Testing IoT Systems

Jeff Voas
Computer Secuirty Division
NIST
Gaithersburg, USA
jeff.voas@nist.gov

Rick Kuhn
Computer Secuirty Division
NIST
Gaithersburg, USA
rkuhn@nist.gov

Phil Laplante
Great Valley School of Graduate and Professional Studies
Penn State
Malvern, USA
plaplante@psu.edu

***Abstract*— This article presents challenges and solutions to testing systems based on the underlying products and services commonly referred to as the Internet of 'things' (IoT).**

Keywords—Internet of Things, testing, Domain Range Ratio

I. Introduction

Can you test the Internet? No - it is unbounded. Can you test the Internet of Things (IoT)? Same answer.

You could test sub-nets of the IoT and other bounded components of it. The Internet and its 'things' are only bound-able for mere instants in time, therefore testing is problematic. Testing systems-at-rest is easier than testing systems reorganizing themselves in real-time and at massive scale. The Internet at time zero is different than the Internet at time zero + *x*, where *x* is a millisecond.

We argue that testing the Internet and the IoT is not feasible. We further argue that we can use the concept of a Network of 'Things' (NoT) [1] to create testing schemes that are practical. This definition allows for measurement, and allows for one NoT to be compared to another. In addition, this definition allows for estimating the *testability*[1] of a specific NoT, which when said slightly differently, asks the question: is this NoT *testable*, meaning is testing even worth the effort you will put into it?

The general concept behind the term Network of 'Things' involves communication, computation, sensing, and actuation. These are simple ideas that have existed in distributed computing for years. But what makes IoT and NoT different from previous large-scale distributed computing systems is scale, heterogeneity, data integrity, sensing, and possible non-ownership of the assets in a purposed and proprietary NoT. By 'non-ownership' of assets, we include leased cloud services, leased data from vendor sensors, leased wireless communications, leased hardware and 3rd party software, and so on. This paper uses the concept of a NoT as the *entity under test*.

[1]Testability here refers to the likelihood that defects can be discovered during testing [3]; testability is clearly a function of what type of testing is occurring and how test cases are selected.

II. Understanding a network of 'things'

To address such concerns, the National Institute of Standards (NIST) Special Publication 800-183 [1] offered a scientific foundation to describe the underpinnings of a Network of 'things.' It breaks these four activities into core distributed system components termed "primitives." The document then defines a simple class of "elements" that allow for the foreshadowing of the trustworthiness of systems built from IoT-based components, services, and commercial products. (NIST has not released a specific definition for IoT at this time).

The primitives proposed in [1] are: 1) Sensor, 2) Aggregator, 3) Communication channel, 4) eUtility, and 5) Decision trigger. Here are their descriptions from the document:

1. A *sensor* is an electronic utility that digitally measures physical properties (e.g. temperature, acceleration, weight, sound, etc.) and outputs raw data.

2. An *aggregator* is a software implementation based on mathematical function(s) that transforms/consolidates groups of raw data into intermediate data.

3. A *communication channel* is a medium by which the data is transmitted (e.g., physical via USB, wireless, wired, verbal, etc.) between sensor, aggregator, communication channel, decision trigger, or eUtility.

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

4. An *eUtility* (external utility) is a software or hardware product or service, providing computing power that aggregators will likely network of ‘things’ have.

5. A *decision trigger* creates the final result(s) needed to satisfy the purpose, specification, and requirements of a specific network of ‘things.’

III. Testability of a Network of ‘Things’

A specific, purposed network of ‘things’ is likely to have a dynamic and rapidly changing dataflow and workflow. It will likely have numerous inputs from a variety of sources. This will, in turn, create a massive internal state space of data states created throughout the computational workflow of a NoT, and a vast number of potential interactions among components.

One way to think about the testing problems from this state-space explosion is by trying to answer the question: Do NoTs of large scale have an impact on testability? To answer that, we will first look at the Domain Range Ratio (DRR) [2][3] metric, first proposed in the 1990s by Voas and Miller.

We contend that the DRR metric addresses the inherent problem of testing networks that employ IoT-based components and services, e.g. clouds. The DRR is simply the cardinality of the set of all possible test cases for that system divided by the cardinality of the set of all possible outputs. A fundamental issue is that a particular network of ‘things’ is likely to process large amounts of data for the purpose of making rather limited output decisions, such as ‘actuate’ or ‘do not actuate.’ This situation makes it difficult to observe internal failures due to corrupted internal data during testing time.

