Report from Dagstuhl Seminar 11062

Self-Repairing Programs

Edited by

Mauro Pezzè[1], Martin C. Rinard[2], Westley Weimer[3], and Andreas Zeller[4]

1 **University of Lugano, CH,** `mauro.pezze@usi.ch`
2 **MIT – Cambridge, US,** `rinard@lcs.mit.edu`
3 **University of Virginia, US,** `weimer@cs.virginia.edu`
4 **Saarland University, DE,** `zeller@cs.uni-saarland.de`

Abstract

Dagstuhl seminar 11062 "Self-Repairing Programs" included 23 participants and organizers from research and industrial communities. Self-Repairing Programs are a new and emerging area, and many participants reported that they initially felt their first research home to be in another area, such as testing, program synthesis, debugging, self-healing systems, or security. Over the course of the seminar, the participants found common ground in discussions of concerns, challenges, and the state of the art.

Seminar 06.–11. February, 2011 – www.dagstuhl.de/11062
1998 ACM Subject Classification D.2.1 Requirements/Specifications, D.2.4 Software/Program Verification, D.2.5 Testing and Debugging, D.2.7 Distribution, Maintenance, and Enhancement, F.3.1 Specifying and Verifying and Reasoning about Programs, I.2.2 Automatic Programming
Keywords and phrases automated program repair, contract, debugging, fault, patch, self-healing
Digital Object Identifier 10.4230/DagRep.1.2.16

1 Executive Summary

Mauro Pezzè
Westley Weimer
Andreas Zeller

Dagstuhl seminar 11062 "Self-Repairing Programs" included 23 participants and organizers from research and industrial communities. Self-Repairing Programs are a new and emerging area, and many participants reported that they initially felt their first research home to be in another area, such as testing, program synthesis, debugging, self-healing systems, or security. Over the course of the seminar, the participants found common ground in discussions of concerns, challenges, and the state of the art.

Why Self-Repairing Programs?

Recent years have seen considerable advances in automated debugging. Today, we have techniques that automatically determine problem causes — in the source code, in program input, in the change history, or in internal data structures. While these approaches make it considerably easier to find the causes of defects, their precision is still insufficient to suggest a single concrete course of action — a human in the loop is still required to design and apply the patch. At the same time, there is an ongoing need for self-healing systems — systems that can recover from failures and even reconfigure themselves such that the failure no longer occurs. Most research efforts in this direction, though, assume planned recovery — that is, well-defined recovery strategies for anticipated failures.

An alternative is to explore self-repairing systems from a generic perspective — that is, to develop techniques that repair systems that are as generic, unassuming, and non-intrusive as program analysis and debugging. The idea is to determine actual fixes to state, to configuration, or to code. These fixes can be seen as guidance for the developer on how to fix the problem and evolve the software. However, fixes can also be deployed automatically, and effectively lead to programs that fix themselves. Such techniques may be particularly useful for orphaned systems that are no longer maintained or for critical software for which downtime is extremely expensive or even unacceptable. In these situations, there is no time to wait for a human developer to find and fix the bug. A synthesized patch can form a first line of defense against failures and attacks, a "first aid" approach to buy time while more expensive or manual methods are deployed. At the same time, automatically generated fixes provide a much richer diagnostic quality then simple fault localization, and thus may dramatically reduce the time it takes to debug a problem.

Goals of the Seminar

This main goals of this seminar was to provide knowledge exchange, mutual inspiration, and opportunities for collaborations for a rapidly developing field. The seminar aimed to bring together researchers in dynamic program analysis, automated debugging, specification mining, software survival techniques, and autonomic computing to increase awareness of these issues and techniques across relevant disciplines (program analysis, debugging and self-adaptive systems), and to discuss:

- how to monitor systems to detect abnormal state and behavior
- how to generate fixes and how to choose the best fixes
- how to deploy them in real-life systems and how to deal with the issues that arise when automatically correcting errors in software systems

Format and Presentations

The seminar started with summary presentations to bring all participants up to the speed on the state of the art and establish a common terminology. Subsequent activities alternated between technical presentations and plenary discussion sections. The seminar participants also split into two groups based on the self-identified focus areas of "The Architecture of Self-Repairing Systems" and "Validating Automated Repairs via Testing and Specification". Some evenings featured demonstrations or special-interest talks.

