Testing the Untestable*

Model Testing of Complex Software-Intensive Systems

Lionel Briand, Shiva Nejati, Mehrdad Sabetzadeh, Domenico Bianculli
SnT Centre - University of Luxembourg, Luxembourg, Luxembourg
{lionel.briand, shiva.nejati, mehrdad.sabetzadeh, domenico.bianculli}@uni.lu

ABSTRACT

Increasingly, we are faced with systems that are *untestable*, meaning that traditional testing methods are expensive, time-consuming or infeasible to apply due to factors such as the systems' continuous interactions with the environment and the deep intertwining of software with hardware.

In this paper we outline our vision to enable testing of untestable systems. Our key idea is to frame testing on *models* rather than operational systems. We refer to such testing as *model testing*. Our goal is to raise the level of abstraction of testing from operational systems to models of their behaviors and properties. The models that underlie model testing are executable representations of the relevant aspects of a system and its environment, alongside the risks of system failures. Such models necessarily have uncertainties due to complex, dynamic environment behaviors and the unknowns about the system. This makes it crucial for model testing to be uncertainty-aware. We propose to synergistically combine metaheuristic search, increasingly used in traditional software testing, with system and risk models to drive the search for faults that entail the most risk.

We expect model testing to bring early and cost-effective automation to the testing of many critical systems that defy existing automation techniques, thus significantly improving the dependability of such systems.

1. OVERVIEW, MOTIVATIONS, AND AIMS

Although testing is arguably the most prevalent verification and validation (V&V) technique, for many systems that we label as *untestable*, testing is impossible or highly expensive to automate. An example of untestable systems, discussed in more detail below, is the control systems in automobiles. A more complex example is a fleet of autonomous but communicating vehicles that are integrated into an urban smart traffic grid. Specific challenges in untestable systems include automated generation and execution of test cases and identification of failures. In this paper we outline our vision *to bring test automation* to untestable systems *by raising the level of abstraction at which testing is performed.*

We envision to raise the level of abstraction of testing by shifting the bulk of testing from implemented systems to *models* of such systems. In our context, models represent relevant aspects of system behavior, environment, structure, and properties, and are used as the basis for test execution and analysis. We refer to such testing, that is, test execution on models, as *model testing*. Our goal is to bring about advancements in test case generation, execution, evolution, and failure detection, to develop a new set of technologies that reconceptualize the foundations of software testing and offer a novel testing paradigm.

Automated testing aims to provide confidence about system dependability, within reasonable effort, by identifying and exercising a small fraction of the execution space where faults are more likely to lie. However, many complex systems are nearly untestable due to factors such as hardware constraints, the lack of a precise understanding of the expected system behavior and the complex time-dependent properties that these systems must satisfy. These factors often prevent the full automation and execution of large numbers of test scenarios over the implemented system, e.g., when the factors necessitate that the hardware be manually set up before each test execution. Model testing tackles the untestability challenge by enabling software engineers to (1) execute and validate a much larger number of test scenarios, (2) develop test strategies that carefully select which scenarios should be also executed on the deployed system depending on the level of risk they exhibit, and (3) specify automated oracles, i.e., mechanisms for detecting failures.

Modeling is by no means new to the V&V community and is already the cornerstone of a number of well-studied V&V techniques. *Model checking*, which is a formal verification technique over concurrent finite state models, has a long history of application in software and hardware quality assurance [3]. *Model-based testing*, which relies on models to generate test scenarios and oracles for implementation-level artifacts, has also shown promise and is gaining traction in industry [4]. For many untestable systems such as Cyber-Physical Systems (CPSs), however, these approaches suffer from the problems of scale and practicality. Model checking focuses on exhaustive exploration of model executions and is often hindered by the state explosion problem [3]. This scalability challenge is further exacerbated when one has to account for the properties of systems involving physical devices with continuous dynamics and complex,

*This work has been partially supported by the National Research Fund, Luxembourg (FNR/P10/03).

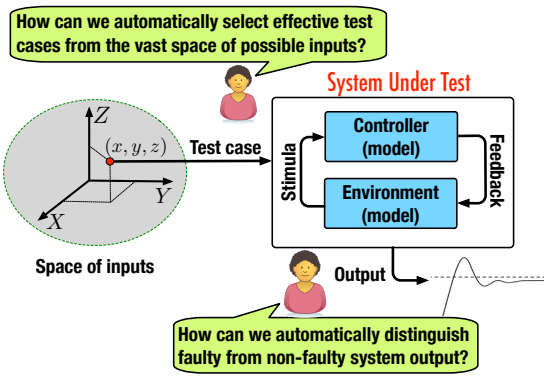

Figure 1: Key test automation needs for an untestable system from the user's perspective

concurrent interactions between the system and its environment (networks, devices, and people). As for model-based testing, the shortcomings are largely due to the fact that testing implementation-level software in fully realistic conditions and over actual hardware may be infeasible (e.g., when the hardware is developed in tandem or after the software), extremely expensive (e.g., when the hardware may quickly wear out or sustain damage during testing) or highly time-consuming (e.g., when the hardware or the environment reacts at a much slower rate than software). Having learned from the successes and drawbacks of both model checking and model-based testing, our vision is to introduce and operationalize the paradigm of model testing. Specifically, our goals are to:

