Obtaining and Reasoning About Good Enough Software

Martin Rinard
MIT EECS, MIT CSAIL
rinard@csail.mit.edu

Abstract

Software systems often exhibit a surprising flexibility in the range of execution paths they can take to produce an acceptable result. This flexibility enables new techniques that augment systems with the ability to productively tolerate a wide range of errors. We show how to exploit this flexibility to obtain transformations that improve reliability and robustness or trade off accuracy in return for increased performance or decreased power consumption. We discuss how to use empirical, probabilistic, and statistical reasoning to understand why these techniques work.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Error Handling and Recovery

General Terms Reliability, Security, Verification

Keywords Recovery, Fault, Error

1. Introduction

A primary goal of many software development projects is to produce a system that is as close to correct as possible (in the sense that it contains as few errors as possible). In support of this goal, the programming languages and software engineering communities have invested significant time and effort developing techniques to either ensure the absence of errors in the system or to detect errors before the system is deployed (which the developers would then presumably correct before deployment).

In this position paper we present an alternate perspective. Instead of viewing systems as correct or incorrect, instead of viewing actions that the system takes as correct actions or errors, we instead propose to take a broader, more general perspective. This perspective focuses on systems and actions as acceptable or unnacceptable. Unless an action causes the system to behave in an unacceptable way, we may see no need to classify the action as an error or the system as incorrect. And even if the action does cause the system to behave unacceptably, it is often possible to apply a simple modification that (while not eliminating the error) rehabilitates the action to have an acceptably benign effect on the overall behavior of the system.

1.1 Good Enough Software

So while we may not have correct software, or even software that a traditional software engineer would call good, we can obtain *good enough* software. And good enough software can be far better than software that aspires (and inevitably fails) to be correct when one considers broader aspects such as development cost, performance, robustness, reliability, and fault tolerance.

This perspective makes new techniques, optimizations, and approaches available to us. Freed from the burden of developing correct systems, we can instead focus on developing systems that best satisfy a range of desirable properties. We can appropriately invest engineering effort where it is most effectively deployed — if certain kinds of correctness are not directly relevant, we have the freedom to invest only as much engineering effort as necessary to produce an acceptable, good enough, but not necessarily correct system.

1.2 Obtaining Good Enough Software

With this perspective, we can use acceptably incorrect components with no modifications whatsoever. Given an unacceptably incorrect component, we can apply simple transformations that rehabilitate the incorrectness to give us an acceptably incorrect component. Given an overly engineered or rigid correct component, we can apply transformations that relax the correctness to obtain other benefits such as robustness, reliability, performance, or reduced resource consumption. Examples of such transformations include the following:

- **Precondition Expansion:** Many components execute correctly only if their inputs satisfy certain preconditions. In some systems it may be desirable to use the component with inputs that violate the preconditions. Precondition expansion transforms the component so that it can survive any otherwise fatal errors that might occur when given an input that does not conform to the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2012, June 3-7, 2012, San Francisco, California, USA.
Copyright © 2012 ACM 978-1-4503-1199-1/12/06...$10.00

preconditions. Examples of such transformations include infinite loop termination [3] and failure-oblivious computing [20]. Such techniques enable the component to generate (ideally acceptable) outputs even for inputs that do not satisfy the precondition. They also enable the component to survive to process additional inputs.

- **Input Rectification:** Instead of modifying the component to process inputs that violate its precondition, *input rectification* instead modifies inputs so that they satisy the precondition [8]. In many cases, input rectification preserves most or even all of the useful information in the input, nullifies otherwise fatal vulnerabilities, and enables the component to produce acceptable outputs even for otherwise problematic inputs.
- **Discarding Computation:** Task skipping [16, 17], loop perforation [10, 22], and reduction sampling [23] discard computations. When appropriately applied, the result is a significant reduction in the amount of computational resources (time or energy) required to complete the computation combined with an acceptably small change in the output that the computation produces.
- **Removing Functionality:** Many systems build on general-purpose software bases that provide more functionality than the system requires. This excess functionality can make the system vulnerable to security attacks and prone to exhibit irrelevant behaviors. Indeed, acceptable functionality may be available with a fraction of the code in the original implementation [18]. Automatically eliminating excess functionality can shrink the size of the code base and eliminate undesirable unanticipated behaviors.
- **Race-Full Parallelization:** Data races are often seen as unacceptable behavior [1]. The facts show, however, that many acceptable parallelizations have data races [9, 15]. Advantages of considering computations with data races include the elimination of synchronization overhead [15] and compilers that can automatically parallelize a much broader range of computations [9].
- **Data Structure Repair:** If a system's data structures violate key consistency properties, a system can produce unacceptable outputs or crash. Data structure repair detects and repairs corrupted data structures [4–6]. In many cases this technique can rehabilitate the error that originally caused the corruption, enabling the system to generate acceptable output for the input that caused the corruption and (in many cases more importantly) continue to execute to successfully provide service to its clients.

