Testing Elastic Computing Systems

**Alessio Gambi, Waldemar Hummer, Hong-Linh Truong,
and Schahram Dustdar** • *Vienna University of Technology*

Elastic computing systems are a new breed of software system that arose with cloud computing and continue to gain increasing attention and adoption. They stretch and contract in response to external stimuli such as the input workload, aiming to balance resource use, costs, and quality of service. Here, the authors introduce novel ideas on testing for elastic computing systems, identify some primary research challenges, and discuss future directions for this topic.

Cloud computing has paved the way for a new class of computing systems in which elasticity emerges as a core design principle. Elasticity has multidimensional properties with regard to resources, quality, and costs.[1] *Resource elasticity* is the most common form; it lets computing systems dynamically acquire and release resources (such as virtual machines) in reaction to workload fluctuations. Systems supporting such elasticity stretch the resources they utilize to increase computation capacity. These systems can implement elasticity in terms of infrastructure resources, processes, and human resources.

Systems supporting *quality elasticity* dynamically adjust the level of quality properties, such as quality of service (QoS). For instance, they can run data-intensive applications that dynamically adjust the data's consistency level while minimizing poor performance or resource allocation.[2] Systems supporting *cost elasticity* dynamically adjust the budget devoted to their runs while tolerating variable QoS. Most often, systems for scientific workflows[3] exploit this type of elasticity.

As effort goes increasingly toward researching and developing elastic computing systems, we naturally need testing techniques for them. To test such systems, we must understand that elasticity is dynamic[2] and is influenced by a rich set of factors, including the application's business logic, the input workload, the control logic that determines resource allocation (or system adaptation), the infrastructure that provides the resources, and other working conditions. These factors are complex, so correctly designing elastic systems is challenging, and predicting their evolution under all possible combinations of factors is difficult. To date, the research community hasn't paid enough attention to testing elastic systems (see the "State-of-the-Art in Testing Elastic Computing Systems" sidebar).

To this end, we aim to define new methodologies that extend traditional software and system testing with concepts defined in the elasticity context and that are tailored to identify those problems rooted in the elevated flexibility inherent to elastic computing systems. Because resource elasticity is the most prevalent form, we focus on testing techniques for resource-elastic systems before investigating those for other types.

Metaphors for Testing

Intuitively, resource-elastic computing systems mimic elastic materials that respond to external stimuli by self-adapting. In a sense, elastic computing systems stretch when we apply external stimuli, and will eventually contract to their original shape once such stimuli are removed. We thus study two metaphors: that of elastic materials, to identify the main concepts, properties, and terminology for describing system elasticity in computing systems; and that of mechanical testing, to define suitable techniques for testing such systems.

State-of-the-Art in Testing Elastic Computing Systems

To date, we lack conceptual frameworks and research directions for testing elastic systems. Previous work has focused primarily on load testing,[1] scalability testing,[2] and providing the technological foundation for efficient test execution in the cloud. One study elaborates on different aspects of *testing as a service* (TaaS) in the cloud,[3] including test parallelization, fault tolerance, and cost considerations. Elasticity is considered for the TaaS platform itself, but not particularly for the systems and applications under test, which is the focus of our work.

Because elastic systems are closely related to self-adaptive ones, software engineering principles in this field[4] are partly applicable to elastic systems, yet can't capture all issues. Often, engineers make strong assumptions about the environment, the application, and the workload to achieve feasible designs. For example, many approaches assume perfect actuators, immediate effects on system behavior due to adaptation, performance stability, and predictable workloads. However, these assumptions hardly hold in practice, particularly in the cloud, where noise, transitory behaviors, unexpected events, and fast workload dynamics are common.[5] Indeed, by relying on these assumptions, elastic computing systems might become too fragile, and their behavior in terms of quality of service and cost savings might be far from optimal and sometimes even counterintuitive.[6]

