STAR Laboratory of Advanced Research on Software Technology

Source Code-based Risk Analysis

W. Eric Wong
Department of Computer Science
The University of Texas at Dallas
ewong@utdallas.edu
http://www.utdallas.edu/~ewong

Speaker Biographical Sketch

- Professor & Director of International Outreach
  Department of Computer Science
  University of Texas at Dallas
- Guest Researcher
  Computer Security Division
  National Institute of Standards and Technology (NIST)
- Vice President, IEEE Reliability Society
- Secretary, ACM SIGAPP (Special Interest Group on Applied Computing)
- Principal Investigator, NSF TUES (Transforming Undergraduate Education in Science, Technology, Engineering and Mathematics) Project
  - *Incorporating Software Testing into Multiple Computer Science and Software Engineering Undergraduate Courses*
- Founder & Steering Committee co-Chair for the SERE conference (*IEEE International Conference on Software Security and Reliability*) (http://paris.utdallas.edu/sere13)

Outline

- Motivation
- Our Ongoing Effort
- Case Study I
- Case Study II
- Related Studies
- Case Study III
- Future Directions

Motivation

Motivation

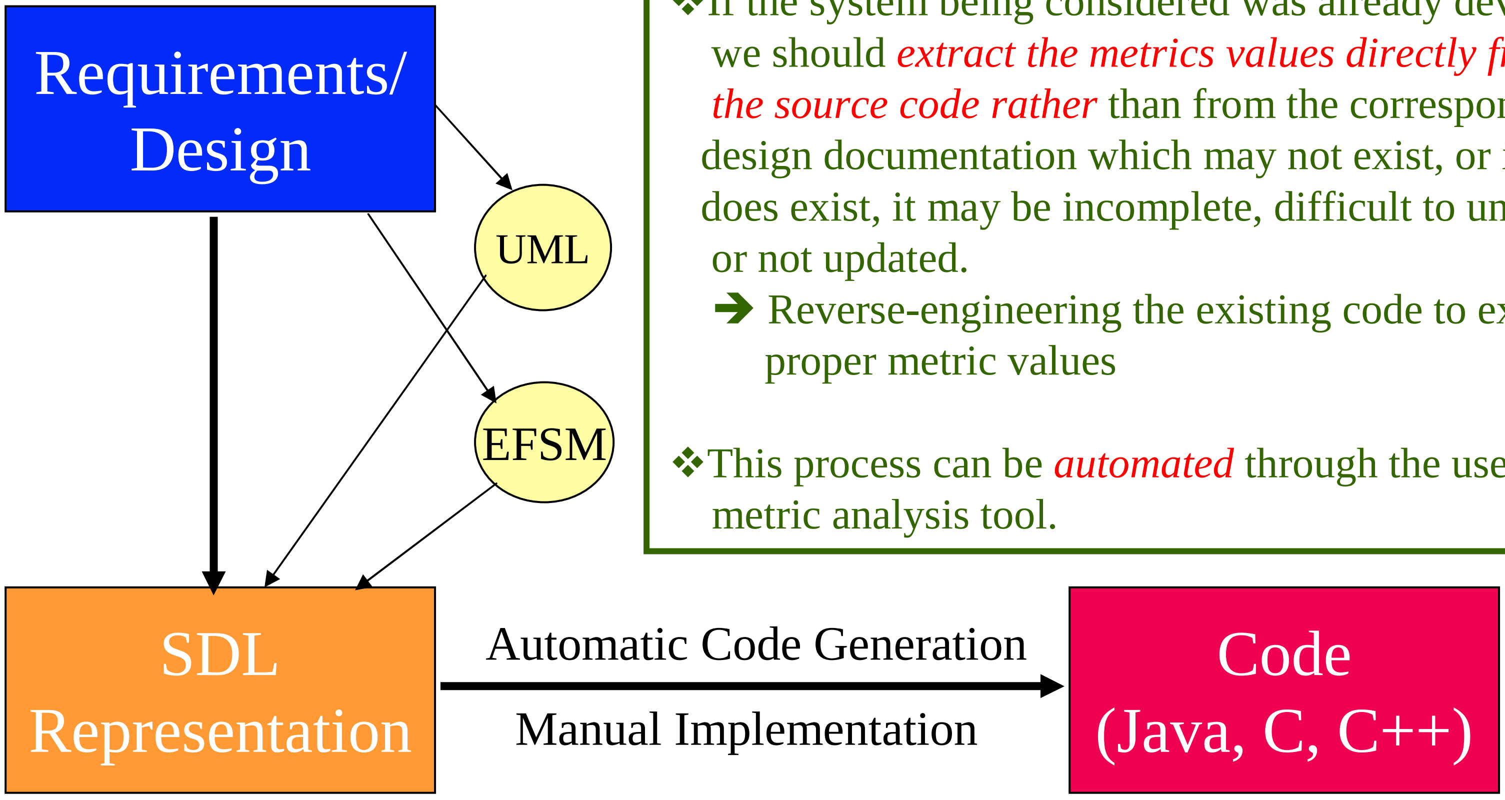
- Not all modules can be extensively reviewed and tested
- Approximately 20% of a software system is responsible for 80% of the faults
  - Pareto's principle: 80% of wealth is held by 20% of population
  - The same principle is applicable to other areas
- Need measurements to identify only *a small percentage* of the modules in a software system as fault-prone

V. Pareto, "On the Distribution of Wealth and Income, in Roots of the Italian School of Economics and Finance: From Ferrara (1857) to Einaudi (1944)," M. Baldassarri and P. Ciocca, eds., vol. 2, Houndmills: Palgrav, 2001.