1.3 Critical and Forgiving Regions and Developer Observation Bias

In our experience, many developers are surprised that the techniques we outline above can improve software systems — the perception is that the system must walk a narrow path to execute correctly and that any deviation from this path is likely to cause the system to fail. Our experimental results show that, at least for the benchmark systems that we use in our experiments, this perception is simply false.

So why do some developers have this incorrect perception? Our hypothesis is that observation bias is a large part of the reason. Our results indicate that systems usually have *critical* regions, which must be close to correct for the system to operate acceptably, and *forgiving* regions, which can tolerate significant changes [2, 10, 16, 17, 21, 22]. Every developer has encountered a memorable situation (typically associated with debugging) in which a small change to the system caused large changes to its behavior. Our hypothesis is that these memorable events often involve errors in the critical regions of the system. Errors in forgiving regions, to the extent that developers notice them at all, may have much less memorable consequences. This experience may bias the perceptions of some developers and impair their ability to conceive of, understand, and realize the significant benefits that are available from the techniques outlined above.

1.4 Reasoning About Good Enough Software

So how might we help such developers obtain a more balanced understanding of the systems that they develop? And how might we develop better explanations for the reasons why systems exhibit these surprising characteristics?

We present several different reasoning approaches that can help explain results that we have observed in existing systems and predict outcomes across a broader range of systems. We consider different transformations in turn and reason about the interaction of the system with each of these transformations. Depending on the transformation, the system, and the usage context, different reasoning approaches may be appropriate.

2. Empirical Reasoning

With empirical reasoning, we apply the transformation, then use executions on representative inputs to explore the effect of the transformation. This is essentially a form of software testing, which is currently the dominant way to validate software systems. An advantage of this approach is its universality — it is possible to apply it to virtually any transformation and any system.

As with any reasoning approach based on empirical observations, a question that can arise is how to generalize the reasoning to other systems and other inputs. Our initial approach analyzes the implementation of the system to understand why the transformation produces an acceptable system. This analysis often enables us to recognize general system properties that make the system interact well with the transformation. When given a new system or input, we can then analyze the system (potentially in the context if the input) to understand if it satisfies these properties.

2.1 Failure-Oblivious Computing

Failure-oblivious computing is a technique that renders systems oblivious to memory access errors such as out of bounds accesses or null pointer dereferences [20]. We have explored two techniques for out of bounds writes: discarding the write and modulo writes (in which each out of bounds access wraps back around to write a location in the accessed data block). We have also explored two similar techniques for out of bounds reads: manufactured values (which makes up values for out of bounds reads) and modulo reads (in which the out of bounds access wraps back around to read a location in the accessed data block). Our empirical results indicate that, for a range of applications, failure-oblivious computing can eliminate security vulnerabilities and enable applications to survive otherwise fatal memory accessing errors.

Failure-oblivious computing works well for applications with short error propagation distances. In many servers, for example, the computations that process each request are largely independent. Failure-oblivious computing can eliminate data structure corruption and prevent the server from crashing when it encounters an out of bounds access or null pointer dereference. The server can then survive to successfully process subsequent requests. Modulo accesses can be effective in ensuring that the server observes values that conform to the data structure consistency constraints even for out of bounds accesses. Consider, for example, out of bounds accesses to an array of structures. Modulo reads redirect the accesses back into the array to observe an existing structure that will typically conform to the consistency constraints.

We anticipate that failure-oblivious computing will also work well with self-stabilizing computations, which, as long as they survive and continue to execute, eventually discard the effect of any errors or perturbations [4–6].

Finally, we anticipate that failure-oblivious computing may be appropriate in any situation with a need for continued execution. For example, it may be critical in ensuring the continued execution of systems that control unstable physical phenomema. To cite one example, simply ignoring arithmetic overflow and using whatever value was produced would have eliminated the cause of the Ariane 5 launch failure [7].

