AFRL-OSR-VA-TR-2014-0338

SAFEGUARDING END-USER MILITARY SOFTWARE

Gregg Rothermel
UNIVERSITY OF NEBRSKA

12/04/2014
Final Report

Air Force Research Laboratory
AF Office Of Scientific Research (AFOSR)/ RTC
Arlington, Virginia 22203
Air Force Materiel Command

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
31-11-2014	Final Report	1-9-2010 -- 31-8-2014

4. TITLE AND SUBTITLE

Safeguarding End-user Military Software

5a. CONTRACT NUMBER

5b. GRANT NUMBER

FA9550-10-1-0406

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Rothermel, Gregg
Cohen, Myra
Dwyer, Matthew
Elbaum, Sebastian
Sarma, Anita

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Nebraska - Lincoln
2200 Vine St, 151 Whittier
Lincoln, NE 68583-0861

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFOSR/RTC
875 N Randolph
Arlington, Va 22203

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Distribution A - Approved for Public Release

13. SUPPLEMENTARY NOTES

14. ABSTRACT

We have performed research on approaches for integrating the of personnel who interact with software systems more directly in a collaborative effort of developing, deploying, validating and maintaining the various components of the systems, with the primary objective of enhancing their dependability. We have created techniques that capture and encode user expertise, verification and other dependability-enhancing techniques that use this expertise to function cost-effectively both collaboratively and across system lifetimes, and mechanisms for better coordinating the development and maintenance processes for these systems.

15. SUBJECT TERMS

Modeling systems, modeling user expectations, testing software, test oracles, performance analysis, combinatorial testing, regre

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**	UU		Gregg Rothermel
U	U	U			**19b. TELEPHONE NUMBER** *(include area code)* (402) 472-2184

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

SAFEGUARDING END-USER MILITARY SOFTWARE

FA9550-10-1-0406

FINAL REPORT
NOVEMBER 30, 2014.

As technology advances we are moving toward an age of military systems of particularly high complexity: ultra-large-scale (ULS) systems – ecosystems involving interdependent systems of systems. ULS systems include not only software, but also the hardware platforms on which the software runs, the people who interact with the systems, and the policies and resources that govern them. People are elements, not just users of the systems, and both affect and are affected by their behavior. Current software engineering techniques fail to address the salient characteristics of ULS systems. Furthermore, they do not integrate the feedback, expertise and preferences of users into the systems, yet these are integral to system behavior.

To address this problem, we have performed research on approaches for integrating the various groups of personnel who interact with ULS systems more directly in a collaborative effort of developing, deploying, validating and maintaining the various components of the systems, with the primary objective of enhancing their dependability. We have created techniques that capture and encode user expertise, verification and other dependability-enhancing techniques that use this expertise to function cost-effectively both collaboratively and across system lifetimes, and mechanisms for better coordinating the development and maintenance processes for these systems.

In the years covered by this final report, we have performed research in all three of the primary focus areas of this grant, including: (A) capturing and modeling user expertise, (B) creating techniques for enhancing dependability, and (C) coordinating development and maintenance processes. We summarize the results of our efforts in each of these three areas, in turn.

1 Capturing and Modeling User Expertise

1.1 Conditional component dependence analysis for distributed robotics software [13]

Modern robotics systems are composed of complex assemblies of hardware and software components. Distributed event-based frameworks are used to facilitate the assembly of software for such systems out of collections of reusable components. These frameworks express component dependencies in data that encode event publish-subscribe relations. This makes it difficult for developers to understand the dependencies and to predict the impacts of a change to a component. Moreover, this encoding of dependencies renders traditional techniques for analyzing component dependencies inapplicable.

We have created a program analysis technique that automatically extracts a model of component dependencies from distributed system source components, but also the conditions under which those dependencies are realized. We have implemented the analysis and applied it to systems developed in ROS. The resulting models are succinct and precise, which suggests that programmers will find them comprehensible, and they can be used to document important global dependencies in a system, to compare different versions to identify the impacts of component changes, and to help locate errors.

1.2 Probabilistic symbolic execution [4]

It is important to develop techniques to capture and model human and environmental components of large systems. These components are often modeled using stochastic or probabilistic techniques since this permits a reasonable level of fidelity even when the underlying mechanisms that generate component behavior are not well understood (e.g. human cognitive processes) or cannot be effectively modeled (e.g., dynamic models of weather). Incorporating such models into system level analysis requires methods that can incorporate information from probabilistic/stochastic components into software analysis. In this work we have layed the foundations for how to achieve this for symbolic execution, a widely studied software analysis technique.