When to Apply the Measurements **(1)**

- Can be applied to both *new and existing* projects *at different stages*
  - Design Documentation, SDL, Java, C, C++
- Possible Benefits
  - Determine better design when alternative choices exist
  - Assign fault-prone modules to experienced programmers
  - Allow more testing effort for these modules
    - ❑ Unit testing
    - ❑ Regression testing

When to Apply the Measurements (2)



- ❖If the system being considered was already developed, we should *extract the metrics values directly from the source code rather* than from the corresponding design documentation which may not exist, or if it does exist, it may be incomplete, difficult to understand, or not updated.
  - ➔ Reverse-engineering the existing code to extract the proper metric values
- ❖This process can be *automated* through the use of a metric analysis tool.

Our Ongoing Effort

Risk Analysis Model **(1)**

- A risk analysis-based model supported by tools to identify fault-prone software modules
  - Model Construction
    - ❑ Static metrics: can be collected at compile time (e.g., lines of code, McCabe complexity)
    - ❑ Dynamic metrics: cannot be collected until run-time (e.g., test coverage)
    - ❑ Internal metrics: focus on the internal structure of a software module (e.g., number of lines of code)
    - ❑ External metrics: emphasize the interactions between a software module and the rest of the system
    - ❑ Special metrics: unique to multi-threaded embedded software (e.g., number of threads through each module)
    - ❑ Process-based metrics: how a software system is built and operated

Risk Analysis Model **(2)**

– Model Validation and Refinement

- ❑ Evaluate the model against real defect data
- ❑ Refine it, if necessary, to improve its accuracy on fault-proneness prediction
- ❑ Use open-source applications and software developed at our member companies, as available

– Model Application

- ❑ Repeat the risk analysis at different stages of the software development lifecycle
- ❑ Our model is *flexible* as it uses various data depending on its availability

Problems of Current Approaches

- Only static and/or process-based metrics are used
- No dynamic information such as how each module has been tested is considered
- Results of many studies are inconsistent or contradictory

Our Solution

- Our solution overcomes existing drawbacks by *calibrating the static metrics-based complexity using dynamic metrics*, and also incorporating special metrics unique to software being analyzed.

Case Study I

Case Study I

- A Target Software System from Telcordia (formerly Bellcore)
  - A distributed software system with a client-server architecture
  - Three major components: server, clients, and database
  - Server written in C with embedded SQL statements
    - 35,000 non-blank/non-comment lines of C code
    - 20% of them converted from the embedded SQL statements
    - 97 files/530 functions

W. E. Wong, J. R. Horgan, M. Syring, W. M. Zage, and D. M. Zage, "Applying Design Metrics to Predict Fault-Proneness: A Case Study on a Large-Scale Software System," *Software-Practice and Experience*, 30(14):1587-1608, November 2000

Case Study I: Metrics $\mathcal{M}_i$, $\mathcal{M}_e$, **and** *$\mathcal{M}_c$*

- Five metrics were used
  - $\mathcal{M}_i$: an internal complexity metric which incorporates factors related to a function’s internal structure
  - $\mathcal{M}_e$: an external complexity metric which focuses on a function’s external relationships to the rest of the software system
  - $\mathcal{M}_c$,: a composite complexity metric which is a linear combination of $\mathcal{M}_i$ and $\mathcal{M}_e$
    - ❑ Other combinations should also be considered
  - The union of $\mathcal{M}_i$, $\mathcal{M}_e$, and $\mathcal{M}_c$
    - ❑ Do not care how a function gets selected (whether by $\mathcal{M}_i$, $\mathcal{M}_e$, or $\mathcal{M}_c$), just that it is
  - The intersection of $\mathcal{M}_i$, $\mathcal{M}_e$, and $\mathcal{M}_c$

W. Zage and D. Zage, “Evaluating Design Metrics on Large-Scale Software,” IEEE Software, 10(4):75-81, July 1993.

Case Study I: Metrics $\mathcal{M}_i$, $\mathcal{M}_e$, **and** *$\mathcal{M}_c$ (cont'd)*

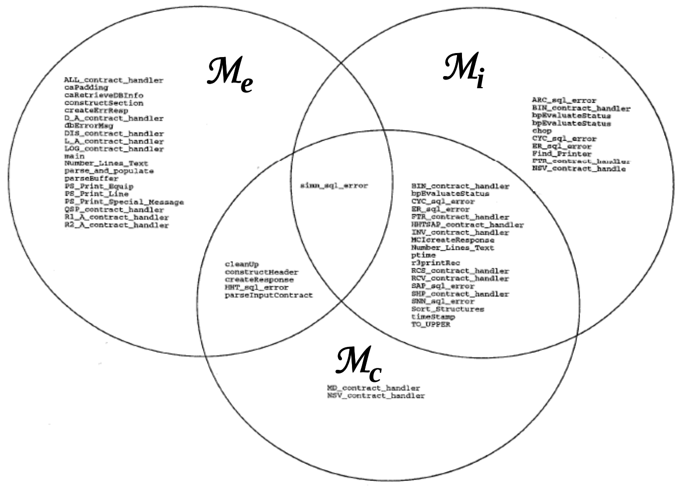
- $\mathcal{M}_i$: = i_1 (functions calls) + i_2 (Data Structure Manipulations) + i_3 (I/O)
  - Possible locations for the majority of faults
    - the point of a function call
    - statement with complex data structure manipulations
    - input/output statements
  - A function has a high $\mathcal{M}_i$ if it has complex code
  - Examples of functions with high $\mathcal{M}_i$
    - A function which has many pointer and array accesses
    - A *print* function which does a great deal of I/O
    - A function which calls a few functions (low fan-out) but calls them many times
  - Question: Is a function with a high $\mathcal{M}_i$ also fault-prone?

- $\mathcal{M}_e = e_1$ (inflows * outflows) + e_2 (fan-in * fan-out)
  - inflows and outflows: amount of data flow
  - fan-in: number of functions that call the target functions
  - fan-out: number of functions that the target function calls
  - A function has a high $\mathcal{M}_e$ if it has many interactions with other parts of the program
  - Examples of functions with high $\mathcal{M}_e$
    - a high level control function responsible for calling many other functions
    - A *utility* function called by many other functions
  - Question: Is a function with a high $\mathcal{M}_e$ also fault-prone?

