Why We Need New Software Testing Technologies

Carol Oliver, Ph.D.

carol@carolcodes.com

Abstract

Mobile and IoT software must perform in a dramatically greater variety of environments than traditional software. Yet the core testing technologies in widespread use today do not directly address this vast environmental variability. Cloud-based device testing is bridging the gap for now, but it is an incomplete solution. This paper presents a Release-Readiness Levels Framework that provides vocabulary and a structure for discussing the gaps between what software testers would like to be able to test and what the existing tools and technologies enable them to test. This paper then identifies the existing software testing technologies that might be extended to better meet practitioner needs, describes the requirements for entirely new software testing technologies to target the needs of mobile and IoT software testing into the future, and offers a glimpse into how the emergence of new testing technology is likely to proceed.

Biography

Dr. Carol Oliver earned a B.S. in Computer Science from Stanford University and a Ph.D. in Computer Science from Florida Institute of Technology. In between, she worked for about 15 years for campus infrastructure IT services at Stanford University, where she did some web app and a lot of middleware software testing.

1 Introduction

My academic research argues that practitioners need a new generation of tools to support mobile and IoT (Internet of Things) software testing. In this paper, I will present a brief history showing how mobile and IoT software differs significantly from traditional software. Then I will present the Release-Readiness Levels Framework, a tool for discussing the possible extents of software testing on a project – and for features and smaller details within any project. Next, I will survey the Seven Core Software Testing Technologies available today, highlighting their strengths and limitations and how those different technologies can fulfill (or not) the needs of the Release-Readiness Levels. Finally, I will present the requirements for the next generation of software testing technology and tools, and set context for how the emergence of the new technology is likely to proceed.

This paper is drawn from my Ph.D. dissertation (Oliver 2018) and many details and specific citations have been omitted from this paper in the interests of time and space; they are available in my dissertation, which is the primary reference for this paper.

1.1 Scope Note

Many types of testing concerns need addressing when preparing software for production release. Application usability and accessibility, installation and update accuracy, security and performance, etc. All these are very important concerns, but this work limits its attention to Functionality Testing.

1.2 Vocabulary Note

This work discusses code moving from Development Focus to Testing Focus. These phrases are used to emphasize that no adherence to any particular software development methodology is implied (e.g. Waterfall, Agile, etc.). In a company with separate departments for development and testing, this change in focus corresponds to sending the code from one department to another. In a one-person shop, this change of focus corresponds to removing the Developer Hat and donning the Tester Hat. Who does the work and when that work is done are not in scope in this paper. The scope of this paper is what work it is possible to do and why that work may be worthwhile.

Development Focus work is about trying to find a way to make something work. Testing Focus work is about challenging what has been created to see if it will hold up no matter what adverse circumstances occur. These are two creative efforts with a shared purpose (the production of successful software) but different – and contradictory – goals.

2 Major Phases of Computing and Software Testing

History tells how software testing came to be what it is today. A look at the arc of computing history reveals how software testing is inevitably entwined with the state of computing technology.

The earliest computers created a 100% known software execution environment: Programs were written in each computer's particular language to take advantage of each computer's unique capabilities and restrictions. Unless a moth flew into the hardware[1], the full context of the computer while running the program was predictable. Mainframes diversified, computing languages abstracted, and programs came to be written for multiple hardware platforms. The software execution environment could no longer be fully predicted, but the variations between computing execution environments remained relatively constrained.

The emergence of desktop computing created an explosion of change. Many new hardware manufacturers emerged, making both new computers and various accessories to extend their capabilities. Peripherals all spoke different hardware languages and operating systems relied on the hardware manufacturers to create functioning drivers to enable their devices to work on consumers'

[1] The historical first bug, found in 1947. See http://www.computerhistory.org/tdih/September/9/.

systems. For some years, computing saw rampant device driver incompatibilities. When a consumer purchased desktop printing software, the consumer needed to check if their printer was on the list of compatible printers for that program. It was hard to predict the execution environment for software, and strange errors occurred (e.g. a program might not run if a peripheral not even used by that program was attached to the system).

Then standards emerged and the operating systems with consumer-market dominance took over the bulk of interfacing to hardware (C. Kaner, pers. comm.). Most hardware manufacturers implemented exactly that interface, most software manufacturers programmed for only that interface, and the operating system passed information across the border. Execution environments for programs became largely predictable again, and problems related to environmental conditions became a relatively small aspect of software testing.

The emergence of mobile computing created another explosion of change, several magnitudes larger. There is now a very rapid change cycle for hardware, operating systems, and applications. Rather than years, merely months pass between major changes in each category. A few consumers change quickly to the most recent everything; many lag a few revisions behind state of the art (Android Open Source Project 2018); and a few still run systems others have consigned to museums (Google Tech Talks 2013). Just that aspect of environmental conditions is wildly variable for mobile apps.

The mere aspect of mobility adds to the unpredictability of environmental conditions while a program runs. As they move about the world, mobile devices change their network connectivity methods and parameters, and they are impacted by changes in the geography and weather conditions around them. Anecdotally, network connectivity around skyscrapers and device stability in extreme temperatures are both problems. Exacerbating the unpredictability problem, mobile devices consume input from a wide variety of sensors that previously were specialty equipment but now are commonplace, like magnetometers and accelerometers. Applications are expected to respond to a great deal of that data, and sometimes the mobile operating system imposes the new conditions on the programs whether the programs handle them gracefully or not (e.g. screen orientation changes, dimming the screen in response to changes in ambient light, replacing the active network connection details at any moment, etc.). These types of environmental unpredictability cannot be solved by standardization of interfaces; they are inherent aspects of mobile computing.

IoT software shares many of the environmental characteristics of mobile computing, especially the deep reliance on embedded sensors.

This history shows that mobile and IoT apps operate within a fundamentally different scale and scope of environmental unpredictability than programs in any prior computing era. To be effective, automated software testing tools need to address this unpredictability directly.

3 Release-Readiness Levels Framework

When I worked as a practitioner, specific tacit goals underpinned my software testing activities. As I met other senior testers and discussed experiences with them, I found resonance in how we approached our software testing efforts. We all wrestled with the question of judging when software was performing well enough that we could recommend its release. Our project managers made the release decisions, but they expected us to render professional judgments and supporting evidence to inform those decisions.

We lacked a good framework for discussing the options. Historically, the ISO/IEC/IEEE 29119[2] standards and their predecessors have not been entirely helpful. Guidance on managing the testing process or doing requirements traceability does not directly answer the core problem: How to judge if the software will perform adequately in the field. There is tension between how much testing is feasible to do and how much productive information about the range of field conditions is needed to enable sound decision-making.

[2] http://softwaretestingstandard.org/

My Release-Readiness Levels Framework (Figure 1) captures one perspective on how to assess that tension and how to discuss what additional testing might be desirable at any point in the project. Underlying this model is the concept of exposing the software to increasingly difficult challenges, thereby increasing knowledge about the variety of conditions in which the software will perform acceptably.

Each level in the Release-Readiness Levels Framework applies to individual features, communicating feature sets, and to the program as an integrated whole. The further testing goes through the levels, the further confidence develops that the software will behave desirably in a wider variety of circumstances.

Note that this is a map of the testing *possibilities* – not a statement of required steps. The appropriate point at which to release the software in question is a Project Management decision and varies for each program and often for each release.

