

# Quantifying the Closeness between Program Components and Features

W. Eric Wong, Swapna S. Gokhale, Joseph R. Horgan  
Applied Research  
Telcordia Technologies\*  
Morristown, NJ 07960  
{ewong,swapna,jrh}@research.telcordia.com

## Abstract

One of the most important steps towards effective software maintenance of a large complicated system is to understand how program features are spread over the entire system and their interactions with the program components. However, we must first be able to represent an *abstract* feature in terms of some *concrete* program components. In this paper, we use an execution slice-based technique to identify the basic blocks which are used to implement a program feature. Three metrics are then defined, based on this identification, to determine *quantitatively*, the disparity between a program component and a feature, the concentration of a feature in a program component, and the dedication of a program component to a feature. The computations of these metrics are automated by incorporating them in a tool ( $\chi$ Suds), which makes the use of our metrics immediately applicable in real-life contexts. We demonstrate the effectiveness of our technique by experimenting with a reliability and performance evaluator. Results of our study suggest that these metrics can provide an indication of the *closeness* between a feature and a program component which is very useful for software programmers and maintainers to better understand the system at hand.

**Keywords:** program comprehension, program features, execution slice, invoking input, excluding input, feature concentration, component dedication, disparity between feature and component.

## 1 Introduction

An ideal software system<sup>1</sup> should follow certain standards to have a high degree of cohesion and a low degree of coupling among its various components [8, 9]. It should also have a clear mapping between each feature and its corresponding code segments [5, 7]. However, this is seldom the case in practice and one may actually find that in a large complicated software system a program feature is spread across a number of components. Code used to implement a feature may be found in components which are seemingly unrelated to each other. As a result, there is no clear traceability for one to follow in order to understand *quantitatively* how features and program components interact. This is especially true for some legacy systems as many fast patches may have been included to keep up with pressure to respond to customer feedback in a timely fashion.

---

\*Telcordia Technologies was formerly known as Bell Communications Research or, simply, Bellcore.

<sup>1</sup>We use the words “system,” “program” and “application” interchangeably.

One can certainly argue that an experienced programmer may have good knowledge of the software system at hand. Nevertheless such knowledge is only qualitative. Carefully defined metrics are necessary to obtain a quantitative measure about the interactions between program components and features. For example, a programmer may realize that a feature is related to a program component. But, how much is it related? How much of the code in the component is used to implement this feature – 50, 70, or more than 90%? And how much of the code related to this feature is in this component? All these important questions cannot be answered in a quantitative way unless we clearly define how to compute the *program dedication* and the *feature concentration*.

Since a feature is an *abstract* description of a functionality given in the specification, and a program component is a *concrete* element which constitutes a software system, it is difficult to make a connection between these two without first representing an *abstract* feature in terms of some *concrete* program components. In this paper, we use an execution sliced-based technique, where an execution slice is the set of program code (basic blocks<sup>2</sup> in our case) executed by an input that exercises a feature, to locate code that is used to implement the feature. A very important aspect of our technique is that it is supported by a suite of tools and can be immediately applied to large complicated real applications. More details about feature representation and tool support can be found in Sections 2.4 and 4.2, respectively.

Once a feature is mapped into a set of basic blocks, we can then examine quantitatively the interaction between a feature and a program component. To do so, we compute three metrics:

- the *disparity*, which measures how close a feature is to a program component,
- the *concentration*, which shows how much a feature is concentrated in a program component, and
- the *dedication*, which indicates how much a program component is dedicated to a feature.

These metrics provide a significantly different view of where a program feature resides in a software system than presented in [13, 14, 15] which focus on the code that is *uniquely* related to a given feature. In other words, methodologies discussed in the latter only pinpoint some *starting points* for locating program features, whereas metrics proposed in this paper provide *a much more complete picture* of how a feature spreads over a software system. Although it is important to follow an *as-needed* program understanding strategy to locate, as quickly as possible, certain code segments as the starting points, it is also crucial to understand where the majority of the code related to a given feature resides. The key issue we want to answer is how to measure, in a quantitative way, the closeness between a feature and a program component in a large complicated software system. It is clear that information obtained by using metrics discussed in this paper should complement, rather than compete with, that collected using

---

<sup>2</sup>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.

the heuristics described in [13, 14, 15]. Together, they can help software programmers and maintainers better understand the system at hand.

The remainder of the paper is organized as follows. Section 2 presents general concepts, such as how to represent a feature using a set of basic blocks. It also defines the three metrics: disparity, concentration, and dedication. Section 3 gives a numerical example to show how these metrics are computed. Then, a case study is reported in Section 4 to illustrate the advantages of using our metrics. The effectiveness of our metrics is also discussed. Our conclusions appear in Section 5.

## 2 General Concepts and Definitions

In this section we first explain program features, program components, invoking inputs, and excluding inputs; then we discuss how to represent a feature in terms of a set of basic blocks followed by how to compute the disparity between a feature and a program component, the concentration of a feature in a program component, and the dedication of a program component to a feature. For the purpose of discussion, let  $P$  be a program,  $F$  a feature of  $P$ , and  $C$  a component of  $P$ .

### 2.1 Program features

Generally speaking, a feature is an abstract description of a functionality given in the specification. Therefore, one good way to describe the features of a given program is based on its specification. For example, the specification of the UNIX wordcount program ( $wc$ ) is to count the number of lines, words, and/or characters given to it as input. Based on this, we can specify three features (with respect to three functionalities): one which returns the number of lines, another which returns the number of words, and a third which returns the number of characters.

