NASA/~~CR~~ TM - 1998 - 208614

SOFTWARE ENGINEERING LABORATORY SERIES

SEL-97-002

IN-82
415445
10P

COLLECTED SOFTWARE ENGINEERING PAPERS: VOLUME XV

OCTOBER 1997

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

SOFTWARE ENGINEERING LABORATORY SERIES SEL-97-002

COLLECTED SOFTWARE ENGINEERING PAPERS: VOLUME XV

OCTOBER 1997

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

FOREWORD

The Software Engineering Laboratory (SEL) is an organization sponsored by the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) and created to investigate the effectiveness of software engineering technologies when applied to the development of application software. The SEL was created in 1976 and has three primary organizational members:

NASA/GSFC, Systems Integration and Engineering Branch

University of Maryland, Department of Computer Science

Computer Sciences Corporation, Development and Systems Engineering organization

The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effect of various methodologies, tools, and models on this process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document.

Documents from the Software Engineering Laboratory Series can be obtained via the SEL homepage at:

http://fdd.gsfc.nasa.gov/seltext.html

or by writing to:

Systems Integration and Engineering Branch
Code 581
Goddard Space Flight Center
Greenbelt, Maryland, U.S.A. 20771

TABLE OF CONTENTS

SECTION 1—INTRODUCTION

This document is a collection of selected technical papers produced by participants in the Software Engineering Laboratory (SEL) from September 1996 through September 1997. The purpose of the document is to make available, in one reference, some results of SEL research that originally appeared in a number of different forums. This is the 15th such volume of technical papers produced by the SEL. Although these papers cover several topics related to software engineering, they do not encompass the entire scope of SEL activities and interests. Additional information about the SEL and its research efforts may be obtained from the sources listed in the bibliography at the end of this document, or via the SEL Home Page on the World Wide Web at *http://fdd.gsfc.nasa.gov/seltext.html.*

For the convenience of this presentation, the fourteen papers contained here are grouped into four major sections:

- Software Measurement (Section 2)
- Software Models (Section 3)
- Technology Evaluations (Section 4)
- Ada Technology (Section 5)

Section 2 includes several papers that describe software system measurement, measurement scales, software properties and reliability studies. It also outlines an approach for defining evaluation criteria for reusable software components. The paper in Section 3, indicates a study where the researchers characterize and model the cost of rework in a Component Factor organization. Section 4 includes papers that discuss a knowledge-based analysis approach that generates fir order predicate logic annotations of loops, outlines the Riskit method in a case study, provides a description of an empirical study which addresses the issue of communication among members of a software development organization, and how reuse may influence productivity in object-oriented systems. Lastly, papers in Section 5 discuss the use of the Intermetrics AppletMagic tool to build an applet to display a satellite ground track on a world map, and describes the Generalized Support Software (GSS) architecture and process research results in the Flight Dynamics Division (FDD) of NASA's Goddard Space Flight Center.

The SEL is actively working to understand and improve the software development process at the Goddard Space Flight Center (GSFC). Future efforts will be documented in additional volumes of the *Collected Software Engineering Papers* and other SEL publications.

OMIT THIS PAGE

SECTION 2—SOFTWARE MEASUREMENT

The technical papers included in this section were originally prepared as indicated below.

- "Property-Based Software Engineering Measurement," L. C. Briand, S. Morasca and V. R. Basili, *IEEE Transactions on Software Engineering*, vol. 22, no. 1, January 1996, pp. 68-85
- "A Validation of Object-Oriented Design Metrics as Quality Indicators," V. R. Basili, L. C. Briand and W. L. Melo, *IEEE Transactions on Software Engineering*, vol. 22, no. 10, October 1996, pp. 751-761
- Comments on "Towards a Framework for Software Measurement Validation," S. Morasca, L. C. Briand, E. J. Weyuker and M. V. Zelkowitz, *IEEE Transactions on Software Engineering*, vol. 23, no. 3, March 1997, pp. 187-188
- Response to: Comments on "Property-Based Software Engineering Measurement: Refining the Additivity Properties", L. C. Briand, S. Morasca and V. R. Basili, *IEEE Transactions on Software Engineering*, vol. 23, no. 3, March 1997, pp. 196-197
- "Analytical and Empirical Evaluation of Software Reuse Metrics," P. Devanbu, S. Karstu, W. L. Melo and W. Thomas, *Proceedings of the 18th International Conference on Software Engineering (ICSE-18),* Berlin, Germany, March 1996, pp. 189-199
- "Why Software Reliability Predictions Fail," F. Lanubile, *IEEE Software*, pp. 131-132 and 137, July 1996
- "Defining Factors, Goals and Criteria for Reusable Component Evaluation," J. Kontio, G. Caldiera and V. R. Basili, *CASCON '96 Conference*, Toronto, Canada, November 1996

S1-61
415785
360712
18P

Property-Based Software Engineering Measurement

Lionel C. Briand, Sandro Morasca, *Member, IEEE Computer Society*, and Victor R. Basili, *Fellow, IEEE*

Abstract—Little theory exists in the field of software system measurement. Concepts such as complexity, coupling, cohesion or even size are very often subject to interpretation and appear to have inconsistent definitions in the literature. As a consequence, there is little guidance provided to the analyst attempting to define proper measures for specific problems. Many controversies in the literature are simply misunderstandings and stem from the fact that some people talk about different measurement concepts under the same label (complexity is the most common case).

There is a need to define unambiguously the most important measurement concepts used in the measurement of software products. One way of doing so is to define precisely what mathematical properties characterize these concepts, regardless of the specific software artifacts to which these concepts are applied. Such a mathematical framework could generate a consensus in the software engineering community and provide a means for better communication among researchers, better guidelines for analysts, and better evaluation methods for commercial static analyzers for practitioners.

In this paper, we propose a mathematical framework which is generic, because it is not specific to any particular software artifact, and rigorous, because it is based on precise mathematical concepts. We use this framework to propose definitions of several important measurement concepts (size, length, complexity, cohesion, coupling). It does not intend to be complete or fully objective; other frameworks could have been proposed and different choices could have been made. However, we believe that the formalisms and properties we introduce are convenient and intuitive. This framework contributes constructively to a firmer theoretical ground of software measurement.

Index Terms—Software measurement, measure properties, measurement theory, size, complexity, cohesion, coupling.

1 Introduction

Many concepts have been introduced through the years to define the internal attributes [1] of the artifacts produced during the software process. For instance, one speaks of size and complexity of a software specification, design, and code, or cohesion and coupling of a software design or code. Several techniques have been introduced, with the goal of producing software which is better with respect to these concepts. As an example, Parnas' [2] design principles attempt to decrease coupling between modules, and increase cohesion within modules. These concepts are used as a guide to choose among alternative techniques or artifacts. For instance, a technique may be preferred over another because it yields artifacts that are less complex; an artifact may be preferred over another because it is less complex. In turn, lower complexity is believed to provide advantages such as lower maintenance time and cost. In general, it is commonly believed that there is a relationship between internal attributes (e.g., size, complexity, cohesion) and external attributes (e.g., maintainability, understandability). This shows the importance of a clear and unambiguous understanding of what these concepts actually mean, to make choices on more objective bases. The definition of relevant concepts (i.e., classes of software characterization measures) is the first step towards quantitative assessment of software artifacts and techniques, which is needed to assess risk and find optimal trade-offs among software quality, schedule, and cost of development.

To capture these concepts in a quantitative fashion, hundreds of software measures have been defined in the literature. However, the vast majority of these measures did not survive the proposal phase, and did not manage to get accepted in the academic or industrial worlds. One reason for this is the fact that they have not been built by using a clearly defined process for defining software measures. As we propose in [3], such a process should be driven by clearly identified measurement goals and knowledge of the software process. One of its crucial activities is the precise definition of relevant concepts, necessary to lay down a rigorous framework for software engineering measures and to define meaningful and well-founded software measures. The theoretical soundness of a measure, i.e., the fact that it really measures the software characteristic it is supposed to measure, is an obvious prerequisite for its acceptability and use. The exploratory process of looking for correlations is not an acceptable scientific validation process in itself if it is not accompanied by a solid theory to support it [4]. Unfortunately, new software measures are very often defined to capture elu-

• *L.C. Briand is with Centre de Recherche Informatique de Montréal (CRIM), 1801 McGill College Ave., Montréal, Québec, H3A 2N4, Canada. E-mail: lionel.briand@crim.ca.*
• *S. Morasca is with Dip. di Elettronica e Informazione, Politecnico di Milano, Piazza Leonardo da Vinci 32, I-20133 Milano, Italy. E-mail: morasca@elet.polimi.it.*
• *V.R. Basili is with the Computer Science Department, University of Maryland, College Park, MD 20742. E-mail: basili@cs.umd.edu.*

Manuscript received November 1994; revised September 1995.

For information on obtaining reprints of this article, please send e-mail to: transactions@computer.org, and reference IEEECS Log Number S95066.

sive concepts such as complexity, cohesion, coupling, connectivity, etc. (Only size can be thought to be reasonably well understood.) Thus, it is impossible to assess the theoretical soundness of newly proposed measures, and the acceptance of a new measure is mostly a matter of belief.

To this end, several proposals have appeared in the literature [5], [6], [7] in recent years to provide desirable properties for software measures. These works (especially [7]) have been used to "validate" existing and newly proposed software measures. Surprisingly, whenever a new measure which was proposed as a software complexity measure did not satisfy the set of properties against which it was checked, several authors failed to conclude that their measure was not a software complexity measure, e.g., [8], [9]. Instead, they concluded that their measure was a complexity measure that does not satisfy that set of properties for complexity measures. What they actually did was provide an absolute definition of a software complexity measure and check whether the properties were consistent with respect to the measure, i.e., check the properties against their own measure.

This situation would be unacceptable in other engineering or mathematical fields. For instance, suppose that one defines a new measure, claiming it is a distance measure. Suppose also that that measure fails to satisfy the triangle inequality, which is the characterizing property of distance measures. The natural conclusion would be to realize that that is not a distance measure, rather than to say that it is a distance measure that does not satisfy the conditions for a distance measure. However, it is true that none of the sets of properties proposed so far has reached so wide an acceptance to be considered "the" right set of necessary properties for complexity. It is our position that this odd situation is due to the fact that there are several different concepts which are still covered by the same word: complexity.

Within the set of commonly mentioned software characteristics, size and complexity are the ones that have received the widest attention. However, several authors have been inclined to believe that a measure captures either size or complexity, as if, besides size, all other concepts related to software characteristics could be grouped under the unique name of complexity. Sometimes, even size has been considered as a particular kind of complexity measure.

Actually, these concepts capture different software characteristics, and, until they are clearly separated and their similarities and differences clearly studied, it will be impossible to reach any kind of consensus on the properties that characterize each concept relevant to the definition of software measures. The goal of this paper is to lay down the basis for a discussion on this subject, by providing properties for a—partial—set of measurement concepts that are relevant for the definition of measures of internal software attributes. Many of the measure properties proposed in the literature are generic in the sense that they do not characterize specific measurement concepts but are relevant to all syntactically-based measures (see [10], [6], [7]). In this paper, we want to focus on properties that differentiate measurement concepts such as size, complexity, cohesion and coupling, which are the ones that are most commonly found in the scientific literature. Thus, we want to identify and clarify the essential properties behind these concepts that are commonplace in software engineering and form important classes of measures. Thus, researchers will be able to validate their new measures by checking properties specifically relevant to the class (or concept) they belong to (e.g., size should be additive). *By no means should these properties be regarded as the unique set of properties that can be possibly defined for a given concept.* Rather, we want to provide a theoretically sound and convenient solution for differentiating a set of well known concepts and check their analogies and conflicts. In other words, we attempt to define these concepts through different sets of unambiguous and intuitive properties. Possible applications of such a framework are to guide researchers in their search for new measures and help practitioners evaluate the adequacy of measures provided by commercial tools.

All of the previously mentioned measurement concepts are related to internal software attributes. In particular, we will focus on one of the "flavors" of complexity that has been used in the literature—related to the structure of a software system. Our definition of complexity does not encompasses external attributes, i.e., we do not provide properties for understandability, etc. Therefore, the part of our work related to complexity is in the same line of thought as [7]. Weyuker [7], one of the earliest works on the subject and by far the most referenced set of properties, has been criticized by several authors as being inconsistent [11] and incomplete [12] and is still intensively discussed. Other definitions, corresponding to different "flavors" of complexity, have been provided in the literature, e.g., [13].

We also believe that the investigation of measures should also address artifacts produced in the software process other than code. It is commonly believed that the early software process phases are the most important ones, since the rest of the development depends on the artifacts they produce. Oftentimes, the concepts (e.g., size, complexity, cohesion, coupling) which are believed relevant with respect to code are also relevant for other artifacts. To this end, the properties we propose will be general enough to be applicable to a wide set of artifacts.

The paper is organized as follows. In Section 2, we introduce the basic definitions of our framework. Section 3 provides a set of properties that characterize and formalize intuitively relevant measurement concepts: size, length, complexity, cohesion, coupling. We also discuss the relationships and differences between the different concepts and how they relate to the measurement theory framework [14]. Some of the best-known measures are used as examples to illustrate our points. Section 4 contains comparisons and discussions regarding the set of properties for complexity measures defined in the paper and in the literature. The conclusions and directions for future work come in Section 5.

2 Basic Definitions

Before introducing the necessary properties for the set of concepts we intend to study, we provide basic definitions related to the objects of study (to which these concepts can be applied), e.g., size and complexity of *what*?

2.1 Systems and Modules

Two of the concepts we will investigate, namely, size (Section 3.1) and complexity (Section 3.3) are related to systems in general, i.e., one can speak about the size of a system and the complexity of a system. We also introduce a new concept, length (Section 3.2), which is related to systems. In our general framework—recall that we want these properties to be as independent as possible of any product abstraction—a system is characterized by its elements and the relationships between them. Thus, we do not reduce the number of possible system representations, as elements and relationships can be defined according to needs.