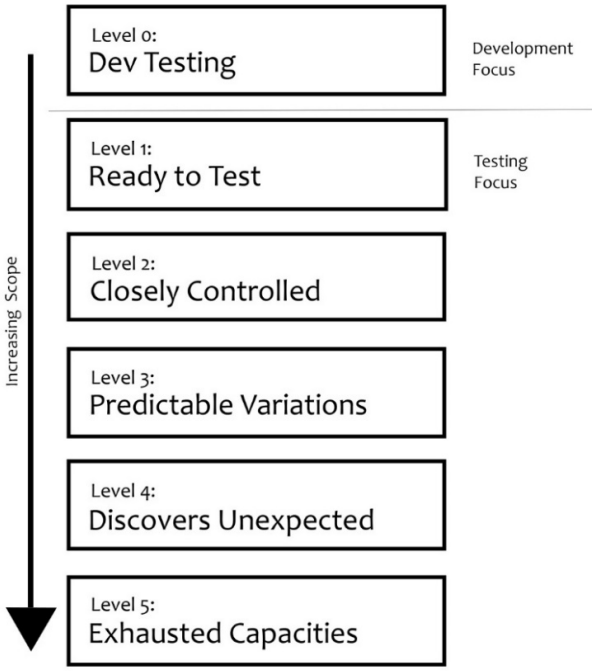

Figure 1. Release-Readiness Levels Framework

3.1 Level 0: Dev Testing

Level 0 testing occurs during Development Focus, and its goal is to assess whether the intended functionality has been implemented. Characteristically, very small aspects of program behavior are analyzed separately from all other behaviors of the program. In modern software development, Dev Testing consists mostly of unit tests, but this is also where style checkers and other static code analyzers are usually applied.

Software released at Level 0 (Dev Testing) may contain missing, partly-functional, and incomplete features. It may not install cleanly into any environment other than the Development Environment, and it may not work as a cohesive whole even if it installs cleanly. Professional software development shops usually test further than this before production release.

3.2 Level 1: Ready to Test

The goal at Level 1 is to assess whether the installed software is ready to be challenged by Testing Focus work. Characteristically, functionality is tested only surficially, just enough to show whether the

program crashes with trivial ease or permits access to all features of current interest; it is common in the early stages of a project for features to be missing or known to be not ready for Testing Focus yet.

Specific behaviors checked at Ready to Test tend to focus upon whether the install process worked correctly and upon the basic stability of features to be tested. Happy-path scenarios in which the software is used exactly as intended are checked, along with the most common or most predictable error-path scenarios.

Software released at Level 1 (Ready to Test) should install properly into expected operating environments but may break if used even slightly differently than anticipated, whether by user action, data state, or an operating environment that was not tested.

3.3 Level 2: Closely Controlled

The goal at Level 2 is to assess whether features work when challenged by tests with carefully controlled parameters. Characteristically, test data is hardcoded but may occasionally be drawn from small lists. Testing activities consist of challenging the program's assumptions in various ways, subjecting it to atypical usage patterns, data that is potentially hard to process, easily-triggered system and environmental interruptions, etc.

Software released at Level 2 (Closely Controlled) is likely to suffer many field failures on systems that differ from Development and Testing Environments; it may also break if used in unanticipated ways or if used on a continuous basis for some block of time, as characteristically tests at this level consist of very brief runs of the app to test a specific behavior, followed by resetting the app to a clean state before testing the next specific behavior. Such run intervals provide clear boundaries between tests, making it easy to identify which test failed, but these runs are markedly shorter than most scenarios in which users will actually use the app.

3.4 Level 3: Predictable Variations

The goal at Level 3 is to assess whether features of the program continue to work when test conditions are significantly loosened and many predictable variations in those parameters are exercised. Test cases at this level take different values at each execution, drawing specifics from large sets rather than fixed details or small lists of possibilities. Characteristically, Predictable Variations tests intensely vary one or a few variables while holding others constant, as the knowledge sought is about how those specific variations alter the program's response.

Software released at Level 3 (Predictable Variations) is markedly more stable in a diverse variety of conditions than software released at Level 2 (Closely Controlled), but it may still experience a moderate number of field failures. These stem largely from unexpected patterns of use and from untested environmental and data conditions. The more aspects of environmental unpredictability not tested, the higher the likelihood of finding many problems after release.

3.5 Level 4: Discovers Unexpected

The goal at Level 4 is for testing to uncover problems that no one on the project could predict, sometimes that no one on the project could imagine. Experienced practitioners regularly exchange tales of finding bugs like this, often accidentally, because modern programs are complex and contain many tacit dependencies within themselves, with their operating environment, and with their data.

In various ways, a program and its operating environment tested at this level are placed under stress. Many of the test techniques for finding these unpredictable bugs consist of HiVAT (High Volume Automated Testing) techniques, such as long-sequence tests, long-duration tests, and random variations of structured input (i.e. fuzzing). Characteristically, the scale of tests increases dramatically at this level. HiVAT techniques are intended to efficiently run hundreds of thousands, millions, or scrillions of specific tests.

Senior technical testers in the practitioner community have created HiVAT testing efforts for decades, but the approach typically requires programming expertise and these techniques are not widespread in general practice.

Software released at Level 4 (Discovers Unexpected) tends to experience some field failures as combinations of conditions are discovered in the field that were not reached by intensive testing, but these failures are rarer than at earlier levels.

3.6 Level 5: Exhausted Capacities

The goal at Level 5 is for testing efforts to exhaust their capacities, given the capabilities of the suite of tools and techniques and the time available. Stresses upon the software are dramatically increased, sampling scope approaches closer to exhaustive, and system testing increases the variations in exercising interleaved, cooperating, and coexisting features.

This level of testing intensity is typically done when the risk of failure is extremely high (e.g. data-processing logic that could corrupt the database if done wrong, life-support systems, high-impact infrastructure systems, etc.). Possibly the best-known example of Level 5 (Exhausted Capacities) testing is the preparations made for Y2K as computing approached the year 2000.

Computing professionals not involved in the Y2K transition often lack understanding of the details. The problem was caused by a vast quantity of legacy code that assumed all years fit the format 19xx, and so stored only the last two digits to distinguish the year; the imminent arrival of the year 2000 meant all that code was going to break and comparisons for which data was older or newer would be incorrect unless four digits were compared. Surficially, the answer appears to be as simple as just replacing two-digit storage with four-digit storage.

However, the reality was a great deal more complex than that. Many companies attempted to patch their legacy software only to find that every patch spawned multiple new problems; the harder they tried to fix their code, the less functional their software became. In the United States, many banks sold themselves to the few banks with Y2K-compliant software, and antitrust regulators approved the sales because the financial infrastructure of the country had to continue to work reliably (C. Kaner, pers. comm.).

Problems riddled software at all levels, and companies in a vast array of disciplines created entire duplicate computing systems and networks so they could advance the date in this test environment and see what would break and what fixes actually worked. That degree of duplicate computing hardware alone cost an exorbitant amount (easily millions or billions or more, worldwide), and very significant human time was invested in testing and retesting critical systems. Y2K was a worldwide test effort that covered years. Popular understanding is that Y2K was a lot of noise made about nothing, because when the clock turned over, only very minor things broke; the reality is that Y2K was an incredibly successful, Level 5 testing effort that exhausted the capacities of the people, technology, and time invested in it.

Software released at Level 5 (Exhausted Capacities) experiences the fewest possible field failures, but some failures remain possible.

3.7 Release-Readiness Levels Recap

To properly assess whether mobile and IoT software will perform adequately in their field conditions of massive environmental unpredictability, software testing efforts need to include tests at Level 3 (Predictable Variations) and Level 4 (Discovers Unexpected). Some critical behaviors within many apps likely deserve tests at Level 5 (Exhausted Capacities); in some cases, whole software systems may merit that level of testing (e.g. autonomous vehicles). So, what levels of testing are enabled by our current software testing technologies and tools?

4 Core Software Testing Technologies

The process of software testing involves creating tests, executing them, and evaluating what happened as a result. Whether manual or automated, executing a test relies on specific technology to make it function. These software testing technologies are mechanisms for exercising a program in directed ways or for obtaining information about the software's behavior. They are the keys which enable different test methods, the gateways which define the character of what kinds of tests are possible.

I identified seven core testing technologies present in today's readily available tools and the academic research literature:

1. Physical devices
2. Virtual devices (emulators and simulators)
3. Simulation environments
4. Mechanisms for interacting with the Visual GUI Tree
5. Image-comparison
6. Code manipulation (e.g. xUnit, code instrumentation, etc.)
7. System monitoring

Each technology enables certain kinds of tests and is better-suited to some types of investigations than others. Existing tools and testing approaches fulfill a technology's potential to differing degrees, and each implementation of a technology may be evaluated in terms of how well it handles the environmental unpredictability characteristic of mobile and IoT software.

