Run-Time Connector Synthesis for Autonomic Systems of Systems

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

***Abstract*—A key objective of autonomic computing is to reduce the cost and expertise required for the management of complex IT systems. As a growing number of these systems are implemented as hierarchies or federations of lower-level systems, techniques that support the development of *autonomic systems of systems* are required. This article introduces one such technique, which involves the run-time synthesis of *autonomic system connectors*. These connectors are specified by means of a new type of autonomic computing policy termed a *resource-definition policy*, and enable the dynamic realisation of collections of collaborating autonomic systems, as envisaged by the original vision of autonomic computing. We propose a framework for the formal specification of autonomic computing policies, and use it to define the new policy type and to describe its application to the development of autonomic system of systems. To validate the approach, we present a sample data-centre application that was built using connectors synthesised from resource-definition policies.**

***Keywords*—autonomic computing; systems of systems; autonomic system connectors; model-driven development; automated code generation.**

I. Introduction

The original vision of autonomic computing was one of self-managing systems comprising multiple inter-operating, collaborating *autonomic elements* [2]. Like the components of a biological system, these autonomic elements were envisaged to contain well-differentiated *resources* and deliver *services* to each other, to integrate seamlessly, and to work together towards a global set of system-level objectives [2][3][4].

This view of self-managing systems as collections of collaborating autonomic elements is entirely consistent with the way in which many of the most important computing systems are designed, built and managed. Indeed, key computing systems in areas ranging from transportation and healthcare to aerospace and defence can no longer be implemented as monolithic systems. Instead, the capabilities and functionality that these systems are required to provide can only be achieved through employing collections of collaborative, heterogeneous and autonomously-operating systems [5][6].

Thus, the characteristics of the computing systems that would benefit most from self-managing capabilities, and the vision behind the sustained research efforts in the area of autonomic computing are well aligned. Nevertheless, the same is hardly true about the outcome of these efforts: what the overwhelming majority of the autonomic computing research to date has produced is a collection of methods, tools and components for the development of *monolithic* autonomic systems; and exemplars of such systems. The few notable exceptions (overviewed in Section II) are either high-level architectures [7] that refine the "collection-of-autonomic-elements" architecture from the seminal autonomic computing paper [2], or domain-specific instances of this architecture that involve pre-defined interactions between small numbers of autonomic element types [8][9][10][11].

What is needed to bridge this gap between vision and reality is a generic technique for the development of systems comprising multiple autonomic elements. The technique should support autonomic elements that can join and leave the system dynamically, which have local objectives to achieve in addition to the overall system objectives, and whose characteristics are unknown at system design time.

This extended version of [1] presents such a generic technique for the development of *autonomic systems of systems* by means of a new type of autonomic computing policy. The new policy type is termed a *resource-definition policy*, and defines the *autonomic system connectors* through which 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 autonomic computing loop from [2] 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 (or "executes") these changes through *effectors*. Additionally, the autonomic manager module that implements the "synthesise" step has the role to dynamically generate connectors that expose the underlying system to its environment as requested by the user-specified resource-definition policies. This new functionality supports the flexible, run-time integration of existing autonomic systems into hierarchical, collaborating or hybrid autonomic systems of systems, as illustrated in Figure 2.

The main contributions of the article are:

1) A theoretical framework for the formal, unified specification and analysis of autonomic computing policies.
2) The introduction of resource-definition policies within this framework.
3) A set of automated code generation techniques for the run-time synthesis of autonomic system connectors as web services, and an augmented version of the author's general-purpose autonomic manager [12][13] that uses these techniques to support the new policy type.

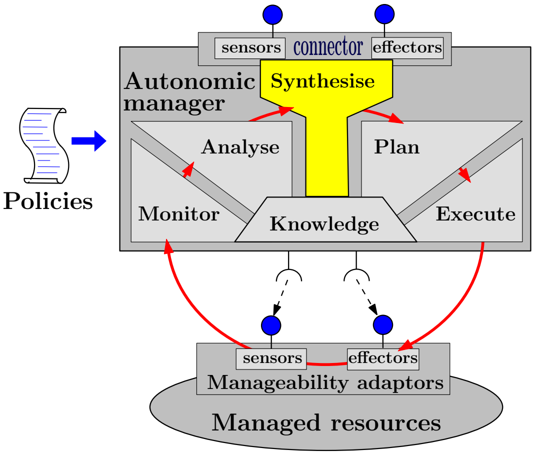


Fig. 1. Policy-based autonomic computing system whose control loop is augmented with a *connector synthesis step*.

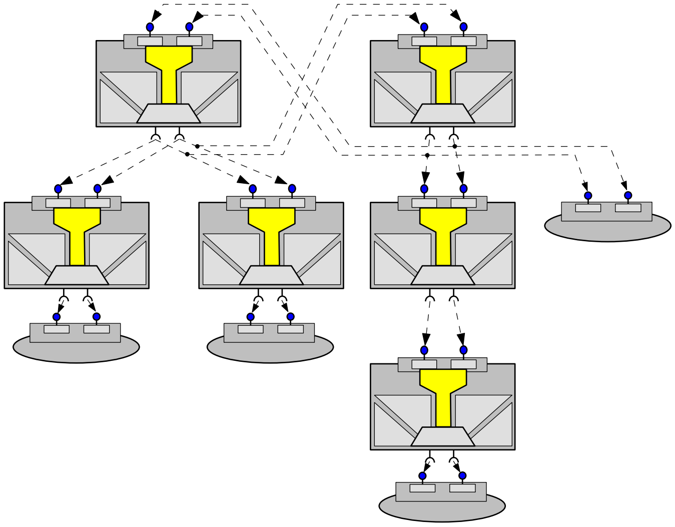


Fig. 2. Resource-definition policies support the run-time integration of autonomic systems into autonomic systems of systems.