For example, Fig. 1 represents a simplistic NoT purposed to buy or not buy a certain stock. (Figure 1 is not intended to represent real NoTs, but rather to highlight the primitives in [1].) This NoT has 15 sensors clustered into 3 groups, 5 aggregators, and 3 eUtilities (2 clouds, and 1 laptop). Note that Sensor 5 and Sensor 6 are blue, illustrating that they are sending out data of suspicious integrity. This NoT has 22 communication channels that carry the data that eventually gets aggregated and then fed into the NoT’s decision trigger. The decision trigger is binary –a value of ‘1’ means buy the stock, a value of ‘0’ means do not. Stated simply, the combinatorics of faults and internal failures that can go wrong with 15 sensors, 5 aggregators, 3 eUtilities, and 22 communications channels is quite large.

Now assume that each binary value of the decision trigger variable is obtained approximately 50% of the time. Because of this minimal output space size, a fair coin toss also has a 50-50 chance of providing a correct output for any given input. Hence building a system that generates a random ‘1’ or ‘0’ result via a coin flip is equivalent to and cheaper than building this complex and expensive NoT. Worse, consider the scenario where ‘1’ and ‘0’ are not evenly distributed, e.g., the specification states that for 1 million unique test cases only 10 should produce a ‘1’ and the other 999,990 should produce a ‘0’. One could build a NoT to compute this function or one could write a piece of code that just says: **for all inputs output (‘0’)**. This incorrect code is still 99.999 reliable, and it would be nearly impossible to discover the defect in the code with a handful of random tests sampled from the 1 million. In short, random testing here has a minimal probability of detecting this fault because each test case has a very low probability of revealing the defective logic due to the tiny output space, and its probability density function for each output. Note also that this argument likely applies to aggregators – if an aggregator is fed much sensor data and reduces that data to a single output value, in particular a binary value, this problem is the same.

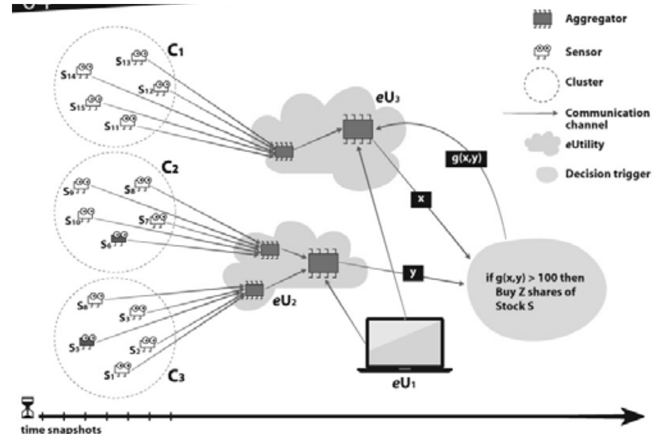

Fig. 1. A Simple NoT With a Feed-Back Loop

Furthermore, in this system, corrupted data (regardless of the reason for the corruption) can travel through any of the communication channels – it can originate from eUtilities, aggregators, sensors, and even communication channels. Given the many data-related events that happen before a decision trigger is executed, how does a system-level test of a NoT give assurance and confidence to the prior event’s trustworthiness? This is the same question that resulted in the concept of software unit testing. We apply the same principles here to NoTs. We will discuss this situation further using assertions.

IV. The Oracle Problem for Networks ‘Things’

The traditional software testing problem of having access to a usable oracle can be paraphrased as follows: if you do not know if an output is correct after a test is performed, what is the point of testing? Further, this problem can be subdivided into two problems: (1) a defective oracle, and (2) not having an oracle at all.

For NoTs, the oracle problem is exacerbated by this: it is unlikely that the intended functionality of a general-purpose, short-lived NoT will remain static long enough for an oracle to be built. (Hopefully for most security-critical and safety-critical properties of NoTs, this concern can be avoided.) This problem exists because NoTs offer more *extensibility* and *malleability* of the intended functionality than in previous distributed systems. For example, the sensors, eUtilities, and communication channels can all be quickly swapped out and replacements swapped in; and the algorithms and the software (in the aggregators, communication channels, and decision trigger) can be continuously tweaked when the purpose of a NoT changes.

