

Learning and Repair Techniques for Self-Healing Systems

Michael Ernst and Martin Rinard
MIT CSAIL

Self-Regenerative Systems PI Meeting
July 12, 2005

Outline

Overview

Current research progress: learning and repair

SRS program metrics

Applications

Technical objective and approach

Goal: Prevent user-visible data structure corruption errors

- crashes
- security breaches
- improper behavior

Approach:

- Automatically **learn** consistency properties
- Automatically **detect** violations
- Automatically **repair** corrupted data structures

Improvements to current practice

Existing practice (optimistic view!)

- **Manually** specify consistency properties
- Automatically detect violations
- **Manually** repair corrupted data structures

The manual steps are error-prone, tedious, and rare in practice

We reduce costs in human time, not computer time

- Automate previously impractical techniques

Expected achievements

Self-learning and self-healing systems

- observe their own behavior
- preserve that behavior in the future

Applications:

- prevent errors
- diagnose errors (early warning of root causes)
- ...

Key metric: increased system robustness

Objectives for remaining 6 months of SRS (and beyond!)

- scale to larger systems
 - new algorithms (e.g., dynamic type inference)
 - engineering (e.g., work on x86 binaries)
- distill lessons learned into reusable frameworks
- experimentation and red team evaluation

Outline

Overview

Current research progress: learning and repair

SRS program metrics

Applications

Repair

Given:

- data structure consistency constraint
- a data structure not satisfying the constraints

A **repair** modifies the data structure to satisfy a constraint

- example constraint: $x = \text{size}(s)$
- example repair: re-assign x
- example repair: add elements to s
- example repair: remove elements from s

Compute an ordering of repairs that guarantees termination

- naive ordering could loop forever

Result: system continues to execute successfully

- often preferable to catastrophic failure

Learning

Machine learning technique: **dynamic invariant detection**

Input is values (from a program execution)

Output is an **operational abstraction**

- consists of statistically justified properties
- syntactically similar to a formal specification

Examples:

- $x.\text{field} > \text{abs}(y)$
- $y = 2x + 3$
- a is sorted by $<$
- $\text{listnode.next.prev} = \text{listnode}$
- $\text{treenode.left.value} < \text{treenode.right.value}$
- $(p \neq \text{null}) \Rightarrow p.\text{content} \in \text{myArray}$
- $(\text{proc.priority} < 0) \Rightarrow p.\text{proc.status} = \text{active}$

Learning technique

Basic idea: generate-and-check

- postulate many possible invariants
- discard those that are falsified

Avoid checking invariants that are implied by existing ones

- for good run-time performance
- to avoid producing too much output

Can be formulated as an AI problem, but cannot be solved by previous AI techniques

- not classification or clustering
- no noise
- many positive examples, no negative examples
- intelligible output

Alternatives (human effort, static analysis) are tedious, error-prone, conservative, difficult on realistic programs

Incremental, online learning

Incremental: process each input value once, then discard

Online: run in conjunction with target program; no saved files

Key observation: many properties are redundant

- $(x = y) \wedge \text{odd}(x) \Rightarrow \text{odd}(y)$
- $(x = 5) \wedge (y = 6) \Rightarrow (x < y)$
- $(x < y) \Rightarrow (x \leq y)$
- $(x \geq y)$ at class `Stack` \Rightarrow $(x \geq y)$ at method `Stack.top()`

Redundant properties are not instantiated or checked

- easy, in a non-incremental (multiple passes) algorithm

Incremental approach: at any moment

- a property may be falsified
- anything it implied should never have been suppressed

Richer learned properties

New properties (e.g., domain-specific)

- tool architecture makes this easy

Properties closer to program semantics

- use source code as a guide
- pure methods become pseudo-fields (new variables)
- expressions and methods become properties to check

Implications

- for all properties p_1, p_2 , check $p_1 \Rightarrow p_2$
- squares the number (millions) of properties
- our approach: restrict the form of p_1
 - source code, machine learning, heuristics, ...

