclusion into a self-managing system (i.e., *autonomic-enabled ICT resources*) and components not originally intended for this purpose (i.e., *legacy ICT resources*) should be catered for.[1]

C2 **Support for a broad spectrum of self-* functional areas and autonomic computing policies.** The framework should aid the development of self-management capabilities spawning a rich spectrum of self-* functional areas, e.g., self-configuration, self-healing, self-optimisation and self-protection [21, 31, 34]. This must be achieved through supporting all types of autonomic computing policies, including action, goal and utility-function policies [44, 45].

C3 **Support for the cost-effective development of self-managing systems for a large variety of application domains and use cases.** The framework must reduce the effort and costs incurred in the development of today's autonomic systems significantly through enabling the extensive reuse of components and the sharing of autonomic computing models and policies. It should drive the standardisation of interfaces, policies, models and components for autonomic computing, and should allow and encourage the modular development of complex self-managing systems and systems of systems. Last but not least, the framework must provide a generic method for developing autonomic systems from any combination of legacy and/or autonomic-enabled ICT resources.

[1] The ICT components to be integrated into an autonomic system will be termed *(ICT) resources*.

To demonstrate the feasibility of general-purpose autonomic computing, we introduce a novel policy-based autonomic computing framework comprising an autonomic architecture designed around a reconfigurable policy engine, and a four-step method for the effective development of self-managing systems. This framework builds on recent advances in autonomic computing [9, 13, 17, 34], and extends the author's previous work in this area [4, 5, 6, 7] in several new directions. Thus, we describe for the first time how multiple instances of the same general-purpose autonomic architecture can be organised into self-managing systems of systems by means of a new type of autonomic policy termed a *resource-definition policy*. Also, we present the first-ever integration of quantitative model checking techniques [23, 24] into autonomic policy engines, and show how the use of this new capability enables the specification of powerful utility-function policies. Finally, we present a new four-step method for the development of self-managing systems starting from a model of their ICT resources, and we illustrate its application to several case studies that spawn different application domains and employ a wide range of policy types.

The remainder of the chapter is organised as follows. In Sect. 2, we contrast our framework with other approaches to autonomic solution development. We then describe the general-purpose autonomic architecture and the reconfigurable policy engine at its core in Sect. 3 and 4, respectively. A prototype implementation of the policy engine is presented in Sect. 5, followed by the description of our generic method for the development of self-managing systems in Sect. 6, and by several case studies that illustrate its use in a number of different real-world applications in Sect. 7. Sect. 8 analysis the extent to which our candidate general-purpose autonomic framework satisfies the criteria stated at the beginning of the chapter, and suggests ways for extending our current results.

2 Related Work

The autonomic infrastructure proposed in [35] 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 [19]. Our framework is building on the powerful approach in [19, 35], and has the added capability to handle heterogeneous types of resources unknown until runtime, and to support the development of autonomic systems of systems through the use of resource-definition policies.

In [20], the authors define an autonomic architecture meta-model that extends IBM's autonomic computing blueprint [16], 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-activity generation process described in [20] by using a universal policy engine that can be dynamically redeployed to handle any use cases encoded within its resource model and policy set.

Several research projects propose the use of Model-Driven Architecture (MDA) techniques to develop autonomic computing policies and self-managing systems starting from high-level behavioural models of the system or of its components [10, 36, 39]. Two of these approaches [10, 36] are targeted at bespoke systems whose components already exhibit sophisticated autonomic behaviour, and thus cannot be readily extended to handle generic legacy resources. In contrast, our framework can accommodate any type of ICT resource whose characteristics can be modelled as described in Sect. 6. The preliminary work described in [39] is closer to our approach in that it advocates the importance of using MDA techniques in the development of generic self-managing systems, however the authors do not substantiate their proposal with any concrete solution, but rather qualify it as an open challenge.

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, with the Policy Core Information Model [30] 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 [32]. An integrated development environment for the implementation of WSDM-compliant interfaces is currently available from IBM [17].

In [12], the authors take a view similar to ours by introducing a paradigm termed *model-driven autonomic computing*, and explaining that the model-based validation of self-management decisions represents a more reliable and flexible approach than the use of pre-set policies. A powerful hierarchical model of NASA's Autonomous Nano-Technology Swarm missions is successfully used in [12] to achieve the self-managing functionality that these missions depend on, and thus to illustrate the benefits of the approach. Our work complements the results in [12] with a new model-based approach to developing self-management functionality and a generic method that uses existing tools and standards for the implementation of autonomic systems.

Finally, we build on recent advances in component-based programming, by using an approach to ICT resource composition and dynamic configuration that resembles the one supported by reflective component models such as FRACTAL [3]. In additional to the FRACTAL functionality, our framework automates the generation of most component interfaces and the management of the targeted system.

3 General-Purpose Autonomic Architecture

Fig. 1 depicts our general-purpose autonomic architecture, a preliminary version of which was introduced in [5, 6]. The core component of the architecture is a universal policy engine that organises a heterogeneous collection of legacy ICT resources and autonomic-enabled resources into a self-managing system. To reduce the effort

required to develop autonomic solutions, the policy engine can handle resources whose types are unknown during its implementation and deployment. This unique capability is achieved through runtime configuration: a *model* of the system to be managed is supplied to the policy engine for this purpose. As a result, the engine can implement the high-level goals described by a set of user-specified policies that make reference to the resources defined in the system model.

As recommended by IBM's architectural blueprint for autonomic computing [16], standardised adaptors are used to expose the *manageability* of all types of legacy ICT resources in a uniform way, through sensor and effector interfaces. The autonomic-enabled resources in the self-managing system are either typical ICT resources designed to expose sensor and effector interfaces allowing their direct interoperation with the policy engine, or other instances of the architecture. The latter option is possible because the policy engine exposes the entire system as an atomic ICT resource through *high-level sensors* and *high-level effectors*. A detailed description of the architecture and an overview of existing standards and technologies that can be used to implement it in practice are available in [5, 6].

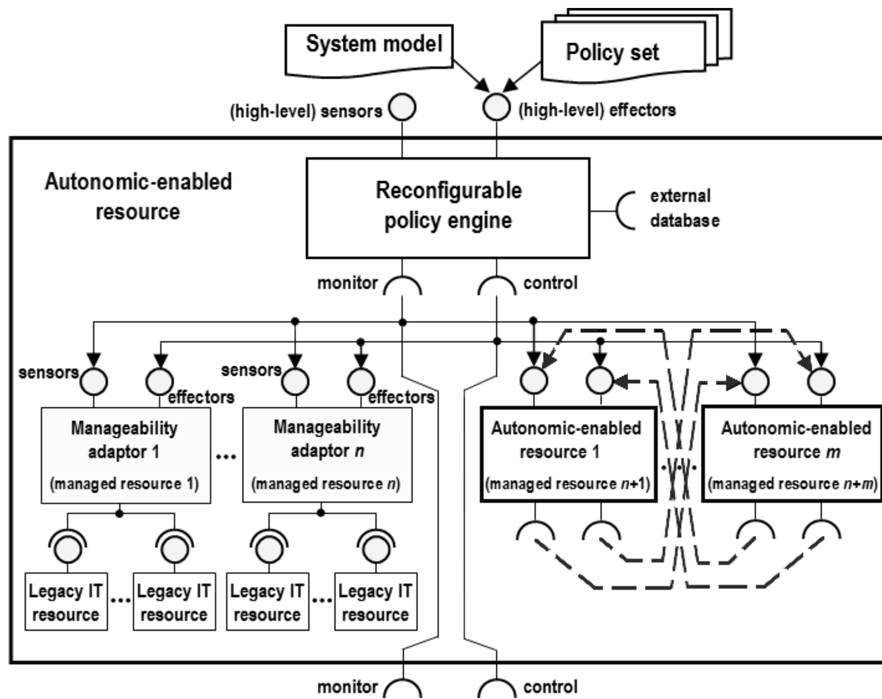

Fig. 1 UML component diagram of the autonomic architecture. The architecture supports the development of two types of autonomic systems-of-systems: a hierarchical topology that allows an instance of the policy engine to manage other instances of the architecture (i.e., the managed resources n+1 to n+m in the diagram); and a federation of collaborating instances of the architecture that use each others' high-level sensors and effectors, as shown by the dashed lines in the diagram.