4) A case study that illustrates the use of resource-definition policies to implement an autonomic data-centre application.

The remainder of the article starts with an overview of related work in Section II, followed by the description of our framework for the specification of autonomic computing policies in Section III. Section IV provides a formal introduction to resource-definition policies. Section V 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 VI. Finally, Section VII provides an overview of the problem addressed by our research work and the solution proposed in the article, a summary of the case study used to validate this solution, and a list of future work directions.

II. 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 [2] envisaged *autonomic elements* that "interact with other elements and with human programmers via their autonomic managers". More recently, this vision was adapted and extended successfully to specific application domains including aerospace [7] and "beyond 3G" networking [10]. However, these extensions represent work in progress.

The high-level architecture proposed in [7] is also 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 this approach are limited to those pre-planned when the autonomic elements are originally developed.

The ongoing research project whose agenda is described in [10] will employ *event services* to enable flexible interactions among *autonomic network management elements*. When this work will be completed, the functionality of the resulting framework will resemble that of the autonomic system connectors described in our article, but in the area of autonomic networking.

The work described in [9] proposes the use of hierarchical systems of event-driven sensors for monitoring and symptom recognition within autonomic systems. The use of dedicated policies to dynamically configure an instance of the architecture in [9] is somewhat similar to our use of resource-definition policies. However, unlike our approach that supports the automated, run-time synthesis of connectors comprising a combination of autonomic computing sensors and effectors, the framework in [9] focuses exclusively on sensors, and requires manual coding for the implementation of these sensors.

Fully-fledged instances of the data-centre autonomic systems comprising a pair of collaborating autonomic elements and a two-level hierarchy of autonomic elements are described in [8] and [11], respectively. The two successful domain-specific applications by research teams at IBM Research demonstrate the benefits of partitioning the autonomic functionality of large computing systems among multiple autonomic elements. However, the use of hard-coded autonomic manager interfaces limits the inter-operation of the autonomic elements of the applications in [8][11] to pre-defined types of interactions. Furthermore, the need to implement each of these interfaces upfront requires a significant amount of coding effort, and does not scale well to autonomic systems of systems comprising more than a few autonomic elements (the systems in [8] and [11] comprise two and three autonomic managers, respectively).

In contrast with all these approaches, the use of resource-definition policies makes possible the automated, runtime generation of the sensor-effector connectors of an autonomic element. Thus, the main advantage of our novel approach to implementing autonomic systems of systems resides in its unique ability to support the development of unforeseen applications by dynamically (re)defining the interfaces of existing autonomic systems. As an added benefit, the use of resource-definition policies eliminates completely the effort and costs associated with the manual implementation of these interfaces.

The architecture and functionality of the autonomic systems of systems built using resource-definition policies resemble those of intelligent multi-agent systems [14]. However, the standard approaches used for the development of multi-agent systems differ significantly from the approach proposed in this paper. Thus, the typical approach to multi-agent system development [14] 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 [15]. 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 [15], 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.

III. 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

$$P_{ij} = (\mathit{paramId}_{ij}, \mathit{ValueDomain}_{ij}, \mathit{type}_{ij}) \tag{3}$$

where $\mathit{paramId}_{ij}$ is a `string`-valued identifier used to distinguish the different parameters of a resource; $\mathit{ValueDomain}_{ij}$ is the set of possible values for P_{ij}; and $\mathit{type}_{ij} \in \{\texttt{ReadOnly}, \texttt{ReadWrite}\}$ 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.,

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

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.,

$$S = \bigtimes_{1 \leq i \leq n} \; \bigtimes_{\substack{1 \leq j \leq m_i \\ \mathit{type}_{ij} = \texttt{ReadOnly}}} \mathit{ValueDomain}_{ij} \tag{4}$$

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.,

$$C = \bigtimes_{1 \leq i \leq n} \; \bigtimes_{\substack{1 \leq j \leq m_i \\ \mathit{type}_{ij} = \texttt{ReadWrite}}} \mathit{ValueDomain}_{ij} \tag{5}$$

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 [12].

The standard types of autonomic policies described in [15][11][16] 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:

$$p_{\text{action}} : S \times C \nrightarrow C. \tag{6}$$

A graphical representation of an action policy is shown in Figure 3a. Note that an action policy can be implemented even when the autonomic manager has no knowledge about the behaviour of the managed resources, i.e., when $\mathrm{domain(f)} = \emptyset$ in eq. (1).

2) A *goal policy* partitions the state/configuration combinations for the system into desirable and undesirable:

$$p_{\text{goal}} : S \times C \rightarrow \{\texttt{true}, \texttt{false}\}, \tag{7}$$

with the autonomic manager requested to maintain the system in an operation area for which p_{goal} is `true`. Figure 3b shows a graphical representation of a goal policy within the theoretical framework introduced in this section.