### 2.2 Program components

A program component can have many different meanings, depending on the system being analyzed. In our case study (see Section 4), we treat each *file* as a program component. However, metrics discussed in this paper can also work at a different granularity level (e.g., a single function, a group of functions, or a group of files) as long as information is collected with respect to that particular granularity.

### 2.3 Invoking inputs and excluding inputs

An input  $t$  is an *invoking* input with respect to a feature  $F$  if, when executed on  $P$ , it shows the functionality of  $F$ . Otherwise,  $t$  is an *excluding* input. Let us use the UNIX wordcount program again to illustrate the concept of invoking and excluding inputs. Suppose  $F$  is the functionality for counting the number of lines. An input (say  $t_1$ ) such as “`wc -l data`” that returns the number of lines in the file

*data* is an invoking input; whereas another input (say  $t_2$ ) “wc -w *data*” that gives the number of words (instead of the number of lines) in the file *data* is an excluding input.

An invoking input is said to be *totally focused* on a given feature if it exhibits only this feature and no other features. Hereafter, such an invoking input is also referred to as a *focused* invoking input with respect to that feature. If we limit ourselves to *wc* features which output line, word, and character counts,  $t_1$  is a focused invoking input with respect to  $F$  which counts the number of lines. But this is not true for an input “wc *data*” (referred to as  $t_3$ ) even though it also returns the number of lines in the file *data*. This is because in addition to the number of lines,  $t_3$  also returns the number of words and characters. That is,  $t_3$  also exhibits features which count the number of words and characters, respectively.

## 2.4 Representation of a feature in terms of basic blocks

Before we can compute the disparity between  $F$  and  $C$ , or the concentration of  $F$  in  $C$ , or the dedication of  $C$  to  $F$ , we must first represent  $F$  (an *abstract* feature) in terms of  $C$  (some *concrete* program elements). One choice is to use *basic blocks*. A description of a basic block appears in the first footnote in the introduction. Hereafter, we refer to a basic block simply as a block.

Researchers have proposed several execution slice-based heuristics to identify code that is *uniquely* related to a given feature [13, 14, 15]. Although code so identified provides an excellent starting point for program understanding, it is not sufficient for computing the disparity, the feature concentration, and the program dedication. For these computations, we need to identify *more* code that is relevant or important from the point of view of a functionality. One simple and yet effective approach is to use the union of the execution slices (in terms of blocks) of *many* invoking inputs to find a set of code (blocks in our case) that is used to implement  $F$ . Theoretically, we may need to use *all* the invoking inputs for a given  $P$  and  $F$ . In practice, this is impossible and unnecessary. The alternative is to use a good set of focused invoking inputs to identify most of the code that is used to implement  $F$ .

The reason for using focused invoking inputs with respect to the feature being examined is to avoid including code that has nothing to do with this feature. If this is not possible (i.e., every invoking input with respect to this feature also exhibits some other features), we need to subtract code that is uniquely used to implement the other features from the code identified by the union of such invoking inputs. A simple example explanation is as follows. Suppose a feature (say  $F_\alpha$ ) cannot be exhibited without also having another feature  $F_\beta$  exhibited. Suppose also that  $F_\beta$  can be exhibited by itself. Under this situation, a good way to find code used to implement  $F_\alpha$  is to first find code used to implement  $F_\alpha$  and  $F_\beta$ , then subtract the code uniquely related to  $F_\beta$  from that.

## 2.5 Disparity, Concentration, and Dedication

Let  $T$  be a small set of carefully selected invoking inputs with the focus on  $F$ . Recall that  $P$  is a program,  $F$  a feature of  $P$ , and  $C$  a component of  $P$ . We define the following notation:

- $B_{t_i}$  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  $C$ .
- $B_C \cap F$  is the intersection of  $B_C$  and  $B_F$ , i.e., the set of blocks in  $C$  which are used to implement  $F$ .
- $B_C \cup F$  is the union of  $B_C$  and  $B_F$ .
- $B_C \oplus F$  is the set of blocks in either  $B_C$  or  $B_F$ , but not both, i.e.,  $B_C \oplus F$  equals  $(\overline{B_C} \cap B_F) \cup (B_C \cap \overline{B_F})$ , 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,  $\overline{B_C} \cap B_F$  contains the blocks in  $B_F$  but not in  $B_C$ , and  $B_C \cap \overline{B_F}$  contains the blocks in  $B_C$  but not in  $B_F$ .

