Software Engineering Techniques for the Development of Systems of Systems

Radu Calinescu and Marta Kwiatkowska

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

Abstract. This paper investigates how existing software engineering techniques can be employed, adapted and integrated for the development of systems of systems. Starting from existing system-of-systems (SoS) studies, we identify computing paradigms and techniques that have the potential to help address the challenges associated with SoS development, and propose an SoS development framework that combines these techniques in a novel way. This framework addresses the development of a class of IT systems of systems characterised by high variability in the types of interactions between their component systems, and by relatively small numbers of such interactions. We describe how the framework supports the dynamic, automated generation of the system interfaces required to achieve these interactions, and present a case study illustrating the development of a data-centre SoS using the new framework.

1 Introduction

The functionality and flexibility underpinning today's applications in areas ranging from transportation and healthcare to aerospace and defence can no longer be provided by a monolithic information system, however complex this might be. Instead, the required capabilities can only be achieved through employing collections of collaborative, heterogeneous and autonomously-operating systems—or *systems of systems*.

The crucial importance of many systems of systems and the high rates of late delivery, over-spending and failure associated with their development have prompted the initiation of research programmes for the investigation of this new class of systems [18, 24, 38, 41] and its extensions [15, 35, 42]. The results of this research provide valuable insights into the distinguishing features of systems of systems [3, 5, 28, 36], the challenges posed by their unprecedented size, diversity, variability, complexity, unforeseen interactions and emergent behaviour [15, 22, 27, 37], and some of the high-level principles and practices to be employed in their development [6, 26, 27, 36].

Most importantly for the progress of the field, broad agreement has been reached on the main features that set systems of systems apart from traditional, monolithic systems:

– The components of a system of systems (SoS) possess a level of operational autonomy that allows them to pursue their own, local objectives, independently and in addition to contributing to the global SoS objective(s) [5, 27].
– The components of a system of systems are often developed, procured and managed independently [26, 36].
– SoS components may belong to multiple open and evolving systems of systems that they could join and leave dynamically [5, 6, 18, 37].
– The behaviour of a system of systems cannot be fully predicted from the behaviour of its component systems [3, 26, 27, 37].

Additionally, an SoS subclass that typifies key information systems of the future—i.e., the so-called *large-scale complex (IT) systems* [35] or *ultra-large-scale systems* [15, 42]—is characterised by incomplete and continually changing requirements and components, and by regarding failures as normal events.

The challenges associated with the development of systems of systems are tremendous. They include the need to ensure the interoperability of a vastly diverse range of components [6, 36], to convey global objectives to SoS components in meaningful ways [15, 18], to achieve these objectives predictably and dependably in a dynamically changing environment [35, 42], and to attain high levels of SoS longevity [15, 37].

These major advances in the understanding of systems of systems laid the foundation for essential work to identify high-level principles and practices governing their development [6, 15, 26, 27, 36]. Our paper takes this work further by investigating for the first time ways in which existing software engineering techniques can contribute to the development of IT systems of systems. Thus, computer science paradigms including formal analysis and verification, model-driven and component-based development, service-oriented architectures and policy-based autonomic computing are analysed for compliance with the recommended SoS development principles, and for their ability to help address SoS engineering challenges. The results of this analysis are presented in Section 2.

In Section 3, we describe a new approach to integrating these techniques within a framework that extends the authors' previous work on quantitative analysis and verification [31, 33, 34], model-driven development [12] and self-* computing [7, 8, 10] to the realm of systems of systems. Our framework supports the development of IT systems of systems by enabling the online, automated generation of the interfaces that the system components of an SoS employ to inter-operate with the other systems within the same SoS. To achieve this, the systems to be integrated within an SoS are augmented with an *autonomic computing policy engine* that exposes their global parameters through runtime-generated interfaces defined by user-specified *policies*. This idea was originally introduced in [8], and in this paper we provide a formal description of the types of policies that can be used for this purpose and of how they can be realised in practice.

Its ability to dynamically generate new interfaces for the system components of an SoS makes our framework particularly suitable for the development of systems of systems characterised by high variability in the types of interactions

between SoS component systems, and by relatively small numbers of such interactions. Therefore, the capabilities of the framework are illustrated using a case study that involves the development of a data-centre SoS with these characteristics, and which is based on a series of real-world use cases that one of the authors encountered in his previous work on a commercial system for data-centre resource management [9, 11]. To add to the readability of the paper, this case study is presented as a running example spread over the next three sections.

A prototype version of the framework was implemented as an extension of our generic autonomic computing framework from [13]. Section 4 describes the novel features of this prototype implementation, and the combination of dynamic code generation, model-based development and dynamic reconfiguration techniques employed to support these features. Finally, Section 5 describes various types of IT systems of systems that can be developed using the framework, and Section 6 summarises our results and discusses a number of future research directions.

2 Software Engineering Techniques for SoS Development

This section examines existing software engineering paradigms and techniques that could help tackle some of the challenges associated with the development of systems of systems, and which are therefore likely to be part of the SoS development frameworks of the future. A summary of this analysis is presented in Table 1.