4 Reconfigurable Policy Engine

The internal architecture of our policy engine (Fig. 2) is influenced by the types of policies it implements and by its ability to handle resources whose characteristics are supplied to the engine at runtime. A "coordinator" module is employing the following components to implement the closed control loop of an autonomic system:

- The *runtime code generator* produces 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 configure 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* are generated.
- The *manageability adaptor proxies* 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.
- The *high-level manageability adaptors* expose the system state and configuration in a format that allows its integration within other instances of the architecture. The way in which these interfaces are dynamically specified by means of resource-definition policies is described later in the chapter.

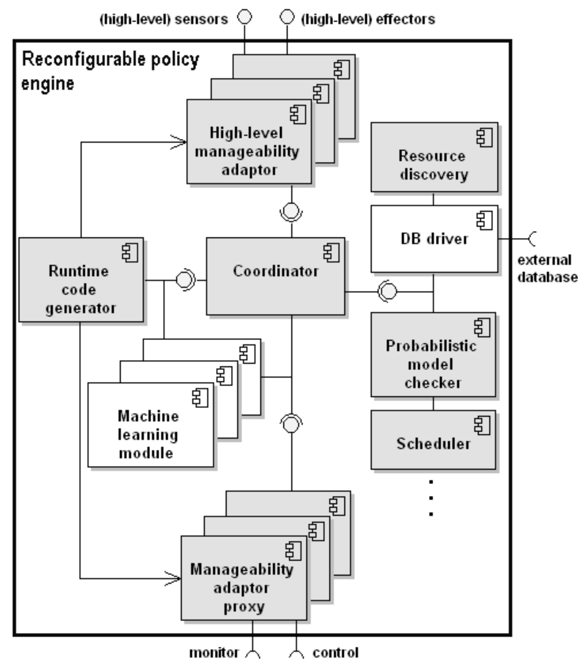

Fig. 2 Architecture of the reconfigurable policy engine. The shaded components are implemented by the prototype described in Section 5. A standards-based database driver will be added in a future version of the prototype. The machine learning modules represent the focus of ongoing research efforts by the autonomic computing community, and will be included in a reference implementation of the engine when the results of this research start to crystallise.

- The *scheduler* is used to support the scheduling operators appearing in policy actions for the goal and utility-function policies handled by the policy engine.
- The *resource discovery* component is used to locate the resources to be managed by the policy engine.
- The *database driver* is used to maintain policy engine data such as historical resource property values in an external persistent storage.
- The *machine learning modules* use machine learning techniques [2] to derive and/or refine a behavioural model of the managed resources based on sensor data and inside policy engine information. This enables the engine to support goal and utility-function policies for systems for which in-depth knowledge about the behavioural characteristics of the managed resources cannot be supplied by system administrator. The usefulness of a *Modeler* component for the implementation of utility-function policies is mentioned in [44], although the authors are not specific about the learning algorithms that such a component might use.
- The *probabilistic model checker* enables the policy engine to take full advantage of the behavioural model supplied by the system administrator or built by its machine learning modules. This is done by using probabilistic model checking to establish quantitative properties of the system [24] and thus to implement the user-specified policies. As will be illustrated by a couple of the case studies in Sect. 7, the integration of these *quantitative verification* techniques into the policy engine enables system administrators to specify powerful goal and utility-function policies that would have been extremely complicated or even impossible to express otherwise. Another use envisaged for the model checker is to help verify the policies implemented by the engine as suggested in [22].

5 Prototype Implementation

In this section we overview a prototype implementation of our autonomic architecture that was originally introduced in [7], and we describe for the first time two of its new features: the integration of a probabilistic model checker with the policy engine, and the implementation of resource-definition policies.

Two major choices influence the realisation of an instance of the architecture: the technology used to represent the system model; and the technology chosen for the implementation of the policy engine components. 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 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 we considered (e.g., [1, 29, 32]). 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.

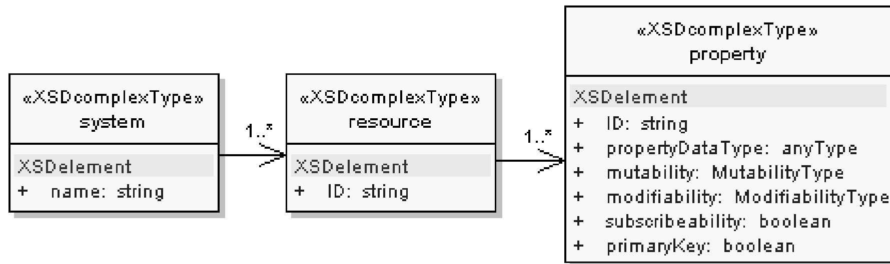

Fig. 3 Meta-model of an ICT system

As shown in Fig. 3, an ICT system is a named set of resources (*resource* in the UML diagram), each comprising a unique identifier *ID* and a set of resource properties with their characteristics. A resource property is associated a unique *ID*, and has a data type (i.e., *propertyDataType*). Several other property characteristics are defined in the meta-model:

- *mutability*—the WS-RMD MutabilityType [33] specifies if the property is "constant", "mutable" or "appendable";
- *modifiability*—tells if the property is "read-only", "read-write", "write-only" or "derived" from other properties and the behavioural model of the system;
- *subscribeability*—specifies whether a client such as the policy engine can subscribe to receive notifications when the value of this property changes;
- *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.

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 [46]. The runtime configuration of the engine required the extensive use of techniques available only in an object-oriented environment, e.g., runtime generation of data types and manageability adaptor proxies, reflection and generics. Based on these requirements, J2EE and .NET were selected as candidate development platforms 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. The components included in the prototype are shown in Fig. 2.

The free, open-source probabilistic model checker PRISM [14] developed by the Quantitative Analysis and Verification Group at the University of Oxford was chosen for integration with the original version of the policy engine described in [7]. This choice was based on an extensive performance analysis of a range of model checkers [18] that ranked PRISM as the best option for analysing large behavioural models such as the ones encountered in autonomic computing systems. Furthermore, PRISM comes with a command-line interface that made possible its direct integration into the existing version of the policy engine, and the runtime execution of *quantitative analysis experiments* [23, 24] that self-managing systems can use to realise powerful goal and utility-function policies as illustrated in Sect. 7.3–7.4.

Another novel feature of the policy engine that we describe for the first time is its ability to handle resource-definition policies, i.e., policies of the form

$$\text{RESDEF}(\mathit{newResourceId}, \mathit{propertyDef}_1, \ldots, \mathit{propertyDef}_m), \tag{1}$$

where *newResourceId* is a string corresponding to the *ID* element of a *resource* definition from the meta-model in Fig. 3 and

$$\mathit{propertyDef}_i = (\mathit{propertyId}_i, \mathit{expr}_i, \mathit{subscribeability}_i, \mathit{primaryKey}_i),\ 1 \leq i \leq m \tag{2}$$

define the properties of the new resource type. The $expr_i$ component in (2) tells the policy engine how to calculate the value of the i-th resource property as a function of the resources in the policy *scope*, or is one of `INTEGER`, `DOUBLE` or `STRING` to indicate that property i is a "read-write" property with one of these primitive types. The other components of $propertyDef_i$ correspond to the property characteristics from the system meta-model in Fig. 3 that cannot be inferred from $expr_i$. To implement a resource-definition policy, the policy engine generates dynamically the data type for the new resource and its manageability adaptor (i.e., a new web service whose URL is built by replacing the suffix `PolicyEngine.asmx` from the policy engine URL with *newResourceId*`ManageabilityAdaptor.asmx`). This manageability adaptor exposes objects of the new data type that are created and whose fields are set in accordance with the property definitions (2). The case study presented in Sect. 7.5 illustrates the use of resource-definition policies.