The ability to “modify-on-the-fly” a NoT is one of the advantages of this advancement in distributed computing and control but is problematic for testing using oracles. Hence, testing efficiency is critical. Using the Template

V. Assertions

Three conditions are necessary for software to fail: (1) a fault is executed, (2) a defective internal data state is created, and (3) the defective state propagates causing incorrect output (failure). By testing internal data states using *assertions*, we observe whether (2) occurs if the assertion is correctly coded and placed.

Assertions are internal self-tests that probe either: (1) the output of a component or function, or (2) data states that exist anytime during computation. Assertions increase testability by increasing the size of the range [4][5]. The goal is not to test the functionality of eUtilities, aggregators, sensors, and communication channels, but rather to test the data they produce.

To do this, the notion of wrappers connected to interfaces (between primitives) is possibly the easiest implementation approach for building assertions. For example, in the interface between a sensor and an aggregator, insert a wrapper on the data leaving the sensor before it rides on the communication channel, or insert a wrapper before it leaves the communication channel and enters the aggregator, or both (if there is concern something might go wrong during transmission). Tests at the interface points offer two advantages for NoTs: (1) this testing can be done even if most of the primitive ‘things’ are black-box entities, and (2) no access to any software or algorithms is required.

VI. Combinatorial Testing

Among the unique challenges of testing NoTs, one of the most significant is the potential for an enormous number of interactions. Instead of two, or a few, components sending and receiving data, NoTs may have 10s or 100s of nodes interacting. While internet e-commerce and information systems include thousands of nodes, interactions are typically client-server. NoTs, in contrast, may require cooperation among a much larger set of nodes to meet their design objectives. The difficulty of testing these networks has led to recognition of the need for combinatorial test methods [12][13], which are designed specifically for testing complex interactions.

Software faults may involve one or multiple factors interacting. For example, a device failure that occurs only when $pressure < 10$ AND $volume > 300$ AND $velocity = 5$ (where *pressure*, *volume* and *velocity* are integer variables) is a 3-way interaction fault. Interaction faults remain dormant until the particular combination of values is encountered in practice. Combinatorial testing (CT) is an increasingly popular method for reducing the cost of finding such complex faults. The empirical basis for CT’s effectiveness was shown in a series of NIST studies [6][7][8][9] that demonstrated the following: most faults involve a single parameter or two parameters; and progressively fewer interaction faults involve 3, 4, 5, and 6 parameters (a fault involving more than six parameters has not been seen) [8]. This empirical finding, referred to as the *interaction rule*, has important implications for software testing, because it means that compressing t-way combinations into a small number of tests can provide more efficient fault detection than conventional methods.

Matrices known as *covering arrays* [10] are used to produce tests covering t-way combinations of values, for some specified level of $t \geqslant 2$; e.g., if $t = 3$, then the covering array contains all 3-way combinations of variable values. The key property of a covering array is that it includes all t-way combinations of values at least once, and algorithms developed in the past 10-15 years can efficiently generate test arrays for high-order interactions, typically up to t=6. (In the past, nearly all combinatorial testing had been limited to pair-wise, or t=2.) For example, suppose we want to test a module for a lighted text display, which might be controlled using an Arduino board or similar small system. The function allows 10 effect settings for enhancing the text, each of which has two possible settings: flashing (on, off), size (large, small), three light colors that can each be on or off to produce different effects, a "glow" effect (on, off), etc. Testing all combinations would require 210, or 1024 tests. The 10 effects are labeled A through J. If we represent “on” as 1, and “off” as 0, then the array in Fig. 2 provides a compact test set that covers all 3-way combinations.

A	B	C	D	E	F	G	H	I	J
0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	0	0	0	0	1
1	0	1	1	0	1	0	1	0	0
1	0	0	0	1	1	1	0	0	0
0	1	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0
1	1	0	1	0	0	1	0	1	0
0	0	0	1	1	1	0	0	1	1
0	0	1	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	0	0
1	0	0	0	0	0	0	1	1	1
0	1	0	0	0	1	1	1	0	1

