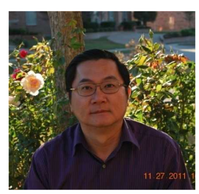
STAR Laboratory of Advanced Research on Software Technology

Understanding Your Code in a More Cost-Effective Way

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

- Locating Program Features using Execution Slices
- Quantifying the Disparity, Concentration, and Dedication between Program Components and Features
- Measuring Distance between Program Features

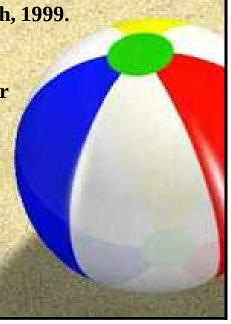
PART I

Locating Program Features using Execution Slices

W. E. Wong, S. S. Gokhale, J. R. Horgan, and K. S. Trivedi, "Locating Program Features using Execution Slices," in *Proceedings of the Second IEEE Symposium on Application-Specific Systems and Software Engineering Technology*, pp. 194-203, Richardson, Texas, March, 1999.

Click on the following pdf icon to download the paper

Adobe Acrobat Document



What is a Feature

- A feature is an *abstract description* of a functionality given in the specification
  - Example: the ATM software has three features
    - Withdraw
    - Deposit
    - Balance
  - Example: the UNIX wordcount program has three features
    - Number of lines, words, characters

Do You Understand Customer Feedback

- Requirement changes and enhancement requests are usually specified in terms of features affected, not in terms of software components that must be modified
- Example: A telephone switch
  - Call Setup
  - Call Waiting
  - Speed Dialing

Here is How

- Software developers must locate and understand the code associated with the affected features before they can translate change requests into code change

Features $\xleftrightarrow[\textbf{mapping}]{}$ **Software components**

Software Structure from the Program Feature Point of View

- Well-designed Software Systems
  - A high degree of cohesion
    - A cohesive module should ideally do just one thing
  - A low degree of coupling
  - Each module addresses a specific subfunction of the requirements and has a simple interface when viewed from other parts of the program structure
  - A clear mapping between each feature and its corresponding code segments
- Software systems in the real world
  - Low cohesion & high coupling
  - Program features are mixed together in the code – across modules which are seemingly unrelated

A Challenge

- In a complex software system it is not unusual to find that modifications made to one feature, which can be viewed as a functionality of the system, have *adverse impacts* on other *seemingly unrelated features*
- Such impacts can subsequently change the behavior of those features and *cause a system failure*
- Need a good understanding of the system

Objective

- Locate program code relevant to a particular feature in order to provide software programmers and maintainers with a good *starting point* for quick program understanding
  - Develop novel heuristics and experiment with them to identify
    - Code unique to the given feature
    - Code common to the given feature and others
  - Examine factors which affect the code so identified

Three Different Approaches

- Systematic
  - Provides a good understanding
  - Impractical for large complicated systems
- As-needed
  - Less expensive and less time-consuming
  - Miss some non-local interactions between features
- Execution Slice-based
  - An execution slice is the set of program components (blocks, decisions, c-uses, or p-uses) executed by a test input
- Qualitative description versus quantitative measurement

Reading Documentation Does Not Work

- Does not exist
- Incomplete and difficult to understand
- Not updated
- Implementation spread across several non-adjacent modules
- Do not want to read it

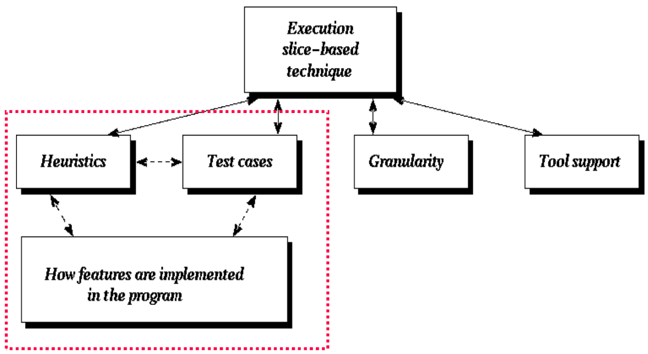
A Factor to Consider

- Most software development projects have a set of regression tests to help find bugs in the next release. But these tests can be used for more than just finding errors.

A Better Strategy

- Instead of spending time trying to understand the system, **let the system tell you how it works.**

Execution Slice-Based Technique



Code identified depends on which *heuristic* is applied, which by itself is affected not only by how *invoking and excluding tests* are selected but also how the *features are implemented* in the program.

Definitions: Invoking & Excluding Tests

- For a given program *P* and a feature *F*
  - An invoking test is a test that when executed on *P* shows the functionality of *F*
  - Example
    - *P*: The wordcount program (wc)
    - *F*: The functionality to count the ***number of characters***
      - ''**wc -c data**'' (say t_1) is an invoking test
  - An excluding test is one that does not
      - ''**wc -w data**'' is an excluding test
  - An invoking test is focused on F if it exhibits only F and no other features
      - t_1 is a focused invoking test
      - "**wc data**" is NOT because in addition to the number of characters, it also returns the number of words and lines

Number of Tests Required

- Do not need to use all the invoking and excluding tests for a given *P* and *F*
- *T* = {a few carefully selected test cases}
  - A few invoking tests (preferred to be the focused ones)
  - A few excluding tests
- Three notations
  - $\cup_{invoking}$ = {code executed by at least one invoking test in *T*}
  - $\cup_{excluding}$ = {code executed by any tests in *T* that do not exhibit *F*}
  - $\cap_{invoking}$ = {code executed by every invoking test in *T*}

Heuristics for Finding Code Unique to a Feature (1)

- Only one invoking test and one excluding test
  - Invoking test : the one with the *smallest* execution slice
  - Excluding test : the one with the *largest* execution slice
- $\cap_{\textbf{invoking}} - \cup_{\textbf{excluding}}$ = {code that is commonly executed by *all* invoking tests but not *any* excluding test}
  - Example
    - F_α can only be exhibited if either F_β or F_γ is also exhibited
      - All the invoking tests for F_α must also exhibit at least F_β or F_γ and perhaps many other features.
      - F_α has *no focused invoking tests.*
    - Use $\cap_{\textbf{invoking}} - \cup_{\textbf{excluding}}$ to find the code that is *uniquely* related to F_α

Heuristics for Finding Code Unique to a Feature (2)

- $\cup_{\textbf{invoking}} - \cup_{\textbf{excluding}}$ = {code that is executed by *any* invoking test but not by *any* excluding test}
  - Example
    - F_α is *not bundled* with F_β or F_γ in the way just described.
    - F_α can be exhibited by itself without other features being exhibited simultaneously

