2009 Fifth International Conference on Autonomic and Autonomous Systems

Resource-Definition Policies for Autonomic Computing

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

Abstract

The paper introduces a framework for the formal specification of autonomic computing policies, and uses it to define a new type of autonomic computing policy termed a resource-definition policy. *We describe the semantics of resource-definition policies, explain how they can be used as a basis for the development of autonomic system of systems, and present a sample data-centre application built using the new policy type.*

1. Introduction

A key objective of autonomic computing is to reduce the cost and expertise required for the management of complex IT systems [6], [7], [9]. As a growing number of these systems are implemented as hierarchies or federations of lower-level systems [4], techniques that support the development of *autonomic systems of systems* are required. This paper introduces one such technique, which is based on the use of a new type of autonomic computing policy. Termed a *resource-definition policy*, the new policy type specifies how the autonomic manager at the core of an autonomic system should expose the system to its environment.

As illustrated in Figure 1, the implementation of resource-definition policies requires that the reference architecture for autonomic computing loops from [7] is augmented with a "*synthesise*" step. Like in the reference architecture, an *autonomic manager* monitors the system resources through *sensors*, uses its *knowledge* to analyse their state and to plan changes in their configurable parameters, and implements these changes through *effectors*. Additionally, the autonomic manager module that implements the "synthesise" step has the role to dynamically generate sensor-effector interfaces that expose the underlying system to its environment as requested by the user-specified resource-definition policies. This new functionality supports the runtime integration of existing autonomic systems into hierarchical, collaborating or hybrid autonomic systems of systems, such as the one depicted in Figure 4 and described later in the paper.

The contributions of the paper include a formal framework for the unified specification and analysis of autonomic computing policies, and the introduction of resource-definition policies within this framework. While previous work by the author mentions resource-definition policies at a high level

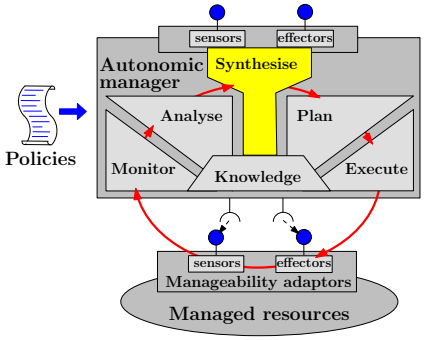

Figure 1. Autonomic computing system whose control loop is augmented with an interface synthesis step.

[3], this is the first time that the new type of autonomic computing policy is introduced formally, and its role and semantics are presented in detail.

The rest of the paper starts with an overview of related work in Section 2, followed by the description of our framework for the specification of autonomic computing policies in Section 3. Section 4 provides a formal introduction to resource-definition policies. Section 5 describes the prototype implementation of an autonomic manager that handles resource-definition policies. System-of-systems application development using the new policy type is discussed in Section 6, followed by concluding remarks in Section 7.

2. Related Work

The idea to integrate a set of autonomic systems into a higher-level autonomic system is not new. Indeed, the seminal paper of Kephart and Chess [7] envisaged *autonomic elements* that "interact with other elements and with human programmers via their autonomic managers". More recently, this vision was realised successfully by autonomic computing systems from areas including aerospace [5] and "beyond 3G" networking [10]. However, these systems are characterised by pre-defined types of interactions between their autonomic elements. Due to the statically implemented sensor-effector interfaces of their autonomic elements, the potential applications of these systems are limited to those pre-planned during their original development. In contrast, the use of resource-definition policies makes possible the automated, runtime generation of the sensor-effector interfaces of autonomic elements, thus supporting the development of unforeseen autonomic applications.

978-0-7695-3584-5/09 $25.00 © 2009 IEEE
DOI 10.1109/ICAS.2009.16

The architecture and functionality of the autonomic systems of systems built using resource-definition policies resemble those of intelligent multi-agent systems [12]. However, the typical approach to multi-agent system development [12] involves defining and implementing the agent interfaces statically, before the components of the system are deployed. In contrast, our approach has the advantage that these interfaces can flexibly be defined at runtime, thus enabling the development of applications not envisaged until after the components of the system are deployed.