2.2 Boundless Memory Blocks

Boundless memory blocks store out of bounds writes in a hash table for retrieval when the system generates a corresponding out of bounds read [19]. With this technique, each memory block is conceptually unbounded, with its initial range implemented efficiently with a continguous block of memory. Boundless memory blocks work well when the developer has produced a program that is mostly correct but produces data block sizes that are smaller than some executions require.

More generally, systems often have multiple interacting aspects, each of which must operate acceptably for the system as a whole to operate acceptably. Because of the redundancy between aspects, it may be possible to use the behavior of one aspect to adjust another aspect to become more correct. It is possible, for example, to examine the array accessing patterns of applications to find out of bounds accesses that expose errors in the computation of the required array size [14]. Using the offset of the out of bounds index to compute a new, larger, array size may (but, unlike boundless memory blocks, is not guaranteed to) eliminate the out of bounds accesses.

2.3 Data Structure Repair

Data structure repair finds data structures that violate key consistency constraints, then modifies the data structures to eliminate the inconsistency [4–6]. Note that there is no guarantee that the repair will create the data structure that a (hypothetical) correct execution would have produced — the error that caused the inconsistency may have destroyed information required to obtain this data structure, or the repair algorithm may be unable to determine which of several alternative consistent data structures the correct execution would have generated. Nevertheless, the results show that data structure repair can restore acceptable, if not perfect, execution and enable the system to continue to execute productively. A key aspect of this technique (like failure-oblivious computing) is that it ensures consistent (even though perhaps not perfect) data structures, prevents the system from crashing, and enables the system to continue to provide service. The general pattern that repeatedly emerges is that continued execution with consistent data structures, regardless of the specific mechanism used to obtain this continued execution, typically delivers acceptable results.

2.4 Cyclic Memory Allocation

Cyclic memory allocation eliminates memory leaks by allocating a fixed-size buffer, then cyclically allocating memory out of that buffer [13]. With this technique, it is possible to allocate two objects into the same slot in the buffer, in effect overlaying live data. Maintaining a separate buffer for each different allocation site tends to ensure that each individual object in the buffer preserves the basic consistency constraints for that object (but not necessarily consistency constraints that involve linked relationships between objects). An examination of the behavior of the systems after overlaying indicates that this form of consistency facilitates continued acceptable execution.

Our results show that, when cyclic memory allocation overlays live objects, the system may lose some functionality, but typically continues to execute acceptably for many inputs. An analysis of the system also indicates that preserving basic object integrity constraints in the face of overlaid live data (by maintaining separate buffers for different allocation sites) facilitates this acceptable continued execution.

Cyclic memory allocation is conceptually similar to failure-oblivious computing and data structure repair in that it is designed to rehabilitate otherwise fatal errors to keep the system executing acceptably although not necessarily perfectly. it differs in that the threat to the application is different. Instead of a single error that kills the application immediately (like a heart attack), memory leaks are a form of unbounded resource consumption that (more like cancer) eventually monopolizes all of the resources that the system needs to survive.

2.5 Infinite Loop Termination

Infinite loops can cause systems to become unresponsive. Infinite loop termination techniques detect (in some cases only likely) infinite loops, then exit the loop [3, 21]. One approach compares states before and after loop iterations to detect repeated states [3]. Another approach learns how many iterations loops typically execute, then terminates loops after they exceed this number of iterations by some conservative factor [21]. After exiting the infinite loop, the system can then proceed on to perform the rest of the computation required to generate the anticipated output. Our results show that this continued execution typically produces a better outcome than the alternative (terminating the program).

Like cyclic memory allocation, infinite loop termination eliminates an (effectively fatal) unbounded resource consumption problem — cyclic memory allocation eliminates the fatal monopolization of memory; infinite loop termination eliminates the fatal monopolization of the program counter (which must typically be shared between different parts of the system for the system to produce acceptable outputs).

2.6 Immortal Systems

It is possible to combine failure-oblivious computing, cyclic memory allocation, and infinite loop termination to obtain a conceptually (at the software level) immortal system. Specifically, the system will keep executing, will not exhaust memory, and will not become stuck in an infinite loop (of course, we provide a way for the developer to specify that a specific loop, for example the main control loop of the system, should never terminate). The acceptability of the results that such an immortal system will produce will vary depending on the system and the context in which it is used. However, our results show that, when augmented with such techniques, systems often have a surprising ability to produce acceptable outcomes even in the fact of otherwise fatal errors.