This situation clearly conflicts with the requirements of high software quality and dependability demanded by the business-critical features that elastic computing systems often implement. It also highlights the need for novel methodologies, methods, and tools that not only focus on elastic computing systems' design and implementation, but also target important concerns such as modeling, benchmarking, and validation. We argue that upcoming research efforts in these directions must have system elasticity as a central point of investigation, and must thus be tailored to its peculiarities. In particular, we aim to improve elastic computing systems' software quality via systematic testing, which we can combine with current work on benchmarking,[7] formalization,[8] and simulation.[9]

References

1. A. Avritzer and E.J. Weyuker, "The Automatic Generation of Load Test Suites and the Assessment of the Resulting Software," *IEEE Trans. Software Eng.*, vol. 21, no. 9, 1995, pp. 705–716.
2. W.-T. Tsai et al., "Testing the Scalability of SaaS Applications," *IEEE Int'l Conf. Service-Oriented Computing and Applications* (SOCA 11), IEEE, 2011, pp. 1–4.
3. L. Yu et al., "Testing as a Service over Cloud," *Proc. IEEE Int'l Symp. Service Oriented System Eng.* (SOSE 10), IEEE, 2010, pp. 181–188.
4. Y. Brun et al., *Software Eng. for Self-Adaptive Systems*, Springer, 2009.
5. A. Iosup et al., "Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing," *IEEE Trans. Parallel and Distributed Systems*, vol. 22, no. 6, 2011, pp. 931–945.
6. A. Gambi et al., "Kriging Controllers for Cloud Applications," *IEEE Internet Computing*, vol. 17, no. 4, 2013, pp. 40–47.
7. S. Islam et al., "How a Consumer Can Measure Elasticity for Cloud Platforms," *Proc. ACM/SPEC Int'l Conf. Performance Eng.* (ICPE 12), ACM, 2012, pp. 85–96.
8. A. Gambi et al., "Iterative Test Suites Refinement for Elastic Computing Systems," *Proc. Joint Meeting of the European Software Eng. Conf. and ACM SIGSOFT Symp. Foundations of Software Eng.*, (ESEC/FSE 13), ACM, 2013, pp. 635–638.
9. R. Buyya et al., "Modeling and Simulation of Scalable Cloud Computing Environments and the CloudSim Toolkit: Challenges and Opportunities," *IEEE Int'l Conf. High Performance Computing & Simulation* (HPCS), IEEE, 2009, pp. 23–50.

In using the elastic materials metaphor, we consider the elastic computing system under test, the workload, and the change in the system's scale as the *specimen* (material), *stress factor* (such as tensile force), and specimen's *deformation* (for example, elongation), respectively. The stress-strain curve (see Figure 1) is a common approach depicting how specimens' elasticity evolves and identifying points at which materials (sometimes irreversibly) change their state. Depending on the stress's intensity, an elastic material passes among different states.

Elasticity occurs when the specimen returns to its original shape after deformation. With elasticity, the relationship between stress-strain is proportional. Under some linearity assumptions, we can express elasticity via the elastic modulus, or Young's modulus — that is, the deformation per unit of stress.

Plasticity occurs when the specimen can't recover its original shape after we remove the stress. In a software context, plastic systems might only be able to scale up (and risk breaking if a certain scale is reached), whereas elastic systems can dynamically scale up and down.

A specimen that doesn't show any deformation is *inelastic*. In our context, if (part of) a computing system is inelastic, it doesn't adapt to changes in the workload, and hence might be over- or under-provisioned. Finally, *necking* occurs when the specimen starts to break.

Of particular interest are the crossover points: the *yield point* between the elastic and plastic states, the *ultimate stress* just before fractures and cracks appear, and the *failure point* where the specimen breaks.

