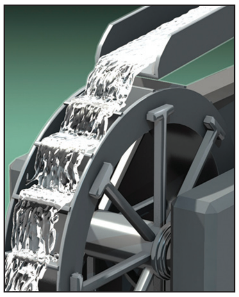

Programming Directives for Elastic Computing

Schahram Dustdar • *Vienna University of Technology*

Yike Guo and Rui Han • *Imperial College London*

Benjamin Satzger and Hong-Linh Truong • *Vienna University of Technology*

Enabling and controlling the elasticity of cloud computing applications is challenging. The developer must deal with daunting tasks using low-level code to implement strategies trading off costs versus quality of service. Programming directives can substantially reduce this overhead by delegating control of elasticity to middleware systems while allowing developers to focus on defining suitable strategies.

Programming directives are popular in parallel programming languages and frameworks such as Open Multiprocessing (OpenMP)[1] and High Performance Fortran (HPF),[2] where they let developers exploit and control the parallelism separating programming logic and computational behavior. Existing programming directives mainly control the number of processors used for a particular code region and the data distribution among them. Some directives also steer application deployment and execution using "program annotations" introduced in general-purpose programming languages (for example, Java Annotations, Python Decorators, and CLI attributes). However, such annotations are mainly for documentation, interface modification, and class overriding. Annotations for configuration, such as in the Spring framework (http://static.springsource.org/spring/docs/3.0.0.M3/reference/html/ch04s11.html) usually only allow for selecting suitable components and performing simple (auxiliary) configuration tasks.

Elasticity in computing isn't limited to resources;[3] cost and quality are also important dimensions we should consider. No existing programming directives address these criteria, which are important in cloud computing environments. While tools and domain-specific languages exist for deploying and configuring cloud applications,[4] they're designed for application submission and deployment with precise configuration and execution targets. They don't include elasticity constraints.

To address these issues, we can apply programming directive principles to managing the intrinsic elasticity of cloud computing scenarios in which it's essential to control resources, cost, and quality constraints. Our approach lets developers separate program logic from control of computing environments. Directive paradigms for elasticity have several key principles, and we can apply a set of basic primitive constructs to any elastic computing environment, such as clouds. We designed our primitives such that developers can incorporate them into mainstream programming languages, letting them control application elasticity. Our directives leverage runtime systems' ability to reduce complex management tasks that developers must deal with in the code, helping to optimize resource use under cost constraints while still achieving the desired quality. Furthermore, developers can change elasticity behavior after software development without touching the actual source code.

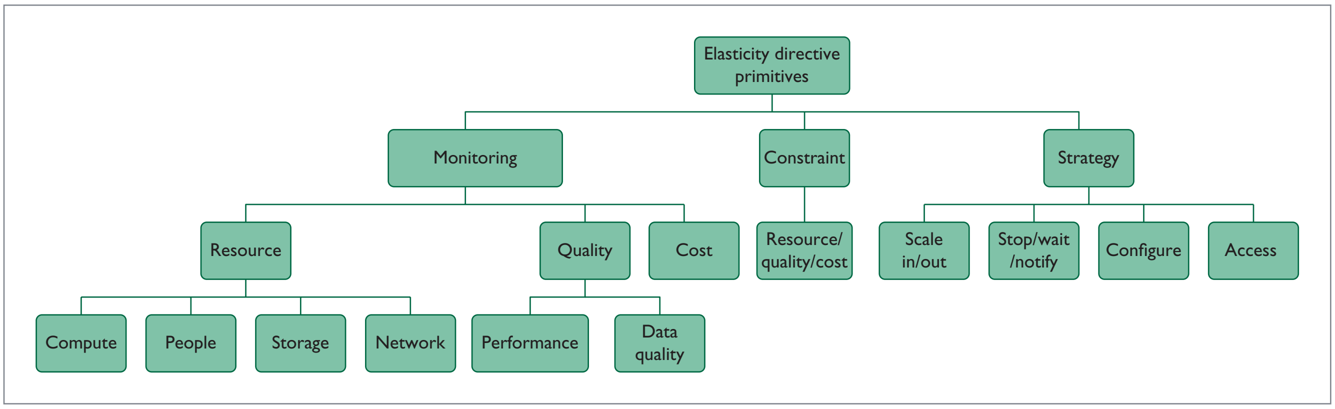

Figure 1. Classes of directive primitives.