Common Concerns and Insights

As a whole, the group identified four challenge areas and opportunities for self-repairing programs: Architecture, Redundancy, Efficiency and Trust. In terms of Architecture, there was an acknowledgment that overall progress could be made by tackling particular problems or subdomains (e.g., fixing only atomicity violations or fixing only web applications, etc.) and potentially combining solutions later. There was a broad realization that redundancy is important on many levels: as a source of comparison for finding bugs or specifications; as a source of repair components; and as a main component of self-healing or self-adaptive systems at the architectural level. In terms of efficiency, the speed of the repair process — including the time required to validate a candidate repair — was of some concern, although many current techniques take minutes rather than hours to produce repairs. Trust was perhaps the most universally accepted issue: a notion that it is the responsibility of the repair process to provide an assurance argument, backed up by evidence, that would give a user or developer confidence that a repair can be applied safely. As general guidelines, we felt that an automatically-generated repair should not (or should at most minimally) regress the program by impairing functionality, and that applying such a repair should not be worse than doing nothing.

The group also identified two cross-cutting concerns related to correctness and evidence. The first was a notion that the evidence used to produce a repair (e.g., a few test cases or a partial specifications used for fault localization or repair construction) might be different from the evidence used to validate a final candidate repair (e.g., a larger test suite or a more complete specification). In addition, emphasis was placed on a clear characterization of common versus anomalous (or incorrect) behavior, possibly via a learned specification.

Challenge Areas Identified

The participants also identified a number of challenge areas or difficult tasks. By far the most popular was a notion of benchmarking. While the group acknowledged that the field is still quite new, and that formal benchmarks may not be appropriate, there was a desire for representative instances of programs with defects, tests that demonstrate those defects, normal regression tests, and indications of how humans fixed those defects.

The second challenge identified was the need for low-overhead, continuous monitoring to learn formal specifications for correct behavior, detect anomalies, and validate a system after repair deployment. The third challenge was to provide "just-in-time" repairs that were as quick as the auto-correction in Word or Eclipse. A fourth challenge related to documenting repairs or otherwise equipping them with evidence and arguments that would give confidence that they fix the system without causing additional harm.

A number of additional concerns were identified but were supported by a smaller segment of the participants. These included focusing on the economic value of repairs (e.g., targeting high severity defects or measuring the effort saved), the desire to repair programs even if a regression test suite is note available, the desire to have tools that succeed or fail with certainty (i.e., rather than producing incorrect repairs), some notion of automated repair techniques fixing 10% (or 50%, or 70%) of all reported bugs with some level of confidence, and the desire to improve automated fault localization techniques and allow them to report causes, not just locations.

2 Table of Contents

3 Overview of Talks

During the seminar, we alternated plenary and subgroup discussions and presentations. Here we summarize the main presentation.

3.1 Self-supervising BPEL processes

Luciano Baresi (Politecnico di Milano, IT)

Service compositions suffer changes in their partner services. Even if the composition does not change, its behavior may evolve over time and become incorrect. Such changes cannot be fully foreseen through pre-release validation, but impose a shift in the quality assessment activities. Provided functionality and quality of service must be continuously probed while the application executes, and the application itself must be able to take corrective actions to preserve its dependability and robustness. The talk proposes the idea of self-supervising BPEL processes, that is, special-purpose compositions that assess their behavior and react through user-defined rules. Supervision consists of monitoring and recovery. The former checks the system's execution to see whether everything is proceeding as planned, while the latter attempts to fix any anomalies. The talk introduces two languages for defining monitoring and recovery and explains how to use them to enrich BPEL processes with self-supervision capabilities. Supervision is treated as a cross-cutting concern that is only blended at runtime, allowing different stakeholders to adopt different strategies with no impact on the actual business logic. The talk also presents a supervision-aware run-time framework for executing the enriched processes.

