Corpus Studies

Corpus Studies

- How often does X occur in the real world? X could be:

  - A bug

    - If it occurs often, it's worth preventing or detecting

  - A use of a feature or tool

- Is my tool applicable to real-world code?

- What is the world like?

  - e.g., what kinds of questions do people ask about Rust?

Corpus Study Techniques

- Get a corpus

  - Replicable studies require a fixed corpus

    - Snapshot of GitHub?

  - Why *this* corpus? Consider external validity.

- Write a tool or analyze manually.

- Sample?

True and False Positives and Negatives

- Context: finding needles in a haystack. (or: finding bugs in programs. or: proving that a program *lacks* a certain bug)

- Procedure: investigate each item in the haystack. Check if each item is a needle.

- True positive: This needle is a needle. 😀

- False positive: This pine cone is a needle. 🙁   *Complete* analyses never make this mistake

- True negative: This pine cone is *not* a needle. 😀

- False negative: This needle is *not* a needle. 🙁   *Sound* analyses never make this mistake

Research Questions

- RQ1: how many bugs of type X are there in this corpus? Possible answers:

    - At least n.

    - Exactly n.

    - No more than n.

- RQ2: Can my tool find bugs?

    - Yes, lots!

    - Yes, but only a small fraction of the ones that are present.

    - No, but that's because there weren't any bugs of that type.

    - No, but it's unsound, and I have no idea how many bugs there are of that type.
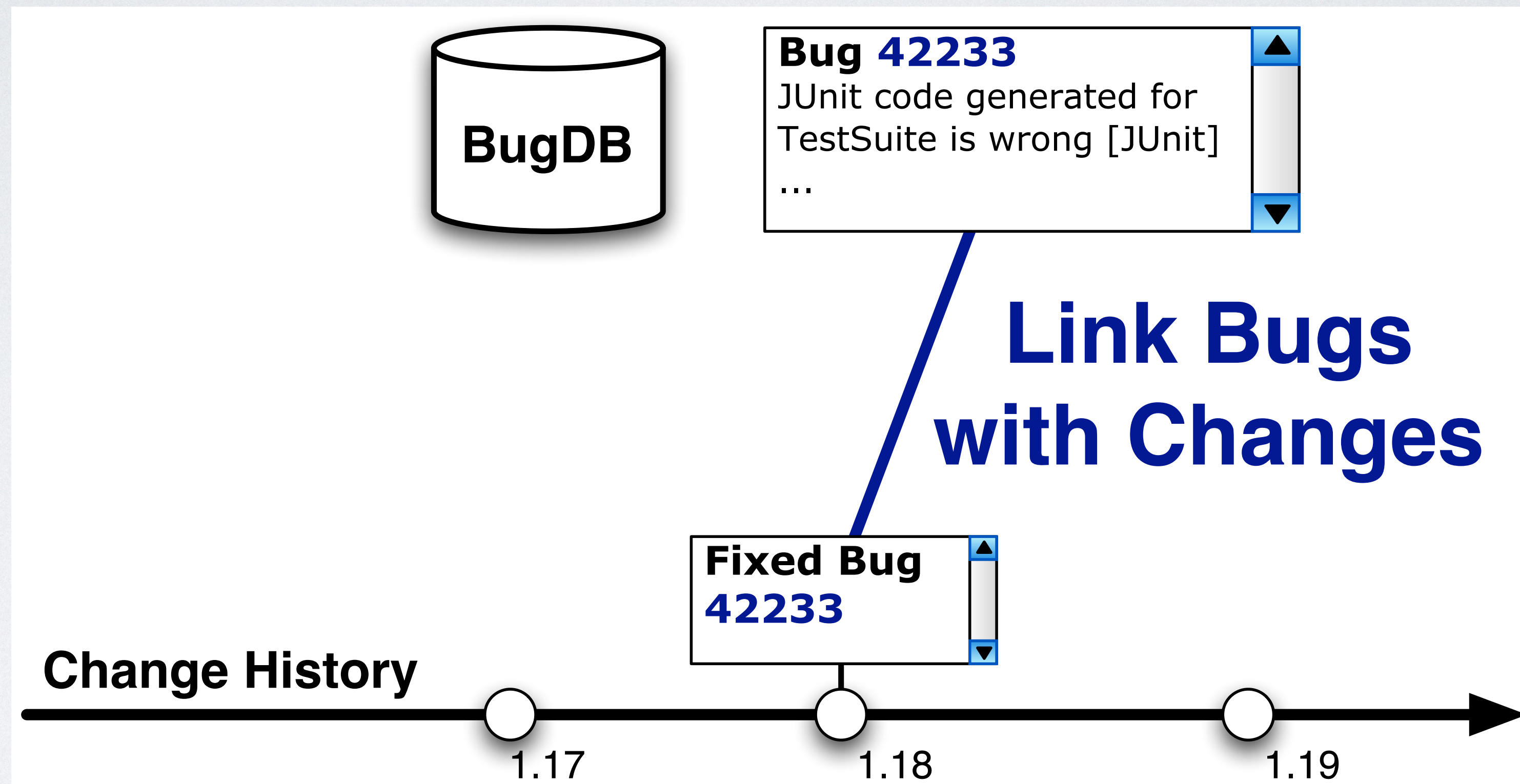
Analysis Techniques