We want to identify the different states and points for characterizing elastic computing systems' behavior, and develop models that intuitively correspond to stress-strain curves. These models will let designers predict how systems behave under different working conditions — that is, stress levels — and easily compare
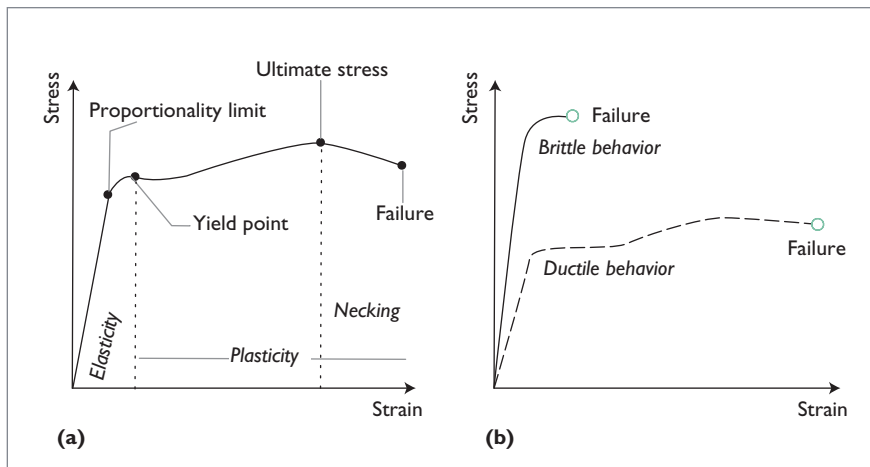
Figure 1. Stress-strain curves for elastic materials. The curve identifies points at which materials change their state. We can see (a) crossover points, which are of particular interest, and (b) behaviors at different stress levels.

different implementations, as engineers do when comparing different materials (such as the ductile and brittle behaviors in Figure 1b). These abilities are fundamental for validating elastic systems against end-user requirements, easing the design of complex systems that employ elastic systems as their inner components, and achieving high software quality. Using the elastic materials metaphor, we can use several types of mechanical testing to reveal computing systems' elastic and inelastic behaviors.

Tensile testing exposes elastic systems to a stable load that increases between each run until failures occur. At each load increment, testers check if the system recovers the initial configuration and collect data about its behavior to derive the stress-strain curve. In a software context, tensile testing follows the same basic process as load testing, but focuses on properties related to elasticity, rather than just performance.

Stress-strain curves and tensile testing don't explicitly consider time, presenting only a static view of system elasticity. Additional modeling and testing methodologies are required when we must understand the dynamic aspects of system elasticity. Being able to expand and contract during operations is useful only if it occurs

correctly — that is, with no failures and in a timely manner. Consider an example in which a resource-elastic system uses automatic scaling to maintain predefined performances when the load fluctuates. If the time to acquire additional resources is too long, then the elastic system fails to provide the expected performance. Similarly, if the time to release resources is too long, then the system ends up costing more than expected. On the flip side, if the time to release resources is too short, then the elastic system might become too aggressive and result in resource thrashing, causing it to be more expensive than expected.

Impact testing is one way to move toward a more dynamic system analysis. It subjects specimens to load peaks to determine how much energy they can absorb, how fast, and whether they fail. In a software context, we can use impact testing to study system adaptations as well as find which conditions lead to failures. This is similar to traditional software stress testing. For example, impact testing can identify the point at which a system rejects too many requests if subjected to particular loads.

Adaptation speed isn't elastic systems' only time-dependent criticality. In fact, system degradation due to cyclic adaptations is also a concern.

For software in particular, we argue that — if improperly designed — elastic systems might show quality degradation sooner than traditional systems because they pass through several cycles of scaling and shrinking that might consume them. For similar reasons, faults can propagate faster in elastic computing systems, and could quickly lead to hard system failures.

Fatigue testing subjects specimens to cyclic stress that causes localized adaptations, and could eventually result in structural damage. Its goals are to measure the system lifetime until failures occur, and to understand how and whether specimens' elastic capabilities degrade over time. The cyclic stress can be of fixed amplitude and frequency, or randomly generated. The motivations and processes underlying fatigue testing are similar to those for endurance testing in software. The difference is that endurance testing targets specific bugs related to memory management, memory leaks, and buffer overflows, which might be hard to identify in "instantaneous" tests. Fatigue testing aims to find problems that relate to continuous and cyclical system adaptations, such as computing nodes joining and leaving elastic clusters.

