Testing Elastic Systems with Surrogate Models

Alessio Gambi*†, Waldemar Hummer*, and Schahram Dustdar*

* Vienna University of Technology, Austria, {lastname}@dsg.tuwien.ac.at
† University of Lugano, Switzerland, {name.surname}@usi.ch

***Abstract*—We combine search-based test case generation and surrogate models for black-box system testing of elastic systems. We aim to efficiently generate tests that expose functional errors and performance problems related to system elasticity.**

Elastic systems dynamically change their resources allocation to provide consistent quality of service in face of workload fluctuations. However, their ability to adapt could be a double edged sword if not properly designed: They may fail to acquire the right amount of resources or even fail to release them. Black-box system testing may expose such problems by stimulating system elasticity with suitable sequences of interactions. However, finding such sequences is far from trivial because the number of possible combinations of requests over time is unbounded.

In this paper, we analyze the problem of generating test cases for elastic systems, we cast it as a search-based optimization combined with surrogate models, and present the conceptual framework that supports its execution.

***Index Terms*—model based testing, load testing, genetic algorithm, surrogate models**

I. Introduction

Test and analysis of *elasticity* has found widespread attention in areas that are (if at all) only remotely related to computer science, for instance in the materials science for rubber and plastics (e.g., [3]). Even though there is no direct relationship between the fields, we argue that we can benefit from these ideas by applying the same rigor to testing and analysis of elastic computer systems. In general terms, an elastic system (ES) is a system that responds to an external force or other stimulus, and has the ability to change its internal structure and characteristics accordingly (see more detailed definition in Section II). In this regard, elasticity can be seen as a specific type of adaptivity, with the peculiarity that an adaptation into one *direction*, i.e., scale out, is typically followed by an adaptation into the opposite direction, i.e., scale in, when the source of stress is removed or is reduced.

Elasticity, if not properly designed, can be a double edged sword that may result in harmful system behaviors or unexpected costs [18]. Exposing an ES to operations which change its elasticity state can cause subtle modifications, which may become, in the worst case, uncontrollable or irreversible. For example, an ES may start to acquire resources in an uncontrolled way, it may fail to release resources, it may oscillate between alternative allocations of resources (denoted *thrashing*), it may be unable to scale back to its initial configuration, it may fail to allocate resources on time to provide consistent quality of service (QoS), and more [16].

This work is partially supported by the Swiss National Science Foundation under the "Fellowship for Prospective Researches" contract PBTIP2-142337, and by the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement 257483 (Indenica).

We distinguish two types (or levels) of strain that can be put on an ES: 1) *fair operations* which are perfectly valid and within the capabilities of the system, and 2) *excessive operations* that simply overload the ES beyond its capabilities or physical resources. While it is easy to break a system by stressing it with excessive load, interestingly we can also observe a tendency of things to break under continuous application of fair operations (e.g., a rubber material that finally breaks after millions of slight stretches; similarly, an ES that becomes unresponsive after failing to free resources in a multitude of acquire/release operations) [16]. However, given the complexity of computer systems (not least in comparison to the relatively homogeneous structure of synthetic materials), applying the right sequence of fair operations to break an ES is comparable to finding the proverbial needle in a haystack.

We tackle this issue and study systematic testing of elastic systems. We aim to generate test cases that stress a system with fair (valid) operations over time, causing a multitude of state changes, potentially identifying faults related to elasticity. With our test results we strive to obtain 1) an understanding of the load and request patterns that an ES is able to handle before it reaches a limit and breaks, and 2) a certain level of confidence (expressed in test coverage) that an ES indeed provides reliable elasticity, within the boundaries of valid operations. To achieve this goal, we propose the use of surrogate models [25] that describe the system with adequate accuracy and level of abstraction, combined with efficient search-based software engineering methods [1]. In concrete terms, our approach integrates model-based testing [24] with heuristic search, exemplified on the basis of Genetic Algorithms [12].

This paper provides an outline of the testing approach, formalizes the search problem for test generation with surrogate models, and discusses the conceptual architecture that supports it. We conclude by listing the main research directions.

II. A Model of Elastic Systems

In the following we provide a simplified model for elastic systems. Our approach relies on information about the state of an ES with respect to its elasticity properties. An *elasticity property* defines a metric about the structure or behavior of a system (e.g., number of computing nodes, thread pool size, allocatable memory, current QoS, etc.). We assume that the property values have a (total) ordering (e.g., QoS q_1 is higher than QoS q_2) and are usually constrained by lower and upper bounds (if numeric). The *elasticity state* is a snapshot of (a subset of) the elasticity properties and their current values.