We now explain how to compute  $DISP_{CF}$ , the disparity between a feature  $F$  and a component  $C$ . Quantitatively, we want to capture by  $DISP_{CF}$  the degree to which feature  $F$  is close to component  $C$ . We observe that the measurement  $DISP_{CF}$  has to satisfy the following properties:

- The numerical value of  $DISP_{CF}$  must be normalized (i.e.,  $0 \leq DISP_{CF} \leq 1$ ) so that the disparities between features and components can be compared in a meaningful way.
- The value assigned should be somewhat *inversely proportional* to the number of blocks in  $B_C \cap F$ , i.e., the more blocks in the intersection of  $B_C$  and  $B_F$ , the smaller the disparity between  $C$  and  $F$ . This makes sense because when  $C$  and  $F$  share more common blocks, their disparity should definitely be smaller.
- The value assigned should be somewhat *proportional* to the number of blocks in  $B_C \oplus F$ , i.e., the more blocks in either  $B_C$  or  $B_F$ , but not both, the larger the disparity between  $C$  and  $F$ . This can also be easily understood because when there are more blocks in  $B_C$  but not in  $B_F$ , or vice versa, the disparity between these two should also be larger.
- The value 1 should be assigned if and only if there is no common block between  $B_F$  and  $B_C$ , i.e., the intersection between  $B_F$  and  $B_C$  is empty ( $B_C \cap F = \phi$ ).
- The value 0 should be assigned if and only if feature  $F$  is totally implemented in component  $C$  and every block in  $C$  is used to implemented  $F$ , i.e.,  $B_C \cap F = B_C \cup F$ . In other words,  $DISP_{CF} = 0$  if and only if  $B_C = B_F$ .

Based on these properties,  $DISP_{CF}$  can be defined as:

$$DISP_{CF} = \frac{|B_C \oplus F|}{|B_C \cup F|} \quad (1)$$

This leads to the computation

$$\begin{aligned} &= \frac{|B_C| + |B_F| - 2 * |B_C \cap F|}{|B_C| + |B_F| - |B_C \cap F|} \\ &= 1 - \frac{|B_C \cap F|}{|B_C| + |B_F| - |B_C \cap F|} \\ &= 1 - \frac{|B_C \cap F|}{|B_C \cup F|} \end{aligned} \quad (2)$$

where  $|B_C|$  represents the number of elements in set  $B_C$ , and so on.

Next, we explain how to compute  $CONC_{FC}$ , the concentration of a feature  $F$  in a component  $C$ , and  $DEDI_{CF}$ , the dedication of a component  $C$  to a feature  $F$ . These two measurements should *quantitatively* reflect how much of a feature is in a component, and how much a component is in a feature, respectively. They should also satisfy the following properties:

- The values of both  $CONC_{FC}$  and  $DEDI_{CF}$  should be normalized in the range from 0 to 1, inclusively.
- The value assigned should be somewhat *proportional* to the number of blocks in  $B_C \cap F$ , i.e., the more blocks in the intersection of  $B_C$  and  $B_F$ , the higher the concentration of  $F$  in  $C$ , and the higher the dedication of  $C$  to  $F$ . The reason is intuitive and simple: when  $C$  and  $F$  have more blocks in common, they have a bigger commitment to each other.
- The value assigned should be somewhat *inversely proportional* to the number of blocks in  $B_F$  for  $CONC_{FC}$ , and  $B_C$  for  $DEDI_{CF}$ , respectively. This is because when  $B_F$  have more blocks, it is less likely for all the blocks to reside in the same component. Similarly, when  $B_C$  have more blocks, it is more likely that some of these blocks have nothing to do with  $F$ .
- $CONC_{FC}$  equals 1 if and only if all the blocks used to implement  $F$  are in  $C$ , whereas  $CONC_{FC}$  equals 0 if and only if none of these blocks is in  $C$ .
- $DEDI_{CF}$  equals 1 if and only if all the blocks in  $C$  are important to  $F$ , whereas  $DEDI_{CF}$  equals 0 if and only if none of these blocks has anything to with  $F$ .

Based on these observations,  $CONC_{FC}$  and  $DEDI_{CF}$  can be defined as:

$$CONC_{FC} = \frac{|B_C \cap F|}{|B_F|} \quad (3)$$

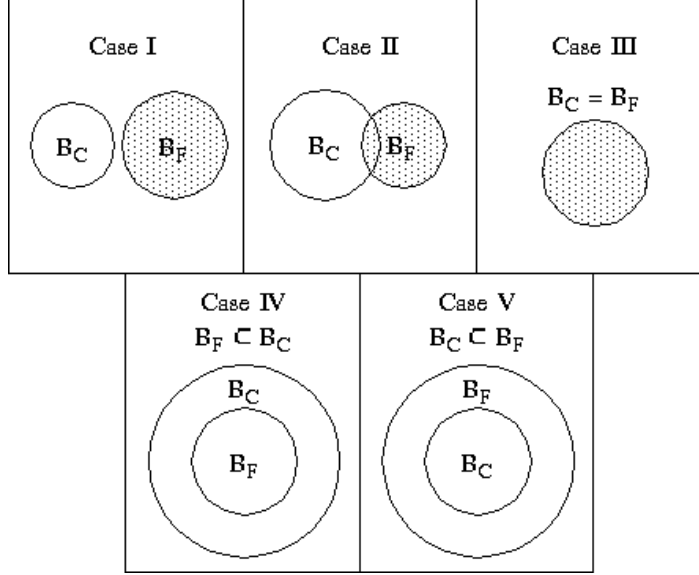


Figure 1: Possible relationship between  $B_F$  and  $B_C$ .

$$DEDI_{CF} = \frac{|B_C \cap F|}{|B_C|} \quad (4)$$

Finally, we examine the relationship among  $DISP_{CF}$ ,  $CONC_{FC}$ , and  $DEDI_{CF}$  for the five cases in Figure 1 which cover all the possible relationships between the sets  $B_F$  and  $B_C$ .