A unified framework that interrelates three types of autonomic computing policies was introduced in [8]. Based on the concepts of *state* and *action* (i.e., state transition) adopted from the field of artificial intelligence, this framework makes possible the effective high-level analysis of the relationships among action, goal and utility-function policies for autonomic computing. Our policy specification framework differs essentially from the one in [8], which it complements through enabling the formal specification and analysis of all these policy types in terms of the knowledge employed by the autonomic manager that implements them.

3. A Framework for the Formal Specification of Autonomic Computing Policies

Before introducing the new type of policy, we will formally define the knowledge module of the autonomic manager in Figure 1. This knowledge is a tuple that models the $n \geq 1$ resources of the system and their behaviour:

$$K = (R_1, R_2, \ldots, R_n, f), \tag{1}$$

where R_i, $1 \leq i \leq n$ is a formal specification for the ith system resource, and f is a model of the *known* behaviour of the system. Each resource specification R_i represents a named sequence of $m_i \geq 1$ resource parameters, i.e.,

$$R_i = (resId_i, P_{i1}, P_{i2}, \ldots, P_{im_i}), \forall 1 \leq i \leq n, \tag{2}$$

where $resId_i$ is an identifier used to distinguish between different types of resources. Finally, for each $1 \leq i \leq n$, $1 \leq j \leq m_i$, the resource parameter P_{ij} is a tuple

$$P_{ij} = (paramId_{ij}, ValueDomain_{ij}, type_{ij}) \tag{3}$$

where $paramId_{ij}$ is a `string`-valued identifier used to distinguish the different parameters of a resource; $ValueDomain_{ij}$ is the set of possible values for P_{ij}; and $type_{ij} \in \{\texttt{ReadOnly}, \texttt{ReadWrite}\}$ specifies whether the autonomic manager can only read or can both read and modify the value of the parameter. The parameters of each resource must have different identifiers, i.e.,

$$\forall 1 \leq i \leq n \bullet \forall 1 \leq j < k \leq m_i \bullet paramId_{ij} \neq paramId_{ik}$$

We further define the *state space* S of the system as the Cartesian product of the value domains of all its `ReadOnly` resource parameters, i.e.,

$$S = \bigtimes_{1 \leq i \leq n} \bigtimes_{\substack{1 \leq j \leq m_i \\ type_{ij} = \texttt{ReadOnly}}} ValueDomain_{ij} \tag{4}$$

Similarly, the *configuration space* C of the system is defined as the Cartesian product of the value domains of all its `ReadWrite` resource parameters, i.e.,

$$C = \bigtimes_{1 \leq i \leq n} \bigtimes_{\substack{1 \leq j \leq m_i \\ type_{ij} = \texttt{ReadWrite}}} ValueDomain_{ij} \tag{5}$$

With this notation, the behavioural model f from (1) is a partial function[1]

$$f : S \times C \nrightarrow S$$

such that for any $(\mathbf{s}, \mathbf{c}) \in \mathrm{domain}(f)$, $f(\mathbf{s}, \mathbf{c})$ represents the *expected* future state of the system if its current state is $\mathbf{s} \in S$ and its configuration is set to $\mathbf{c} \in C$. Presenting classes of behavioural models that can support the implementation of different autonomic computing policies is beyond the scope of this paper; for a description of such models see [3].

The standard types of autonomic policies described in [8], [11] can be defined using this notation as follows:

1. An *action policy* specifies how the system configuration should be changed when the system reaches certain state/configuration combinations:
$$p_{\mathrm{action}} : S \times C \nrightarrow C. \tag{6}$$
Note that an action policy can be implemented even when $\mathrm{domain(f)} = \emptyset$ in (1).
2. A *goal policy* partitions the state/configuration combinations for the system into desirable and undesirable:
$$p_{\mathrm{goal}} : S \times C \rightarrow \{\texttt{true}, \texttt{false}\}, \tag{7}$$
with the autonomic manager requested to maintain the system in an operation area for which p_{goal} is `true`.
3. A *utility policy* associates a value with each state/configuration combination, and the autonomic manager should adjust the system configuration such as to maximise this value:
$$p_{\mathrm{utility}} : S \times C \rightarrow \mathbb{R}. \tag{8}$$