- Sound analysis: find everything in category X

  - Might also find things NOT in category X

  - Mitigate with manual analysis

- Complete analysis: only find things in category X

  - But we have no idea how many we missed

- Sound and complete: nice if you can get it…

  - Rice's theorem: "all non-trivial, semantic properties of programs are undecidable."

What Corpus Studies Should We Do?

- (Discuss!)

Commits Typically Specify Which Bugs They Fix

BugDB

Bug 42233
JUnit code generated for TestSuite is wrong [JUnit] …

Link Bugs with Changes

Fixed Bug 42233

Change History

1.17 1.18 1.19

Authors mined commit logs to find bug numbers

Corpus Study 1: When Do Changes Induce Fixes?

- Jacek Śliwerski, Thomas Zimmermann, Andreas Zeller

- RQ: When do developers insert bugs?

  - Imagine if your IDE could tell you: "your change is likely buggy!"

- Problem: how do you tell when a change inserted a bug?

- Approach: if a change's code needs to be changed AGAIN, the change induced a fix.

Fix-Inducing Changes

- "A fix-inducing change is a change that later gets undone by a fix."

- Suppose change δ changes line 42 to fix bug B.

- Aha, revisions 675d7f and 56879a changed line 42.

- But 675d7f was committed AFTER bug B was reported, so that wasn't the cause.

- 56879a is suspect.

Fix-Inducing Commits Are Large

	fix-inducing	¬fix-inducing	all
fix	3.82±26.32	2.08± 7.42	2.73± 7.87
¬ fix	11.30±63.02	2.77±14.94	3.81±26.32
all	7.49±44.37	2.61±13.66	3.52±22.81

Table 3: Average sizes of fix and fix-inducing transactions for ECLIPSE

Don't Commit on Fridays

% of revisions	Day of Week							
	Mon	Tue	Wed	Thu	Fri	Sat	Sun	avg
$P(\textit{fix})$	18.4	20.9	20.0	22.3	24.0	14.7	16.9	20.8
$P(\textit{bug})$	11.3	10.4	11.1	12.1	12.2	11.7	11.6	11.4
$P(\textit{bug} \cap \textit{fix})$	4.6	4.8	4.6	5.2	5.6	4.5	4.5	4.9
$P(\neg\textit{bug} \cap \neg\textit{fix})$	74.9	73.5	73.5	70.8	63.4	78.1	76.0	72.7
$P(\textit{bug} \,|\, \textit{fix})$	25.1	22.9	23.3	23.5	23.2	30.3	26.4	23.7
$P(\textit{bug} \,|\, \neg\textit{fix})$	8.2	7.1	8.1	8.8	8.7	8.4	8.6	8.1

Table 5: Distribution of fixes and fix-inducing changes across day of week in ECLIPSE
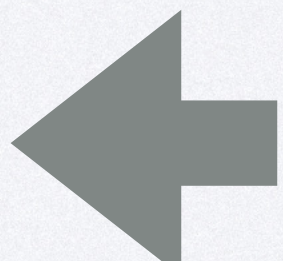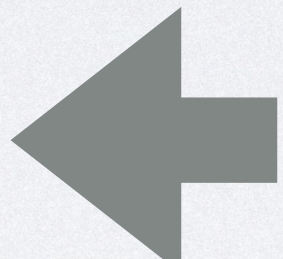
Explanation?