Programming Directives for Elasticity

Programming directives for elasticity must let developers specify runtime properties related to resources, cost, and quality:

- *Monitoring* checks a computation's current status, determining the resources used, costs accrued, and the quality being maintained. This property should also check the computing environment's system status, including to monitor spot prices and overall resource provisioning costs.
- *Constraints* specify required conditions on resources, cost, and quality. They enable trade-offs between quality and cost associated with resources used in programs.
- *Strategies* let developers express possible actions that can control program behaviors based on monitored data and constraints.

We categorize programming directives into three classes based on the aforementioned properties — that is, `MONITORING`, `CONSTRAINT`, and `STRATEGY`. Within each class, developers can monitor and specify properties using predefined directives and runtime functions, as well as user-defined functions. Figure 1 describes subclasses of elasticity primitives. The hierarchical view of primitive classes covers possible monitoring information relevant for elasticity, constraints, and strategies for diverse resources, quality aspects including performance, and the quality of data and cost.

Monitoring primitives let us gather information related to resources (compute, storage, network, and human), quality, and cost. Elasticity in compute and storage resources is examined elsewhere, but few have addressed related principles for people or networks; exceptions include the concept of humans as programmable units[5] or software-defined networks (SDNs).[6] Developers and consumers must monitor several dimensions of quality as regards these resources, such as performance and data quality, as well as the costs for using resources under different expected and delivered quality levels.

Constraint directives let developers define conditions or states that the program must maintain. They further let developers specify trade-offs between opposing cost and quality attributes. These directives often trigger strategies to execute. Determining whether a program can meet defined conditions usually depends on data from the monitoring directives.

Strategy directives let developers influence the computing environment and execute applications. They execute logic that results in modified elasticity properties. A strategy manager (either the program itself or an external program execution management system) can carry out strategies explicitly or as a response to a change in a constraint's value. Exemplary strategies can influence the environment (for example, `Scale in/out`, `Configure`) or the application's execution (such as `Stop`, `Wait`, or `Notify`).

Based on these ideas, we're developing the Simple-Yet-Beautiful Language (SYBL), which specifies possible directives and runtime functions for managing elasticity in cloud-based applications. Describing SYBL in detail is beyond this article's scope, but we explain basic principles regarding its syntax and semantics.

Runtime and User-Defined Functions

As with existing programming directive systems, SYBL defines a set of runtime functions that can obtain and control the properties of computing environments in which developers deploy and execute applications:

- `balance([time])` checks a program's cost balance at a given time. For example, we can determine the cost per time interval as `cost=balance(now-interval)-balance(now)`.
- `set/get_env([property_name])` sets or obtains application and execution environment properties, such as the bid price defined for compute resources or the location where the application executes. As an example, we would obtain the default bid for a compute

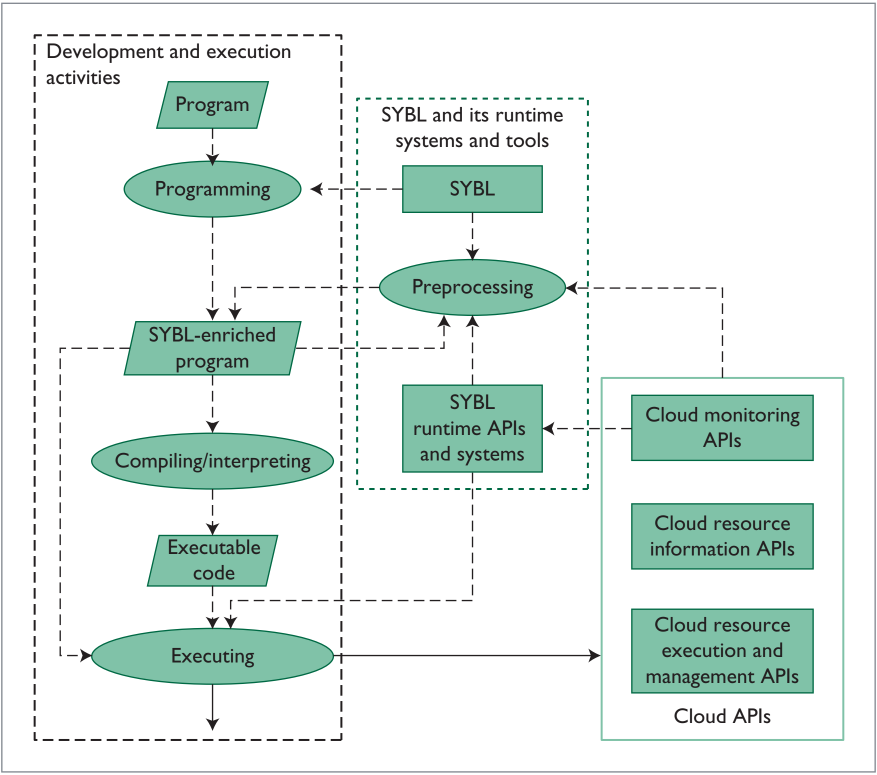
Development and execution activities
Program
Programming
SYBL-enriched program
Compiling/interpreting
Executable code
Executing
SYBL and its runtime systems and tools
SYBL
Preprocessing
SYBL runtime APIs and systems
Cloud monitoring APIs
Cloud resource information APIs
Cloud resource execution and management APIs
Cloud APIs

