Automatic Recovery from Runtime Failures

presenter name(s) removed for FERPA considerations

Key Idea

- Use automatic recovery tools to generate alternative ways to achieve the same functionality of code that can potentially fail
- Identify potential areas for failure from library calls
- Create semantically equivalent functions using different library calls
- Employ tool to avoid failures at run-time

Study Questions

- Is automatic recovery effective in making applications more resilient to faults?
- Are the techniques efficient enough to be practically usable?
- Is modular software to some significant extent intrinsically redundant?

Why?

- Their argument: Modern software is intrinsically redundant
- Redundancy is an intrinsic property of modular software
- Implementations tend to be semantically equivalent
- Use of libraries
  - Backwards compatible functionality equivalence
  - Cross-library semantic equivalence
  - Functional equivalence for different use cases (based on input)
- The point is to find operations equivalent in intended behavior, but not in actual observable behavior

Equivalence of Tested Libraries

TABLE I
EQUIVALENT SEQUENCES FOUND IN REPRESENTATIVE JAVA LIBRARIES

Library	Guava	SWT	JodaTime
Classes considered	116	252	12
Total equivalences found	1715	1494	135
Average per class	14.78	5.93	11.25

How it Works - ARMOR

- Preprocessor identifies Roll-Back Areas (RBAs)
  - Static Analysis
  - Calls to the library that could be re-written
  - May result in Failure but minimal
  - Supports nested RBAs
  - Creates checkpoints before RBAs
  - Method bodies / singular field initialization expression (encapsulated as a method)
- Rewrites RBAs and compiles them
- Each new solution wrapped in a loop based on passing checkpoints
- On failure loop iterates to next available solution for implementation
  - Based on past success
- Rolls back to last checkpoint if notified of failure
- Runs until failure-free or runs out of solutions

Example: JodaTime

```
1 // failing operation
2 DateTime beginDay = dt.millisOfDay().withMinimumValue();
3 // workaround 1
4 DateTime beginDay = dt.toDateMidnight().toDateTime();
5 // workaround 2
6 DateTime beginDay = dt.withTimeAtStartOfDay();
```

Listing 2. Workarounds for issue n. 3304757 of JodaTime

```
1  class CurrentMidnight {
2    DateTimeZone tz = DateTimeZone.forID("America/Sao_Paulo");
3    DateTime midnight;

5    public void initDayAndZone(){
6      DateTimeZone.setDefault(tz);
7      DateTime dt = new DateTime();
8       ...
9      setMidnight(dt);
10   }

12   private void setMidnight(DateTime dt){
13     midnight = dt.millisOfDay().withMinimumValue();
14   }

16   public DateTime getMidnight(){
17     return midnight;
18   }
19 }

21 class Main {
22   public static void main(String args[]){
23     ...
24     CurrentMidnight cm = new CurrentMidnight();
25     cm.initDayAndZone();
26     ...
27     cm.getMidnight();
28     ...
29   }
30 }
```

Listing 3. Example application code

```
class CurrentMidnight {
  DateTimeZone tz = tz_init();
  public DateTimeZone tz_init_original() {
    return DateTimeZone.forID("America/Sao_Paulo");
  }
  public DateTimeZone tz_init() {
    try {
      create_checkpoint();
      return tz_init_original ();
    } catch (Exception ex) {
      while (more_rba_variants_available) {
        try {
          restore_checkpoint();
          load_new_rba_variant();
          return tz_init_original ();
        } catch (Exception ex1) {
          // record variant failure and adjust priorities
          ...
        }
      }
      throw ex;
    } finally {
      discard_checkpoint();
    }
  }
  DateTime midnight;
  ...
  // initDayAndZone proxy method not shown
  ...
  public void setMidnight_original(DateTime dt) {
    midnight = dt.millisOfDay().withMinimumValue();
  }
  public void setMidnight(DateTime dt) {
    try {
      create_checkpoint();
      setMidnight_original(dt);
    } catch (Exception ex) {
      boolean success = false;
      while (!success && more_rba_variants_available) {
        try {
          restore_checkpoint();
          load_new_rba_variant();
          setMidnight_original(dt);
          success = true;
        } catch (Exception ex1) {
          // record variant failure and adjust priorities
          ...
        }
      }
      if (!success) throw ex;
    } finally {
      discard_checkpoint();
```