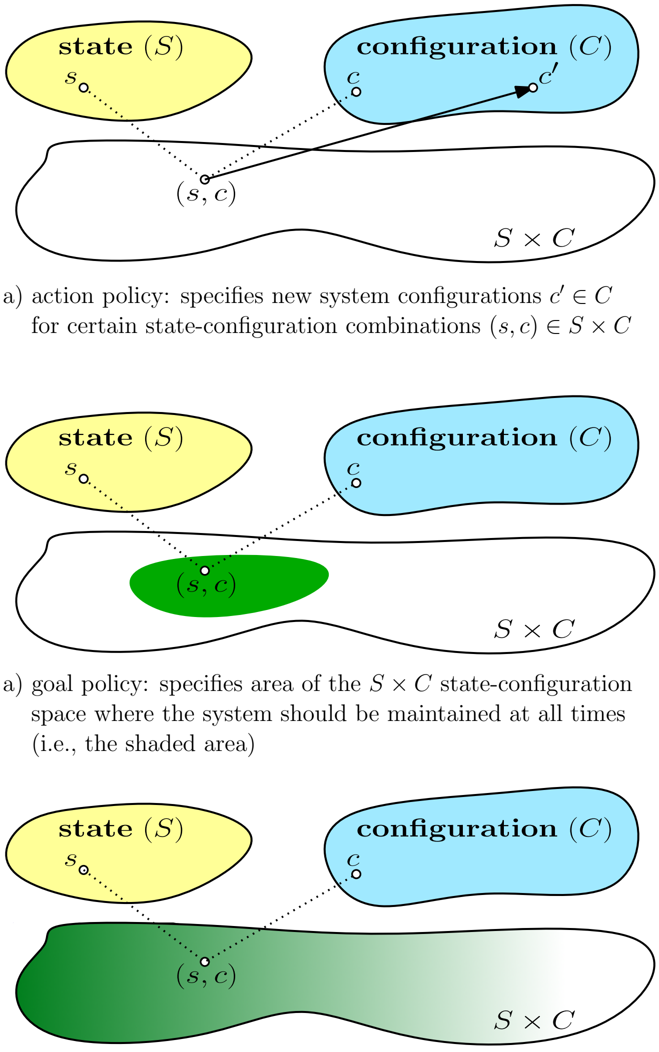
3) A *utility policy* (Figure 3c) associates a value with each state/configuration combination, and the autonomic manager should adjust the system configuration such as to maximise this value:

$$p_{\text{utility}} : S \times C \rightarrow \mathbb{R}. \tag{8}$$

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

$$K = (\mathit{ServerPool}, \mathit{Service}_1, \ldots, \mathit{Service}_n, f) \tag{9}$$

[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.



a) action policy: specifies new system configurations $c' \in C$ for certain state-configuration combinations $(s,c) \in S \times C$

a) goal policy: specifies area of the $S \times C$ state-configuration space where the system should be maintained at all times (i.e., the shaded area)

c) utility policy: specifies the utility of each state-configuration combination $(s,c) \in S \times C$ for the system—darker shading indicates higher utility

Fig. 3. Graphical representation of the standard types of autonomic computing policies

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

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

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 $c = (n_1, n_2, \ldots, n_N) \in C$:

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

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

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

We will describe these policies one at a time. First, 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 i, $1 \le i \le N$. The first of these conditions states that a service should be allocated servers only if all services of higher priority have already been allocated $\lceil \alpha w_i \rceil$ servers. The second condition states that a service should be allocated no server only if all n available servers have been 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. 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 4. More realistic utility functions u and matching behavioural models f from eq. (9) are described in [12].

IV. Resource-Definition Policies

A. 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

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

$$u : \mathsf{R}_+ \times \mathsf{R}_+ \to [0,1]$$

$$u(w,n) = \begin{cases} 0, & \text{if } n < (2-\alpha)w \\ \frac{n-(2-\alpha)w}{2(\alpha-1)w}, & \text{if } (2-\alpha)w \leq n \leq \alpha w \\ 1, & \text{if } n > \alpha w \end{cases}$$

$$\alpha \in (1,2)$$

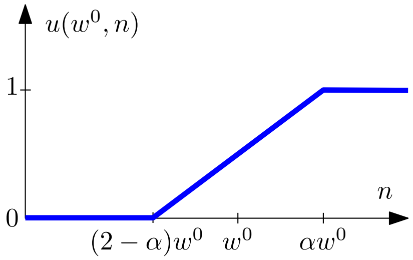


Fig. 4. Sample function u for Example 1—the graph shows u for a fixed value of its first argument, i.e., $w = w^0$

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, $\ldots$, E_q specify how the autonomic manager will calculate the values of the $q \geq 0$ `ReadOnly` parameters of R as functions of $(\mathbf{s}, \mathbf{c})$. Assuming that the value domain for the ith `ReadOnly` parameter of R, $1 \leq i \leq q$ is $\mathit{ValueDomain}_i^{\text{synth\_RO}}$, we have

$$E_i : S \times C \to \mathit{ValueDomain}_i^{\text{synth\_RO}}. \tag{16}$$

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

$$\begin{aligned} p_{\text{def}}(s,c) = ((&\texttt{"dataCentre"}, \\ &(\texttt{"id"}, \mathit{String}, \texttt{ReadOnly}) \\ &(\texttt{"maxUtility"}, \mathbb{R}, \texttt{ReadOnly}), \\ &(\texttt{"utility"}, \mathbb{R}, \texttt{ReadOnly})), \\ &\texttt{"dataCentre A"}, \\ &\max_{(x_1,x_2,\ldots,x_N)\in\mathbb{N}^N} \textstyle\sum_{w_i>0}^{1\leq i\leq N} p_i u(w_i, x_i), \\ &\textstyle\sum_{w_i>0}^{1\leq i\leq N} p_i u(w_i, n_i)) \end{aligned} \tag{17}$$

This policy requests the autonomic manager to synthesise a resource termed a `"dataCentre"`. This resource comprises 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 the utility policy (13). (The term $\epsilon \sum_{i=1}^{N} n_i$ from the policy definition is insignificant, and was not included in (17) for simplicity.) Exposing the system through this synthesised resource enables an external autonomic manager to monitor how close the data-centre is to achieving its maximum utility.