Case Study I: Tool & Results

- Automatically collects and analyzes metrics values
  - A metric analysis tool (χMetrics) was implemented
  - Built on top of ATAC (an Automatic Testing and Analysis tool for C by Bellcore)
  - Code to be analyzed must be syntactically correct

- Identify functions with high metrics values
  - Functions identified by top 5% of $\mathcal{M}_i$, $\mathcal{M}_e$, and $\mathcal{M}_c$
    - ❑57 functions identified
    - ❑1 by all three
    - ❑20 by $\mathcal{M}_i$ and $\mathcal{M}_c$
    - ❑6 by $\mathcal{M}_e$ and $\mathcal{M}_c$
    - ❑10 by $\mathcal{M}_i$ only
    - ❑20 by $\mathcal{M}_e$ only
    - ❑2 by $\mathcal{M}_c$ only



Case Study I: Results

- Validate fault-prone functions by real defect data between two major releases for their fault-proneness

- Assume there are *m* functions identified as *fault-prone*; of these *l* are *fault-laden*. Assume *n* functions contain at least one fault based on the real defect data

- precision = $(\frac{l}{m})$ 100%, where 0% $\leq precision \leq$ 100%

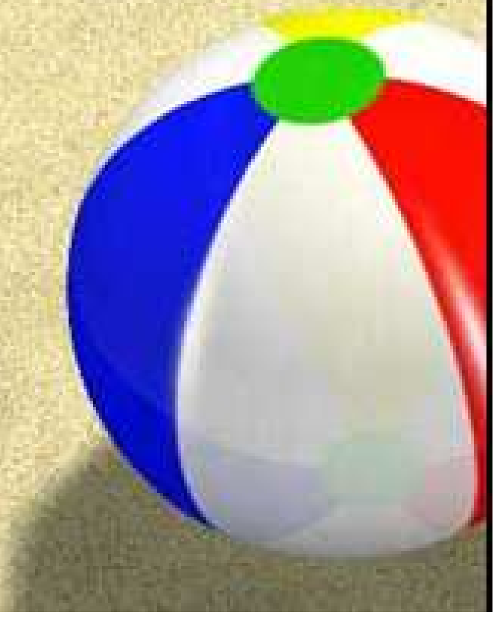
- recall = $(\frac{l}{n})$ 100%, where 0% $\leq recall \leq$ 100%

- Our experiment results suggest that
  - Metrics $\mathcal{M}_e$ and $\mathcal{M}_i$ can be good indicators of fault prone functions
  - No particular metric has a clear advantage over the others in determining which function is fault-laden

Impact of Function Size

- Function size may be related to the metrics values
- Do we need to do the normalization?
  - Dividing metrics values by function size determined by non-comment and non-blank lines of code
- Normalized metrics identify *very different functions* than non-normalized metrics.
- Validation using real defect data to determine which one is better?
  - Non-normalized metrics are better than the normalized metrics
- Function size is a factor that should be considered
  - Ostrand, Weyuker and Bell (*IEEE Transactions on Software Engineering*, 31(4):340-355, April 2005)
  - Our data indicate that in no case does the size-based selection catch all the functions selected by $\mathcal{M}_i$, $\mathcal{M}_e$,or $\mathcal{M}_c$.
  - This suggests that function size cannot replace any of the metrics.

Case Study II



Possible Improvement

- Questions
  - Why do we need a different set of complexity metrics?
    - ❑ If not, what kind of metrics can be used?
  - Is it easy to collect these metrics values?
    - ❑ "Data Structure Manipulations" is not only ill-defined but also difficult to collect

W. E. Wong, J. R. Horgan, J. C. Maldonado, and J. V. LaGrange, "Integrating Testing and Design Metrics to Predict Fault-Prone Software Modules," in *Proceedings of the 11th International Conference on Software Engineering & its Applications*, Paris, France, December, 1998.

Case Study II: Motivation

- Many studies explore the fault detection capability of test sets adequate with respect to
  - controlflow-based criteria (statement, block, decision)
  - dataflow-based criteria (all-uses, potential all-uses)
  - mutation-based criteria (interface mutation)

- No study has used testing metrics such as
  - number of all-uses
  - number of interface mutants to identify fault-prone software modules even before any testing is conducted.

Case Study II: Objective

- Develop a hybrid metric as an integration of *controlflow-, dataflow-* and *mutation-based testing metrics* and some components of the internal and/or external complexity metrics to better predict the fault-prone software modules.