We define changes in an ES based on the notion of *elastic transition sequences* (ETS). Two key differences to transitions

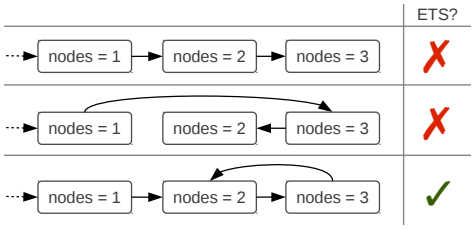

Fig. 1. Illustration of Elastic Transition Sequences

in purely adaptive systems are that 1) transitions along an ETS are defined over subsequent values (no values are "skipped"), and 2) an ETS involves at least a transition into one direction *and* an eventual transition back in the opposite direction. A *full* ETS is an ETS with starting state equal to final state. Figure 1 illustrates a simple ES with an elasticity property that denotes the number of nodes ($nodes \in \{1, 2, 3\}$). The first transition sequence is not an ETS, because the system only scales up, but never down. The second example is not an ETS, because it skips $nodes = 2$. The third example is an ETS (but not a full ETS). The example in Figure 1 can be extended for higher dimensions with multiple elasticity properties.

The behavior of an ES, in particular triggering of state transitions, is basically determined by external inputs. Inputs can be either user requests, or incoming data to be processed, or any other external aspect that influences the processing. We abstract from the actual content, and assume that inputs can be categorized into sets of input types $I = \{i_1, i_2, ..., i_n\}$, and measured or counted (e.g., requests per time unit). For instance, consider an event-based system which runs continuous queries and elastically scales with the number of processed events [13]. The inputs could be defined as $I = \{$number of queries, data rate$\}$. The outputs (categorized into types $O = \{o_1, o_2, ..., o_m\}$) determine the ES's correct operation and QoS. Moreover, S denotes the set of possible elasticity states, and s_t is the state at time t. Apart from s_t, we assume a black-box model: no access is required to the internals, as long as we can probe the ES and observe its elasticity properties.

III. Testing Goals

The basic motivation behind our work is that implementation of elastic systems is inherently prone to errors, and hence requires thorough testing [16]. We target two specific problem areas: 1) different request loads and patterns may cause functional errors due to implementation bugs, which can be revealed by systematic testing, and 2) elasticity generally affects resource utilization, and as resources are associated with costs, elasticity properties of an ES are considered a critical feature that deserves to be tested.

The problem we tackle is to generate test cases in the form of input traces over a predefined time interval. The test cases should stimulate the triggering of elasticity within the system and expose bugs that derive from it. The problem is challenging because the possible traces that can be generated are infinite, and the behavior of the ES may depend heavily on the past interactions with the system.

We define two orthogonal test goals, which can be combined using our approach: 1) finding specific cases that break the system using fair (valid) operations, and 2) generating suites that achieve a certain test coverage. For 1), we utilize search combined with surrogate models to predict the behavior of the ES. For 2), we can utilize the model of ES and generate minimal test suites with test cases that cover, e.g., all ETSs in the system. The details discussed in the following (Section V) cover the problem encoding for 1), but the search formalization can be easily extended to cover the goal in 2).

IV. Search Formalization

We formalize the problem of generating traces as search problem, and employ Genetic Algorithm [2] (GA) as search-based technique to solve it. Therefore in the remainder of this section we focus on concepts that are specific to GA.

Individuals encoding. Test cases are traces, that is, sequences of inputs in time. We represent them as sequences of workload vectors, i.e., amount of requests for each type over a very small interval [19]. As request traces can be very long, a direct encoding of workload vectors as genomes may be too hard to manage. Hence, we decide to compact traces by aggregating them over larger intervals and represent the load distribution in each interval in a functional form by means of patterns.

A *pattern* encodes a family of distributions of requests over a limited time period (pattern duration, PD) using a base functional and a set of parameters. Sample patterns are: constant pattern with value parameter; ramp with slope parameter; oscillating pattern with amplitude, frequency and other parameters. Islam et al. [16] list other interesting patterns.

By sequencing patterns we can encode long traces in genomes of predefined length, denoted number of patterns (NP). Each genome cell contains a mapping to the pattern, its duration, and parameterizations. We call this pattern sequence the *master trace* (cf. Figure 2). We account for different types of inputs by means of *ratio* parameters ($R_1, R_2, \ldots$): In every cell these parameters multiply the values of the master trace to obtain per-request traces. Once per-request sequences are ready, we derive test cases by discretizing and merging their functions into a single trace. Figure 2 exemplifies this process.