Fig. 2. 3-way covering array

Fig. 2 shows a 3-way covering array for 10 variables with two values each, where each row represents a test and each column specifies values for a variable setting. The interesting property of this array is that any three columns contain all eight possible values for three binary variables. For example, taking columns D, E, and G, we can see that all eight possible 3-way combinations (000, 001, 010, 011, 100, 101, 110, 111) occur somewhere in the three columns together. In fact, any combination of three columns chosen in any order will also contain all eight possible values. Collectively, therefore, this set of tests will exercise all 3-way combinations of input values in only 13 tests, as compared with 1,024 for exhaustive coverage. Similar arrays can be generated to cover up to all 6-way combinations. The larger the problem, the greater the improvement over exhaustive testing, because for a given

interaction strength, the number of rows (tests) in a t-way covering array increases with $log\ n$, for n parameters, while the exhaustive test set size of course increases exponentially. (Note that covering arrays are not restricted to binary variables; these are used here to simplify the presentation.) For example, with 34 on-off switches, there are approximately 17 billion possible combinations, but all 3-way settings can be covered with only 33 tests, and all 4-way combinations with 85 tests.

An extensive body of empirical work shows that these methods are highly cost effective. In practical applications, combinatorial testing has been shown to provide significant advantages, with reduced cost and greater fault detection [10][11]. Fortunately, these methods can solve some of the unique problems of testing NoTs.

VII. Applying Combinatorial Testing to the Example

How can we adequately test NoTs, given the constraints identified by the DRR for these systems? Consider again the example for which 10 inputs from an input domain of 10 million produce a ‘1’, with the rest producing ‘0’. For any computed result, there must be some specification that defines the conditions under which each possible result is produced. In general, these conditions will be specified by some logical predicate, especially for NoTs, where decision predicates receive inputs from various sensors and generated values elsewhere in the system. One such example is shown in figure 1. Here we have the decision trigger “*if* $g(x,y) > 100$ then buy Z shares of stock S”. In this example, an expression of two values, x and y, is used in the decision; we assume different combinations of x and y values not shown in the figure may generate different results. For decision triggers in the network, many additional expressions with different combinations of values may be used in decision triggers. It is easy to produce positive tests for this example: simply specify values to make the expression $g(x,y) > 100$ true. But what if we want to ensure that “buy Z shares of stock S” action is not triggered under some other conditions? If the DRR tells us that there is only a small portion of the input space for which this action is correct, how will it be possible to test the huge portion of inputs for which the result should not be to buy Z shares of stock S?

One approach to achieving this assurance is to use the pseudo exhaustive test method described in [14], where we have a small set of possible outputs. This method produces two test arrays for each possible output, or class of outputs. One array includes tests for each condition where a particular result should be produced. For example, suppose a decision trigger is “$x+y > 200$ && $x > 100$ || $x < 100$ && $y > 500$ || $y > 1000$ then *R1*”, where *R1* is a Boolean output corresponding to some action that the system performs. Only three positive tests are needed, one for each conjunct within the expression. A more difficult challenge is showing that the code implementing the expression does not generate the result for some combination of variables inappropriately. This expression is in disjunctive normal form, where each term contains at most two variables, so it is in 2-DNF. By generating a 2-way covering array of values for x and y, but excluding the positive cases, we have a test set with all possible 2-way combinations of values where *R1* should not be produced. Thus we address the oracle problem by verifying that results for tests in each array are equivalent, rather than specifying a set of inputs and determining the output specifically for each one. In the first array, we should see *R1* as the output for each test, and the second array we should not see *R1*. Because we verify positive and negative results for each output *Ri, (i=1,2,3,…)*, we have a sound and complete set of tests without the conventional test oracle problem of computing outputs for each set of inputs.

To see the power of this method, consider the example introduced previously, with roughly one million possible inputs, where 10 produce an output of '1' and all others produce an output of '0'. This could occur with a system of 20 Boolean parameters, for example, resulting in 220 = 1,048, 576 possible inputs. From the specification, we derive the conditions under which '1' is produced, in the form of *if-then-else* rules or a decision tree. Transforming these rules into k-DNF form, we produce a set of conjunctions that result in the '1' output. Suppose that the 10 conditions that result in an output of '1' contain at most three Boolean literals (e.g., *x && ~a && y*). It is easy to produce tests to verify this output for each of the 10 conditions, but how can we ensure that none of the other 1,048, 566 inputs will produce a '1' instead of the correct value of '0'? Surprisingly, we can verify this for all 3-way combinations of inputs with only 28 tests, by generating a covering array of all 3-way combinations, excluding the 10 conditions that should produce '1' [14]. The test arrays will also catch a large proportion of combinations with more than three Booleans, or we can generate arrays up to 6-way with less than 400 tests. This method is not restricted to Boolean inputs, but may be applied to complex conditionals as well, and as a result is especially well suited to testing complex conditionals in decision triggers.