More specifically, we have explored the adaptation of symbolic execution to perform a more quantitative type of reasoning – the calculation of estimates of the probability of executing portions of a program. We have created an extension of the widely used Symbolic PathFinder symbolic execution system that calculates path probabilities. Our approach exploits state-of-the-art computational algebra techniques to count the number of solutions to path conditions, yielding exact results for path probabilities. To mitigate the cost of

using these techniques, we have created two optimizations, PC slicing and count memoization, that significantly reduce the cost of probabilistic symbolic execution. Finally, we have conducted an empirical evaluation applying our technique to challenging library container implementations and shown the benefits that adding probabilities to program analyses may offer.

1.3 Easing the generation of predictive human systems performance models from legacy systems [23]

There has been an increase in tools for predictive human performance modeling – tools that mimic an experienced user and predict how long particular tasks will take to perform on an interface. Often these are used on legacy systems to compare new designs against old, or to compare two competing versions of a similar system. However, the state of the art tools still require a large amount of manual effort because a designer must re-create each interface for analysis. We have created CogTool-Helper as an exemplar of a tool that eases the manual burden on the UI designer. CogTool-Helper leverages research and tools from graphical user interface (GUI) testing community, (automatic UI-model extraction and test case generation), to automatically create storyboards and models, and infer methods to accomplish tasks beyond what the UI designer has specified. A walk through of the approach with experienced UI designers has supported the potential for its use.

1.4 AutoInSpec: Using missing test coverage to improve specifications in GUIs [2]

Developers of a software system's graphical user interface (GUI) often fail to document the interface specifications. Without these, models used for automated test generation and execution, remain imperfect and incomplete. This leads to unexpected behavior that creates unrecoverable situations for test harnesses, and missed coverage. To help with this, we have created AutoInSpec, a technique to infer an important class of specifications, temporal and state-based invariants between GUI events that have been incorrectly modeled. Unlike existing specification mining approaches that require full execution traces or source code, and that mine all invariants, we have simplified the problem by guiding AutoInSpec with coverage criteria and using a previously develop repair framework that builds coverage-adequate test suites, removing un-executable sub-sequences from consideration. These failing subsequences are input to a logic-based inference engine, that returns the missing specifications. We have validated AutoInSpec on a set of well-studied GUI applications.

1.5 Solving the search for source code [21]

Programmers frequently search for source code to reuse using keyword searches. The search effectiveness in facilitating reuse, however, depends on the programmer's ability to specify a query that captures how the desired code may have been implemented. Further, the results often include many irrelevant matches that must be filtered manually. More semantic search approaches could address these limitations, yet existing approaches are either not flexible enough to find approximate matches or require the programmer to define complex specifications as queries. In this work we propose a novel approach to semantic code search that addresses several of these limitations and is designed for queries that can be described using a concrete input/output example. In this approach, programmers write lightweight specifications as inputs and expected output examples. Unlike existing approaches to semantic search, we use an SMT solver to identify programs or program fragments in a repository, which have been automatically transformed into constraints using symbolic analysis, that match the programmer-provided specification. We instantiated and evaluated this approach in subsets of three languages, the Java String library, Yahoo! Pipes mashup language, and SQL select statements, exploring its generality, utility, and trade-offs. The results indicate that this approach is effective at finding relevant code, can be used on its own or to filter results from keyword searches to increase search precision, and is adaptable to find approximate matches and then guide modifications to match the user specifications when exact matches do not already exist. These gains in precision and flexibility come at the cost of performance, for which underlying factors and mitigation strategies are identified.

2 Creating Techniques for Enhancing Dependability

2.1 SimTester: A controllable and observable testing framework for embedded systems [26]

In software for embedded systems, the frequent use of interrupts for timing, sensing, and I/O processing can cause concurrency faults to occur due to interactions between applications, device drivers, and interrupt handlers. This type of fault is considered by many practitioners to be among the most difficult to detect, isolate, and correct, in part because it can be sensitive to execution interleavings and often occurs without leaving any observable incorrect output. As such, commonly used testing techniques that inspect program outputs to detect failures are often ineffective at detecting them. To test for these concurrency faults, test engineers need to be able to control interleavings so that they are deterministic. Furthermore, they also need to be able to observe faults as they occur instead of relying on observable incorrect outputs.

To address this problem we have created SimTester, a framework that allows engineers to effectively test for subtle and non-deterministic concurrency faults by providing them with greater controllability and observability. We have implemented our framework on a commercial virtual platform that is widely used to support hardware/software co-designs to promote ease of adoption. We have evaluated its effectiveness by using it to test for data races and deadlocks. The result shows that our framework can be effective and efficient at detecting these faults.