We make the following simplifications to ease the evolution of genomes during the search: We assume that the length of the genome is predefined, that each pattern has a constant duration, and that we use only the oscillating pattern, encoded as $A*sin(B*t+C)+D$, to generate the sequence. Furthermore, we consider only the rate of input requests and not their input data. We assume that input requests can be partitioned into homogeneous classes, achieving a desired level of test coverage [14]. We argue that these choices ease the search but provide good levels of expressiveness and flexibility. In fact the oscillating pattern has four degrees of freedom and can generate very different load distributions (see Figure 2). Moreover, in case different input data result in very different demands on the system, we can encode them as requests belonging to different classes, and use the ratio parameters to govern their relative distribution. Our intuition is that the oscillating pattern can suitably capture the load oscillations that normally trigger system elasticity.

Mutation operators. We randomly select and mutate individuals by applying the following operators: i) *mutate pattern*: we

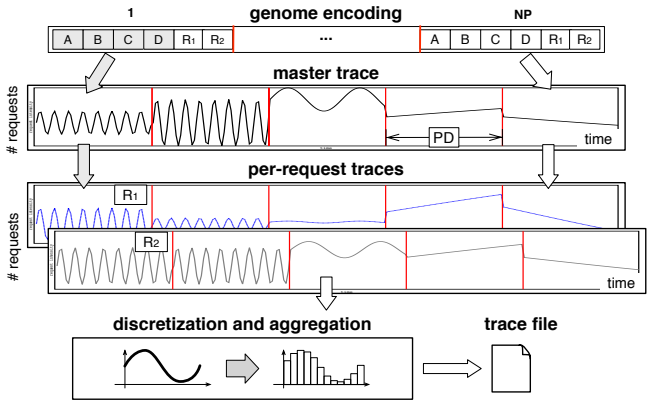

Fig. 2. Pattern-based test case encoding using the oscillating pattern

select a pattern in the sequence and update the value of some of its parameters; ii) *mutate request mix*: we select a pattern in the sequence and update the value of ratios parameters. All the values are constrained by specified validity ranges.

Initial population. Provided we have the population size, and the values for NP and PD, we build the initial population by randomly initializing the pattern of the individuals.

Fitness function. The fitness function is at the heart of the search process and guides the evolution of the population towards the most suitable solutions. For testing purposes, the actual fitness function to be used depends on the test goal that must be achieved. Nevertheless, for the case of testing elastic systems we identify three core concepts that should be considered when building a fitness function: 1) Promote tests that trigger system elasticity, i.e., both system expansions *and* contractions. This can be captured by counting the occurrences of changes, their direction and their entity, i.e., the difference in the amount of resource used. 2) Promote tests that cause system misbehaviors, such as performance issues, or that result in system behaviors that are close to the limit. This can be captured by tracking the number of times the behavior of a system is out of range of the expected behavior, and how much the behavior deviates (severity distance metric). 3) Promote tests that use few resources to stress the system. This can be captured by counting the total amount of generated requests.

Parent selection. We use the Roulette Wheel selection method to select more frequently fitter parents [5].

Recombination operators. We enable elitism and we use one- and two- point cross-over to recombine parents.

Termination criteria. We default the termination criteria to the maximum number of iterations.

V. Combining Search and Models

The fitness function depends on the system behavior, so we need to generate and execute traces at every search round. Unfortunately, in our context this process is impractical because each test may take a long time to complete, and there are several tests to run at each round. To overcome this limitation we take inspiration from model-driven design optimization and propose to use *surrogate models* within the search cycle [22].

Surrogate Models. The key idea is to use specific models as "surrogate" of an ES – hence their name. Surrogate models employ powerful techniques to correlate and interpolate data, for solving either classification or regression problems. Common examples of surrogate models are artificial neural networks, support vector machines, Bayesian networks, regression trees and Kriging models (see details in [25])

The feasibility of our approach relies on the assumption that we can find suitable surrogate models that 1) simulate/predict the behavior of elastic systems and 2) reflect on the (un)certainty of their own predictions. Among the available models we deem Kriging models and Gaussian processes (GP) [21] as suitable models for implementing the proposed test case generation for ES: They are proven to accurately capture the behavior of elastic systems, and they natively provide a confidence measure for prediction quality [7], [8].

In this paper, we argue that extensions of Kriging and GP models for time series analysis [4] and uncertainty propagation [9] fulfill the requirements on accuracy and uncertainty measure that our test case generation demands. These surrogate models are able to predict the system behavior under different conditions, e.g., under varying load, in a matter of (milli-)seconds [23]. Therefore, the efficiency of the overall process (see Figure 3) is greatly improved.