DEFINITION 1: Representation of Systems and Modules. *A system* S *will be represented as a pair* <E, R>, *where* E *represents the set of elements of* S, *and* R *is a binary relation on* E ($R \subseteq E \times E$) *representing the relationships between S's elements.*

Given a system S = <E, R>, *a system* m = <E_m, R_m> *is a* module *of* S *if and only if* $E_m \subseteq E$, $R_m \subseteq E_m \times E_m$, *and* $R_m \subseteq R$. *As an example,* E *can be defined as the set of code statements and* R *as the set of control flows from one statement to another. A module m may be a code segment or a subprogram.*

The elements of a module are connected to the elements of the rest of the system by incoming and outgoing relationships. The set InputR(m) *of relationships from elements outside module* m = <E_m, R_m> *to those of module m is defined as*

$$\text{InputR}(m) = \{<e_1, e_2> \in R \mid e_2 \in E_m \text{ and } e_1 \in E - E_m\}$$

The set OutputR(m) *of relationships from the elements of a module* m = <E_m, R_m> *to those of the rest of the system is defined as*

$$\text{OutputR}(m) = \{<e_1, e_2> \in R \mid e_1 \in E_m \text{ and } e_2 \in E - E_m\}$$

□

We now introduce inclusion, union, intersection operations for modules and the definitions of empty and disjoint modules, which will be used often in the remainder of the paper. For notational convenience, they will be denoted by extending the usual set-theoretic notation. We will illustrate these operations by means of the system S = <E, R> represented in Fig. 1, where E = {a, b, c, d, e, f, g, h, i, j, k, l, m} and R = {<b, a>, <b, f>, <c, b>, <c, d>, <c, g>, <d, f>, <e, g>, <f, i>, <f, k>, <g, m>, <h, a>, <h, i>, <i, j>, <k, j>, <k, l>}. We will consider the following modules

- m_1 = <E_{m1}, R_{m1}> = <{a, b, f, i, j, k}, {<b, a>, <b, f>, <f, i>, <f, k>, <i, j>, <k, j>}> (area filled with ▨)
- m_2 = <E_{m2}, R_{m2}> = <{f, j, k}, {<f, k>, <k, j>}> (area filled with ▩)
- m_3 = <E_{m3}, R_{m3}> = <{c, d, e, f, g, j, k, m}, {<c, d>, <c, g>, <d, f>, <e, g>, <f, k>, <g, m>, <k, j>}> (area filled with ▧)
- m_4 = <E_{m4}, R_{m4}> = <{d, e, g}, {<e, g>}> (area filled with ■)

Inclusion. Module m_i = <E_{mi}, R_{mi}> is said to be included in module m_j = <E_{mj}, R_{mj}> (notation: $m_i \subseteq m_j$) if $E_{mi} \subseteq E_{mj}$ and $R_{mi} \subseteq R_{mj}$. In Fig. 1, $m_4 \subseteq m_3$.

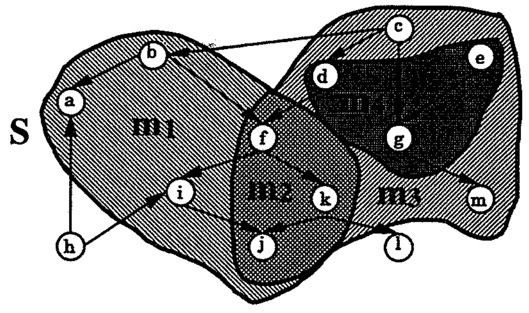


Fig. 1. Operations on modules.

Union. The union of modules m_i = <E_{mi}, R_{mi}> and m_j = <E_{mj}, R_{mj}> (notation: $m_i \cup m_j$) is the module <$E_{mi} \cup E_{mj}$, $R_{mi} \cup R_{mj}$>. In Fig. 1, the union of modules m_1 and m_3 is module m_{13} = <{a, b, c, d, e, f, g, i, j, k, m}, {<b, a>, <b, f>, <c, b>, <c, d>, <c, g>, <d, f>, <e, g>, <f, i>, <f, k>, <g, m>, <i, j>, <k, j>}> (area filled with ▨ or ▩ or ▧).

Intersection. The intersection of modules m_i = <E_{mi}, R_{mi}> and m_j = <E_{mj}, R_{mj}> (notation: $m_i \cap m_j$) is the module <$E_{mi} \cap E_{mj}$, $R_{mi} \cap R_{mj}$>. In Fig. 1, $m_2 = m_1 \cap m_3$.

Empty module. Module <∅, ∅> (denoted by ∅) is the empty module.

Disjoint modules. Modules m_i and m_j are said to be disjoint if $m_i \cap m_j = \emptyset$. In Fig. 1, $m_1 \cap m_4 = \emptyset$.

Since in this framework modules are just subsystems, all systems can theoretically be decomposed into modules. The definition of a module for a particular measure in a specific context is just a matter of convenience and programming environment (e.g., language) constraints.

2.2 Modular Systems

The other two concepts we will investigate, cohesion (Section 3.4) and coupling (Section 3.5), are meaningful only with reference to systems that are provided with a modular decomposition, i.e., one can speak about cohesion and coupling of a whole system only if it is structured into modules. One can also speak about cohesion and coupling of a single module within a whole system.

DEFINITION 2: Representation of Modular Systems. *The 3-tuple* MS = <E, R, M> *represents a modular system if* S = <E, R> *is a system according to Definition 1, and* M *is a collection of modules of* S *such that*

$$\forall e \in E\ (\exists m \in M\ (m = <E_m, R_m> \text{ and } e \in E_m)) \text{ and}$$

$$\forall m_i, m_j \in M\ (m_i = <E_{mi}, R_{mi}> \text{ and } m_j = <E_{mj}, R_{mj}> \text{ and } E_{mi} \cap E_{mj} = \emptyset)$$

i.e, the set of elements E *of* MS *is partitioned into the sets of elements of the modules.*

We denote the union of all the Rms *as* IR. *It is the set of* intramodule relationships. *Since the modules are disjoint, the union of all* OutputR(m)*s is equal to the union of all* InputR(m)*s, which is equal to* R-IR. *It is the set of* intermodule relationships. □

As an example, E can be the set of all declarations of a given set of Ada modules, R the set of dependencies between them, and M the set of Ada modules.

Fig. 2 shows a modular system MS = <E, R, M>, obtained by partitioning the set of elements of the system in Fig. 1 in a different way. In this modular system, E and R are the same as in system S in Fig. 1, and M = {m_1, m_2, m_3}. Besides, IR = {<b, a>, <c, d>, <c, g>, <e, g>, <f, i>, <f, k>, <g, m>, <h, a>, <i, j>, <k, j>, <k, l>}.

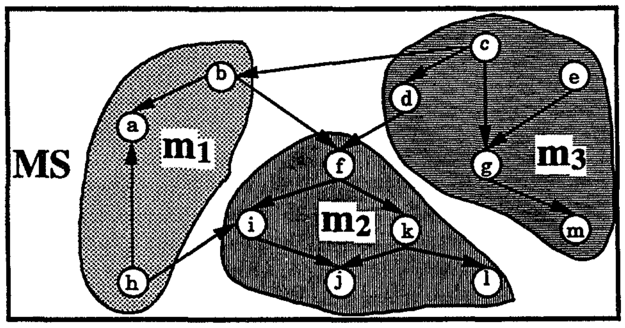


Fig. 2. A modular system.

It should be noted that some measurement concepts do not take into account the modular structure of a system. As already mentioned, our concepts of size and complexity (defined in Sections 3.1 and 3.3) are such examples.

We have defined concept properties using a graph-theoretic approach to allow us to be general and precise. It is general because our properties are defined so that no restriction applies to the definition of vertices and arcs. Many well known product abstractions fit this framework, e.g., data dependency graphs, definition-use graphs, control flow graphs, USES graphs, Is_Component_of graphs. It is precise because, based on a well defined formalism, all the concepts used can be mathematically defined, e.g., system, module, modular system, and so can the properties presented in the next section.

3 Measurement Concepts and their Properties

It should be noted that the concepts defined below are to some extent subjective. However, we wish to assign them unambiguous, intuitive, and convenient properties. We consider these properties necessary but not sufficient because they do not guarantee that the measures for which they hold are useful or even make sense. On the other hand, these properties will constrain the search for measures and therefore make the measure definition process more rigorous and less exploratory [3]. Several relevant concepts are studied: size, length, complexity, cohesion, and coupling. They do not represent an exhaustive list but a starting point for discussion that should eventually lead to a standard definition set in the software engineering community.

In what follows, we do not provide any informal definition for the concepts introduced (e.g., complexity) because we consider that the properties themselves uniquely characterize and therefore define the concepts in an unambiguous manner. However, intuitive justifications are provided to support the properties.

3.1 Size

3.1.1 Motivation

Intuitively, size is recognized as being an important measurement concept. According to our framework, size cannot be negative (property Size.1), and we expect it to be null when a system does not contain any elements (property Size.2). When modules do not have elements in common, we expect size to be additive (property Size.3).

Definition 3: Size. *The size of a system* S *is a function* Size(S) *that is characterized by the following properties* Size.1-Size.3. □

Property Size.1: Nonnegativity. The size of a system S = <E, R> is nonnegative

$$\text{Size}(S) \geq 0 \qquad \text{(Size.I)} \qquad \square$$

Property Size.2: Null Value. The size of a system S = <E, R> is null if E is empty

$$E = \varnothing \Rightarrow \text{Size}(S) = 0 \qquad \text{(Size.II)} \qquad \square$$

Property Size.3: Module Additivity. The size of a system S = <E, R> is equal to the sum of the sizes of two of its modules m_1 = <E_{m1}, R_{m1}> and m_2 = <E_{m2}, R_{m2}> such that any element of S is an element of either m_1 or m_2

$$(m_1 \subseteq S \text{ and } m_2 \subseteq S \text{ and } E = E_{m1} \cup E_{m2} \text{ and } E_{m1} \cap E_{m2} = \varnothing)$$
$$\Rightarrow \text{Size}(S) = \text{Size}(m_1) + \text{Size}(m_2) \qquad \text{(Size.III)} \qquad \square$$

For instance, the size of the system in Fig. 2 is the sum of the sizes of its three modules m_1, m_2, m_3.

The following three unnumbered properties follow from the above properties Size.1-Size.3.

Property Size.3 provides the means to compute the size of a system S = <E, R> from the knowledge of the size of its—disjoint—modules m_e = <{e}, R_e> whose set of elements is composed of a different element e of E.[1]

$$\text{Size}(S) = \sum_{e \in E} \text{Size}(m_e) \qquad \text{(Size.IV)}$$

Therefore, adding elements to a system cannot decrease its size (*size monotonicity property*)

$$(S' = \langle E', R' \rangle \text{ and } S'' = \langle E'', R'' \rangle \text{ and } E' \subseteq E'')$$
$$\Rightarrow \text{Size}(S') \leq \text{Size}(S'') \qquad \text{(Size.V)}$$

From the above properties, Size.1-Size.3, it follows that the size of a system S = <E, R> is not greater than the sum of the sizes of any pair of its modules m_1 = <E_{m1}, R_{m1}> and m_2 = <E_{m2}, R_{m2}>, such that any element of S is an element of m_1, or m_2, or both, i.e.,

$$(m_1 \subseteq S \text{ and } m_2 \subseteq S \text{ and } E = E_{m1} \cup E_{m2})$$
$$\Rightarrow \text{Size}(S) \leq \text{Size}(m_1) + \text{Size}(m_2) \qquad \text{(Size.VI)}$$

The size of a system built by merging such modules cannot be greater than the sum of the sizes of the modules, due to the presence of common elements (e.g., lines of code, operators, class methods).

1. For each m_e, it is either $R_e = \varnothing$ or R_e = {<e, e>}.

Properties Size.1-Size.3 hold when applying the admissible transformation of the ratio scale[2] (i.e., $f(x) = \alpha x$) [14]. Therefore, there is no contradiction between our concept of size and the definition of size measures on a ratio scale. In other words, the properties do not block the way to the ratio scale. Further discussions on measurement theory and its relationship to our framework will be provided in Section 3.6.

3.1.2 Examples and Counterexamples of Size Measures

Several measures introduced in the literature can be classified as size measures, according to our properties Size.1-Size.3. With reference to code measures, we have: LOC, #Statements, #Modules, #Procedures, Halstead's Length [15], #Occurrences of Operators, #Occurrences of Operands, #Unique Operators, #Unique Operands. In each of the above cases, the representation of a program as a system is quite straightforward. Each counted entity is an element, and the relationship between elements is just the sequential relationship.

Some other measures that have been introduced as size measures do not satisfy the above properties. Instances are the Estimator of length and Volume [15], which are not additive when software modules are disjoint (property Size.3). Indeed, for both measures, the value obtained when two disjoint software modules are concatenated may be less than the sum of the values obtained for each module, since they may contain common operators or operands. Note that, in this context, the graph is just the sequence of operand and operator occurrences. Disjoint code segments are disjoint subgraphs.

On the other hand, other measures, that are meant to capture other concepts, are indeed size measures. For instance, in the object-oriented suite of measures defined in [8], Weighted Methods per Class (WMC) is defined as the sum of the complexities of methods in a class. First, it is straightforward to show that properties Size.1 and Size.2 are true for WMC. In addition, when two classes without methods in common are merged, the resulting class's WMC is equal to the sum of the two WMCs of the original classes (property Size.3 is satisfied). As a consequence, when two classes with methods in common are merged, then the WMC of the resulting class may be lower than the sum of the WMCs of the two original classes (formula Size.VI, which can be deduced from properties Size.1-3). Therefore, since all size properties hold, this is a class size measure. However, WMC does not satisfy our properties for complexity measures (see Section 3.3). Likewise, NOC (Number Of Children of a class) and Response For a Class (RFC) [8] are other size measures, according to our properties.

3.2 Length

3.2.1 Motivation

Properties Size.1-Size.3 characterize the concept of size as is commonly intended in software engineering. Actually, the concept of size may have different interpretations in everyday life, depending on the measurement goal. For instance, suppose we want to park a car in a parallel parking space. Then, the "size" we are interested in is the maximum distance between two points of the car linked by a segment parallel to the car's motion direction. The above properties Size.1-Size.3 do not aim at defining such a measure of size. With respect to physical objects, volume and weight satisfy the above properties. In the particular case that the objects are unidimensional (or that we are interested in carrying out measurements with respect to only one dimension), then these concepts coincide.