1. Service-oriented architectures (SOA) SoS development involves the integration and secure interoperation of vastly diverse technical systems [3, 5, 6, 15, 18, 26, 37]. Thanks to their platform independence, loose coupling and support for security, SOA solutions [46] represent strong candidates for implementing new computer systems or front-ends to legacy systems that need to be integrated into an SoS.

2. Policy-based autonomic computing Ecosystems, cities and economies are often pointed out as examples of effective systems of systems. A common characteristic of all these systems of systems is the way in which their global objectives are specified through high-level incentives, rewards and penalties rather than by setting concrete, precise targets [15, 35, 36]. Thus, the behaviour of ecosystems is governed by laws of nature. The development and everyday life of cities are subject to common or civil laws and regulations. The evolution of economies is guided by taxation policies. If these successful real-world examples are to be followed, techniques will be required that can convey the global objectives of systems of systems as *high-level policies* to their autonomous components. (Policy-based) autonomic computing addresses the development of systems that can manage themselves based on a set of high-level policies [30], and therefore represents an ideal paradigm for developing the computer-system components of an SoS.

3. Formal analysis and verification A major concern of systems of systems is their ability to achieve an overall objective in predictable and dependable ways, through the collaboration of component systems with different (and potentially conflicting) local goals [15, 35, 39]. Formal analysis and verification, and in particular model checking [16], quantitative model checking [34] and quantitative analysis and verification [31, 33], comprise a range of techniques that could be used or adapted for use in the verification of SoS policies, and ultimately for SoS dependability management and assurance.

4. Model-driven development and code generation The open, evolving nature of systems of systems allows their components to join and leave dynamically [35, 39]. Having SoS components collaborate with peer systems whose characteristics are often unknown until runtime is a major challenge. A combination of model-driven development and runtime code generation in which a dynamically acquired model of a peer system is used to generate the necessary interfaces and logic for collaborating with this peer system [13] represents a promising approach to addressing this challenge.

5. Component-based development SoS engineering requires the integration of existing and future commercial, open-source and proprietary systems, and component-based development provides techniques that can help achieve this goal [1, 2, 17].

6. Dynamic reconfiguration Systems of systems are required to adapt continually to changes in their environment, structure and objectives [6, 26]. Recent advances in the study of dynamically reconfigurable software and hardware [19, 23] provide promising approaches for the development of the computer systems to be incorporated into the systems of systems of the future.

7. Online machine learning The levels of self-management that SoS components must achieve in impossible to anticipate circumstances are significantly beyond what can be pre-programmed into a computer system [22, 35, 42]. The online use of techniques specific to machine learning [4] is therefore likely to play a major role in the development of computer-based SoS components.

8. Resource discovery In the era of mobile computing, SoS components are expected to actively seek partner systems and establish collaborations with them, thus joining (and leaving) loosely-coupled federations of systems on a regular basis [5, 15, 35]. The rich spectrum of resource[1] discovery techniques employed by today's distributed (e.g., grid- and web-based) computer systems [43] can be used as a basis for the development of techniques to support these capabilities.

[1] The terms *resource* and *component* are used interchangeably in the paper.

Table 1. Software engineering techniques that can help address SoS challenges

	SoS challenges							
Techniques and paradigms	interoperability, security	dependability (assurance)	collaboration	global-objective specification	predictability	adaptibility	longevity	flexibility
Service-oriented architectures	✓							
Policy-based autonomic computing				✓				✓
Formal analysis and verification		✓			✓			
Model-driven development/code generation						✓		
Component-based development			✓					
Dynamic reconfiguration						✓	✓	✓
Online machine learning						✓	✓	
Resource discovery			✓					✓

3 A Framework for System-of-Systems Development

3.1 Overview

Our approach to integrating the software engineering techniques analysed in the previous section involves the use of a reconfigurable policy engine with the structure in Figure 1.[2] The **SOA implementation** of this policy engine as a web service (described in Section 4) takes a model of a system and a set of policies, and ensures that the system achieves the objectives specified by these policies through adapting continually to changes in its environment.

When a running instance of the policy engine is **dynamically reconfigured** by means of a system model, its *runtime* ***code generator*** employs **model-driven development** techniques to generate *manageability adaptor proxies*, i.e., software components whose *monitor* and *control* interfaces allow the engine to read and to modify the parameters of the system components, respectively. This functionality is described in our previous work [8, 13]. Additionally, the policy engine in [8, 13] supports all types of policies that are standard in **policy-based autonomic computing** [44, 45], and uses **resource discovery** tech-

[2] The use of these techniques is emphasised in **bold text** in the framework overview.

