Refactoring for Changeability: A way to go?

B. Geppert, A. Mockus, and F. Roessler

{bgeppert, audris, roessler}@avaya.com

Avaya Labs Research

Basking Ridge, NJ 07920

http://www.research.avayalabs.com/user/audris

Objective

- Legacy software is difficult and fault prone to change
  - Is it possible to do re-engineering on “live” system given the need to support several deployed releases (streams of fixes) and in parallel with new feature introduction (streams of features)?
  - What value such re-engineering may bring?
  - Will it survive through future changes?
- Intuitive conjecture: re-engineering will increase changeability — ability to make changes to software with minimal effort and without introducing many defects

Context

- One domain of Avaya’s IP telephony software
- 30 KLOC C++, ASN.1 generated code, 3rd party protocol stack within 7 MLOC system
- 40 different developers over 5 years
- Design degradation
- Constant change
  - inflow of defects from 5+ deployed releases
  - changes to implement new functionality for 2+ future releases

Outline for the remaining talk

- Refactoring and re-design
- Hypotheses
- Methodology
- Results
- Validation
- Conclusions

Software Refactoring

- For migrating legacy code to a target design
- Improve code structure without changing external behavior
- Sequence of simple behavior preserving code transformation steps
- For instance: “Extract Method”: Turn a code fragment into a method whose name explains the purpose of the method.

```
void printItinary() {
     printBanner();

     //print outbound flight details
     System.out.println (getAirline());
     System.out.println (“Flight: " +
     getFlightNumber());
     …
     }
```

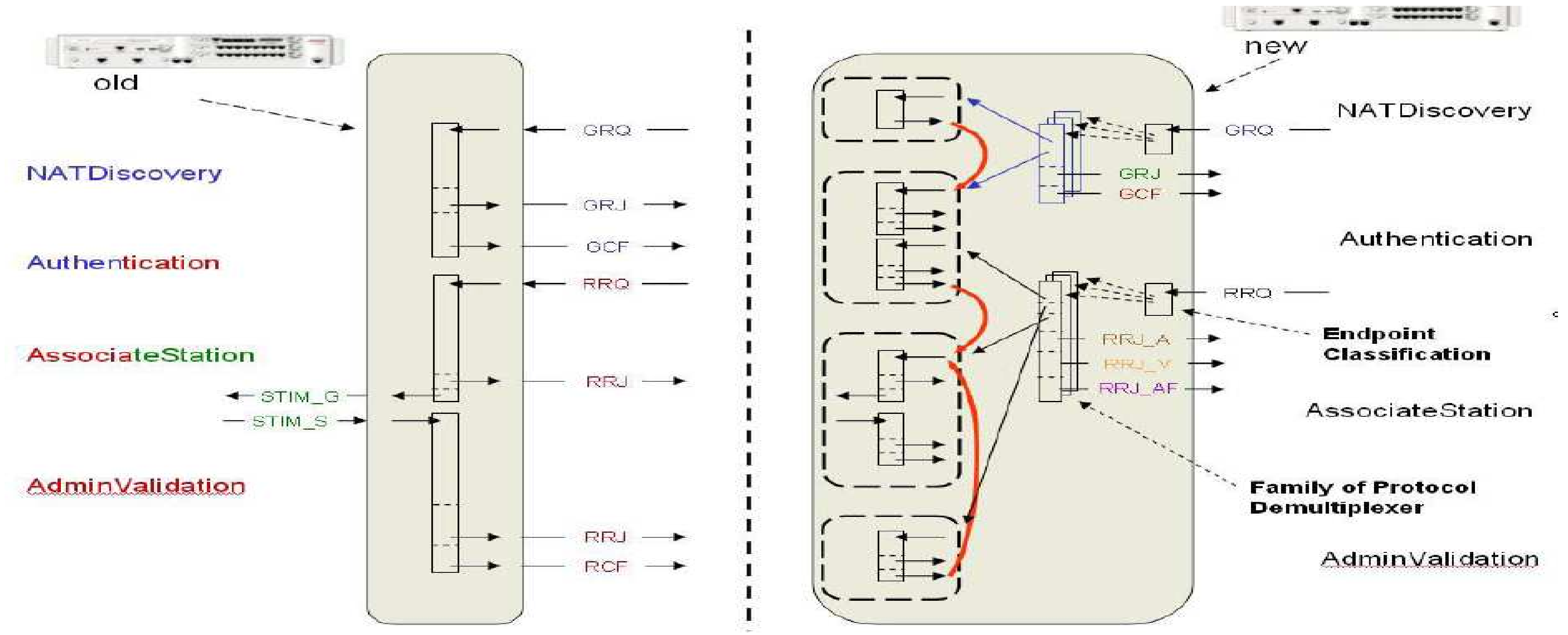
```
void printItinary() {
     printBanner();
     printOutFlightDetails(getAirline(),
                      getFlightNumber());
}
void printOutFlightDetails (char airline,

        int flightnumber){
     System.out.println (airline);
     System.out.println (“Flight: " +
        flightnumber);
}
```