Note that the generic form of a resource-definition policy (14)-(15) allows the policy to request the autonomic manager to synthesise different types of resources for different state/configuration combinations of the system. While the preliminary use cases that we have studied so far can be handled using resource-definition policies in which the resource model R from (15) is fixed for all $(\mathbf{s}, \mathbf{c}) \in S \times C$, we envisage that this capability will be useful for more complex applications of resource-definition policies.

B. Synthesised Resources with `ReadWrite` *Parameters*

In this section we 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 to 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 $\{P_1^{\text{synth\_RW}}, P_2^{\text{synth\_RW}}, \ldots, P_r^{\text{synth\_RW}}\}$, the state space definition (4) is replaced by:

$$S = \left(\bigtimes_{1\leq i\leq n} \bigtimes_{\substack{1\leq j\leq m_i \\ \mathit{type}_{ij}=\texttt{ReadOnly}}} \mathit{ValueDomain}_{ij} \right) \times \left(\bigtimes_{1\leq i\leq r} \mathit{ValueDomain}_i^{\text{synth\_RW}} \right), \tag{18}$$

where $\mathit{ValueDomain}_i^{\text{synth\_RW}}$ represents the value domain of the ith synthesised resource parameter P_i^{synth}, $1 \leq i \leq r$. Given that the synthesised sensors involve setting the value of the connector parameters by the autonomic manager, the configuration space of the system can be redefined in an analogous way as:

$$C = \left(\bigtimes_{1\leq i\leq n} \bigtimes_{\substack{1\leq j\leq m_i \\ \mathit{type}_{ij}=\texttt{ReadWrite}}} \mathit{ValueDomain}_{ij} \right) \times \left(\bigtimes_{1\leq i\leq q} \mathit{ValueDomain}_i^{\text{synth\_RO}} \right). \tag{19}$$

Figure 5 illustrates graphically the way in which a policy-definition policy extends the state and configuration spaces of an autonomic system.

Example 3 Consider again our running example of an autonomic data-centre. The resource-definition policy in (17) can be extended to allow a peer data-centre (such as a data-centre

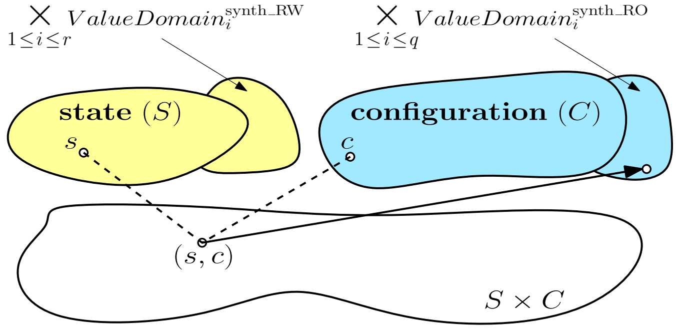

$\times_{1\leq i\leq r} ValueDomain_i^{\text{synth\_RW}}$
$\times_{1\leq i\leq q} ValueDomain_i^{\text{synth\_RO}}$
state (S)
s
configuration (C)
c
(s,c)
$S \times C$

Fig. 5. Resource-definition policies extend the state and configuration spaces of an autonomic system with the value domains of the synthesised connectors.

running the same set of services within the same security domain) to take advantage of any spare servers:

$$
\begin{aligned}
p'_{\text{def}}(s,c) = ((&\texttt{"dataCentre"},\\
&(\texttt{"id"}, \mathit{String}, \texttt{ReadOnly})\\
&(\texttt{"maxUtility"}, \mathbb{R}, \texttt{ReadOnly}),\\
&(\texttt{"utility"}, \mathbb{R}, \texttt{ReadOnly})),\\
&(\texttt{"nSpare"}, \mathbb{N}, \texttt{ReadOnly})),\\
&(\texttt{"peerRequest"}, \mathbb{N}^N, \texttt{ReadWrite})),\\
&\texttt{"dataCentre A"},\\
&\max_{(x_1,x_2,\ldots,x_N)\in\mathbb{N}^N} \textstyle\sum_{w_i>0}^{1\leq i\leq N} p_i u(w_i,x_i),\\
&\textstyle\sum_{w_i>0}^{1\leq i\leq N} p_i u(w_i,n_i),\ n-\sum_{i=1}^N n_i)
\end{aligned}
\tag{20}
$$

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 $1 \leq i \leq N$.

To illustrate how this is achieved, we will consider two data-centres that each implements the policy in (20), and which have access to each other's "dataCentre" resource as shown in the lower half of Figure 10. For simplicity, we will further assume that the data-centres are responsible for disjoint sets of services (i.e., there is no $1 \leq i \leq N$ 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 $\texttt{peerRequest}^r$ parameter of its peer, and how it should use its own $\texttt{peerRequest}^l$ 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. (18)); 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 (20) is given below:

$$
p'_{\text{utility}}(s,c) =
$$
$$
= \begin{cases}
-\infty, & \text{if } \sum\limits_{i=1}^N n_i > n \vee \sum\limits_{i=1}^N n_i^r > n^r\\[2ex]
\begin{array}{l}\sum\limits_{\substack{i\,=\,1\\ w_i\,>\,0}}^N p_i u(w_i, n_i + n_i^r) - \epsilon \sum\limits_{i=1}^N n_i -\\ -\lambda \sum\limits_{i=1}^N n_i^r + \mu \sum\limits_{\substack{i\,=\,1\\ n_i^l\,>\,0}}^N \min\Big(1, \frac{n_i}{n_i^l}\Big)\end{array} & \text{, otherwise}
\end{cases}
\tag{21}
$$

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 $\mu \sum_{n_i^l>0}^{1\leq i\leq N} \min(1, \frac{n_i}{n_i^l})$ 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-centre increases the system utility.