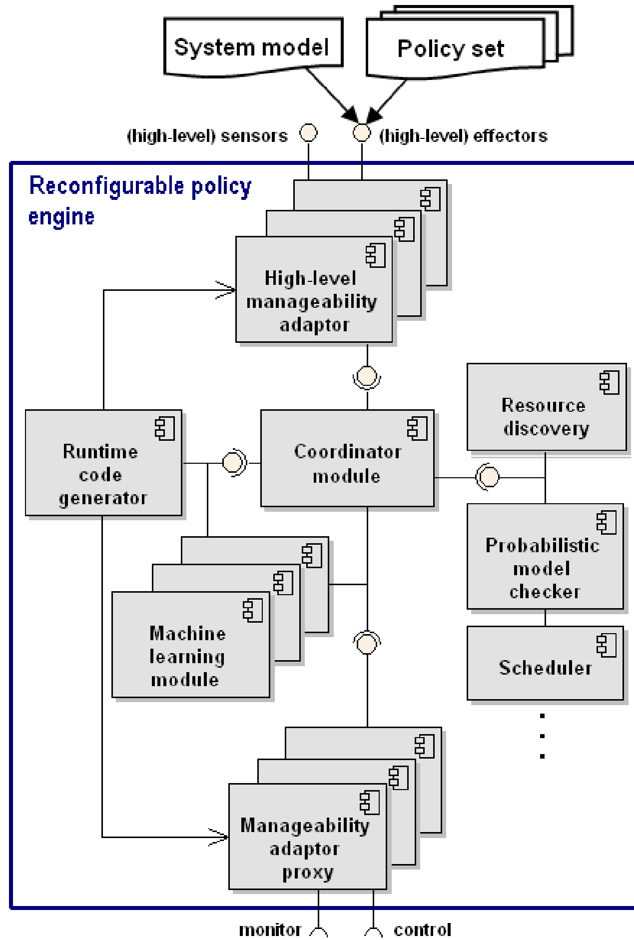

Fig. 1. Reconfigurable policy engine

niques to identify the system components to which these policies need to be applied. **Component-based development** techniques introduced in Section 3.3 are employed to integrate multiple autonomic IT systems into an autonomic system of systems.

Furthermore, we recently extended our policy engine with the ability to employ online **formal analysis and verification** techniques for the implementation of a powerful class of autonomic computing policies [10]. Finally, we are in the process of augmenting our framework with **online machine learning** capabilities by integrating it with the work presented in [20].

The remainder of this section provides brief presentations of how each software engineering technique was or, in the case of online machine learning, will be integrated into the SoS development framework. References to full descriptions

of these integrations are provided for those interested in learning more about the framework.

1. Service-oriented architectures (SOA) For the reasons explained in Section 2, the components of the framework were implemented as web services. This applies to the autonomic computing policy engine from Figure 1, as well as to the software adaptors between the policy engine and the existing, legacy components that the engine is managing.

Because systems of systems comprise components that are often unknown until run time, the policy engine was provided with the capability to handle such components through run-time reconfiguration. This required the extensive use of techniques available only in an object-oriented programming environment, including reflection, polymorphism, automated generation of web-service proxies, and generic programming. Based on these requirements, J2EE and .NET were selected as candidate platforms for the framework, with .NET being eventually preferred due to its better handling of dynamic proxy generation and its slightly easier-to-use implementation of reflection.

A detailed description of the SOA implementation of the framework, and of several case studies involving the development of monolithic autonomic systems using the framework are available in [13].

2. Policy-based autonomic computing All standard types of autonomic computing policies [29, 44, 45] are supported by the framework. A formal description and simple examples of these policies are provided in Section 3.2. For real-world applications of each policy type within the scope of the framework, the reader should refer to [8].

3. Formal analysis and verification One type of autonomic computing policy supported by the framework is termed a *utility-function policy.* This policy specifies a multi-objective optimisation to be performed through continually adapting the configuration of a system to its workload and environment. Examples of utility-function policies taken from [10] are:

– optimise the parameters of a dynamically power-managed disk drive to achieve user-specified trade-offs between the response time and the power consumption of the disk drive, under variable workload;
– optimise the allocation of data-centre servers to clusters of different priorities and variable workloads, subject to using the fewest possible servers and to ensuring user-prescribed levels of cluster availability, in the presence of data-centre component failures and repairs.

To achieve such policies, the system model used to configure the policy engine includes a precise mathematical description of the system behaviour, and formally-specified quantitative properties derived from the multi-objective functions are exhaustively analysed at runtime to identify optimal system configurations. This analysis is performed using PRISM—a free, open-source tool for the

formal modelling and analysis of real-time and stochastic systems [31, 32] that an extensive, independent survey [25] ranked as the most effective tool for the quantitative analysis of large system models.

For the latter of the above-mentioned policies, for instance, PRISM was used to calculate the "probability that a cluster can handle its workload successfully" (i.e., the expected *cluster availability*) for each possible configuration of the cluster. This calculation was performed automatically whenever there was a significant change in the cluster workload, and the configuration that achieved the user-prescribed availability using the fewest servers was chosen.

A full description of the use of quantitative analysis within our policy engine is available from [10].

4. Model-driven development and code generation Given that many systems of systems are characterised by dynamically changing components, the policy engine at the core of our framework was designed to handle IT components whose attributes are unknown until run time. This unique capability necessitates the run-time use of model-driven and automated code generation techniques within the policy engine. Thus, two software artefacts are generated automatically based on the system model supplied to a running instance of the policy engine: (a) data types (i.e., classes) for the new types of IT components; and (b) proxies for the adaptor web services associated with the new component types. This code generation process is discussed in detail in [13].