Refactoring team

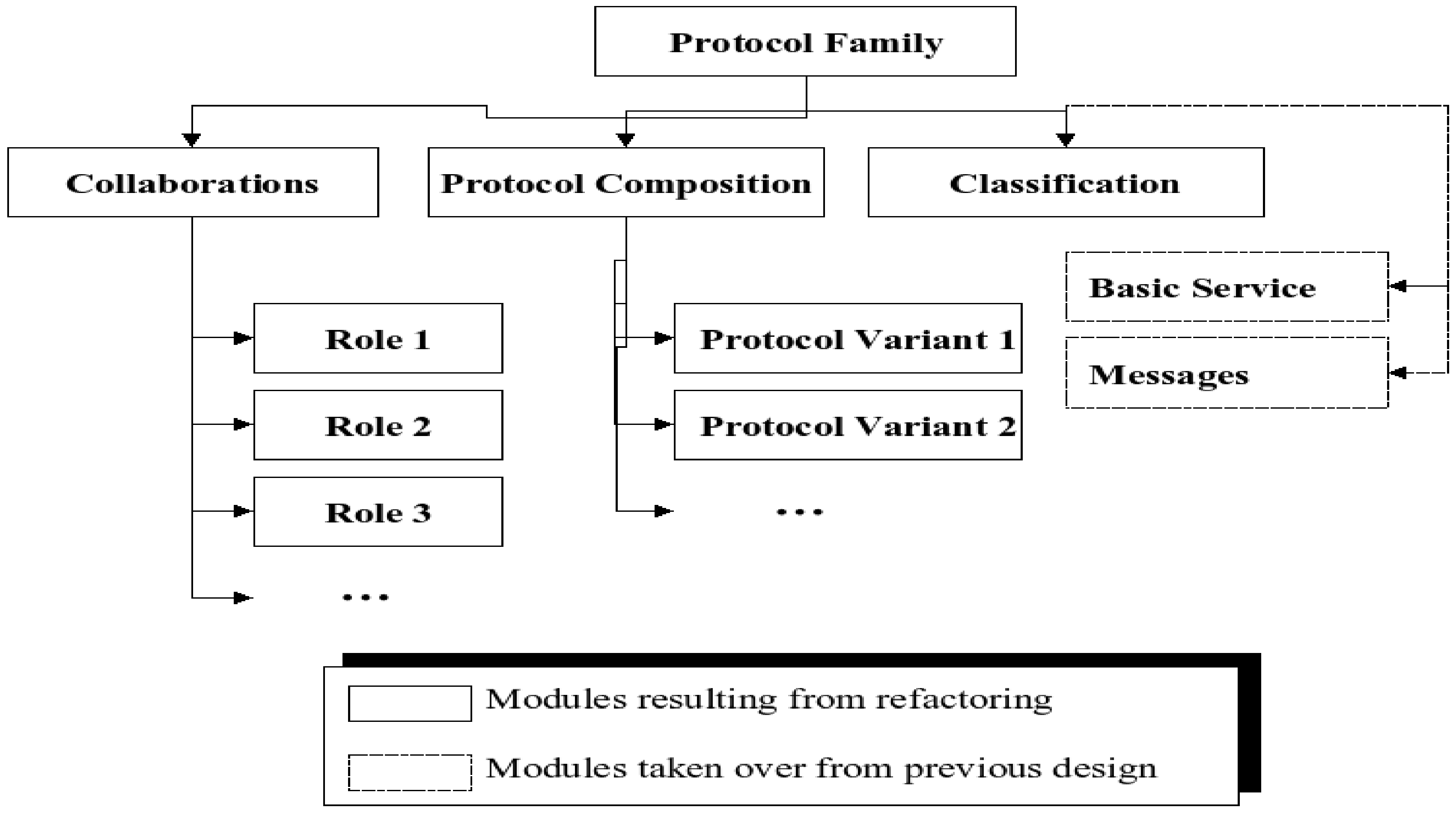
- 2 developers without experience with the legacy code, but experts in protocol-composition-design and software refactoring
  - Analysis, design, and refactoring
- 3 subject matter experts knowledgeable in target subsystem, development environment, and test environment
  - Consulting, design reviews, and code reviews