4.1 Physical Devices

Testing on physical devices is the baseline testing technology. When programs were written for just one computer, testing the program meant running it to see if it worked, at least well enough for the intended purpose at that time. Beginning programming students do the same thing today.

As computers diversified, software testing on physical devices expanded to include compatibility assessments between the program and different execution environments. In the desktop computing era, software development companies typically had test labs containing a variety of supported equipment. Companies could buy representative hardware for all their major supported systems and expect to use those machines for years; significantly new systems came out every few years when chipsets changed, and the operating systems updated every few years as well.

In the mobile era, the comprehensive in-house test lab has ceased to be feasible. Instead of dozens of distinct hardware platforms, there are thousands in use worldwide (OpenSignal 2015). New device models appear at least once a year from most device manufacturers, and sometimes more frequently than that; operating systems receive frequent major updates (sometimes several times per year) and nearly continuous minor updates. The scale and pace of mobile environment changes make maintaining traditional in-house test labs of physical devices for all in-scope systems prohibitively expensive.

Thus, the environmental unpredictability problem mobile software testing faces for this technology is simply access to a diverse-enough collection of physical devices.

4.1.1 Accessing Sufficient Diversity

One strategy is to test only on the top 10-15 devices in use by a mobile app's target users, broken into strata of high-end devices, mid-range devices, and low-end devices; which devices belong in which grouping changes very quickly, as new devices are released. This is an example of Level 2 (Closely Controlled) testing because the test data varied here is a small list (10-15 devices). Despite its limitations, this strategy should work adequately for reasonably homogeneous target populations; as the diversity of target users increases, more of their systems will not be represented in this sampling strategy.

Arguably, the most common option in modern app development is cloud-based testing. Fundamentally, these cloud services leverage existing software testing technologies to provide networked manual or automated access to physical devices hosted and maintained by the cloud service provider; various providers exist and exactly who is in business changes over time. Because so little can be inferred about the stability of software on other devices based upon its behavior on one device, mere replication of tests across many different mobile devices is not very informative. Tens or even a few hundreds of devices tested is still a small fraction of the possible thousands of devices that a mobile app could be deployed to, so it is difficult for mere replication across devices to lift testing beyond Level 2 (Closely Controlled). Further restricting the power of cloud-based device testing, the kinds of tests that can be performed on such devices are bounded by the testing interfaces provided by the service, and that limits their power.

4.1.2 Technology Strengths and Limitations

The greatest strength of testing on physical devices is trustworthy realism about how apps will behave on that device. Although that observation seems obvious, fidelity to reality is a great enough issue in mobile app testing that the point is emphasized by academics and practitioners alike (Delamaro, Vincenzi, and Maldonado 2006; Ridene and Barbier 2011; Muccini, Di Francesco, and Esposito 2012; Nagowah and Sowamber 2012; Gao et al. 2014; Vilkomir and Amstutz 2014; Knott 2015).

One of the motivators for the emphasis on realism is the mobile fragmentation problem. Mobile hardware varies greatly in chipsets; screen size, display resolution and pixel density; number and types of network interfaces, sensors, and embedded devices (e.g. camera, speakers); quantity of working memory and local storage space; and battery performance. The current state-of-the-art solution to this vast variety is to test on a sufficient number of real devices, where “sufficient” is interpreted individually by every software publisher.

Some limitations apply to using physical devices as a testing technology. Mobile devices are consumer devices, and (Google Tech Talks 2013) points out that consumer devices are not designed to run 24x7, so hardware used for extensive testing will fail after a few months and need replacing.

4.1.3 Implementation Limitations

Another pragmatic limitation on the number and type of tests executed remains the financial cost of testing. Even though cloud-based testing services have dramatically lowered the costs to software producers of provisioning and maintaining physical devices, these services still cost money. Budget and service constraints limit the testing minutes purchased for a project, forcing tradeoffs in what is tested and how extensively.

Cloud-based pools of physical devices vastly improved access to a variety of devices, but there is little variety in the physical or network environments of server rooms, providing little opportunity to exercise the embedded sensors and other equipment in cloud-based mobile devices. Examining these aspects of mobile devices in any detail still requires physical access to the appropriate device, and the common solution is for individuals to move around the world, interacting with each device one at a time. The obvious problems in scaling this approach explain the appeal of crowdsourcing[3] testing, but that approach lends itself to haphazard data collection and great challenges in repeatability.

Again, cloud-based testing is limited in its possibilities by the testing interfaces that the service providers support. Generally, it is not possible to extend or replace a provider’s testing framework, so tests are constrained to the types and design styles the provided frameworks support.

[3] The practice of outsourcing testing to a crowd of people, mostly without any technical training, in an attempt to obtain useful feedback about how the software will perform for real users (Knott 2015, 141–45).

4.1.4 Physical Devices Technology Summary

Testing on physical devices yields highly trustworthy results but also incurs test-management overhead costs that can be significant, somewhat limiting scale. Cloud-based providers close the gap in trying to examine the effects of all the combinations of hardware and operating system versions, but they do so in a server room environment that provides little scope for exercising the embedded sensors and other equipment in mobile devices. A robust solution to scaling testing that exercises embedded sensors and equipment is either not yet invented or not yet widely available. The types and design styles of tests run on cloud-based devices are limited to what the provider's frameworks support; support for custom testing needs is quite rare.

4.2 Virtual Devices

Virtual devices leverage readily-available and affordable hardware to mimic the specific hardware and operating environment of less easily obtained systems. The manual analog is the collection of techniques involved in desk-checking, where a human reads through the code, pretending to be the computer, analyzing how the program is going to run.

Computer-based virtual environments appear to have their roots in the first time-sharing operating systems of the 1960s (Creasy 1981; Pettus 1999). Over the next several decades, computing professionals developed a decided preference for emulators over simulators. Simulators ran faster, but they characteristically used the full resources of the host computer to perform the computing tasks. Emulators created representations of the foreign hardware within the host computer, using just the resources the real device would have available and using a virtual chipset to execute the foreign program at the binary or system-call level. Testing on simulators gave an approximation of how the program would behave on its target computer, but the exact fidelity in representing the emulated device created great trust in the results obtained while using the emulator.

At the beginning of the mobile era, devices were especially resource-constrained, making it difficult to run testing software on the physical devices. Vendors addressed this problem by providing virtual devices to enable testing (Satoh 2003). Although traditional programs tend to be developed on their target platforms, mobile apps are developed on desktop computers and deployed to mobile devices later. Consequently, virtual devices are everyday tools during mobile app development today.

4.2.1 Technology Strengths and Limitations

The primary strengths of virtual devices for mobile apps today are enabling development tasks on non-mobile systems and the ease of switching between devices. The practical matter of cost savings is another strength inherent in the technology, as the actual physical devices are not a factor.

The history of simulators and emulators from the desktop era has shaped expectations about them in the mobile era. When the term "emulator" is applied (e.g. the Android Emulator), the implication is that it will provide the exact fidelity trustworthiness of the prior era.

Unfortunately, there are significant limitations to the virtual devices representing mobile devices. Although the mobile CPU, OS version, RAM, and screen size, resolution, and pixel density are virtually represented, it is well-understood that sensors and other embedded hardware commonly are not. Functionality involving the camera, GPS, accelerometer, microphone, speakers, gyroscope, or compass; sensors for ambient light, pressure, temperature, humidity, or proximity; and network connections like Bluetooth, Wi-Fi, NFC (Near Field Communication), and cellular communication for 3G, 4G, etc. – all these are difficult or impossible to test on virtual devices, especially those provided by the platform vendors (Muccini, Di Francesco, and Esposito 2012; Griebe and Gruhn 2014; Knott 2015, 115). It is not clear that including all these embedded components in a virtual device would even be useful; most of these are pure input mechanisms, not manipulated by either the user or the program. Merely representing them as pieces of hardware in a virtual device is not enough to make them usable; a different technology is required.

4.2.2 Implementation Limitations

The Android ecosystem faces another set of serious limitations, stemming from its open-source nature. Microsoft controls their operating system, and Apple controls both their hardware and software, so these ecosystems may not have the same problem.

