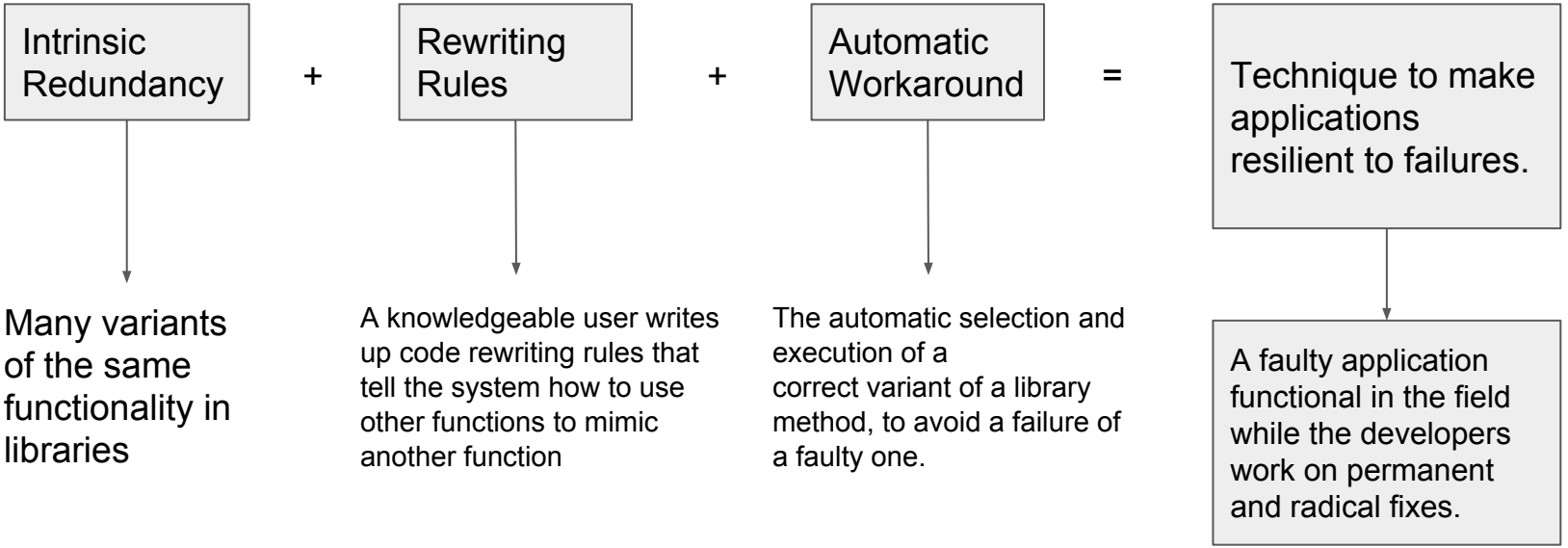
Automatic Recovery from Runtime Failures

Authors:
Antonio Carzaniga, Alessandra Gorlay, Andrea Mattavelli, Nicol`o Perino, Mauro Pezz`e

slide author names omitted for FERPA compliance

The Approach



Assumptions:

1. The library comes with a specification of the equivalence between the operations it supports
2. Failures are somehow detected and reported

Research Questions

1. Is modern software intrinsically redundant and if so to what extent?
2. By executing variants of methods, can we reduce the amount of runtime failures in a Java application that uses a library?
3. Does using automatic workaround increase the amount of overhead and prolong the time it takes for an operation to run?

Contribution

1. A new generic technique that uses the redundancy of libraries to attempt to automatically recover from runtime exceptions
2. ARMOR, a system that allows for automatic recovery in java applications by taking advantage of the natural redundancy of libraries
3. Code-rewriting rules for the JodaTime and Guava libraries. These could be used with their system to provide automatic recovery when using those libraries in your application

Key Idea

If an automatic workaround tool can find variations of procedures from 3rd party libraries and implement them in runtime to reduce total application failures, then developers will be able to deploy their working applications with small bugs and will be able to identify and fix the bugs as the application is being used by their users.