Figure 2. Development and runtime for SYBL.

resource as `"resourcebid=get_env("COMPUTE_BID")`.

- `runscript([file])` executes external scripts that can, for instance, implement user-defined functions.

Runtime functions are implemented and provided by SYBL runtime systems (as we describe later) and are used in the program directives we discuss next.

Directives

SYBL's core is a set of directives (see Figure 1) that developers can use separately or in combination. Abstractly, SYBL directives begin with `#SYBL`, followed by directive class names (that is, `MONITORING`, `CONSTRAINT`, and `STRATEGY`) and different directive clauses. These clauses let us use runtime and user-defined functions together with other variables, functions, and clauses. Table A in the Web appendix at http://doi.ieeecomputersociety.org/10.1109/MIC.2012.99 shows some examples of directives. To support directives, we can use source code preprocessing techniques or interpreter/compiler techniques together with SYBL runtime libraries.

Development and Runtime Systems

Generally, we can apply primitives that define elasticity to both general-purpose programming languages and system-configuration ones. Developers insert elasticity directives into programs to control how they use computational and financial resources and achieve desired quality. For system configuration, we can use elasticity directives to specify how to configure systems in an elastic environment.

For general-purpose programming languages such as Java, SCALA, C++, C#, and Python, we can implement SYBL directives using language-specific annotation support (for example, Java Annotation, C# Attribute Declaration, or Python Decorators). SYBL runtime systems can use existing cloud frameworks and APIs — such as CloudFoundry (www.cloudfoundry.com), JClouds (www.jclouds.org), Boto (http://docs.pythonboto.org), and OpenStack (www.openstack.org) — to implement provider-independent monitoring and adaptation features.

To control system elasticity during configuration and deployment activities, developers can use SYBL to generate different configuration strategies based on their specifications. For example, given a deployment configuration specified in the Topology and Orchestration Specification for Cloud Applications (TOSCA),[7] the developer can specify elasticity conditions that enrich the TOSCA-based configuration with new configuration plans.

For directives that are specified in programming and configuration languages, we need tools that can transform them into a set of system-specific runtime APIs. Such APIs will depend on the programming languages in which the directives are embedded. In this sense, runtime APIs can come from compiler runtime systems, OSs, and middleware. Figure 2 describes relationships between SYBL runtime systems, cloud APIs, SYBL-enriched programs, and programming tasks. By (pre)processing SYBL directives in programs, SYBL tools will enrich programs with SYBL runtime APIs and (specific) cloud APIs. When a SYBL-enriched program is executed, SYBL runtime functions and cloud APIs are invoked to monitor and manage the resources, cost, and quality associated with the program.

Illustrative Examples

Let's look at two examples that illustrate how developers can use SYBL to control the elasticity of cloud-based applications.

High-Level Elasticity Control: SYBL Combined with JClouds

The source code snippet in Figure 3 demonstrates how we can use JClouds

```
1 ComputeServiceContext context =
2    new ComputeServiceContextFactory().createContext("terremark", user, password);
3
4 ComputeService computeService = context.getComputeService();
5
6 Template template = computeService.templateBuilder().osFamily(CENTOS).build();
7
8 NodeSet nodes = computeService.createNodesInGroup("group01", 2, template);
```

Figure 3. Using JClouds. This Java source code snippet demonstrates how we would use JClouds without the Simple-Yet-Beautiful Language (SYBL) to access two virtual machine instances.