However, the Android OS is commonly customized by both hardware manufacturers and cell service providers. Some of this customization is required to make Android work on the equipment; other aspects are customized for user experience, and sometimes these changes are very significant functionality differences. Complicating matters, most of these customizations are proprietary and therefore not publicly documented.

This means the best the Android Emulator can do after setting up the partial-hardware representation of a device is layer on a stock version[4] of Android OS. Seen in that light, it is not just that features to send context data to the virtual device are awkward to use or unavailable, it is that testing on a virtual device *lies* about how well things will work in the field. The virtual devices are idealized deployment environments that incompletely and inaccurately represent the real devices. This limits the power of testing on Virtual Devices to Level 1 (Ready to Test) because the information obtained is only a ghost of reality.

This explains why experienced practitioners strongly advise testing primarily on physical devices in real environments (Knott 2015, 3–4), limiting the use – if any – of virtual devices to only the most basic, simplistic tests (Kohl 2012, 278–79; Knott 2015, 52).

4.2.3 Virtual Devices Technology Summary

Testing on virtual devices enables effective development of mobile apps, since development is done on different systems than where the software is deployed. However, all mobile virtual devices lack completeness; significant hardware components are routinely omitted, and Virtual Device technology alone is insufficient to exercise the sensors and other embedded equipment even if they were included. The Android Emulator diverges even further from exact fidelity in representing a physical device because it can only apply a stock version of the Android operating system, absent all proprietary customizations made by hardware manufacturers or cell service providers.

4.3 Simulation Environments

Simulation environments take simulation beyond the device level, instead creating virtual representations of various aspects of the world surrounding the mobile device. Academic work has mostly studied networked communication scenarios and a few richer environmental context simulations.

There is little discussion in the practitioner literature about simulation environments being used for mobile software testing. This suggests that good tools for useful mobile simulation environments are not generally accessible to practitioners.

4.3.1 Technology Strengths and Limitations

Simulation environments bring realistic deployment scenarios into a controlled lab environment, allowing intensive variation of specific aspects of interest without the expense of travelling to the correct conditions or waiting long enough for the correct conditions to occur. Theoretically, this should enable testing at Level 3 (Predictable Variations), Level 4 (Discovers Unexpected), and Level 5 (Exhausted Capacities).

[4] (Google Tech Talks 2013) explains that all Android Emulator images are rooted (i.e. the user account has been given root access, meaning full control of the OS), although consumer devices generally are not; and that the Android OS applied in the Emulator has been modified to be more test-friendly and to do more logging. For testing purposes, these are practical, probably desirable things, but they further distance the results from exact fidelity.

Most applications of simulation environments also involve sophisticated modeling of the conditions of interest, to ensure the simulation meaningfully represents real conditions. For example, performance testing of websites typically involves estimates mixing percentages of different representative types of users so that realistic traffic and session loads can be generated. This complexity tends to isolate use of simulation tools (like those for performance testing websites) to only a few, highly-specialized testers brought in as consultants when a company has a specific need. Simulation environments are rarely used – if at all – by most practitioner software testers.

4.3.2 Implementation Limitations

The primary limitation of simulation environments applied to mobile testing today is that there are so few instances exploring it. A secondary limitation is that existing tools focus on testing user-generated events in short bursts but do not enable testing extended usage scenarios (an hour or more) (Meads, Naderi, and Warren 2015); this means that the simulation environments are not representative of reality in terms of how users will use the apps, the devices, the network resources, etc. Furthermore, the focus on user-generated events means little testing is being triggered by sensor readings or background services, despite those being extremely common vectors of input to modern mobile and IoT devices.

4.3.3 Simulation Environments Summary

Simulation environments contain rich potential to assist in testing mobile and IoT devices, but their potential is not well-addressed at this time. Academic work investigates networked communication scenarios and some environmental context simulation, but the scope of the tests is limited overall. Practitioners seem to have no useful access to tools for applying simulation environments to testing mobile apps.

4.4 Visual GUI Tree

Mechanisms for interacting with the Visual GUI Tree form arguably the most commonly used GUI application testing technology in use today. The Visual GUI Tree is the hierarchical arrangement of objects forming the rendered display of a modern GUI. All the currently-visible objects on the screen (e.g. buttons, images, etc.) are part of it, but so are many invisible objects that control how objects on the screen are placed (e.g. layout managers which arrange contained objects in a row horizontally, vertically, or in a grid). Assistive technologies like screen readers for blind users leverage the Visual GUI Tree, so it has wide availability on desktop systems, in web browsers, and on mobile devices.

To encode a test accessing the Visual GUI Tree, some absolute identification of the object in question is required. The earliest interaction mechanisms identified objects based on their coordinates on the screen, and that remains a common fallback mechanism today. Some interaction mechanisms scrape the screen to locate specific text, which is then used to identify the object of interest; this allows the text to appear at different coordinates and not break the encoded test. The most reliable means of accessing a specific object is to use a distinctive attribute of that object, commonly some kind of identifier or xname value. If such a unique identifier is not available, then the object's XPath location in the Visual Tree (i.e. the path of elements that must be traversed to reach the desired element) may be used to navigate precisely to the desired object. Once a desired GUI object is located, testing frameworks interact with it as an object, entering text or sending click commands; more mobile-centric frameworks provide gesture commands, such as swiping the screen in a particular direction.

Tools using the Visual GUI Tree are prolific and widely used today, as this is an excellent technology for replicating user interactions with a GUI. The vendor-provided automation tools for each mobile platform use this technology, as do GUI Record and Replay tools. All the cloud-based device services primarily leverage the Visual GUI Tree to replicate user interactions.

Most of the tests implemented in Visual GUI Tree tools are Level 2 (Closely Controlled) tests, restricted to specific data hardcoded into each test, repeated verbatim with every test run. This level of detail is the baseline scenario for automated testing. Unfortunately, it is rare for these tools to offer features to

parameterize test case data, much less to determine it programmatically at runtime, features that are required to enable tests at Level 3 (Predictable Variations).

Whether customizations to extend the power of tests is possible in Visual GUI Tree tools usually depends on how flexible a programming language the tool uses as its test scripting language. The need to program scripts in a more sophisticated manner is at odds with the marketing of these tools, much of which focuses on enabling testing by non-programmers, so features enabling coding flexibility are not a high-priority for many tool-makers.

In recent years, an increasing number of testing services have advertised pre-packaged test suites designed to test any app without human effort. These typically install the software and apply Random Monkey testing to the GUI, looking for crashes. Anecdotally, these generic test suites are finding bugs in many mobile apps, but they necessarily cannot be checking substantial functionality of any of them because no generic test suite can know anything about the functionality of any program. Consequently, these tests operate at Level 1 (Ready to Test).

4.4.1 Technology Strengths and Limitations

Ever since GUIs became widely-implemented using objects, interacting with those objects by accessing the Visual GUI Tree has become widespread across desktop, web, and mobile applications. It is an excellent way to automate interaction with a GUI in a manner that replicates how humans interact with it, through exactly the same objects accessed in almost the same ways.

When the Visual GUI Tree works, it works very well, but it cannot work when it cannot uniquely identify an object. (Chang, Yeh, and Robert C. Miller 2010) note that sometimes GUI objects are not given unique identifier attributes. This happens rather frequently in practice, because developers often have no need of such identifiers to uniquely access the object; consequently, the identifiers required by Visual GUI Tree testing tools may only be added by special request to accommodate testers. As a matter of human process, this is not a reliable system; it is tedious work for the developers and adds clutter to their code.

When unique identifier attributes are unavailable, Visual GUI Tree tools fall back to XPath navigational routes to objects or to specific coordinates on the screen. Neither of these works with variable GUI elements – items that may change position in a list as more data is added to the application, or items that are programmatically populated into a display template based on runtime selection filters.

Consider a kitchen pantry app that is asked to display only the perishable items expiring in the next week. Each of those items is selected for display at runtime based on the state of the database and the value of the timestamp when the query is issued; in mobile apps, these are typically displayed as individual GUI objects, but there is no reliable identifying attribute that can be assigned to them and expected to persist across repeated queries. Sometimes identifiers are dynamically created for such elements while the program runs, but new values are dynamically created the next time the program runs, rendering scripting a repeatable test impossible. There is also no guarantee that each type of item will be unique (how would one distinguish between five different bananas?), and elements may not be populated into the display in the same order every time. The identification information simply is not naturally present to set up repeatable tests that will succeed across runs of the application. Automating tests in these situations quickly becomes more a matter of coordinating and maintaining test data (self-verifying records or known-state of the whole database) than about testing the behavior of the app.