Temporal properties

- call sequences: both observed and inferred

Outliers

- detecting outliers
- avoiding outlier contagion

Dynamic type inference (value partitioning)

Problem:

```
int weight = 150;
```

```
int birthyear = 1967;
```

system output: "weight < birthyear"

Solution: determine which variables are **comparable**

Comparability is computed by dynamic type inference

- we attempted to apply static techniques, without success

Track program execution, record interactions between values

- extra tag for each value
- tags are merged when values are operated on
 - "a+b" means that a and b hold values of the same type
- translate value tags into variable comparability

Advantages

- reduces learning time
- reduces quantity of learner output (more important)

Previous approaches to instrumentation

Source instrumentation approach

- easy to produce output in terms of source code constructs
- as portable as the target program
- easy to insert instrumentation
- instrumentation is written at a high level
- instrumentation is manipulated by standard tools

Binary instrumentation approach

- target language (machine code) is smaller and simpler
- easier to capture low-level operations, determine layout, detect memory errors
- language-independent
- analyzes libraries
- no need for source code
- easier for users
- can map to source constructs at the end of execution

Mixed-level approach to instrumentation

Binary instrumentation

Map to source code continuously

- important for correctness and precision

Simpler and more robust than a one-level approach

Dynamic instrumentation toolkit: Fjalar

- used for Daikon front end (value tracing)
- used for comparability (dynamic type inference)

C++ instrumentation infrastructure

Uses mixed-level approach

Works on Linux and x86

- PowerPC and ARM ports underway

Incorporates comparability (dynamic type inference) algorithm

Use: `kvasir-dtrace myprogram --myoption myfile`

Scales to (core of) CTAS system (1,000,000 lines of code)

Cross-platform C++ instrumentation

Use binary instrumentation (Purify) for memory tracking

Use source code instrumentation for reporting

Implementation in process

Pro:

- works on many operating systems: Windows, Solaris, Linux, MacOS, ...
- works on many platforms: x86, PowerPC, ARM, ...

Con:

- requires source code
- requires commercial third-party library (Purify)
- no type inference (dynamic comparability)

Java results

Processed most of javac (5 of 7 packages)

- commercial Java compiler, 43,000 lines of code
- usually process a critical core, not an entire program

Annotate: duplicate info retained

- *this.attr.names = this.names*
- *this.attr.log = this.log*

Attr data structure properties


- *this = this.memberEnter.attr*
- *this = this.enter.attr*

Attr duplicate info retained

- *this.names = this.rs.names*
- *this.names = this.chk.names*

ConstFold: class should be singleton

- *this* has only one value

Variables that should be final in Annotate, ConstFold 

Outline

Overview

Current research progress: learning and repair

SRS program metrics

Applications

Program metrics

Goal

- diagnose 10% of root causes
- automatically correct 5%

Progress

- automatic diagnosis and correction of root causes
- no averages (yet) over a set of corruption errors
 - what is the population from which to sample failures?

Impediments to order-of-magnitude improvement

How many errors are data structure corruption errors?

- all errors, at some level

“Root cause” may be outside the program’s purview

- technique works on the program, not the domain

Performance

- restrict to a critical subsystem

Complex, interconnected data structures

- harder to learn about
- harder to repair

Specification language

- learning can produce too much output
- repair needs to be extended to additional properties

Outline

Overview

Current research progress: learning and repair

SRS program metrics

Applications

Data structure repair experiments

In each experiment:

- buggy version crashes
- version with repair continues (nearly correct) execution

Freeciv

- interactive strategy game, similar to Civilization
- 100,000 lines of code

AbiWord

- word processor, compatible with Microsoft Word
- 360,000 lines of code

CTAS

- fielded air traffic control system
- 1,000,000 lines of code

Applications of data structure constraints

Key application: data structure repair

Applicable to any problem that requires a specification

- sometimes **superior** to human-written specifications

Program steering: select best mode for current situation

Bug detection

- find latent code errors (and propose fixes)
- generate test inputs, test oracles, and stubs

Proofs

- upgrades: will a new component de-stabilize the system?
- proofs of learned constraints makes system sound

Detect outliers in data

Many others (45 distinct technical papers that I know of)

Daikon tool for constraint learning

Freely available (source and binaries) at

<http://pag.csail.mit.edu/daikon/>

- new release every month
- 160-page manual

Works on Java, C, C++, Perl, ...

Works on compiled binaries

- for C++, currently only works on Linux x86 binaries
- cross-platform C++ source front end is in progress

Produces output in 8 different formats

- repair tool, JML, Java expressions, theorem prover, ...
- easy to extend to new domains

Tool and technique is being used commercially

- available to other SRS projects

Use of data structure learning and repair

Learning

- execute system, operating correctly
- observe values computed during execution
- generalize via machine learning to obtain constraints

Instrumentation

- filter output, selecting desired constraints (manual step)
- insert constraints in program

Repair: if constraints are violated at run time

- compute a repair strategy
- modify data structures to satisfy constraints

Demo all pieces tonight