2.2 Detecting problematic message sequences and frequencies in distributed systems [12]

Testing the components of a distributed system is challenging as it requires consideration of not just the state of a component, but also the sequence of messages it may receive from the rest of the system or the environment. Such messages may vary in type and content, and more particularly, in the frequency at which they are generated. All of these factors, in the right combination, may lead to faulty behavior.

We have created an approach to address these challenges by systematically analyzing a component in a distributed system to identify specific message sequences and frequencies at which a failure can occur. At the core of the analysis is the generation of a test driver that defines the space of message sequences to be generated, the exploration of that space through the use of dynamic symbolic execution, and the timing and analysis of the generated tests to identify problematic frequencies. We have implemented our approach in the context of the popular Robotic Operating System and investigated its application on three systems of increasing complexity.

2.3 Integration testing of software product lines using compositional symbolic execution [17]

Software product lines are families of products defined by feature commonality and variability, with a well-managed asset base. Recent work in testing of software product lines has exploited similarities across development phases to reuse shared assets and reduce test effort. The use of feature dependence graphs has also been employed to reduce testing effort, but little work has focused on code level analysis of dataflow between features. We have developed a compositional symbolic execution technique that works in concert with a feature dependence graph to extract the set of possible interaction trees in a product family. It composes these to incrementally and symbolically analyze feature interactions. Our experiments have shown that our technique can reduce the overall number of interactions that must be considered during testing, and requires less time to run than traditional techniques.

2.4 Using feature locality: Can we leverage history to avoid failures during reconfiguration? [3]

Despite the best efforts of software engineers, faults still escape into deployed software. Developers need time to prepare and distribute fixes, and in the interim deployments must either tolerate or avoid failures. Self-adaptive systems — systems that adapt to meet changing requirements in a dynamic environment — have a daunting task if their reconfiguration involves adding or removing functional features, because configurable software is known to suffer from failures that appear only under certain feature combinations.

Although configuration-dependent failures may be difficult to provoke, and thus hard to detect in testing, we believe that they also constitute opportunities for reconfiguration to increase system reliability. We also believe that the failures that are sensitive to a system configuration depend on similar feature combinations, a phenomenon we call feature-locality, and that this locality can be combined with historical data to predict failure-prone configurations. To investigate these beliefs, we conducted a case study on 128 failures reported against released versions of an open source configurable system. We have found evidence to support both of our hypotheses. We have shown that only a small number of features affect the visibility of these failures and that over time we can learn these features to avoid future failures.

2.5 On the relative strengths of model-based and dynamic event extraction-based GUI testing techniques [1]

Many software systems rely on graphical-user interfaces (GUIs) to support user interactions. The correctness of these GUIs affects the overall quality of the systems, and thus, it is important that GUIs be tested. To support such testing, GUI test case generation techniques based on graph models such as event flow graphs (EFGs) have been used to generate test cases in the form of sequences of events. Models, however, are abstractions of dynamic behavior and may not accurately reflect actual system behavior; thus, test case generation techniques based on models may create nonexecutable test cases and miss important events. Test case generation techniques based on dynamic event extraction-based approaches, in contrast, may suffer less from these effects. As a consequence, we expect that the two approaches will differ in terms of cost and effectiveness. To investigate this expectation, we have conducted an empirical study comparing the cost and effectiveness of model-based and dynamic event extraction-based test case generation approaches. Our results show that event extraction-based approaches, though more expensive than model-based approaches, are indeed more effective in terms of their ability to achieve code coverage and to cover events, but at additional cost, with implications for both researchers and practitioners.

2.6 Oracle-centric test case prioritization [19]

Recent work in testing has demonstrated the benefits of considering test oracles in the testing process. Unfortunately, this work has focused primarily on developing techniques for generating test oracles, in particular techniques based on mutation testing. While effective for test case generation, existing research has not considered the impact of test oracles in the context of regression testing tasks. Of interest here is the problem of test case prioritization, in which a set of test cases are ordered to attempt to detect faults earlier and to improve the effectiveness of testing when the entire set cannot be executed. We have created a technique for prioritizing test cases that explicitly takes into account the impact of test oracles on the effectiveness of testing. Our technique operates by first capturing the flow of information from variable assignments to test oracles for each test case, and then prioritizing to "cover" variables using the shortest paths possible to a test oracle. As a result, we favor test orderings in which many variables impact the test oracle's result early in test execution. Our results demonstrate improvements in rate of fault detection relative to both random and structural coverage based prioritization techniques when applied to faulty versions of three synchronous reactive systems.