Model-Enhanced Search. We evolve the population according to the *expected* system behavior, as predicted by the models, and not according to the real one. In Figure 3, we mark the fitness function computed on the expect behavior with a star symbol. This raises questions about models accuracy and search effectiveness. If models are inaccurate, the search may point towards test cases that are *presumed* fitter while they are not. We address these concerns by letting the search execute some, but not all, the test cases on the real system. By collecting data from real executions, we can achieve a double goal: 1) evaluate the true fitness function, and 2) improve the surrogate models. At the same time, by executing only a small fraction of the tests, we maintain the search process efficient.

To decide which tests to execute, a trade-off accounts for the model inaccuracy and the expected improvement of the fitness function, as described by Jones et al. in the domain of design optimization [17]. In essence, the system should run only tests that are promising, i.e., expected to be fitter according to our incomplete knowledge of the system. A test case is promising if the model is highly accurate and predictions show that the system will behave differently from its specifications. Conversely, a test is promising if the model has broad confidence intervals, and there is chance to discover unexpected behavior that the surrogate model inherently cannot capture (given the limited knowledge of the system).

At the beginning of the search, we may have very limited knowledge, and we may need to execute several tests. However, as the search continues to improve the model, less and less executions are needed because at every execution our knowledge of the system increases.

In summary, we use surrogate models to improve the efficiency of the search process, effectively making it practical for long test traces geared to the target ES; conversely, we use the search process to improve the accuracy of surrogate models only where its predictions are more critical.

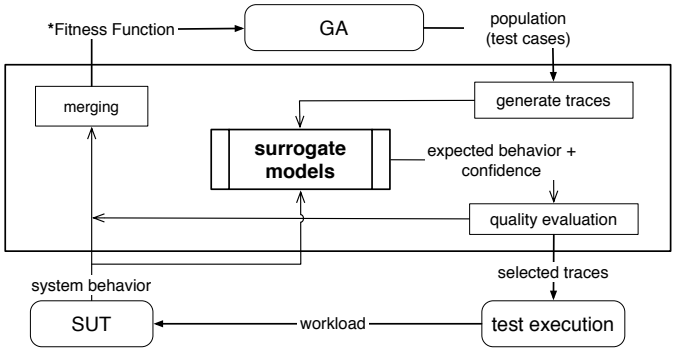

Fig. 3. Model-driven search-based generation of test cases

VI. Related Work

Considerable body of work consider meta-heuristic techniques for functional testing [2] as well as non functional testing [1]. For brevity reasons, here we can discuss only a limited subset of them.

Di Penta et al. [6] use search-based optimization to generate test data that cause service level agreement (SLA) violations in service oriented systems. We share a similar test goal, that is, to find valid tests that induce system misbehaviors in a black-box style. Consequently, our fitness functions contain similar attributes, and demand monitoring data from test executions. To reduce the amount of test executions, we propose surrogate models to generalize from single executions.

Surrogate models are used to predict the value of an unknown function starting from observations, and are commonly used for model-assisted design optimization in several engineering fields. For example, Gramacy et al. use them to optimize the design of the NASA re-entry vehicles [10], while Mariani et al. use them in the design of multi-processor systems-on-chip [20].

Our search is toward specific distributions of requests in time that is similar to what Briand and coauthors proposed for stress testing real-time applications [5]. In our approach, we need a running system and we use surrogate models to drive the search while Briand et al. refer to the structure of tasks to be executed and do not require a running version the SUT. Similarly, Hänsel et al. [11] propose meta-heuristic to generate test cases in the form of timed traces. *Blocks* serve as the basic construct for compacting long traces into short genomes. We propose pattern that are similar to blocks but encode also a functional form of requests distribution.

Surrogate models are black-box models of the system behavior, and we use them to derive tests that encode environment evolution. Our *test oracle* evaluates test results using systems outputs and end-to-end behavior, and does not consider system interactions with the environment. In this sense, our approach is specular to the one proposed by Iqbal and coauthors [15] that use models of the environment to generate black-box tests, and that define test oracles in terms of the expected interactions between the SUT and environment.

VII. Conclusions and Future Directions

We position our recent work in the context of search-based software testing: We introduce the problem of testing elastic systems, formalize the main point of the search problem, and discuss how we combine search and surrogate models.