5. Component-based development In this paper, we define formally a new type of autonomic computing policy that supports adaptive component-based development. Originally suggested in [8], this *resource-definition policy* specifies how the *high-level sensors* and *high-level effectors* interfaces of the policy engine should expose the system under its control as a single component, thus enabling its integration into a system of systems.

Figure 2 depicts the generic architecture of an SoS built around an extension of the policy engine from [13] that supports resource-definition policies. Each of the top-level *autonomic-enabled components* 1 to N in this architecture is a system managed by an appropriately configured instance of the policy engine. At the SoS level, the policy engine instances expose the state and configuration of their systems, employ resource discovery to identify peer SoS components, and collaborate with these. At the local level, the policy engines organise heterogeneous collections of components into a single system. These collections can comprise legacy components whose interfaces are accessed through *manageability adaptors* [13] and autonomic-enabled components (i.e., new systems that expose *sensor* and *effector* interfaces permitting their direct management by the policy engine, or other instances of the top-level autonomic-enabled components in Figure 2).

The theoretical foundation, implementation and applications of resource-definition policies are presented in Sections 3.3, 4 and 5, respectively.

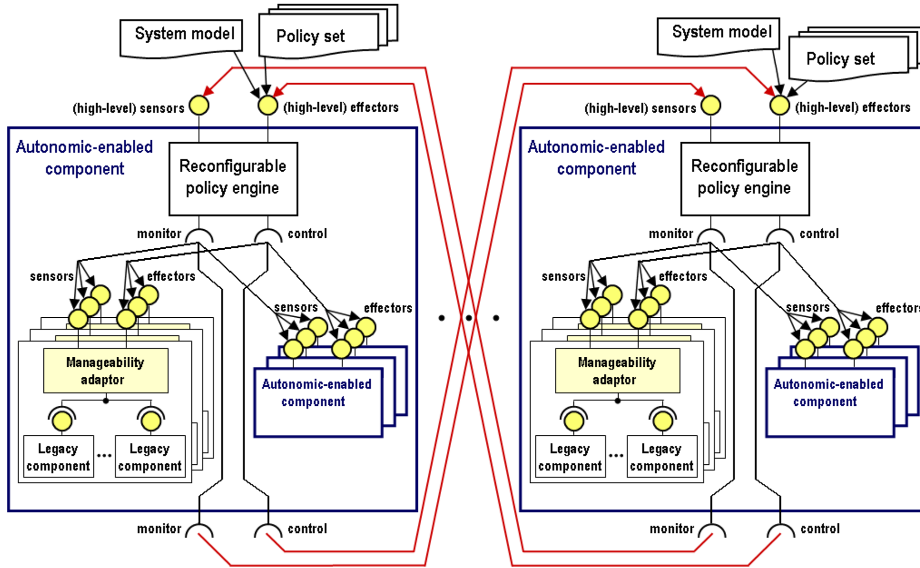

Fig. 2. System-of-systems architecture

6. Dynamic reconfiguration When supplied with a new system model, the policy engine within our framework becomes capable of managing the previously unknown IT components whose characteristics are described in this model. This *dynamic reconfiguration* of a running instance of the engine involves the automated synthesis of component-specific software artefacts for each IT component specified in the system model, as already mentioned earlier in this section and described in depth in [8, 13].

7. Online machine learning In a future version of the policy engine, online machine learning will be employed to continually improve the accuracy of the formal behavioural model that the engine uses to implement autonomic computing policies. The approach that we are working on involves learning the actual values of the model parameters from the observed behaviour of the SoS components. This extension of the framework builds on the research described in [20] and is being carried out jointly with its authors.

8. Resource discovery The resource discovery mechanism employed by the framework has two parts. First, the framework includes a resource discovery web service that policy engine instances can query to identify the locations of the components they are required to manage—namely the URLs of the web-service adaptors that enable the engine to inter-operate with existing SoS components. Additionally, each such adaptor comprises a `SupportedResource` web method that policy engine instances call to discover the type of resource that the adaptor is exposing, as explained in detail in [8].

We will start the detailed presentation of the framework in the next section, where we formally define the system model from Figure 1 and the standard types of autonomic computing policies supported by the policy engine. This will be followed in Section 3.3 by the specification of the resource-definition policies that underlie the component-based development of IT systems of systems using our framework.

3.2 System Model and Standard Autonomic Computing Policies

The system model used to configure the policy engine from Figure 1 is a tuple that defines the $n \geq 1$ resources of the system and their behaviour:

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

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 policy engine 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[3]

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

[3] 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.

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 [8, 10].