In order to differentiate this measurement concept from size, we call it *length*. Length is nonnegative (property Length.1), and equal to 0 when there are no elements in the system (property Length.2). In extreme situations where systems are composed of unrelated elements this property allows length to be nonnull. If a new relationship is introduced between two elements belonging to the same connected component[3] of the graph representing a system, the length of the new system is not greater than the length of the original system (property Length.3). The idea is that, in this case, a new relationship may make the elements it connects "closer" than they were. This new relationship may reduce the greatest distance between elements in the connected component of the graph, but it may never increase it. On the other hand, if a new relationship is introduced between two elements belonging to two different connected components, the length of the new system is not smaller than the length of the original system. This stems from the fact that the new relationship creates a new connected component, where the maximum distance between two elements cannot be less than the maximum distance between any two elements of either original connected component (property Length.4). Length is not additive for disjoint modules. The length of a system containing several disjoint modules is the maximum length among them (property Length.5).

DEFINITION 4: Length. *The length of a system* S *is a function* Length(S) *characterized by the following properties* Length.1-*Length.5.* □

PROPERTY LENGTH.1: Nonnegativity. The length of a system S = <E, R> is nonnegative

$$\text{Length}(S) \geq 0 \qquad (\text{Length.I})$$ □

PROPERTY LENGTH.2: Null Value. The length of a system S = <E, R> is null if E is empty

$$(E = \emptyset) \Rightarrow (\text{Length}(S) = 0) \qquad (\text{Length.II})$$ □

PROPERTY LENGTH.3: Nonincreasing Monotonicity for Connected Components. Let S be a system and m be a module of S such that m is represented by a connected component of the graph representing S. Adding relationships between elements of m does not increase the length of S.

2. In other words, these properties hold when $\text{Size}(m_i)$ is substituted with $\alpha\,\text{Size}(m_i)$, where α is an arbitrary coefficient.

3. Here, two elements of a system S are said to belong to the same connected component if there is a path from one to the other in the nondirected graph obtained from the graph representing S by removing directions in the arcs.

(S = <E, R> and $m = <E_m, R_m>$ and $m \subseteq S$
and m "is a connected component of S" and
$S' = <E, R'>$ and $R' = R \cup \{<e_1, e_2>\}$ and $<e_1, e_2> \notin R$
and $e_1 \in E_{m1}$ and $e_2 \in E_{m1}$)
$\Rightarrow$ Length(S) $\geq$ Length(S') (Length.III) □

PROPERTY LENGTH.4: Nondecreasing Monotonicity for Nonconnected Components. Let S be a system and m_1 and m_2 be two modules of S such that m_1 and m_2 are represented by two separate connected components of the graph representing S. Adding relationships from elements of m_1 to elements of m_2 does not decrease the length of S

(S = <E, R> and $m_1 = <E_{m1}, R_{m1}>$ and $m_2 = <E_{m2}, R_{m2}>$
and $m_1 \subseteq S$ and $m_2 \subseteq S$ "are separate connected components of S" and
$S' = <E, R'>$ and $R' = R \cup \{<e_1, e_2>\}$ and $<e_1, e_2> \notin R$
and $e_1 \in E_{m1}$ and $e_2 \in E_{m2}$)
$\Rightarrow$ Length(S') $\geq$ Length(S) (Length.IV) □

PROPERTY LENGTH.5: Disjoint Modules. The length of a system S = <E, R> made of two disjoint modules m_1, m_2 is equal to the maximum of the lengths of m_1 and m_2

($S = m_1 \cup m_2$ and $m_1 \cap m_2 = \emptyset$ and $E = E_{m1} \cup E_{m2}$)
$\Rightarrow$ Length(S) = max{Length(m_1), Length(m_2)}
(Length.V) □

Let us illustrate the last three properties with systems S, S', S", represented in Fig. 3. The length of system S, composed of the three connected components m_1, m_2, and m_3, is the maximum value among the lengths of m_1, m_2, and m_3 (property Length.V). System S' differs from system S only because of the added relationship <c, m> (represented by the thick dashed arrow), which connects two elements already belonging to a connected component of S, m_3. The length of system S' is not greater than the length of S (property Length.III). System S" differs from system S only because of the added relationship <b, f> (represented by the thick solid arrow), which connects two elements belonging to two different connected components of S, m_1, and m_2. The length of system S" is not less than the length of S (property Length.IV).

Properties Length.1-Length.5 hold when applying the admissible transformation of the ratio scale. Therefore, there is no contradiction between our concept of length and the definition of length measures on a ratio scale.

3.2.2 Examples of Length Measures

Several measures can be defined at the system or module level based on the length concept. A typical example is the depth of a hierarchy or lattice/network. Therefore, the nesting depth in a program [14] and DIT (Depth of Inheritance Tree—which is actually a hierarchy, in the general case) defined in [8] are length measures.

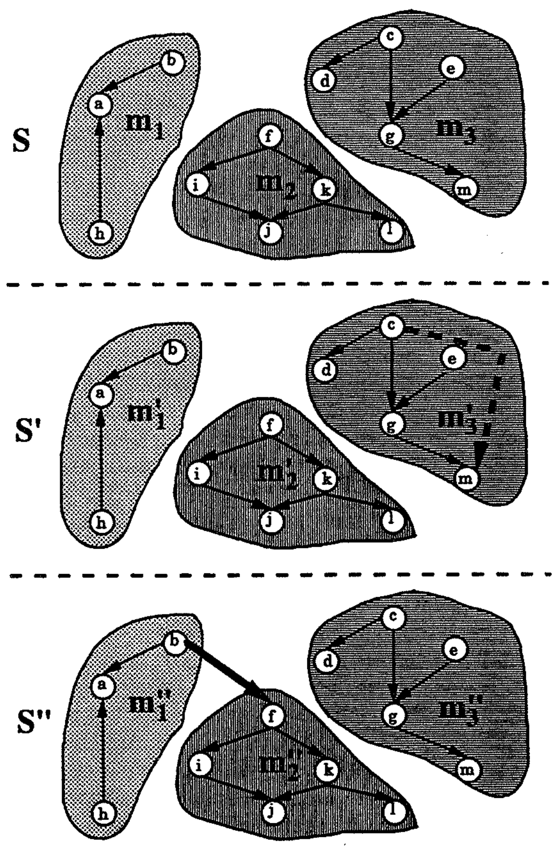


Fig. 3. Properties of length.

3.3 Complexity

3.3.1 Motivation

Complexity is a measurement concept that is considered extremely relevant to system properties. It has been studied by several researchers (see Section 4 for a comparison between our framework and the literature). It is important to note that the notion of complexity we are going to define through a set of specific properties is intentionally more restrictive than that of many researchers [16]. This will allow us to provide a precise definition of artifact complexity through a well defined set of properties. Complexity is defined as an intrisic attribute of an object and not its perceived psychological complexity as perceived by an external observer. Our intention is clearly different from the one of Curtis et al. [16] who were referring to complexity when studying the impact of software on other systems, e.g., people. This issue is further discussed below. In our framework, we expect complexity to be nonnegative (property Complexity.1) and to be null (property Complexity.2) when there are no relationships between the elements of a system. However, it could be argued that the complexity of a system whose elements are not connected to each other does not need to be necessarily null, because each element of E may have some complexity of its own. In our view, complexity is a system property that depends on the relationships between elements, and is not an isolated element's property. The complexity that an element taken in isolation may—intuitively—bring can only originate from the relationships between its "subelements." For instance, in a modular system, each module can be viewed as a "high-level element" encapsulating "subelements." However, if we want to consider the system as composed of such "high-level elements" (E), we should not "unpack" them, but only consider them and their relationships, without considering their "subelements" (E′). Otherwise, if we want to consider the contribution of the relationships between "subelements" (R′), we actually have to represent the system as $S = \langle E', R \cup R' \rangle$.

Complexity should not be sensitive to representation conventions with respect to the direction of arcs representing system relationships (property Complexity.3). A relation can be represented in either an "active" (R) or "passive" (R^{-1}) form. The system and the relationships between its elements are not affected by these two equivalent representation conventions, so a complexity measure should be insensitive to this.

Also, the complexity of a system S should be at least as much as the sum of the complexities of any collection of its modules, such that no two modules share relationships, but may only share elements (property Complexity.4). *We believe that this property is the one that most strongly differentiates complexity from the other system concepts.* Intuitively, this property may be explained by two phenomena. First, the transitive closure of R is a graph not smaller than the graph obtained as the union of the transitive closures of R′ and R″ (where R′ and R″ are contained in R). As a consequence, if any kind of *indirect* (i.e., *transitive*) relationships between elements is considered in the computation of complexity, then the complexity of S may be larger than the sum of its modules' complexities, when the modules do not share any relationship. Otherwise, they are equal. Second, merging modules may implicitly generate relationships between the elements of each modules. (e.g., definition-use relationships may be created when blocks are merged into a common system). As a consequence of the above properties, system complexity should not decrease when the set of system relationships is increased (property Complexity.4).

However, it has been argued that it is not always the case that the more relationships between the elements of a system, the more complex the system. For instance, it has been argued that adding a relationship between two elements may make the understanding of the system easier, since it clarifies the relationship between the two. This is certainly true, but we want to point out that this assertion is related to understandability (i.e., ease of understanding in terms of effort needed which is an external attribute), rather than complexity which is seen in this paper as an internal attribute [1]. Complexity is only *one* of the factors that contribute to understandability and may help predict it. There are other factors that have a strong influence on understandability, such as the amount of available context information and knowledge about a system. In the literature [17], it has been argued that the inner loop of the ShellSort algorithm, taken in isolation, is less understandable than the whole algorithm, since the role of the inner loop in the algorithm cannot be fully understood without the rest of the algorithm. This shows that understandability improves because a larger amount of context information is available, rather than because the complexity of the ShellSort algorithm is less than that of its inner loop. As another example, a relationship between two elements of a system may be added to *explicitly* state a relationship between them that was implicit or uncertain. This adds to our knowledge of the system, while, at the same time, increases complexity (according to our properties). In some cases (see above examples), the gain in context information/knowledge may overcome the increase in complexity and, as a result, may improve understandability. This stems from the fact that several phenomena concurrently affect understandability and does not mean in any way that an increase in complexity increases understandability.

Last, the complexity of a system made of disjoint modules is the sum of the complexities of the single modules (property Complexity.5). Consistent with property Complexity.4, this property is intuitively justified by the fact that the transitive closure of a graph composed of several disjoint subgraphs is equal to the union of the transitive closures of each subgraph taken in isolation. Furthermore, if two modules are put together in the same system, but they are not merged, i.e., they are still two disjoint module in this system, then no additional relationships are generated from the elements of one to the elements of the other.

The properties we define for complexity are, to a limited extent, a generalization of the properties several authors have already provided in the literature (see [5], [6], [7]) for software code complexity, usually for control flow graphs. We generalize them because we may want to use them on artifacts other than software code and on abstractions other than control flow graphs.

DEFINITION 5: Complexity. *The complexity of a system S is a function Complexity(S) that is characterized by the following properties Complexity.1-Complexity.5.* □

PROPERTY COMPLEXITY.1: Nonnegativity. The complexity of a system $S = \langle E, R \rangle$ is nonnegative

$$\text{Complexity}(S) \geq 0 \qquad (\text{Complexity.I}) \quad \square$$

PROPERTY COMPLEXITY.2: Null Value. The complexity of a system $S = \langle E, R \rangle$ is null if R is empty

$$R = \varnothing \Rightarrow \text{Complexity}(S) = 0 \qquad (\text{Complexity.II}) \quad \square$$

PROPERTY COMPLEXITY.3: Symmetry. The complexity of a system $S = \langle E, R \rangle$ does not depend on the convention chosen to represent the relationships between its elements

$$(S = \langle E, R \rangle \text{ and } S^{-1} = \langle E, R^{-1} \rangle) \Rightarrow \text{Complexity}(S) = \text{Complexity}(S^{-1}) \qquad (\text{Complexity.III}) \quad \square$$

PROPERTY COMPLEXITY.4: Module Monotonicity. The complexity of a system $S = \langle E, R \rangle$ is no less than the sum of the complexities of any two of its modules with no relationships in common

$$(S = \langle E, R\rangle \text{ and } m_1 = \langle E_{m1}, R_{m1}\rangle \text{ and } m_2 = \langle E_{m2}, R_{m2}\rangle \text{ and } m_1 \cup m_2 \subseteq S \text{ and } R_{m1} \cap R_{m2} = \emptyset) \Rightarrow \text{Complexity}(S) \geq \text{Complexity}(m_1) + \text{Complexity}(m_2) \quad (\text{Complexity.IV}) \quad \square$$

For instance, the complexity of the system shown in Fig. 4 is not smaller than the sum of the complexities of m_1 and m_2.

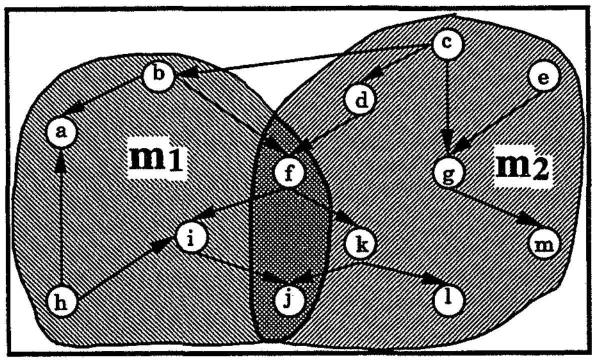


Fig. 4. Module monotonicity of complexity.

PROPERTY COMPLEXITY.5: Disjoint Module Additivity. The complexity of a system $S = \langle E, R\rangle$ composed of two disjoint modules m_1, m_2 is equal to the sum of the complexities of the two modules

$$(S = \langle E, R\rangle \text{ and } S = m_1 \cup m_2 \text{ and } m_1 \cap m_2 = \emptyset) \Rightarrow \text{Complexity}(S) = \text{Complexity}(m_1) + \text{Complexity}(m_2) \quad (\text{Complexity.V}) \quad \square$$

As a consequence of the above properties Complexity.1-Complexity.5, it can be shown that adding relationships between elements of a system does not decrease its complexity

$$(S' = \langle E, R'\rangle \text{ and } S'' = \langle E, R''\rangle \text{ and } R' \subseteq R'') \Rightarrow \text{Complexity}(S') \leq \text{Complexity}(S'') \quad (\text{Complexity.VI})$$

Properties Complexity.1-Complexity.5 hold when applying the admissible transformation of the ratio scale. Therefore, there is no contradiction between our concept of complexity and the definition of complexity measures on a ratio scale.

Comprehensive comparisons and discussions of previous work in the area of complexity properties are provided in Section 4.

