Section A

Title: Automated Composition of Heterogeneous Tests for Cyber-Physical Systems

Alex Groce and Paul Flikkema

School of Informatics, Computing, and Cyber Systems

Northern Arizona University, USA

November 28, 2017

1 Section B

2 Section C

2.1 Introduction

Cyber-physical systems (CPS) exhibit a number of characteristics that challenge current design paradigms and testing methods. Key characteristics inherent to deployed systems include the need for reactive or real time processing; networks that inject stochastic delays and loss of reliability in communication; nodes with strongly heterogeneous processing platforms; and brittle user interfaces. Growing demand for distributed intelligence and security will increase the complexity of CPS software. These characteristics in turn influence the CPS design and design processes through layering and the refinement of abstractions. To manage design complexity, CPS design employs layering in multiple domains, e.g., computation, networking, and the modeling of the embedding physical system and the system's sensors and actuators. However, current layering approaches do not capture non-functional system properties essential to CPS, e.g., timing and energy use, that emerge via testing. To manage design process complexity, iterative development is commonplace: while the long-term trend is refinement of abstract models, engineers often need to shift back and forth between implementation-level models and more abstract models to gather new data, gain knowledge and insight, and optimize system performance. The integration of effective testing into the design process will be central to the success of CPS in critical applications, but the question is how to do this.

The same multiplicity of layers also often applies to existing tests for CPS (and other embedded systems): often components of a system, such as a file system or actuators, are tested by one set of engineers, and using completely different methods than are used to produce tests at the high level of either control software with humans-in-the-loop or autonomous control systems. The core functionality of a CPS is usually written in low-level,

embedded systems languages, such as C. In the ideal case, such systems are developed using both formal specification and verification and sophisticated automated testing. In some cases the formal specification is used to generate executable tests to ensure the real system matches the formal models; in other cases there is at least a very determined test generation effort, including efforts to produce very high-coverage tests. In contrast, the user-centric or high-level autonomy aspects of a CPS are often developed in higher-level languages, such as Java, and with a much more informal approach to testing and verification. Increasingly, mission- and safety-critical low-level CPS elements interact with user-centric systems, in order to allow users more control. Even when the behavior of each system, in isolation, is valid, their composition may compromise performance, user experience or (in the worst case) safety.

We propose a two-part solution to the problems of using testing to guide CPS design, and ensuring that completed CPS implementations have effectives tests that validate a system not only at each layer, but in terms of unexpected interactions between layers. First, we propose the problem of automatically *composing tests*, including heterogeneous tests targeting different layers of a system. Often there exist tests for behaviors at different levels, and these have considerable value — but the combined behaviors are not well tested, and the tests cannot be combined. Second, we propose the development of architectural methods that facilitate such automated composition and explicitly represent the domain of a design, and the level of abstraction in that domain. The key point is to construct test architectures that enable CPS models for different domains at diverse levels of abstraction in a plug-and-play manner.

As an example of the practical aims of our proposed approach, it would be helpful to run the low-level tests in one domain in the context of other domains expressed using high-level models. For example, consider the following closed-loop scenario: a unit-level test of implementation-level code running on a target sensor/actuator (S/A) node connected to a meteorological model running in the cloud that drives an emulated transducer at the

S/A node, and, through a packet-layer communication link model, to a high-level control system model running in real-time on an engineer's workstation; this model in turn drives an S/A node actuator via commands sent through the communication link model. As another example, existing production-level, server-based control code may need to be integrated with a new actuating subsystem. Here, functional tests could be performed using a sequence of models (of increasing refinement) prior to integration of the target actuating hardware. Finally, in our running example in the paper, even the "same" component of a CPS may have multiple levels: we consider the case of a file system on a remotely controlled robot exploring another planet, which has both a NAND flash file system and a high-level interface that is commanded by humans. While the problem of composing tests across system layers is not unique to CPS or embedded systems, their tendency to combine human-facing interfaces with critical components that are more thoroughly tested makes them a primary target for such efforts.

2.2 Automated Composition of Tests

Even without a better system architecture, some of the problems of CPS design could be mitigated by making use of existing tests for layers of a system. These tests are often, particularly for the safety-critical system, extensive and useful; however, they typically fail to cover the interaction of the systems in any way. In addition to the basic problem that the interactions are not explored in existing tests, however, is a deeper problem: tests do not compose. Even within a single system, executing test `A` followed by test `B` seldom produces the desired union of behaviors (e.g., even covering all code covered by `A` or `B`). The actions of `A` often interfere with those of `B` (or vice versa): that is, some action in `A` is either illegal in a composed context, causing `B` (and thus the entire test) to become invalid, or disables some behavior of `B`, lowering test effectiveness. The ordering of test operations also matters: e.g., some actions in `A` must be before some actions in `B` to produce interaction, while other actions must be after some `B` action. For compositions of safety-critical and user-centric systems,

and heterogeneous systems in general, it would be highly desirable to be able to automatically produce tests that are valid, have as little interference as possible, and maximize the sum of behaviors from the composed tests. The inability to compose tests results in poorer testing, and thus more fault-prone and brittle systems.