- Case I:  $B_C \cap F = \phi$ . In this case,  $DISP_{CF} = 1$ ,  $CONC_{FC} = 0$ , and  $DEDI_{CF} = 0$ .
- Case II:  $B_C$  and  $B_F$  have some blocks in common, i.e.,  $B_C \cap F \neq \phi$ , but neither subsumes the other. Here,  $DISP_{CF}$ ,  $CONC_{FC}$ , and  $DEDI_{CF}$  are all between 0 and 1.
- Case III:  $B_C$  equals  $B_F$  which makes  $B_C \cap F = B_C \cup F$ . As a result,  $DISP_{CF} = 0$ , and  $CONC_{FC} = DEDI_{CF} = 1$ .
- Case IV:  $B_F$  is a subset of  $B_C$  and  $B_F \neq B_C$ , i.e.,  $B_F \subset B_C$ . We have  $DISP_{CF}$  and  $DEDI_{CF}$  between 0 and 1, and  $CONC_{FC} = 1$ .
- Case V:  $B_C$  is a subset of  $B_F$  and  $B_C \neq B_F$ , i.e.,  $B_C \subset B_F$ . We have  $DISP_{CF}$  and  $CONC_{FC}$  between 0 and 1, and  $DEDI_{CF} = 1$ .

A point worth noting is that a 100% concentration ( $CONC_{FC} = 1$ ) or a 100% dedication ( $DEDI_{CF} = 1$ ) does not guarantee a zero disparity between  $F$  and  $C$ . This disparity is 0 if and only if both concentration and dedication are 100%, i.e.,  $B_F = B_C$ . This disparity is 1 if and only if both concentration and dedication are 0, i.e.,  $B_C \cap F = \phi$ . As for feature concentration and component dedication, if  $CONC_{FC} \neq 0$  (i.e.,  $B_C \cap F \neq \phi$ ) then  $DEDI_{CF} \neq 0$ , and vice versa. Similarly, if  $CONC_{FC} = 0$  (i.e.,  $B_C \cap F = \phi$ ) then  $DEDI_{CF} = 0$ , and vice versa.

### 3 An Example

In this section, we use a numerical example to explain in detail how to compute the disparity, the concentration, and the dedication. To explain, without having the complex notation detract, we assume the program being examined has only two components ( $C_1$  and  $C_2$ ), two features ( $F_1$  and  $F_2$ ), and two invoking inputs ( $t_1$  and  $t_2$ ) focused on  $F_1$ . This simplification is not necessary and is removed from our case study on a real application in Section 4.

The first step is to find out how the program and each component are executed, in terms of how many and which blocks, by each invoking input. Suppose we have the following observations with the corresponding diagrammatic representation in Figure 2:

- $C_1$  has 5 blocks ( $b_{1i}$ ,  $1 \leq i \leq 5$ ) of which three ( $b_{11}$ ,  $b_{12}$ , and  $b_{13}$ ) are executed by  $t_1$ ; two ( $b_{13}$ ,  $b_{15}$ ) by  $t_2$ .
- $C_2$  has 6 blocks ( $b_{2j}$ ,  $1 \leq j \leq 6$ ) of which two ( $b_{21}$  and  $b_{22}$ ) are executed by  $t_1$ ; three ( $b_{22}$ ,  $b_{25}$ , and  $b_{26}$ ) by  $t_2$ .

Since  $B_{F_1}$  is the union of all the blocks in the program executed by at least one of the invoking inputs with respect to  $F_1$ , we have

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

In addition,

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

The disparities between  $C_1$  and  $F_1$ , and  $C_2$  and  $F_1$  are:

$$DISP_{C_1 F_1} = 1 - \frac{|B_{C_1} \cap F_1|}{|B_{C_1} \cup F_1|} = 1 - \frac{4}{9} = 0.556$$

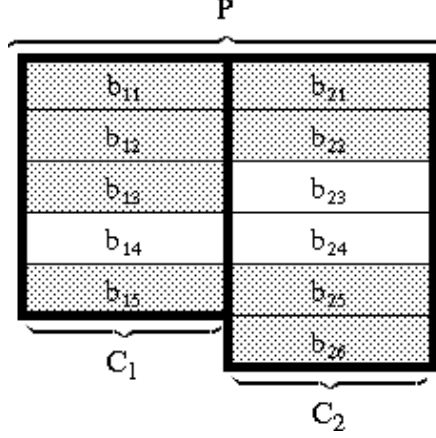


Figure 2: A diagrammatic representation of a program  $P$  with two components  $C_1$  and  $C_2$ . Each cell in the diagram is a block. The shaded ones are related to the feature  $F_1$ .

$$DISP_{C_2 F_1} = 1 - \frac{|B_{C_2} \cap F_1|}{|B_{C_2} \cup F_1|} = 1 - \frac{4}{10} = 0.60$$

The concentrations of  $F_1$  in  $C_1$  and  $C_2$ , respectively, are:

$$CONC_{F_1 C_1} = \frac{|B_{C_1} \cap F_1|}{|B_{F_1}|} = \frac{4}{8} = 0.50$$

$$CONC_{F_1 C_2} = \frac{|B_{C_2} \cap F_1|}{|B_{F_1}|} = \frac{4}{8} = 0.50$$

The dedications of  $C_1$  and  $C_2$ , respectively, to  $F_1$  are:

$$DEDI_{C_1 F_1} = \frac{|B_{C_1} \cap F_1|}{|B_{C_1}|} = \frac{4}{5} = 0.80$$

$$DEDI_{C_2 F_1} = \frac{|B_{C_2} \cap F_1|}{|B_{C_2}|} = \frac{4}{6} = 0.667$$

These measurements are consistent with the properties listed in Section 2.5. For example, feature  $F_1$  has 8 blocks: 4 in  $C_1$  and 4 in  $C_2$ . Therefore,  $F_1$  should have the same concentration (50% each) in  $C_1$  and  $C_2$ , respectively. On the other hand, since  $C_2$  has more blocks than  $C_1$ , the dedication of  $C_2$  to  $F_1$  (0.667) should be less than that of  $C_1$  (0.80). For the same reason, the disparity between  $C_2$  and  $F_1$  (0.60) is larger than that between  $C_1$  and  $F_1$  (0.556).