3.3.2 Examples and Counterexamples of Complexity Measures

In [18], Oviedo proposed a data flow complexity measure (DF). In this case, systems are programs, modules are program blocks, elements are variable definitions or uses, and relationships are defined between the definition of a given variable and its uses. The measure in [18] is simply defined as the number of definition-use pairs in a block or a program. Property Complexity.4 holds. Given two modules (i.e., program blocks) which may only have common elements (i.e., no definition-use relationship is contained in both), the whole system (i.e., program) has a number of relationships (i.e., definition-use relationships) which is at least equal to the sum of the numbers of definition-use relationships of each module. Property Complexity.5 holds as well. The number of definition-use relationships of a system composed of two disjoint modules (i.e., blocks between which no definition-use relationship exists), is equal to the sum of the numbers of definition-use relationships of each module. As a conclusion, DF is a complexity measure according to our definition.

In [19], McCabe proposed a control flow complexity measure. Given a control flow graph $G = \langle E, R\rangle$ (which corresponds—unchanged—to a system for our framework), Cyclomatic Complexity is defined as

$$v(G) = |R| - |E| + 2p$$

where p is the number of connected components of G. Let us now check whether v(G) is a complexity measure according to our definition. It is straightforward to show that, except Complexity.4, the other properties hold. In order to check property Complexity.4, let $G = \langle E, R\rangle$ be a control flow graph and $G_1 = \langle E_1, R_1\rangle$ and $G_2 = \langle E_2, R_2\rangle$ two nondisjoint control flow subgraphs of G such that they have nodes in common but no relationships. We have to require that G_1 and G_2 be control flow subgraphs, because cyclomatic complexity is defined only for control flow graphs, i.e., graphs composed of connected components, each of which has a start node—a node with no incoming arcs—and an end node—a node with no outgoing arcs. Property Complexity.4 requires that the following inequality be true for all such G_1 and G_2

$$|R| - |E| + 2p \geq |R_1| - |E_1| + 2p_1 + |R_2| - |E_2| + 2p_2$$

i.e., $2(p_1 + p_2 - p) \leq |E_1| + |E_2| - |E|$, where p_1 and p_2 are the number of connected components in G_1 and G_2, respectively. This is not always true. For instance, consider Fig. 5. G has three elements and one connected component; G_1 and G_2 have two nodes and one connected component apiece. Therefore, the above inequality is not true in this case, and the cyclomatic number is not a complexity measure according to our definition. However, it can be shown that v(G)-p satisfies all the above complexity properties. From a practical perspective, especially in large systems, this correction does not have a significant impact on the value of the measure.

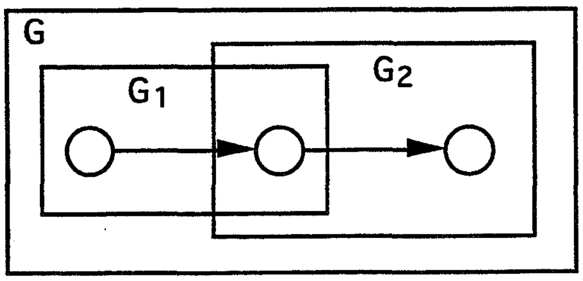


Fig. 5. Control flow graph.

Henry and Kafura [20] proposed an information flow complexity measure. In this context, elements are subprogram variables or parameters, modules are subprograms, relationships are either fan-ins or fan-outs. For a subprogram SP, the complexity is expressed as *length* × (*fan-in* × *fan-out*)^2, where fan-in and fan-out are, respectively, the local (as defined in [20]) information flows from other subprograms to SP, and from SP to other subprograms. Such local information flows can be represented as relationships between parameters/variables of SP and parameters/variables of the other subprograms. Subprograms' parameters/variables are the system elements and the subprograms' fan-in and fan-out links are the relationships. Any size measure can be used for *length* (in [20] LOC was used). The justification for multiplying *length* and (*fan-in* × *fan-out*)^2 was that "The complexity of a procedure depends on two factors: The complexity of the procedure code and the complexity of the procedure's connections to its environment." The complexity of the procedure code is taken into account by *length*; the complexity of the subprogram's connections to its environment is taken into account by (*fan-in* × *fan-out*)^2. The complexity of a system is defined as the sum of the complexities of the individual subprograms. For the measure defined above, properties Complexity.1-Complexity.4 hold. However, property Complexity.5 does not hold since, given two disjoint modules S_1 and S_2 with a measured information flow of, respectively, $length_1 \times (fan\text{-}in_1 \times fan\text{-}out_1)^2$ and $length_2 \times (fan\text{-}in_2 \times fan\text{-}out_2)^2$, the following statement is true:

$$\text{length} \times (\text{fan-in} \times \text{fan-out})^2 \geq \text{length}_1 \times (\text{fan-in}_1 \times \text{fan-out}_1)^2 + \text{length}_2 \times (\text{fan-in}_2 \times \text{fan-out}_2)^2$$

where length = $\text{length}_1 + \text{length}_2$, fan-in = $\text{fan-in}_1 + \text{fan-in}_2$, and fan-out = $\text{fan-out}_1 + \text{fan-out}_2$.

However, equality does not hold because of the exponent 2, which is not fully justified, and multiplication of fan-in and fan-out. Therefore, Henry and Kafura [20] information flow measure is not a complexity measure according to our definition. However, fan-in and fan-out taken as separate measures, without exponent 2, are complexity measures according to our definition since all the required properties hold.

Similar measures have been used in [21] and referred to as *structural complexity* (SC) and defined as:

$$SC = \frac{\sum_{i \in [1 \ldots n]} \text{fan-out}^2(\text{subroutine}_i)}{n}$$

Once again, property Complexity.5 does not hold because fan-out is squared in the formula.

A metric suite for object-oriented design is proposed in [8]. A system is an object oriented design, modules are classes, elements are either methods or instance variables (depending on the measure considered) and relationships are calls to methods or uses of instance variables by other methods. An attempt was made to validate these measures against Weyuker's properties for complexity measures, thereby implicitely implying that they were complexity measures. However, none of the measures defined by [8] is a complexity measure according to our properties:

- Weighted Methods per Class (WMC) and Number Of Children of a class (NOC) are size measures (see Section 3.1);
- Depth of Inheritance Tree (DIT) is a length measure (see Section 3.2);
- Coupling Between Object classes (CBO) is a coupling measure (see Section 3.4);
- Response For a Class (RFC) is a size and coupling measure (see Sections 3.1 and 3.5);
- Lack of COhesion in Methods (LCOM) cannot be classified in our framework. This is consistent with what was said in the introduction: Our framework does not cover all possible measurement concepts.

This is not surprising. In [8], it is shown that all of the above measures do not satisfy Weyuker's property 9, which is a weaker form of property Complexity.4 (see Section 4).

3.4 Cohesion

3.4.1 Motivation

The concept of cohesion has been used with reference to modules or modular systems. It assesses the tightness with which "related" program features are "grouped together" in systems or modules. It is assumed that the better the programmer is able to encapsulate related program features together, the more reliable and maintainable the system [14]. This assumption seems to be supported by experimental results [22]. Intuitively, we expect cohesion to be nonnegative and, more importantly, to be normalized (property Cohesion.1) so that the measure is independent of the size of the modular system or module. Moreover, if there are no internal relationships in a module or in all the modules in a system, we expect cohesion to be null (property Cohesion.2) for that module or for the system, since, as far as we know, there is no relationship between the elements and there is no evidence they should be encapsulated together. Additional internal relationships in modules cannot decrease cohesion since they are supposed to be additional evidence to encapsulate system elements together (property Cohesion.3). When two (or more) modules showing no relationships between them are merged, cohesion cannot increase because seemingly unrelated elements are encapsulated together (property Cohesion.4).

Since the cohesion (and, as we will see in Section 3.5, the coupling) of modules and entire modular systems have similar sets of properties, both will be described at the same time by using brackets and the alternation symbol '|' For instance, the notation [A | B], where A and B are phrases, will denote the fact that phrase A applies to module cohesion, and phrase B applies to entire system cohesion.

DEFINITION 6: Cohesion of a [Module | Modular System]. *The cohesion of a* [module m = $<E_m, R_m>$ *of a modular system* MS | *modular system* MS] *is a function* [Cohesion(m) | Cohesion(MS)] *characterized by the following properties Cohesion.1-Cohesion.4.* □

PROPERTY COHESION.1: Nonnegativity and Normalization. The cohesion of a [module m = $<E_m, R_m>$ of a modular system MS = <E, R, M> | modular system MS = <E, R, M>] belongs to a specified interval

$$[\text{Cohesion}(m) \in [0, \text{Max}] \mid \text{Cohesion}(MS) \in [0, \text{Max}]] \quad (\text{Cohesion.I}) \ \square$$

Normalization allows meaningful comparisons between the cohesions of different [modules | modular systems], since they all belong to the same interval.

PROPERTY COHESION.2: Null Value. The cohesion of a [module $m = <E_m, R_m>$ of a modular system $MS = <E, R, M>$ | modular system $MS = <E, R, M>$] is null if [R_m | IR] is empty

$$[R_m = \varnothing \Rightarrow \text{Cohesion}(m) = 0 \mid IR = \varnothing \Rightarrow \text{Cohesion}(MS) = 0] \quad (\text{Cohesion.II})$$

(Recall that IR is the set of intramodule relationships, defined in Definition 2.) □

If there is no intramodule relationship among the elements of a (all) module(s), then the module (system) cohesion is null.

PROPERTY COHESION.3: Monotonicity. Let $MS' = <E, R', M'>$ and $MS'' = <E, R'', M''>$ be two modular systems (with the same set of elements E) such that there exist two modules $m' = <E_m, R_{m'}>$ and $m'' = <E_m, R_{m''}>$ (with the same set of elements E_m) belonging to M' and M'', respectively, such that $R' - R_{m'} = R'' - R_{m''}$, and $R_{m'} \subseteq R_{m''}$ (which implies $IR' \subseteq IR''$). Then,

$$[\text{Cohesion}(m') \le \text{Cohesion}(m'') \mid \text{Cohesion}(MS') \le \text{Cohesion}(MS'')] \quad (\text{Cohesion.III}) \ \square$$

Adding intramodule relationships does not decrease [module | modular system] cohesion. For instance, suppose that systems S, S', and S'' in Fig. 3 are viewed as modular systems $MS = <E, R, M>$, $MS' = <E', R', M'>$, and $MS'' = <E'', R'', M''>$ (with $M = \{m_1, m_2, m_3\}$, $M' = \{m_1', m_2', m_3',\}$, and $M'' = \{m_1'', m_2'', m_3''\}$). We have [$\text{Cohesion}(m_3') \ge \text{Cohesion}(m_3)$ | $\text{Cohesion}(MS') \ge \text{Cohesion}(MS)$].

PROPERTY COHESION.4: Cohesive Modules. Let $MS' = <E, R, M'>$ and $MS'' = <E, R, M''>$ be two modular systems (with the same underlying system <E, R>) such that $M'' = M' - \{m_1', m_2'\} \cup \{m''\}$, with $m_1' \in M'$, $m_2' \in M'$, $m'' \notin M'$, and $m'' = m_1' \cup m_2'$. (The two modules m_1' and m_2' are replaced by the module m'', union of m_1' and m_2'.) If no relationships exist between the elements belonging to m_1' and m_2', i.e., $\text{InputR}(m_1') \cap \text{OutputR}(m_2') = \varnothing$ and $\text{InputR}(m_2') \cap \text{OutputR}(m_1') = \varnothing$, then

$$[\max\{\text{Cohesion}(m_1'), \text{Cohesion}(m_2')\} \ge \text{Cohesion}(m'') \mid \text{Cohesion}(MS') \ge \text{Cohesion}(MS'')] \quad (\text{Cohesion.IV}) \ \square$$

The cohesion of a [module | modular system] obtained by putting together two unrelated modules is not greater than the [maximum cohesion of the two original modules | the cohesion of the original modular system].

Properties Cohesion.1-Cohesion.4 hold when applying the admissible transformation of the ratio scale. Therefore, there is no contradiction between our concept of cohesion and the definition of cohesion measures on a ratio scale.

3.4.2 Examples of Cohesion Measures

In [22], cohesion measures for high-level design are defined and validated, at both the abstract data type (module) and system (program) levels. For brevity's sake, the term software part here denotes either a module or a program. A high-level design is seen as a collection of modules, each of which exports and imports constants, types, variables, and procedures/functions. A widely accepted software engineering principle prescribes that each module be highly cohesive, i.e., its elements be tightly related to each other. [22] focuses on investigating whether high cohesion values are related to lower error-proneness, due to the fact that the changes required by a change in a module are confined in a well-encapsulated part of the overall program. To this end, the exported feature A is said to interact with feature B if the change of one of A's definitions or uses may require a change in one of B's definitions or uses.

In the approach of the present paper, each feature exported by a module is an element of the system, and the interactions between them are the relationships between elements. A module according to [22] is represented by a module according to the definition of the present paper. At high-level design time, not all interactions between the features of a module are known, since the features may interact in the body of a module, and not in its visible part. Given a software part sp, three cohesion measures NRCI(sp), PRCI(sp), and ORCI(sp) (respectively, Neutral, Pessimistic, and Optimistic Ratio of Cohesive Interactions) are defined for software as follows

$$\text{NRCI}(sp) = \frac{|SDD(sp)| + |K(sp)|}{|SDD(sp)| + |M(sp)| + |SSR(sp)| - |U(sp)|}$$

$$\text{PRCI}(sp) = \frac{|SDD(sp)| + |CI(sp)|}{|SDD(sp)| + |M(sp)| + |SSR(sp)|}$$

$$\text{ORCI}(sp) = \frac{|SDD(sp)| + |K(sp)| + |U(sp)|}{|SDD(sp)| + |M(sp)| + |SSR(sp)|}$$

where

- *M(sp)* is the set of all possible intramodule interactions between the features exported by each module of the software part sp (intermodule interactions are not considered cohesive; they may contribute to coupling, instead).
- *K(sp)* is the set of *known interactions* at high-level design time between the features exported by each module of the software part sp.
- *U(sp)* is the set of *unknown interactions* at high-level design time between the features exported by each module of the software part sp.
- *SDD(sp)* will denote the set of modules of *sp* that only contain a single data declaration and no subroutines (even though their sets of potential interactions are empty, these modules are highly cohesive, as far as our notion of cohesion—related to abstract data types—is concerned).

- SSR(sp) will denote the set of subroutines belonging to modules that only contain subroutines (these modules are not cohesive, as far as our notion of cohesion is concerned).