The Visual GUI Tree also does not handle any case where one GUI element combines several different behaviors within it, determined by position of interaction on the element rather than by its object identity. Image maps on web pages are good examples, as clicking on different parts of the image triggers different actions. Many mobile games have visual control screens that surficially resemble image maps, where things the user can choose to do are carefully integrated into an overall image.

The Visual GUI Tree is excellent technology for testing behaviors triggered by user actions, but not sufficient for testing behaviors not triggered by user actions. Context-dependent inputs from various

sensors and most error scenarios need another testing technology applied before they manifest, because they result from something else happening while the user is using the app.

4.4.2 Implementation Limitations

On the one hand, using the Visual GUI Tree for testing is a mature technology that has been well-developed across several generations of computing platforms, with well-understood recurrent difficulties and well-known solutions to those difficulties. On the other hand, certain usage limitations have never been robustly supported.

Examples from desktop applications (Chisholm 1997; Nyman 1999; Borjesson and Feldt 2012; Bauersfeld and Vos 2014) and web applications (Stocco et al. 2014) note that custom objects are not handled well in Visual GUI Tree tools. Custom objects derive from established objects but behave differently, often using the technology in non-standard ways (e.g. a list element that contains different background colors for each list item but no text, making a color-picker). The testing tools that rely on the Visual GUI Tree only know about the standard GUI widgets; they cannot know about the custom-created ones, and few tools enable programming extensions to the tool to allow accurate processing of custom GUI widgets.

In a similar vein, (Nyman 1999) and (Nguyen et al. 2014) document difficulties in handling GUI components with context-dependent text strings, such as windows titled after documents or error messages with non-constant details. Sometimes the record-and-replay usage scenario for Visual GUI Tree tools is blamed for the lack of features to handle this need, as the naïve understanding of record-and-replay expects exact repetition of all details and does not imagine tuning the recorded script to be more powerful and to handle variations fitting specific patterns.

(Chang, Yeh, and Rob Miller 2011) point out that some textual data that is read programmatically from object attributes may not be formatted the same way as the text string is displayed on screen. Date and time stamps are particularly likely to diverge. Where the discrepancy matters, this ought to be easily addressed with a little code during Testing Focus work. However, this faces the same difficulty as handling custom GUI widgets and handling parameterized scripts: Much of the marketing around these tools focuses on enabling testing by non-programmers, so features enabling coding flexibility are not high-priority for the tool-makers.

(Gronli and Ghinea 2016) note that mobile apps reliant on animations and clock-based display changes can exhibit erratic failures when tests are replayed, due to timing issues that render some GUI widgets inaccessible or not present when the test script expects to find them. This is something of a surprise, as the need for a “wait for element” feature is well-established in the history of using these tools prior to the mobile era; however, (Google Tech Talks 2013) suggests that new instantiations of using this technology may be independently rediscovering this need rather than considering it essential from the beginning.

4.4.3 Visual GUI Tree Technology Summary

The Visual GUI Tree is both an excellent and dominant technology for testing mobile apps, but the technology suffers from a persistent lack of capacity to handle custom objects and display patterns that vary in specific details, such as error messages. These limitations are exacerbated by variable GUI elements, which are becoming more common in mobile apps because it saves memory to not instantiate elements not visible on the screen; typically populated as list elements at runtime, these elements commonly lack unique identifiers, reliable paths to their location in the GUI Tree, and unique text contents. Another testing technology is required to test behaviors that result from context-data input or most error conditions, as these situations are not triggered by user actions.

4.5 Image Comparison

Image comparison tools take a snapshot of the rendered display of a running program then compare that snapshot to a reference image previously captured. Either whole-image or part-image comparison may be done. The manual version requires humans to visually compare images to a reference standard.

Testing tools at least as far back as the 1980s used whole-image comparison techniques (C. Kaner, pers. comm.), doing pixel-by-pixel comparisons of whole windows. These tools were fragile (every tiny change invalidated the reference image, even if an element just shifted left by one pixel) and fell out of general use, but there has been a resurgence in their use for testing mobile apps, where they are used to compare an entire screen's content to a baseline image (Knott 2015, 109).

The second generation of this technology is part-image comparison, which analyzes the snapshot of a running program to see if it contains the reference image somewhere within the whole picture. This is much more resilient to changes in the GUI layout. (Potter 1993)'s demo tool Triggers is commonly cited as the first work of this kind, although (Horowitz and Singhera 1993) used bitmap part-images in their fully-implemented GUI testing tool XTester, which went far beyond a demo implementation of the idea.

However, comparison of part-images did not occur widely until the introduction of Sikuli in 2009; the technology required advances in computer vision algorithms and in hardware capacities to become practical (Yeh, Chang, and Robert C. Miller 2009). Sikuli emerged from work seeking to help users confused by a GUI icon find help documentation without needing to guess the icon's text name. (Chang, Yeh, and Robert C. Miller 2010) applied Sikuli specifically to automated testing, focusing primarily on the benefits of unit tests established to check the implementation of Visual-Meaning behaviors, such as the Play button on a video player changing images from the Play triangle to the Pause bars[5] after the Play button was clicked; their scope limited itself to checking the GUI layer only, without any underlying connections to application functionality, so in this example nothing would be known about whether or not the video player could successfully show recorded video.

4.5.1 Technology Strengths and Limitations

Part-image comparisons are significantly better-suited for testing Visual-Meaning situations than tests that try to use the Visual GUI Tree for this purpose. (Groder 10/26/2016) contains an enlightening example test case in which the image searched for on-screen is that of Lake Superior rather than of a traditional GUI element object. The picture of Lake Superior is a more accurate indicator of the intended content than any other; if the image claims to be of Lake Superior but actually depicts an elephant, tools using the Visual GUI Tree data will not notice the problem, but image-comparison technology should. However, searching the whole screen for part-image matches is a computationally expensive operation. This means these tests run slowly, which limits how many are worth running.

Another problem with using image-comparison for tests is that the technology can only deal with what is currently visible on the screen – if the target is scrolled off the visible portion of the screen or occluded by something else, image comparison cannot find it (Chang, Yeh, and Rob Miller 2011; Groder 10/26/2016). Scrolling may alleviate this problem, but anything that cannot currently be seen cannot be tested.

4.5.2 Implementation Limitations

The current set of image-comparison tools is hampered in effectiveness by several technology problems that may be addressable with improved code and algorithms. One impact of these is that the tools do not cope gracefully with differences in image size, color, or style (St. Amant et al. 2000; Chang, Yeh, and Rob Miller 2011; Borjesson and Feldt 2012; Groder 10/26/2016; Gronli and Ghinea 2016; Garousi et al. 2017); Sikuli was designed to address these variations (Yeh, Chang, and Robert C. Miller 2009), but in practice it

[5] Notice how you had to pause and translate those word-descriptions into the pictures ▶ and ⏸? That's the essence of a Visual-Meaning situation: The picture conveys the meaning directly and anything else just points to the intended meaning through a stand-in representation.

is not functioning well enough. Practitioners relying on image comparison tests also have to rely on workarounds they need to implement themselves to get around tools that can only recognize one baseline image per screen; this implies that every device tested requires its own library of baseline images (Austin Test Automation Bazaar, pers. comm.). This means tests relying on this data are at best Level 2 (Closely Controlled).

Some commercial testing tools, like Eggplant, are encouraging companies to replace their Visual GUI Tree test suites with Image Comparison test suites, selling this next-generation of tools as based on how users interact with applications rather than on "outdated" code-centric testing approaches[6]. What that pitch does not make clear is that the resulting test scripts require collecting and maintaining a mammoth library of GUI element images, because images need to be collected for each individual element in all its states (selected, enabled, disabled), in every language, in every platform look-and-feel, in various sizes, etc. Collecting and maintaining all that test data is a significant and continuing operational cost, consuming time and energy testers could otherwise spend on testing the app instead of maintaining test data.