The standard types of autonomic policies described in [29, 44, 45] 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 domain(f) $= \emptyset$ in (1).
2. A *goal policy* partitions the state/configuration combinations for the system into desirable and undesirable:

$$p_{\mathrm{goal}} : S \times C \to \{\mathtt{true}, \mathtt{false}\}, \tag{7}$$

   with the policy engine 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 policy engine should adjust the system configuration such as to maximise this value:

$$p_{\mathrm{utility}} : S \times C \to \mathbb{R}. \tag{8}$$

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 model (1) for this system can be expressed as a tuple

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

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

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

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 $\mathbf{s} = (n, p_1, w_1, p_2, w_2, \ldots, p_N, w_N) \in S$ and configuration $\mathbf{c} = (n_1, n_2, \ldots, n_N) \in C$:

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

$$p_{\text{goal}}(\mathbf{s}, \mathbf{c}) = \forall 1 \le i \le N \bullet (n_i > 0 \implies (\forall 1 \le j \le N \bullet p_j > p_i \implies n_j = \lceil \alpha w_j \rceil)) \tag{12}$$

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

We will describe each of these policies in turn. 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 service i, $1 \le i \le N$, is allocated servers only if all services of higher priority have already been allocated $\lceil \alpha w_i \rceil$ servers.

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

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

$\alpha \in (1, 2)$

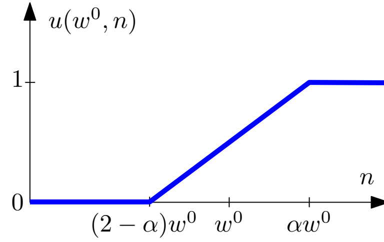

Fig. 3. Sample function u for Example 1 (the graph shows u for a fixed value w^0 of its first argument)

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 3; a more realistic u and a matching behavioural model f from (9) are described in [8].

3.3 Resource-definition policies for runtime interface generation

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 policy engine is required to synthesise, and the expressions E_1, E_2, ..., E_q specify how the engine will calculate the values of the $q \geq 0$ `ReadOnly` parameters of R as functions of $(\mathbf{s}, \mathbf{c})$. Assuming that $q > 0$ and the value domain for the ith `ReadOnly` parameter of R, $1 \leq i \leq q$ 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

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

This policy requests the synthesis of 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 (16) for simplicity.) Exposing the system through this synthesised resource enables an external policy engine 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 users to request the policy engine 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.

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 policy engine. 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 an individual system from the SoS architecture in Figure 2 is to take them into account within the set of policies implemented by the policy engine associated with that system. 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:

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

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

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:

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

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 $1 \leq i \leq N$.

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 5 from one of the next sections of the paper. 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 the superscripts ^{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 $\mathbf{s} = (n, p_1, w_1, p_2, w_2, \ldots, p_N, w_N, n^r, n_1^l, n_2^l, \ldots, n_N^l)$ (cf. (17)); and the system configuration has the form $\mathbf{c} = (n_1, n_2, \ldots, n_N, n_1^r, n_2^r, \ldots, n_N^r)$. The utility policy to use alongside policy (18) is given below:

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

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 $\mu \sum_{n_i^l>0}^{1\leq i\leq N} \min(1, \frac{n_i}{n_i^l})$ 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 pa-

rameter values used in policy (19) may be out of date.[4] 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.

4 Prototype Implementation

The *policy engine* introduced 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 from [13] can manage IT resources whose model is supplied to the engine in a runtime configuration step. The IT resource models 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, and takes advantage of object-oriented technology features such as polymorphism, reflection[5] and generics[6] in its handling of IT resources whose characteristics are unknown until runtime.

The manageability adaptors from Figure 2 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. Second, the manageability adaptor $\mathsf{ManagedT}_i$ for resources of type T_i is implemented by specialising our generic *ManagedResource*$\langle\rangle$ adaptor, i.e., $\mathsf{ManagedT}_i$: *ManagedResource*$\langle\mathsf{T}_i\rangle$. This process is described in [13].

Adding support for the implementation of the resource-definition policy in (14)–(15) involved extending the policy engine described above 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.

[4] In practical scenarios that we investigated this happened very infrequently relative to the time required to solve the linear optimisation problem (19) automatically within the policy engine.

[5] Reflection is an object-oriented programming technique that allows the runtime discovery and creation of objects based on their metadata [40].

[6] Generics or generic programming represents an object-oriented programming technique enabling code to be written in terms of data types unknown until runtime [21].

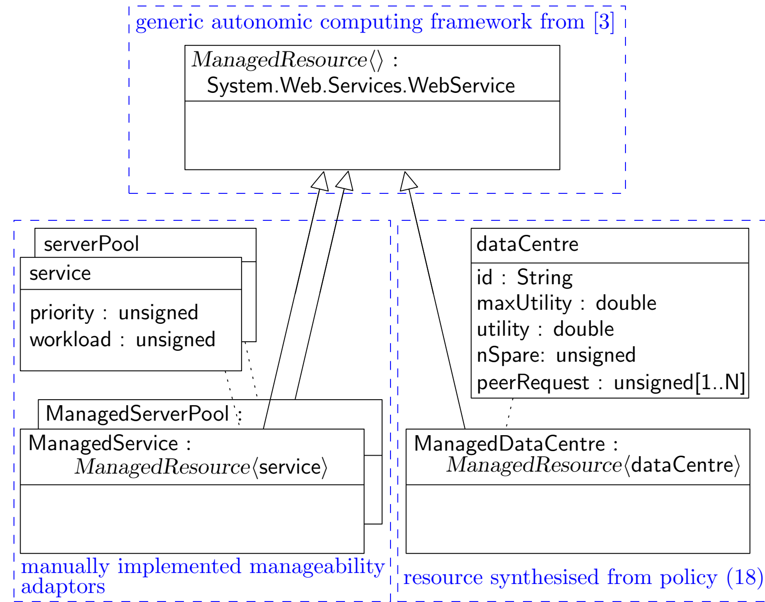

Fig. 4. Class diagram for Example 4

2. 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 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 policy engines) 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 IT resource is available as soon as the engine completes its handling of the resource-definition policy.