Measures NRCI, PRCI, and ORCI satisfy the above properties Cohesion.1-Cohesion.4.

Other examples of cohesion measures can be found in [23], where new functional cohesion measures are introduced. Given a procedure, function, or main program, only *data tokens* (i.e., the occurrence of a definition or use of a variable or a constant) are taken into account. The *data slice* for a data token is the sequence of all those data tokens in the program that can influence the statement in which the data token appears, or can be influenced by that statement. Being a sequence, a data slice is ordered: It lists its data tokens in order of appearance in the procedure, function or main program. If more than one data slice exists, some data tokens may belong to more than one data slice: these are called *glue tokens*. A subset of the glue tokens may belong to all data slices: These are called *super-glue tokens*. Functional cohesion measures are defined based on data tokens, glue tokens, and super-glue tokens. This approach can be represented in our framework as follows. A data token is an element of the system, and a data slice is represented as a sequence of nodes and arcs. The resulting graph is a Directed Acyclic Graph, which represents a module. ([23] introduces functional cohesion measures for single procedures, functions, or main programs.) Given a procedure, function, or main program p, the following measures SFC(p) (Strong Functional Cohesion), WFC(p) (Weak Functional Cohesion), and A(p) (adhesiveness) are introduced

$$\mathrm{SFC(p)} = \frac{\#\,\mathrm{SuperGlueTokens}}{\#\,\mathrm{AllTokens}}$$

$$\mathrm{WFC(p)} = \frac{\#\,\mathrm{GlueTokens}}{\#\,\mathrm{AllTokens}}$$

$$\mathrm{A(p)} = \frac{\sum_{\mathrm{GT} \in \mathrm{GlueTokens}} \#\,\mathrm{SlicesContainingGlueTokenGT}}{\#\,\mathrm{AllTokens}.\#\,\mathrm{DataSlices}}$$

It can be shown that these measures satisfy the above properties Cohesion.1-Cohesion.4.

3.5 Coupling

3.5.1 Motivation

The concept of coupling has been used with reference to modules or modular systems. Intuitively, it captures the amount of relationship between the elements belonging to different modules of a system. Given a module m, two kinds of coupling can be defined: inbound coupling and outbound coupling. The former captures the amount of relationships from elements outside m to elements inside m; the latter the amount of relationships from elements inside m to elements outside m.

We expect coupling to be nonnegative (property Coupling.1), and null when there are no relationships among modules (property Coupling.2). When additional relationships are created across modules, we expect coupling not to decrease since these modules become more interdependent (property Coupling.3). Merging modules can only decrease coupling since there may exist relationships among them and therefore, intermodule relationships may have disappeared (property Coupling.4, property Coupling.5).

In what follows, when referring to module coupling, we will use the word coupling to denote either inbound or outbound coupling, and OuterR(m) to denote either InputR(m) or OutputR(m).

DEFINITION 7: Coupling of a [Module | Modular System]. *The coupling of a* [module m = <E_m, R_m> *of a modular system* MS | *modular system* MS] *is a function* [Coupling(m) | Coupling(MS)] *characterized by the following properties Coupling.1-Coupling.5.* □

PROPERTY COUPLING.1: Nonnegativity. The coupling of a [module m = <E_m, R_m> of a modular system | modular system MS] is nonnegative

$$[\mathrm{Coupling(m)} \geq 0 \;\mid\; \mathrm{Coupling(MS)} \geq 0] \quad \text{(Coupling.I)} \;\square$$

PROPERTY COUPLING.2: Null Value. The coupling of a [module m = <E_m, R_m> of a modular system | modular system MS = <E, R, M>] is null if [OuterR(m) | R – IR] is empty

$$[\mathrm{OuterR(m)} = \varnothing \Rightarrow \mathrm{Coupling(m)} = 0 \;\mid\; R - IR = \varnothing \Rightarrow \mathrm{Coupling(MS)} = 0] \quad \text{(Coupling.II)} \;\square$$

PROPERTY COUPLING.3: Monotonicity. Let MS′ = <E, R′, M′> and MS″ = <E, R″, M"> be two modular systems (with the same set of elements E) such that there exist two modules m′ ∈ M′, m″ ∈ M″ such that R′ – OuterR(m′) = R″ – OuterR(m″), and OuterR(m′) ⊆ OuterR(m″). Then,

$$[\mathrm{Coupling(m')} \leq \mathrm{Coupling(m'')} \;\mid\; \mathrm{Coupling(MS')} \leq \mathrm{Coupling(MS'')}] \quad \text{(Coupling.III)} \;\square$$

Adding intermodule relationships does not decrease coupling. For instance, if systems S, and S″ in Fig. 3 are viewed as modular systems (see Section 3.4), we have [Coupling(m_1'') ≥ Coupling(m_1) | Cohesion(MS″) ≥ Cohesion(MS)].

PROPERTY COUPLING.4: Merging of Modules. Let MS′ = <E′, R′, M′> and MS″ = <E″, R″, M″> be two modular systems such that E′ = E″, R′ = R″, and M″ = M′ – {m'_1, m'_2} ∪ {m″}, where m'_1 = <$E_{m'1}$, $R_{m'1}$>, m'_2 = <$E_{m'2}$, $R_{m'2}$>, and m″ = <$E_{m''}$, $R_{m''}$>, with m'_1 ∈ M′, m'_2 ∈ M′, m″ ∉ M′, and $E_{m''} = E_{m'1} \cup E_{m'2}$ and $R_{m''} = R_{m'1} \cup R_{m'2}$. (The two modules m'_1 and m'_2 are replaced by the module m″, whose elements and relationships are the union of those of m'_1 and m'_2.) Then

$$[\mathrm{Coupling}(m'_1) + \mathrm{Coupling}(m'_2) \geq \mathrm{Coupling(m'')} \;\mid\; \mathrm{Coupling(MS')} \geq \mathrm{Coupling(MS'')}] \quad \text{(Coupling.IV)} \;\square$$

The coupling of a [module | modular system] obtained by merging two modules is not greater than the [sum of the couplings of the two original modules | coupling of the original modular system], since the two modules may have common intermodule relationships. For instance, suppose that the modular system MS_{12} in Fig. 6 is obtained from the modular system MS in Fig. 2 by merging modules m_1 and m_2 into module m_{12}. Then, we have [Coupling(m_1) + Coupling(m_2) $\geq$ Coupling(m_{12}) | Coupling(MS) $\geq$ Coupling(MS_{12})].

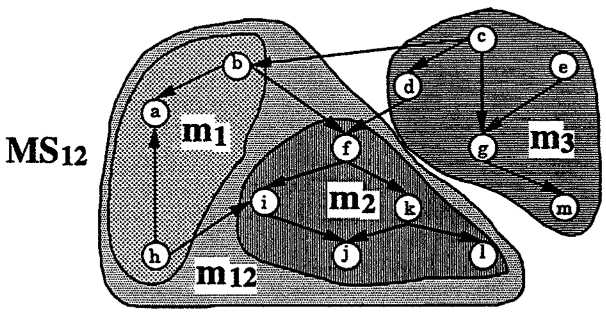


Fig. 6. The effect of merging modules on coupling.

PROPERTY COUPLING.5: Disjoint Module Additivity. Let $MS' = <E, R, M'>$ and $MS'' = <E, R, M''>$ be two modular systems (with the same underlying system $<E, R>$) such that $M'' = M' - \{m'_1, m'_2\} \cup \{m''\}$, with $m'_1 \in M', m'_2 \in M'$, $m'' \notin M'$, and $m'' = m''_1 \cup m'_2$. (The two modules m'_1 and m'_2 are replaced by the module m'', union of m'_1 and m'_2) If no relationships exist between the elements belonging to m'_1 and m'_2, i.e., $\text{InputR}(m'_1) \cap \text{OutputR}(m'_2) = \varnothing$ and $\text{InputR}(m'_2) \cap \text{OutputR}(m'_1) = \varnothing$, then

$$[\text{Coupling}(m'_1) + \text{Coupling}(m'_2) = \text{Coupling}(m'') \mid \text{Coupling}(MS') = \text{Coupling}(MS'')] \qquad (\text{Coupling.V}) \quad \square$$

The coupling of a [module | modular system] obtained by merging two unrelated modules is equal to the [sum of the couplings of the two original modules | coupling of the original modular system].

Properties Coupling.1-Coupling.5 hold when applying the admissible transformations of the ratio scale. Therefore, there is no contradiction between our concept of coupling and the definition of coupling measures on a ratio scale.

3.5.2 Examples and Counterexamples of Coupling Measures

Fenton has defined an ordinal coupling measure between pairs of subroutines [14] as follows:

$$C(S, S') = i + \frac{n}{n+1}$$

where i is the number corresponding to the worst coupling type (according to Myers' ordinal scale [14]) and n the number of interconnections between S and S', i.e., global variables and formal parameters. In this case, systems are programs, modules are subroutines, elements are formal parameters and global variables. If coupling for the whole system is defined as the sum of coupling values between all subroutine pairs, properties Coupling.1-Coupling.5 hold for this measures, and we label it as a coupling measure. However, Fenton proposes to calculate the median of all the pair values as a system coupling measure. In this case, property Coupling.3 does not hold since the median may decrease when intermodule relationships are added. Similarly for Coupling.4, when subroutines are merged and intermodule relationships are lost, the median may increase. Therefore, the system coupling measure proposed by Fenton is not a coupling measure according to our definitions.

In [22], coupling measures for high-level design are defined and validated, at both the module (abstract data type) and system (program) levels. They are based on the notion of interaction introduced in the examples of Section 3.4. Import Coupling of a module m is defined as the extent to which m depends on imported external data declarations. Similarly, export coupling of m is defined as the extent to which m's data declarations affect the other data declarations in the system. At the system level, coupling is the extent to which the modules are related to each other. Given a module m, Import Coupling of m (denoted by IC(m)) is the number of interactions between data declarations external to m and the data declarations within m. Given a module m, Export Coupling of m (denoted by EC(m)) is the number of interactions between the data declarations within m and the data declarations external to m. As shown in [22], our coupling properties hold for these measures.

Coupling Between Object classes (CBO) of a class is defined in [8] as the number of other classes to which it is coupled. It is a coupling measure. Properties Coupling.1 and Coupling.2 are obviously satisfied. Property Coupling.3 is satisfied, since CBO cannot decrease by adding one more relationship between features belonging to different classes (i.e., one class uses one more method or instance variable belonging to another class). Property Coupling.4 is satisfied: CBO can only remain constant or decrease when two classes are grouped into one. Property Coupling.5 is also satisfied.

Response For a Class (RFC) [8] is a size and a coupling measure at the same time (see Section 3.1). Methods are elements, calls are relationships, classes are modules. Coupling.3 holds, since adding outside method calls to a class can only increase RFC and Coupling.4 holds because merging classes does not change RFCs value since RFC does not distinguish between inside and outside method calls. Similarly, when there are no calls between the classes' methods, Coupling.5 holds. This result is to be expected since RFC is the result of the addition of two terms: the number of methods in the class, a size measure, and the number of methods called, a coupling measure.

3.6 Concept Properties within the Context of Measurement Theory

The properties we defined in the previous subsections must be discussed in the context of measurement theory. For the reader's convenience, we now report (in italic) the basic definitions and notation of measurement theory, as defined in [11, pp. 40-51], based on [24].

A relational system A [24] is an ordered tuple (A, R1, ⋯, Rn, o1, ⋯, om) where A is a nonempty set of objects, the Ri, i = 1, ⋯, n are ki-ary relations on A and the oj, j = 1, ⋯, m are closed binary operations. For measurement we consider two relational systems: the empirical and formal relational systems.

Empirical Relational System:

A	*= (A, R1, ⋯, Rn, o1, ⋯ om).*
A	*is a nonempty set of empirical objects that are to be measured (in our case program texts, flowgraphs, etc.).*
Ri	*are ki-ary empirical relations on A with i = 1, ⋯, n. For example, the empirical relation "equal or more complex."*
oj	*are binary operations on the empirical objects A that are to be measured (for example a concatenation of control flowgraphs) with j = 1, ⋯, m.*

The empirical relational system describes the part of reality on which measurement is carried out (via the set of objects A) and our empirical knowledge on the objects' attributes we want to measure (via the collection of empirical relations Ri's). Depending on the attributes we want to measure, different relations are used. For instance, if we are interested in program length, we may want to use the relation "longer than" (e.g., "program P1 is longer than program P2"); if we are interested in program complexity, we may want to use the relation "more complex than" (e.g., "program P3 is more complex than program P4"). Binary operations may be seen as a special case of ternary relation between objects. For instance, suppose that o1 is the concatenation operation between two programs. We may see it as a relation Concat(Program1, Program2, Program3), where Program3 is obtained as the concatenation of Program1 and Program2, i.e., Program3 = Program1 o1 Program2. It is important to notice that an empirical relational system does not contain any reference to measures or numbers. Only "qualitative" statements are asserted, based on our understanding of the attribute. These statements are then translated into relations that belong to a formal relational system, as explained below.

Formal Relational System:

B	*= (B, S1, ⋯, Sn, •1, ⋯, •m).*
B	*is a nonempty set of formal objects, for example numbers or vectors.*
Si	*are ki-ary relations on B such as "greater than" or "equal or greater."*
•j	*are closed binary operations B such as the addition or multiplication.*

The formal relational system describes (via the set B) the domains of the measures for the studied objects' attributes. For instance, these may be integer numbers, real numbers, vectors of integer, and/or real numbers, etc. A formal relational system also describes (via the collection of relations Sis) the relations of interest between the measures. The link between the empirical relational system and the formal relational system is provided by measures, as follows.

DEFINITION 4.1: Measure μ. *A measure μ is a mapping $\mu : A \rightarrow B$ which yields for every empirical object $a \in A$ a formal object (measurement value) $\mu(a) \in B$.*

Every object *a* of A is mapped into a value of B, i.e., it is measured according to measure $\mu(a)$. Every empirical relation Ri is mapped into a formal relation Si. For instance, the relation "more complex than" between two programs is mapped into the relation ">" between the complexity measures of two programs. The formal relations must preserve the meaning of the empirical statements. For instance, suppose that R1 is the empirical relation "more complex than," S1 is the formal relation ">," and μ is a complexity measure. Then, we must have that program P1 is more complex than program P2 if and only if $\mu(P1) > \mu(P2)$.