3.2 Angelic Debugging

Satish Chandra (IBM TJ Watson Research Center – Hawthorne, US)

Software ships with known bugs because it is expensive to pinpoint and fix the bug exposed by a failing test. To reduce the cost of bug identification, we compute expressions that are likely causes of bugs and thus candidates for repair. Our symbolic method closely approximates an ideal approach to fixing bugs, which is to explore the space of all edits to the program, searching for one that repairs the failing test without breaking any passing test. We approximate this expensive ideal by computing not syntactic edits to an expression but instead the set of values whose substitution for the expression results in a correct execution. We observe that an expression is a repair candidate if it can be replaced with a value that fixes a failing test and, crucially, in each passing test, its value can be changed to another value without breaking the test. Such an expression is flexible because it permits multiple values; therefore, the repair of the expression is less likely to break a passing test. The method is called angelic debugging because the values are computed by angelically non-deterministic statements. We implemented the method on top of the Java PathFinder model checker. Our experiments show that angelic debugging can pinpoint the source of the bug in both synthetic and realistic programs.

Based on joint work with Emina Torlak (formerly, IBM Research), Shaon Barman (Berkeley) and Ras Bodik (Berkeley).

3.3 What should we repair and how

Brian Demsky (University of California – Irvine, US)

I presented a summarizing talk entitled "What should we repair and how?" that covered previous work on repair. My talk covered work on repairing data structures, program values, program environment, and source code. The talk then extracted common themes. One theme is the problem of selecting repairs — avoiding trivial and undesirable themes. The next theme is a tradeoff between providing strong guarantees and the expressiveness of the system. Another theme is where the repair actions come from. Finally the question of whether the human is in the loop.

3.4 Introductory Rabble Rousing Talk

Stephanie Forrest (University of New Mexico – Albuquerque, US)

This introductory talk attempted to lay out some of the big questions for the field of self-repairing programs, including: How we know what a program should be doing; "How we know that a program is behaving incorrectly;" and "How do we find the bug"? The talk briefly highlighted common approaches to these problems, emphasizing anomaly detection approaches.

3.5 Mutational robustness

Stephanie Forrest (University of New Mexico – Albuquerque, US)

One form of redundancy in software is those statements or instructions that have no discernible effect on the execution of the program. We have measured this effect at both the Abstract Syntax Tree (using CIL) and at the assembly code level using the following procedure:

1. Start with an unmutated working program
2. Generate a random mutation using the mutation mechanisms described in our ICSE 09 and ASE 10 papers.
3. Run the mutated program on the test cases for the program.
4. Call a mutation that does not change test case behavior "neutral."
5. Repeat Steps 1–4.

This experiment produces a rate of neutral mutations ranging from 20 to 60%. Subsequent experiments showed that these startling results are not due to expected sources (e.g., code coverage of test suites, insertion of dead code, etc.).

We believe that this may be an important source of free redundancy for automated program repair.

3.6 Automatic Workarounds for Web Applications

Alessandra Gorla (University of Lugano, CH)

Faults in Web APIs can escape the testing process, and consequently applications relying on these libraries may fail. Reporting an issue and waiting until developers fix faults in failing Web APIs is a time consuming activity, and in this time frame many users may be affected by the same issue.

In this talk I will present a technique that finds and executes workarounds for faulty Web applications automatically and at runtime. Automatic workarounds exploit the inherent redundancy of Web applications, whereby a functionality of the application can be obtained through different sequences of invocations of Web APIs. In general, runtime workarounds are applied in response to a failure, and require that the application remain in a consistent state before and after the execution of a workaround. Therefore, they are ideally suited for interactive Web applications, since those allow the user to act as a failure detector with minimal effort, and also either use read-only state or manage their state through a transactional data store. This work focuses on faults found in the access libraries of widely used Web applications such as Google Maps. It starts with a classification of a number of reported faults of the Google Maps and YouTube APIs that have known workarounds. From those we derive a number of general and API-specific program-rewriting rules, which we then apply to other faults for which no workaround is known. Our experiments show that workarounds can be readily deployed within Web applications, through a simple client-side plug-in, and that program-rewriting rules derived from elementary properties of a common library can be effective in finding valid and previously unknown workarounds.