2.7 Injected Errors

Given the success of these techniques in enabling systems to tolerate otherwise fatal errors, a natural question to ask is How many errors can the system contain and still execute acceptably? We explored this question by injecting errors into the source code of the system, then using various techniques to ensure that the system executes through the errors [21]. Our specific error injection mechanism changed loop termination conditions to simulate off by one errors. Our results indicate that software systems with these injected off by one errors often execute acceptably even when the errors visibly perturb the execution.

2.8 Task Skipping and Loop Perforation

Inspired by our success in enabling programs to tolerate off by one errors, we next explored transformations designed to increase robustness and performance. Two transformations include skipping tasks in parallel programs [16, 17] and skipping iterations of time-consuming loops [10, 22]. The motivation is to discard pieces of computation that contain errors (thereby preserving the integrity of the system and enabling it to survive the error) or to reduce the amount of computational resources required to obtain the result. Our results show that this technique can deliver significant improvements in robustness and performance at the cost of small changes in the result that the system produces.

2.9 Critical and Forgiving Code and Data

One of the results of this research was the distinction between critical and forgiving code and data. Our results indicated that systems typically contain some components that must be essentially perfect for the system to execute acceptably. Other components can, if appropriately augmented with techniques such as failure-oblivious computing that enable the system to execute through errors, tolerate significant imperfection or transformations that significantly change what the component does [2, 10, 16, 17, 21, 22].

2.10 Developer Observation Bias

In our experience many software developers view systems as walking a single narrow correct execution path, with any deviation from this path causing the system to execute incorrectly. This belief has produced software development approaches that focus on bringing systems as close to perfect as possible — after all, if the slightest deviation from correct execution is unacceptable, anything less than perfection is simply pointless. Another counterproductive consequence of this belief is underinvestment in techniques that enable systems to tolerate errors — after all, if only the correct execution is acceptable, techniques that attempt to rehabilitate systems when they diverge from the correct path are irrelevant.

Our experimental results show that this understanding of software systems is simply incorrect — our results demonstrate, time and again, that software systems, when appropriately transformed to better tolerate unanticipated errors, exhibit remarkable flexibility in generating acceptable results across a large range of behaviors.

So why do some software professionals believe something that is simply wrong? Observation bias may account for part of this misconception. Every developer has encountered situations in which a very small change to the source code of the software system has a huge impact on the overall behavior. Developers may be (mistakenly) generalizing from this experience to conclude that *any* small change will make a large difference. The concept of critical and forgiving regions may be particularly important here [2, 10, 16, 17, 21, 22]. Our results indicate that programs tend to have critical regions which must be perfect (or close to perfect) for the system to execute acceptably. It is our hypothesis that errors in these critical regions shape some developers' beliefs about the need for perfection in software systems — errors in forgiving regions typically have less memorable effects and may even go largely unnoticed.

3. Probabilistic Reasoning

Probabilistic reasoning models uncertainty about various aspects of the system and its execution (for example, the values of input variables or the local effect of certain transformations), then reasons how this uncertainty may affect the execution of the system and the results that it produces. In our research we have focused on obtaining probabilistic bounds of the form $\Pr(e > b) < p$, where e is a measure of the inaccuracy of the transformed computation, b is a bound on the inaccuracy, and p is an upper bound on the probability with which the inaccuracy e exceeds the inaccuracy bound b [12, 23]. The analyzed transformations include loop perforation [12] and the combination of approximate function substitution (using less accurate but more efficient implementations of functions) and reduction sampling (approximating a reduction using only a subset of the inputs to the reduction) [23].

An advantage of probabilistic reasoning is that it provides guarantees that are quantified over all inputs and all executions. This universal quantification is important because the guarantee characterizes all system behaviors, not just those exposed via representative inputs.

4. Statistical Reasoning

Our statistical reasoning uses families of mathematical objects to model aspects of the transformed system. We then use observations from representative executions to select a specific mathematical object, or, more generally, a set of mathematical objects, that characterize the transformed computation. For example, we use multiple linear regression to model the effect of task skipping on the accuracy of the result that the system produces [16, 17]. Starting with observations from representative executions, the regression algorithm computes linear coefficients to obtain a single linear model for the effect of task skipping on the system. We have also used statistical approaches to select appropriate probability distributions to model the values that perforated loops manipulate [11]. With these probability distributions, we then use probabilistic reasoning to model the effect of loop perforation.