6 A Generic Method for the Development of Autonomic Systems

Our method for the development of autonomic systems comprises four steps:

1. development of a model of the system to which autonomic capabilities are added;
2. generation of manageability adaptors for the legacy resources in the system;
3. reconfiguration of the policy engine by means of the system model from step 1;
4. development of autonomic computing policies that handle the required use cases.

To illustrate these steps, we will apply them to a system comprising a set of services of different priorities, subjected to different workloads, and sharing the CPU capacity of the same server. The aim of the case study is to develop an autonomic solution for managing the allocation of CPU to services such that high-priority services are treated preferentially, subject to each service getting a minimum amount of CPU.

Several policy types are typically used in autonomic systems [44, 45]: *action policies* provide a low-level specification of how the system configuration should be changed to match its state; *goal policies* specify precise constraints that should be met by varying the system configuration; and *utility-function policies* supply a "measure of success" that the self-managing system should optimise by appropriately varying its configuration. In our running example we will use a utility-function policy, which is the most flexible of these policy types.

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) properties. Let S and C be the value domains for $\mathbf{s}$ and $\mathbf{c}$, respectively.[2] A behavioural model of the resource is a function

$$behaviouralModel : S \times C \to S, \tag{3}$$

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 $\mathbf{c}$.

Our policy engine works both with an approximation of the behavioural model that consists of a set of discrete values of the *behaviouralModel* in (3) and with a continuous-time Markov chain (CTMC) [23] representation of (3). For our running example, we will use the former type of behavioural model; the use of CTMC behavioural models is described in Sect. 7. As the current version of the policy engine does not include the machine learning modules described in Sect. 4, it acquires these behavioural models from the manageability adaptors for the managed resources. With the future addition of machine learning modules (Fig. 2), the policy engine will gain the ability to use learning techniques to refine and, eventually, to derive these behavioural models automatically based on its observation of the managed resources.

Step 1: Model Development Let *System* be the set of all instances of the metamodel in Fig. 3; the purpose of this step is to find a system model

$$M \in System \tag{4}$$

that can be used to implement the desired autonomic solution. To achieve this goal, we identify the system resources involved in the autonomic solution and their relevant properties. Given the ability to reconfigure the policy engine at any time, it makes sense to keep this model as simple as possible: additional resources and/or resource properties can be specified in new versions of the model, and conveyed to the policy engine as and when necessary. For instance, the single resource type for our example system is *service*, and its properties are: `name`, a unique identifier used to distinguish between different services; `priority`, an integer value; `cpuAllocation`, the percentage of the server CPU allocated to the service; `responseTime`, the service response time, averaged over the past one-second time interval; `interArrivalTime`, the request inter-arrival time, averaged over the past one-second time interval; and `behaviouralModel`, an approximation of the service behaviour that provides information on how the service response time varies with its CPU allocation and the request inter-arrival time.

Each resource property is then analysed in order to identify its value domain, mutability, modifiability and all of the other characteristics specified by the meta-

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

model in Fig. 3. This information is encoded as an instance of the system meta-model, ready to be used in the subsequent steps of the method. By analysing these resource properties for our running example and representing the analysis results as an instance of the system meta-model, we produced with the system model in Fig. 4.

Step 2: Manageability Adaptor Generation Given a system model M, this step generates manageability adaptors for each type of legacy resource. Off-the-shelf tools can be used to automate most of this generation. First, an XSLT transformation

$$schemaGen : System \rightarrow XmlSchema \tag{5}$$

is applied to the system model in order to obtain an XML schema for the resource types in the system. The XML schema generated when this transformation is applied to our sample system model is depicted as UML in Fig. 5a. A standard data type generator such as Microsoft's XML Schema Definition tool [28] is then used to automatically generate the data type set associated with this schema:

```
<system xmlns="...">
 <name>server</name>
 <!-- Services running within a server -->
 <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:minInclusive 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>responseTime</ID>
   . . .
  </property>
  <property>
   <ID>interArrivalTime</ID>
   . . .
  </property>
  <property>
   <ID>behaviouralModel</ID>
   <propertyDataType>
    <xs:complexType
                name="serviceBehaviouralModel">
     <xs:sequence>
      <xs:element name="modelElement"
                type="serviceModelElement"
                maxOccurs="unbounded"/>
     </xs:sequence>
    </xs:complexType>
    <xs:complexType name="serviceModelElement">
     <xs:sequence>
      <xs:element name="responseTime"
                type="serviceResponseTime"/>
      <xs:element name="interArrivalTime"
                type="serviceInterArrivalTime"/>
      <xs:element name="cpuAllocation"
                type="serviceCpuAllocation"/>
     </xs:sequence>
    </xs:complexType>
   </propertyDataType>
   <mutability>constant</mutability>
   <modifiability>read-only</modifiability>
   <subscribeability>false</subscribeability>
   <primaryKey>false</primaryKey>
  </property>
 </resource>
</system>
```

Fig. 4 System model for the running example

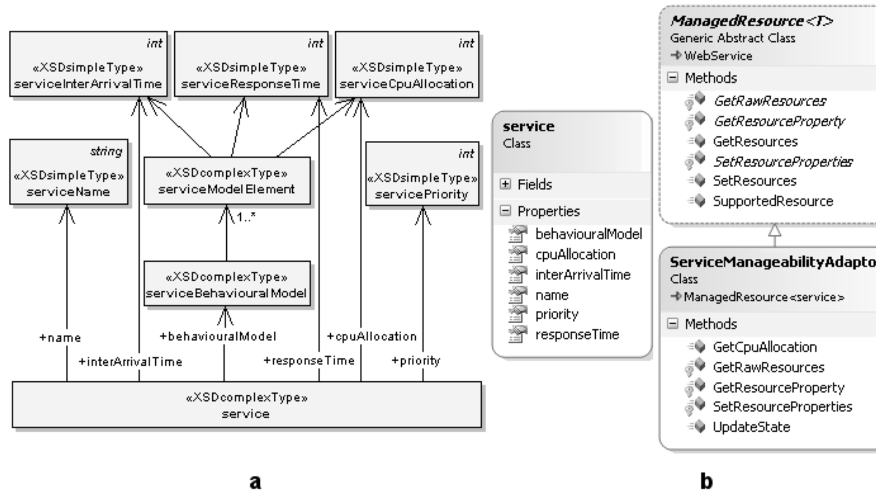

Fig. 5 Generated XML schema (a) and manageability adaptor (b) for the sample system

$$dataTypeGen : XmlSchema \to \mathbb{P}\, DataType. \tag{6}$$

Finally, a simple transformation was implemented to automate the generation of manageability adaptor stubs for the legacy resources in the system:

$$adaptorGen : XmlSchema \to \mathbb{P}\, ManageabilityAdaptor. \tag{7}$$

As shown in Fig. 5b, which depicts the data type (i.e., **service**) and the manageability adaptor (i.e., **ServiceManageabilityAdaptor**) for the system in our running example, all manageability adaptors are subclassing the generic abstract web service *ManagedResource*<*T*>. The bulk of the sensor and effector functionality associated with a manageability adaptor is implemented in this base abstract class, and only a small number of simple, resource-specific methods that are declared abstract in *ManagedResource*<*T*> need to be implemented manually in each manageability adaptor. Note that the policy engine is itself implemented as a subclass of *ManagedResource*<*T*>, so that an instance of the architecture can be readily included as a managed resource into a larger autonomic system as described in Sect. 3.