Example 1 To illustrate the application of the notation introduced so far, consider the example of an autonomic data-centre comprising a pool of $nServers \geq 0$ servers that need to be partitioned among the $N \geq 1$ services that the data-centre can provide. Assume that each such service has a $priority$ and is subjected to a variable $workload$. The knowledge (1) for this system can be expressed as a tuple

$$K = (ServerPool, Service_1, \ldots, Service_n, f) \tag{9}$$

where the models for the server pool and for a generic service i, $1 \leq i \leq N$ are given by:

1. A partial function on a set X is a function whose domain is a subset of X. We use the symbol $\nrightarrow$ to denote partial functions.

$$
\begin{aligned}
\mathit{ServerPool} &= (\texttt{"serverPool"},\\
&\quad\ (\texttt{"nServers"}, \mathbb{N}, \texttt{ReadOnly}),\\
&\quad\ (\texttt{"partition"}, \mathbb{N}^N, \texttt{ReadWrite}))\\
\mathit{Service}_i &= (\texttt{"service"},\\
&\quad\ (\texttt{"priority"}, \mathbb{N}_+, \texttt{ReadOnly}),\\
&\quad\ (\texttt{"workload"}, \mathbb{N}, \texttt{ReadOnly}))
\end{aligned}
\tag{10}
$$

The state and configuration spaces of the system are $S = \mathbb{N} \times (\mathbb{N}_+ \times \mathbb{N})^N$ and $C = \mathbb{N}^N$, respectively. For simplicity, we will consider that the *workload* of a service represents the minimum number of operational servers it requires to achieve its service-level agreement. Sample action, goal and utility policies for the system are specified below by giving their values for a generic data-centre state $s = (n, p_1, w_1, p_2, w_2, \ldots, p_N, w_N) \in S$ and configuration $c = (n_1, n_2, \ldots, n_N) \in C$:

$$
p_{\text{action}}(s, c) = (\lceil \alpha w_1 \rceil, \lceil \alpha w_2 \rceil, \ldots, \lceil \alpha w_N \rceil) \tag{11}
$$

$$
\begin{aligned}
&p_{\text{goal}}(s, c) = \forall 1 \le i \le N \bullet\\
&\quad (n_i > 0 \implies (\forall 1 \le j \le N \bullet p_j > p_i \implies n_j = \lceil \alpha w_j \rceil)) \ \wedge\\
&\quad \left(n_i = 0 \implies \sum_{\substack{j=1 \\ p_j \ge p_i}}^{N} n_j = n \right)
\end{aligned}
\tag{12}
$$

$$
p_{\text{utility}}(s, c) = \begin{cases} -\infty, & \text{if } \sum_{i=1}^{N} n_i > n \\ \sum_{\substack{i=1 \\ w_i > 0}}^{N} p_i u(w_i, n_i) - \epsilon \sum_{i=1}^{N} n_i, & \text{otherwise} \end{cases} \tag{13}
$$

The action policy (11) prescribes that $\lceil \alpha w_i \rceil$ servers are allocated to service i, $1 \le i \le N$ at all times. Notice how a *redundancy factor* $\alpha \in (1, 2)$ is used in a deliberately simplistic attempt to increase the likelihood that at least w_i servers will be available for service i in the presence of server failures. Also, the policy is (over)optimistically assuming that $n \ge \sum_{i=1}^{N} \lceil \alpha w_i \rceil$ at all times.

The goal policy (12) specifies that the desirable state/configuration combinations of the data-centre are those in which two conditions hold for each service. The first condition states that a service i should be allocated servers only if each service j of higher priority has already been allocated $\lceil \alpha w_j \rceil$ servers. The second condition states that a service should be allocated no server only if all n available servers were allocated to services of higher or equal priority.