- enable software engineers, in the context of untestable systems, to validate large numbers of test execution scenarios by means of model executions;
- help engineers define testable models, i.e., models that enable the selection and execution of test scenarios at an adequate level of detail for failure detection while accounting for uncertainty in the system and environment behavior;
- develop techniques that combine metaheuristic search techniques (such as evolutionary computing) with testable models, to automate the identification of cost-effective test scenarios;
- provide means to select execution scenarios that should be replicated on the system implementation depending on the level of risk they exhibit, i.e., the probability and level of damage.

2. USER'S PERSPECTIVE

To illustrate untestable systems, we use a highly-simplified example of a controller system from the automotive domain. Figure 1 provides an overview of this controller and its environment in a feedback loop. The large majority of controllers in the automotive domain, including the one in our example, are designed using differential equations over continuous time. They are typically developed as executable MATLAB/Simulink [6] models (a de facto industry standard) allowing engineers to simulate the controllers, analyze their outputs and eventually generate code from the controller models.

Such controllers are untestable because of the characteristics of their inputs and outputs. Specifically, the controller input, which is comprised of several time-continuous variables and calibration parameters, gives rise to multi-dimensional and large input spaces containing in the order of 10^{100} individual input points even for the simplest controllers. The controller output is a function over time (signal). To verify the controller's behavior, engineers have to evaluate several aspects of the output signals, particularly whether the signal reaches appropriate values at the right time instants, whether the time period that the controller takes to change its values is within acceptable limits, and whether the signal shape is free of erratic and unexpected changes that violate continuous dynamics of physical processes or objects. To perform such evaluation, engineers have to evaluate the changes in the output over a continuous time period. In contrast, existing software testing approaches focus on discrete outputs, evaluating outputs at a few discrete time instances (states) and essentially ignoring the output changes in between the states. As a result, the current practice on testing controllers is highly manual, both on test generation and detection aspects, thus entailing incomplete and highly-expensive testing.

Although computation and software are integral parts of continuous controllers, such controllers lie in unchartered territory as far as software V&V is concerned, primarily because existing V&V methods do not adequately address the scalability issues and the physical and timing aspects of these systems.

For untestable systems such as the controller in Figure 1, we envision model testing to provide automated support for two key tasks, among others, that engineers have to handle on a recurring basis: (1) selecting effective test cases from the vast space of possible inputs and (2) automatically distinguishing faulty from non-faulty system outputs. As a result, model testing will enable much more effective fault detection, higher dependability, and lower operational risks.

3. BACKGROUND & STATE OF THE ART

The notion of untestable (non-testable) system was originally coined by Elaine Weyuker [10] to refer to systems where test oracles cannot be defined. Nevertheless, there are several other factors driving untestability that need to be addressed, such as the size of the input space, the complexity of inputs and outputs, and the complex interaction with the physical environment. Metaheuristic search has been applied to deal with large system input spaces [8] but most of the existing work focuses on unit testing. Scalability remains an open issue for large systems since, in practice, there are limits to the amount of testing that can be performed on the implemented system [4].

Other contributions have relied on modeling to simulate the system environment [5], or to perform model-in-the-loop testing [9], where both the system and its environment are simulated. The executable modeling languages used for design-time simulation so far do not rely on standardized modeling languages, thus preventing software engineers from exploiting existing design artifacts and industry-strength design tools. Furthermore, when such exploitation is possible (e.g., in the case of approaches [9] based on Simulink models), the supporting technology fails to address the system-level testing of heterogeneous software systems because it lacks the capability of handling complex properties like continuous behaviors [7] or uncertainty [1].

Partial oracles, which do not guarantee the correctness of an execution but simply detect the presence of specific

failures, are useful when systems are untestable [10]. One important category of partial oracles are metamorphic relations among test outputs [2]. Despite their usefulness, they are limited in scope and can mostly be applied in the context of mathematical functions.

To summarize, what is missing from the state of the art is a comprehensive solution for test automation in the context of untestable systems, such that cost-effective testing strategies can be devised and test oracles, whether exact or heuristic, can be defined for failure detection and analysis over such systems. Model testing will tackle this gap and expand the scope of application of automated testing to new domains and systems.