3.7 Improving population-based automated patch generation

Dongsun Kim (The Hong Kong University of Science & Technology, HK)

Generating patches is one of key activities in software maintenance. Once a buggy code is located, developers try to add, remove, or change source code in order to fix the bug. This bug resolution work is tedious and time consuming. Recent few work attempted to automate bug resolution using population-based approaches. However, only few cases can be resolved by these techniques. Our goal is to improve the current state of the art. A novel technique includes fix patterns and similarity measures to enhance the current practice. Preliminary experimental results show our technique expands the space of automated patch generation.

3.8 Automated Atomicity-Violation Fixing

Ben Liblit (University of Wisconsin – Madison, US)

Fixing software bugs has always been an important and time-consuming process in software development. Fixing concurrency bugs has become especially critical in the multicore era. However, fixing concurrency bugs is challenging, in part due to non-deterministic failures and tricky parallel reasoning. Beyond correctly fixing the original problem in the software, a good patch should also avoid introducing new bugs, degrading performance unnecessarily, or damaging software readability. Existing tools cannot automate the whole fixing process and provide good-quality patches.

I will present AFix, a tool that automates the whole process of fixing one common type of concurrency bug: single-variable atomicity violations. AFix starts from the bug reports of existing bug-detection tools. It augments these with static analysis to construct a suitable patch for each bug report. It further tries to combine the patches of multiple bugs for better performance and code readability. Finally, AFix's run-time component provides testing customized for each patch. Experimental evaluation shows that patches automatically generated by AFix correctly eliminate six out of eight real-world bugs and significantly decrease the failure probability in the other two cases. AFix patches never introduce new bugs and have similar performance to manually-designed patches.

3.9 Dynamic Analysis for Diagnosing Integration Faults

Leonardo Mariani (Università di Milano–Bicocca, IT)

In this talk we present the BCT analysis technique. BCT uses dynamic analysis to automatically identify the causes of failures and locate the related faults. BCT augments dynamic analysis techniques with model-based monitoring. In this way, BCT identifies a structured set of interactions and data values that are likely related to failures (failure causes), and indicates the components and the operations that are likely responsible for failures (fault locations).

3.10 Automated Regression Testing of Modified Software

Alessandro Orso (Georgia Institute of Technology, US)

Throughout its lifetime, software is modified to enhance its functionality, repair it, eliminate faults, and adapt it to new platforms. One common way to ensure that changes made on the program behave as intended and did not introduce unintended side effects is to run the new version of the program against a set of test cases (i.e., a test suite). Unfortunately, the existing test suite for the program may be inadequate for this task. First, the test suite may contain too many test cases that do not test the modified parts of the program, and thus waste testing resources if run. Second the test suite may not contain test cases needed to adequately exercise the changes in the code. To address these issues, we present two approaches: the first approach analyzes the changes between two versions of a program and identifies the test cases in an existing test suite that do not need to be rerun; the second approach identifies behavioral differences between the two versions through test generation and differential dynamical analysis, and suitably presents them to the developers. In this talk, we present the two techniques, discuss their applicability in the context of self-repairing programs, and sketch possible future research directions.

3.11 Evidence-based automated program fixing

Yu Pei (ETH Zürich, CH)

Many programmers, when they encounter an error, would like to have the benefit of automatic fix suggestions—as long as they are, most of the time, adequate. Initial research in this direction has generally limited itself to specific areas, such as data structure classes with carefully designed interfaces, and relied on simple approaches.

To provide high-quality fix suggestions in a broad area of applicability, the present work relies on the presence of contracts in the code, and on the availability of dynamic analysis to gather evidence on the values taken by expressions derived from the program text.

The ideas have been built into the AutoFix-E2 automatic fix generator. Applications of AutoFix-E2 to general-purpose software, such as a library to manipulate documents, show that the approach provides an improvement over previous techniques, in particular purely model-based approaches.