Case Study II: Our Method

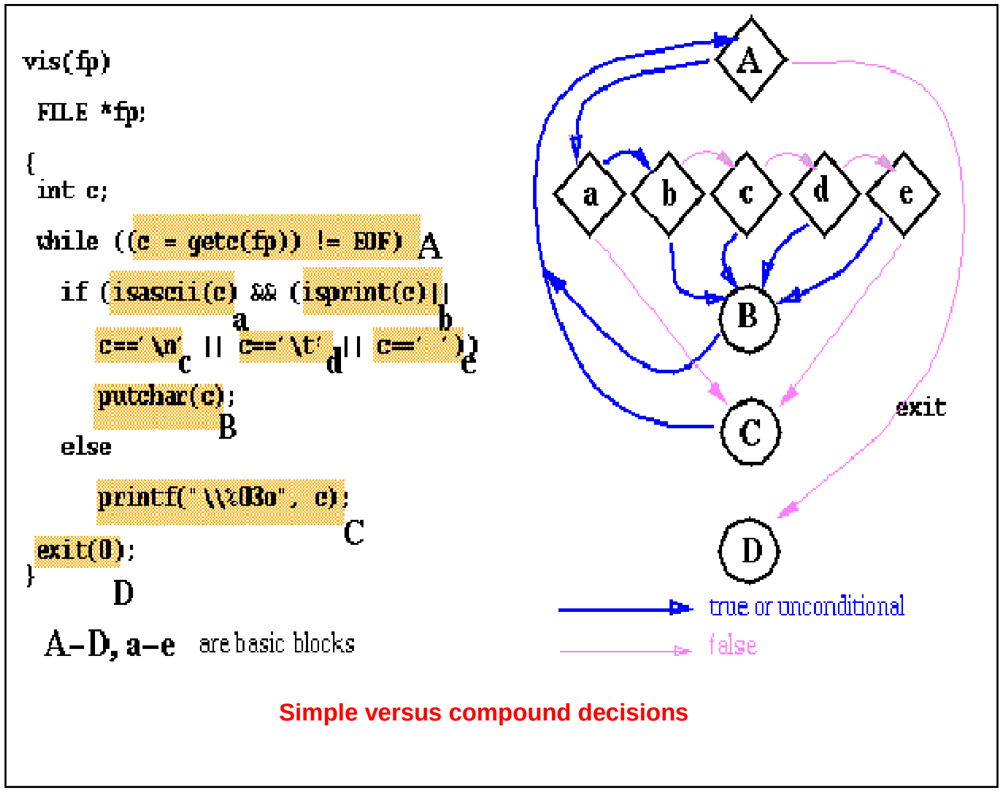
- Replace the *difficult-to-collect* and *mistake-prone* metrics used in computing the internal complexity $\mathcal{M}_i$ and the external complexity $\mathcal{M}_e$ by other *better defined* and *more easily collected* ones.
  - Replace "Data Structure Manipulations" (number of references to complex data types) by DEC (the number of decisions), AU (the number of all-uses), or PAU (the number of potential all-uses)
    - ❑ *None of the decisions, all-uses, and potential all-uses need to be covered*
  - Use IM (number of interface mutants) as an index for the probability of faults introduced because of the communications between two software modules
    - ❑ None of the mutants have to be compiled
    - ❑ *None will be executed on any test case*

Case Study II: Metrics

- $\mathcal{M}_{i0}$ = the original internal complexity metric
- $\mathcal{M}_{i1}$= DEC + AU
- $\mathcal{M}_{i2}$= DEC + AU + Function Calls + I/O
- $\mathcal{M}_{i3}$= DEC + PAU
- $\mathcal{M}_{i4}$= DEC + PAU + Function Calls + I/O
- $\mathcal{M}_{i5}$= DEC
- $\mathcal{M}_{i6}$= DEC + Function Calls + I/O
- $\mathcal{M}_{i7}$= AU
- $\mathcal{M}_{i8}$= AU + Function Calls + I/O
- $\mathcal{M}_{i9}$= PAU
- $\mathcal{M}_{i10}$= PAU + Function Calls + I/O
- $\mathcal{M}_{e0}$ = the original external complexity metric
- $\mathcal{M}_{e1}$= IM
- $\mathcal{M}_{c0} = \mathcal{M}_{i0} + \mathcal{M}_{e0}$
- $\mathcal{M}_{ck} = \mathcal{M}_{ik} + \mathcal{M}_{e1}$ where $1 \le k \le 10$

DEC - Number of Decisions

- A decision is a boolean predicate with two possible values, *true* and *false*



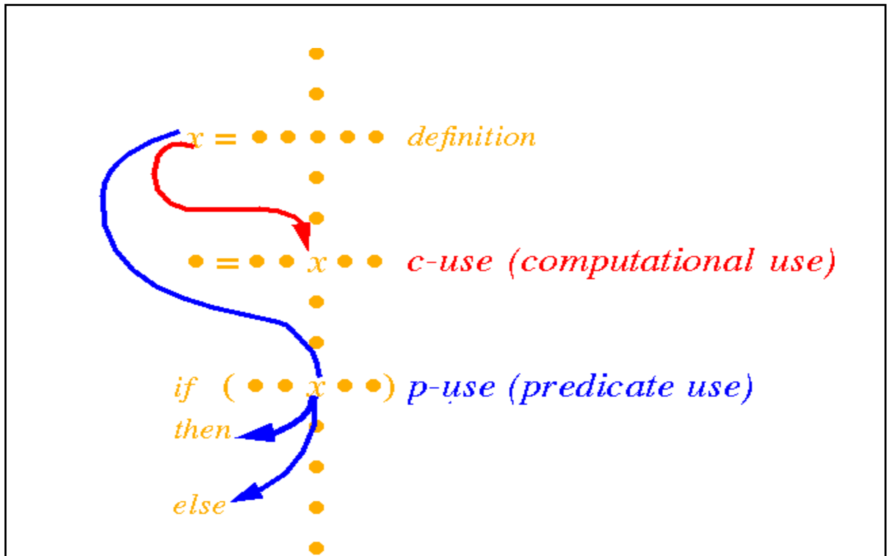
```
vis(fp)
 FILE *fp;
{
 int c;
 while ((c = getc(fp)) != EOF) A
   if (isascii(c) && (isprint(c)||
                 a             b
     c=='\n' || c=='\t' || c==' '))
           c          d         e
     putchar(c);
               B
   else
     printf("\\%03o", c);
                         C
 exit(0);
}
       D
```