While complete automation of test composition, in general, is impossible, we believe existing software testing algorithms, used in a novel way, could increase the composability of tests, even for heterogeneous systems. The widely known delta-debugging algorithms can be used to manipulate tests for producing "quick tests" for embedded systems [4]. In this proposal, we suggest that further generalizing delta-debugging can effectively automatically compose some (even quite heterogeneous) tests. The key concept is to let the delta-debugging algorithm remove portions of a constructed *hypothesis composition* to produce a test that has more behavior than the naïve composition, and ideally detects a fault in the composition of the systems [5].

As an example of the problem, consider the following situation, a simplified generalization of testing efforts at NASA's Jet Propulsion Laboratory, during the development of the Curiosity Mars Rover (Mars Science Laboratory project) [8, 7, 6]. The file system for the Curiosity Rover can be considered from two points of view. There is the low-level, embedded flash file system, implemented as (essentially) a library in C. There is also a higher-level process, essentially a file catalog, through which other components of the Curiosity software interact with the file system, and which is directly accessed by ground operations teams controlling the rover. The low-level file system, which interacts with the flash hardware, was extensively tested using both model checking and random testing, by a team of formal verification and software engineering researchers, who also developed the file system software. The high-level catalog process was also tested extensively. However, in this case the testing was primarily performed manually by systems engineers and the Curiosity QA team, using less formal and intensive approaches, due in part to a much more complex but more limited specification of correctness. Every catalog test also tests the underlying file system,

of course, but file system tests do not test the catalog. In practice, the two sets of tests exist completely separately: the catalog tests as Python scripts to issue commands, and the file-system tests as C programs or tools to generate tests. This separation means that the catalog cannot benefit from the more extensive tests produced for the low-level file system. In operation, some faults related to interaction of the catalog and the file system were discovered. We hypothesize that being able to compose high-level catalog tests and low-level file system tests might have detected some of these faults.

The basic technical approach is best explained using a variation of the motivating Mars Rover testing scenario. Naïve composition of tests for the file system and the catalog will not work. Low-level file system tests may include operations that change the file system state in a way that the catalog, which has sole control of the contents of the file system in most directories during normal rover operation, cannot handle. Consider the composition of test `A`, for the file system, and test `B` for the catalog. Neither `A+B` (composition with `A` followed by `B`) nor `B+A` will provide the desired testing functionality. If we execute `A` then `B`, there may be little interaction between systems, and `A` may produce an initial state that the catalog cannot handle. If we execute `B` then `A`, the much more extensive testing of the underlying flash file system provided by `A` will not impact the catalog behavior at all, resulting in even less interaction. How to interleave the behaviors, while avoiding actions in `A` that violate catalog constraints, is a challenge even for engineers well-versed in both systems. However, imagine that we construct a new test, `(A+B)`$\times k$, consisting of `A` followed by `B`, repeated k times. This test will also, due to interference (let us assume `A` violates a catalog constraint) tend to fail immediately without exposing a real fault. How can we avoid these problems?

Delta-debugging works due to the high probability that contiguous parts of a test are related: removing *chunks* of a test can eliminate many behaviors that are irrelevant or interfering. Given a test `a1.a2.a3.a4.a5.a6.a7.a8` that fails, delta-debugging might first determine if either of `a1.a2.a3.a4` or `a5.a6.a7.a8` fails; if so, it proceeds from either. If not, it increases the granularity of reduction, and considers candidates `a3.a4.a5.a6.a7.a8`,

`a1.a2.a5.a6.a7.a8`, `a1.a2.a3.a4.a7.a8`, and `a1.a2.a3.a4.a5. a6`, until no single component of the test can be removed without the test no longer failing.

Cause reduction [4, 3] modifies delta-debugging to reduce tests with respect to an arbitrary property of the test, not just failure. For example to produce very fast regression tests (called "quick tests"), automated tests can be minimized to find smaller tests that retain full code coverage. In past work, this approach produced highly efficient tests for real-world systems such as Mozilla's SpiderMonkey JavaScript engine and the YAFFS2 flash file system used in Android.