The Experiment

- Libraries used
  - JodaTime: Library of utility functions to represent and manipulate dates and time.
  - Guava: The Google “core” library for collections, I/O, caching, concurrency, string processing, etc.
- Application Software
  - Fb2pdf: A command-line utility to convert files from the FB2 e-book format into PDF. Fb2pdf uses the Java date/time library but they changed it to use the fully compatible JodaTime library.
  - Carrot2:A search results clustering engine. Carrot2 uses the Guava library.
  - Caliper:A framework for writing, running and viewing the results of Java microbenchmarks. Caliper uses the Guava library.
  - Closure:A source-to-source optimizing JavaScript compiler. Closure uses the Guava library

Experiment cont.

- Formulated code re-writing rules and ran ARMOR preprocessor

TABLE II
RESULTS OF THE PREPROCESSING ON THE SELECTED APPLICATIONS

Application	*Caliper*	*Carrot2*	*Closure*	*Fb2pdf*
Total RBAs	130	139	2099	17
RBAs with variants	60	106	687	17

Experiment cont.

- Conducted a more extensive evaluation using seeded faults with both libraries and all four applications
- Ran mutation generation on all programs
- Used Daikon to get invariants
    - Used invariants that worked in the original code but failed in RBAs in mutated code
    - Added these as assertions in RBA

Results

TABLE III
MUTATION ANALYSIS AND EFFECTIVENESS OF ARMOR

				Caliper	*Carrot2*	*Closure*	*Fb2pdf*
Total mutants				21297	21297	21297	16858
Relevant mutants				309	187	344	2200
execution	*success*	*equivalent*		210	120	177	1805
		non-equivalent	*detected*	**0**	**2**	**0**	**0**
			not detected	0	8	3	1
	loop	*detected*		**0**	**1**	**0**	**0**
		not detected		12	9	15	47
	error			**87**	**47**	**149**	**347**
Total mutants run with ARMOR				**87**	**50**	**149**	**347**
Mutants where ARMOR is successful				(28%) 24	(48%) 24	(47%) 70	(19%) 67

TABLE IV
OVERHEAD INCURRED BY ARMOR IN NORMAL NON-FAILING EXECUTIONS (MEDIAN OVER 10 RUNS)

		Caliper	*Carrot2*	*Closure*	*Fb2pdf*
Time (seconds)	Original total running time	30.13	2.43	5.40	2.26
	Exception-handling only (no checkpoints)	(1%) 30.41	(69%) 4.15	(95%) 10.53	(68%) 3.79
	Snapshot-based checkpoints	(5%) 31.78	(117%) 5.32	>1h	(121%) 4.99
	Change-log-based checkpoints	(2%) 30.87	(94%) 4.75	(194%) 15.90	(114%) 4.70
Memory (MB)	Original total memory allocated	1.40	8.87	30.56	17.90
	Snapshot-based checkpoints	12.30	23.78	—	90.94
	Change-log-based checkpoints	10.18	11.37	120.58	25.93
Number of recorded checkpoints (approx.)		30	2,350	1,255,000	4
Values saved in change-log-based checkpoints (approx.)		26,000	270,000	1,880,000	9,000

Results cont.

- Fb2pdf had the smallest amount of RBAs, largest amount of mutants affecting execution, and lowest success rate
  - Applications with calls to the library that make up a larger portion of the library code
- Carrot2 had failed for one fault
  - Library call within a library code
  - High overhead with regard to checkpoint stack
- Overall, author asserts that the program results are positive
- In the future, to solve more issues deep in library code / test on different libraries

What’s New?

- Others have exploited redundancy
  - Requires additional code
  - Cabral - requires written exception handlers
  - Chang - manually write patches
  - Harmanci - code alternative code blocks
- Is different, as it exploits redundancy already available in libraries
  - Less design cost
- General purpose
- Designed to work on deployed applications
  - Prevent mistakes at run-time
- The notion of intrinsic redundancy in modern software applications

Discussion

1. Could this be potentially dangerous if implemented for a live system built on several nested library dependency?
2. Is the overhead worth it?
3. As library redundancy increases, will this process become more and more expensive?
4. Worth it for smaller / newer libraries / lower-level code?
5. Can we use this process to replace non-library code with semantically equivalent syntax?