Fatigue testing can also help us study how faults propagate in elastic materials with regard to their cyclic adaptations. During testing, cracks and ruptures might result from material weaknesses, or testers can inject them. While subjecting a specimen to cyclic stress, testers investigate whether cracks expand and eventually lead to specimen failure. Similarly, in software we might study how localized faults propagate in the system, whether elastic systems can automatically absorb them (self-healing), or if faults will eventually lead to hard system failures. In the form of *resonance testing*, fatigue testing lets us discover cyclic stress that leads to uncontrolled oscillations in resource allocation. Understanding whether systems have

Table 1. Mapping metaphors to elastic computing systems.	
Mechanical testing metaphor	**Analogy in elastic computing system**
Specimen deformation (elongation or shortening)	Increase or decrease in system scale when resources are allocated or removed
Recovery of original shape	Release of computing resources
Plasticity	System inability to scale down
Necking	(Unrecoverable) system failure, caused by scale-in and out operations
Tensile testing	Detecting elastic states and elastic transitions along particular system configuration paths — for example, generating a finite number of requests at constant rates to force system scale-out and check whether the system can recover its initial configuration
Impact testing	Testing techniques that study how fast resources can be allocated to an elastic computing system — for example, generating requests according to a step function to trigger system scale-out without reaching the necking state (without failures), and measuring the time to reach the final configuration
Fatigue testing	Testing techniques that study whether systems can go over budget within an observation period — for example, generating waves of requests that trigger consecutive scale-up and down, and then measure the resource usage costs according to a specified billing model
Shear testing	Testing techniques that study the changes in systems' elastic behavior that occur due to interference and physical resource contentions in the underlying platform — for example, pinning the system's computing resources under test on physical servers, deploying resource-eager virtual machines next to them on the same physical servers, and then comparing the resulting elastic behavior with baseline cases

such resonant frequencies might let designers compensate or even filter them out.

Several factors that aren't directly observable or controllable can affect the elasticity that a given software system achieves at runtime. For example, if an elastic system shares infrastructure resources with other cloud users that are resource-eager (sometimes called "noisy neighbors"[4]), then the system's quality might degrade as a secondary effect of resource contention. This situation can also drive the elastic system toward non-optimal and even risky states. So, understanding the side effects from other infrastructure users can help developers measure elastic systems' robustness.

For these cases, *shear testing* might be useful. In the context of mechanical testing, shear tests subject specimens to lateral forces that aren't directed toward elongation but that still result in material deformations. Under this stress, the material might adapt up to the point at which it eventually breaks. Similarly, for software, we might find tests that lead the system to misbehave as a consequence of "lateral forces" such as resource contention.

We can map these metaphors to the context of elastic computing system testing to determine our testing framework's main concepts and identify important guidelines on how to implement such tests. Table 1 lists some important examples of possible mappings between our mechanical testing metaphors and testing techniques for elastic systems.

Conceptual Framework

Guided by elasticity concepts and our testing metaphors, we propose a conceptual framework for testing resource-elastic computing systems (see Figure 2). We developed this framework around testers — that is, the actors in charge of testing the system. It employs four common test activities: test-case generation, test execution, data analysis, and test evolution. Testers provide the initial input as regards primary testing goals and optional test constraints. Then, the testing activity sequence begins, and each activity's output becomes input for subsequent activities.