Within the context defined above, concept properties may be seen as properties characterizing, for each measurement concept (i.e., family of measures), formal relational systems. These properties are preserved from the corresponding empirical relational systems when formal relational systems are derived. However, our set of properties for a concept does not fully characterize a formal relational system since, for a particular measurement application, many properties will be specific to the working environment and experience of the modeler (captured in the empirical relational system).

For example, if we take Property Size.3 (Module additivity),

$$(m_1 \subseteq S \text{ and } m_2 \subseteq S \text{ and } E = E_{m1} \cup E_{m2} \text{ and } E_{m1} \cap E_{m2} = \varnothing) \Rightarrow Size(S) = Size(m_1) + Size(m_2)$$

we can see that this property is expressed in terms of a measure *Size* and arithmetic operators such as "+". Clearly, our properties are formal relational system properties. However, these properties are derived from a corresponding empirical relational system. For instance, the above property can be derived from the following property of the empirical relational system:

$$(m_1 \subseteq S \text{ and } m_2 \subseteq S \text{ and } E = E_{m1} \cup E_{m2} \text{ and } E_{m1} \cap E_{m2} = \varnothing) \Rightarrow S \approx m_1 \oplus m_2)$$

where "⊕" could be a concatenation operation between two modules and "≈" an equivalence relation (i.e., "same size as") between two objects of the empirical relational system. In this context, any valid formal relational system is supposed to preserve properties such as the one above.

The concept properties defined above may be seen as properties to consider (i.e., consciously accept or reject) and therefore as guidelines when building empirical and formal relational systems and deriving measures of product internal attributes. The properties above were defined for the formal relational systems for two reasons:

- We characterize families of measures (i.e., "measurement concepts") and therefore we want those properties to be expressed in terms of those measures.
- Defining both the empirical and formal relational systems' properties would be redundant for the purpose of this paper.

To conclude and as discussed above, these properties are intuitive and convenient and provide a self-consistent formal framework to build measurement models.

3.7 Comparison of Concept Properties

We want to summarize the important differences and similarities between the system concepts introduced in this paper. Table 1 uses only criteria that can be compared across the concepts of size, length, complexity, cohesion, and coupling. First, it is important to recall that coupling and cohesion are only defined in the context of modular systems, whereas size, length and complexity are defined for all systems.

Second, the concepts appear to have the null value (second column) and monotonicity (third column) properties based on different sets. The behavior of a measure with respect to variations in such sets characterizes the nature of the measure itself, i.e., the concept(s) it captures. As RFC, defined in [8], shows (see Sections 3.1 and 3.5), the same measure may satisfy the sets of properties associated with different concepts. As a matter of fact, similar sets of properties associated with different concepts are not contradictory.

Third, when systems are made of disjoint modules, size, complexity and coupling are additive (properties Size.3, Complexity.5, and Coupling.5). Cohesion and length are not additive.

TABLE 1
COMPARISON OF CONCEPT PROPERTIES

Concepts\ Properties	Null Value	Monotonicity	Additivity
Size	$E = \emptyset$	E	Yes
Length	$E = \emptyset$	R	No
Complexity	$R = \emptyset$	R	Yes
System Cohesion	$IR = \emptyset$	IR	No
System Coupling	$R - IR = \emptyset$	R – IR	Yes

This summary shows that these concepts are really different with respect to basic properties. Therefore, it appears that desirable properties are likely to vary from one measurement concept to another.

4 COMPARISON WITH RELATED WORK

We mainly compare our approach with the other approaches for defining sets of properties for software complexity measures, because they have been studied more extensively and thoroughly than other kinds of measures. In addition, we compare our approach with the axioms introduced by Fenton and Melton [25] for software coupling measures. As already mentioned, our approach generalizes previous work on properties for defining complexity measures. Unlike previous approaches, it is not constrained to deal with software code only, but, because of its generality, can be applied to other artifacts produced during the software lifecycle, namely, software specifications and designs. Moreover, it is not defined based on some control flow operations, like sequencing or nesting, but on a general representation, i.e., a graph.

4.1 Weyuker[4]

Weyuker's work [7] is one of the first attempts to formalize the fuzzy concept of program complexity. This work has been discussed by many authors [8], [14], [5], [6], [11] and is still a point of reference and comparison for anyone investigating the topic of software complexity.

To make Weyuker's properties comparable with ours, we will assume that a program according to Weyuker is a system according to our definition; a program body is a module of a system. A whole program is built by combining program bodies, by means of sequential, conditional, and iterative constructs (plus the program and output statements, which can be seen as "special" program bodies), and, correspondingly, a system can be built from its constituent modules. Since some of Weyuker's properties are based on the sequencing between pairs of program bodies P and Q, we provide more details about the representation of sequencing in our framework. Sequencing of program bodies P and Q is obtained via the composition operation (P; Q). Correspondingly, if $S_P = <E_P, R_P>$ and $S_Q = <E_Q, R_Q>$ are the modules representing the two program bodies P and Q[5], then, we will denote the representation of P; Q as $S_{P;Q} = <E_{P;Q}, R_{P;Q}>$. In what follows, we will assume that $E_{P;Q} = E_P \cup E_Q$ and $R_{P;Q} \subseteq R_P \cup R_Q$, i.e., the representation of the composition of two program bodies contains the elements of the representation of each program body, and at least contains all the relationships belonging to each of the representations of program bodies. In other words, S_P and S_Q are modules of $S_{P;Q}$.

W1: *A complexity measure must not be "too coarse" (1).*

$\exists\ S_P, S_Q$ Complexity(S_P) $\neq$ Complexity(S_Q)

W2: *A complexity measure must not be "too coarse" (2).* Given the nonnegative number c, there are only finitely many systems of complexity c.

W3: *A complexity measure must not be "too fine."* There are distinct systems S_P and S_Q such that Complexity(S_P) = Complexity(S_Q).

W4: *Functionality.* There is no one-to-one correspondence between functionality and complexity $\exists\ S_P, S_Q$ P, and Q are functionally equivalent and Complexity(S_P) $\neq$ Complexity(S_Q).

W5: *Monotonicity with respect to composition.*

$\forall\ S_P, S_Q$

Complexity(S_P) $\leq$ Complexity($S_{P;Q}$) and Complexity(S_Q) $\leq$ Complexity($S_{P;Q}$)

W6: *The contribution of a module in terms of the overall system complexity may depend on the rest of the system.*

(a) $\exists\ S_P, S_Q, S_T$ Complexity(S_P) = Complexity(S_Q) and Complexity($S_{P;T}$) $\neq$ Complexity($S_{Q;T}$)

(b) $\exists\ S_P, S_Q, S_T$ Complexity(S_P) = Complexity(S_Q) and Complexity($S_{T;P}$) $\neq$ Complexity($S_{T;Q}$)

4. We will list properties/axioms by the initial of the proponents. So, Weyuker's properties will be referred to as W1, W2, ..., W9, Tian and Zelkowitz's as TZ1 to TZ5, and Lakshmanian et al.'s as L1 to L9.

5. In what follows, we will use the notation $S_P = <E_P, R_P>$ to denote the representation of program body P.

W7: *A complexity measure is sensitive to the permutation of statements.*

$\exists$ S_P, S_Q Q is formed by permuting the order of statements of P and Complexity(S_P) $\neq$ Complexity(S_Q)

W8: *Renaming*. If P is a renaming of Q, then Complexity(S_P) = Complexity(S_Q).

W9: *Module monotonicity*. $\exists$ S_P, S_Q Complexity(S_P) + Complexity(S_Q) $\leq$ Complexity($S_{P,Q}$)

4.4.1 Analysis of Weyuker's properties

W1, W2, W3, W4, W8: These are not implied by our properties, but they do not contradict any of them, so they can be added to our set, if desired. However, we think that these properties are general to all syntactically-based product measures and do not appear useful in our framework to differentiate concepts.

W5: This is implied by our properties, as shown by inequality (Complexity.VI), since S_P and S_Q are modules of $S_{P,Q}$.

W6, W7: These properties are not implied by the above properties Complexity.1-Complexity.5. However, they show a very important and delicate point in the context of complexity measure definition.

By assuming properties W6(a) and W6(b) to be false, one forces all complexity measures to be strongly related to control flow, since this would exclude that the composition of two program bodies may yield additional relationships between elements (e.g., data declarations) of the two program bodies. If properties W6(a) and W6(b) are assumed true, one forces all complexity measures to be sensitive to at least one other kind of additional relationship.

Similarly, W7 states that the order of the statements, and therefore the control flow, should have an impact on all complexity measures. By assuming property W7 to be false, one forces all complexity measures to be insensitive to the ordering of statements. If property W7 is assumed true, one forces all complexity measures to be somehow sensitive to the ordering of statements, which may not always be useful.

W8: We analyze this property again, to better explain the relationship between complexity and understandability. According to this property, renaming does not affect complexity. However, it is a fact that renaming program variables by absurd or misleading names greatly impairs understandability. This shows that other factors, besides complexity, affect understandability and the other external qualities of software that are affected by complexity.

As for properties W1-W8, our approach is somewhat more liberal than Weyuker's. For instance, the constant null function is an acceptable complexity measure according to our properties, while it is not acceptable according to Weyuker's properties. It is evident that the usefulness of such a complexity measure is questionable. We think that properties should be used to check whether a measure actually addresses a given concept (e.g., complexity). However, given any set of properties, it is almost always possible to build a measure that satisfies them, but is of no practical interest (see [12]). At any rate, this is not a sensible reason to reject a set of properties associated with a concept. Rather, measures that satisfy a set of properties must be later assessed with regard to their usefulness.

W9: This is probably the most controversial property. The above properties Complexity.1-Complexity.5 imply it. Actually, our properties imply the stronger form of W9, the unnumbered property following W9 in Weyuker's paper [7] (see also [26])

$\forall$ S_P, S_Q Complexity(S_P) + Complexity(S_Q) $\leq$ Complexity($S_{P,Q}$)

Weyuker rejects it on the basis that it might lead to contradictions: she argues that the effort needed to implement or understand the composition of a program body P with itself, is probably not twice as much as the effort needed for P alone. Our point is that complexity is not the only factor to be taken into account when evaluating the effort needed to implement or understand a program, nor is it proven that this effort is in any way "proportional" to product complexity.

4.2 Fenton

In addition to Weyuker's work, Fenton [1] shows that, based on measurement-theoretic mathematical grounds, there is no chance that a general measure for software complexity will ever be found, nor even for control flow complexity, i.e., a more specific kind of complexity. We totally agree with that. By no means do we aim at defining a single complexity measure, which captures all kinds of complexity in a software artifact. Instead, our set of properties define constraints for any specific complexity measure, whatever facet of complexity it addresses.

Fenton and Melton [25] introduced two axioms that they believe should hold for coupling measures. Both axioms assume that coupling is a measure of connectivity of a system represented by its module design chart (or structure chart). The first axiom is similar to our monotonicity property (Coupling.3). It states that if the only difference between two module design charts D and D' is an extra interconnection in D,' then the coupling of D' is higher than the coupling of D. The second axiom basically states that system coupling should be independent from the number of modules in the system. If a module is added and shows the same level of pairwise coupling as the already existing modules, then the coupling of the system remains constant. According to our properties, coupling is seen as a measure which is to a certain extent dependent on the number of modules in the system and we therefore do not have any equivalent axiom. This shows that the sets of properties that can be defined above are, to some extent, subjective.

4.3 Zuse

In his article in the *Encyclopaedia of Software Engineering* [27, pp. 131-165], Zuse applies a measurement-theoretic approach to complexity measures. The focus is on the conditions that should be satisfied by empirical relational systems in order to provide them with additive ratio scale measures. This class of measures is a subset of ratio scale measures, characterized by the additivity property

(Theorems 2 and 3 of [27]). Given the set P of flowgraphs and a binary operation * between flowgraphs (e.g., concatenation), additive ratio scale complexity measures are such that, for each pair of flowgraphs P1, P2,

$$\text{Complexity}(P1 * P2) = \text{Complexity}(P1) + \text{Complexity}(P2)$$

This property shows that a different concept of complexity is defined by Zuse, with respect to that defined by Weyuker's (W9) and our properties (Complexity.4). It is our belief that, by requiring that complexity measures be additive, important aspects of complexity may not be fully captured, and complexity measures actually become quite similar to size measures. Considering complexity as additive means that, when two modules are put together to form a new system, no additional dependencies between the elements of the modules should be taken into account in the computation of the system complexity. We believe this is a very questionable assumption for product complexity [28].

4.4 Tian and Zelkowitz

Tian and Zelkowitz [6] have provided axioms (necessary properties) for complexity measures and a classification scheme based on additional program characteristics that identify important measure categories. In the approach, programs are represented by means of their abstract syntax trees (e.g., parse trees). To translate this representation into our framework, we will assume that the whole program, represented by the entire tree, is a system, and that any part of a program represented by a subtree is a module.

TZ1: Systems with identical functionality are comparable, i.e., there is an order relation between them with respect to complexity.

TZ2: A system is comparable with its module(s).

TZ3: Given a system S_Q and any module S_P whose root, in the abstract tree representation, is "far enough" from the root of S_Q, then S_P is not more complex than S_Q. In other words, "small" modules of a system are no more complex than the system.

TZ4: If an intuitive complexity order relation exists between two systems, it must be preserved by the complexity measure (it is a weakened form of the representation condition of Measurement Theory [14]).

TZ5: Measures must not be too coarse and must show sufficient variability.

TZ1, TZ2, TZ5 do not differentiate software characteristics (concepts) and can be used for all syntactic product measures. **TZ3** can be derived from our set of properties. **TZ4** captures the basic purpose behind the definition of all measures: preserving an intuitive order on a set of software artifacts [17].

The additional set of properties which is presented in [6] is used to define a measure classification system. It determines whether or not a measure is based exclusively on the abstract syntax tree of the program, whether it is sensitive to renaming, whether it is sensitive to the context of definition or use of the measured program, whether it is determined entirely by the performed program operations regardless of their order and organization, and whether concatenation of programs always contribute positively toward the composite program complexity (i.e., system monotonicity).

Some of these properties are related to the properties defined in this paper and we believe they are characteristic properties of distinct system concepts (e.g., system monotonicity). Others do not differentiate the various concepts associated with syntactically-based measures (e.g., renaming).