Heuristics for Finding Code Common to Features

- Code common to F_1 and F_2 is what is executed by at least a test that exhibits only F_1 and not F_2, and at least a test that exhibits only F_2 and not F_1
  - $\cup_{\text{invoking}}$ for F_1 = {code executed by tests which exhibit only F_1 and no other features (or at least not F_2)}
  - $\cup_{\text{invoking}}$ for F_2 = {code executed by tests which exhibit only F_2 and no other features (or at least not F_1)}
  - Code common to F_1 and F_2 = $(\cup_{\text{invoking}}$ for $F_1) \cap (\cup_{\text{invoking}}$ for $F_2)$

Selecting Invoking and Excluding Tests (1)

- Different sets of code may be identified by different sets of invoking and excluding tests
- Poorly selected tests will lead to inaccurate identification
  - Example
    - Including code that is not unique to a given feature
    - Excluding code that should not be excluded
- Find code unique to a given feature
  - Invoking tests should be *focused* on the feature being located, if possible
  - Excluding tests should be *as similar* (in terms of execution slice) *as possible* to the invoking tests in order to *filter out as much common code as possible*

Selecting Invoking and Excluding Tests (2)

- Find code common to a group of features
  - For each feature, its invoking tests should be *focused* with respect to this feature, if possible
  - The invoking tests for a feature should be *as dissimilar* (in terms of execution slice) *as possible* to the invoking tests for other features in the group in order to *exclude as much uncommon code as possible*

Components

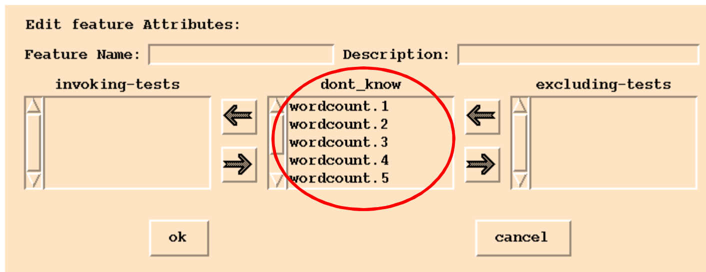
- Program components can be files, functions, blocks, decisions, c-uses, and p-uses

Requirements for Using χVue

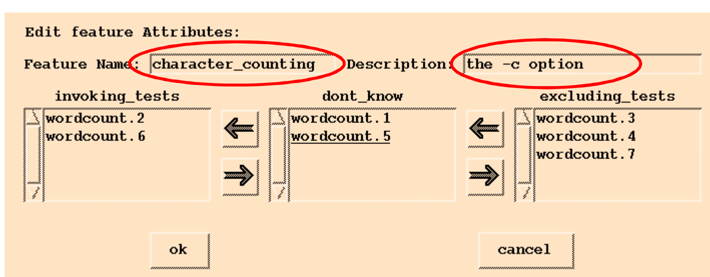
- χVue is part of χSuds (a Software Understanding and Diagnosis System) developed at Telcordia (formerly Bellcore)
- Effective use of χVue requires only that the programmer has a basic understanding of the program's features and can identify some invoking as well as excluding tests
- By default, every test is in the dont_know category

Visualizing Features in Code Using χVue (1)

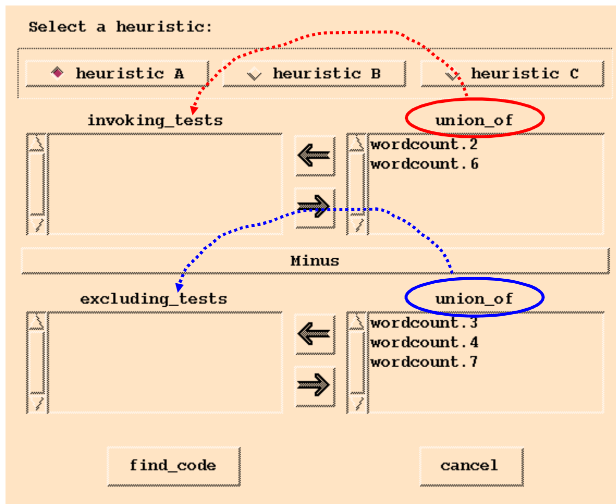
- Slice 15 (counting *characters* of the *wordcount* program)
- Chapter 12 of the χSuds User's Manual



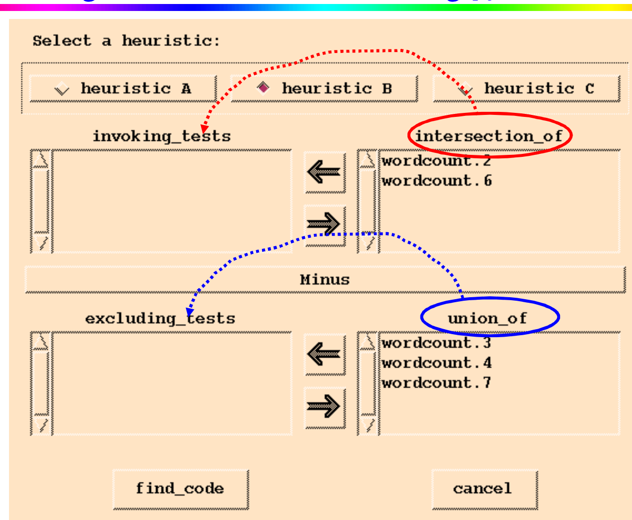
Visualizing Features in Code Using χVue (2)



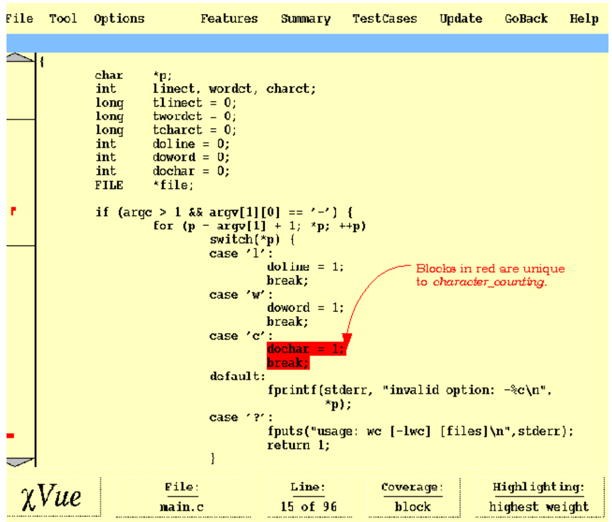
Visualizing Features in Code Using χVue (3)



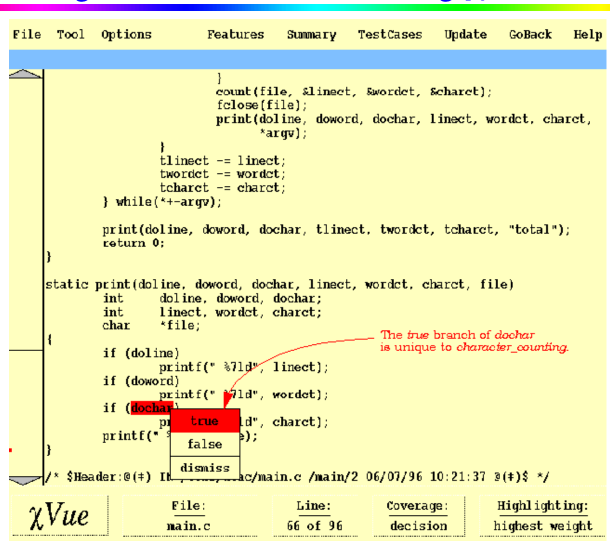
Visualizing Features in Code Using χVue (4)