Example 4 Returning to our running example of an autonomic data-centre, the class diagram in Figure 4 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). 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).

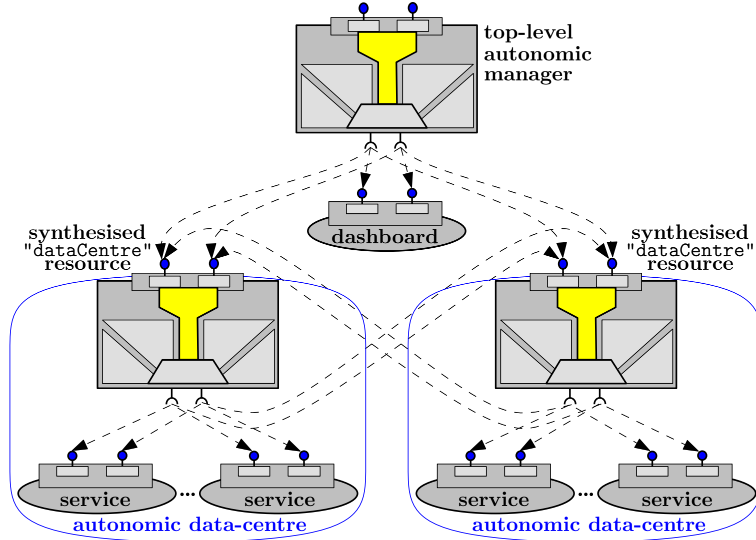

Fig. 5. Autonomic system of systems for Example 5

5 System of Systems Development

System-of-systems application development using the framework described in Sections 3 and 4 involves supplying resource-definition policies to existing autonomic systems whose policy engines support the new policy type. Hierarchical systems of systems can then be built by setting a higher-level policy engine 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 and effectors. 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 4 was used to simulate an autonomic system of systems comprising the pair of autonomic data-centres described in

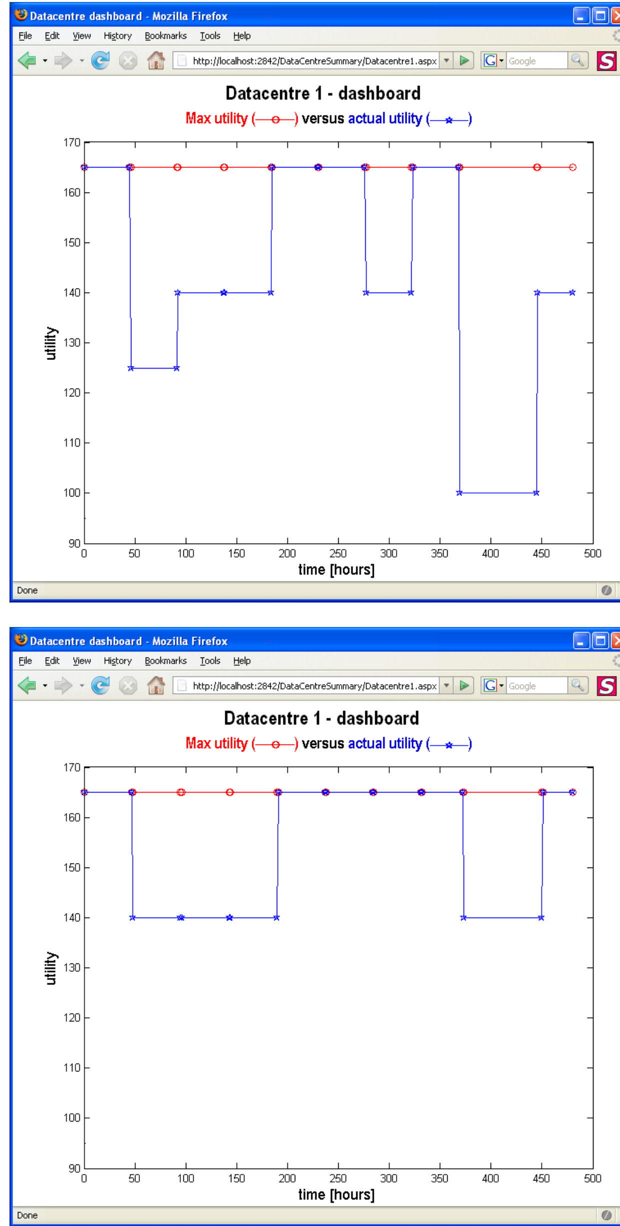

Fig. 6. Dashboard for isolated data-centre (top) and for identical data-centre operating as part of the autonomic system of systems from Figure 5 (bottom)

Example 3, and a top-level policy engine that monitors and summarises their performance using a dashboard resource (Figure 5). The policies implemented by the policy engines local to each data-centre are policies (18)–(19) from Ex-

ample 3. The top-level policy engine 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 [8], and implemented the dashboard resource as an ASP.NET web page provided with a manageability adaptor built manually as described in Section 4 and in [13]. 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 6 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.