Image Comparison is also a poor technique for text recognition. (Chang, Yeh, and Rob Miller 2011) points out that existing OCR[7] techniques are designed for printed pages; thus, they expect white backgrounds, high-resolution text, and rectilinear layouts. Yet screen images may be significantly lower resolution than paper, contain colorful backgrounds, and be laid out much less predictably – all details which challenge the success of text-recognition via images. (Borjesson and Feldt 2012) observed a 50% failure rate in Sikuli's ability to distinguish between the single characters '6' and 'C'.

Lastly, the existing tools poorly handle moving image data, whether in GUI animations or in extracting data from video clips (Borjesson and Feldt 2012; Chang, Yeh, and Robert C. Miller 2010). It is not clear how much this is a limit of the technology, how much a limit of the computer vision algorithms, and how much a limit of the hardware processing resources in use at this time. However, mobile platforms seem to be standardizing on applying more animation to GUI elements for interface usability reasons (Tissoires and Conversy 2011), so this limitation to the tools is a significant challenge.

4.5.3 Image Comparison Technology Summary

Computer vision technology has enabled the ability to test for Visual Meaning problems, which is a great advance, but even part-image comparison is not suited for all purposes. The approach fundamentally cannot address functionality not visible on the screen; as such, it cannot trigger tests for incoming context data nor for most error conditions. Large libraries of images are currently required to apply the tools across platforms and different display characteristics, and searching for images on the screen slows the tests compared to looking for an identifying attribute via the Visual GUI Tree. Image-comparison of text is error-prone because OCR algorithms are designed for a much higher resolution and sharper contrast scenario. Image Comparison Technology is an excellent complement to Visual GUI Tree Technology, but it is not suited to be a complete replacement of it.

4.6 Code Manipulation

This large category encompasses many types of code analysis and modification. All the techniques utilize some insight into the code's details to enable testing, and all require programming.

In modern software development, arguably the most commonly used of these techniques is the xUnit framework. Primary use of xUnit tests is by developers during code creation and code maintenance activities; tiny, specific, very quick execution unit tests are the backbone of most Continuous Integration systems, serving as quick-feedback flags if code changes break the build (Beck 1999, 59–60). These

[6] This sales tactic is afflicting web-based tests also. (Stocco et al. 2014) is about a tool to automatically transform DOM-based tests to image-comparison ones. DOM (Document Object Model) is the object-based representation of a webpage.

[7] Optical Character Recognition

kinds of unit tests form a very important foundation for stabilizing code before it passes from Development Focus to Testing Focus.

Other applications of Code Manipulation Technology include code coverage tools, which report the percentage of the lines of source code executed during a test suite run; code style-checkers and other static code-inspection checkers; and a variety of approaches that directly modify that app code to create mock responses from external resources, primarily network accesses. At the extreme end of Code Manipulation approaches are cases of modifying the Android distribution itself to enable functionality required for testing purposes; it is unlikely that practitioners are going to these lengths.

4.6.1 Technology Strengths and Limitations

The scope and variety of Code Manipulation testing approaches is vast. Essentially, anything can be implemented, so long as it can be coded. Human capacity to address the problem is the limiting factor, not the testing technology. The cost, though, is that some of the knowledge required is deeply technical, and people with that knowledge are often in very senior development positions instead of in testing positions.

4.6.2 Implementation Limitations

More significant limitations on Code Manipulation testing approaches come from the mobile security policies on the devices, which significantly restrict inter-application communications relative to desktop behaviors. (Amalfitano et al. 2015) notes that every Android app (e.g. an app under test and a test tool app) runs under its own user identity rather than sharing the same identity with other applications launched by the user of the computer, as is common in traditional desktops. This leads to a need to set up the test tool to run in the same process as the app under test, rather than independently of it, which demands a tight integration between the tool and the app's source code (Takala, Katara, and Harty 2011; Amalfitano et al. 2012). Different apps can communicate only through the features provided in Android's Binder inter-process communication system (H. van der Merwe, B. van der Merwe, and Visser 2012). Binder calls in turn require global JNI[8] references (Yan, Yang, and Rountev 2013), and JNI is Android's integration mechanism with C/C++ libraries and code. As a further difficulty, Android's test automation frameworks limit tests to interacting with one Activity[9] (Kropp and Morales 2010), meaning that these built-in test mechanisms cannot be used to automate tests that flow between Activities – roughly equivalent to the different visible pages a user sees while navigating through an app.

4.6.3 Code Manipulation Technology Summary

Code Manipulation is an extremely versatile testing technology, capable of testing at all Release-Readiness Levels, with the caveat that it can grow extremely complicated. xUnit frameworks are in widespread use in industry, as are code coverage tools and static analysis style checkers. These are all used primarily while code is in Development Focus. In academia (and perhaps in some highly technical development shops), code is being modified to provide mock responders for external resource requests, primarily network accesses; this is a step towards Simulation Environments. The extreme complexity end of Code Manipulation involves researchers creating customized versions of the Android distribution just to enable their testing activities; this is technically out of reach of most practitioner shops.

4.7 System Monitoring

System monitoring represents all mechanisms for observing and noting device-level system behavior while an app is running.

[8] Java Native Interface

[9] An Android Activity is approximately the same as one screen view of an app, similar in concept to the scope of a webpage.

4.7.1 Technology Strengths and Limitations

System monitoring of the core computing system is a mature and well-developed technology. CPU usage, memory usage (RAM), and disk usage are fundamental diagnostics on most computing systems. Because computers have been networked together for so long, monitoring of network communication interfaces is fairly robust, including wireless signal strength. But monitoring and diagnostic queries of other embedded equipment and sensors is less clear; sometimes these hardware components cannot be queried via software and require external monitoring devices (e.g. power meters to measure battery demand).

4.7.2 Implementation Limitations

Very few testing approaches or tools are manipulating device system-state, even though that can have a dramatic effect on software behavior. Resource-starved computers can behave oddly and fail in unexpected ways. The impacts of misbehaving sensors and other embedded equipment are acknowledged but poorly mapped; computing professionals know systems can be affected by these things, but the types of impacts are not well-understood.

4.7.3 Built-in Program Diagnostics

Similar to how system monitoring inspects and makes visible device-level system behavior, diagnostics built into a program inspect and make visible internal program state while software is running. Such diagnostics are more like features of the program than they are like Code Manipulations made for testing purposes after the features pass out of Development Focus. These built-in diagnostics may take the form of logging messages to log files as code execution passes through certain points; assertions and exception checking within the code's logic may be used. In some cases, comprehensive diagnostic languages and controls may be built into the software to allow inspection of hundreds of different details about the execution of the software as it runs; this was common in testing devices like switchboard telephones and laser printers (C. Kaner, pers. comm.) and seems likely to be vital in the testing of IoT software.

4.7.4 System Monitoring Technology Summary

System Monitoring is primarily applied to core computing elements and network communications, but other sensors and embedded equipment are rarely monitored. Built-in software diagnostics features are powerful testing tools, but they require extensive Development Focus work to design and implement; they provide an entire alternate interface to interacting with the software than the routes used by the ordinary traffic handled by the software.

Because System Monitoring gathers data from a running system, it works in concert with other software testing technologies, widening the view of environmental and program state gathered from tests executed via other technologies. Monitoring the system with generic monitoring functions (e.g. CPU usage, network usage, etc.) increases the Level 2 (Closely Controlled) information available about test execution conditions. Programs with extensive built-in diagnostic features can be flexible enough to uncover Level 4 (Discovers Unexpected) test execution conditions.

4.8 Mobile Software Testing Technologies Recap

As a software testing technology, Physical Devices yield highly trustworthy results. Cloud-based providers greatly expand access to the thousands of active mobile device types but do so in a server room environment that provides little scope for exercising embedded sensors and other equipment in mobile devices. No robust system for testing embedded components at scale is apparent. Cloud-based testing options are limited to what the frameworks supported by that service support, and customized testing options are quite rare.