Visualizing Features in Code Using χVue (5)



Visualizing Features in Code Using χVue (6)



A Case Study on SHARPE

- SHARPE
  - A Symbolic Hierarchical Automated Reliability and Performance Evaluator
  - 35,412 lines of C code in 30 files
  - 373 functions
  - Examined five features are studied
    - ❑ MC (Markov Chain) (f_1)
    - ❑ MRM (Markov Reward Models) (f_2)
    - ❑ GSPN (Generalized Stochastic Petri-Nets) (f_3)
    - ❑ PFQN (Product-Form Queuing Networks) (f_4)
    - ❑ FT (Fault Trees) (f_5)

Number of Blocks Unique to $F_{3,j}$ ($1 \leq j \leq 10$)

	$F_{3,1}$	$F_{3,2}$	$F_{3,3}$	$F_{3,4}$	$F_{3,5}$	$F_{3,6}$	$F_{3,7}$	$F_{3,8}$	$F_{3,9}$	$F_{3,10}$
analyze.c	9	26	9	26		9		9	26	26
multpath.c										
share.c	33	63	33	63		33		33	63	63
reachgraph.c	11	32	11	32		11		11	32	32
pfqn.c										
mpfqn.c										
util.c		3		3					3	3
inspade.c										
indist.c										
inshare.c										
maketree.c										
results.c		1					1			
cg.c		136		136					136	136
in_qn_pn.c	3	7	3	7		3		3	7	7
inchain.c	3	31	3	31		3		3	31	31
bind.c	21	26	19	39	19	14	12	14	35	27
bitlib.c										
sor.c	137		137			137		137		
newcg.c										
phase.c		307		307					307	307
newphase.c										
newlinear.c										
cexpo.c		37		72	25				37	37
expo.c		25		29	4				25	25
read1.c	9	14	9	33	4	4	5	9	15	15
symbol.c		20		20					20	20
ftree.c										
debug.c										
uniform.c										
mtta.c										
Total	226	728	224	798	52	214	18	219	737	729
Percentage	1.92	6.19	1.91	6.79	0.44	1.82	0.15	1.86	6.27	6.20

- A blank entry means no block in the corresponding file is unique to the specified feature
- Only *a small percentage* of blocks are selected

Number of Blocks Common to Each Pair of Features

	F_1/F_2	F_1/F_3	F_1/F_4	F_1/F_5	F_2/F_3	F_2/F_4	F_2/F_5	F_3/F_4	F_3/F_5	F_4/F_5
analyze.c	65	64	64		64	64		64		16
multpath.c										
share.c	363	308	200	124	299	201	123	200	124	143
reachgraph.c										
pfqn.c										
mpfqn.c										
util.c	63	63	71	51	66	66	54	66	54	55
inspade.c										
indist.c						43	49			43
inshare.c	137	138	130	68	136	135	73	134	73	75
maketree.c										
results.c	211	181	95	183	186	95	180	95	180	94
cg.c	83	83			147					
in_qn_pn.c										
inchain.c	191	181	164		181	171		170		
bind.c	237	196	178	145	208	243	217	176	136	219
bitlib.c										
sor.c	144	154	144		144	144		145		
newcg.c		75								
phase.c	335	336	25		343	25		25		
newphase.c		399								
newlinear.c		260								
cexpo.c	246	237	181	142	246	181	161	181	139	103
expo.c	25	30	25	25	25	25	24	25	27	24
read1.c	238	237	229	169	242	234	170	233	169	162
symbol.c	172	155	173	151	248	261	236	243	221	233
ftree.c										
debug.c										
uniform.c		140								
mtta.c										
Total	2510	3237	1679	1058	2535	1888	1287	1757	1123	1167
Percentage	21.36	27.54	14.29	9.00	21.57	16.07	10.95	14.95	9.56	9.93

- The notation F_1/F_2 indicates code common to features F_1 and F_2
- A blank entry means no common block in the corresponding file

Code Common to All Five Features

	Number of blocks	Number of decisions	Number of c-uses	Number of p-uses
analyze.c				
multpath.c				
share.c	122	32	68	32
reachgraph.c				
pfqn.c				
mpfqn.c				
util.c	51	34	44	45
inspade.c				
indist.c				
inshare.c	64	29	11	17
maketree.c				
results.c	94	40	72	36
cg.c				
in_qn_pn.c				
inchain.c				
bind.c	123	72	160	56
bitlib.c				
sor.c				
newcg.c				
phase.c				
newphase.c				
newlinear.c				
cexpo.c	13	64	182	111
expo.c	24	15	37	18
read1.c	161	114	205	159
symbol.c	136	68	109	95
ftree.c				
debug.c				
uniform.c				
mtta.c				
Total	788	468	888	569
Percentage	6.71	6.60	4.05	3.76

- A blank entry means no common code in the corresponding file

Verification with SHARPE Experts

- The identified *files, functions, blocks, and decisions* are either unique to a feature as they should be or shared by a pair of features or common to all five features.
- *No complete verification was done with respect to the identified c-uses and p-uses*
  - Very difficult for humans to have a complete understanding of a complicated system at such a fine granularity
  - *Some* identified c-uses and p-uses are verified and agreed on by the experts
- *Need more objective verification*
  - Ask experts to highlight code segments they think are important to each feature
  - Different segments might be highlighted by different people
  - Need to summarize such divergent information

Conclusion (PART I)

- Code identified using the execution slice-based technique (either unique to a feature or common to a group of features) can be used as a good *starting point* for studying program features
  - C-uses and p-uses provide an in-depth understanding
- This technique may *not* find all relevant code that makes up a feature
- Apply to the Y2K problems
  - Identify *''date-sensitive''* code which may be only a few lines in a system consisting of millions of lines of code
- Extend to Program Debugging
  - ''Invoking tests'' correspond to the ''failed tests''
  - ''Excluding tests'' correspond to the ''successful tests''

PART II

Quantifying the Disparity, Concentration, and Dedication between Program Components and Features

W. Eric Wong, Swapna S. Gokhale, and Joseph R. Horgan, "Quantifying the closeness between program components and features," *Journal of Systems and Software*, Volume 54, Issue 2, pp. 87-98, October 2000

Click on the following pdf icon to download the paper

Adobe Acrobat Document

Objective

- Measure, *in a quantitative way*, the closeness between a feature and a program component
  - Example:
    - How much of the code in a program component is used to implement a given feature?
    - 50%, 70%, or more than 90%?
- Previous studies highlight code that is *uniquely* related to a given feature.
  - Can be used as good starting points for locating program features
  - Does not identify where the *majority* of the code related to a given feature resides
- Metrics proposed
  - Provide a much more complete picture of how a feature spreads over a software system