6 Conclusion

A common finding of SoS studies is that existing techniques and tools are unable to address the whole spectrum of challenges associated with the development of systems of systems [3, 15, 22, 27]. Notwithstanding the disparity between what can be achieved using current approaches and these challenges, the SoS development frameworks of the future are likely to incorporate some of today's software engineering techniques or adapted, enhanced variants of them. This paper examined techniques that are candidates for this role, including formal analysis and verification, model-driven development, service-oriented architectures, component-based development and policy-based autonomic computing. Having first identified the SoS challenge(s) that each such technique can help address, we then proposed a new approach to combining these techniques into a framework for the development of a class of IT systems of systems.

The administrators of an SoS developed using our framework can specify at run time how the SoS components inter-operate with each other. To handle runtime changes in this specification, our framework employs a combination of model-based and online code generation techniques to automatically build and deploy the interfaces necessary to support new types of inter-operation among SoS components. This capability is particularly useful given that SoS components often belong to open, evolving systems of systems that they could join and leave dynamically [5, 6, 18, 37]. Additionally, our framework is suitable for the development of IT systems of systems that need to adapt their inter-component interactions to changes in the SoS global objectives and/or context.

The automation of time-demanding processes such as the run-time synthesis of the interfaces between SoS component systems and the reconfiguration of the policy engine represents a key benefit of the framework. The use of an earlier

version of the framework to develop monolithic autonomic systems yielded a tenfold reduction in development time [14], and preliminary experiments with its extended prototype presented in Section 4 indicate that significant reductions are also possible in the case of systems of systems.

In the current version of our SoS development framework, the dynamically generated interfaces among SoS component systems can be used only in asynchronous mode, and involve periodical polling by the reconfigurable policy engines within the SoS. For this reason, the systems of systems whose development is currently supported by the framework are those characterised by asynchronous and relatively infrequent interactions between SoS component systems. A temporary workaround for this limitation is to mix dynamically generated component interfaces suffering from this constraint with statically implemented interfaces. For a long-term solution, we are investigating the possibility to use a notification mechanism within our reconfigurable policy engine in order to support the runtime specification of synchronous SoS component interfaces.

Additional areas of future work include the validation of the proposed framework within new application domains, the development of SoS-specific online machine learning techniques, the synthesis of high-level SoS policies from specifications, and the design of metrics for the assessment of global SoS effectiveness.

Acknowledgement

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

References

1. Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.
2. Farhad Arbab. Abstract behavior types: a foundation model for components and their composition. *Science of Computer Programming*, 55(1–3):3–52, March 2005.
3. Y. Bar-Yam et al. The characteristics and emerging behaviors of system-of-systems. Complex physical, biological and social systems project report, New England Complex Systems Institute, January 2004. http://necsi.org/education/oneweek/winter05/NECSISoS.pdf.
4. Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2007.
5. John Boardman and Brian Sauser. System of systems – the meaning of *of*. In *Proceedings of the 2006 IEEE/SMC International Conference on System of Systems Engineering*, pages 118–123, 2006.
6. Lisa Brownsword, David Fisher, Ed Morris, James Smith, and Patrick Kirwan. System-of-systems navigator: An approach for managing system-of-systems interoperability. Technical Report CMU/SEI-2006-TN-019, Carnegie Mellon Software Engineering Institute, April 2006. http://www.sei.cmu.edu/pub/documents/06.reports/pdf/06tn019.pdf.