Finally, the utility policy requires that the value of the expression in (13) is maximised. The value $-\infty$ in this expression is used to avoid the configurations in which more servers than the n available are allocated to the services. When this is not the case, the value of the policy is given by the combination of two sums. The first sum encodes the utility $u(w_i, n_i)$ of allocating n_i servers to each service i with $w_i > 0$, weighted by the priority p_i of the service.

$$
\begin{aligned}
&u : \mathsf{R}_+ \times \mathsf{R}_+ \to [0, 1]\\
&u(w, n) = \begin{cases} 0, & \text{if } n < (2 - \alpha) w \\ \frac{n - (2-\alpha)w}{2(\alpha - 1)w}, & \text{if } (2 - \alpha) w \le n \le \alpha w \\ 1, & \text{if } n > \alpha w \end{cases}\\
&\alpha \in (1, 2)
\end{aligned}
$$

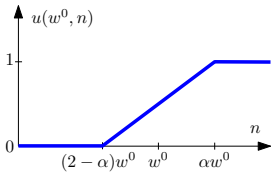

Figure 2. Sample function u for Example 1 (the graph shows u for a fixed value w^0 of its first argument)

By setting ϵ to a small positive value (i.e., $0 < \epsilon \ll 1$), the second sum ensures that from all server partitions that maximise the first sum, the one that uses the smallest number of servers is chosen at all times. A sample function u is shown in Figure 2; a more realistic u and a matching behavioural model f from (9) are described in [3].

4. Resource-Definition Policies

Definition Using $\mathcal{R}$ to denote the set of all resource models with the form in (2), and $\mathcal{E}(S, C)$ to denote the set of all expressions defined on the Cartesian product $S \times C$, we can now give the generic form of a resource-definition policy as

$$
p_{\text{def}} : S \times C \to \mathcal{R} \times \mathcal{E}(S, C)^q, \tag{14}
$$

where, for any $(\mathbf{s}, \mathbf{c}) \in S \times C$,

$$
p_{\text{def}}(\mathbf{s}, \mathbf{c}) = (R, E_1, E_2, \ldots, E_q). \tag{15}
$$

In this definition, R represents the resource that the autonomic manager is required to synthesise, and the expressions E_1, E_2, …, E_q specify how the autonomic manager will calculate the values of the $q \ge 0$ `ReadOnly` parameters of R as functions of $(\mathbf{s}, \mathbf{c})$. Assuming that the value domain for the ith `ReadOnly` parameter of R, $1 \le i \le q$ is $\mathit{ValueDomain}_i$, we have $E_i : S \times C \to \mathit{ValueDomain}_i$.

Example 2 Consider again the autonomic data-centre from Example 1. A sample resource-definition policy that complements the utility policy in (13) is given by

$$
\begin{aligned}
p_{\text{def}}(s, c) = ((&\texttt{"dataCentre"},\\
&\quad (\texttt{"id"}, \mathit{String}, \texttt{ReadOnly})\\
&\quad (\texttt{"maxUtility"}, \mathbb{R}, \texttt{ReadOnly}),\\
&\quad (\texttt{"utility"}, \mathbb{R}, \texttt{ReadOnly})),\\
&\texttt{"dataCentre A"},\\
&\quad \max_{(x_1, x_2, \ldots, x_N) \in \mathbb{N}^N} \textstyle\sum_{w_i > 0}^{1 \le i \le N} p_i u(w_i, x_i),\\
&\quad \textstyle\sum_{w_i > 0}^{1 \le i \le N} p_i u(w_i, n_i))
\end{aligned}
\tag{16}
$$

This policy requests the autonomic manager to synthesise a resource termed a `"dataCentre"` and comprising three `ReadOnly` parameters: `id` is a string-valued identifier with the constant value `"dataCentre A"`, while `maxUtility` and `utility` represent the maximum and actual utility values associated with the autonomic data-centre when it implements policy (13). (The term $\epsilon \sum_{i=1}^{N} n_i$ from the policy definition is insignificant, and was not included in (16) for simplicity.) Exposing the system through this synthesised

resource enables an external autonomic manager to monitor how close the data-center is to achieving its maximum utility.