- Maybe developers commit without as much testing or review on Fridays because they don't want to leave tasks pending over the weekend.

Another Example: Object Protocols

- Example: a File is either `Open` or `Closed`.

- On `Open` file: `close()` ok. `open()` illegal.

- On `Closed` file: `open()` ok. `close()` illegal.

- "Definition: A type defines an object protocol if the concrete state of objects of that type can be abstracted into a finite number of abstract states of which clients must be aware in order to use that type correctly, and among which object instances will dynamically transition."

Finding Protocols

```
1   // from java.util.concurrent.ArrayBlockingQueue.Itr
2   public void remove() {
3     final ReentrantLock lock = ArrayBlockingQueue.this.lock;
4     lock.lock();
5     try {
6       int i = this.lastRet;
7       if (i == -1)
8         throw new IllegalStateException();
9       lastRet = -1;
10      // ... method continues
11  }
12
13  // from javax.swing.undo.AbstractUndoableEdit
14  public void undo() throws CannotUndoException {
15    if (!canUndo()) {
16      throw new CannotUndoException();
17    }
18    hasBeenDone = false;
19  }
20
21  public boolean canUndo() { return alive && hasBeenDone; }
```

Strategy: find exceptions that are thrown conditionally, depending on field data.

Sound? Complete? What is an object protocol, anyway?

Method

- Phase 1: find protocols

  - Static analysis to find candidate protocols (unsound, incomplete)

  - Manual investigation of reports

- Phase 2: find how often protocols are used

  - Find calls to protocol methods

An Empirical Study of Object Protocols in the Wild
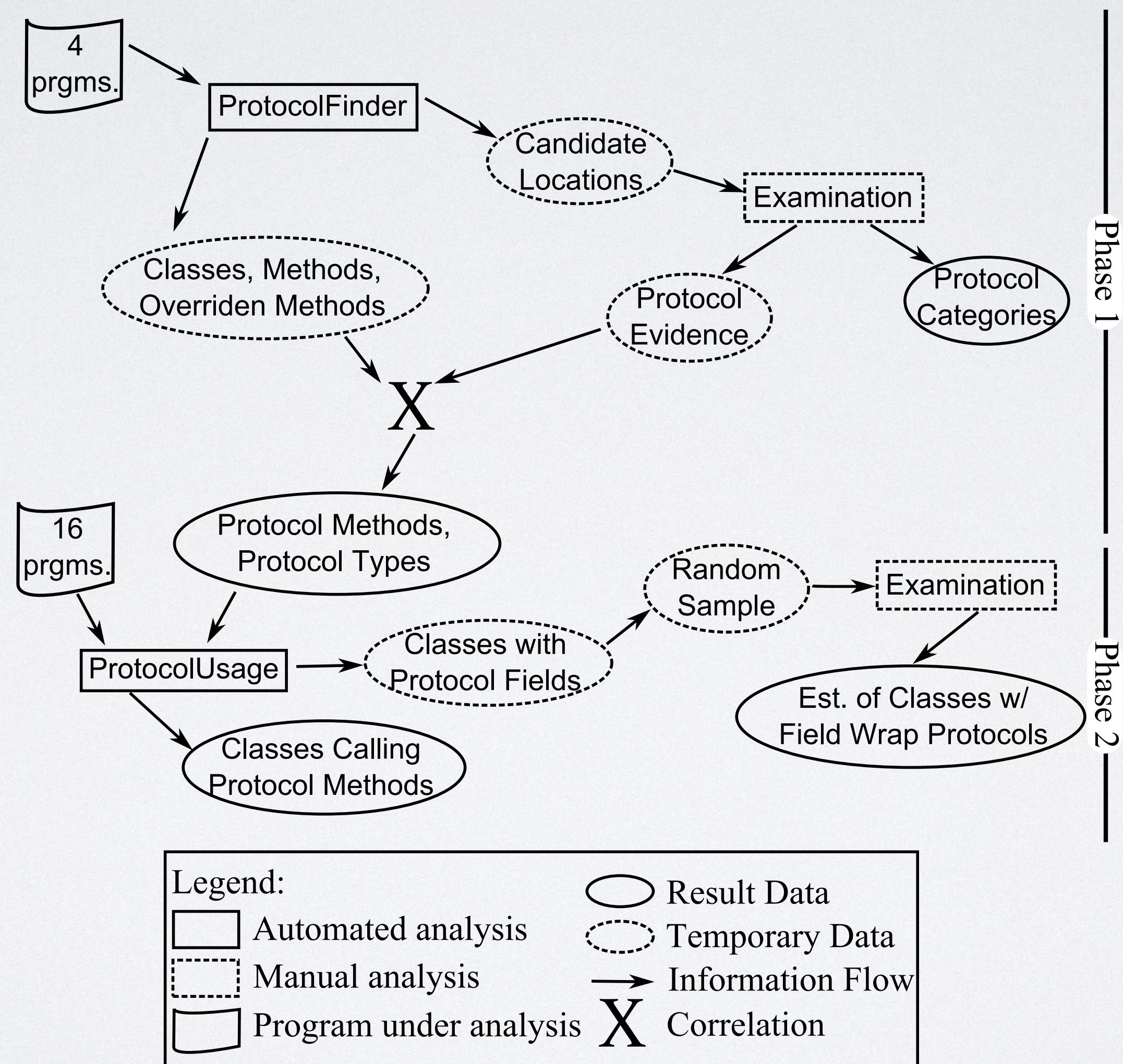
Analysis