2.7 Understanding user understanding: Determining correctness of generated program invariants [18]

Recently, work has begun on automating the generation of test oracles, which are necessary to fully automate the testing process. One approach to such automation involves dynamic invariant generation, which extracts invariants from program executions. To use such invariants as test oracles, however, it is necessary to distinguish correct from incorrect invariants, a process that currently requires human intervention. We have empirically examined this process. In particular, we have studied the ability of 30 users, across two empirical studies, to classify invariants generated from three Java programs. Our results indicate that users struggle to classify generated invariants: on average, they misclassify 9.1% to 31.7% of correct invariants and 26.1%-58.6% of incorrect invariants. These results contradict prior studies that suggest that classification by users is easy, and indicate that further work needs to be done to bridge the gap between the effectiveness of dynamic invariant generation in theory, and the ability of users to apply it in practice.

2.8 On-demand test suite reduction [5]

Most test suite reduction techniques aim to select, from a given test suite, a minimal representative subset of test cases that retains the same code coverage as the suite. Empirical

studies have shown, however, that test suites reduced in this manner may lose fault detection capability. Techniques have been proposed to retain certain redundant test cases in the reduced test suite so as to reduce the loss in fault-detection capability, but these still do concede some degree of loss. Thus, these techniques may be applicable only in cases where loose demands are placed on the upper limit of loss in fault-detection capability. We have created an on-demand test suite reduction approach, which attempts to select a representative subset satisfying the same test requirements as an initial test suite conceding at most $l\%$ loss in fault-detection capability for at least $c\%$ of the instances in which it is applied. Our technique collects statistics about loss in fault-detection capability at the level of individual statements and models the problem of test suite reduction as an integer linear programming problem. We have evaluated our approach in the contexts of three scenarios in which it might be used. Our results show that most test suites reduced by our approach satisfy given fault detection capability demands, and that the approach compares favorably with an existing test suite reduction approach.

2.9 A hybrid directed test suite augmentation technique [24]

Test suite augmentation techniques are used in regression testing to identify code elements affected by changes and to generate test cases to cover those elements. In previous work, we studied two approaches to augmentation, one using a concolic test case generation algorithm and one using a genetic test case generation algorithm. We found that these two approaches behaved quite differently in terms of their costs and their abilities to generate effective test cases for evolving programs. In this work, we present a hybrid test suite augmentation technique that combines these two test case generation algorithms. We report the results of an empirical study that shows that this hybrid technique can be effective, but with varying degrees of costs, and we analyze our results further to provide suggestions for reducing costs.

2.10 Monitoring finite state properties: Algorithmic approaches and their relative strengths [14]

One key class of techniques for enhancing the dependability of end-user military software, especially as it relates to self-healing, is the runtime monitoring of software to detect violations of correctness properties, security policies, and other characterizations of expected usage. In the literature, there have been two basic approaches to monitoring of temporal properties put forth: "state" and "object" based approaches. In this work, we have defined a third orthogonal "symbol" based approach. We have compared the three and discovered several strenghts and weaknesses.

2.11 Compositional load test generation for software pipelines [30]

A key issue in preparing software for acceptable use is not only the correctness of computed results, but also non-functional properties such as performance. We have been exploring test generation techniques that target system performance. Load tests validate whether a systems performance is acceptable under extreme conditions. Traditional load testing approaches are black-box, inducing load by increasing the size or rate of the input. Symbolic execution based load testing techniques complement traditional approaches by enabling the selection of precise input values. However, as the programs under analysis or their required inputs increase in size, the analyses required by these techniques either fail to scale up or sacrifice test effectiveness. We have proposed a new approach that addresses this limitation by performing load test generation compositionally. Our approach uses existing symbolic execution based techniques to analyze the performance of each system component in isolation, summarizes the results of those analyses, and then performs an analysis across those summaries to generate load tests for the whole system. In its current form, the approach can be applied to any system that is structured in the form of a software pipeline. A study of the approach revealed that it can generate effective load tests for Unix and XML pipelines while outperforming state-of-the-art techniques.

2.12 Reducing masking effects in combinatorial interaction testing [25]

Masking effects occur in configurable software systems. This phenomenon is encountered when unknown constraints exist between configurations that are tested together. The tests may fail to run to completion (i.e. they may "skip") despite the lack of a fault. While these configurations appear to be tested, they are in fact only partially tested, due to this masking and thereby reduce the quality of testing. During development and system maintenance, constraints may be added and removed, but these are rarely documented. We have developed an iterative technique (FDA-CIT) for automatically exposing combinations of configurations that mask each other, and then regenerate samples of configurations that avoid those combinations. We have applied this technique on several open source systems to evaluate its effectiveness measured as the increase in both combinatorial and code coverage. FDA-CIT outperforms traditional combinatorial testing as well as another state-of-the-art approach, using error locating arrays.