(without SYBL) to access two computing resources — that is, two virtual machine (VM) instances. The JClouds library provides a unified Java API for accessing different vendors' cloud services. Note that APIs besides JClouds can be used for accessing cloud services instead.

Lines 1 and 2 create a context that defines the binding to a certain cloud service provider (Terremark in this example). Line 6 shows how to create templates that define the types of VMs we want to instantiate. This example defines only the OS family, but much more detailed templates specifying CPU, RAM, and so on are possible. Line 8 starts two VM instances according to the template and adds them to a group.

Although JClouds is a big improvement as regards vendor lock-in compared to using vendor-specific APIs, in this particular example, users must still be concerned with which cloud provider they employ and how to achieve elasticity. Using SYBL in combination with Java we can create a set of annotations that lets users transparently control elasticity. SYBL can derive from directive type, variable context, and variable type that the SYBL directive refers to a strategy for which a cloud resource should be injected into the subsequent variable `computeService`. The file `template.xml` contains a template specification for the required VM. The simple `CHEAPEST` strategy advises the SYBL reasoner to inject an object that links to the cloud provider with the cheapest offer for the specified VM:

```
@SYBLLang ("STRATEGY
  EXECUTE(ResourceStrategy.
  CHEAPEST, \"template.xml\")")
ComputeService
  computeService;

NodeSet nodes =
  computeService.
  createNodesInGroup
  ("groupname", 2);
```

The file `template.xml` contains information about the resource to be injected. SYBL defines the respective XML schema:

```
<resource type="compute">
<osfamily>centos</osfamily>
</resource>
```

Another simplified example based not only on strategy but also on monitoring and constraint directives is to postpone remote execution of a computationally expensive task until the price for spot instances is below a certain threshold:

```
@SYBLLang ("MONITORING
  SPOT_PRICE=get_env(AVG_SPOT_
  PRICE)")
@SYBLLang ("CONSTRAINT LOW_
  SPOT_PRICE =(SPOT_PRICE < 1.2)")
@SYBLLang ("STRAGEGY EXECUTE
  (ExecutionStrategy.WAIT_
  UNTIL, LOW_SPOT_PRICE)")
Solution s = solveOnSpot
  Instance(OptimizationProblem p);
```

In the aforementioned examples, the developer defines the elasticity using SYBL directives specified in annotations without worrying about cloud-specific APIs.

Integration of SYBL into TOSCA

In addition to programming languages, we can also use SYBL with configuration languages. Consider TOSCA,[4] an emerging framework for specifying cloud components' dependencies and deployment plans. TOSCA-based configuration and deployment plans are described in XML. TOSCA lets providers specify various resource and dependency types. Figure 4a shows a TOSCA server template that describes a cloud service with four sections: a topology template that describes the dependency between two components, *MyApplication* and *MyAppServer*; these two components' node and relationship types; and two deployment plans, *DeployNewApplication* and *RemoveApplication*.

However, TOSCA doesn't specify elasticity. More concretely, basic TOSCA server templates contain infrastructure information that describes cloud applications at build-time, but they lack information needed for application deployment and scaling at runtime. We can extend standard TOSCA with elasticity-enriched directives using SYBL and service-level agreements (SLAs) to guide application elasticity in clouds. With SYBL, the service provider or developer can easily specify different elasticity strategies. For example, a provider could define a scaling in/out strategy using SLAs based on performance and budgets.