- Dynamic analysis?

  - False negatives (missing protocols due to lack of test cases)

- Manual analysis?

  - Too slow/expensive

- Static analysis?

  - False positives

ProtocolFinder

- Sound (does it find every protocol)?

  - No (not even with respect to the paper's definition of *protocol*)

- Complete (does it ONLY find protocols)?

  - That depends on what a protocol is.

    ```
    if (f()) {

        throw new Error();

    }
    ```

- What about a broader definition of *protocol?*

Corpus (Phase 1)

- Four open source programs from Qualitas corpus

  - Large, popular programs; mix of libraries and applications

- 1.9 MLOC (Java)

- To what do you expect the results to generalize?

- What corpus would you have picked?

Number of Protocols

Table 2. The results of running the ProtocolFinder on the four phase one code bases

Program	Protocol Candidates	Protocol Evidence	E.C.	P.T.	T.S.E.C.	Precision	%E.C.	%P.T.
JSL	2,690	613	195	842	54	22.8%	2.3%	8.2%
PMD	32	7	3	10	0	21.9%	0.8%	2.4%
Azureus	136	24	19	32	4	17.6%	2.1%	2.6%
JDT	62	4	4	5	0	6.5%	1.3%	1.5%
Total	2,920	648	221	889	58	22.2%	2.2%	7.2%

T.S.E.C.=Thread-Safe Evidence Classes E.C.= Evidence Classes
P.T.= Protocol Types

Types of Protocols

Table 3. Categorization of each of the 648 reports issued by the ProtocolFinder that were evidence for actual protocols.

Category	Protocol Evidence	%
Initialization	182	28.1%
Deactivation	167	25.8%
Type Qualifier	106	16.4%
Dynamic Preparation	52	8.0%
Boundary	51	7.9%
Redundant Operation	47	7.3%
Domain Mode	31	4.8%
Others	12	1.9%

Phase 2: Protocol Usage

Table 4. The results of running the ProtocolUsage analysis on the sixteen candidate code bases.

Program	Classes Calling Protocol Methods	%	Classes w/ Prot. Fields	%	Exposes Protocol Rate	Est. Classes From Total
JSL	1012	12%	1082	13%	15%	157
PMD	85	22%	29	7%	0%	0
Azureus	198	22%	763	8%	31%	234
JDT	13	4%	18	6%	0%	0
ant	269	28%	187	19%	20%	37
antlr	20	11%	16	9%	0%	0
aoi	25	6%	37	8%	0%	0
columba	120	12%	246	25%	8%	18
crystal	9	5%	2	1%	0%	0
drjava	49	8%	107	17%	0%	0
freecol	94	22%	117	27%	0%	0
log4j	39	22%	32	18%	0%	0
lucene	30	11%	27	10%	0%	0
poi	41	10%	13	3%	100%	13
quartz	16	13%	10	8%	0%	0
xalan	91	9%	142	14%	13%	17
Total	2111	13%	2141	13%	17%	356
W/O JSL	1099	15%	1059	14%	18%	196

Implications on Language Design

- Do you wish you had typestate?

- Do these results mean we should?