To complete this step, the manageability adaptor produced by the generator in (7) and depicted in Fig. 5b was manually extended, and then connected to a server discrete-event simulator running a high-priority 'premium' service and a low-priority 'standard' service. These services handled simulated requests with normally-distributed CPU utilisation and exponentially-distributed inter-arrival time.

Step 3: Engine Configuration This step consists in supplying the system model to the instance of the policy engine used in the autonomic solution. As stated before, the policy engine was realised as a web service, so we implemented a web interface for its simple configuration. Fig. 6 shows a snapshot of this interface after the sys-

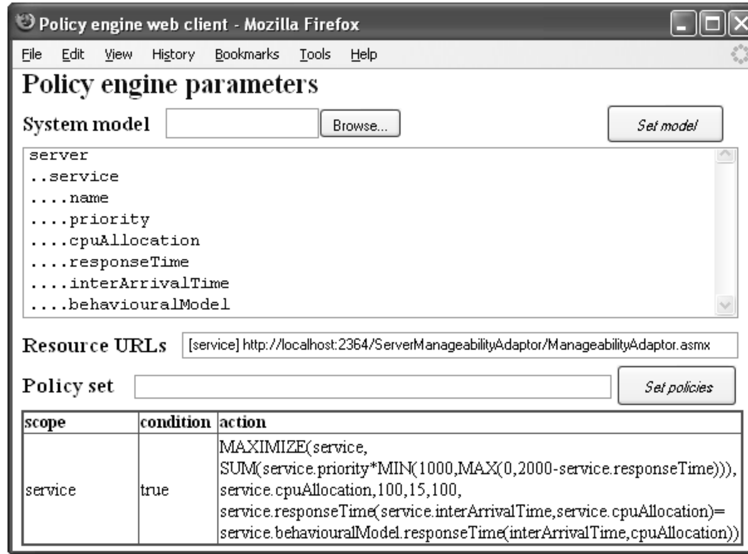

Fig. 6 Policy engine configuration

tem model from our running example, and the utility-function policy that will be presented in step 4 were supplied to the engine.

Step 4: Policy Development In this step, autonomic computing policies are designed that support the use cases of the envisaged autonomic solution. The scope, priority, condition and action components of these policies make reference to the resources and resource properties defined in the system model used to configure the policy engine. Each of these policy components can be specified using a rich set of operators and functions [6] that allow the definition of action, goal, utility-function and, in the latest version of the engine, of resource-definition policies.

The policy set is applied to all resources whose locations are known to the policy engine,[3] and which are in the scope of the policies. Policy development is generally a complex, error-prone and iterative process [4], and our framework improves the effectiveness of this process significantly by: (a) enabling and encouraging the reuse of system models and policies; and (b) simplifying the iterative development and testing of policies for new types of resources and of policies that explore the use of new properties of existing resources in novel ways.

For our autonomic solution, we defined a utility function that models the business gain associated with running a set of `service` resources R with different levels of service:

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

[3] The policy engine employs a resource discovery service (Fig. 2) to obtain the URLs of the resources to be managed.

Table 1 The arguments of the MAXIMIZE(R, *utility*, *property*, *capacity*, *min*, *max*, *model*) policy action for the running example of an autonomic system

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

Fig. 7a depicts the utility function for a server running a “premium” service with priority 100 and a “standard” service with priority 10. The policy action implemented by the autonomic system (Fig. 6 and Table 1) 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: (a) maximize the value of the *utility* function; and (b) 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*.

This policy provides the definition of the utility function, and the link between the `responseTime`, `interArrivalTime` and `cpuAllocation` properties of a `service` resource and 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. The experimental results of applying this policy to our example system are presented in Sect. 7.1.

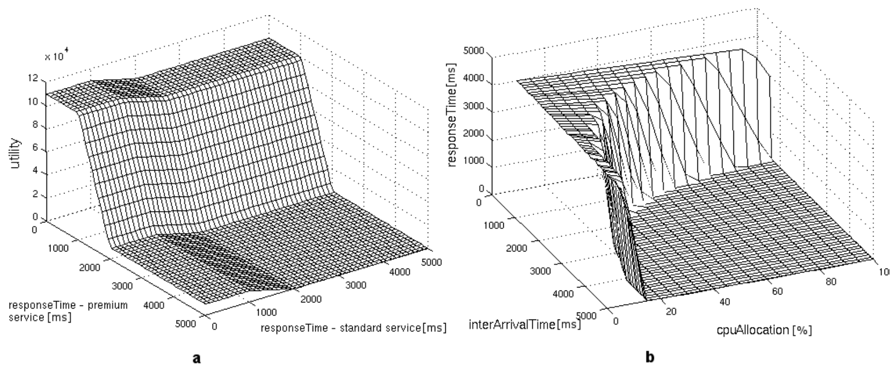

Fig. 7 Utility function (a) and service behavioural model (b) for the running example

7 Case Studies

7.1 Utility-Driven Allocation of CPU Capacity

We start our presentation of case studies with the experimental results for the running example of an autonomic system from the previous section. Variants of this system were used to validate autonomic computing frameworks in the past (e.g., [44]), hence this well-understood use case provides a good basis for a first assessment of the framework. To evaluate our autonomic solution, the behavioural model for a service was obtained from 100 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 (Fig. 7b). Fig. 8 shows a typical experiment in which the utility-function policy in Table 1 was used to manage the allocation of CPU to our 'premium' and 'standard' services, when their request inter-arrival times were varied to simulate different workloads. The

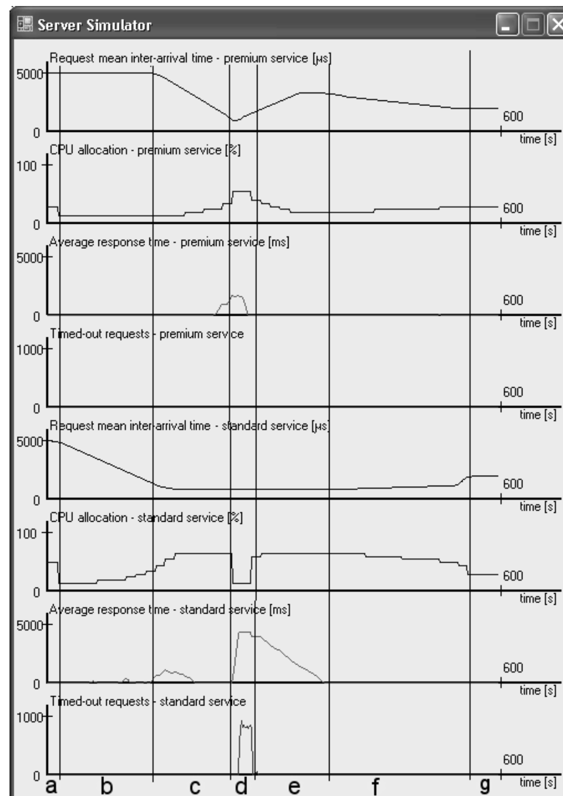

Fig. 8 Experimental results for Sect. 7.1. The CPU allocations for the services are initially decreased to match their light workload (5ms request inter-arrival time during time interval a). As the service workloads increase, so do the CPU allocations, until the CPU required to satisfy the demand from the premium service leaves insufficient CPU capacity for the standard service to make any contribution to the utility function (time interval d), hence it is allocated the minimum amount of CPU specified in the policy (i.e., 15%). As soon as less CPU capacity is required to satisfy the needs of the premium service (time interval e), the standard service is swiftly allocated sufficient CPU to bring it back into a region of operation in which it contributes to the utility function. Subsequently, the CPU allocations are varied to accommodate more gradual changes in the workloads (time intervals f-g).