Given a formal specification of the conditions for each decision trigger, the test arrays described above can be produced mechanically, but many tests will still be needed. Thus, even though a conventional test oracle is not needed (because each of the two arrays should produce the same result), the large number of tests may be prohibitive in some applications. The DRR calculation can be used to identify the most difficult to test interface points, helping to establish priorities and allocate testing resources.

VIII. Summary

We believe that because of the necessary role of decision triggers, specifically purposed NoTs have testability concerns. We explained how the testing oracle problem applies to NoTs as well as other distributed systems, and that the problem may be worse compared with other complex IT systems due to “leased assets.” And finally, we described applications of combinatorial testing and the domain range ratio, and how these methods provide a practical approach to IoT testing complexities.

We hope this review of challenges and potential solutions will offer new insights into how to more efficiently test NoTs.

References

[1] J. Voas, “Networks of ‘Things’”, *NIST Special Publication SP 800-183* (July 2016), http://dx.doi.org/10.6028/NIST.SP.800-183.

[2] Voas, J.M. and Miller, K.W., "Semantic metrics for software testability", The Journal of Systems and Software, vol. 20, no. 3, March 1993, pp. 207-216.

[3] J. M. Voas and K. W. Miller, "Software testability: the new verification", *IEEE Software*, vol. 12, no. 3, March 1995, pp. 17-28.

[4] J. Voas, "Software Testability Measurement for Intelligent Assertion Placement," *Software Quality Journal*, vol. 6, no. 4, December 1997, pp. 327-335.

[5] J. Voas and L. Kassab, "Using Assertions to Make Untestable Software More Testable," *Software Quality Professional*, vol. 1, no. 4, September 1999, pp. 31-40.

[6] D. R. Wallace and D. R. Kuhn "Failure Modes in Medical Device Software: an Analysis of 15 Years of Recall Data*," International Journal of Reliability, Quality, and Safety Engineering*, vol. 8, no. 4, 2001, pp. 351-371.

[7] D. R. Kuhn and M. J. Reilly, "An Investigation of the Applicability of Design of Experiments to Software Testing," *27th NASA/IEEE Software Engineering Workshop*, NASA Goddard Space Flight Center, 4-6 December, 2002, pp. 91-95 .

[8] D. R. Kuhn, D. R Wallace and A. Gallo, "Software Fault Interactions and Implications for Software Testing," *IEEE Transactions on Software Engineering*, vol 30, no. 6, 2004, pp. 418-421.

[9] D. R. Kuhn and V. Okum, "Pseudo-exhaustive testing for software," *30th Annual IEEE/NASA Software Engineering* , 2006, pp. 153-158.

[10] Y. Lei, R. Kacker, R, Kuhn, V. Okun and J Lawrence, "IPOG: a General Strategy for t-way Software Testing," *14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, Tucson, Arizona, March 26-29, 2007, pp. 549–556.

[11] J. D. Hagar, T. L. Wissink, D. R. Kuhn, D. R. and R. N. Kacker, "Introducing combinatorial testing in a large organization," *Computer,* vol. 48, no. 4, April 2015, pp. 64-72.

[12] A.H. Patil, N. Goveas, and K. Rangarajan, "Test Suite Design Methodology Using Combinatorial Approach for Internet of Things Operating Systems," *J. Software Eng.Applications*, vol. 8, no. 7, 2015, p. 303.

[13] G. Dhadyalla, N. Kumari, and T. Snell, "Combinatorial Testing for an Automotive Hybrid Electric Vehicle Control System: A Case Study," Proc. IEEE 7th Int'l Conf. Software Testing, Verification and Validation Workshops (ICSTW 14), 2014, pp. 51–57.

[14] D. R. Kuhn, V. Hu, D. Ferraiolo, R. Kacker, R. and Y. Lei, "Pseudo-exhaustive Testing of Attribute Based Access Control Rule," *5th International Workshop on Combinatorial Testing*, 2016, pp.1-10.