As these examples illustrate, our statistical approaches combine elements of both the probabilistic approach (they produce probabilistic models of the transformed system) and the empirical approach (they rely on observations from representative executions to select the final model).

5. Future Directions

At this point we have accumulated significant empirical evidence that systems have a significant degree of flexibility in the computation they execute to produce an acceptable result. Systems typically have both critical parts, which exhibit little flexibility to vary their execution, and forgiving parts, which exhibit substantial flexiblity. Our results show that empirical test executions on transformed programs can effectively separate critical and forgiving regions [2, 10, 16, 17, 22]. These results, along with our manual analysis of the behavior of the transformed systems, also show that empirical techniques can identify transformations that are appropriate for all inputs and not just those inputs used in the representative executions used to evaluate the effect of the transformations.

The next step is to develop more general explanations for these phenomena. We have already obtained the first results in this area, which use probabilistic and statistical reasoning to model systems which exhibit these phenomena. But these results, as important as they may be, only explain a few classes of behaviors. As this new approach to program analysis and transformation continues to evolve, we anticipate the development of increasingly sophisticated techniques that explain ever broader ranges of techniques. And we also anticipate the development of new and more powerful techniques for productively tolerating errors and optimizing various aspects of (not necesssarily perfect) systems. While the resulting systems may not be correct or even good, they will be better than correct and better than good — they will be good enough. An exciting and interesting time to be working in this area!

References

[1] H. Boehm and S. Adve. You don't know jack about shared variables or memory models. *Commun. ACM*, 55(2), 2012.

[2] M. Carbin and M. C. Rinard. Automatically identifying critical input regions and code in applications. In *ISSTA*, pages 37–48, 2010.

[3] M. Carbin, S. Misailovic, M. Kling, and M. C. Rinard. Detecting and escaping infinite loops with jolt. In *ECOOP*, pages 609–633, 2011.

[4] B. Demsky and M. C. Rinard. Automatic detection and repair of errors in data structures. In *OOPSLA*, pages 78–95, 2003.

[5] B. Demsky and M. C. Rinard. Data structure repair using goal-directed reasoning. In *ICSE*, pages 176–185, 2005.

[6] B. Demsky and M. C. Rinard. Goal-directed reasoning for specification-based data structure repair. *IEEE Trans. Software Eng.*, 32(12):931–951, 2006.

[7] J. L. Lions. Ariane 5 flight 501 failure report by the inquiry board, July 1996. URL `http://www.di.unito.it/ damiani/ariane5rep.html`.

[8] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and M. Rinard. Automatic input rectification. In *ICSE*, 2012.

[9] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. Technical Report MIT-CSAIL-TR-2010-038, MIT, 2010.

[10] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. C. Rinard. Quality of service profiling. In *ICSE (1)*, pages 25–34, 2010.

[11] S. Misailovic, D. Roy, and M. Rinard. Probabilistic and statistical analysis of perforated patterns. Technical Report MIT-CSAIL-TR-2011-003, MIT, 2011.

[12] S. Misailovic, D. M. Roy, and M. C. Rinard. Probabilistically accurate program transformations. In *SAS*, pages 316–333, 2011.

[13] H. H. Nguyen and M. C. Rinard. Detecting and eliminating memory leaks using cyclic memory allocation. In *ISMM*, pages 15–30, 2007.

[14] G. Novark, E. Berger, and B. Zorn. Exterminator: Automatically correcting memory errors with high probability. In *PLDI*, 2007.

[15] M. Rinard. A lossy, synchronization-free, race-full, but still acceptably accurate parallel space-subdivision tree construction algorithm. Technical Report MIT-CSAIL-TR-2012-005, MIT, 2012.

[16] M. C. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *ICS*, pages 324–334, 2006.

[17] M. C. Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *OOPSLA*, pages 369–386, 2007.

[18] M. C. Rinard. Living in the comfort zone. In *OOPSLA*, pages 611–622, 2007.

[19] M. C. Rinard, C. Cadar, D. Dumitran, D. M. Roy, and T. Leu. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *ACSAC*, pages 82–90, 2004.

[20] M. C. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, pages 303–316, 2004.

[21] M. C. Rinard, C. Cadar, and H. H. Nguyen. Exploring the acceptability envelope. In *OOPSLA Companion*, pages 21–30, 2005.

[22] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. C. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *SIGSOFT FSE*, pages 124–134, 2011.

[23] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. C. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *POPL*, pages 441–454, 2012.