Virtual Devices effectively support Development Focus activities, but all mobile virtual devices lack completeness because significant hardware components are routinely omitted; Virtual Device technology alone is not capable of exercising sensors and other embedded equipment. Although pre-mobile expectations lead computing professionals to expect high-fidelity behavior from virtual devices labeled as “emulators”, the Android Emulator only represents an idealized runtime environment because it cannot know about the proprietary customizations of the Android OS made by hardware manufacturers and cell service providers.

Simulation Environments contain rich potential to exercise all the embedded equipment in mobile devices, but they are complex and do not seem to be available to practitioners.

The Visual GUI Tree offers direct access to the GUI objects users use, but historically fails to handle custom objects or objects which match a general pattern but vary in details (like error messages). It also does not handle variable GUI elements that are conditionally present based on runtime details, may appear in different places within a data set (like a list), and do not naturally contain uniquely identifying attributes. Visual GUI Tree technology alone is not capable of exercising sensors and other embedded equipment.

Image Comparison technology – especially part-image comparisons powered by computer vision technology – filled a significant gap in automated GUI testing, enabling the testing of Visual Meaning problems. It runs more slowly than tests via the Visual GUI Tree, cannot address GUI elements not fully visible at the expected time, and currently does not handle basic display variations well. Changes in size, color, or style confuse the current tools, which also struggle with moving image content and with accurately reading text on-screen. Image Comparison technology alone is not capable of exercising sensors and other embedded equipment.

Code Manipulation approaches can do anything software can do, but they rapidly grow quite complicated to create. Tools that propagate to industry typically encapsulate a small set of behaviors. The widespread use of xUnit frameworks, code coverage tools, and style checkers shows that well-packaged behaviors will be extensively used by practitioners. Some mobile mocking tools exist for testing requests of external resources, but these are primarily limited to network access.

System Monitoring is robust for watching core computing elements and network communications but is not commonly used for checking sensors, other embedded equipment, or internal program state. Although this technology may be applied during Development Focus to isolate issues, very few Testing Focus tools or automation approaches incorporate system-state conditions to test mobile apps.

Overall, the great variety of environmental unpredictability faced by mobile and IoT apps is poorly addressed because five of these seven technologies do not handle testing input from embedded sensors and equipment at scale. Most of the existing testing tools and research efforts function at Level 1 (Ready to Test) or Level 2 (Closely Controlled). Yet the remaining levels are where the capacity to truly address widespread environmental unpredictability – the hallmark condition of mobile and IoT computing – becomes viable. Simulation Environments and Code Manipulation technologies can do this, but at the cost of high complexity.

In practice, cloud-based access to physical devices is somewhat mitigating the paucity of tools that can be used at Level 3 (Predictable Variations), Level 4 (Discovers Unexpected), and Level 5 (Exhausted Capacities), but these cloud services exercise only the device environment variable, not the embedded sensors and equipment of mobile devices.

5 Vision of a New Generation of Software Testing Tools

The high-level review of computing history in Section 2 established that mobile and IoT computing are operating within a fundamentally different scale and scope of environmental unpredictability than programs in any prior computing era.

The Release-Readiness Levels discussion in Section 3 clarified that mobile and IoT software need to be tested at Levels 3 (Predictable Variations) and Level 4 (Discovers Unexpected) to adequately assess likely field behavior within this vast environmental unpredictability. Some specific behaviors and some whole software systems need to be tested at Level 5 (Exhausted Capacities) because of their inherent life and safety impact or their core data integrity impact.

Touring the seven core testing technologies in Section 4 showed that most existing tools and research efforts function at Level 1 (Ready to Test) or Level 2 (Closely Controlled), with a very few stretching partly into Level 3 (Predictable Variations). Only two software technologies provide means for exercising the functionality related to embedded sensors and equipment – Simulation Environments and Code Manipulation. Unfortunately, these two technologies quickly become very complex to apply, so their reach into the general practitioner testing population is limited.

What most practitioners need to perform more effective mobile and IoT software testing are new tools that directly address the gaps not covered by the existing array of tools.

5.1 Requirements for Next Generation Testing Tools

Therefore, to improve automated testing of mobile and IoT software, new approaches are required that meet the following requirements:

- Requirement 1: Directly target functionality dealing with embedded sensors and equipment.
- Requirement 2: Scale easily to vast variations in data readings, data fidelity, data delivery methods, etc.
- Requirement 3: Constrain technological complexity to be within reach of non-specialists.

Because Simulation Environments and Code Manipulation technologies satisfy Requirements 1 and 2, initial steps to produce new software testing tools are likely to be technically complex and to require programming skills to apply them.

However, even these tools need to be within reach of non-specialists. In my dissertation work, attempting to build a tiny tool of this new generation, I encountered the need to be proficient in hardware device driver implementation, Linux kernel programming, cross-language programming and compilation, graphics subsystem implementation at the OS level, analyzing and fixing complex build dependencies, and navigating and deciphering very large codebases. That is a broad and deep set of technical skills, rare among senior developers and rarer amongst software testers. That skill set is more likely to be satisfied by a team of senior developers and build mavens than by one individual.

Therefore, when I use “non-specialists” in Requirement 3, I mean: Experienced software testers with competent, generalist programming skills but without special technical expertise in any of the subsystems comprising mobile or IoT computing environments and without special technical expertise in the mathematical modelling commonly associated with simulation efforts.

Once several new tools and software testing technologies have been developed to sufficient maturity, it will become possible to package some of those features into much more generally-accessible tools, as has already happened with code coverage tools and style checkers.

5.2 One Vision of New Types of Tools

Extremely briefly, my vision for a new testing technology addressing these requirements combines integration testing and compiler knowledge. Integration testing combines multiple individual units of functionality into features of varying sizes and scopes without involving the application as a whole. Inside the program, data flow for these features begins in specific places and ends in other specific places. Interfaces written for mobile and IoT software communicate with these beginning and ending points, and the compiler knows how to connect these interfaces to the objects in the program’s code that do the work – it has to know this information to successfully build a deployable application.

I envision testing environments that help testers interact with arbitrary paths through the source code by exposing these data-flow beginning and ending points and allowing testers to specify the data details inserted at the data-flow beginning points, as well as any other critical program state. Tests would then conclude after the actual source code processes the data, and the results of that processing would be read at the data-flow ending points. Such compiler-assisted testing tools would allow testers to perform rich Simulation Environment testing on custom slices of functionality, without requiring fully-featured reality-based environments and without requiring testers to understand all the underlying code – just the intent of the feature and how to manipulate data at the endpoints.

6 Conclusion

Mobile and IoT computing operate within a fundamentally different scale and scope of environmental unpredictability than programs in prior computing eras. Accurately assessing the field performance of software for such devices requires testing at Level 3 (Predictable Variations), Level 4 (Discovers Unexpected), and Level 5 (Exhausted Capacities) – yet most existing tools and research function at Level 1 (Ready to Test) or Level 2 (Closely Controlled), with very few stretching partly into Level 3 (Predictable Variations). This is not good enough.

Mobile and IoT computing need better software testing tools, ones that directly handle vast environmental unpredictability and operate easily at great scale. I have one vision for a possible new direction for tool development, but this is a large and complex problem. It needs the skills and experience of many minds brought to bear upon it. What types of tools can you imagine that would be useful to the field?

7 References

Amalfitano, Domenico, Anna Rita Fasolino, Porfirio Tramontana, Salvatore de Carmine, and Gennaro Imparato. 2012. “A toolset for GUI testing of Android applications.” In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, 650–53.

Amalfitano, Domenico, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. 2015. “MobiGUITAR: Automated Model-Based Testing of Mobile Apps.” *IEEE Softw.* 32 (5): 53–59. https://doi.org/10.1109/MS.2014.55.

Android Open Source Project. 2018. “Distribution Dashboard.” (Updates every 7 days.). https://developer.android.com/about/dashboards/.

Austin Test Automation Bazaar. 2016. Various conversations, January 15.

Bauersfeld, Sebastian, and Tanja E. J. Vos. 2014. “User interface level testing with TESTAR; what about more sophisticated action specification and selection?” In *SATToSE*, 60–78.