We are currently implementing the software framework to support our approach, and evaluating its feasibility and applicability with elastic systems in our Cloud. In the near future, we will investigate the use of models to filter out invalid and infeasible tests and also to limit the occurrence of tests that are too similar to the ones already considered in the process.

References

[1] W. Afzal, R. Torkar, and R. Feldt. A systematic review of search-based testing for non-functional system properties. *Inf. Soft. Tech.*, 51(6), 2009.
[2] S. Ali, L. Briand, H. Hemmati, and R. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Trans. on Software Eng.*, 36(6):742–762, 2010.
[3] E. M. Arruda and M. C. Boyce. A three-dimensional constitutive model for the large stretch behavior of rubber elastic materials. *Journal of the Mechanics and Physics of Solids*, 41(2):389 – 412, 1993.
[4] S. Brahim-Belhouari and A. Bermak. Gaussian process for nonstationary time series prediction. *Comp. Statistics & Data Analysis*, 47(4), 2004.
[5] L. Briand, Y. Labiche, and M. Shousha. Stress testing real-time systems with genetic algorithms. In *GECCO'05*, pages 1021–1028, 2005.
[6] M. Di Penta, G. Canfora, G. Esposito, V. Mazza, and M. Bruno. Search-based testing of service level agreements. In *GECCO'07*, 2007.
[7] A. Gambi, G. Toffetti, and M. Pezzè. Assurance of self-adaptive controllers for the cloud. In *Assurances for Self-Adaptive Systems*. 2013.
[8] A. Gambi, G. Toffetti Carughi, C. Pautasso, and M. Pezzè. Kriging controllers for cloud applications. *Internet Computing*, (accepted), 2013.
[9] A. Girard, C. Rasmussen, J. Quinonero-Candela, R. Murray-Smith, O. Winther, and J. Larsen. Multiple-step ahead prediction for non linear dynamic systems–a gaussian process treatment with propagation of the uncertainty. *Advances in Neural Inf. Processing Syst.*, 15:529–536, 2002.
[10] R. Gramacy, H. Lee, and W. Macready. Parameter space exploration with gaussian process trees. In *ICML'04*, pages 45–52, 2004.
[11] J. Hänsel, D. Rose, P. Herber, and S. Glesner. An evolutionary algorithm for the generation of timed test traces for embedded real-time systems. In *ICST'11*, pages 170–179, 2011.
[12] M. Harman. The current state and future of search based software engineering. In *FOSE'07*, pages 342–357, 2007.
[13] W. Hummer, C. Inzinger, P. Leitner, B. Satzger, and S. Dustdar. Deriving a unified fault taxonomy for distributed event-based systems. In *Int. Conf. on Distributed Event-Based Systems, DEBS*, pages 167–178, 2012.
[14] W. Hummer, O. Raz, and S. Dustdar. Towards Efficient Measuring of Web Services API Coverage. In *PESOS Workshop at ICSE'11*, 2011.
[15] M. Z. Z. Iqbal, A. Arcuri, and L. C. Briand. Empirical investigation of search algorithms for environment model-based testing of real-time embedded software. In *ISSTA'12*, pages 199–209, 2012.
[16] S. Islam, K. Lee, A. Fekete, and A. Liu. How a consumer can measure elasticity for cloud platforms. In *ICPE'12*, pages 85–96, 2012.
[17] D. Jones, M. Schonlau, and W. Welch. Efficient global optimization of expensive black-box functions. *J. of Global Optim.*, 13:455–492, 1998.
[18] D. Kossmann, T. Kraska, and S. Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *ACM SIGMOD International Conference on Management of data*, pages 579–590, 2010.
[19] S. J. Malkowski, M. Hedwig, J. Li, C. Pu, and D. Neumann. Automated control for elastic n-tier workloads based on empirical modeling. In *ICAC'11*, pages 131–140, 2011.
[20] G. Mariani, G. Palermo, C. Silvano, and V. Zaccaria. Meta-model assisted optimization for design space exploration of multi-processor systems-on-chip. In *Euromicro DSD'09*, pages 383–389, 2009.
[21] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning*. The MIT Press, 2006.
[22] J. Sacks, W. Welch, T. Mitchell, and H. Wynn. Design and analysis of computer experiments. *Statistical science*, 4(4):409–423, 1989.
[23] G. Toffetti, A. Gambi, C. Pautasso, and M. Pezzè. Engineering autonomic controllers for virtualized web applications. In *ICWE'10*, 2010.
[24] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *STVR*, 22(5):297–312, 2012.
[25] G. G. Wang and S. Shan. Review of metamodeling techniques in support of engineering design optimization. *Mechanical Design*, 129(4), 2007.