4. RESEARCH CHALLENGES

Fulfilling our vision for model testing requires addressing the following challenges.

Definition and execution of testable models. We refer to a model as "testable" if it enables (1) the execution of test case scenarios, (2) the selection of such scenarios according to cost-effective strategies, and (3) the automated detection of failures or the evaluation of risks during execution. As an example, when dealing with CPSs, test automation necessarily has to consider the interaction between these systems and their (physical) environment. This environment must also be modeled to some extent in order to enable its simulation. Failure detection and cost-effective test scenario generation are possible only by checking the environment's dynamics during and after test executions. Developing proper formalisms for expressing testable model and suitable simulation algorithms depends on the system under test, the nature of the environment the system interacts with, and the types of failures targeted by testing. One challenge to this end is in providing precise guidance to software engineers, based on rigorous scientific investigation, in the selection of modeling methodologies to use for enabling model testing. The execution of a testable model in many large-scale systems is also complicated by the heterogeneity of the system components. The heterogeneity may be caused by the components having been modeled in different formalisms (e.g., differential equations, state machines or automata) or the need to account for legacy and third-party components whose implementation is not available (e.g., commercial off-the-shelf components). Another challenge is then to enable effective model testing when we have only a partial understanding of the system components, and are thus uncertain about their behavior.

Automated detection of failures. Automated testing is significantly hampered by the oracle problem, i.e., the capability to automatically detect execution failures. For untestable systems, this problem is often exacerbated by the dynamic and physical properties of the environment over time, and by the fact that engineers are typically unable to conclusively distinguish correct from incorrect system behaviors at the time of testing. This entails particular challenges for test automation. First, test oracles are not as clear-cut as in other systems because they also involve assessing a probability of failure and the risks involved, based on the extent of deviation from what is targeted with respect to dynamic and physical properties. Further, test oracles must take into account the effects of possible uncertainty in the models, and therefore raising the need for probabilistic oracles. In other words, the challenge is to devise clear methodologies and guidelines for defining different kinds of oracles for continuous and discrete systems properties, accounting for different types of uncertainties, and mechanisms to rank test executions according to risk models.

Automated test selection and generation. The space of possible test scenarios and interactions with the environment is usually far too large to be amenable to simple test strategies. Since many untestable systems are safety- or business-critical, any effective test strategy has to be geared towards the identification of worst-case scenarios or scenarios that entail the highest level of risk. Developing test strategies of such nature presents certain challenges. First, there must be a domain-specific risk model associating a level of risk with any feasible test scenario — a problem that has received little attention so far. Second, searching for worst-case scenarios usually requires evaluating, in a quantitative fashion, how "bad" test scenarios are. This is in many cases extremely expensive from a computational angle as such quantitative analysis involves repeated model executions. A challenge is therefore to find effective heuristics for test selection and generation to scale. A final challenge is about devising strategies, based on the above risk analysis, to select test cases to run on the implemented system.

Regression testing. Many systems are subject to frequent changes in hardware, regulations, or assumptions about the environment in which the system will operate. Changes entail, among other things, expensive re-testing which needs to be minimized. Regression testing aims at ensuring that changes do not lead to undesirable side effects in the unchanged parts of the system. For untestable systems, regression testing is more challenging because of more complex dependencies (e.g., captured by differential equations), heterogeneity, and thus increased uncertainty in determining the impact of changes on regression test cases.

Assessing the cost-effectiveness of model testing. With testing strategies being heuristic in nature, one question is how to evaluate their cost-effectiveness. In the context of model testing, we need to determine, through empirical studies, how to make testing affordable and the extent to which we can rely on model testing results as an indicator of the dependability of the implemented system. Such empirical studies are crucial for establishing a roadmap to support the adoption of research results in practical settings. A challenge is therefore to define appropriate empirical methods to assess, in a credible manner, the benefits of model testing in realistic contexts, where models cannot be expected to be fully complete or accurate.

5. OBJECTIVES

To address the challenges discussed above, a fundamental shift is needed in the way we address test automation. Our overall vision for *model testing* is *to move the bulk of system testing to a higher level of abstraction* where automated testing is performed on executable models of the system under test and its environment. Our rationale is that, with executable models of the system and its environment, one can run a large number of test scenarios in a completely automated fashion, while accounting for uncertainty, and at a chosen level of details for detecting specifically-targeted faults. This enables us to explore the system input space to a much larger extent than when testing the implemented system, thus increasing our chances of detecting high-risk scenarios, e.g., triggering highly damaging failures. In addi-

tion, models are expected to contain information that can help guide the test selection and generation process, along with test execution results and the derivation of test oracles. Finally, model testing offers opportunities for performing early testing, concurrently with the system implementation, and for guiding later stages of testing on the implemented system.