A-D, a-e are basic blocks

Simple versus compound decisions

- McCabe’s Cyclomatic complexity = number of decision nodes + 1

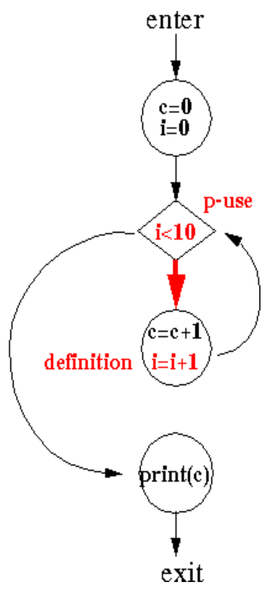
AU – Number of All-Uses



x = • • • • • definition
• = • • x • • c-use (computational use)
if (• • x • •) p-use (predicate use)
then
else

$$\text{All-uses coverage} = \frac{\text{(\# of p- and c-uses covered)}}{\text{(Total \# of feasible p- and c-uses)}}$$

```
main() {
  c=0;
  i=0;
  while ( i<10 ) {
    c=c+1;
    i=i+1;
  }
  print(c);
}
```



PAU – Number of Potential All-Uses

- With respect to variable *x*
  - one c-use (s_2, s_5)
  - no p-use
  - two potential c-uses (s_2, s_5) and (s_2, s_6)
  - two potential p-uses (s_2, s_4, s_5) and (s_2, s_4, s_6)
  - *x*=0 and *y*=3 are in the same block
    - ❑ No substitution

```
s1    read a;
s2    x = 0;
s3    y = 3;
s4    if (a > 0)
      then
s5       print x + 1;
      else
s6       print y + 2;
```

- All-uses: an *explicit* use of *x*
- Potential all-uses: a *potential* use of *x*

IM – Number of Interface Mutants

- Function parameters replacement
  - replacing a parameter by another appropriate one
- Function parameters switch
  - permuting the order of parameters in a call statement
- Function parameters elimination
  - deleting one parameter at a time
- Function parameters increment/decrement
  - inserting a pre-increment operator (++) or pre-decrement operator (--) before a parameter
- Unary operator insertion
  - inserting an arithmetic negation, a logical negation, or a bit negation before a parameter
- Function call deletion
  - deleting the call to a function. If the function call is part of an expression, it is replaced by a value from a pre-defined set.

Target Program & Tools Used

- The *SPACE* program which provides a language-oriented user interface for configuration of antenna arrays.
  - about 10K lines of C code in 134 files
  - Defect data were obtained from the error-log maintained during its testing and integration phases

- Tools
  - ATAC (χSuds): A data-flow coverage measurement tool to compute DEC and AU
  - POKE: another data-flow coverage measurement tool to compute PAU
  - Proteum/IM: a mutation analysis tool to compute IM
  - χMetrics: a metrics analysis tool to compute $\mathcal{M}_{i0}$, $\mathcal{M}_{e0}$, and $\mathcal{M}_{c0}$

Case Study II: Results

- DEC (number of decisions) by itself is *not* a good metric for predicting fault-proneness
- $\mathcal{M}_{i0}$ does not provide as good identification as other $\mathcal{M}_i$'s which use either AU (number of all-uses) or PAU (number of potential all-uses)
  - The internal complexity of a module can be better represented by AU or PAU than DSM (number of references to complex data types).
- The number of interface mutants serves as a better metric for identifying modules with faults due to the external communication of a module and the rest of the system.
- Our hybrid metrics can *better predict* fault-prone software modules than the internal and external complexity metrics used in Case Study I.

Case Study II: Results (cont'd)

- The purpose of using metrics is to get insight, not to compare their absolute values across different projects.
  - Relative comparison of metrics values within the same project (or similar projects) is useful
- This is supported by the study conducted by Nagappan, Ball, and Zeller (Mining Metrics to Predict Component Failures, in *Proceeding of the 28th International Conference on Software Engineering*, pp. 452-461, Shanghai, China, May 2006.)
  - Predictors are accurate only when obtained from the same or similar projects

Related Studies

Related Studies **(1)**

- There are many related studies. Here are a few.
  - Norman E. Fenton and Niclas Ohlsson “Quantitative Analysis of Faults and Failures in a Complex Software System,” *IEEE Transactions on Software Engineering*, 26(8):797-814, August 2000.
  - A. G. Koru and Jeff Tian, “An Empirical Comparison and Characterization of High Defect and High Complexity Modules,” *The Journal of Systems and Software*, 67(3):153-163, September 2003
  - A.G. Koru and J. Tian, “Comparing High-Change Modules and Modules with the Highest Measurement Values in Two Large-Scale Open-Source Products**,”** *IEEE Transactions on Software Engineering*, 31(8):625-642, August 2005
  - N. Nagappan, T. Ball, and A. Zeller, “Mining Metrics to Predict Component Failures,” in *Proceeding of the 28th International Conference on Software Engineering*, pp. 452-461, Shanghai, China, May 2006.
  - T. Ostrand, E. Weyuker, and R. Bell, “Predicting the Location and Number of Faults in Large Software Systems,” *IEEE Transactions on Software Engineering*, 31(4):340-355, April 2005
  - V. R. Basili and B.T. Perricone, “Software Errors and Complexity: An Empirical Investigation,” *Communications of the ACM*, 27(1)42-52, January 1984
  - T. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan, and N. Goel, “Early Quality Prediction: A Case Study in Telecommunications," *IEEE Software*, 13(1):65-71, January 1996
  - S. R. Chidamber and C. F. Kemerer, “A Metrics Suite for Object Oriented Design,” *IEEE Transactions on Software Engineering*, 20(6):476-493, June 1994.
  - M. Lorenz and J. Kidd, *Object-Oriented Metrics: Measures of Complexity*. Pearson Education, December 1995.
  - L. C. Briand, J. W. Daly, and J. K. Wust, “A Unified Framework for Cohesion Measurement in Object-Oriented Systems,” *Journal of Empirical Software Engineering*, 3(1)65-117, 1998.
  - L. C. Briand, J. W. Daly, and J.K. Wust, “A Unified Framework for Coupling Measurement in Object-Oriented Systems,” *IEEE Transactions on Software Engineering*, 25(1):91-121, January/February 1999

Related Studies (2)

- Norman E. Fenton and Niclas Ohlsson "Quantitative Analysis of Faults and Failures in a Complex Software System," *IEEE Transactions on Software Engineering*, 26(8):797-814, August 2000.
  - A small number of modules contain most of the faults discovered in prerelease testing, and a very small number of modules contain most of the faults discovered in operation.
  - *Neither of these phenomena can be explained by the size or the complexity of the modules*.
  - No evidence to support the relationship between module size and fault density
  - *No evidence that popular complexity metrics are good predictors of either fault-prone or failure-prone modules*

Related Studies **(2)** *(cont'd)*

– The number of faults discovered in prerelease testing is an order of magnitude *greater* than the number discovered in 12 months of operational use

– Those modules which are the most fault-prone prerelease are among the least fault-prone post-release.