2.13 Beyond the rainbow: Self-adaptive failure avoidance in configurable systems [22]

Self-adaptive software systems monitor their state and then adapt when certain conditions are met, guided by a global utility function. These may, for instance, guide UAVs to use different navigation strategies during unique environmental conditions. In prior work we developed algorithms and conducted a post-hoc analysis demonstrating the possibility of adapting to avoid software failures by judiciously changing configurations (adding or removing features in the system). In this work we have created the REFRACT framework that realizes this idea in practice by building on the self-adaptive Rainbow architecture. REFRACT extends Rainbow with new components and algorithms targeting failure avoidance. We used REFRACT in a case study running four independently executing Firefox clients . The study showed that workarounds for all but one of the seeded faults are found and the one that is not found never fails because it is guarded.

2.14 PrefFinder: Getting the right preference in configurable software systems [9]

Highly configurable software systems have a large number of preferences that the user can customize, but documentation of them may be scarce or distributed. A user, tester or service technician may have to search through hundreds or thousands of choices in multiple documents when trying to identify which preference will modify a particular system behavior. In this work we created PrefFinder, a natural language framework that finds (and changes) user preferences. It is tied into an application's preference system and static documentation. We instantiated PrefFinder as a plugin on two open source applications, and as a stand-alone GUI for an industrial application. PrefFinder found the correct answer between 76-96% of the time on more than 175 queries. When compared to asking questions on a help forum or through the company's service center, we can potentially save days or even weeks of time.

2.15 Configurations everywhere: Implications for testing and debugging in practice [8]

Many industrial systems are highly-configurable, complicating the testing and debugging process. While researchers have developed techniques to statically extract, quantify and manipulate the valid system configurations by creating models, we conjecture that many of these techniques will fail in practice. In this work we analyzed a highly-configurable industrial application and two open source applications in order to quantify the true challenges that configurability creates for software testing and debugging. We found that (1) all three

applications consist of multiple programming languages, hence static analyses need to cross programming language barriers to work, (2) there are many access points and methods to modify configurations, implying that practitioners need configuration traceability and should gather and merge metadata from more than one source and (3) the configuration state of an application on failure cannot be reliably determined by reading persistent data; a runtime memory dump or other heuristics must be used for accurate debugging. We present a roadmap and lessons learned to help practitioners better handle configurability now, and that may lead to new configuration-aware testing and debugging techniques in the future.

2.16 Amplifying tests to validate exception handling code [31]

Validating code handling exceptional behavior is difficult, as it requires (1) systematic exploration of the space of exceptions that may be thrown by the external resources, and (2) setup of the context to trigger specific patterns of exceptions. In this work we first conducted a study quantifying the magnitude of the problem by inspecting the bug repositories of a set of popular applications in the increasingly relevant domain of Android mobile applications. The study revealed that 22% of the confirmed and fixed bugs have to do with poor exceptional handling code, and half of those correspond to interactions with external resources. We present an approach that addresses this challenge by performing an systematic amplification of the program space explored by a test by manipulating the behavior of external resources. Each amplification attempts to expose a program's exception handling constructs to new behavior by mocking an external resource so that it returns normally or throws an exception following a predefined set of patterns. Our assessment of the approach indicates that it can be fully automated, is powerful enough to detect 67% of the faults reported in the bug reports of this kind, and is precise enough that 78% of the detected anomalies are fixed, and it has a great potential to assist developers.

2.17 Safely reducing the cost of unit level symbolic execution through read/write analysis [15]

Symbolic execution is a powerful tool to systematically explore programs, but it must be applied judiciously to be cost-effective, particularly with large program units. Existing approaches addressing this problem either incur considerable overhead or can skip valuable sequences. In this work we reduce this cost by leveraging a read/write analysis to track possible changes between methods. By identifying calls to a class's fields, sequences involving methods that cannot interact with accesses by previous methods may be pruned. This drastically reduces the cost to determine whether a method call could lead to unique states while considerably reducing the method tree. Preliminary evaluations show our technique can be more efficient than alternative approaches that guarantee complete path coverage.