3.12 Self healing lessons that may be learned from concurrency testing to self healing

Shmuel Ur (University of Bristol, GB)

I started by discussing how healing is done in the context of deadlock. One very interesting point about concurrent software is that bugs cause failure for specific inputs only some times. This means that for the same input there are interleaving that fail and some that succeed. The healing can be done by reducing the interleaving to those that do not have the bug without modifying the source code. Care must be taken not to reduce interleaving in such a way that a new interleaving is created. I showed how gate locks can be added to the code to protect from deadlock resulting from violation of lock discipline. It is interesting that in order to heal deadlocks we do not need one to occur first but the possibility of one is enough. I explained that care must be taken as the healing itself may cause new deadlocks and explained how to avoid such results.

The second topic I discussed was how to pinpoint the location of concurrent bugs. We use instrumentation to evaluate how timing change at each point are likely to expose the bug. This give each point in the program a bug finding score. We refine the score by looking at the delta between close points in the control flow. The idea is that if one point is good at finding bugs and the next one is not as good, than this is a good location to show the programmer.

I briefly described additional relevant concurrent research. I showed how concurrent coverage is used. I explained that the noise that is used to reveal bugs must be carefully applied as too much will actually not be good at finding bugs and we discussed performance healing by slowing down some requests.

3.13 How should repairs be validated and deployed?

Westley Weimer (University of Virginia, US)

The validation and deployment of repairs is of critical concern to self-repairing systems. This summary presentation discusses possible settings (e.g., human-in-the-loop, short-term fix, long-term repair) as well as issues of trust. One key goal is that automated repairs must provide an assurance argument, based on evidence, that they are safe to apply. Most research thus far has focused on safety properties, but liveness properties and notions like dependability and reliability are also important. In addition, it is possible that insights from formal verification can be used to aid automated repair. Some researchers have taken advantage of the special structure or rich semantics of languages to provide additional information to the repair process. Others obtain help from humans or automatically mine partial specifications. By far the most common approach, however, is to use test cases to validate repairs.

3.14 Automated Fixing of Programs with Contracts

Yi Wei (ETH Zürich, CH)

In program debugging, finding a failing run is only the first step; what about correcting the fault? Can we automate the second task as well as the first? The AutoFix-E tool automatically generates and validates fixes for software faults. The key insights behind AutoFix-E are to rely on contracts present in the software to ensure that the proposed fixes are semantically sound, and on state diagrams using an abstract notion of state based on the boolean queries of a class.

Out of 42 faults found by an automatic testing tool in two widely used Eiffel libraries, AutoFix-E proposes successful fixes for 16 faults. Submitting some of these faults to experts shows that several of the proposed fixes are identical or close to fixes proposed by humans.

3.15 First Step Towards Automatic Correction of Firewall Policy Faults

Tao Xie (North Carolina State University, US)

Firewalls are critical components of network security and have been widely deployed for protecting private networks. A firewall determines whether to accept or discard a packet that passes through it based on its policy. However, most real-life firewalls have been plagued with policy faults, which either allow malicious traffic or block legitimate traffic. Due to the complexity of firewall policies, manually locating the faults of a firewall policy and further correcting them are difficult. Automatically correcting the faults of a firewall policy is an important and challenging problem. In this paper, we make three major contributions. First, we propose the first comprehensive fault model for firewall policies including five types of faults. For each type of fault, we present an automatic correction technique. Second, we propose the first systematic approach that employs these five techniques to automatically correct all or part of the misclassified packets of a faulty firewall policy. Third, we conducted extensive experiments to evaluate the effectiveness of our approach. Experimental results show that our approach is effective to correct a faulty firewall policy with three of these types of faults.