– Fail to take into account *testing effort* and *operational usage*.

Related Studies (3)

- A. G. Koru and Jeff Tian, “An Empirical Comparison and Characterization of High Defect and High Complexity Modules,” *The Journal of Systems and Software*, 67(3):153-163, September 2003
  - Six software products: two from IBM and four from Nortel Networks
  - Examine the relationship between high defect modules and high complexity modules
    - ❑ positively correlated but not monotonic relationship
  - The most complex modules often have an acceptable quality
    - ❑ Many complex modules are intrinsically complex because of the problem they are dealing with, and are recognized as such. Consequently, highly skilled personnel and adequate effort were allocated to such modules, resulting in their relative high quality
  - The high defect modules are not typically the most complex ones, but slightly below the most complex ones
    - ❑ Those ‘‘not too big (complex) not too small (simple)’’ modules may contain the highest number of defects.

Related Studies **(4)**

- A.G. Koru and J. Tian, “Comparing High-Change Modules and Modules with the Highest Measurement Values in Two Large-Scale Open-Source Products,” IEEE Transactions on Software Engineering, 31(8):625-642, August 2005
  - Two large-scale open-source products, Mozilla and OpenOffice
  - Identifying *change-prone modules* can help programmers take preventive actions to reduce maintenance cost and improve software quality.
  - Top modules in change count rankings and the modules with the highest values of static metrics are different
    - ❑ High-change modules have high metrics values but not the highest
    - ❑ In Mozilla, the top 2%, 5%, and 20% modules in the change rankings include 34.78%, 51.45%, and 82.43% of the total change count
      - ➢ Similar to the 80/20 principle

Related Studies (5)

- N. Nagappan, T. Ball, and A. Zeller, "Mining Metrics to Predict Component Failures," in *Proceeding of the 28th International Conference on Software Engineering*, pp. 452-461, Shanghai, China, May 2006.
  - A study uses five projects from Microsoft
  - For each project, we can find a set of complexity metrics that correlates with post-release defects and the corresponding failures
  - There is no single set of metrics that fits all projects
  - Predictors obtained from principal component analysis are useful in building regression models to estimate post-release defects
  - Predictors are accurate only when obtained from the same or similar projects
  - DO NOT use complexity metrics without validating them for your project DO use metrics that are validated from history to identify low-quality components

Related Studies **(6)**

- T. Ostrand, E. Weyuker, and R. Bell, "Predicting the Location and Number of Faults in Large Software Systems," *IEEE Transactions on Software Engineering*, 31(4):340-355, April 2005
  - Develop a model to predict the expected number of faults in each file of the next release of a system based on *the code in the current release*, and *fault and modification history* of the file from previous releases
  - Two projects from AT&T
    - ❑ one with 17 consecutive quarterly releases over 4 years
    - ❑ one with 9 releases over 2 years
  - The 20% of the files with the highest predicted number of faults contain between 71% and 92% of the faults that were actually detected, with the overall average being 83%
    - ❑ another support for the 80/20 principle

Related Studies **(6)** *(cont'd)*

– A simplified version of the predictor (*sorting files by their size*) selects 20% of the files that are the largest containing on average, 73% and 74% of the faults for the two systems.

❑ Module size is the most significant factor influencing the number of faults

- El Emam *et al.* also noted that there is *a continuous relationship* between size and defects.

Related Studies (7)

- V. R. Basili and B.T. Perricone, “Software Errors and Complexity: An Empirical Investigation,” Communications of the ACM, 27(1)42-52, January 1984
  - Larger modules tended to have a lower fault density than smaller ones, where fault density is the number of faults discovered divided by a measure of module size (e.g., thousands lines of code)
- T. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan, and N. Goel, “Early Quality Prediction: A Case Study in Telecommunications," *IEEE Software*, 13(1):65-71, January 1996
  - Measures of early products of development can predict the final product’s quality.
    - ❑ We should apply metrics to SDL specifications

Related Studies **(8)**

- Metrics for Object-Oriented Code
  - S. R. Chidamber and C. F. Kemerer, “A Metrics Suite for Object Oriented Design,” *IEEE Transactions on Software Engineering*, 20(6):476-493, June 1994.
    - ❑ Depth of Inheritance Tree (DIT)
    - ❑ Coupling between object classes (CBO)
    - ❑ Number of Children (NOC) = number of immediate subclasses subordinated to a class in the class hierarchy
    - ❑ Lack of Cohesion in Methods (LCOM)
  - M. Lorenz and J. Kidd, *Object-Oriented Metrics: Measures of Complexity*. Pearson Education, December 1995.

Related Studies (8) (cont'd)

- L. C. Briand, J. W. Daly, and J. K. Wust, "A Unified Framework for Cohesion Measurement in Object-Oriented Systems," *Journal of Empirical Software Engineering*, 3(1)65-117, 1998

- L. C. Briand, J. W. Daly, and J.K. Wust, "A Unified Framework for Coupling Measurement in Object-Oriented Systems," *IEEE Transactions on Software Engineering*, 25(1):91-121, January/February 1999

❑ Coupling and inheritance measures are strongly related to the probability of having defects in a class

Case Study III

What is next ?