Finally, note that because the requests for remote servers and the allocation of such servers take place asynchronously, there is a risk that the parameter values used in policy (21) may be out of date.[2] However, this is not a problem, as the allocation of fewer or more remote servers than ideally required is never decreasing the utility value for 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 the utility expression.

V. Prototype Implementation

The *policy engine* (i.e., policy-based autonomic manager) introduced by the author in [13] was extended with the ability to handle the new type of autonomic computing policy. Implemented as a model-driven, service-oriented architecture with the characteristics presented in [12], the policy engine

[2] In practical scenarios that we investigated this happened very infrequently relative to the time required to solve the linear optimisation problem (21) automatically within the policy engine.

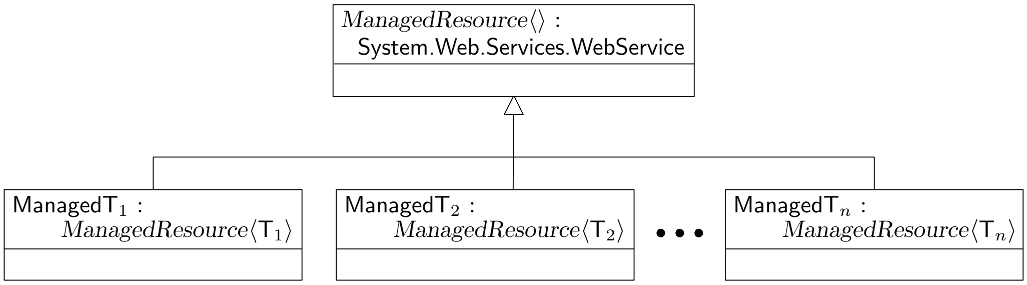

$\mathit{ManagedResource}\langle\rangle$: System.Web.Services.WebService
ManagedT$_1$: $\mathit{ManagedResource}\langle \mathsf{T}_1\rangle$
ManagedT$_2$: $\mathit{ManagedResource}\langle \mathsf{T}_2\rangle$
• • •
ManagedT$_n$: $\mathit{ManagedResource}\langle \mathsf{T}_n\rangle$

Fig. 6. Generic programming is used to implement the manageability adaptors for the system resources.

from [13] can manage IT resources whose model is supplied to the engine in a runtime configuration step. The IT resource models described by eq. (1) are represented as XML documents that are instances of a pre-defined meta-model encoded as an XML schema [12][13]. This choice was motivated by the availability of numerous off-the-shelf tools for the manipulation of XML documents and XML schemas—a characteristic largely lacking for the other technologies we considered. The policy engine is implemented as a .NET web service. In its handling of IT resources whose characteristics are unknown until runtime, the policy engine takes advantage of object-oriented (OO) technology features including:

- *polymorphism*—the ability to use instances of derived data types in place of instances of their base data types;
- *reflection*—an OO programming technique that allows the runtime discovery and creation of objects based on their metadata [17].
- *generic programming*—the ability to write code in terms of data types unknown until runtime [18].

The manageability adaptors from Figure 1 are implemented by the framework in [13] as web services that specialise a generic, abstract web service *ManagedResource*$\langle\rangle$. 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, $\forall 1 \leq i \leq n$. Second, the manageability adaptor ManagedT$_i$ for resources of type T_i is implemented by specialising our generic *ManagedResource*$\langle\rangle$ adaptor, i.e., ManagedT$_i$: *ManagedResource*$\langle\mathsf{T}_i\rangle$. This process is illustrated in Figure 6 and described in detail in [13].

Adding support for the implementation of the resource-definition policy in (14)–(15) required the extension of the policy engine described above with the following functionality:

1) The 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) The automated creation of an instance of T. Reflection is employed to create an instance of T for the lifespan of the resource-definition policy. The `ReadOnly` fields of this object are updated by the policy engine using the expressions E_1, E_2, …, E_q from eq. (16); the updates are carried out whenever the object is accessed by an external entity.
3) The automated generation of a manageability adaptor web service ManagedT : *ManagedResource*$\langle$T$\rangle$. The web methods provided by this *autonomic system connector* 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. The synthesised connector is available as soon as the engine completes its handling of the resource-definition policy. The URL of the connector has the same prefix as that of the policy engine, and ends in "`Managed`$resID$`.asmx`", where $resID$ is the resource identifier specified in the resource-definition policy. For instance, if the policy engine URL is `http://www.abc.org/PolicyEngine.asmx`, the synthesised connector for a resource of type dataCentre will be accessible at the address `http://www.abc.org/ManagedDataCentre.asmx`.

Example 4 Returning to our running example of an autonomic data-centre, the class diagram in Figure 7 depicts the manageability adaptors in place after policy (20) 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*$\langle\rangle$ (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 (20). Notice the one-to-one mapping between the fields of these classes and the parameters of their associated resources (described in Examples 1 and 3).