Test-case generation receives testing goals as input and produces a set of test specifications — that is, a test suite — as output. Test cases in our context specify the characteristics and configurations of the elastic system under test, the load generators that will generate the input workload, and the type of workload being generated. This includes defining request intensity, the request mix, input data, and their variations during test execution to create load fluctuations.
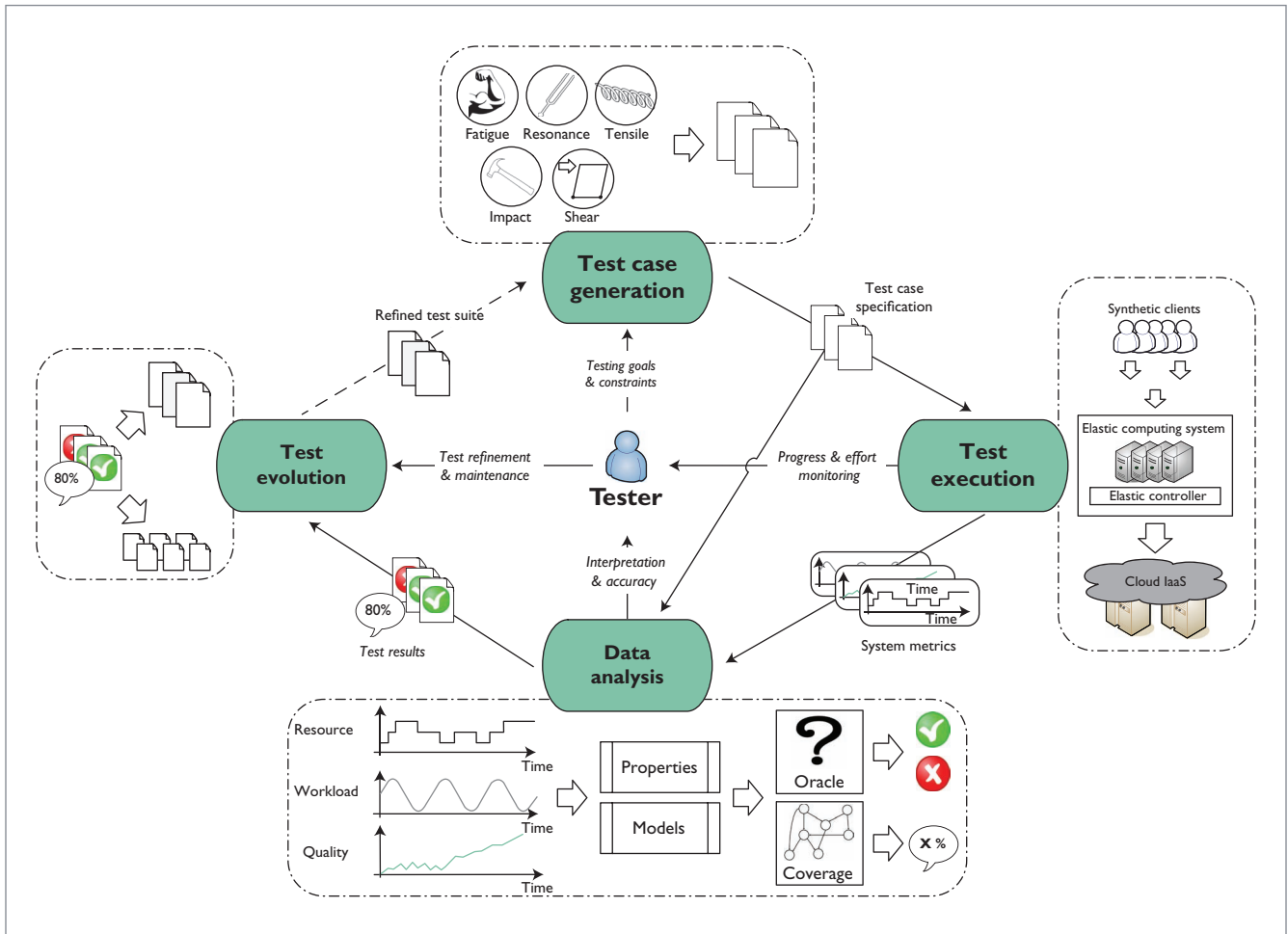
Figure 2. Conceptual framework for testing elastic computing systems. Testers provide the initial input as regards primary testing goals and optional test constraints. Then, the testing activity sequence begins, and each activity's output becomes input for subsequent activities.

For example, a test case might vary the number of active users according to a time-dependent, wave-oscillating pattern. A workload generated in this way reaches the elastic system and triggers its elasticity mechanisms. Test cases can contain additional specifications about environmental settings and variations. For example, a fatigue test might specify that in a given moment, a network link fails, or additional resource-eager virtual machines must be deployed in the cloud to increase resource contention.