4.5 Lakshmanian et al.

Lakshmanian et al. [5] have attempted to define necessary properties for software complexity measures based on control flow graphs. In order to make these properties comparable to ours, we will use a notation similar to the one used to introduce Weyuker's properties. A program according to Lakshmanian et al. (represented by a control flow graph) is a system according to our definition, and a program segment is a module. In addition to sequencing, these properties use the nesting program construct denoted as @. "A program segment Z is said to be obtained by nesting [program segment] Y at the control location i in [program segment] X (denoted by $Y@X_i$) if the program segment X has at least one conditional branch, and if Y is embedded at location i in X in such a way that there exists at least one control flow path in the combined code Z that completely skips Y." "The notation Y@X refers to any nesting of Y in X if the specific location in X at which Y is embedded is immaterial."

In what follows, X, Y, Z will denote programs or program segments; S_X, S_Y, S_Z will denote the corresponding systems or modules according to our definition. Lakshmanian et al. [5] introduce nine properties. However, only five out of them can be considered basic, since the remaining four can be derived from them. Therefore, below we will only discuss the compatibility of the basic properties with respect to our properties.

L1: Nonnegativity.

L1(a): Null value.
If the program only contains sequential code (referred to as a basic block B) then

$$\text{Complexity}(S_B) = 0$$

L1(b): Positivity.
If the program X is not a basic block, then

$$\text{Complexity}(S_X) > 0 \quad \square$$

Property L1 does not contradict any of our properties (in particular, Complexity 1 and Complexity 2).

L5: Additivity under sequencing.

$$\text{Complexity}(S_{X;Y}) = \text{Complexity}(S_Y) + \text{Complexity}(S_X) \quad \square$$

This property does not contradict properties Complexity.4 and Complexity.5, where the equality sign is allowed. By requiring that complexity be additive under sequencing, Lakshmanian et al. take a viewpoint which is very similar to that of Zuse.

L6: Functional independence under nesting.
Adding a basic block B to a system X through nesting does not increase its complexity

$$\text{Complexity}(S_{B@X}) = \text{Complexity}(S_X) \qquad \Box$$

L7: Monotonicity under nesting.

$$\text{Complexity}(S_{Y@X}) < \text{Complexity}(S_{Z@X}) \quad \text{if } \text{Complexity}(S_Y) < \text{Complexity}(S_Z) \qquad \Box$$

These properties are compatible with our properties.

L9: Sensitivity to nesting.

$$\text{Complexity}(S_{X;Y}) < \text{Complexity}(S_{Y@X}) \text{ if } \text{Complexity}(S_Y) > 0 \qquad \Box$$

This property does not contradict our properties.

In conclusion, none of the above properties contradicts our properties. However, the scope of these properties is limited to the sequencing and nesting of control flow graphs, and therefore to the study of control flow complexity.

As for the other properties, we now show how they can be derived from L1, L5, L6, L7, and L9.

L2: Functional independence under sequencing.

$$\text{Complexity}(S_{X;B}) = \text{Complexity}(S_X)$$

This property follows from L5 (first equality below) and L1 (second equality below):

$$\text{Complexity}(S_{X;B}) = \text{Complexity}(S_X) + \text{Complexity}(S_B) = \text{Complexity}(S_X) \qquad \Box$$

L3: Symmetry under sequencing.

$$\text{Complexity}(S_{X;Y}) = \text{Complexity}(S_{Y;X})$$

This property follows from L5 (both equalities)

$$\text{Complexity}(S_{X;Y}) = \text{Complexity}(S_X) + \text{Complexity}(S_Y) = \text{Complexity}(S_{Y;X}) \qquad \Box$$

L4: Monotonicity under sequencing.

$$\text{Complexity}(S_{X;Y}) < \text{Complexity}(S_{X;Z}) \quad \text{if } \text{Complexity}(S_Y) < \text{Complexity}(S_Z)$$

$$\text{Complexity}(S_{X;Y}) = \text{Complexity}(S_{X;Z}) \quad \text{if } \text{Complexity}(S_Y) = \text{Complexity}(S_Z)$$

This property follows from L5:

if $\text{Complexity}(S_Y) < \text{Complexity}(S_Z)$, then
$\text{Complexity}(S_{X;Y}) = \text{Complexity}(S_X) + \text{Complexity}(S_Y)$
$< \text{Complexity}(S_X) + \text{Complexity}(S_Z) = \text{Complexity}(S_{X;Z})$

if $\text{Complexity}(S_Y) = \text{Complexity}(S_Z)$, then
$\text{Complexity}(S_{X;Y}) = \text{Complexity}(S_X) + \text{Complexity}(S_Y)$
$= \text{Complexity}(S_X) + \text{Complexity}(S_Z) = \text{Complexity}(S_{X;Z})$ $\Box$

L8: Monotonicity under nesting.

$$\text{Complexity}(S_Y) < \text{Complexity}(S_{Y@X})$$

This property follows from L1 (first inequality below, since $\text{Complexity}(S_X) > 0$—X cannot be a basic block), L5 (equality below) and L9 (second inequality below)

$$\text{Complexity}(S_Y) < \text{Complexity}(S_X) + \text{Complexity}(S_Y) = \text{Complexity}(S_{X;Y}) < \text{Complexity}(S_{Y@X}) \qquad \Box$$

In conclusion, certain properties covered by some of the works mentioned above (Weyuker, and Tian and Zelkowitz) are general and characterize all syntactically based measures. As such, they are not covered by our framework. On the other hand, Lakshmanian et al. provide a more specialized framework focusing on control flow complexity and some of their properties are not covered, because specific of their context of study, in our framework. Other properties are weaker (e.g., W9) than some of the properties we propose and this will ultimately be a matter of choice and a consensus in the software engineering community will have to be reached.

5 Conclusion and Directions for Future Work

In order to provide some guidelines for the analyst in charge of defining product measures, we propose a framework for software measurement where various software measurement concepts are distinguished and their specific properties defined in a generic manner. Such a framework is, by its very nature, somewhat subjective and there are possible alternatives to it. However, it is a practical framework since the properties we capture are, we believe, interesting and all the concepts can be distinguished by different sets of properties.

For example, these properties can be used to guide the search for new product measures as shown in [3]. Moreover, we hope this framework will help avoid future confusion, often encountered in the literature, about what properties product measures should or should not have. Studying measure properties is important in order to provide discipline and rigor to the search for new product measures. However, the relevancy of a property to a given measure must be assessed in the context of a well defined measurement concept, e.g., one should not attempt to verify if a length measure is additive.

This framework does not prevent useless measures from being defined. The usefulness of a measure can only be assessed in a given context (i.e., with respect to a given experimental goal and environment) and after a thorough experimental validation [3]. This framework is not a global answer to the problems of software engineering measurement; it is just one of the necessary components of a measure definition process as presented in [3].

Future research will include the definition of more specific measurement frameworks for particular product abstractions, e.g., control flow graphs, data dependency graphs. Also, new concepts could be defined, such as information content (in the information theory sense).

Acknowledgments

We would like to thank Khaled El Emam, Wolfgang Heuser, Yong-Mi Kim, Bryan Hsueh, Oliver Laitenberger, Carolyn Seaman, and Marvin Zelkowitz for reviewing the drafts of this paper. We would also like to thank the anonymous referees and Ross Jeffery for their thorough review of the paper.

This work was supported in part by NASA grant NSG-5123, UMIACS, NSF grant 01-5-24845, NSERC Canada, MURST, and CNR.

REFERENCES

[1] N. Fenton, "Software measurement: A necessary scientific basis," *IEEE Trans. Software Eng.*, vol. 20, no. 3, pp. 199-206, Mar. 1994.
[2] D.L. Parnas, "On the criteria to be used in decomposing systems into modules," *Comm. ACM*, vol. 15, pp. 1,053-1,058, May 1972.
[3] L. Briand, S. Morasca, and V.R. Basili, "A goal-driven definition process for product metrics based on properties," Univ. of Maryland, Dept. of Computer Science, Tech. Report CS-TR-3346, UMIACS-TR-94-106, 1994. Submitted for publication.
[4] R. Courtney and D. Gustafson, "Shotgun correlations in software measures," *Software Eng. J.*, vol. 8, no. 1, pp. 5-13, Jan. 1993.
[5] K.B. Lakshmanian, S. Jayaprakash, and P.K. Sinha, "Properties of control-flow complexity measures," *IEEE Trans. Software Eng.*, vol. 17, no. 12, pp. 1,289-1,295, Dec. 1991.
[6] J. Tian and M.V. Zelkowitz, "A formal program complexity model and its application," *J. Systems Software*, vol. 17, pp. 253-266, 1992.
[7] E.J. Weyuker, "Evaluating software complexity measures," *IEEE Trans. Software Eng.*, vol. 14, no. 9, pp. 1,357-1,365, Sept. 1988.
[8] S.R. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Software Eng.*, vol. 20, no. 6, pp. 476-493, June 1994.
[9] W. Harrison, "An entropy-based measure of software complexity," *IEEE Trans. Software Eng.*, vol. 18, no. 11, pp. 1,025-1,029, Nov. 1992.
[10] M. Shepperd, "Algebraic models and metric validation," *Formal Aspects of Measurement*, T. Denvir, R. Herman, and R.W. Whitty, eds., pp. 157-173, *Lecture Notes in Computer Science*, Springer-Verlag, 1992.
[11] H. Zuse, *Software Complexity: Measures and Methods*. Amsterdam: de Gruyter, 1991.
[12] J.C. Cherniavsky and C.H. Smith, "On Weyuker's axioms for software complexity measures," *IEEE Trans. Software Eng.*, vol. 17, no. 6, pp. 636-638, June 1991.
[13] B. Henderson-Sellers and J. Edwards, *Book Two of the OO Knowledge: The Working Object*. Prentice Hall Object-Oriented Series, 1994.
[14] N. Fenton, *Software Metrics, A Rigorous Approach*. Chapman and Hall, 1991.
[15] M.H. Halstead, *Elements of Software Science*. Elsevier North-Holland, 1977.
[16] B. Curtis, S. Sheppard, P. Milliman, M. Borst, and T. Love, "Measuring the psychological complexity of software maintenance task with the Halstead and McCabe metrics," *IEEE Trans. Software Eng.*, vol. 5, no. 2, pp. 96-104, Mar. 1979.
[17] A.C. Melton, D.A. Gustafson, J.M. Bieman, and A.A. Baker, "Mathematical perspective of software measures research," *IEE Software Eng. J.*, vol. 5, no. 5, pp. 246-254, 1990.
[18] E.I. Oviedo, "Control flow, data flow and program complexity," *Proc. IEEE COMPSAC*, pp. 146-152, Nov. 1980.
[19] T.J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. 2, no. 5, pp. 308-320, Apr. 1976.
[20] S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE Trans. Software Eng.*, vol. 7, no. 5, pp. 510-518, Sept. 1981.
[21] D. Card and R. Glass, *"Measuring software design quality."* Englewood Cliffs, N.J.: Prentice Hall, 1990.
[22] L. Briand, S. Morasca, and V. Basili, "Defining and validating high-level design metrics," CS-TR 3301, Univ. of Maryland, College Park, Md. Submitted for publication.
[23] J. Bieman and L.M. Ott, "Measuring functional cohesion," *IEEE Trans. Software Eng.*, vol. 20, no. 8, pp. 644-657, Aug. 1994.
[24] F. Roberts, *Measurement Theory with Applications to Decisionmaking, Utility, and the Social Sciences*. Addison-Wesley, 1979.
[25] N. Fenton and A. Melton, "Deriving structurally based software measures," *J. Systems Software*, vol. 12, pp. 177-187, 1990.
[26] R.E. Prather, "An axiomatic theory of software complexity measure," *The Computer J.*, vol 27, no. 4, pp. 340-346, 1984.
[27] *Encyclopaedia of Software Eng.* John Wiley & Sons, 1994.
[28] L. Briand, K. El Emam, and S. Morasca, "On the application of measurement theory in software engineering," to appear in *Empirical Software Engineering, An Int'l J.*

Lionel C. Briand received the BS degree in geophysics, with honors, and the MS degree in computer science from the University of Paris VI, France, in 1985 and 1987, respectively. He received the PhD degree, with high honors, in computer science from the University of Paris XI, France, in April 1994.

From 1989 to 1994, Dr. Briand served as a research scientist at the NASA Software Engineering Laboratory, a research consortium formed by the NASA Goddard Space Flight Center, the University of Maryland, and the Computer Science Corporation. Since 1994, he has been the lead researcher of the software engineering group at the Computer Research Institute of Montreal (Centre de Recherche Informatique de Montréal). His current research interests include measurement and modeling of software development products and processes, software quality assurance, domain specific architectures, reuse, and reengineering. Dr. Briand is currently involved in several measurement, quality assurance, and process improvement programs in industrial and government organizations.

Sandro Morasca received his Dr. Eng. degree (Italian Laurea) cum laude in 1985 and his PhD in computer science 1991, both from the Politecnico di Milano. He is currently an assistant professor of computer science at the Politecnico di Milano. From 1991 to 1993, Dr. Morasca was a faculty research associate at the University of Maryland Institute for Advanced Computer Studies (UMIACS). His research interests include the specification and verification of concurrent and real-time systems, formal methods, and empirical studies in software engineering. Dr. Morasca is a member of the IEEE Computer Society.

52-61
415786
360714
12p

A Validation of Object-Oriented Design Metrics as Quality Indicators

Victor R. Basili, *Fellow, IEEE*, Lionel C. Briand,
and Walcélio L. Melo, *Member, IEEE Computer Society*

Abstract—This paper presents the results of a study in which we empirically investigated the suite of object-oriented (OO) design metrics introduced in [13]. More specifically, our goal is to assess these metrics as predictors of fault-prone classes and, therefore, determine whether they can be used as early quality indicators. This study is complementary to the work described in [30] where the same suite of metrics had been used to assess frequencies of maintenance changes to classes. To perform our validation accurately, we collected data on the development of eight medium-sized information management systems based on identical requirements. All eight projects were developed using a sequential life cycle model, a well-known OO analysis/design method and the C++ programming language. Based on empirical and quantitative analysis, the advantages and drawbacks of these OO metrics are discussed. Several of Chidamber and Kemerer's OO metrics appear to be useful to predict class fault-proneness during the early phases of the life-cycle. Also, on our data set, they are better predictors than "traditional" code metrics, which can only be collected at a later phase of the software development processes.

Index Terms—Object-oriented design metrics, error prediction model, object-oriented software development, C++ programming language.