Three Metrics

- Disparity: measures how close a feature is to a program component
- Concentration: shows how much a feature is concentrated in a program component
- Dedication: indicates how much a program component is dedicated to a feature

Represent a Feature Using a Set of Blocks

- Represent an *abstract* feature in terms of some *concrete* program elements
- Use the *union of the execution slices of invoking inputs* to find a set of code that is used to implement a given feature
  - Theoretically, we may need to use *all* the invoking inputs for a given program and feature
  - In practice, this is *impossible* and *unnecessary*
  - An alternative is to use a good set of *focused invoking inputs* (which can be obtained from the regression test suite) to identify *most of the code* that is used to implement the feature

Notation

- Let P be a program, F a feature of P, C a component of P, and T a small set of carefully selected invoking inputs with the focus on F
- B_{t_i} is the set of blocks in P executed by input $t_i \in T$.
- B_F is the union of B_{t_i} such that $t_i \in T$, i.e., the set of blocks in P executed by at least one input in T. In other words, B_F is a set of blocks in P which are used to implement F.
- B_C is the set of blocks in C.
- $B_{C \bigcap F}$ is the intersection of B_C and B_F, i.e., the set of blocks in C which are used to implement F.
- $B_{C \bigcup F}$ is the union of B_C and B_F.
- $B_{C \bigoplus F}$ is the set of blocks in either B_C or B_F, but not both, i.e., $B_{C \bigoplus F}$ equals $(\overline{B_C} \bigcap B_F) \bigcup (B_C \bigcap \overline{B_F})$, where $\overline{B_C}$ and $\overline{B_F}$ are the complements of B_C and B_F in the set of blocks in P, respectively, $\overline{B_C} \bigcap B_F$ contains the blocks in B_F but not in B_C, and $B_C \bigcap \overline{B_F}$ contains the blocks in B_C but not in B_F.

Disparity Metric: $DISP_{CF}$

- $\mathrm{DISP}_{CF} = \dfrac{|B_{C \bigoplus F}|}{|B_{C \bigcup F}|} = 1 - \dfrac{|B_{C \bigcap F}|}{|B_{C \bigcup F}|}$
- $0 \le \mathrm{DISP}_{CF} \le 1$
- Inversely proportional to the number of blocks in $B_{C \cap F}$
  - When C and F share more common blocks, their disparity should be smaller
- Proportional to the number of blocks in B
  - The more blocks in either B_C or B_F, but not both, the larger the disparity between C and F
- $\mathrm{DISP}_{CF} = 1$ if and only if there is no common block between B_C and B_F
  - $B_{C \cap F} = \phi$
- $\mathrm{DISP}_{CF} = 0$ if and only if feature F is totally implemented in component C and every block in C is used to implement F
  - $B_{C \cap F} = B_{C \cup F}$ (i.e., $B_C = B_F$)

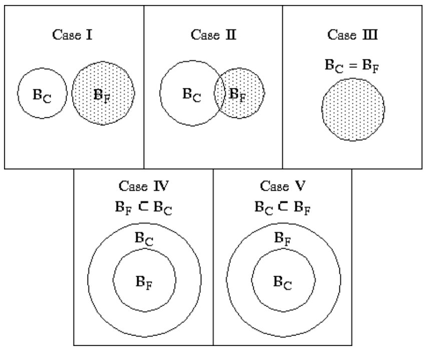
Concentration Metric: $CONC_{FC}$

- $\text{CONC}_{FC} = \frac{|B_{C \cap F}|}{|B_F|}$
- $0 \le \text{CONC}_{FC} \le 1$
- Inversely proportional to the number of blocks in B_F
  - When B_F has more blocks, it is less likely for all the blocks to reside in the same component
- Proportional to the number of blocks in $B_{C \cap F}$
  - When C and F have more blocks in common, they have a bigger commitment to each other
- $\text{CONC}_{FC} = 1$ if and only if all the blocks used to implement F are in C
- $\text{CONC}_{FC} = 0$ if and only if none of these blocks is in C

Dedication Metric: $DEDI_{CF}$

- $\text{DEDI}_{CF} = \frac{|B_{C \cap F}|}{|B_C|}$
- $0 \le \text{DEDI}_{CF} \le 1$
- Inversely proportional to the number of blocks in B_C
  - When B_C has more blocks, it is more likely that some of these blocks have nothing to do with F
- Proportional to the number of blocks in $B_{C \cap F}$
  - When C and F have more blocks in common, they have a bigger commitment to each other
- $\text{DEDI}_{CF} = 1$ if and only if all the blocks in C are used to implement F
- $\text{DEDI}_{CF} = 0$ if and only if none of these blocks has anything to do with F

Possible Relationship between B_F and B_C



- Case I: $B_{C \bigcap F} = \phi$:
  $DISP_{CF} = 1$, $CONC_{FC} = DEDI_{CF} = 0$
- Case II: B_C and B_F have some blocks in common (i.e., $B_{C \bigcap F} \neq \phi$, but neither subsumes the other):
  $0 < DISP_{CF}, CONC_{FC}$ and $DEDI_{CF} < 1$
- Case III: B_C equals B_F (i.e., $B_{C \bigcap F} = B_{C \bigcup F}$):
  $DISP_{CF} = 0$, $CONC_{FC} = DEDI_{CF} = 1$
- Case IV: B_F is a subset of B_C and $B_F \neq B_C$ (i.e., $B_F \subset B_C$):
  $0 < DISP_{CF}$ and $DEDI_{CF} < 1$, $CONC_{FC} = 1$
- Case V: B_C is a subset of B_F and $B_C \neq B_F$ (i.e., $B_C \subset B_F$):
  $0 < DISP_{CF}$ and $CONC_{FC} < 1$, $DEDI_{CF} = 1$

$DISP_{CF}$, $CONC_{FC}$ and $DEDI_{CF}$

- A 100% concentration or a 100% dedication *does not guarantee* a zero disparity between *F* and *C*
- Disparity is 0 if and only if both concentration and dedication are 100%, (i.e., $B_F = B_C$)
- Disparity is 1 if and only if both concentration and dedication are 0, (i.e., $B_{C \cap F} = \phi$)
- If $CONC_{FC} \neq 0$ (i.e., $B_{C \cap F} \neq \phi$) then $DEDI_{CF} \neq 0$, and vice versa
- If $CONC_{FC} = 0$ (i.e., $B_{C \cap F} = \phi$) then $DEDI_{CF} = 0$, and vice versa

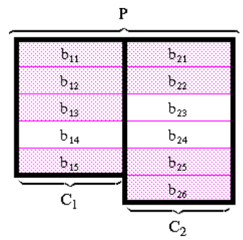
Example (1)