## 4 A Case Study

We now present a case study to show how our metrics help programmers and maintainers better understand their applications. We first provide an overview of the subject program and the tool used, describe the data collected, give our observations, and then discuss the issues related to the effectiveness of our metrics.

Table 1: Number of blocks in each file

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	mpfq.c	429	results.c	604
debug.c	94	mtta.c	134	share.c	702
expo.c	186	multipath.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

## 4.1 Subject Program

The program being studied is a Symbolic Hierarchical Automated Reliability and Performance Evaluator (SHARPE) [10]. It has 35K lines of C code in 30 files and has a total of 373 functions and 11,752 blocks. The size of each file in terms of the number of blocks is listed in Table 1. SHARPE was first developed in 1986 for three groups of users: practicing engineers, researchers in performance and reliability modeling, and students in engineering and science courses. Since then, several major revisions have been made to fix bugs and adopt new requirements.

SHARPE provides a specification language and analysis algorithms for nine different model types. Various transient and steady-state measures of interest are possible using the built-in functions in SHARPE. In this study, we view the specification of a model and the built-in functions that can be used for computing the measures of interest pertaining to that model as a feature. For example, the specification of Markov chains and the grouping of the built-in functions which facilitate the analysis process constitute a single feature referred to as MC. The other five features (i.e., five models) being examined are: Fault Trees (FT), Generalized Stochastic Petri-Nets (GSPN), Product-Form Queuing Networks (PFQN), Reliability Graphs (RELG), and Markov Reward Models (MRM).

## 4.2 Tool Support

Theoretically, one could argue that for a given feature and a program component, the disparity, concentration, and dedication can be computed manually without any tool. However, even if this is true for some rare cases, doing such computations by hand is not only time-consuming but also likely to be mistake-prone. In practice this is essentially impossible, especially for large complicated software systems. Therefore, we need a tool such as  $\chi$ Suds<sup>TM</sup> [2, 12], a Software Understanding System developed at Telcordia Technologies (formerly Bellcore).

Given a program,  $\chi$ Suds instruments it at compile time by inserting probes at appropriate locations. Then an executable is built on the instrumented code. Each time when the program is executed on an input, the complete traceability of that input, such as how many times each block is executed by that input, is appended to a corresponding trace file. With this information the execution slice of an input can be represented, for example, in terms of blocks. Once we have such slices (i.e.,  $B_{t_i}$  where  $t_i$  is an invoking input focused on a feature  $F$ ), we can easily compute  $B_F$  (the set of blocks used to implement  $F$ ). We can also compute, with respect to a given component  $C$ ,  $B_C \cap F$  (the set of blocks in the intersection of  $B_F$  and  $B_C$ ), and  $B_C \cup F$  (the set of blocks in the union of  $B_F$  and  $B_C$ ).

Note that while execution slices are represented only in blocks in this study, they can also be represented in decisions, c-uses, p-uses, and all-uses. More information is available in references [2, 4, 12] about  $\chi$ Suds and how blocks, decisions, c-uses, p-uses, and all-uses are defined in  $\chi$ Suds.

### 4.3 Data Collection & Verification

Since SHARPE is written in C, a natural way to decompose it is to treat each *.c* file as a separate program component. In our experiment, a set of invoking inputs focused on each of the six features (MC, FT, GSPN, PFQN, RELG, and MRM), respectively, was carefully selected from the regression test suite of SHARPE. One advantage of using regression tests is that they are the real inputs used during the integration and system testing. Another advantage is that there exist clear descriptions for many of these tests, which made it very easy for us to select invoking inputs focused on a given feature. The execution slice in terms of blocks of each of these inputs was computed. Computed also were the  $B_F$  for each of the six features and the  $B_C$  for each of the 30 files of SHARPE. Using  $B_F$ 's and  $B_C$ 's, we computed the dedication  $DEDI_{CF}$ , concentration  $CONC_{FC}$ , and disparity  $DISP_{CF}$  with respect to every possible pair of  $F$  and  $C$ . Metrics so computed are listed in Tables 2, 3, and 4.

The data collected in our experiments were presented to experts who were very familiar with the features of SHARPE and their corresponding location in the source code. As indicated by these experts, our data are promising as the identified blocks are used to implement the designated feature as they should be.

### 4.4 Observations

From our experimental data, we make the following observations:

- **Component Dedication (Refer to Table 2):**

1. None of the dedication is 1.

Table 2: 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
mpfq.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.