We plan to realize our vision by achieving the following objectives:

Executable and testable models of the system and its environment. Appropriate methodologies to build executable and testable models of systems and their environment are necessary. Models should be detailed enough to execute test scenarios at a level of detail required to check key properties acting as test oracles, for example time-dependent state changes. At the same time, modeling should remain as simple and cost-effective as possible by focusing on what is required for checking the properties of interest. Facilities to define test oracles should be provided to help the testers choose appropriate properties based on the models. Different types of systems and domains will require different methodologies fitting the types of properties that are relevant in their context.

Finding, executing, and analyzing high-risk test scenarios. Assuming that a large number of test scenarios can be executed at the required level of detail to identify relevant violations of key system and environment properties, we can explore the system input space to the extent permitted by the computational power at our disposal. To be effective at finding high-risk scenarios in a large input space, we first need a clear, context-dependent definition of risk, which assesses some relevant form of expected damage or loss due to failure. We then need efficient search algorithms to explore the input space and converge towards higher-risk areas. Test execution results should also be analyzed to aid fault localization in models as a way to ensure that engineers can quickly identify the cause of failures.

Modeling, uncertainty and risk. There are typically many sources of uncertainty in what we know about a system and its environment, at early stages of development and even at system-testing time. In addition, there may be a degree of non-determinism to contend with. Such uncertainty needs to be modeled by associating probability distributions with critical events, their time of occurrence, and the results of system operations. The resulting distributions then need to be used by the test execution engine to produce, e.g., probability distributions of outputs and key property values, instead of exact values. These would in turn lead to fitness value distributions during the search, which would have to be handled by the search algorithms. Risk analysis of test results would also have to account for their probability distributions when assessing test scenarios.

Guiding testing on the implemented system. Assuming a certain test budget for testing the implemented system, we would like to use the results of model testing to select an optimal subset of test scenarios that need to be executed on the implemented system to further assess its level of risk in fully realistic conditions. A strategy could be to select, based on a careful risk analysis of model testing results, the optimal subset of test scenarios that together capture the largest level of system risk.

Change impact analysis and regression testing. In addition to supporting test automation, models are essential for automatically assessing what test cases are affected and need to be re-run when a change occurs, e.g., in the systems requirements or environmental assumptions. We can achieve this by analyzing model changes, and the dependencies between model elements, test results and oracles. Such analysis can be static and based on analyzing the models themselves, or dynamic and based on analyzing the model execution traces resulting from model testing.

6. EXPECTED IMPACT

The vision proposed in this paper is expected to have a significant impact on both the state of the art and the state of the practice. Model testing will enable the application of automated testing to systems that would otherwise be untestable. The domains where the project outcomes will be directly applicable are numerous and include automotive (e.g., collision avoidance systems), healthcare (e.g., medical devices), manufacturing (e.g., robotic assembly lines), telecommunications (e.g., satellite communication systems) and electronic finance (e.g., credit card transaction processing). We anticipate that model testing, and the guidance that it will provide for later stages of testing over implemented systems, will contribute significantly to reducing overall V&V costs and improving the dependability of large-scale software-intensive systems. Such a research direction will greatly expand the scope of software testing research, which, to date, has focused almost exclusively on implemented systems, with an over-representation of techniques for testing-in-the-small.

7. REFERENCES

[1] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola. Self-adaptive software needs quantitative verification at runtime. *Commun. ACM*, 55(9):69–77, 2012.

[2] T. Y. Chen, T. H. Tse, and Z. Q. Zhou. Fault-based testing without the need of oracles. *Information and Software Technology*, 45(1):1–9, 2003.

[3] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2001.

[4] H. Hemmati, A. Arcuri, and L. Briand. Achieving Scalable Model-Based Testing through Test Case Diversity. *ACM Trans. Softw. Eng. Methodol.*, 22(1):6:1–6:42, 2013.

[5] M. Z. Iqbal, A. Arcuri, and L. Briand. Empirical investigation of search algorithms for environment model-based testing of real-time embedded software. In *Proc. of ISSTA 2012*, pages 199–209. ACM, 2012.

[6] MathWorks. Mathworks MATLAB Simulink. `http://www.mathworks.com/products/simulink/`.

[7] R. Matinnejad, S. Nejati, L. Briand, T. Bruckmann, and C. Poull. Search-based automated testing of continuous controllers: Framework, tool support, and case studies. *Information and Software Technology*, 57:705–722, 2015.

[8] P. McMinn. Search-based software test data generation: A survey. *Softw. Test. Verif. Reliab.*, 14(2):105–156, 2004.

[9] H. Shokry and M. Hinchey. Model-based verification of embedded software. *Computer*, 42(4):53–59, 2009.

[10] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.