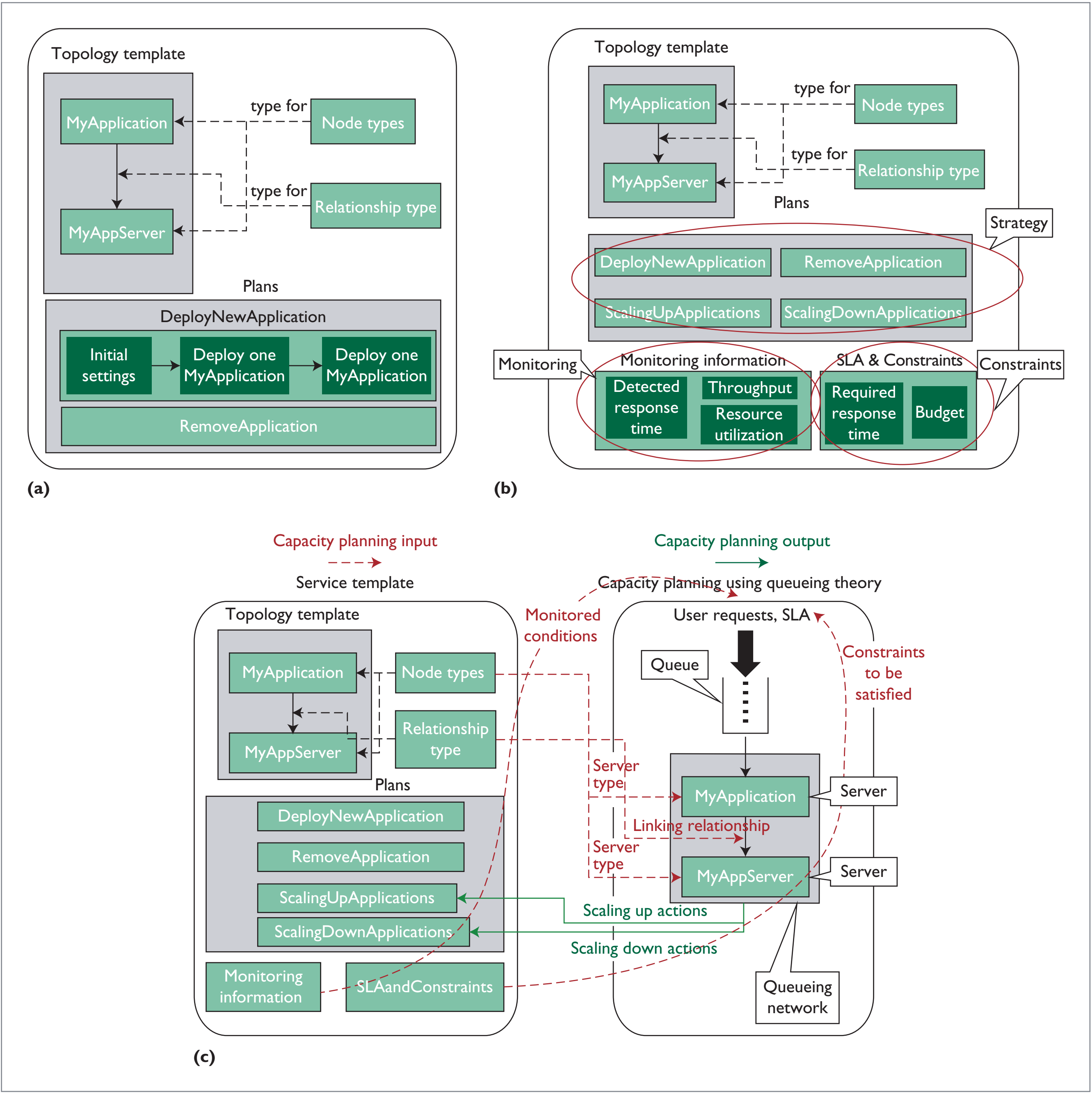

Figure 4. Using SYBL for TOSCA. We can see (a) a server template of the basic TOSCA, (b) a server template of the elasticity-enriched TOSCA, and (c) how we'd use a queuing network to control elasticity strategies.

Using SYBL-enriched XML for *MyApplication*, we can generate a new SYBL-enriched TOSCA-based XML that includes monitoring information structures and new plans for scaling in/out. Figure 4b shows an exemplary server template for elasticity-enriched TOSCA that includes three extended sections, corresponding to three components in programming directives for elasticity.

The monitoring section mainly includes the monitored application's current status and underlying infrastructures. In the example fragment in Figure 5, this section records the detected response time and the application throughput, as well as the utilization of resources in *MyApplication*.

The constraints and SLA section specifies any constraints on quality, budget, and other aspects of the application. In the example in Figure 6, this section specifies the application's required response time and budget.

Finally, we extend the plan section in standard TOSCA to define more actions that handle the application's scaling cases. Figure 4b defines two actions — *ScalingUpApplication* and *ScalingDownApplication*.