2.18 Reducing failure rates of robotic systems though inferred invariants monitoring [7]

System monitoring can help to detect abnormalities and avoid failures. Crafting monitors for today's robotic systems, however, can be very difficult due to the systems' inherent complexity. In this work we address this challenge through an approach that automatically infers system invariants and synthesizes those invariants into monitors. The approach is novel in that it derives invariants by observing the messages passed between system nodes and the invariants types are tailored to match the spatial, temporal, and operational attributes of robotic systems. Further, the generated monitor can be seamlessly integrated into systems built on top of publish-subscribe architectures. An application of the technique on a system consisting of a unmanned aerial vehicle (UAV) landing on a moving platform shows that it can significantly reduce the number of crashes in unexpected landing scenarios.

2.19 SimRT: An automated framework to support regression testing for data races [29]

Concurrent programs are prone to various classes of difficult-to-detect faults, of which data races are particularly prevalent. Prior work has attempted to increase the cost-effectiveness of approaches for testing for data races by employing race detection techniques, but to date, no work has considered cost-effective approaches for re-testing for races as programs evolve. In this work we developed SimRT, an automated regression testing framework for use in detecting races introduced by code modifications. SimRT employs a regression test selection technique, focused on sets of program elements related to race detection, to reduce the number of test cases that must be run on a changed program to detect races that occur due to code modifications, and it employs a test case prioritization technique to improve the rate at which such races are detected. Our empirical study of SimRT reveals that it is more efficient and effective for revealing races than other approaches, and that its constituent test selection and prioritization components each contribute to its performance.

2.20 SimLatte: A framework to support testing for worst-case interrupt latencies in embedded software [27]

Embedded systems tend to be interrupt-driven, yet the presence of interrupts can affect system dependability because there can be delays in servicing interrupts. Such delays can occur when multiple interrupt service routines and interrupts of different priorities compete for resources on a given CPU. For this reason, researchers have sought approaches by which to estimate worst-case interrupt latencies (WCILs) for systems. Most existing ap-

proaches, however, are based on static analysis. In this work, we developed SIMLATTE, a testing-based approach for finding WCILs. SIMLATTE uses a genetic algorithm for test case generation that converges on a set of inputs and interrupt arrival points that are likely to expose WCILs. It also uses an opportunistic interrupt invocation approach to invoke interrupts at a variety of feasible locations. Our evaluation of SIMLATTE on several non-trivial embedded systems reveals that it is considerably more effective and efficient than random testing. We also determine that the combination of the genetic algorithm and opportunistic interrupt invocation allows SIMLATTE to perform better than it can when using either one in isolation.

2.21 An empirical comparison of the fault-detection capabilities of internal oracles [28]

Modern computer systems are prone to various classes of runtime faults due to their reliance on features such as concurrency and peripheral devices such as sensors. Testing remains a common method for uncovering faults in these systems, but many runtime faults are difficult to detect using typical testing oracles that monitor only program output. In this work we empirically investigate the use of internal test oracles: oracles that detect faults by monitoring aspects of internal program and system states. We compare these internal oracles to each other and to output-based oracles for relative effectiveness and examine tradeoffs between oracles involving incorrect reports about faults (false positives and false negatives). Our results reveal several implications that test engineers and researchers should consider when testing for runtime faults.

3 Coordinating Development and Maintenance Processes

3.1 The onion patch: Migration in open source ecosystems [6]

Today's systems form an ecosystem of related projects and products that have common underlying components, technology, and social norms. Understanding how the common technical and social structures of related projects in such systems allow newcomers to easily join a project is essential in understanding how expertise is transferred in larger software ecosystems. Typical migration process in a community (e.g., Open Source community) follows a socialization process called "the onion model", where newcomers join a project by first contributing at the periphery through mailing list discussions and bug trackers and as they develop skill and reputation within the community they advance to central roles of

contributing code and making design decisions. However, participants in an ULS ecosystem may be able to utilize a significant amount of transferable knowledge when moving between projects in the ecosystem and, thereby, skip steps in the onion model. We have shown how developers are able to leverage their knowledge of technical as well as social structures in one project and apply it to another project, and are thereby able to reduce their socialization (training) times.

3.2 End-user programmers and their communities: An artifact-based analysis [20]

End-user programmers outnumber professionals programmers, write software that matters to an increasingly large number of users, and face software engineering challenges that are similar to their professionals counterparts. Yet, we know little about how these end-user programmers create and share artifacts as part of a community. To gain a better understanding of these issues, we have performed an artifact-based community analysis of 32,000 mashups from the Yahoo! Pipes repository. We observed that, like with other online communities, there is great deal of attrition but authors that persevere tend to improve over time, creating pipes that are more configurable, diverse, complex, and popular. We also discovered, however, that end-user programmers employ the repository in different ways than professionals, do not effectively reuse existing programs, and in most cases do not have an awareness of the community.