2. Ten of the 30 files are dedicated to all six features. Among them, cexpo.c (ranging from 38.67% to 76.11%) and read1.c (ranging from 39.91% to 66.03%) are heavily dedicated to every feature. Others such as results.c (61.92% to GSPN but 15.73% to PFQN) and analyze.c (7.49% to FT but 56.59% to MRM) are more dedicated to one feature than the other.
3. Three files are only dedicated to a single feature: 75.57% of reachgraph.c to GSPN, and 86.46% of pfqn.c and 54.78% of mpfq.c to PFQN.
4. Three files (debug.c, mttta.c, and multpath.c) have nothing to do with the features being examined because they are used for other model types in SHARPE which were not examined in this study.
5. The sum of dedications to all six features is listed in the rightmost column. Files whose sum is larger than 1 must have code shared by more than one feature. Files whose sum is smaller than 1 have code not used by any of the six features, but this does not necessarily mean they do not have any code common to multiple features. However, files having a larger sum (such as cexpo.c whose sum equals 3.2191) are, in general, more likely to have more code shared by multiple features than files having a smaller sum (such as in\_qn\_pn.c whose sum equals 0.5827). This implies that a code modification in cexpo.c is more likely to affect more than one feature than that in in\_qn\_pn.c.

• **Feature Concentration (Refer to Table 3):**

1. None of the concentration is 1.
2. For a given feature, the sum of its concentrations over the 30 files of SHARPE should be 1.0.
3. The concentration of each feature in each file is small—less than 0.1141 with only three exceptions: 12.21% of MRM in share.c, 28.75% of FT and 22.79% of RELG in ftree.c. In fact, the

Table 3: 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	
mpfq.c				0.0798		
mtta.c						
multipath.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			
readl.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.

Table 4: 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
mpfq.c	1	1	1	0.9190	1	1
mtta.c	1	1	1	1	1	1
multipath.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
readl.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

majority of the concentrations are less than 0.08 with many less than 0.05 (i.e., less than 5% of the blocks used to implement a feature are in the same file).

4. Feature MC is implemented in 18 files, FT in 14, GSPN in 20, PFQN in 20, RELG in 16, and MRM in 15. This suggests that the code segments for GSPN and PFQN are more widely spread over the system than FT and MRM.

• **Disparity between Feature and Component (Refer to Table 4):**

1. None of the disparity is 0 and the majority is larger than 0.90 with only a few exceptions.
2. Feature MC has a disparity equal to 1 with 12 files. This is consistent with the earlier observation that these 12 files have nothing to do with MC (i.e., they have 0% dedication to MC) and MC is spread over the other 18 files (i.e., the corresponding concentration is non-zero). Similarly, features FT, GSPN, PFQN, RELG, and MRM have a disparity equal to 1 with 16, 10, 10, 14, and 15 files, respectively.
3. Three files (`debug.c`, `mtta.c`, and `multipath.c`) have a disparity equal to 1 with all the six features. This is consistent with the observation based on the component dedication which shows that no block in these files is used to implement any of these six features. It is also consistent with the observation based on the feature concentration which shows none of the blocks that are used to implement these features are in these files.

The overall observation is that SHARPE has a very *delocalized structure* with features spread over many files. Our data show that for a given feature, in general, it has less than 8% concentration in a file, whereas for a given file, if it is used to implement a feature, it normally has at least more than 20% of its blocks dedicated to this feature. The fact that the concentration of the features in the files of SHARPE is small could be attributed to the incremental incorporation of the features into SHARPE, rather than planning for them in the original design.

## 4.5 Discussion

The effectiveness of our metrics in measuring the closeness (in terms of disparity, concentration, and dedication) between a feature and a program component depends on two factors: the accuracy of  $B_F$  for a given  $F$  and the metrics themselves. We discuss these two factors below.

### 4.5.1 Identification of code for a given feature

In theory, to find the complete set of code (blocks in our case) that is used to implement a given feature, one may have to run all possible inputs that exhibit this feature. In practice, this is not necessary. For a given feature  $F$ , one can identify most of its code by executing the program using a set of invoking inputs *focused* on  $F$ . Two questions arise while using this approach: “How many invoking inputs do we have to use?” and “How much code will be missed (i.e., how complete is  $B_F$ ?)” Note that if all the

invoking inputs are focused on  $F$ , we do not have any code in  $B_F$  that should not be included (i.e., all the code in  $B_F$  should be included). On the other hand, if some non-focused invoking inputs are used, we must exclude code which is not related to  $F$  from  $B_F$  (see Section 2.4).

Clearly, question 1 is related to question 2. Intuitively, the more invoking inputs we use, the more code that is used to implement  $F$  is likely to be included in  $B_F$ . However, we have to be very careful without overemphasizing the importance of the number of the invoking inputs. Once  $B_F$  contains code from the union of the execution slices of some invoking inputs, it becomes less and less likely for additional inputs to contribute more code to  $B_F$ . This is because code executed by such additional inputs has a high probability of also having been executed by previous selected invoking inputs. In fact, the more invoking inputs which have already been executed, the higher the probability.

We believe a practical way to select a good set of focused invoking inputs with respect to a given feature is to choose such inputs from the regression test suite of the program being examined (see Section 4.3). In our case study, we reviewed the descriptions of most of the regression tests which have been used for testing SHARPE. Based on these descriptions, all the tests with a clear focus, respectively, on each of the six features being examined were used to generate the corresponding  $B_F$ 's.