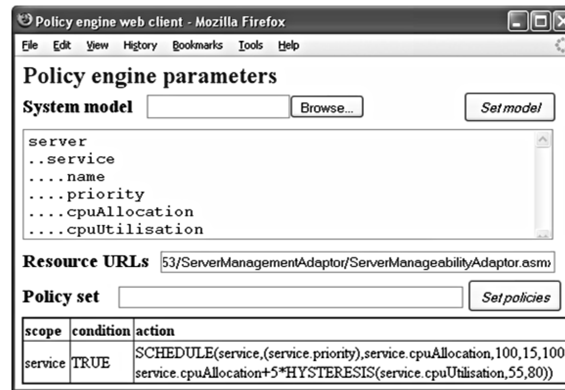

Fig. 9 Policy engine parameters for the case study in Sect. 7.2. The policy engine is configured to monitor the service `cpuUtilisation` (i.e., the amount of CPU utilised by the service, expressed as a percentage of its CPU allocation) and to realise a goal policy requiring that the `cpuUtilisation` is maintained between 55% and 80% of the allocated CPU.

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-optimise with far less frequent policy evaluations.

7.2 Goal-Based Scheduling of CPU Capacity

In the absence of knowledge about the behaviour of the legacy ICT resources that need to be organised into a self-managing system, goal policies can often be used in conjunction with scheduling heuristics. In this section, we consider the same system as in Sect. 7.1, but assume that a behavioural model describing the variation of the service response time with its allocated CPU and request inter-arrival rate is not available. Fig. 9 depicts a concise representation of the system model and a goal policy that can be used in this scenario. The action of this goal policy is specified by means of an expression that uses the SCHEDULE$(R, ordering, property, capacity, min, max, optimal)$ operator that: (a) sorts the resources in *R* in non-increasing order of the comparable expressions in *ordering*; (b) 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; and (c) ensures that the overall sum of all *property* values does not exceed the available *capacity*. Accordingly, the policy action in Fig. 9 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, the *cpuAllocation* should be left unchanged if the $55 \leq cpuUtilisation \leq 85$, decrease by 5% if $cpuUtilisation < 55$ and increase by 5% if $cpuUtilisation >$

85.[4] The experimental results for the resulting autonomic solution (available in [7]) resemble those corresponding to the use of a utility-function policy in Sect. 7.1, but are less effective in two important circumstances:

- several successive policy evaluations are required to handle significant changes in the service workloads because the CPU capacity allocated to services can be modified by only $\pm 5\%$ at a time;
- when insufficient CPU is available to ensure that a low-priority service runs in an operation area that is useful for the business and the utility-function policy in Sect. 7.1 would restrict the CPU allocated to the service to a minimum, the goal policy gives it all available CPU, thus wasting CPU capacity unnecessarily.

7.3 Dynamic Power Management of Disk Drives

When formal methods are used in the development and/or verification of legacy ICT resources, the behavioural models employed by these methods can often be exploited by our framework to augment the legacy ICT resources with autonomic capabilities. Starting from the continuous-time Markov chain (CTMC) model of a Fujitsu disk drive in [38] and its encoding as a PRISM CTMC model [37], we built (Fig. 10) a system model of the disk drive that can be used for the configuration of our policy engine. We then used this system model to add self-optimisation capabilities to the disk drive so that it dynamically adapted its probability of transitioning

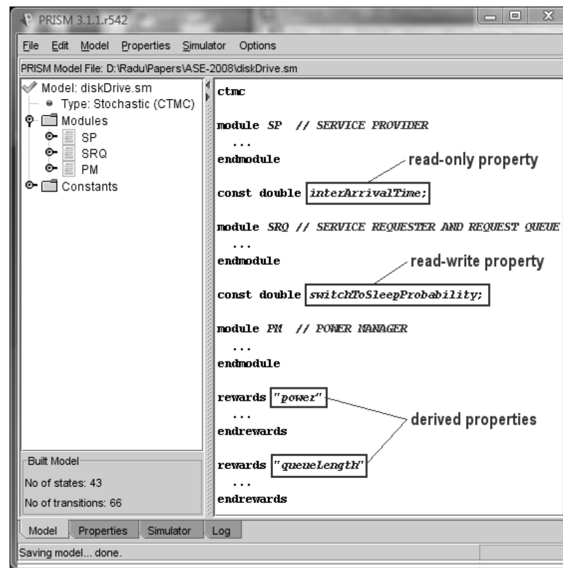

Fig. 10 PRISM CTMC model of a three-state Fujitsu disk drive taken from [37], and used to devise the system model for the configuration of the policy engine. The uninitialised PRISM constants correspond to "read-only" and "read-write" properties of a disk drive resource (i.e., `interArrivalTime` and `switchToSleepProbability`, respectively). PRISM reward structures (i.e., `power` and `queueLength`) correspond to "derived" disk drive properties.

[4] The HYSTERESIS$(val, lower, upper)$ operator used to achieve this behaviour (Fig. 9) returns -1, 0 or 1 if $val < lower$, $lower \leq val \leq upper$ or $upper < val$, respectively.

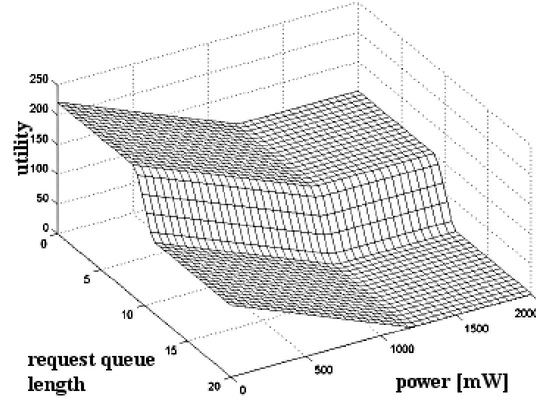

Fig. 11 The utility function (8) (depicted here for $w_1 = w_2 = 100$) was used to achieve a user-customisable trade-off between the disk drive responsiveness (which is provably proportional to its average `queueLength` [38]) and its power consumption (i.e., `power`).

from the *idle* state to the low-power *sleep* state to changes in (a) the request inter-arrival time; and (b) the user-specified utility function:

$$utility = w_1 \min\left(1, \max\left(0, \frac{11 - queueLength)}{2}\right)\right) + w_2 \max(0, 1.2 - power), \tag{8}$$

where the weights w_1 and w_2 are chosen depending on the circumstances in which the disk drive is used (Fig. 11). Given this policy, the policy engine ran PRISM *experiments* [24] to establish the optimal `switchToSleepProbability` for the disk drive at regular, 10-second time intervals. For our simple CTMC model, each of the these experiments took subsecond time, yielding the results in Fig. 12.