VI. Application Development

System-of-systems application development with resource-definition policies involves supplying such policies to existing

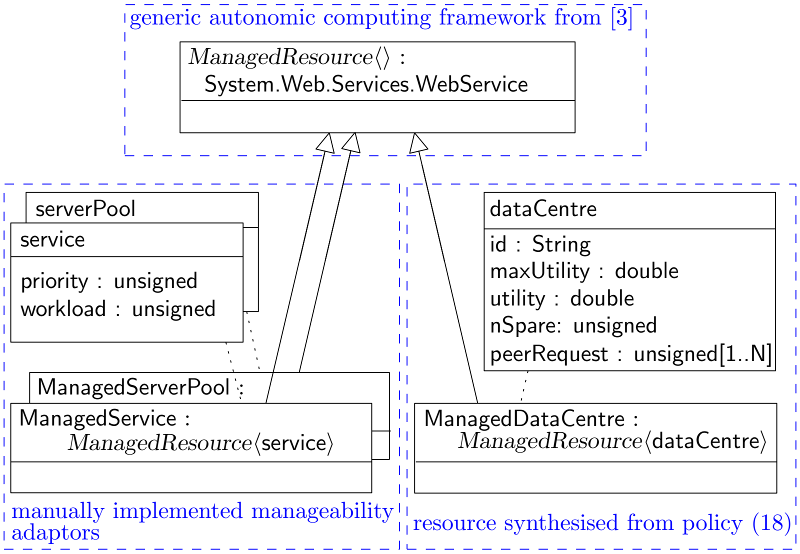

generic autonomic computing framework from [3]
ManagedResource⟨⟩ : System.Web.Services.WebService
serverPool
service
priority : unsigned
workload : unsigned
ManagedServerPool :
ManagedService : ManagedResource⟨service⟩
manually implemented manageability adaptors
dataCentre
id : String
maxUtility : double
utility : double
nSpare: unsigned
peerRequest : unsigned[1..N]
ManagedDataCentre : ManagedResource⟨dataCentre⟩
resource synthesised from policy (18)

Fig. 7. Class diagram for Example 4

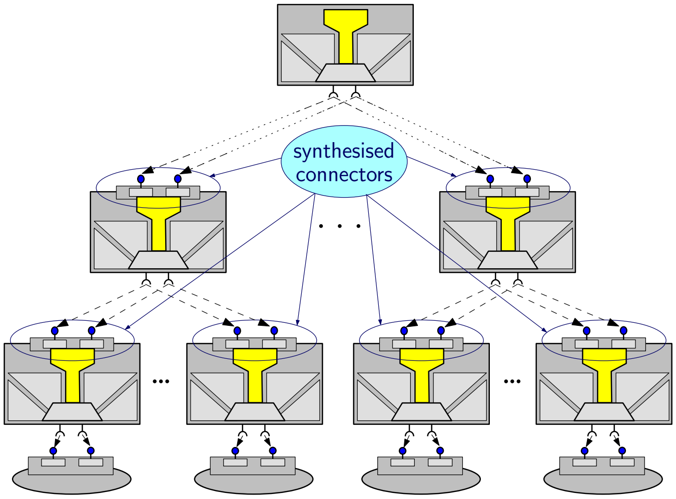


Fig. 8. Hierarchical system-of-systems development with resource-definition policies

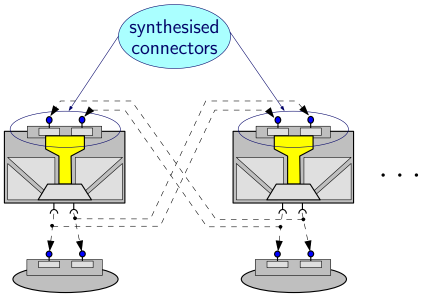


Fig. 9. Collaborating autonomic systems that inter-operate by means of run-time synthesised connectors

autonomic systems whose autonomic managers support the new policy type. Hierarchical systems of systems can be built by setting a higher-level autonomic manager to monitor and/or control subordinate autonomic systems through synthesised connectors that abstract out the low-level details of the systems they expose to the outside world. As illustrated in Figure 8, the hierarchy of autonomic systems can comprise any number of levels.

Alternatively, the original autonomic systems can be configured to collaborate with each other by means of the synthesised resource sensors and effectors. In this scenario, the sensors are used to expose the global state of each system and the effectors are employed to provide *services* to peer autonomic systems as envisaged in the seminal paper of Kephart and Chess [2]. The approach is illustrated in Figure 9.

Hybrid applications comprising both types of interactions mentioned above are also possible, as illustrated by the following example.

Example 5 The policy engine from Section V 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 10). The policies implemented by the autonomic managers local to each data-centre are policies (20)–(21) 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 resource simulators from [12], and implemented the dashboard resource as an ASP.NET web page provided with a manageability adaptor built manually as described in Section V and in [13]. Separate series of experiments for 20-day simulated time periods we 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