Both cause reduction and delta-debugging traditionally require as input a test that satisfies the property of interest, e.g., a failing test or one with certain coverage. However, this is not necessary. Given a test that does not fail (or provide some other useful property of a test), cause reduction/delta-debugging defines a search, based on removal of components, for a test that does meet the criteria. We propose to construct "base" compositions of tests (that do not provide useful testing), and then use cause reduction to search for a test that *does* provide useful composition of the tests. The search has a potential to succeed because in most cases the reason composition fails is interference, which can be avoided by removing the interfering parts of a test, leaving a good interleaving of test actions, made possible by the k repetitions in the base.

A concrete application of our approach to the NASA Mars rover file system testing would work as follows. First, construct the test `(A+B)`$\times k$ (where k is at least 2, and may need to be larger). We can start with small k and increase k if the search for a useful test fails, since the length of the initial composed test determines the cost of cause reduction. The multiple copies of `(A+B)` handle the need to interleave actions from A and B, when combined with cause reduction. If k is at least one more than the max of the lengths of `A` and `B`, then there is a possibility (though not a guarantee) for cause reduction to produce any needed interleaving of actions: removing all but the needed actions from each copy yields all interleavings of a single copy of `A` and `B`. The extra copy is required so that the interleaving can start with

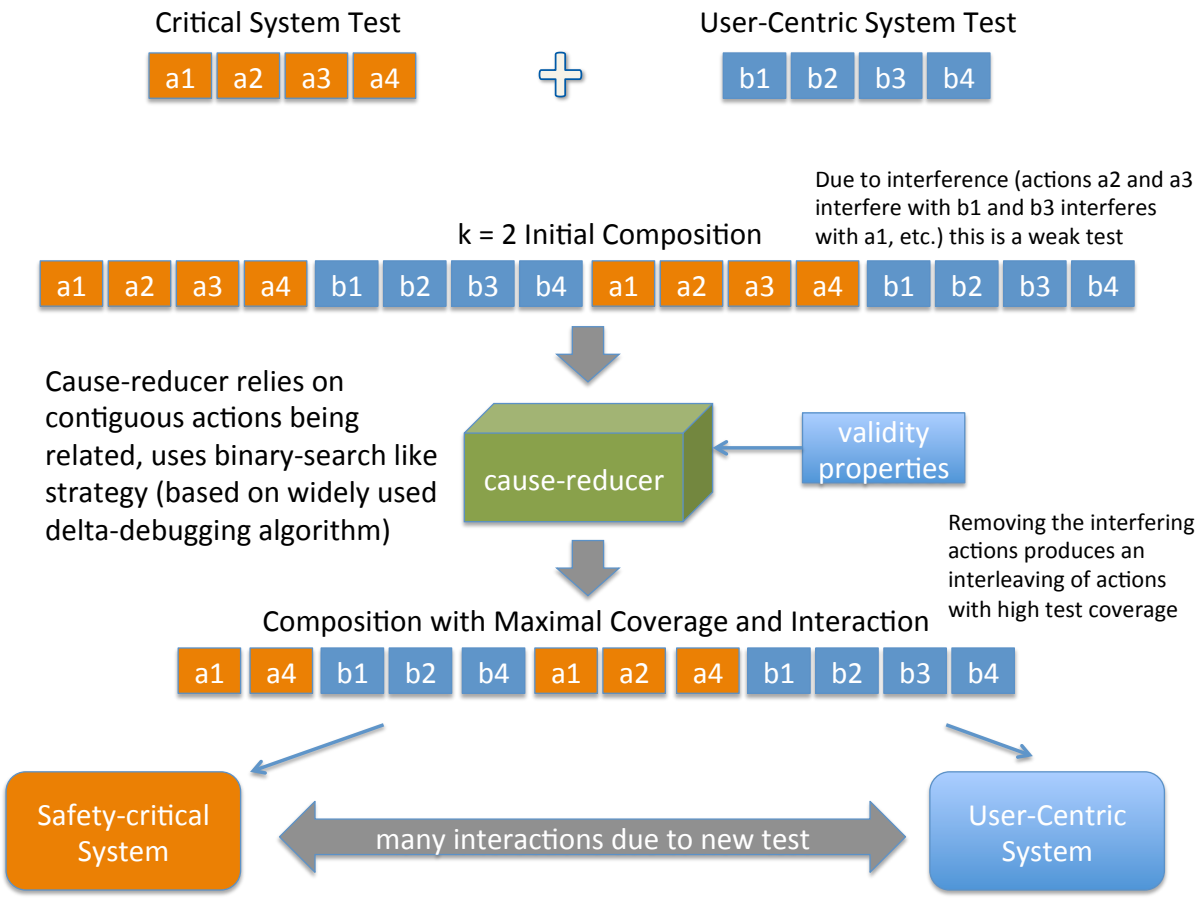

Figure 1: Composition of heterogeneous system test cases.