For example, Joe Schmo deploys shopping web app. Automatic workaround tool finds error in changeItem(), changeItem() is replaced in runtime with deleteItem() and addItem(), then reports error to Joe Schmo. Joe Schmo has happy customers while he fixes the bug.

Preprocessing

1. Finding Roll Back Areas (RBAs) to reset the application to and try other similar functions, if the previous implementation fails.
2. Injects code that creates RBAs, catches exceptions, and calls other functions if original one fails.
3. Compiles modified source code and variants

Only works with unchecked exceptions. Checked exceptions are still thrown.

Runtime

- The only changes at runtime will be the creation of the checkpoints, proxying of the methods, and rolling back to checkpoints and running other variants
- RBAs can be either snapshots or a lazy change log
- Runtime overhead varied from 2% to 194% increase
    - This overhead is mainly due to all of the try/catch blocks and proxy methods

Example

Developer 1

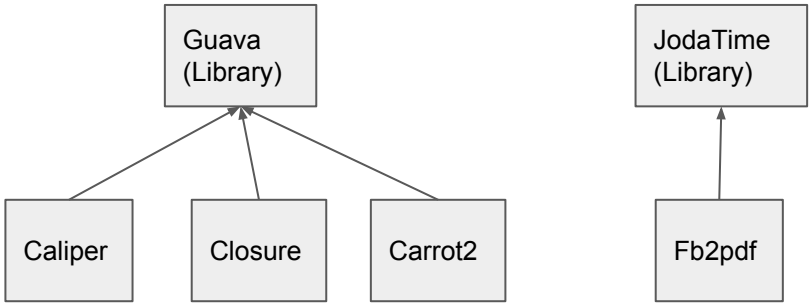
- Uses Apache HTTP Client Library
- Deploys production ready application with small bugs and automatic workaround tool.
- Tool finds methods with errors, and replaces methods with library variants.
- Fixes one bug at a time while application is still in production and useable.
- Takes note of rewriting rules and uses them for future applications using Apache HTTP Client Library.

Developer 2

- Does not use library.
- Does not deploy code because they want to fix bugs.
- Have to write their own custom tests for their custom methods.
- Fixes bugs all at the same time before deploy.
- Has to create new rewriting rules for each future application.

Summary of the Evaluation

- The objective of this evaluation is to determine whether the technique is effective in making applications more resilient to faults, and efficient enough to be practically usable.
- ARMOR is successful with between 19% and 48% of the mutants. These are cases in which ARMOR is completely successful, meaning that the application terminates successfully and with the correct output despite the presence of a failure-inducing fault.



Mutation Analysis

Use Major framework to inject faults into Guava and JodaTime libraries → Execute all applications with mutants and ARMOR

Initial Analysis

Identify equivalences between sequences of calls in libraries. → Write “code-rewriting rules” based off equivalences → Run ARMOR to find RBAs and to produce variants

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

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

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

Discussion Question 1

- We worry that this technique will only be efficient in very few cases as evidenced by 194% runtime overhead in Closure. How can we reduce the overhead to make this technique more efficient?

Discussion Question 2

- This technique currently only works for Java applications. Are there any changes that need to be made to this new technique to make it applicable to other languages or is it possible that it would work fine for other languages as it is?

Discussion Question 3

- The technique right now is only useful for code on the surface and fails as the methods are more embedded in the code under layers of inheritance. How can this new technique be modified so that it can reach as far in the code as possible?

Discussion Question 4

- A big part of this new technique is its dependence on intrinsic redundancy. How could developers creating redundant methods help the quality of code?

Discussion Question 5

- An important goal of software development is to ship a working product. How could this technique help ensure a quality product, that is quickly released to customers?

Thanks for listening!