During test execution, one or multiple instances of an elastic computing system are running, subjected to different workloads and execution conditions. The actual number of concurrent system instances varies depending on cost and resource constraints. During test execution, testers collect the system metrics listed in the test-case specifications. Meanwhile, testers can monitor the execution's progress, as well as effort, in terms of time, costs, and resources invested.

The data analysis stage follows test execution and elaborates the monitored metrics to eventually produce the final test results. Depending on the selected testing goals, test results might contain pass/fail reports, coverage measures, fault-localization results, and more. Testers can inspect the results, interpret their meaning, and check their accuracy, which lets testers identify bugs or problems in the system that might trigger code fixes, as well as flaws and inaccuracies in the test cases that could necessitate test-suite refinement.

Test evolution manages the test-suite life cycle by reflecting changes in the code on one side and improving test-case quality on the other. If a new version is released, testers can repeat the entire activity sequence to test it. Similarly, they can reiterate the cycle if they want to improve test quality.

To put such a conceptual framework into practice, we must address several research challenges.

Methodological Approach
Testers need languages that can easily describe the prescriptive elasticity that

systems under test should achieve, as well as a way to measure the descriptive elasticity actually achieved. These formalisms must be expressive enough to capture elastic behaviors' static (or time-less) and dynamic (time-dependent) characteristics, and the relations among resource allocation, cost, and quality. Testers must couple formalisms to new languages to capture stakeholder requirements on system elasticity, and use novel tools to discover, define, monitor, and check elastic properties.

Testers also need novel coverage metrics and criteria that might require defining new abstractions over the code and runtime behavior. For example, testers might want to cover all possible system configurations and the transitions among them, as well as particular paths across system configurations that form elastic state transitions.

Testers must determine whether to test elastic properties at the system, component, or unit level. Integration tests become critical when software employs elastic systems as components, and testers must understand the mutual effects of putting elastic components together. These additional tests require new tools that support their execution — for example, by creating and managing mockups and scaffoldings in the context of unit- and component-level tests.

No matter the scope, testers must follow precise guidelines about tests' applicability for elastic systems. Some techniques — for example, those that we derived for impact and fatigue testing — can be used to implement similar test goals, whereas others — such as shear testing or resonance testing — have a more focused target and thus limited applicability. Context can also limit a test's applicability. When multiple techniques can achieve a specific test goal, the guidelines must clearly state the conditions under which one technique is preferable.

Test-Case Generation

Testers can create test cases manually or employ test-case generators to create them automatically. Such generators might adopt different techniques to define new test cases, but in general are guided toward achieving specific testing goals, including identifying yield points and plasticity in the system, sub-optimality in system adaptation when it comes to costs and quality, or resonant oscillations.

In our context, elasticity mainly reacts to variations in the load, so we must express test cases in terms of the workload to force a particular state of elasticity or trigger specific elastic adaptations. Generating such loads is challenging: testers or automatic test-case generators must deal with a multidimensional and time-dependent test input space because elastic systems implement various operations. Different operations can result in different resource demand, and the intensity and request mix can vary in time.

To further complicate this situation, elasticity isn't directly observable but must be derived from a set of covariant system metrics, making the test-case-generation process depend significantly on these mappings' complexity. Moreover, because elasticity is a nonfunctional property and clouds are generally noisy environments, test cases must consider accuracy metrics, and testers must account for repeated test executions to gain more evidence.

As much as possible, test cases should be portable and reusable. They must be generated following general principles and processes that don't depend on specific cloud applications or platforms. We argue that test cases specified in terms of timed request traces (but also statistical distributions) can be a first step toward generally tackling test-case generation for elastic systems. When it comes to generating test cases that also consider environmental characteristics, as with

shear testing, testers must specify additional elements such as which faults to inject, virtual machine placement, and other platform settings that are difficult to express abstractly without environmental reference models. Specific models and fault taxonomies are available for particular instantiations of typically elastic systems — for instance, event-based data processing platforms.[5] Further efforts are required, however, to study faults in elastic computing systems in general.