?

Quiz

- Given two modules C_1 and C_2
- C_1 has higher metrics values but also is significantly more tested than C_2

- Is the probability of C_1 containing any faults still higher than that of C_2?

Case Study III: Our Method

- Combining *dynamic testing effort* such as code coverage and execution counts with *static complexity* computed by using the internal and external metrics
  - Fault-proneness of a module with high static complexity should be *appropriately calibrated* based on how much effort has been spent on testing it.

- Fault-proneness of a module =

$$f\,\underbrace{(\text{inflows, outflows, fan-in, fan-out, ………}}_{\text{internal/external complexity metrics}}$$

$$\underbrace{\text{\#of decisions, \# of def-uses, \# of interface mutants, ……}}_{\text{controlflow-/dataflow-/mutation-based testing metrics}}$$

$$\underbrace{\text{block coverage, decision coverage, execution counts}}_{\text{dynamic testing effort}}\text{, …...)}$$

Case Study III: Our Advantage

- We have coverage measurement tools for C, C++, Java bytecode, and SDL
  - Code coverage
  - Execution counts

- These tools can also be easily modified to collect other metrics values
  - χSuds (C and C++)
  - eXVantage (C, C++, and Java bytecode)
  - χSuds/SDL

Case Study III: Source Code-Based Risk Assessment

- A risk in a software system can be viewed as a potential problem, and a problem is a risk that has manifested.
  - In order to reduce the risk of software operations, code which has the potential to cause problems has to be identified so that necessary actions can be taken to prevent any such problems from occurring.

W. E. Wong, Y. Qi, and K. Cooper, “Source Code-Based Software Risk Assessing,” in *Proceedings of The 20th ACM Symposium on Applied Computing* (ACM SAC), pp. 1485-1490, Santa Fe, New Mexico, March 13-17, 2005

Case Study III: Two Important Principles

- The more complex the static structure of the code, the higher the risk it has.
- The more thoroughly the code is tested, the lower the risk it has.

Case Study III: Our Approach (1)

- We propose two types of risk models: *static* models and *dynamic* models using metrics collected based on the source code

- The computation of the risks of code is automated at different granularity levels ranging from basic blocks to functions, files, subsystems, etc.
  - A basic block, also known as a block, is a sequence of consecutive statements or expressions containing no transfers of control except at the end, so that if one element of it is executed, all are.
    - ❑ This of course assumes that the underlying hardware does not fail during the execution of a block.

Case Study III: Our Approach (2)

- The risks of functions or files or subsystems are built *on top of the risks of their member blocks.*
-
  - Their risks are the highest risk of their member blocks.
  - For example, suppose a function f contains three blocks b_1, b_2 and b_3.
  - The risk of these blocks for containing some bug(s) are α, β, and γ, respectively, where $\alpha > \beta > \gamma$.
  - Then, the risk of f containing some bug(s) is the *highest risk of its member blocks*, that is, the highest of α, β, and γ which is α.

Case Study III: Risk Models **(1)**

- The static model is constructed by using metrics related to the *static* structure of the code such as the number of c-uses, p-uses, definitions, decisions and function calls.

- The dynamic model uses whatever is selected by the static model and additional *dynamic* test coverage of the code such as decision, c-use and p-use coverage to calibrate the metric values used in the model.

Case Study III: Risk Models **(2)**

- Users can choose *all or some* of the following metric components with *appropriate weighting factors* for model construction.
  - number of variable definitions (V),
  - number of function calls (F),
  - number of decisions (D),
  - number of c-uses (C), and
  - number of p-uses (P) are selected.
- They can also select from one of the two modeling schemes:
  - a *summation* scheme which constructs a risk model by using *the sum of* the selected metrics, or
  - a *product* scheme which uses *the product of* the selected metrics.
  - Suppose also the summation scheme is selected.
    The corresponding static risk model can be expressed as

$$V * \alpha + F * \beta + D * \gamma + C * \varepsilon + P * \omega$$

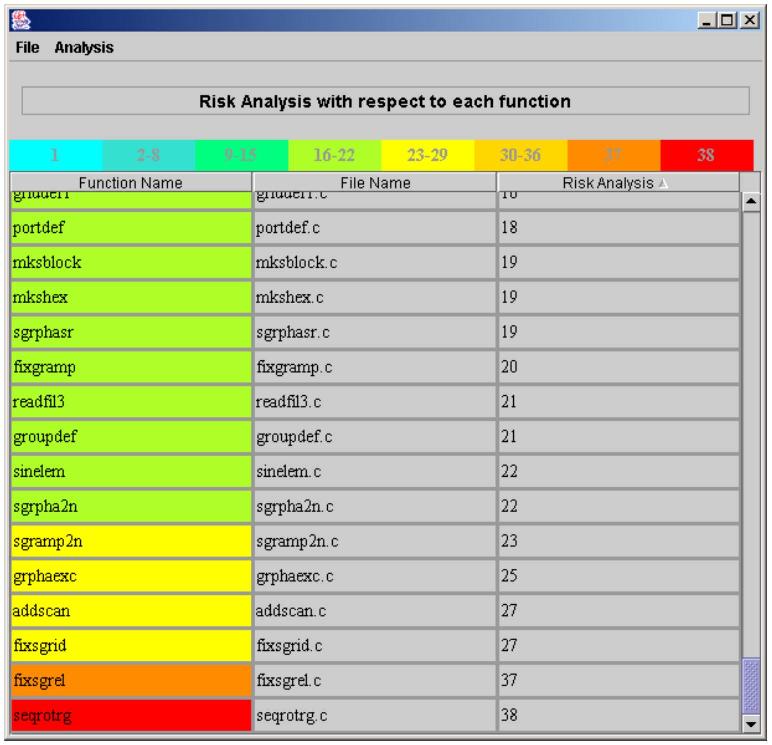
where α, β, γ, ε, and ω are the weighting factors.