- 2 components C_1 and C_2
- 2 features F_1 and F_2
- 2 invoking inputs (t_1 and t_2) focused on F_1
- Assume

$$\begin{aligned} B_{F_1} &= B_{t_1} \bigcup B_{t_2} \\ &= \{b_{11}, b_{12}, b_{13}, b_{21}, b_{22}\} \bigcup \{b_{13}, b_{15}, b_{22}, b_{25}, b_{26}\} \\ &= \{b_{11}, b_{12}, b_{13}, b_{15}, b_{21}, b_{22}, b_{25}, b_{26}\} \end{aligned}$$

- We have

$$\begin{aligned} B_{C_1 \bigcap F_1} &= \{b_{11}, b_{12}, b_{13}, b_{15}\} \\ B_{C_2 \bigcap F_1} &= \{b_{21}, b_{22}, b_{25}, b_{26}\} \\ B_{C_1 \bigcup F_1} &= \{b_{11}, b_{12}, b_{13}, b_{14}, b_{15}, b_{21}, b_{22}, b_{25}, b_{26}\} \\ B_{C_2 \bigcup F_1} &= \{b_{11}, b_{12}, b_{13}, b_{15}, b_{21}, b_{22}, b_{23}, b_{24}, b_{25}, b_{26}\} \\ B_{C_1 \bigoplus F_1} &= \{b_{14}, b_{21}, b_{22}, b_{25}, b_{26}\} \\ B_{C_2 \bigoplus F_1} &= \{b_{11}, b_{12}, b_{13}, b_{15}, b_{23}, b_{24}\} \end{aligned}$$



Each cell in the diagram is a block. The pink ones are related to the feature F_1

Example (2)

- The disparity

$$\begin{aligned} DISP_{C_1 F_1} &= 1 - \frac{|B_{C_1 \bigcap F_1}|}{|B_{C_1 \bigcup F_1}|} = 1 - \frac{4}{9} = 0.556 \\ DISP_{C_2 F_1} &= 1 - \frac{|B_{C_2 \bigcap F_1}|}{|B_{C_2 \bigcup F_1}|} = 1 - \frac{4}{10} = 0.60 \end{aligned}$$

- The concentration

$$\begin{aligned} CONC_{F_1 C_1} &= \frac{|B_{C_1 \bigcap F_1}|}{|B_{F_1}|} = \frac{4}{8} = 0.50 \\ CONC_{F_1 C_2} &= \frac{|B_{C_2 \bigcap F_1}|}{|B_{F_1}|} = \frac{4}{8} = 0.50 \end{aligned}$$

- The dedication

$$\begin{aligned} DEDI_{C_1 F_1} &= \frac{|B_{C_1 \bigcap F_1}|}{|B_{C_1}|} = \frac{4}{5} = 0.80 \\ DEDI_{C_2 F_1} &= \frac{|B_{C_2 \bigcap F_1}|}{|B_{C_2}|} = \frac{4}{6} = 0.667 \end{aligned}$$

A Case Study on SHARPE

- First developed in 1986
- 35,412 lines of C code in 30 files
- 373 functions and 11752 blocks

File	# of blocks	File	# of blocks	File	# of blocks
analyze.c	334	indist.c	243	pfqn.c	325
bind.c	911	inshare.c	608	phase.c	544
bitlib.c	75	inspade.c	305	reachgraph.c	524
cexpo.c	406	maketree.c	176	read1.c	421
cg.c	202	mpfqn.c	429	results.c	604
debug.c	94	mtta.c	134	share.c	702
expo.c	186	multpath.c	128	sor.c	269
ftree.c	993	newcg.c	230	symbol.c	527
in_qn_pn.c	441	newlinear.c	489	uniform.c	227
inchain.c	404	newphase.c	414	util.c	407

- Six features
  - MC (Markov Chain)
  - FT (Fault Trees)
  - GSPN (Generalized Stochastic Petri-Nets)
  - PFQN (Product-Form Queuing Networks)
  - RELG (Reliability Graphs)
  - MRM (Markov Reward Models)

Data Collection

- Each *c* file as a separate program component
- A set of invoking inputs focused on each feature was carefully selected from *the regression test suite* of SHARPE
  - Advantages of using regression tests
    - They were real inputs used during the integration and system testing
    - There exist clear descriptions for many of these tests, which made it very easy to select invoking inputs *focused* on a given feature
- Execution slice in terms of blocks of each of these inputs was computed
- Computed also were B_F for each of the six features and B_C for each of the 30 files of SHARPE
- Using B_F's and B_C's, compute DEDI_{CF} , CONC_{FC} and DISP_{CF} with respect to every possible pair of F and C

Data Verification

- Need an oracle
  - Design documentation
    - Well-designed system: a clear mapping between each feature and its corresponding code segments
  - Experts who have a good knowledge of the system being analyzed
    - May not exist
    - Time consuming & not affordable
    - How to summarize divergent information
- Present each B_F to experts who are familiar with SHARPE
  - The identified blocks are used to implement the designated feature as they should be
  - No additional block that has to be added to each B_F

Feature Concentration in Components

File	MC	FT	GSPN	PFQN	RELG	MRM
analyze.c	0.0195	0.0101	0.0298	0.0282	0.0092	0.0590
bind.c	0.0728	0.1072	0.0779	0.1141	0.0894	0.1109
bitlib.c				0.0115	0.0087	
cexpo.c	0.0734	0.0663	0.0497	0.0615	0.0720	0.0965
cg.c	0.0245		0.0300			0.0459
debug.c						
expo.c	0.0088	0.0457	0.0089	0.0207	0.0326	0.0078
ftree.c		0.2875			0.2279	
in_qn_pn.c			0.0364	0.0258		
inchain.c	0.0548		0.0364	0.0601	0.0353	0.0625
indist.c		0.0150		0.0105	0.0532	0.0353
inshare.c	0.0427	0.0720	0.0358	0.0547	0.1045	0.0459
inspade.c		0.0073		0.0306	0.0353	
maketree.c		0.0117		0.0255	0.0289	
mpfqn.c				0.0798		
mtta.c						
multpath.c						
newcg.c	0.0197		0.0389			
newlinear.c	0.0760		0.0725			
newphase.c	0.1019		0.0765			
pfqn.c				0.0955		
phase.c	0.0916		0.0936	0.0085		0.1002
reachgraph.c			0.0797			
read1.c	0.0704	0.0679	0.0560	0.0812	0.0899	0.0756
results.c	0.0707	0.0821	0.0753	0.0323	0.0503	0.0812
share.c	0.1134	0.0639	0.0626	0.0842	0.0757	0.1221
sor.c	0.0460		0.0315	0.0493		0.0528
symbol.c	0.0516	0.0954	0.0525	0.0917	0.0472	0.0837
uniform.c	0.0407		0.0371			
util.c	0.0215	0.0679	0.0189	0.0343	0.0399	0.0206
Sum	1.00	1.00	1.00	1.00	1.00	1.00

Blank entry means the corresponding feature concentration is zero.