Synthesised Resources with `ReadWrite` Parameters We will next explore the semantics and applications of `ReadWrite` (i.e., configurable) parameters in synthesised resources. These are parameters whose identifiers and value domains are specified through a resource-definition policy, but whose values are set by an external entity such as another autonomic manager. Because these parameters do not correspond to any element of the managed resources within the autonomic system, the only way ensure that they have an influence on the autonomic system in Figure 1 as a whole is to take them into account within the set of policies implemented by the autonomic manager. This is achieved by redefining the state space S of the system. Thus, in the presence of resource-definition policies requesting the synthesis of high-level resources with a non-empty set of `ReadWrite` parameters $\{P_1^{\text{synth}}, P_2^{\text{synth}}, \ldots, P_r^{\text{synth}}\}$, the state space definition (4) is replaced by:

$$S = \left(\bigtimes_{1 \le i \le n} \bigtimes_{\substack{1 \le j \le m_i \\ type_{ij} = \texttt{ReadOnly}}} ValueDomain_{ij} \right) \times \left(\bigtimes_{1 \le i \le r} ValueDomain_i^{\text{synth}} \right), \tag{17}$$

where $ValueDomain_i^{\text{synth}}$ represents the value domain of the ith synthesised resource parameter P_i^{synth}, $1 \le i \le r$.

Example 3 Consider again our running example of an autonomic data-centre. The resource-definition policy in (16) can be extended to allow a peer data-centre (such as a data-centre running the same set of services within the same security domain) to take advantage of any spare servers:

$$\begin{aligned} p'_{\text{def}}(s,c) = (&(\texttt{"dataCentre"}, \\ &\quad (\texttt{"id"}, \mathit{String}, \texttt{ReadOnly}) \\ &\quad (\texttt{"maxUtility"}, \mathbb{R}, \texttt{ReadOnly}), \\ &\quad (\texttt{"utility"}, \mathbb{R}, \texttt{ReadOnly})), \\ &\quad (\texttt{"nSpare"}, \mathbb{N}, \texttt{ReadOnly})), \\ &\quad (\texttt{"peerRequest"}, \mathbb{N}^N, \texttt{ReadWrite})), \\ &\texttt{"dataCentre A"}, \\ &\max_{(x_1,x_2,\ldots,x_N) \in \mathbb{N}^N} \textstyle\sum_{w_i>0}^{1 \le i \le N} p_i u(w_i, x_i), \\ &\textstyle\sum_{w_i>0}^{1 \le i \le N} p_i u(w_i, n_i),\ n - \sum_{i=1}^N n_i) \end{aligned} \tag{18}$$

The synthesised resource has two new parameters: `nSpare` represents the number of servers not allocated to any (local) service; and `peerRequest` is a vector $(n_1^l, n_2^l, \ldots, n_N^l)$ that a remote data-centre can set to request that the local data-centre assigns n_i^l of its servers to service i, for all $1 \le i \le N$.

To illustrate how this is achieved, we will consider two data-centres that each implements the policy in (18), and which have access to each other's "dataCentre" resource as shown in the lower half of Figure 4. For simplicity, we will further assume that the data-centres are responsible for disjoint sets of services (i.e., there is no $1 \le i \le N$ such that $w_i > 0$ for both data-centres). To ensure that the two data-centres collaborate, we need policies that specify how each of them should set the `peerRequest`^{r} parameter of its peer, and how it should use its own `peerRequest`^{l} parameter (which is set by the other data-centre). The "dataCentre" parameters have been annotated with ^{l} and ^{r} to distinguish between identically named parameters belonging to the local and remote data-centre, respectively. Before giving a utility policy that ensures the collaboration of the two data-centres, it is worth mentioning that the state of each has the form $s = (n, p_1, w_1, p_2, w_2, \ldots, p_N, w_N, n^r, n_1^l, n_2^l, \ldots, n_N^l)$ (cf. (17)); and the system configuration has the form $c = (n_1, n_2, \ldots, n_N, n_1^r, n_2^r, \ldots, n_N^r)$. The utility policy to use alongside policy (18) is given below:

$$p'_{\text{utility}}(s,c) = \begin{cases} -\infty, & \text{if } \sum_{i=1}^N n_i > n \vee \sum_{i=1}^N n_i^r > n^r \\ \sum\limits_{\substack{i=1 \\ w_i>0}}^N p_i u(w_i, n_i + n_i^r) - \epsilon \sum\limits_{i=1}^N n_i - & \\ \quad -\lambda \sum\limits_{i=1}^N n_i^r + \mu \sum\limits_{\substack{i=1 \\ n_i^l>0}}^N \min\left(1, \frac{n_i}{n_i^l}\right) & \text{, otherwise} \end{cases} \tag{19}$$

where $0 < \epsilon \ll \lambda, \mu \ll 1$ are user-specified constants. The value $-\infty$ is used to avoid the configurations in which more servers than available (either locally or from the remote data-centre) are allocated to the local services. The first two sums in the expression that handles all other scenarios are similar to those from utility policy (13), except that $n_i + n_i^r$ rather than n_i servers are being allocated to any local service i for which $w_i > 0$. The term $-\lambda \sum_{i=1}^N n_i^r$ ensures that the optimal utility is achieved with as few remote servers as possible, and the term $\mu \sum_{n_i^l>0}^{1 \le i \le N} \min(1, \frac{n_i}{n_i^l})$ requests the policy engine to allocate local servers to services for which $n_i^l > 0$. Observe that the contribution of a term $\mu \min(1, \frac{n_i}{n_i^l})$ to the overall utility increases as n_i grows from 0 to n_i^l, and stays constant if n_i increases beyond n_i^l. Together with the utility term $-\epsilon \sum_{i=1}^N n_i$, this determines the policy engine to never allocate more than the requested n_i^l servers to service i. Small positive constants are used for the weights ϵ, λ and μ so that the terms they belong to are negligible compared to the first utility term. Further, choosing $\epsilon \ll \lambda$ ensures that using a local server decreases the utility less than using a remote one; and setting $\epsilon \ll \mu$ ensures that allocating up to n_i^l servers to a service i at the request of the remote data-center increases the system utility.

Finally, since the requests for remote servers and the allocation of such servers happen asynchronously, there is a risk that the parameter values in policy (19) may be out of date. However, this is not a problem, as the

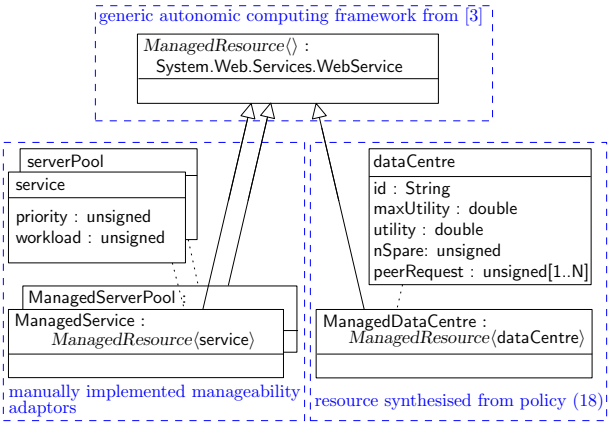

Figure 3. Class diagram for Example 4

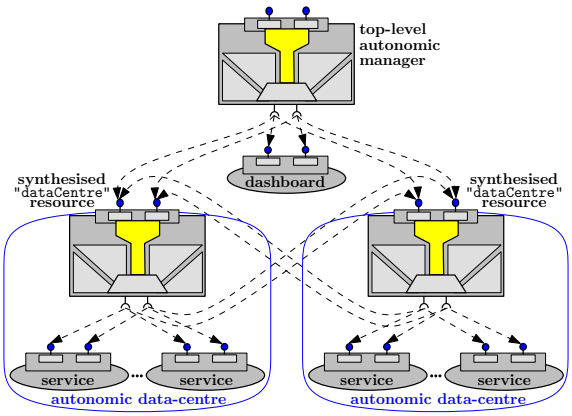

Figure 4. Autonomic system of systems

allocation of fewer or more remote servers than ideally required is never decreasing the utility of a data-centre below the value achieved when the data-centre operates in isolation. Additionally, local servers are never used for remote services at the expense of the local services because $\sum_{w_i>0}^{1\leq i\leq N} p_i u(w_i, n_i) \gg \mu \sum_{n_i^l>0}^{1\leq i\leq N} \min(1, n_i/n_i^l))$ in (19).