7.4 Adaptive Control of Cluster Availability

The case study presented in this section involves the adaptive control of cluster availability within a data centre. The aim of the autonomic solution is to control the number of servers allocated to the $N \geq 1$ clusters of a data centre in order to maximize the utility function

$$utility = \sum_{i=1}^{N} priority_i \cdot \mathrm{GOAL}(availability_i \geq target_availability_i) - \varepsilon \sum_{i=1}^{N} servers_i \tag{9}$$

subject to

$$\sum_{i=1}^{N} servers_i \leq Total_servers \text{ and } required_i \leq servers_i, \tag{10}$$

where $priority_i > 0$, $availability_i \in [0,1]$, $target_availability_i \in [0,1]$, $required_i \geq 1$ and $servers_i \geq 1$ represent the priority, (actual) availability, target availability, number of required servers, and number of (allocated) servers for cluster i, $1 \leq i \leq N$, respectively. The GOAL operator yields 1 when its argument is `true` and 0 otherwise, $Total_servers \geq 1$ is the total number of servers in the data centre, and $0 < \varepsilon \ll 1$ is a

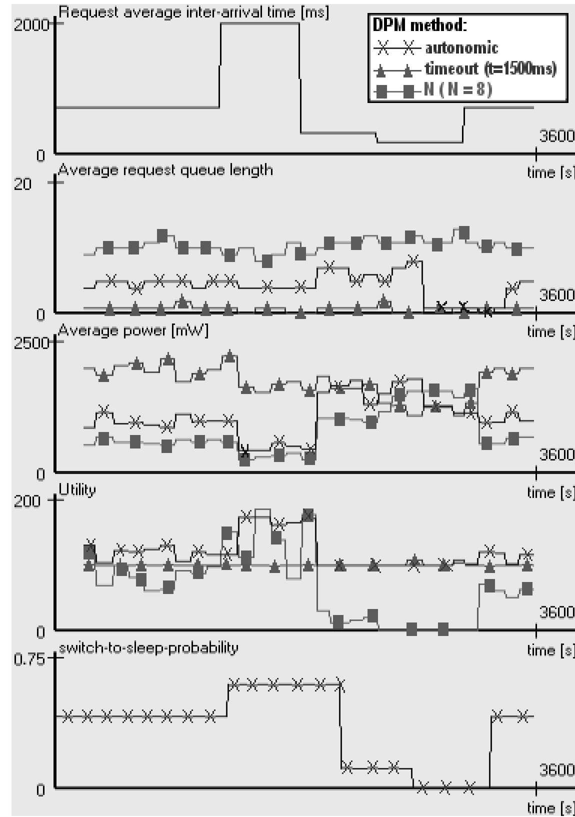

Fig. 12 Discrete-event simulation results contrasting our autonomic approach to disk drive dynamic power management (DPM) with two standard DPM methods [38]: the *timeout method* that moves the disk drive into the sleep state after a period of idleness *t* and "awakens" it immediately after a request has arrived; and the *N method* that moves the disk drive into the sleep state as soon as it becomes idle, and "awakens" it after *N* requests accumulate in its queue. The autonomic DPM approach achieved a better utility than the two standard DPM methods for most of the time, and similar utility to the better of the two for the rest of the time. This is due to the good trade-off that the autonomic approach realised between power consumption and request queue length across a wide workload range, while the other approaches are effective for specific workloads.

constant.[5] The availability of cluster i, $availability_i$, is the fraction of a one-year time period during which at least $required_i$ servers are usable (i.e., they are operational and connected to an operational switch and backbone).

Like in the previous case study, we extracted the system model for the configuration of our policy engine from an existing behavioural model of the targeted ICT resource, namely from the CTMC model of a dependable cluster of workstations introduced in [11]. This model takes into account the failure and repair rates of all components from our targeted cluster architecture (Fig. 13a). Consequently, the policy engine can use PRISM to calculate the cluster availabilities for the data-centre configurations satisfying (10), and to decide the number of servers that each cluster should get so that the value of the utility function (9) is maximised. Given the complexity of the CTMC behavioural model, we implemented a cluster manageability adaptor that uses notifications to inform the policy engine about changes in the number of required servers for the clusters. Hence, the policy engine recalculates the server allocations only when there is a change in the state of the autonomic

[5] The second term of the utility function (9) ensures that when multiple configurations maximise the first term, the configuration that uses the fewest servers is preferred.

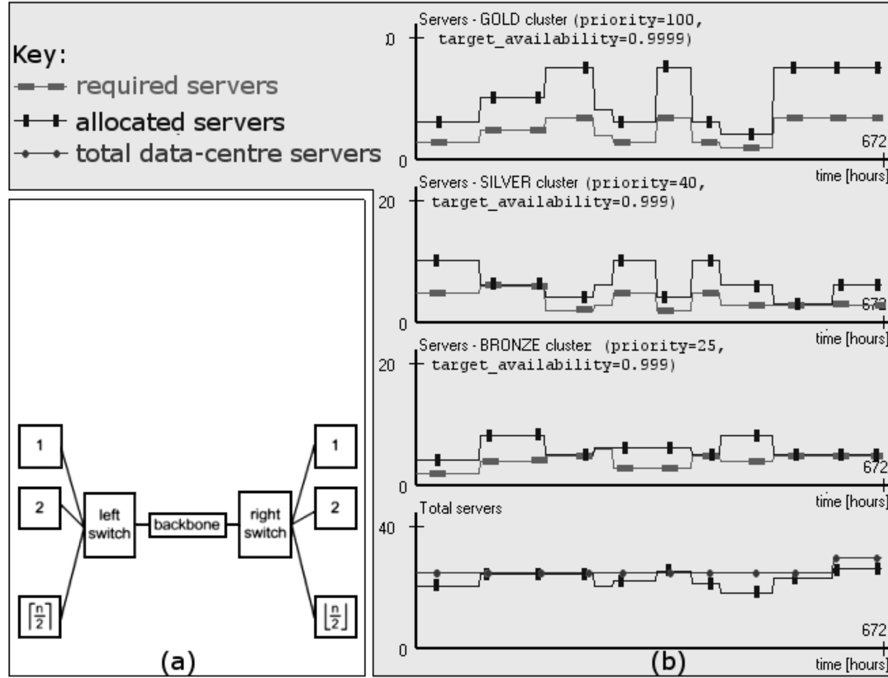

Fig. 13 Architecture of an *n*-server dependable cluster, taken from [11] (a), and simulation results for a three-cluster data centre over a four-week time period (b)

system. In our simulations, this calculation took up to 30 seconds. This response time is acceptable for the considered use case because, based on our previous experience with policy-based data centre management [4], half a minute represents a small delay compared to the time required to provision a server when it is allocated to a new cluster.[6] The experimental results are shown in Fig. 13b.

7.5 Dynamic Web Content Generation

The last case study is extending the autonomic solution from the previous section by incorporating the autonomic system for controlling cluster availability into an autonomic system of systems (Fig. 14). The resource-definition policy action below was supplied to policy engine instances within the autonomic data-centre systems:

$$\begin{aligned}&\mathrm{RESDEF}(\mathit{businessValue},(\mathit{id},\mathrm{CONCAT}(\mathit{cluster.id}),\mathit{false},\mathit{true}),\\&(\mathit{max},\mathrm{SUM}(\mathit{cluster.priority}),\mathit{true},\mathit{false}),(\mathit{actual},\mathrm{SUM}(\mathit{cluster.priority}*\\&\mathrm{GOAL}(\mathit{cluster.availability}>=\mathit{cluster.targetAvailability})),\mathit{true},\mathit{false})).\end{aligned} \tag{11}$$

[6] Sect. 8 suggests techniques for working around the time taken by runtime model checking when such delays are not acceptable.

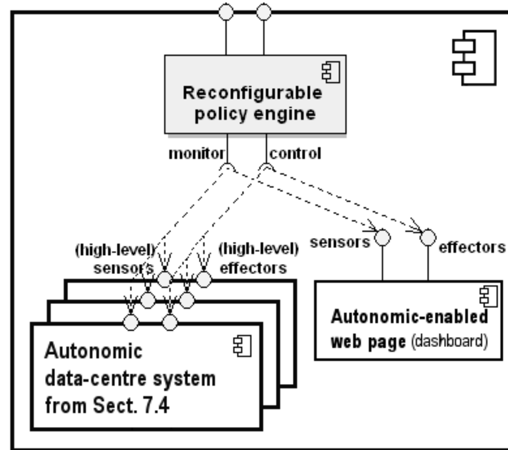

Fig. 14 Autonomic system of systems comprising several instances of the data-centre system from Sect. 7.4, and an autonomic-enabled web page implementing a business dashboard. The data-centre systems were each configured to expose their actual and ideal utility by means of a resource-definition policy, and the top-level policy engine implements an action policy that updates the properties of the autonomic-enabled web page with a summary of these utilities.