either of `A` or `B`. After constructing the initial, but usually not successful, "composition" `(A+B)`$\times k$, our approach applies cause reduction to `(A+B)`$\times k$, searching for a test that: 1) does not violate catalog invariants, so is a valid test, since a core problem of composition is the creation of invalid tests and 2) covers at least as much code as the union of code coverage for test `A` and test `B`. Alternatively, the search can be for a test that fails for any reason other than catalog invariant violation. However, in some cases, both of these searches may fail.

To understand the concept, consider the simple case where `A = a1.a2.a3.a4 and B = b1.b2.b3.b4`, with $k = 3$. If `a1` interferes with `B`, causing the catalog to fail with an invariant violated at action `b3`, then our approach can produce a test such as: `a2.a3.a4.b1.b2.b3.a1.a2.a3.a4.b1.a4`. Here, `a1` is removed from the copy of `A` before any `b3`, but remains in the final version,

from which all `B` actions are removed, because it adds new code coverage of the low-level file system. One `b4` instance is removed, because it causes the low-level file system code to be in a state such that the second copy of `B` exercises less code (it forces an early garbage collection of flash blocks). Notice that this test is not one that delta-debugging's binary search would have proposed from the initial test. Using gains in coverage to change the base test, we can direct the reduction toward this high-coverage, valid, composed test without human intervention. Figure 1 graphically shows the workflow of automated test case composition for a different set of tests.

We implemented a simple version of our approach in TSTL [9], and applied it to tests (with complete code coverage) for a simple data structure (an AVL tree). These tests were unable to detect a subtle, realistic, injected fault, despite covering the code. The naïve composition of all tests (`A+B+C+...`) was also unable to detect the fault, and included many invalid operations, due to interference. Our approach, with $k = 2$, was able to produce a test exposing the fault in only a few seconds, by removing interfering operations.

Because this approach suggests that test composition is essentially a search problem, an obvious question is why we use cause reduction/delta-debugging rather than a more traditional search-based evolutionary or genetic algorithm approach [1, 10, 2]. First, we believe that removal of operations is the only mutation of interest in this context: crossover or random change in test actions is likely to introduce invalid test behavior. Our assumption is that tests to be composed are valid in isolation, and only nearby behaviors are of interest (or likely to maintain single-component validity). Second, many search-based techniques expect access to branch distances and other intrusive instrumentation. This may not be feasible for embedded systems; we can use cause reduction with instrumentation only for user-centric code or, in the worst case, for neither system. This necessitates guidance by other means.

References

[1] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering*, 36:742–762, 2010.

[2] G. Fraser and A. Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In *ACM SIGSOFT Symposium and European Conference on Foundations of Software Engineering*, pages 416–419, 2011.

[3] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr. Cause reduction: Delta-debugging, even without bugs. *Journal of Software Testing, Verification, and Reliability.* accepted for publication.

[4] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr. Cause reduction for quick testing. In *IEEE International Conference on Software Testing, Verification and Validation*, pages 243–252. IEEE, 2014.

[5] A. Groce, P. Flikkema, and J. Holmes. Towards automated composition of heterogeneous tests for cyber-physical systems. In *Workshop on Testing Embedded and Cyber-Physical Systems*, pages 12–15, 2017.

[6] A. Groce, K. Havelund, G. Holzmann, R. Joshi, and R.-G. Xu. Establishing flight software reliability: Testing, model checking, constraint-solving, monitoring and learning. *Annals of Mathematics and Artificial Intelligence*, 70(4):315–349, 2014.

[7] A. Groce, G. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In *International Conference on Software Engineering*, pages 621–631, 2007.

[8] A. Groce, G. Holzmann, R. Joshi, and R.-G. Xu. Putting flight software through the paces with testing, model checking, and constraint-solving. In *Workshop on Constraints in Formal Verification*, pages 1–15, 2008.

[9] J. Holmes, A. Groce, J. Pinto, P. Mittal, P. Azimi, K. Kellar, and J. O'Brien. TSTL: the template scripting testing language. *International Journal on Software Tools for Technology Transfer*, 2017. Accepted for publication.

[10] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14:105–156, 2004.