Test Execution

Testers must deploy and configure elastic computing systems and their load generators in the target cloud, start all components, collect monitoring data, and analyze it at the end of the execution while releasing computing resources. Testers thus need generic tools to automatically manage test executions over different platforms. Tool availability and automation reduce the burden on testers, saving them time, and are thus a priority.

Testers can deploy multiple instances of elastic systems side-by-side to parallelize test execution, or across clouds to compare different outcomes. Unfortunately, without optimization, the increase in resources required to parallelize execution makes it very expensive. Tools must be effective and optimally share computing resources across test executions to fulfill time and budget constraints.

Test execution tools should be able to measure the accuracy of the data they monitor. For example, such tools can schedule repeated executions if the working conditions are too noisy, and should react to transient failures or illegal states in the platforms.

Data Analysis, Test Results, and Evolving Test Cases

Test execution can produce large amounts of data that testers must analyze to compute the final results. The data analysis methods employed to

derive abstractions of elastic behaviors, compute coverage metrics, and check whether system specifications are respected must be very efficient. Because tests must help designers highlight problems and find solutions, test results must also be accurate. Testers must refine tests that don't produce accurate results, and evolve them to reflect changes in elastic computing systems or the platform running them.

Testers must then be able to define appropriate quality metrics and determine when the platforms running the systems undergo some evolution that could make previous test cases either inappropriate or no longer executable. Changes in billing models and cloud offers, for example, could result in test executions that violate testers' budget constraints. In this context, test reuse and regression-testing techniques might need revisiting.

We are currently investigating model-based search techniques to find problematic scenarios and cover elastic transition sequences in the context of resonance and endurance testing. These contributions are the first steps toward proper testing of elastic computing systems, and their promising results stimulate and motivate further research. Furthermore, because elasticity is arising as one of the core system properties that must be provided by cloud-based systems, we argue that additional research in software engineering — from requirements engineering to programming language development to software maintenance — must target it.

References
1. S. Dustdar et al., "Principles of Elastic Processes," *IEEE Internet Computing*, vol. 15, no. 5, 2011, pp. 66–71.
2. D. Agrawal et al., "Database Scalability, Elasticity, and Autonomy in the Cloud," *Proc. Int'l Conf. Database Systems for Advanced Applications*, Springer, 2011, pp. 2–15.
3. E.-K. Byun et al., "Cost-Optimized Provisioning of Elastic Resources for Application Workflows," *Future Generation Computer Systems*, vol. 27, no. 8, 2011, pp. 1011–1026.
4. A. Gulati et al., "Cloud-Scale Resource Management: Challenges and Techniques," *Proc. 3rd Usenix Conf. Hot Topics in Cloud Computing*, Usenix Assoc., 2011, p. 3.
5. W. Hummer et al., "Deriving a Unified Fault Taxonomy for Event-Based Systems," *Proc. 6th ACM Int'l Conf. Distributed Event-Based Systems* (DEBS 12), ACM, 2012, pp. 167–178.

Alessio Gambi is a postdoctoral researcher visiting the Distributed Systems Group at the Vienna University of Technology. He has a PhD in informatics from the University of Lugano, Switzerland. Contact him at alessio.gambi@usi.ch; www.inf.usi.ch/phd/gambi/.

Waldemar Hummer is a PhD candidate in the Distributed Systems Group at the Vienna University of Technology. Contact him at hummer@dsg.tuwien.ac.at; dsg.tuwien.ac.at/staff/hummer.

Hong-Linh Truong is an assistant professor in the Distributed Systems Group at the Vienna University of Technology. Contact him at truong@dsg.tuwien.ac.at; dsg.tuwien.ac.at/staff/truong/.

Schahram Dustdar is a full professor of computer science (informatics) with a focus on Internet technologies and heads the Distributed Systems Group at the Vienna University of Technology. He's an ACM Distinguished Scientist and recipient of the IBM Faculty award 2012. Contact him at dustdar@dsg.tuwien.ac.at; dsg.tuwien.ac.at/.

cn *Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.*