Using the elasticity-enriched TOSCA, we can achieve capacity planning in application deployment and scaling. For example, Figure 4c illustrates how the information stored in a server template of the elasticity-enriched TOSCA maps to the corresponding part of a queuing network (dashed lines). After conducting capacity planning using queuing theory — that is, estimating the number of *MyApplication* and *MyAppServer* servers — we use the result of capacity planning to update the information in the `Plans` section and control the application's deployment and scaling actions.

```
<MonitoringInformationTemplate id="ApplicationMonitor"
                 name="Application Monitor"
                 InformationType="Monitor">
        <DetectedResponseTime>2.5</DetectedResponseTime>
        <Throughput>1000</Throughput>
        <ResourceUtilisation>
             <CPU>80</CPU>
             <Memory>75</Memory>
             <I/O>50</I/O>
        </ResourceUtilisation>
</MonitoringInformationTemplate>
```

Figure 5. Monitoring section. This section records the detected response time and the application throughput.

```
  <ConstraintsAndSLATemplate id="ApplicationSLA"
                 name="Application SLA"
                 InformationType="SLA">
        <RequiredResponseTime>3.0</RequiredResponseTime>
        <Budget>10EUR</Budget>
  </ConstraintsAndSLATemplate>
```

Figure 6. Constraints and SLA section. This section specifies the application's required response time and budget.

Programming directives can be a powerful tool for controlling application elasticity in cloud environments. We've described our initial steps toward developing a full-fledged directive language specification and runtime system explicitly supporting elasticity. Going forward, we're working on the full SYBL specification and its runtime components.

References

1. *OpenMP Application Program Interface*, version 3.1, OpenMP Architecture Rev. Board, July 2011, www.openmp.org/mp-documents/OpenMP3.1.pdf.
2. K. Kennedy, C. Koelbel, and H. Zima, "The Rise and Fall of High-Performance Fortran: An Historical Object Lesson," *Proc. 3rd ACM SIGPLAN Conf. History of Programming Languages* (HOPL III), ACM, 2007, pp. 7-1–7-22; http://doi.acm.org/10.1145/1238844.1238851.
3. S. Dustdar et al., "Principles of Elastic Processes," *IEEE Internet Computing*, vol. 15, no. 5, 2011, pp. 66–71.
4. C. Bunch et al., "Shams: Language and Runtime Support for Automatic Configuration and Deployment of Scientific Computing Software over Cloud Fabrics," *J. Grid Computing*, vol. 10, no. 1, 2012, pp. 23–46.
5. S. Dustdar and H.-L. Truong, "Virtualizing Software and Humans for Elastic Processes in Multiple Clouds — a Service Management Perspective," *Int'l J. Next-Generation Computing*, vol. 3, July 2012; http://perpetualinnovation.net/ojs/index.php/ijngc/article/view/148.
6. T.A. Limoncelli, "OpenFlow: A Radical New Idea in Networking," *Queue*, vol. 10, no. 6, 2012, pp. 40–46; http://doi.acm.org/10.1145/2246036.2305856.
7. T. Binz et al., "Portable Cloud Services Using TOSCA," *IEEE Internet Computing*, vol. 16, no. 3, 2012, pp. 80–85.

Schahram Dustdar is a full professor of computer science (informatics) with a focus on Internet technologies and heads the Distributed Systems Group, Institute of Information Systems, at the Vienna University of Technology. He's an ACM Distinguished Scientist. Contact him at dustdar@dsg.tuwien.ac.at; www.infosys.tuwien.ac.at/.

Yike Guo is a computing science professor in the Department of Computing, Imperial College London. His research is in large-scale scientific data analysis, data mining algorithms and applications, parallel algorithms, and cloud computing. Contact him at yg@doc.ic.ac.uk; www.doc.ic.ac.uk/~yg/.

Rui Han is a researcher and PhD student in the Discovery Science research group, Department of Computing, Imperial College London. Contact him at r.han10@imperial.ac.uk; www.doc.ic.ac.uk/~rh1910/.

Benjamin Satzger is an assistant professor of computer science in the Distributed Systems Group, Institute of Information Systems, at the Vienna University of Technology. Contact him at satzger@infosys.tuwien.ac.at; www.infosys.tuwien.ac.at/staff/bsatzger/.

Hong-Linh Truong is a senior researcher in the Distributed Systems Group, Institute of Information Systems, at the Vienna University of Technology. Contact him at truong@dsg.tuwien.ac.at; www.infosys.tuwien.ac.at/staff/truong/.