As described in Sect. 5, this resulted in each of these policy engines dynamically creating a new ICT resource named `businessValue` and comprising three "read-only" properties: `id`—the concatenated identifiers of its clusters; `max`—its ideal utility, i.e., the maximum possible value of the first term in (9); and `actual`—the actual value of this term. A model of this synthesised ICT resource and of an autonomic-enabled web page was then used to configure the top-level policy engine in Fig. 14, and an action policy was used to ensure that this policy engine updates the web page periodically with a summary based on the `businessValue` of each autonomic data-centre system it knows about (Fig. 15).

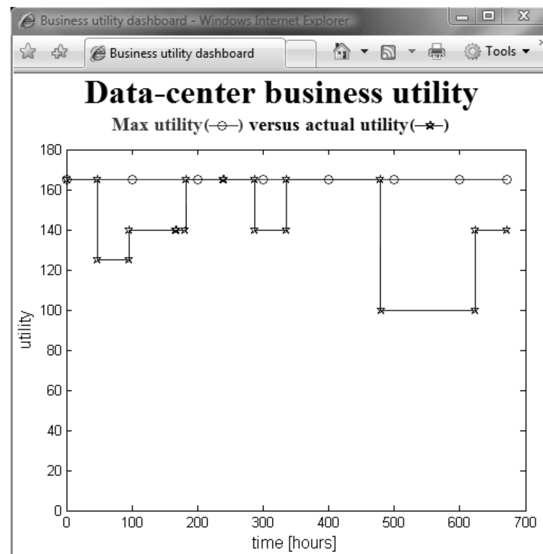

Fig. 15 An autonomic-enabled web page exposes effectors that the top-level policy engine uses to supply it with summary information about the maximum utility and actual utility of a set of autonomic data-centre systems (a single data-centre system was used in the experiment shown here). The web page presents the dynamically acquired information using a graphical representation that is generated at runtime using Matlab. Thus, the information about potential loss of business value is conveyed in a concise format that can be used directly by a data-centre manager.

8 Summary and Future Work

The success of mainstream computing is largely due to the availability of a system development methodology that enables and encourages standardisation, component reuse and user adoption. Building on recent advances in autonomic computing and on our previous work on policy-based autonomic systems, we proposed a general-purpose framework that brings similar benefits to the realm of autonomic computing. We introduced a set of criteria for assessing the generality of autonomic computing frameworks, and a new method for the development of self-managing systems starting from a model of their ICT resources. Also, we presented the integration of a probabilistic model checker into an autonomic computing policy engine, and we described how a new policy type termed a *resource-definition policy* can be used to build autonomic systems of systems.

To validate our framework, we employed it to build autonomic solutions spawning a range of application domains and using a variety of autonomic computing policies. Table 2 uses these case studies to analyse the extent to which the proposed framework satisfies the generality criteria **C1–C3** introduced in Sect. 1:

C1 In terms of supported ICT resources, our case studies demonstrate that the framework can handle the whole range of envisaged ICT resources.

C2 The framework has been used to develop autonomic solutions in several areas of self-* functionality, and to support all types of autonomic computing policies. To further confirm its generality, new applications are being currently investigated that address additional areas of self-* functionality.

C3 The autonomic systems developed for the presented case studies cover a range of application domains, including the development of a hierarchical system of systems. This is a good first step towards establishing that the framework satisfies this criterion. More work is required to assess the feasibility of using the framework in other use cases, and in particular in the development of federations of collaborating autonomic systems with no centralised management.

Table 2 Summary of the case studies presented in the paper

	C1 ICT resources					C2 self-* areas & policies					C3 application domain
	software	hardware	data	legacy	autonomic-enabled	main self-* functional areas	action	goal	utility-function	resource-definition	
Sect. 7.1	√			√		self-monitoring self-optimisation			√		CPU capacity allocation
Sect. 7.2	√			√		self-monitoring self-optimisation		√			CPU capacity allocation
Sect. 7.3		√		√		self-monitoring self-adaptation			√		dynamic power management
Sect. 7.4		√		√		self-configuration self-protection			√		cluster availability control
Sect. 7.5		√	√	√	√	self-monitoring self-generation	√		√	√	dynamic gen. of web content

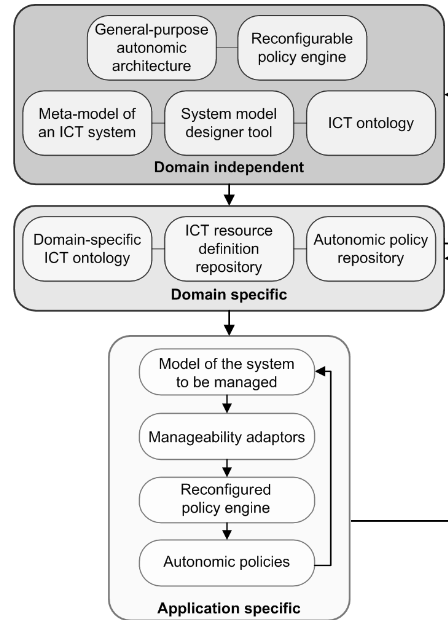

Fig. 16 Proposed autonomic system development methodology. The autonomic architecture, policy engine and system meta-model described in this paper are used at the domain-independent level, alongside a proposed ICT ontology and a proposed tool for designing the meta-model instances used to configure the policy engine. Repositories of ICT resource definitions and autonomic policies, and domain-specific ICT ontologies should be available at the level of an application domain, while our generic method for autonomic system development is employed for the cost-effective development of autonomic systems at the application-specific level.

Based on past experience in using a domain-specific autonomic framework [4] to develop systems similar to those in Sect. 7.1-7.2, we estimate that the use of the generic framework to build these systems reduced the development effort by roughly an order of magnitude, and we expect the same to hold true for other applications.

A key feature of our autonomic computing framework is its use of runtime probabilistic model checking. As shown in Sect. 7.4, model checking large systems can incur significant overheads, and the use of the subscription-notification mechanism supported by the framework (instead of periodical policy evaluation) is one way to accommodate this constraint. Other approaches to be investigated include the use of caching and pre-evaluation techniques to bypass the model checking step during policy evaluation, and the use of a hybrid approach in which a smaller model checking experiment is carried out to produce a close-to-optimal configuration for the autonomic system and a faster technique is then used to refine this configuration.

In addition to reusing components and techniques across a broad range of applications, our approach to autonomic system development allows and encourages the reuse of system models and autonomic computing policies. To take reusability further, these models and policies should draw their elements from domain-specific repositories of resource definitions and autonomic computing policies, respectively. Furthermore, to maximise the sharing of models, policies, manageability adaptors and autonomic-enabled resources, these repositories need to be built around controlled ICT ontologies, as required by the methodology for the cost-effective development of autonomic systems that we are proposing in Fig. 16. This methodology that we are working towards is in line with the excellent principles stated in [43] and successfully applied in the context of autonomic networking by [42].

Acknowledgements The work presented in this chapter was partly supported by the UK Engineering and Physical Sciences Research Council grant EP/F001096/1. The author is grateful to Marta Kwiatkowska, David Parker, Gethin Norman and Mark Kattenbelt for insightful discussions during the integration of the PRISM probabilistic model checker with the autonomic policy engine.