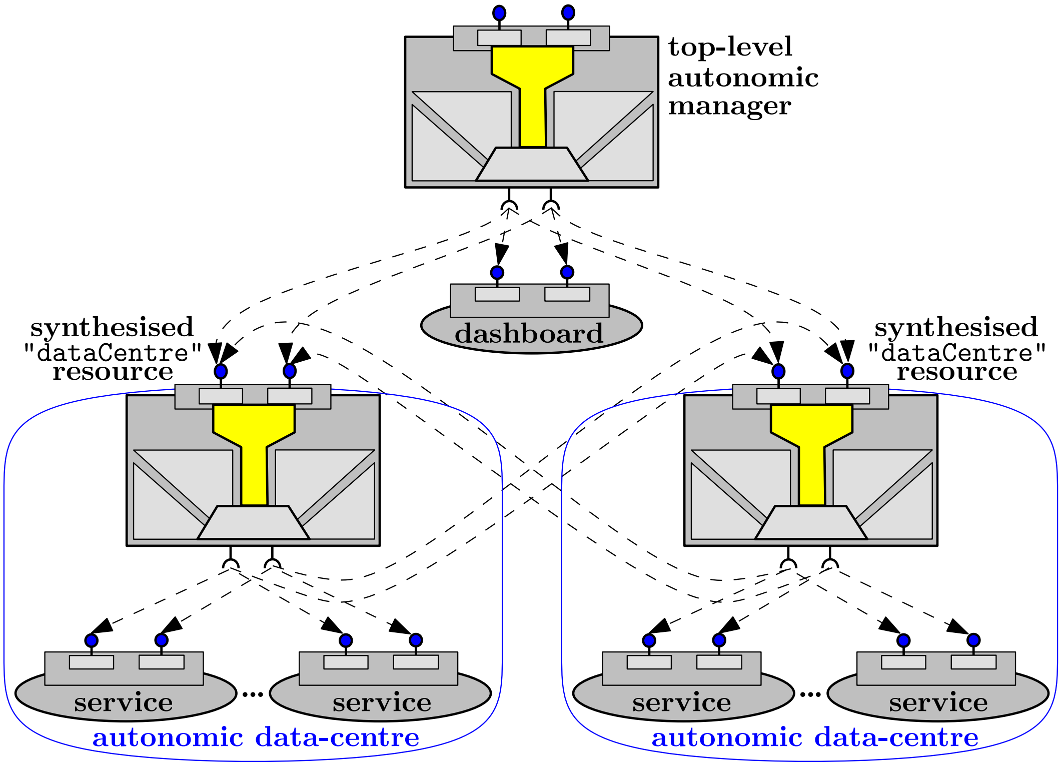


Fig. 10. Autonomic system of systems

data-centres were allowed to discover each other, and thus to collaborate through implementing policy (21). Figure 11 depicts typical snapshots of the dashboard for both scenarios and for one of the data-centres; the same simulated service 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, thus allowing data-centres to utilise each other's spare servers.

VII. Conclusion and Future Work

Many of today's computing systems are built through the integration of components that are systems in their own right: they operate autonomously and follow local goals in addition to contributing to the realisation of the global objectives of the overall system. The research results presented in this article enable the addition of self-managing capabilities to this class of systems. This is achieved through the run-time generation of *autonomic system connectors* that can be used to dynamically assemble autonomic systems of systems out of a set of individual autonomic systems.

We introduced a policy specification framework for autonomic computing, and used it to formally define the existing types of autonomic computing policies from [15][11][16] and a new type of policy that we termed a resource-definition policy. We described the semantics of resource-definition policies, and explained how they can be used to dynamically generate connectors that enable the integration of multiple autonomic systems into autonomic systems of systems. This represents a significant step towards addressing a major challenge of autonomic computing, namely the seamless integration of multiple autonomic elements into larger self-managing computing systems [2].

Additionally, the article presented how resource-definition policies can be implemented by augmenting the reference autonomic computing loop with a *connector synthesis* step, and described a prototype implementation of an autonomic manager that comprises this additional step.

To validate the proposed approach to developing autonomic systems of systems, we used the prototype autonomic manager in a case study that was presented as a running example in Sections III to VI. The case study involved the adaptive allocation of data-centre servers to services with different priorities and variable workloads. The system of systems considered by our case study included a couple of simulated autonomic data-centres that employed utility-function policies to dynamically partition their physical servers among local services. Within this autonomic system of systems, resource-definition policies were used successfully to synthesise a `"dataCentre"` resource fulfilling two distinct purposes:

1) Collaboration between the autonomic data-centres. Each data-centre exposed its number of unused, spare servers through a `ReadOnly` parameter of the synthesised resource, and the peer data-centre could set a `ReadWrite` parameter of the same resource to request that some of its services are run on these spare servers. The effectiveness of the technique was demonstrated by the improved utility achieved by the simulated data-centres when this collaboration was enabled, compared to the scenario when the data-centres operated in isolation from each other.