7. R. Calinescu. Towards a generic autonomic architecture for legacy resource management. In K. Elleithy, editor, *Innovations and Advanced Techniques in Systems, Computing Sciences and Software Engineering*, pages 410–415. Springer, 2008.
8. R. Calinescu. General-purpose autonomic computing. In M. Denko et al., editors, *Autonomic Computing and Networking*, pages 3–20. Springer, 2009.
9. R. Calinescu and J.M.D. Hill. System providing methodology for policy-based resource allocation, July 2005. US Patent Application 20050149940.
10. R. Calinescu and M. Kwiatkowska. Using quantitative analysis to implement autonomic IT systems. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, May 2009.
11. Radu Calinescu. Challenges and best practices in policy-based autonomic architectures. In *Proceedings of the 3rd IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC 2007)*, pages 65–74, Columbia, Maryland, USA, 2007.
12. Radu Calinescu. Model-driven autonomic architecture. In *Proceedings of the 4th IEEE International Conference on Autonomic Computing*, Jacksonville, Florida, June 2007.
13. Radu Calinescu. Implementation of a generic autonomic framework. In D. Greenwood et al., editors, *Proceedings 4th International Conference on Autonomic and Autonomous Systems (ICAS'08)*, pages 124–129. IEEE Computer Society Press, March 2008.
14. Radu Calinescu and Marta Kwiatkowska. CADS*: Computer-aided development of self-* systems. In Marsha Chechik and Martin Wirsing, editors, *Fundamental Approaches to Software Engineering (FASE 2009)*, volume 5503 of *Lecture Notes in Computer Science*, pages 421–424. Springer, March 2009.
15. Carnegie Mellon Software Engineering Institute. *Ultra-Large-Scale Systems. The Software Challenge of the Future.* Carnegie Mellon University, 2006. http://www.sei.cmu.edu/uls/files/ULS_Book2006.pdf.
16. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.
17. Ivica Crnkovic and Magnus Larsson, editors. *Building Reliable Component-Based Software Systems*. Artech House Publishers, 2002.
18. William A. Crossley. System of Sytems: An Introduction of Purdue University Schools of Engineering's Signature Area. In *Proceedings of the Engineering Systems Symposium*, 2004. http://esd.mit.edu/symposium/pdfs/papers/crossley.pdf.
19. Tarek El-Ghazawi, Esam El-Araby, Miaoqing Huang, Kris Gaj, Volodymyr Kindratenko, and Duncan Buell. The promise of high-performance reconfigurable computing. *Computer*, 41(2):69–76, February 2008.
20. Ilenia Epifani, Carlo Ghezzi, Raffaela Mirandola, and Giordano Tamburrelli. Model evolution by runtime adaptation. In *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pages 111–121, May 2009.
21. R. Garcia et al. A comparative study of language support for generic programming. *ACM SIGPLAN Notices*, 38(11):115–134, November 2003.
22. Greg Goth. Ultralarge systems: Redefining software engineering? *IEEE Software*, 25(3):91–94, May/June 2008.
23. Markus Hannebauer. *Autonomous Dynamic Reconfiguration in Multi-agent Systems*. Springer, 2002.
24. Integration of Software-Intensive Systems (ISIS) Initiative: Addressing System-of-Systems Interoperability. http://www.sei.cmu.edu/isis.

25. 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 4899 of *LNCS*, pages 69–85. Springer, 2008.
26. Jeremy M. Kaplan. A new conceptual framework for net-centric, enterprise-wide, system-of-systems engineering. Defense & Technology Papers 30, US Center for Technology and National Security Policy, July 2006. http://www.ndu.edu/ctnsp/Def_Tech/DTP%2030%20A%20New%20Conceptual %20Framework.pdf.
27. Charles Keating. Research foundations for system of systems engineering. In *2005 IEEE International Conference on Systems, Man and Cybernetics*, volume 3, pages 2720–2725, 2005.
28. Charles Keating, Ralph Rogers, Resit Unal, David Dryer, Andres Sousa-Poza, Robert Safford, William Peterson, and Ghaith Rabadi. System of systems engineering. *Engineering Management Journal*, 15(3):36–45, September 2003.
29. 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.
30. Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer Journal*, 36(1):41–50, January 2003.
31. M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic model checking in practice: Case studies with PRISM. *ACM SIGMETRICS Performance Evaluation Review*, 32(4):16–21, 2005.
32. M. Kwiatkowska, G. Norman, and D. Parker. Quantitative analysis with the probabilistic model checker PRISM. *Electronic Notes in Theoretical Computer Science*, 153(2):5–31, 2005.
33. Marta Kwiatkowska. Quantitative verification: Models, techniques and tools. In *Proc. 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE*, pages 449–458. ACM Press, September 2007.
34. Marta Kwiatkowska, Gethin Norman, and David Parker. 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 (Tutorial Volume)*, pages 220–270. Springer, 2007.
35. LSCITS Consortium. Large-Scale Complex Information Technology Systems Initiative. http://www.lscits.org.
36. Mark W. Maier. Architecting principles for systems-of-systems. *Systems Engineering*, 1(4):267–284, February 1999.
37. Abe Meilich. System of systems engineering (SoSE) and architecture challenges in a net centric environment. In *Proceedings of the 2006 IEEE/SMC International Conference on System of Systems Engineering*, pages 1–5, 2006.
38. US National Centers for Systems of Systems Engineering (NCSOSE). http://www.eng.odu.edu/ncsose/.
39. Steven W. Popper, Steven C. Bankes, Robert Callaway, and Daniel DeLaurentis. System of systems symposium: Report on a summer conversation. In *Proceedings of the 1st System of Systems Symposium*, 2004. http://www.potomacinstitute.org/academiccen/SoS%20Summer%20Conversation %20report.pdf.
40. Jonathan M. Sobel and Daniel P. Friedman. An introduction to reflection-oriented programming. In *In Proceedings of Reflection96*, 1996.
41. US System of Systems Engineering Center of Excellence (SoSECE). http://www.sosece.org.

42. US Industry/University Collaborative Research Center for Ultra-Large-Scale Software-Intensive Systems (ULSSIS). http://ulssis.cs.virginia.edu.
43. Koen Vanthournout, Geert Deconinck, and Ronnie Belmans. A taxonomy for resource discovery. *Personal and Ubiquitous Computing*, 9(2):81–89, March 2005.
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.