Component Dedication to Features

File	MC	FT	GSPN	PFQN	RELG	MRM	Sum
analyze.c	0.1976	0.0749	0.4431	0.2485	0.0599	0.5659	1.5899
bind.c	0.2711	0.2909	0.4248	0.3688	0.2141	0.3897	1.9594
bitlib.c				0.4533	0.2533		0.7066
cexpo.c	0.6133	0.4039	0.6084	0.4458	0.3867	0.7611	3.2191
cg.c	0.4109		0.7376			0.7277	1.8762
debug.c							0.00
expo.c	0.1613	0.6075	0.2366	0.3280	0.3817	0.1344	1.8495
ftree.c		0.7160			0.5005		1.2165
in_qn_pn.c			0.4104	0.1723			0.5827
inchain.c	0.4604		0.4480	0.4381	0.1906	0.4951	2.0322
indist.c		0.1523		0.1276	0.4774	0.4650	1.2223
inshare.c	0.2385	0.2928	0.2928	0.2648	0.3750	0.2418	1.7057
inspade.c		0.0590		0.2951	0.2525		0.6066
maketree.c		0.1648		0.4261	0.3580		0.9489
mpfqn.c				0.5478			0.5478
mtta.c							0.00
multpath.c							0.00
newcg.c	0.2913		0.8391				1.1304
newlinear.c	0.5276		0.7362				1.2638
newphase.c	0.8357		0.9179				1.7536
pfqn.c				0.8646			0.8646
phase.c	0.5717		0.8548	0.0460		0.5901	2.0626
reachgraph.c			0.7557				0.7557
read1.c	0.5677	0.3991	0.6603	0.5677	0.4656	0.5748	3.2352
results.c	0.3974	0.3361	0.6192	0.1573	0.1821	0.4305	2.1226
share.c	0.5484	0.2251	0.4430	0.3533	0.2350	0.5570	2.3618
sor.c	0.5799		0.5799	0.5390		0.6283	2.3271
symbol.c	0.3321	0.4478	0.4934	0.5123	0.1954	0.5085	2.4895
uniform.c	0.6079		0.8106				1.4185
util.c	0.1794	0.4128	0.2310	0.2482	0.2138	0.1622	1.4474

Blank entry means the corresponding component dedication is zero.

Disparity between Features and Components

File	MC	FT	GSPN	PFQN	RELG	MRM
analyze.c	0.9816	0.9909	0.9704	0.9733	0.9919	0.9402
bind.c	0.9352	0.9071	0.9242	0.8944	0.9278	0.8957
bitlib.c	1	1	1	0.9885	0.9914	1
cexpo.c	0.9246	0.9357	0.9494	0.9394	0.9309	0.8967
cg.c	0.9758	1	0.9694	1	1	0.9527
debug.c	1	1	1	1	1	1
expo.c	0.9915	0.9536	0.9913	0.9797	0.9681	0.9925
ftree.c	1	0.6522	1	1	0.7720	1
in_qn_pn.c	1	1	0.9641	0.9765	1	1
inchain.c	0.9457	1	0.9639	0.9409	0.9683	0.9376
indist.c	1	0.9860	1	0.9901	0.9471	0.9649
inshare.c	0.9609	0.9347	0.9659	0.9502	0.9023	0.9582
inspade.c	1	0.9934	1	0.9707	0.9670	1
maketree.c	1	0.9888	1	0.9747	0.9718	1
mpfqn.c	1	1	1	0.9190	1	1
mtta.c	1	1	1	1	1	1
multpath.c	1	1	1	1	1	1
newcg.c	0.9808	1	0.9599	1	1	1
newlinear.c	0.9234	1	0.9240	1	1	1
newphase.c	0.8890	1	0.9177	1	1	1
pfqn.c	1	1	1	0.8962	1	1
phase.c	0.9062	1	0.8985	0.9927	1	0.8966
reachgraph.c	1	1	0.9157	1	1	1
read1.c	0.9284	0.9343	0.9425	0.9172	0.9113	0.9229
results.c	0.9318	0.9240	0.9224	0.9717	0.9571	0.9209
share.c	0.8842	0.9447	0.9384	0.9210	0.9354	0.8748
sor.c	0.9534	1	0.9683	0.9504	1	0.9461
symbol.c	0.9510	0.9066	0.9477	0.9079	0.9588	0.9161
uniform.c	0.9587	1	0.9619	1	1	1
util.c	0.9800	0.9340	0.9819	0.9679	0.9640	0.9810

Conclusion (PART II) (1)

- SHARPE has *a very delocalized structure* with features spread over many files
  - For a given feature, in general, it has less than 8% concentration in a file
  - For a given file, if it is used to implement a feature, it normally has at least 20% of its blocks dedicated to this feature
  - Why?
    - Features are incrementally incorporated into SHARPE, rather than planned in the original design

Conclusion (PART II) (2)

- Three metrics are proposed (disparity, concentration and dedication) to provide *a good quantitative measure* of the closeness between a feature and a program component
- Help programmers capture *more precisely* where each feature resides in the system
  - A quantitative measure computed based on carefully defined metrics versus a *qualitative* understanding obtained from intuitive feeling
- Our metrics are complementary to other program comprehension techniques to help software programmers better understand the system at hand

PART III

Measuring Distance between Program Features

W. E. Wong and S. Gokhale, "Static and Dynamic Distance Metrics for Feature-Based Code Analysis," *Journal of Systems and Software*, Volume 74, Number 3, pp. 283-295, February 2005

Click on the following pdf icon to download the paper

Adobe Acrobat Document

Objective

- Develop a metric to determine the distance between features
  - How are features of a system close to each other *in a quantitative way*?
    - Is the distance between features α and β larger than that between β and γ?
    - If so, how much larger?
  - Provide a good start to understanding how a modification made to one feature is likely to affect other features

Our Approach

- Represent an *abstract* feature in terms of some *concrete* program elements
  - Use an execution slice-based technique to identify a set of code (basic blocks in our case) that is used to implement each feature
    - A basic block (also known as a block) is a sequence of consecutive code containing no branch such that if part of the code is executed, other code will also be executed
  - An input is an *invoking input* with respect to a feature if, when executed on the program, it shows the functionality of that feature.
- Compute distance between two features
  - Static distance
    - in terms of code that is used to *implement these features*
  - Dynamic distance
    - in terms of code that is *executed by inputs which exhibit these features*

Notation

- P is a program; α, β and γ are features of P.
- T_α is a small set of carefully selected invoking inputs with the focus on α.
- $B_{t_{\alpha i}}$ is the set of blocks in P executed by input $t_{\alpha i} \in T_\alpha$. Depending on whether the execution count of each block is considered, an appropriate weight factor may have to be assigned to it.
- B_α which equals the union of $B_{t_{\alpha i}}$ for all $t_{\alpha i} \in T_\alpha$ is the set of blocks in P which are used to implement α. A similar definition applies to B_β and B_γ.
- $B_{\alpha \bigcap \beta} = B_\alpha \bigcap B_\beta$ is the set of blocks shared by features α and β.
- $B_{\alpha \bigcup \beta}$ is the set of blocks in the union of B_α and B_β (i.e., $B_\alpha \bigcup B_\beta$).
- $B_{\alpha \bigoplus \beta}$ is the set of blocks in either B_α or B_β, but not both,[3] i.e., $B_{\alpha \bigoplus \beta}$ equals $(\overline{B_\alpha} \bigcap B_\beta) \bigcup (B_\alpha \bigcap \overline{B_\beta})$, where $\overline{B_\alpha}$ and $\overline{B_\beta}$ are the complements of B_α and B_β in the set of blocks in P, respectively, $\overline{B_\alpha} \bigcap B_\beta$ contains the blocks in B_β but not in B_α, and $B_\alpha \bigcap \overline{B_\beta}$ contains the blocks in B_α but not in B_β.
- $DIST_{\alpha\beta}$ is the distance between features α and β.