How good is the set  $B_F$  so generated? Testing the “goodness” of  $B_F$  in terms of whether it contains the complete set of code that is used to implement the feature  $F$  requires an *oracle*. This oracle can be obtained by using a mechanism different from our execution slice-based technique that can provide the expected results. For example, the design documentation can be an oracle. If that is the case, the system being analyzed must have a high degree of cohesion and a low degree of coupling such that each module addresses a specific subfunction of the requirements and has a simple interface when viewed from other parts of the program structure. In other words, there is a clear mapping between each feature and its corresponding code segments. Unfortunately, if a system is so well-designed, it cannot be a representative sample of real world programs (such as SHARPE) which have exactly the opposite structure: low cohesion and high coupling.

Another possibility is to use experts who have a good knowledge of the system being analyzed as the oracle. This is similar to the typical assessment that has been used in many reverse engineering studies where the output from a subsystem classification recovery technique is compared by a tester with the corresponding expected classification [6, 11]. One way to use experts to verify the  $B_F$ 's is to first request them to highlight code segments which they think are related to each feature. Such information then serves as the basis for the verification. An obvious difficulty of this approach is whether any such expert exists. For small programs, we may be able to find these experts. For large complicated systems, it is

impossible for humans to understand the functionality of every code segment at a fine granularity level such as block, even though they can do so at broader granularities such as file or function. Another difficulty is that this approach requires a tremendous amount of human resources which may not be affordable in the real world. The third difficulty is that different segments might be highlighted by different experts, or even the same expert at different times, which raises another series of problems about how to summarize such divergent information. Finally, due to the inevitability of human mistakes, there is no guarantee whatsoever that code so highlighted represents the complete set of code that is used to implement the feature being examined.

To overcome these problems, we took a more economic and a more feasible way by presenting each  $B_F$  to our experts and asking them whether any obvious code segments (blocks in our case) are missing. The result is that there is no additional block that has to be added to the  $B_F$ 's which clearly suggests a very good quality of our  $B_F$ 's. We realize that although very practical, this is not the perfect way to verify  $B_F$ 's. More effort should be devoted to developing a more objective verification. Nevertheless, we are also confident that each of the  $B_F$ 's in our case study contains the absolute majority of blocks that are used to implement the corresponding  $F$ .

#### 4.5.2 The goodness of our metrics

For a given software system, we can often get a *feeling* that one feature is more related (or in other words, *closer*) to one program component than another. The question is how to convert this abstract feeling into a quantitative measure. The first step is to develop a measurement based on an empirical relation to map from data collected in an empirical, real world to a formal, mathematical world [1, 3]. In our case, these measurements are the three metrics: disparity, dedication, and concentration. A measure based on these measurements is the number assigned to a pair of a feature and a program component by one of these mappings in order to characterize their *closeness*. A very important point is that these mappings (i.e., numbers so assigned) must preserve the relationships observed in the real world. For example, if we observe that a feature (say  $F_1$ ) is closer to a program component (say  $C$ ) than another feature (say  $F_2$ ), then the disparity metric must provide a mapping such that the disparity between  $F_1$  and  $C$  is smaller than that between  $F_2$  and  $C$ .

Following the same approach suggested by Fenton and Pfleeger [3], we started to develop our metrics based on our intuitions. We observed a list of properties that these metrics have to satisfy (see Section 2.5). Based on these properties, we defined the disparity metric for the disparity between a feature and a program component, the dedication metric for the dedication of a program component to a feature, and the concentration metric for the concentration of a feature in a program component.

Although these metrics might be approximate, it does not prevent them from being used for at least a preliminary assessment of the closeness between a feature and a program component. In fact, it is always the case that initially an objective measurement is usually established in approximate form by subjective measures [3, 6]. With this in mind, we applied our metrics to a real-world program, SHARPE. Results from our case study are analyzed by experts who know SHARPE well. The goal is to look for ways to improve the accuracy of the measures computed based on the three metrics. We are pleased with the results. All the measures based on our metrics are consistent with experts' expectations. For example, if the experts indicate that a feature has a higher concentration in one component than in another, the measures obtained from our concentration metric also agree with this.

As discussed in Section 4.5.1, since none of our experts have a complete understanding of the functionality of every basic block, the validation of our metrics was done in a *relative* rather than an *absolute* manner. An explanation of this is as follows. Suppose for a given program component  $C$  and three features  $F_\alpha$ ,  $F_\beta$ , and  $F_\gamma$ , our experts claim that  $C$  is most dedicated to  $F_\alpha$  and least to  $F_\gamma$ . Then, we should make sure the measures obtained from our dedication metric preserve the same order. In other words, we should have  $DEDI_{CF_\alpha} > DEDI_{CF_\beta} > DEDI_{CF_\gamma}$ . Furthermore, if the experts assert that  $C$  is very much more dedicated to  $F_\alpha$  but only little to  $F_\beta$  and  $F_\gamma$ , we will have to ensure that the difference between the measures obtained by using the dedication metric on  $F_\alpha$  and  $F_\beta$  is greater than that between  $F_\beta$  and  $F_\gamma$ . Stated differently,  $DEDI_{CF_\alpha} - DEDI_{CF_\beta}$  should be greater than  $DEDI_{CF_\beta} - DEDI_{CF_\gamma}$ . The only type of validation which we did not perform was to check the absolute value of a measure such as whether the dedication of  $C$  to  $F_\alpha$  should be 0.33 or 0.32. This is because none of our experts can provide an exactly complete set of blocks used for implementing each of the six features.