Case Study III: Static Model versus Dynamic Model

- The major difference between a static and a dynamic model is that the latter uses test coverage to calibrate the actual value of each metric component.
  - Suppose we want to use the model listed in Equation (1) to compute the risk of a block *B*.
  - Suppose also block *B* has 20 c-uses.
  - *For a static model*, the metric component *C* (number of c-uses) has the value 20.
  - *For a dynamic model*, the actual value of *C* is the number of c-uses in *B* which are not covered (i.e., not executed) by all the successful tests before an execution failure is observed.
  - Assuming eight (8) of them are covered.
  - This makes the metric component *C* have a value 12 (obtained by subtracting 8 from 20) when a dynamic model (assuming the metric component *C* is selected) is used to compute the risk of block *B*.
    - ❑ A different approach is to make C equal 20 – 8 * ε where $0 < \varepsilon \leq 1$
    - ❑ Disadvantages versus advantages

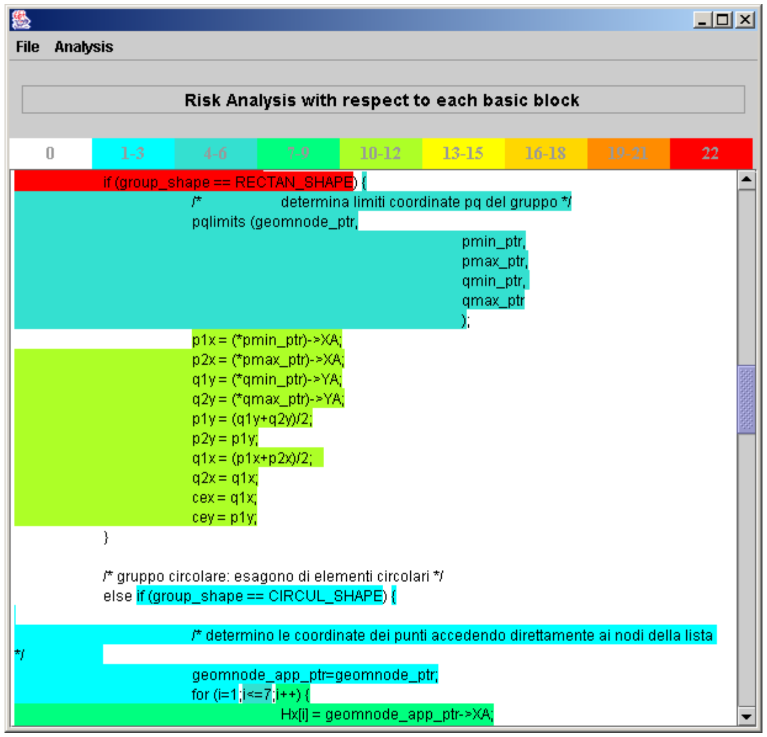
Case Study III: Visualize Risk in Code

- We compute the risk of each block and function, prioritize them in terms of the risk, and highlight them in different colors



Function Name	File Name	Risk Analysis
gridderr	gridderr.c	16
portdef	portdef.c	18
mksblock	mksblock.c	19
mkshex	mkshex.c	19
sgrphasr	sgrphasr.c	19
fixgramp	fixgramp.c	20
readfil3	readfil3.c	21
groupdef	groupdef.c	21
sinelem	sinelem.c	22
sgrpha2n	sgrpha2n.c	22
sgramp2n	sgramp2n.c	23
grphaexc	grphaexc.c	25
addscan	addscan.c	27
fixsgrid	fixsgrid.c	27
fixsgrel	fixsgrel.c	37
seqrotrg	seqrotrg.c	38

A display at the function level



```
        if (group_shape == RECTAN_SHAPE) {
                /*            determina limiti coordinate pq del gruppo */
                pqlimits (geomnode_ptr,
                                              pmin_ptr,
                                              pmax_ptr,
                                              qmin_ptr,
                                              qmax_ptr
                                              );
                p1x = (*pmin_ptr)->XA;
                p2x = (*pmax_ptr)->XA;
                q1y = (*qmin_ptr)->YA;
                q2y = (*qmax_ptr)->YA;
                p1y = (q1y+q2y)/2;
                p2y = p1y;
                q1x = (p1x+p2x)/2;
                q2x = q1x;
                cex = q1x;
                cey = p1y;
        }

        /* gruppo circolare: esagono di elementi circolari */
        else if (group_shape == CIRCUL_SHAPE) {

                /* determino le coordinate dei punti accedendo direttamente ai nodi della lista
*/
                geomnode_app_ptr=geomnode_ptr;
                for (i=1;i<=7;i++) {
                        Hx[i] = geomnode_app_ptr->XA;
```

A display at the block level

Case Study III: Results

- Our static risk models perform well by identifying a *small* percentage of functions and blocks as fault-prone.

- In many cases, the dynamic models perform better than the corresponding static models.

- How much? This depends on a few factors
  - the test coverage of the successful tests
  - the nature of the fault

Case Study III: Three Important Aspects

- Our models use metrics based on the *source code,* whereas other risk assessments rely on the data collected from the *process* of how a software system is built, operated, deployed or maintained, and which organization is responsible for such activities.
  - Data collected for the latter would be affected by different interpretations of the process
  - Data collected with respect to the metrics used for constructing our risk models do not have such a potential inconsistency problem
- Our method allows the risk of code to be computed at a very fine granularity level such as a basic block.
  - This is very different from many other studies which only identify the fault proneness at the module (or function) level.

Case Study III: Three Important Aspects (cont'd)

- Our models use either the static, or the static and the dynamic, data collected from the source code.
  - The risk of code can be computed by using only the static data. However, if the dynamic test coverage is available, it can be used to calibrate the metric values used for computing the risk.
  - This makes our risk model flexible, depending on the available data.