Properties

- The numerical value of $DIST_{\alpha\beta}$ must be normalized between 0 and 1 (i.e., $0 \leq DIST_{\alpha\beta} \leq 1$) so that the distance between two features can be compared in a meaningful way.
- The value assigned should be a *monotonically decreasing* function of the number of blocks in $B_{\alpha \bigcap \beta}$, i.e., the more blocks in the intersection of B_α and B_β, the smaller the distance between α and β. This makes sense because when α and β share more common blocks, their distance should be smaller.
- The value assigned should be a *monotonically increasing* function of the number of blocks in $B_{\alpha \bigoplus \beta}$, i.e., the more blocks in either B_α or B_β, but not both, the larger the distance between α and β. This implies that when there are more blocks in B_α but not in B_β, or vice versa, the distance between these two should also be larger.
- The value 1 should be assigned if and only if there is no common block between B_α and B_β, i.e., the intersection between B_α and B_β is empty.
- The value 0 should be assigned if and only if features α and β use exactly the same set of blocks, i.e., every block in B_α is in B_β, and vice versa.

[3] Notation $\bigoplus$ represents the *exclusive or* relation between two sets.

Distance Metric

$$DIST_{\alpha\beta} = \frac{\left|B_{\alpha \bigoplus \beta}\right|}{\left|B_{\alpha \bigcup \beta}\right|}$$

This leads to the computation

$$\begin{aligned} &= \frac{|B_\alpha| + |B_\beta| - 2 * \left|B_{\alpha \bigcap \beta}\right|}{|B_\alpha| + |B_\beta| - \left|B_{\alpha \bigcap \beta}\right|} \\ &= 1 - \frac{\left|B_{\alpha \bigcap \beta}\right|}{|B_\alpha| + |B_\beta| - \left|B_{\alpha \bigcap \beta}\right|} \\ &= 1 - \frac{\left|B_{\alpha \bigcap \beta}\right|}{\left|B_{\alpha \bigcup \beta}\right|} \end{aligned}$$

where $|B_\alpha|$ represents the number of elements in set B_α, and so on.

Three Axioms

- $DIST_{\alpha\alpha} = 0$
- $DIST_{\alpha\beta} = DIST_{\beta\alpha}$
- $DIST_{\alpha\beta} + DIST_{\beta\gamma} >= DIST_{\alpha\gamma}$

Observation

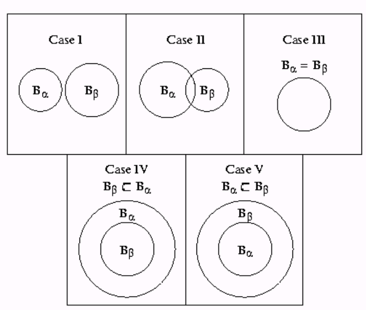


Figure 1: Possible relationship between B_α and B_β.

- Case I: $B_{\alpha\bigcap\beta} = \phi$. In this case, $DIST_{\alpha\beta} = 1$.
- Case II: B_α and B_β have some blocks in common, i.e., $B_{\alpha\bigcap\beta} \neq \phi$, but neither subsumes the other. Here, $DIST_{\alpha\beta}$, is between 0 and 1.
- Case III: B_α equals B_β which makes $B_{\alpha\bigcap\beta} = B_{\alpha\bigcup\beta}$. As a result, $DIST_{\alpha\beta} = 0$.
- Case IV: B_β is a subset of B_α (i.e., $B_\beta \subset B_\alpha$) but $B_\beta \neq B_\alpha$, $DIST_{\alpha\beta}$ is between 0 and 1.
- Case V: B_α is a subset of B_β (i.e., $B_\alpha \subset B_\beta$) but $B_\alpha \neq B_\beta$, $DIST_{\alpha\beta}$ is also between 0 and 1.

Static Distance versus Dynamic Distance (1)

- Depending on whether the *execution frequency* of each block is considered during the construction of the sets of code
  - Static distance
    - Only depends on how features are implemented in the system
    - The execution frequency of each block is *not used* in computing the distance
  - Dynamic distance
    - Depends on how each feature is implemented
    - Also takes into account *how each feature is executed based on a user's operational profile*
- Static distance gives the closeness of two features from the *system implementation* point of view, whereas the dynamic distance presents such closeness from a *user's execution* point of view

Static Distance versus Dynamic Distance (2)

- Static distance between two features is fixed once their implementation is completed, but the dynamic distance changes depending on how these features are executed
- The dynamic distance computed *using one user's operational profile can be different from that using another profile* even though the corresponding static distance stays the same

Static Distance versus Dynamic Distance (3)

- If the static distance between two features is unity (i.e., no overlap in the implementation), the dynamic distance between these two features must also be unity

static distance = 1 ➔ dynamic distance = 1 (YES)

  - *Is the reverse true?*
    - Consider "error handling code" used for two features but not executed
      - Dynamic distance equals one but not the static distance
      - In general, do NOT expect this to happen

dynamic distance = 1 ➔ static distance = 1 (not necessarily)

Static Distance versus Dynamic Distance (4)

- A zero static distance between two features (i.e., the same set of code is used to implement both features), does not necessarily give a zero dynamic distance
  - Different blocks may have different execution counts

static distance = 0 ➔ dynamic distance = 0 (not necessarily)

- A zero dynamic distance between two features does not imply a zero static distance
  - Different error handling routines are implemented for these two features. The dynamic distance can be zero if none of their invoking inputs trigger the errors.
  - Nevertheless, after the error routines are taken into account, the static distance is greater than zero.

dynamic distance = 0 ➔ static distance = 0 (not necessarily)

Static Distance versus Dynamic Distance (5)

- For two given features, if their static distance is between zero and one, their dynamic distance, in general, is also between zero and one, and vice versa
- A smaller static distance does not imply a smaller dynamic distance, and vice versa
- A larger static distance does not imply a larger dynamic distance, and vice versa

A Case Study on SHARPE (1)