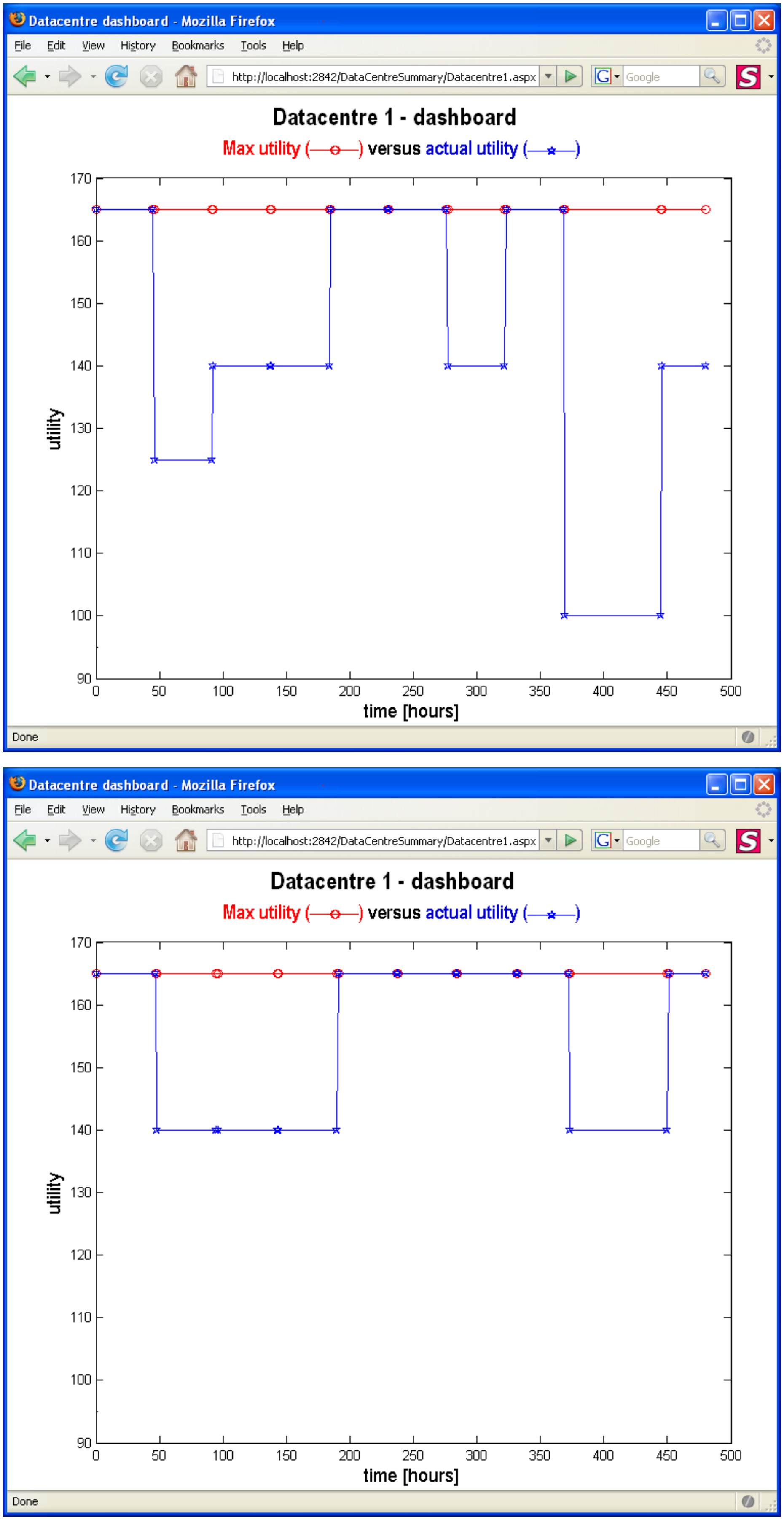


Fig. 11. Dashboard for isolated data-centre (top) and for identical data-centre operating as part of the autonomic system of systems from Figure 10 (bottom).

2) High-level reporting of data-centre utility. Two ReadOnly parameters of the synthesised resource were used to reflect the instantaneous data-centre utility and maximum (or ideal) utility attainable, respectively. A top-level instance of the prototype autonomic manager was employed to monitor this information and, through suitable action policies, to forward it to a web-based dashboard for real-time plotting.

The two uses of resource-definition policies show that both federations of collaborative autonomic elements and hierarchical autonomic systems can be implemented successfully using the tool-supported development technique proposed in the article.

The work presented in the article is part of an ongoing project aimed at devising novel approaches for the effective development of autonomic systems and systems of systems. Related future objectives of this project include extending the policy specification framework with techniques for the analysis of policy properties such as state/configuration space coverage and conflict freeness, and applying resource-definition policies to additional, real-world case studies from the area of data-centre resource management.

Acknowledgement

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

References

[1] R. Calinescu, “Resource-definition policies for autonomic computing,” in *Proceedings of the 5th International Conference on Autonomic and Autonomous Systems (ICAS 2009)*, 2009, pp. 111–116.

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

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

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

[5] Carnegie Mellon Software Engineering Institute, *Ultra-Large-Scale Systems. The Software Challenge of the Future*. Carnegie Mellon University, 2006.

[6] G. Goth, “Ultralarge systems: Redefining software engineering?” *IEEE Software*, vol. 25, no. 3, pp. 91–94, May/June 2008.

[7] 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*, 2007, pp. 250–257.

[8] J. O. Kephart, H. Chan, R. Das, D. W. Levine, G. Tesauro, F. L. R. III, and C. Lefurgy, “Coordinating multiple autonomic managers to achieve specified power-performance tradeoffs,” in *Proceedings Fourth International Conference on Autonomic Computing (ICAC’07)*, 2007.

[9] J. Li, P. Martin, W. Powley, K. Wilson, and C. Craddock, “A sensor-based approach to symptom recognition for autonomic systems,” in *Proceedings of the 5th International Conference on Autonomic and Autonomous Systems (ICAS 2009)*, 2009, pp. 45–50.

[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*, 2008, pp. 174–183.

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

[12] R. Calinescu, “General-purpose autonomic computing,” in *Autonomic Computing and Networking*, M. Denko *et al.*, Eds. Springer, 2009, pp. 3–20.

[13] ——, “Implementation of a generic autonomic framework,” in *Proc. 4th Intl. Conf. Autonomic and Autonomous Systems*, D. Greenwood *et al.*, Eds., March 2008, pp. 124–129.

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

[15] 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.

[16] S. White *et al.*, “An architectural approach to autonomic computing,” in *Proc. 1st IEEE Intl. Conf. Autonomic Computing*. IEEE Computer Society, 2004, pp. 2–9.

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

[18] R. Garcia *et al.*, “A comparative study of language support for generic programming,” *ACM SIGPLAN Notices*, vol. 38, no. 11, pp. 115–134, November 2003.