References

1. John Arwe et al. Service Modeling Language, version 1.0, March 2007. http://www.w3.org/Submission/2007/SUBM-sml-20070321.
2. Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2007.
3. E. Bruneton et al. The FRACTAL component model and its support in Java. *Softw. Pract. Exper.*, 36:1257–1284, 2006.
4. R. Calinescu. Challenges and best practices in policy-based autonomic architectures. In *Proc. 3rd IEEE Intl. Symp. Dependable, Autonomic and Secure Computing*, pages 65–74, 2007.
5. R. Calinescu. Model-driven autonomic architecture. In *Proc. 4th IEEE Intl. Conf. Autonomic Computing*, June 2007.
6. R. Calinescu. Towards a generic autonomic architecture for legacy resource management. In *Innovations and Advanced Techniques in Systems, Computing Sciences and Software Engineering*. Springer, 2008. To appear.
7. R. Calinescu. Implementation of a generic autonomic framework. In D. Greenwood et al., editor, *Proc. 4th Intl. Conf. Autonomic and Autonomous Systems*, pages 124–129, March 2008.
8. M. Devarakonda et al. Policy-based autonomic storage allocation. In *Self-Managing Distributed Systems*, volume 2867 of *LNCS*, pages 143–154. Springer, 2004.
9. S. Dobson et al. A survey of autonomic communications. *ACM Transactions on Autonomous and Adaptive Systems*, 1(2):223–259, December 2006.
10. D. Gracanin et al. Towards a model-driven architecture for autonomic systems. In *Proc. 11th IEEE Intl. Conf. Engineering of Computer-Based Systems*, pages 500–505, 2004.
11. B. Haverkort et al. On the use of model checking techniques for dependability evaluation. In *Proc. 19th IEEE Symp. Reliable Distributed Systems*, pages 228–237, October 2000.
12. M. Hinchey et al. Modeling for NASA autonomous nano-technology swarm missions and model-driven autonomic computing. In *Proc. 21st Intl. Conf. Advanced Networking and Applications*, pages 250–257, 2007.
13. M.G. Hinchey and R. Sterritt. Self-managing software. *Computer*, 39(2):107–109, Feb. 2006.
14. A. Hinton et al. PRISM: A tool for automatic verification of probabilistic systems. In H. Hermanns and J. Palsberg, editors, *Proc. 12th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *LNCS*, pages 441–444. Springer, 2006.
15. IBM Corporation. Autonomic computing: IBM's perspective on the state of information technology, October 2001.
16. IBM Corporation. An architectural blueprint for autonomic computing, 2004. http://www-03.ibm.com/autonomic/pdfs/ACBP2_2004-10-04.pdf.
17. IBM Corporation. Autonomic integrated development environment, April 2006. http://www.alphaworks.ibm.com/ tech/aide.
18. D.N. Jansen et al. How fast and fat is your probabilistic model checker? An experimental comparison. In K. Yorav, editor, *Hardware and Software: Verification and Testing*, volume 4489 of *LNCS*, pages 69–85. Springer, 2008.
19. G. Kaiser et al. Kinesthetics extreme: An external infrastructure for monitoring distributed legacy systems. In *Proc. of the 5th Annual Intl. Active Middleware Workshop*, June 2003.
20. H. Kasinger and B. Bauer. Towards a model-driven software engineering methodology for organic computing systems. In *Proc. 4th Intl. Conf. Comput. Intel.*, pages 141–146, 2005.
21. J.O. Kephart and D.M. Chess. The vision of autonomic computing. *IEEE Computer Journal*, 36(1):41–50, January 2003.
22. S. Kikuchi et al. Policy verification and validation framework based on model checking approach. In *Proc. 4th IEEE Intl. Conf. Autonomic Computing*, June 2007.

23. M. Kwiatkowska. Quantitative verification: Models, techniques and tools. In *Proc. 6th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. Foundations of Software Engineering*, pages 449–458. ACM Press, September 2007.
24. M. Kwiatkowska et al. Stochastic model checking. In M. Bernardo and J. Hillston, editors, *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07)*, volume 4486 of *LNCS*, pages 220–270. Springer, 2007.
25. C. Lefurgy et al. Server-level power control. In *Proc. 4th IEEE Intl. Conf. Autonomic Computing*, June 2007.
26. T. Lenard and D. Britton. *The Digital Economy Factbook*. The Progress and Freedom Foundation, 2006.
27. Wen-Syan Li et al. Load balancing for multi-tiered database systems through autonomic placement of materialized views. In *Proc. 22nd IEEE Intl. Conf. Data Engineering*, April 2006.
28. Microsoft Corporation. Xml schema definition tool (xsd.exe), 2007. http://msdn2.microsoft.com/en-us/library/x6c1kb0s(VS.80).aspx.
29. Microsoft Corporation. System Definition Model overview, April 2004. http://download.microsoft.com/download/b/3/8/b38239c7-2766-4632-9b13-33cf08fad522/sdmwp.doc.
30. B. Moore. Policy Core Information Model (PCIM) extensions, January 2003. IETF RFC 3460, http://www.ietf.org/rfc/rfc3460.txt.
31. Richard Murch. *Autonomic Computing*. IBM Press, 2004.
32. B. Murray et al. 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.
33. OASIS. Web Services Resource Metadata 1.0, November 2006.
34. M. Parashar and S. Hariri. *Autonomic Computing: Concepts, Infrastructure & Applications*. CRC Press, 2006.
35. J. Parekh et al. Retrofitting autonomic capabilities onto legacy systems. *Cluster Computing*, 9(2):141–159, April 2006.
36. J. Pena et al. A model-driven architecture approach for modeling, specifying and deploying policies in autonomous and autonomic systems. In *Proc. 2nd IEEE Intl. Symp. Dependable, Autonomic and Secure Computing*, pages 19–30, 2006.
37. PRISM Case Studies: Dynamic Power Management. http://www.prismmodelchecker.org/casestudies/power.php.
38. Q. Qiu et al. Stochastic modeling of a power-managed system: construction and optimization. In *Proc. Intl. Symp. Low Power Electronics and Design*, pages 194–199. ACM Press, 1999.
39. M. Rohr et al. Model-driven development of self-managing software systems. In *Proc. 9th Intl. Conf. Model-Driven Engineering Languages and Systems*. Springer, 2006.
40. R. Sterritt et al. Sustainable and autonomic space exploration missions. In *Proc. 2nd IEEE Intl. Conf. Space Mission Challenges for Information Technology*, pages 59–66, 2006.
41. R. Sterritt and M.G. Hinchey. Biologically-inspired concepts for self-management of complexity. In *Proc. 11th IEEE Intl. Conf. Engineering of Complex Computer Systems*, pages 163–168, 2006.
42. J. Strassner et al. Providing seamless mobility using the FOCALE autonomic architecture. In *Proc. 7th Intl. Conf. Next Generation Teletraffic and Wired/Wireless Advanced Networking*, volume 4712 of *LNCS*, pages 330–341, 2007.
43. J. Strassner et al. Ontologies in the engineering of management and autonomic systems: A reality check. *Journal of Network and Systems Management*, 15(1):5–11, 2007.
44. W.E. Walsh et al. Utility functions in autonomic systems. In *Proc. 1st Intl. Conf. Autonomic Computing*, pages 70–77, 2004.
45. S.R. White et al. An architectural approach to autonomic computing. In *Proc. 1st IEEE Intl. Conf. Autonomic Computing*, pages 2–9. IEEE Computer Society, 2004.
46. O. Zimmermann et al. *Perspectives on Web Services: Applying SOAP, WSDL and UDDI to Real-World Projects*. Springer, 2005.