Beck, Kent. 1999. *eXtreme programming eXplained: Embrace change / Kent Beck.* Reading, MA: Addison-Wesley.

Borjesson, Emil, and Robert Feldt. 2012. “Automated system testing using visual gui testing tools: A comparative study in industry.” In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, 350–59.

Chang, Tsung-Hsiang, Tom Yeh, and Rob Miller. 2011. “Associating the visual representation of user interfaces with their internal structures and metadata.” In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, 245–56.

Chang, Tsung-Hsiang, Tom Yeh, and Robert C. Miller. 2010. “GUI testing using computer vision.” In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1535–44.

Chisholm, Brad L. 1997. “Automated Testing of SAS System GUI Applications.” In *Proceedings of the Twenty-Second Annual SAS®Users Group International Conference*. http://www2.sas.com/proceedings/sugi22/APPDEVEL/PAPER10.PDF.

Creasy, Robert J. 1981. “The origin of the VM/370 time-sharing system.” *IBM Journal of Research and Development* 25 (5): 483–90. ftp://ftp.cis.upenn.edu/pub/cis700-6/public_html/04f/papers/creasy-vm-370.pdf. Accessed May 08, 2018.

Delamaro, M. E., A. M. R. Vincenzi, and J. C. Maldonado. 2006. “A Strategy to Perform Coverage Testing of Mobile Applications.” In *Proceedings of the 2006 International Workshop on Automation of Software Test*, 118–24. AST ’06. New York, NY, USA: ACM. http://doi.acm.org.portal.lib.fit.edu/10.1145/1138929.1138952.

Gao, Jerry, Wei-Tek Tsai, Ray Paul, Xiaoying Bai, and Tadahiro Uehara. 2014. “Mobile Testing-as-a-Service (MTaaS) -- Infrastructures, Issues, Solutions and Needs.” In *High-Assurance Systems Engineering (HASE), 2014 IEEE 15th International Symposium on*, 158–67.

Garousi, Vahid, Wasif Afzal, Adem Çağlar, \.Ihsan Berk Işık, Berker Baydan, Seçkin Çaylak, Ahmet Zeki Boyraz, Burak Yolaçan, and Kadir Herkiloğlu. 2017. “Comparing automated visual GUI testing tools: An industrial case study.” In *Proceedings of the 8th ACM SIGSOFT International Workshop on Automated Software Testing*, 21–28.

Google Tech Talks. *Breaking the Matrix - Android Testing at Scale*. GTAC 2013, 2013. https://www.youtube.com/watch?v=uHoB0KzQGRg.

Griebe, Tobias, and Volker Gruhn. 2014. “A model-based approach to test automation for context-aware mobile applications.” In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, 420–27.

Groder, Chip. 2016. “Objects vs. Images: Choosing the Right GUI Test Tool Architecture.” Unpublished manuscript, last modified March 25, 2018. https://www.stickyminds.com/presentation/objects-vs-images-choosing-right-gui-test-tool-architecture.

Gronli, Tor-Morten, and Gheorghita Ghinea. 2016. “Meeting Quality Standards for Mobile Application Development in Businesses: A Framework for Cross-Platform Testing.” In *System Sciences (HICSS), 2016 49th Hawaii International Conference on*, 5711–20.

Horowitz, Ellis, and Zafar Singhera. 1993. “Graphical user interface testing.” In *Proceedings of the Eleventh Annual Pacific Northwest Software Quality Conference*. Vol. 4, 391–410. http://www.uploads.pnsqc.org/proceedings/pnsqc1993.pdf. Accessed April 18, 2018.

Kaner, Cem. 2017a. Conversation, June 2017.

——— . 2018b, October 11.

Knott, Daniel. 2015. *Hands-on mobile app testing: A guide for mobile testers and anyone involved in the mobile app business.* New York: Addison-Wesley.

Kohl, Jonathan. 2012. *Tap Into Mobile Application Testing:* Leanpub. http://leanpub.com/testmobileapps.

Kropp, Martin, and Pamela Morales. 2010. “Automated GUI testing on the Android platform.” *Testing Software and Systems*, 67.

Meads, Andrew, Habib Naderi, and Ian Warren. 2015. “A Test Harness for Networked Mobile Applications and Middleware.” In *Software Engineering Conference (ASWEC), 2015 24th Australasian*, 1–10.

Muccini, Henry, Antonio Di Francesco, and Patrizio Esposito. 2012. “Software testing of mobile applications: Challenges and future research directions.” In *Proceedings of the 7th International Workshop on Automation of Software Test*, 29–35.

Nagowah, Leckraj, and Gayeree Sowamber. 2012. “A novel approach of automation testing on mobile devices.” In *Computer & Information Science (ICCIS), 2012 International Conference on*. Vol. 2, 924–30.

Nguyen, Bao N., Bryan Robbins, Ishan Banerjee, and Atif Memon. 2014. “GUITAR: An innovative tool for automated testing of GUI-driven software.” *Automated Software Engineering* 21 (1): 65–105.

Nyman, Noel. 1999. “A look at QARun, a GUI test automation tool.” *Software Testing & Quality Engineering* (Jan/Feb): 54–56. https://www.stickyminds.com/better-software-magazine/look-qarun-gui-test-automation-tool. Accessed March 25, 2018.

Oliver, Carol. 2018. “First Steps in Retrofitting a Versatile Software Testing Infrastructure to Android.” Ph.D., Florida Institute of Technology.

OpenSignal. 2015. “Android Fragmentation (August 2015).” Accessed March 25, 2018. http://opensignal.com/reports/2015/08/android-fragmentation/.

Pettus, Sam. 1999. “The History of Emulation and its relationship to computer and videogame history.” https://www.zophar.net/articles/art_14-2.html.

Potter, Richard. 1993. “Guiding Automation with Pixels (Abstract): A Technique for Programming in the User Interface.” In *Proceedings of the INTERACT ’93 and CHI ’93 Conference on Human Factors in Computing Systems*, 530. CHI ’93. New York, NY, USA: ACM. http://doi.acm.org.portal.lib.fit.edu/10.1145/169059.169526.

Ridene, Youssef, and Franck Barbier. 2011. “A model-driven approach for automating mobile applications testing.” In *Proceedings of the 5th European Conference on Software Architecture: Companion Volume*, 9.

Satoh, Ichiro. 2003. “A testing framework for mobile computing software.” *IEEE transactions on software engineering* 29 (12): 1112–21.

St. Amant, Robert, Henry Lieberman, Richard Potter, and Luke Zettlemoyer. 2000. “Programming by Example: Visual Generalization in Programming by Example.” *Commun. ACM* 43 (3): 107–14. https://doi.org/10.1145/330534.330549.

Stocco, Andrea, Maurizio Leotta, Filippo Ricca, and Paolo Tonella. 2014. “PESTO: A tool for migrating DOM-based to visual web tests.” In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, 65–70.

Takala, Tommi, Mika Katara, and Julian Harty. 2011. “Experiences of System-Level Model-Based GUI Testing of an Android Application.” In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, 377–86. ICST ’11. Washington, DC, USA: IEEE Computer Society. http://dx.doi.org.portal.lib.fit.edu/10.1109/ICST.2011.11.

Tissoires, Benjamin, and Stéphane Conversy. 2011. “Hayaku: designing and optimizing finely tuned and portable interactive graphics with a graphical compiler.” In *EICS '11 Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*, edited by Fabio Paternò, Kris Luyten, and Frank Maurer, 117–26.

van der Merwe, Heila, Brink van der Merwe, and Willem Visser. 2012. “Verifying android applications using Java PathFinder.” *ACM SIGSOFT Software Engineering Notes* 37 (6): 1–5.

Vilkomir, Sergiy, and Brandi Amstutz. 2014. “Using combinatorial approaches for testing mobile applications.” In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*, 78–83.

Yan, Dacong, Shengqian Yang, and Atanas Rountev. 2013. “Systematic testing for resource leaks in Android applications.” In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, 411–20.

Yeh, Tom, Tsung-Hsiang Chang, and Robert C. Miller. 2009. “Sikuli: Using GUI screenshots for search and automation.” In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, 183–92.