3.16 Pex for Fun: Tool Support for Human to Repair Programs for Fun and Learning

Tao Xie (North Carolina State University, US)

Although there are various emerging serious games developed for education and training purposes, there exist few serious games for practitioners or students to improve their programming or problem-solving skills in the computer science domain. To provide an open platform for creating serious games in learning computer science, in 2010 summer, Microsoft Research released a web-based serious gaming environment called Pex for Fun, in short as Pex4Fun (http://www.pexforfun.com/), for learning critical computer science skills such as problem solving skills and abstraction skills.

Within Pex4Fun, coding duels are interactive puzzles to offer both fun and learning. In a coding duel, a player's task is to implement the Puzzle method to have exactly the same behavior as another secret Puzzle method (which is never shown to the player), based on feedback on what selected values the player's current version of the Puzzle method behaves differently and the same way, respectively. Pex4Fun uses Pex, a white-box test generation tool for .NET based on dynamic symbolic execution, to automatically generate such feedback.

3.17 Programming with Delegation

Jean Yang (MIT – Cambridge, US)

Access control and information flow are outside core functionality but critical to program correctness. Such issues are at odds with innovation because they are global concerns: securely adding a program feature requires reasoning about interaction with existing functionality. Supporting the separation core functionality and security concerns would facilitate rapid development. I describe programming with delegation, a programming model and execution strategy that allows a program to be run according to a security policy that the programmer provides but that the system automatically enforces. With delegation, the programmer can associate values with policies and the runtime system is responsible for ensuring these policies are satisfied. The programmer gives the system flexibility to do so by introducing nondeterminism; the programmer governs the nondeterminism using constraints. The system executes such programs using symbolic execution and constraint propagation. In this talk, I describe the Jeeves programming language for programming with delegation, implementation of the Jeeves interpreter, and performance results that suggest the feasibility of this approach.

3.18 Dynamic Generation of Processes

Rogerio de Lemos (University of Kent, GB)

In this talk we present the development of a framework for the dynamic generation of processes that factors out common process generation mechanisms and provides explicit customisation points to tailor process generation capabilities to different application domains. The framework encompasses a reference process for managing the dynamic generation of processes, a reusable infrastructure for generating processes and a methodology for its instantiation in different application domains. The framework explores model driven technology for simplifying the generation of processes in different domains, and includes fault-tolerance mechanisms for dealing with faults during generation and execution of processes.

3.19 How should repairs be validated and deployed?

Rogerio de Lemos (University of Kent, GB)

This talk has presented an overview of validation and deployment of self-repair of software from two perspectives: feedback control loop (MAPE or CADA) and fault tolerance. The objective is to scope the issues that could be covered in an overview kind of talk. Based on a brief introduction several other points were raised. What to validate? Whether the actual system or a model of the system should be validated. What kind of evidence can be obtained when the validation produces inconclusive results? How to combine development- and run-time evidence? The talk concluded with the presentation of some challenges.

Participants

- Luciano Baresi
Politecnico di Milano, IT
- Michael Carbin
MIT – Cambridge, US
- Antonio Carzaniga
Universitt Lugano, CH)
- Satish Chandra
IBM TJ Watson Research Center – Hawthorne, US
- Rogerio de Lemos
University of Kent, GB
- Brian Demsky
Univ. of California – Irvine, US
- Stephanie Forrest
University of New Mexico – Albuquerque, US
- Alessandra Gorla
University of Lugano, CH
- Patrick Hurley
AFRL/RIGA - New York, US
- Dongsun Kim
The Hong Kong University of Science & Technology, HK
- Ben Liblit
University of Wisconsin – Madison, US
- Leonardo Mariani
Universita Bicocca–Milano, IT
- Alessandro Orso
Georgia Institute of Tech., US
- Yu Pei
ETH Zürich, CH
- Mauro Pezzè
University of Lugano, CH
- Stelios Sidiroglou-Douskos
MIT – Cambridge, US
- Armando Solar-Lezama
MIT – Cambridge, US
- Shmuel Ur
University of Bristol, GB
- Yi Wei
ETH Zürich, CH
- Westley Weimer
University of Virginia, US
- Tao Xie
North Carolina State Univ., US
- Jean Yang
MIT – Cambridge, US
- Andreas Zeller
Universität des Saarlandes, DE