Overall, we have validated our metrics in such a way that measures obtained from them provide a good *inference* on how close a feature is to a program component. Although our metrics may have to be continuously evolved by using more data collected from other software systems, they can serve as a very good start to quantifying the closeness between a feature and a program component.

## 5 Conclusions

Three metrics are proposed: a disparity metric for the disparity between a feature and a program component, a dedication metric for the dedication of a program component to a feature, and a concentration metric for the concentration of a feature in a program component. One essential aspect of our metrics is that they provide a quantitative measure about possible interactions between program components and features. For example, given two features  $F_\alpha$  and  $F_\beta$ , as well as a component  $C$ , suppose that the dedication of  $C$  to  $F_\alpha$  is much less than that to  $F_\beta$ . Then, we can claim that modifications to  $C$  can very possibly have a much bigger impact on  $F_\beta$  than on  $F_\alpha$ . Another aspect is that from the dedications of a

component (say  $C$ ) to different features, we can predicate the *overlap* among these features in terms of how much of the code in  $C$  is shared by more than one feature. This information in turn can be a good measure of possible feature interactions in  $C$ . In addition, data listed in Tables 2, 3, and 4 (which give “component dedication to features,” “feature concentration in components” and “disparity between features and components,” respectively) provide a clear *traceability* for tracking each program component back to features which by themselves are defined based on a program’s requirements specification. For example, if the dedication of a component (say  $C$ ) to a feature (say  $F$ ) is zero, it implies that  $C$  is not used for implementing  $F$ . Thus, there is no traceability between  $C$  and  $F$ , or more specifically, there is no traceability between  $C$  and the requirements specification of  $F$ . The same conclusion can be obtained if the disparity between  $C$  and  $F$  is one or the concentration of  $F$  in  $C$  is zero. Altogether, results from our metrics help programmers understand how a feature spreads over a software system. Such results can also be used to estimate possible interactions between program components and features, as well as interactions between different features.

A case study was done on a real-world program, SHARPE. The feedback we received from experts who know SHARPE well indicates that these kinds of *quantitative measurements* have helped them in capturing more precisely where each feature resides in the system than using their *intuitive feeling* for the system, which provides only a *qualitative understanding* of where each feature is implemented in the system. This is especially true for applications which have essentially evolved and incrementally incorporated more features than originally conceived and planned for in the system design. The reason is that features in such systems are more likely to be scattered across many components than in systems where maintenance activities are limited to identifying and fixing the bugs.

For a program (such as SHARPE) which has a low cohesion and high coupling, the disparity between a feature and a component is in general large, and the concentration of a feature in a component as well as the dedication of a component to a feature are small. On the other hand, a well-designed software system should have a high degree of cohesion and a low degree of coupling such that each program component addresses a specific point of the requirements and has a simple interface when viewed from other parts of the program structure. Under this condition, the disparity between a feature and a component is in general small, and the concentration of a feature in a component as well as the dedication of a component to a feature are large.

To conclude, the encouraging results we have experienced lead us to believe that metrics discussed in this paper are complementary to other program comprehension techniques to help software programmers and maintainers better understand the system being examined.

## References

- [1] V. R. Basili and H. D. Rombach, "The TAME project: Towards improvement-oriented software environments," *IEEE Trans. on Software Engineering*, 14(6):758-773, June 1988.
- [2] J. R. Horgan et al., "Mining system tests to aid software maintenance," *IEEE Computer*, 31(7):64-73, July 1998.
- [3] N. E. Fenton and S. L. Pfleeger, "*Software Metrics: A Rigorous and Practical Approach*," PWS Publishing Company, Boston, MA, 1997.
- [4] J. R. Horgan and S. A. London, "Data flow coverage and the C language," in *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification*, pp 87-97, Victoria, British Columbia, Canada, October 1991.
- [5] "IEEE Guide to Software Requirements Specifications," ANSI/IEEE Std 830-1984, 1984.
- [6] A. Lakhotia and J. M. Gravley, "Toward experimental evaluation of subsystem classification recovery techniques," in *Proceedings of the Second Working Conference on Reverse Engineering*, Toronto, Canada, July 1995.
- [7] "Military Standard: Defense System Software Development (DOD-STD-2167A)," Department of Defense, February 1988.
- [8] Meilir Page-Jones, "*The Practical Guide to Structured Systems Design (Yourdon Press Computing Series)*," Prentice Hall, Englewood Cliffs, New Jersey, 1988.
- [9] R. S. Pressman, "*Software Engineering: A Practitioner's Approach*," McGraw-Hill, New York, 1997.
- [10] R. A. Sahner, K. S. Trivedi, and A. Puliafito, "*Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package*," Kluwer Academic Publishers, Boston, 1996.
- [11] R. Schwanke, "An intelligent tool for reengineering software modularity," in *Proceedings of the 13th IEEE International Conference on Software Engineering*, 1991.
- [12] "χSuds User's Manual," Telcordia Technologies, 1998.
- [13] N. Wilde and C. Casey, "Early field experience with the software reconnaissance technique for program comprehension," in *Proceedings of the International Conference on Software Maintenance*, pp 312-318, Monterey, CA, November 1996.

- [14] N. Wilde and M. S. Scully, "Software reconnaissance: Mapping program features to code," *Software Maintenance: Research and Practice*, 7(1):49-62, 1995.
- [15] 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, TX, March 1999.