3.3 History repeats itself more easily when you log it: Versioning for mashups [10]

Web mashup environments provide a way for professional and end-user programmers to combine data from various web applications and services to create new content. By making mashups available in repositories, these environments also help mashup programmers understand and reuse mashups. Current mashup environments, however, do not provide support for tracking the development histories of mashups. To address this lack, we have been working to add configuration management support to the Yahoo! Pipes mashup environment. We have presented results of a controlled experiment studying both end-user and experienced programmers, in terms of their ability to create and debug mashups in the presence of that support. Our results show that versioning support can help both groups of users do both tasks better.

3.4 Development context driven change awareness and analysis framework [16]

Distributed software development requires coordination across members of the team, who are often isolated from each other and performing development tasks in parallel. In such situations a developer needs to understand how his changes may impact other ongoing changes and how changes made by other developers may impact his tasks. Traditional change impact analysis techniques identify the impact of a change set on the original code base, whereas conflict prediction requires the computation of a developers changes on not only the central code base, but, also on the other ongoing changes in remote, parallel workspaces. Designing the right change impact analysis technique that can assist in development tasks requires determining (a) how to perform an analysis when there are no changes, (b) which versions should be treated as source and target programs when there are many developers, (c) how to bound the scope of the analysis to answer specific questions and facilitate scalability, (d) how to configure the precision of the analysis such that it provides meaningful results in a timely manner, and (e) how to process and present the results such that they are useful to the developers. In this work, we propose a novel context driven change awareness and analysis framework, Development Context Analysis Framework (DeCAF) that leverages the context of the distributed software development environment to scope the analysis space. DeCAF can be configured to scope the region over which impact is calculated, the precision of the analysis, and the extent of analysis based on the client requirements. DeCAF uses a multi-stage change impact analysis as the underlying analysis engine.

3.5 On the benefits of providing versioning support for end-users: An empirical study [11]

End users with little formal programming background are creating software in many different forms, including spreadsheets, web macros, and web mashups. Web mashups are particularly popular because they are relatively easy to create, and because many programming environments that support their creation are available. These programming environments, however, provide no support for tracking versions or provenance of mashups. We believe that versioning support can help end users create, understand, and debug mashups. To investigate this belief, we have added versioning support to a popular wire-oriented mashup environment, Yahoo! Pipes. Our enhanced environment, which we call "Pipes Plumber," automatically retains versions of pipes and provides an interface with which pipe programmers can browse histories of pipes and retrieve specific versions. We have conducted two studies of this environment: an exploratory study and a larger controlled experiment. Our results provide evidence that versioning helps pipe programmers create and debug mashups. Subsequent qualitative results provide further insights into the barriers faced by pipe programmers, the support for reuse provided by our approach, and the support for debugging provided.

References

[1] G. Bae, G. Rothermel, and D. H. Bae. On the relative strengths of model-based and dynamic event extraction-based GUI testing techniques: An empirical study. In *Proceedings of the International Symposium on Software Reliability Engineering*, November 2012, (to appear).

[2] M.B. Cohen, S. Huang, and A. M Memon. AutoInSpec: Using missing test coverage to improve specifications in GUIs. In *Proceedings of the International Symposium on Software Reliability Engineering*, November 2012 (to appear).

[3] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Using feature locality: Can we leverage history to avoid failures during reconfiguration? In *Proceedings of the ESEC/FSE Workshop on Assurances for Self-Adaptive Systems*, pages 24–33, September 2011.

[4] J. Geldenhuys, M. B. Dwyer, and W. Visser. Probabilistic symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 166–176, July 2012.

[5] D. Hao, L. Zhang, X. Wu, H. Mei, and G. Rothermel. On-demand test suite reduction. In *Proceedings of the International Conference on Software Engineering*, pages 738–748, June 2012.

[6] C. Jergensen, A. Sarma, and P. Wagstrom. The onion patch: Migration in open source ecosystems. In *Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 70–80, September 2011.

[7] H. Jiang, S. Elbaum, and C. Detweiler. Reducing failure rates of robotic systems though inferred invariants monitoring. In *IROS*, 2013.

[8] D. Jin, X. Qu, M. B. Cohen, and B. Robinson. Configurations everywhere: Implications for testing and debugging in practice. In *Proceedings of the International Conference on Software Engineering*, June 2014.

[9] D. Jin, X. Qu, M. B. Cohen, and B. Robinson. PreFinder: Getting the right preference in configurable software systems. In *Proceedings of Automated Software Engineering*, September 2014.