5. Prototype Implementation

The *policy engine* (i.e., policy-based autonomic manager) introduced by the author in [2] was extended with the ability to handle resource-definition policies. Implemented as a model-driven, service-oriented architecture with the characteristics presented in [1], the policy engine from [2] can manage IT resources whose model is supplied to the engine in a runtime configuration step. The policy engine is implemented as a .NET web service, and the manageability adaptors from Figure 1 are implemented as web services that specialise a generic, abstract web service *ManagedResource*⟨⟩. For each type of resource in the system, a manageability adaptor is built in two steps. First, a class (i.e., a data type) T_i is generated from the resource model (2) that will be used to configure the policy engine. Second, the manageability adaptor $\mathsf{ManagedT}_i$ for resources of type T_i is implemented by specialising our generic *ManagedResource*⟨⟩ adaptor, i.e., $\mathsf{ManagedT}_i$: *ManagedResource*⟨T_i⟩. This process is described in [2]. Adding support for the implementation of resource-definition policies involved extending the policy engine with the following functionality:

1. Automated generation of a .NET class T for the synthesised resource R from (15). This class is built by including a field and the associated getter/setter methods for each parameter of R. The types of these fields are given by the value domains of the resource parameters.
2. Automated creation of an instance of T using *reflection* (i.e., an object-oriented programming technique that allows the runtime discovery and creation of objects based on their metadata). The `ReadOnly` fields of this object are updated by evaluating expressions E_1, E_2, …, E_q whenever the object is accessed by an external entity.
3. Automatic generation of a manageability adaptor web service ManagedT : *ManagedResource*⟨T⟩. The web methods provided by this manageability adaptor allow entities from outside the autonomic system (e.g., external autonomic managers) to access the object of type T maintained by the policy engine. The fields of this object that correspond to `ReadOnly` parameters of R can be read, and those corresponding to `ReadWrite` parameters can be read and modified, respectively.

The .NET components generated in steps 1 and 3 are deployed automatically, and made accessible through the same Microsoft IIS instance as the policy engine.

Example 4 Returning to our running example of an autonomic data-centre, the class diagram in Figure 3 depicts the manageability adaptors in place after policy (18) was supplied to the policy engine. Thus, the ManagedServerPool and ManagedService classes in this diagram represent the manageability adaptors implemented manually for the *ServerPool* and *Service* resources described in Example 1. The other manageability adaptor derived from *ManagedResource*⟨⟩ (i.e., ManagedDataCentre) was synthesised automatically by the policy engine as a result of handling the resource-definition policy. Also shown in the diagram are the classes used to represent instances of the IT resources within the system—serverPool and service for the original autonomic system, and dataCentre for the resource synthesised from policy (18).

6. Application Development

System-of-systems application development with resource-definition policies involves supplying such policies to existing autonomic systems whose autonomic managers support the new policy type. Hierarchical systems of systems can then be built by setting a higher-level autonomic manager to monitor and/or control the resources synthesised as a result of implementing these policies. Alternatively, the original autonomic systems can be configured to collaborate with each other by means of the synthesised resource sensors