Design



	GRQ	GRJ	GCF	RRQ	RRJ	RCF	STIM_G	STIM_S
NATDiscovery	X	X	X					
Authentication	X	X	X	X	X	X		
AssociateStation				X	X	X	X	X
AdminValidation				X	X	X		

Modules


Protocol Family
Collaborations
Protocol Composition
Classification
Basic Service
Messages
Role 1
Role 2
Role 3
...
Protocol Variant 1
Protocol Variant 2
...
Modules resulting from refactoring
Modules taken over from previous design

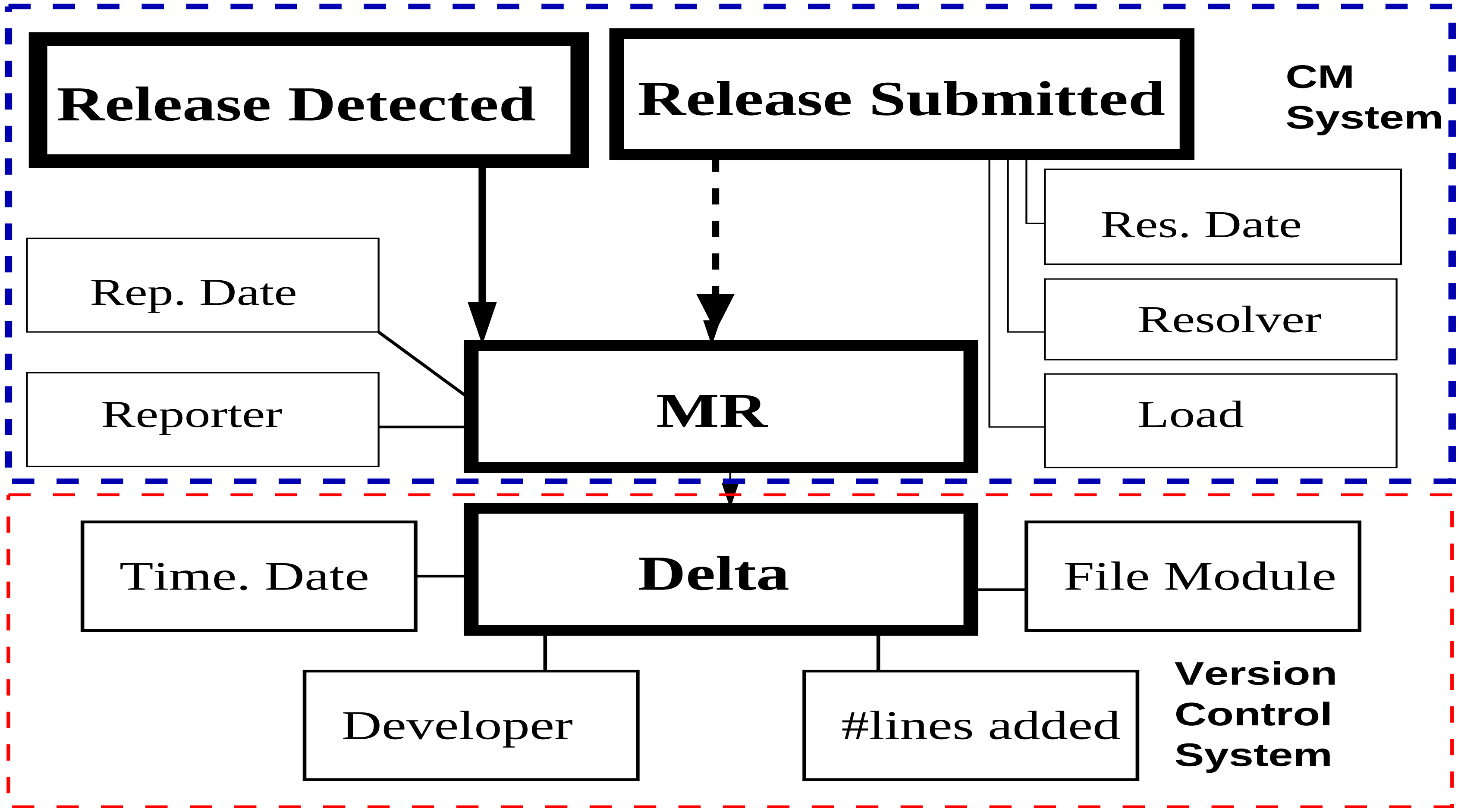
Refactoring Hypotheses

- H1: The customer reported defect rate will improve
  - collaboration-based design
  - refactoring exposed pre-existing issues
- H2: The refactoring reduces the effort required to make changes
  - information hiding
- H3: The refactoring reduces the scope of changes within the restructured domain
  - if design is good changes will be within modules
  - changes within a module are likely to touch few files

Measurement Methodology

- Software is created by making changes to it
  - A delta is a single checkin (ci/commit/edput) representing an atomic modification of a single file with following attributes
    - File, Date, Developer, Comment
  - Other attributes that often can be derived:
    - Size (number of lines added,deleted)
    - Lead time (interval from start to completion)
    - Purpose (Fix/New)
- Approach
  - Use project’s repositories of change data to model (explain and predict) phenomena in software projects and to create tools that improve software productivity/quality/lead times

Change Data


Release Detected
Release Submitted
CM System
Res. Date
Rep. Date
Resolver
Reporter
MR
Load
Time. Date
Delta
File Module
Developer
#lines added
Version Control System

Measures

- H1: the number of field MRs found and the root cause of these problems
- H2: change effort and the amount of code that needs to be inspected to make the change
- H3: the number of files touched in a change, the number of lines added, and the number of lines in the files that are modified

H1: Defect Density

- The number of defects depends on release size [1]
- Reported defects and submitted changes in registration domain
- Four pre- and one post-refactoring release

	Release Size	Field defects
pre-Refactoring	526	41
post-Refactoring	80	0

- Adjust for the shorter exposure of the last release
- assume only 50% of defects in the first 7 months (20)
- Fisher's exact test p-value 0.06

H1: Defect Density

- Large differences needed to get significance for rare events
- Alpha and beta trials