[10] S. K. Kuttal, A. Sarma, and G. Rothermel. History repeats itself more easily when you log it: Versioning for mashups. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 69–72, September 2011.

[11] Sandeep K. Kuttal, Anita Sarma, and Gregg Rothermel. On the benefits of providing versioning support for end users: An empirical study. *ACM Transactions on Computer-Human Interaction*, 21(2):9:1–9:43, 2014.

[12] C. Lucas, S. Elbaum, and D. S. Rosenblum. Detecting problematic message sequences and frequencies in distributed systems. In *Proceedings of the International Conference on Object Oriented Programming, Systems, Languages and Applications*, October 2012, (to appear).

[13] R. Purandare, J. Darsie, S. Elbaum, and M. B. Dwyer. Conditional component dependence analysis for distributed robotics software. In *Proceedings of the 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, October 2012, (to appear).

[14] R. Purandare, M. B. Dwyer, and S. Elbaum. Monitoring finite state properties: Algorithmic approaches and their relative strengths. In *Proceedings of the 2nd International Conference on Runtime Verification*, pages 381–395, September 2011.

[15] E. F. Rizzi, M. B. Dwyer, and S. B. Elbaum. Safely reducing the cost of unit level symbolic execution through read-write analysis. *ACM SIGSOFT Software Engineering Notes*, 39(1), 2014.

[16] A. Sarma, B. Branchaud, M. B. Dwyer, S. Person, and N. Rungta. Development context driven change awareness and analysis framework. In *Proceedings of the International Conference on Software Engineering*, June 2014.

[17] J. Shi, M. B. Cohen, and M. B. Dwyer. Integration testing of software product lines using compositional symbolic execution. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, pages 270–284, March 2012.

[18] M. Staats, S. Hong, M. Kim, and G. Rothermel. Understanding user understanding: Determining correctness of generated program invariants. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 188–198, July 2012.

[19] M. Staats, P. Loyola, and G. Rothermel. Oracle-centric test case prioritization. In *Proceedings of the International Symposium on Software Reliability Engineering*, November 2012, (to appear).

[20] K. T. Stolee, S. Elbaum, and A. Sarma. End-user programmers and their communities: An artifact-based analysis. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, pages 147–156, September 2011.

[21] K. T. Stolee, S. G. Elbaum, and D. Dobos. Solving the search for source code. *ACM Transactions on Software Engineering and Methodology*, 23(3), 2014.

[22] J. Swanson, M. B. Cohen, M. B. Dwyer, B. J. Garvin, and J. Firestone. Beyond the rainbow: Self-adaptive failure avoidance in configurable systems. In *Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering*, November 2014.

[23] A. Swearngin, M. B. Cohen, B. E. John, and R. K. E. Bellamy. Easing the generation of predictive human systems performance models from legacy systems. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 2489–2498, May 2012.

[24] Z. Xu, Y. Kim, M. Kim, and G. Rothermel. A hybrid directed test suite augmentation technique. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 150 – 159, November 2011.

[25] C. Yilmaz, E. Dumlu, M. B. Cohen, and A. A. Porter. Reducing masking effects in combinatorial interaction testing: A feedback driven adaptive approach. *IEEE Transactions on Software Engineering*, 41(1):43–66, 2014.

[26] T. Yu, W. Srisa-an, and G. Rothermel. SimTester: A controllable and observable testing framework for embedded systems. In *Proceedings of the ACM International Conference on Virtual Execution Environments*, pages 51–62, March 2012.

[27] Tingting Yu, Witawas Srisa-an, Myra B. Cohen, and Gregg Rothermel. SimLatte: A framework to support testing for worst-case interrupt latencies in embedded software. In *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation*, pages 313–322, 2014.

[28] Tingting Yu, Witawas Srisa-an, and Gregg Rothermel. An empirical comparison of the fault-detection capabilities of internal oracles. In *Proceedings of the International Conference on Software Reliability Engineering*, pages 11–20, 2013.

[29] Tingting Yu, Witawas Srisa-an, and Gregg Rothermel. SimRT: An automated framework to support regression testing for data races. In *Proceedings of the 36th International Conference on Software Engineering*, pages 48–59, 2014.

[30] P. Zhang, S. Elbaum, and M. B. Dwyer. Compositional load test generation for software pipelines. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 89–99, July 2012.

[31] P. Zhang and S. G. Elbaum. Amplifying tests to validate exception handling code: An extended study in the mobile application domain. *ACM Transactions on Software Engineering and Methodology*, 23(4), 2014.