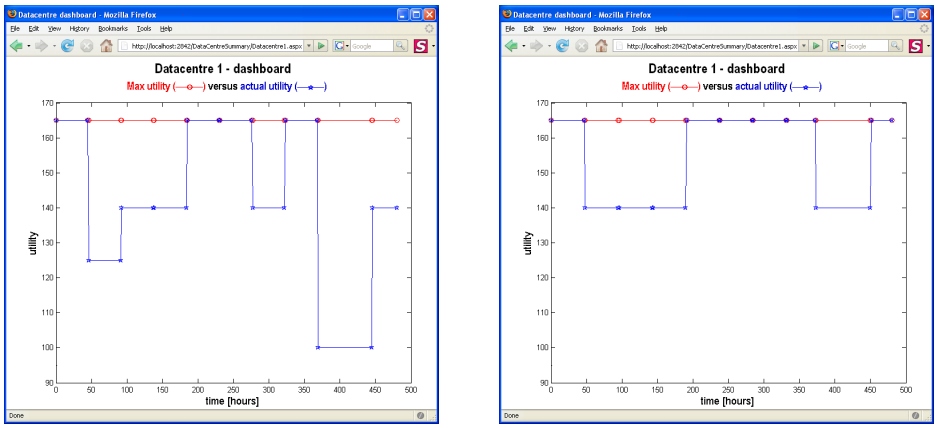

Figure 5. Dashboard for isolated data-centre (left) and for identical data-centre operating as part of the autonomic system of systems from Figure 4 (right)

and effectors. Hybrid applications comprising both types of interactions are also possible, as illustrated below.

Example 5 The policy engine from Section 5 was used to simulate an autonomic system of systems comprising the pair of autonomic data-centres described in Example 3, and a top-level autonomic manager that monitors and summarises their performance using a dashboard resource (Figure 4). The policies implemented by the autonomic managers local to each data-centre are policies (18)–(19) from Example 3. The top-level autonomic manager implements a simple action policy that periodically copies the values of the `maxUtility` and `utility` parameters of the `"dataCentre"` resources synthesised by the data-centres into the appropriate `ReadWrite` parameters of the dashboard. For brevity, we do not give this policy here; a sample action policy was presented earlier in Example 1.

We used the data-centre simulators from [3], and implemented the dashboard as an ASP.NET web page provided with a manageability adaptor built manually as described in Section 5 and in [2]. Separate series of experiments for 20-day simulated time periods were run for two scenarios. In the first scenario, the data-centres were kept operating in isolation, by blocking the mechanisms they could use to discover each other. In the second scenario, the data-centres were allowed to discover each other, and thus to collaborate through implementing policy (19). Figure 5 depicts typical snapshots of the dashboard for both scenarios; the same workloads were used in both experiments shown. As expected from the analysis in Example 3, the system achieves higher utility when data-centre collaboration is enabled.

7. Conclusions

We introduced a policy specification framework for autonomic computing, and used it to formally define the existing types of autonomic computing policies from [8], [11] and a new type of policy that we termed a resource-definition policy. The semantics of resource-definition policies and their use in the development of hierarchical, collaborating and hybrid autonomic systems of systems are described in the paper. Also, we presented the change to the reference autonomic computing loop from [7] that is required to implement these policies, and described a prototype implementation of an autonomic manager that supports them.

Acknowledgement This work was partly supported by the UK EPSRC grant EP/F001096/1.

References

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

[2] R. Calinescu. Implementation of a generic autonomic framework. In D. Greenwood et al., editors, *Proc. 4th Intl. Conf. Autonomic and Autonomous Systems*, pages 124–129, 2008.

[3] R. Calinescu. General-purpose autonomic computing. In M. Denko et al., editors, *Autonomic Computing and Networking*. Springer, April 2009.

[4] G. Goth. Ultralarge systems: Redefining software engineering? *IEEE Software*, 25(3):91–94, May/June 2008.

[5] M. Hinchey et al. Modeling for NASA autonomous nanotechnology swarm missions and model-driven autonomic computing. In *Proc. 21st Intl. Conf. Advanced Networking and Applications*, pages 250–257, 2007.

[6] M. Huebscher and J. McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Comput. Surv.*, 40(3):1–28, 2008.

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

[8] 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.

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

[10] D. Raymer et al. From autonomic computing to autonomic networking: an architectural perspective. In *Proc. 5th IEEE Workshop on Engineering of Autonomic and Autonomous Systems*, pages 174–183, 2008.

[11] W. Walsh et al. Utility functions in autonomic systems. In *Proc. 1st Intl. Conf. Autonomic Computing*, pp. 70–77, 2004.

[12] M. Wooldridge. *An Introduction to Multi-agent Systems*. J. Wiley and Sons, 2002.