- A Symbolic Hierarchical Automated Reliability and Performance Evaluator
  - 35,412 lines of C code in 30 files
  - 373 functions
  - Examined 5 features
    - Fault Trees (FT) (f_1)
    - Markov Chains (MC) (f_2)
    - Generalized Stochastic Petri-Nets (GSPN) (f_3)
    - Product-Form Queuing Networks (PFQN) (f_4)
    - Reliability Graph (RELG) (f_5)

Static Distance between Each Pair of Features in SHARPE

File	$f_1 f_2$†	$f_1 f_3$	$f_1 f_4$	$f_1 f_5$	$f_2 f_3$	$f_2 f_4$	$f_2 f_5$	$f_3 f_4$	$f_3 f_5$	$f_4 f_5$
analyze.c	1	1	0.429	0.269	1	1	1	1	1	0.227
bind.c	0	0	0	0.185	0	0	0.185	0	0.185	0.185
bitlib.c	1	1	1	1	1	1	1	1	1	1
cexpo.c	0.286	0.524	1	0.364	0.333	1	0.533	1	0.682	1
cg.c	1	1	1	1	0.143	1	1	1	1	1
debug.c	1	1	1	1	1	1	1	1	1	1
expo.c	0	0	0	0	0	0	0	0	0	0
ftree.c	1	1	1	0.609	1	1	1	1	1	1
in_qn_pn.c	1	1	1	1	1	1	1	1	1	1
inchain.c	1	1	1	1	0.111	0.158	0.581	0.059	0.536	0.509
indist.c	1	1	0.122	0.093	1	1	1	1	1	0.204
inshare.c	0.368	0.480	0.214	0.324	0.381	0.186	0.221	0.354	0.465	0.232
inspade.c	1	1	1	1	1	1	1	1	1	0
maketree.c	1	1	1	1	1	1	1	1	1	0.408
mpfqn.c	1	1	1	1	1	1	1	1	1	1
mtta.c	1	1	1	1	1	1	1	1	1	1
multpath.c	1	1	1	1	1	1	1	1	1	1
newcg.c	1	1	1	1	0.034	1	1	1	1	1
newlinear.c	1	1	1	1	1	1	1	1	1	1
newphase.c	1	1	1	1	0.028	1	1	1	1	1
pfqn.c	1	1	1	1	1	1	1	1	1	1
phase.c	1	1	1	1	0.333	1	1	1	1	1
reachgraph.c	1	1	1	1	1	1	1	1	1	1
read1.c	0	0.059	0	0.059	0.059	0	0.059	0.059	0	0.059
results.c	0.103	0.111	0.111	0	0.143	0.143	0.103	0	0.111	0.111
share.c	0	0	0	0	0	0	0	0	0	0
sor.c	1	1	1	1	0	0	1	0	1	1
symbol.c	0	0	0	0.125	0	0	0.125	0	0.125	0.125
uniform.c	1	1	1	1	0	1	1	1	1	1
util.c	1	1	1	1	1	1	1	1	1	1

†Notation $f_1 f_2$ represents the static distance between f_1 and f_2 ($DIST^S_{f_1 f_2}$), and so on.

Static Distance between Each Pair of Features in SHARPE

File	$f_1 f_2$†	$f_1 f_3$	$f_1 f_4$	$f_1 f_5$	$f_2 f_3$	$f_2 f_4$	$f_2 f_5$	$f_3 f_4$	$f_3 f_5$	$f_4 f_5$
analyze.c	1	1	0.938	0.939	1	1	1	1	1	0.242
bind.c	0.931	0.786	0.643	0.791	0.678	0.808	0.986	0.402	0.955	0.925
bitlib.c	1	1	1	1	1	1	1	1	1	1
cexpo.c	0.433	0.844	1	0.999	0.724	1	0.999	1	0.999	1
cg.c	1	1	1	1	0.471	1	1	1	1	1
debug.c	1	1	1	1	1	1	1	1	1	1
expo.c	0.063	0.191	0.996	0.821	0.137	0.996	0.810	0.995	0.779	0.977
ftree.c	1	1	1	0.613	1	1	1	1	1	1
in_qn_pn.c	1	1	1	1	1	1	1	1	1	1
inchain.c	1	1	1	1	0.964	0.916	0.862	0.727	0.920	0.718
indist.c	1	1	0.990	0.510	1	1	1	1	1	0.979
inshare.c	0.449	0.294	0.326	0.665	0.248	0.579	0.799	0.477	0.749	0.529
inspade.c	1	1	1	1	1	1	1	1	1	0.667
maketree.c	1	1	1	1	1	1	1	1	1	0.858
mpfqn.c	1	1	1	1	1	1	1	1	1	1
mtta.c	1	1	1	1	1	1	1	1	1	1
multpath.c	1	1	1	1	1	1	1	1	1	1
newcg.c	1	1	1	1	0.918	1	1	1	1	1
newlinear.c	1	1	1	1	1	1	1	1	1	1
newphase.c	1	1	1	1	0.941	1	1	1	1	1
pfqn.c	1	1	1	1	1	1	1	1	1	1
phase.c	1	1	1	1	0.641	1	1	1	1	1
reachgraph.c	1	1	1	1	1	1	1	1	1	1
read1.c	0.140	0.852	0.932	0.915	0.833	0.940	0.925	0.990	0.987	0.297
results.c	0.377	0.852	0.941	0.789	0.907	0.963	0.872	0.600	0.314	0.726
share.c	0.070	0.172	0.275	0.625	0.110	0.326	0.651	0.400	0.690	0.483
sor.c	1	1	1	1	0.881	0.927	1	0.521	1	1
symbol.c	0.613	0.725	0.804	0.554	0.888	0.921	0.827	0.292	0.404	0.573
uniform.c	1	1	1	1	0.931	1	1	1	1	1
util.c	1	1	1	1	1	1	1	1	1	1

†Notation $f_1 f_2$ represents the dynamic distance between f_1 and f_2 ($DIST^D_{f_1 f_2}$), and so on.

A Case Study on SHARPE (2)

- SHARPE has *a very delocalized structure* with features spread over many files
  - Confirmed by those who are very familiar with SHARPE
    - Features are incrementally incorporated into SHARPE, rather than planned in the original design
- Provide quantitative measurements (both static and dynamic) to help programmers accurately capture how far two features are from each other rather than rely on some intuitive feel for the system which provides only a qualitative description of whether two features are close to each other.

Conclusion (PART III)

- Our metrics can be used to measure the distance between two features
- The distance measurement can serve as *a good starting point* to understanding how a modification made to one feature is likely to affect other features
  - $DIST_{\alpha\beta} << DIST_{\alpha\gamma}$ ➔ modifications to α can very possibly have a higher impact on β than on γ
- Allow programmers to understand the possibility for interactions between features *from a different perspective*
- Our next step is to investigate how our metrics complement other methodologies for better detecting of interactions between program features

Overall Conclusion

- A set of techniques are developed to help programmers *understand their code in a more cost effective way*