  - All problems were in preexisting functionality — i.e., refactoring faithfully reproduced them

H2: Change Effort

Stage	Number of changes	average($\log(\text{PersonMonths})$)
Pre-Refactoring	292	-1.12
Post-Refactoring	151	-1.23

- two-sample t-test of $\log(\text{effort})$ p-value .06
- Mann-Whitney of $\log(\text{effort})$ p-value .06
- The LOC in the refactored area decreased by 50%

H3: Scope Reduction

Measure	In Registration	Refactored
Files	UP	
Delta	UP	
Lines Added		Down
Lines Modified		Down

- a single file to several files after refactoring
- feature changes have larger scope than fixes
- refactoring reduced size 50%
- the trend in change scope depends operationalization
- when functionality should be kept in a single versus multiple files, what is the optimal file size?

Validation

- Reality
  - Verified the process
  - Verified selection of relevant changes (MRs)
  - Manually inspected all field MRs
  - Several operationalizations
- Modeling
  - Distribution: take logs or use nonparametric
  - Normalize by size where needed
  - Apply relevant models
- A case study — no causal inference

Summary

- Changeability as top objective
- Practical impact of the study
  - Organizational support
  - Two other domains undergoing refactoring
  - A course on refactoring taken by 20 developers
- Other insights
  - Complex practical constraints on re-engineering
  - Difficult to detect impact even when techniques appear to work
  - Effort impact of around 11%, defect - significant

References

•

[1] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(2), 2000.

Bio

Audris Mockus
Avaya Labs Research
233 Mt. Airy Road
Basking Ridge, NJ 07920
ph: +1 908 696 5608, fax:+1 908 696 5402
http://mockus.org, mailto:audris@mockus.org,
picture:http://mockus.org/images/small.gif

Audris Mockus conducts research of complex dynamic systems. He designs data mining methods to summarize and augment the system evolution data, interactive visualization techniques to inspect, present, and control the systems, and statistical models and optimization techniques to understand the systems. Audris Mockus received B.S. and M.S. in Applied Mathematics from Moscow Institute of Physics and Technology in 1988. In 1991 he received M.S. and in 1994 he received Ph.D. in Statistics from Carnegie Mellon University. He works at Software Technology Research Department of Avaya Labs. Previously he worked at Software Production Research Department of Bell Labs.