1 Introduction

1.1 Motivation

The development of a large software system is a time- and resource-consuming activity. Even with the increasing automation of software development activities, resources are still scarce. Therefore, we need to be able to provide accurate information and guidelines to managers to help them make decisions, plan and schedule activities, and allocate resources for the different software activities that take place during software development. Software metrics are, thus, necessary to identify where the resources are needed; they are a crucial source of information for decision-making [22].

Testing of large systems is an example of a resource- and time-consuming activity. Applying equal testing and verification effort to all parts of a software system has become cost-prohibitive. Therefore, one needs to be able to identify fault-prone modules so that testing/verification effort can be concentrated on these modules [21]. The availability of adequate product design metrics for characterizing error-prone modules is, thus, vital.

Many product metrics have been proposed [16], [26], used, and, sometimes, empirically validated [3], [4], [19], [30], e.g., number of lines of code, McCabe complexity metric, etc. In fact, many companies have built their own cost, quality, and resource prediction models based on product metrics. TRW [7], the Software Engineering Laboratory (SEL) [31], and Hewlett Packard [20] are examples of software organizations that have been using product metrics to build their cost, resource, defect, and productivity models.

1.2 Issues

In the last decade, many companies have started to introduce object-oriented (OO) technology into their software development environments. OO analysis/design methods, OO languages, and OO development environments are currently popular worldwide in both small and large software organizations. The insertion of OO technology in the software industry, however, has created new challenges for companies which use product metrics as a tool for monitoring, controlling, and improving the way they develop and maintain software. Therefore, metrics which reflect the specificities of the OO paradigm must be defined and validated in order to be used in industry. Some studies have concluded that "traditional" product metrics are not sufficient for characterizing, assessing, and predicting the quality of OO software systems. For example, in [12] it was reported that McCabe cyclomatic complexity appeared to be an inadequate metric for use in software development based on OO technology.

To address this issue, OO metrics have recently been proposed in the literature [1], [6], [13]. However, with a few exceptions [10], [30], most of them have not undergone an

• V.R. Basili is with the University of Maryland, Institute for Advanced Computer Studies and Computer Science Dept., A.V. Williams Bldg., College Park, MD 20742. E-mail: basili@cs.umd.edu.
• L.C. Briand is with Fraunhofer-Institute for Experimental Software Engineering, Technologiepark II, Sauerwiesen 6, D-67661, Kaiserslautern, Germany. E-mail: briand@iese.fhg.de.
• W.L. Melo is with the Centre de Recherche Informatique de Montréal, 1801 McGill College Ave., Montréal, Québec, H3A 2N4, Canada. E-mail: wmelo@crim.ca.

Manuscript received Sept. 7, 1995; accepted Aug. 30, 1996.
Recommended for acceptance by H. Muller.
For information on obtaining reprints of this article, please send e-mail to: transse@computer.org, and reference IEEECS Log Number S95819.

empirical validation (see [9] and [35] for further discussion of the empirical validation of measures). Empirical validation aims at demonstrating the usefulness of a measure in practice and is, therefore, a crucial activity to establish the overall validity of a measure. A measure may be correct from a measurement theory perspective (i.e., be consistent with the agreed upon empirical relational system) but be of no practical relevance to the problem at hand. On the other hand, a measure may not be entirely satisfactory from a theoretical perspective but can be a good enough approximation and work fine in practice.

In this paper, we present the results of a study in which we performed an empirical validation of the OO metric suite defined in [13] with regard to their ability to identify fault-prone classes. However, the theoretical validation of these metrics is not addressed here and, as a complement to this paper, the reader may refer to a discussion about the mathematical properties of Chidamber and Kemerer's metrics in [11], [24].

Data were collected during the development of eight medium-sized information management systems based on identical requirements. All eight projects were developed using a sequential life cycle model, a well-known Object-Oriented analysis/design method [33], and the C++ programming language [36]. Despite the fact that these projects were run in a university setting, we set up a framework that was representative of currently used technology in industrial settings.

1.3 Outline

This paper is organized as follows. Section 2 presents the suite of OO metrics proposed by Chidamber and Kemerer [13], offers the experimental hypotheses to be tested, and then shows a case study from which process and product data were collected allowing a quantitative validation of this suite of metrics. Section 3 presents the actual data collected together with the statistical analysis of the data. Section 4 compares our study with other works on the subject. Section 5 concludes the paper by presenting lessons learned and future work.

2 Design of the Empirical Study

2.1 Dependent and Independent Variables

The goal of this study was to analyze empirically the OO design metrics proposed in [13] for the purpose of evaluating whether or not these metrics are useful for predicting the probability of detecting faulty classes. Assuming testing was performed properly and thoroughly, the probability of fault detection in a class during acceptance testing should be a good indicator of its probability of containing a fault and, therefore, a relevant measure of fault-proneness. The construct validity[1] of our dependent variable can, thus, be demonstrated.

Other measures such as class fault density could have been used. However, the variability in terms of number of faults in our data set is small: Faults were detected only in 36 percent of the classes and 84 percent of the classes contain less than three faults. Therefore, using a dependent variable with low variability would have affected our ability to identify significant relationships between OO design metrics and this dependent variable.

In addition, it was difficult to decide what was the best way to measure the size of classes given the large number of alternatives (e.g., LOC, SLOC, number of methods, number of attributes, etc.). The probability of fault detection was, therefore, the most straightforward and practical measure of fault-proneness and, therefore, a suitable dependent variable for our study. Based on [13], [14], and [15], it is clear that the definitions of these metrics are not language independent. As a consequence, we had to slightly adjust some of Chidamber and Kemerer's metrics in order to reflect the specificities of C++. These metrics are as follows:

- Weighted Methods per Class (WMC). WMC measures the complexity of an individual class. Based on [13], if we consider all methods of a class to be equally complex, then WMC is simply the number of methods defined in each class. In this study, we adopted this approach for the sake of simplicity and because the choice of a complexity metric would be somewhat arbitrary since it is not fully specified in the metric suite. Thus, WMC is defined as being the number of all member functions and operators defined in each class. However, "friend" operators (C++ specific construct) are not counted. Member functions and operators inherited from the ancestors of a class are also not counted. This definition is identical to the one described in [14].

  In [15], Churcher and Shepperd have argued that WMC can be measured in different ways depending on how member functions and operations defined in a C++ class are counted. We believe that the different counting rules proposed in [15] correspond to different metrics, similar to the WMC metric, and which must be empirically validated as well. A validation of Churcher and Shepperd's WMC-like metrics is, however, beyond the scope of this paper.
- Depth of Inheritance Tree of a class (DIT)—DIT is defined as the maximum depth of the inheritance graph of each class. C++ allows multiple inheritance and, therefore, classes can be organized into a directed acyclic graph instead of trees. DIT, in our case, measures the number of ancestors of a class.
- Number Of Children of a Class (NOC)—This is the number of direct descendants for each class.
- Coupling Between Object classes (CBO)—A class is coupled to another one if it uses its member functions and/or instance variables. CBO provides the number of classes to which a given class is coupled.
- Response For a Class (RFC)—This is the number of methods that can potentially be executed in response to a message received by an object of that class. In our study, RFC is the number of C++ functions directly invoked by member functions or operators of a C++ class.

1. Construct validity is discussed further in [27]. It is defined as the extent to which the theoretical construct of interest (e.g., our dependent variable: fault-proneness) is measured successfully, i.e., do we really measure what we purport to measure?

- Lack of Cohesion on Methods (LCOM)—This is the number of pairs of member functions without shared instance variables, minus the number of pairs of member functions with shared instance variables. However, the metric is set to zero whenever the above subtraction is negative.

Readers acquainted with C++ can see that some particularities of C++ are not taken into account by Chidamber and Kemerer's metrics, e.g., C++ templates, friend classes, etc. In fact, additional work is necessary in order to extend the proposed OO metric set with metrics specifically tailored to C++.

2.2 Hypotheses

In order to validate the above metrics as quality indicators, their expected relationship with fault-proneness (or rather the measure we selected for this attribute: probability of fault detection) must be validated. The experimental hypotheses to be statistically tested are, for each metric, as follows:

- H-WMC: A class with significantly more member functions than its peers is more complex and, by consequence, tends to be more fault-prone.
- H-DIT: Well-designed OO systems are those structured as forests of classes, rather than as one very large inheritance lattice. In other words, a class located deeper in a class inheritance lattice is supposed to be more fault-prone because the class inherits a large number of definitions from its ancestors. In addition, deep hierarchies often imply problems of conceptual integrity, i.e., it becomes unclear which class to specialize from in order to include a subclass in the inheritance hierarchy [17].
- H-NOC: Classes with large number of children (i.e., subclasses) are difficult to modify and usually require more testing because the class potentially affects all of its children. Furthermore, a class with numerous children may have to provide services in a larger number of contexts and must be more flexible. We expect this to introduce more complexity into the class design and, therefore, we expect classes with large number of children to be more fault-prone.
- H-CBO: Highly coupled classes are more fault-prone than weakly coupled classes because they depend more heavily on methods and objects defined in other classes.
- H-RFC: Classes with larger response sets implement more complex functionalities and are, therefore, more fault-prone.
- H-LCOM: Classes with low cohesion among its methods suggests an inappropriate design (i.e., the encapsulation of unrelated program objects and member functions that should not be together) which is likely to be more fault-prone.

2.3 Study Participants

In order to validate the hypotheses stated in the previous section, we ran an empirical study over four months (from September to December 1994). The study participants were the students of an upper division undergraduate/graduate level course offered by the Department of Computer Science at the University of Maryland. The objective of this class was to teach OO software analysis and design. The students were not required to have previous experience or training in the application domain or OO methods. All students had some experience with C or C++ programming and relational databases and, therefore, had the basic skills necessary for such a study.

In order to control for differences in skills and experience among students, the students were randomly grouped into eight teams of three students. Furthermore, in order to ensure the groups were comparable with respect to the ability of their members, the following procedure (i.e., known as "blocking" [27]) was used to assign students to groups:

- First, the level of experience of each student was characterized at the beginning of the study. We used questionnaires and performed interviews. We asked the students information regarding their previous working experience, their student status (part-time, full-time student), their computer science degree (BS, MSc, PhD), their previous experiences with analysis/design methods, and their skill regarding various programming languages.
- Second, each of the eight most experienced students was randomly assigned to a different group (i.e., team). Students considered most experienced were computer science PhD candidates who had already implemented large ($\geq$ 10 thousands source lines of code, KSLOC) C or C++ programs and those with industrial experience greater than two years in C programming. None of the students had significant experience in Object-Oriented software analysis and design methods. Similarly, each of the eight next most experienced students were randomly assigned to different groups and this was repeated for the remaining eight students.

2.4 The Development Process

Each team was asked to develop a medium-sized management information system that supports the rental/return process of a hypothetical video rental business, and maintains customer and video databases. Such an application domain had the advantage of being easily comprehensible and, therefore, we could make sure that system requirements could be easily interpreted by students regardless of their educational background.

The development process was performed according to a sequential software engineering life-cycle model derived from the Waterfall model. This model includes the following phases: analysis, design, implementation, testing, and repair. At the end of each phase, a document was delivered: Analysis document, design document, code, error report, and finally, modified code, respectively. Requirement specifications and design documents were checked to verify that they matched the system requirements. Errors found in these first two phases were reported to the students. This maximized the chances that the implementation began with a correct OO analysis/design. Acceptance testing was performed by an independent group (see Section 2.5). During the repair phase, the students were asked to correct their system based on the errors found by the independent test group.

OMT, an OO Analysis/Design method, was used during the analysis and design phases [33]. The C++ programming language, the GNU software development environment, and OSF/MOTIF were used during the implementation. Sparc Sun stations were used as the implementation platform. Therefore, the development environment and technology we used are representative of what is currently used in industry and academia. Our results are, thus, more likely to be generalizable to other development environments (external validity).

The following libraries were provided to the students:

1) *MotifApp*. This public domain library provides a set of C++ classes on top of OSF/MOTIF for manipulation of windows, dialogues, menus, etc. [37]. The MotifApp library provides a way to use the OSF/Motif widgets in an OO programming/design style.
2) *GNU library*. This public domain library is provided in the GNU C++ programming environment. It contains functions for manipulation of string, files, lists, etc.
3) C++ *database library*. This library provides a C++ implementation of multi-indexed B-Trees.

We also provided a specific domain application library in order to make our study more representative of industrial conditions. This library implemented the graphical user interface for insertion/removal of customers and was implemented in such a way that the main resources of the OSF/Motif widgets and MotifApp library were used. Therefore, this library contained a small part of the implementation required for the development of the rental system.

No special training was provided for the students to teach them how to use these libraries. However, a tutorial describing how to implement OSF/Motif applications was given to the students. In addition, a C++ programmer, familiar with OSF/Motif applications, was available to answer questions about the use of OSF/Motif widgets and the libraries. A hundred small programs exemplifying how to use OSF/Motif widgets were also provided. In addition, the source code and the complete documentation of the libraries were made available. Finally, it is important to note the students were not required to use the libraries and, depending on the particular design they adopted, different reuse choices were expected.

2.5 Testing

The testing phase was accomplished by an independent group composed of experienced software professionals. This group tested all systems according to similar test plans and using functional testing techniques, spending eight hours testing each system.

2.6 Nature of the Study

Our empirical study is not what could be called formally a *controlled* experiment since the independent variables (i.e., OO design metrics) are not controlled for and not assigned randomly to classes. Such a design would not be implementable. Rather, our study is more observational in nature. However, it is important to note that we have tried to make the results of our study as generalizable as possible (i.e., maximizing external validity) by a careful selection of the study participants, the study material, and the development process. Nevertheless, there is a greater danger that the study be exposed to confounding variables and all significant relationships should be carefully interpreted.

2.7 Data Collection Procedures and Measurement Instruments

We collected:

1) the source code of the C++ programs delivered at the end of the implementation phase,
2) data about these programs,
3) data about errors found during the testing phase and fixes during the repair phase, and
4) the repaired source code of the C++ programs delivered at the end of the life cycle.

GEN++ [18] was used to extract Chidamber and Kemerer's OO design metrics directly from the source code of the programs delivered at the end of the implementation phase. To collect items 2) and 3), we used the following forms, which have been tailored from those used by the Software Engineering Laboratory [23]:

- Fault Report Form.
- Component Origination Form.

In the following sections, we comment on the purpose of the Component Origination and Fault Report forms used in our study and the data they helped collect.

2.7.1 Data Collection Forms

A fault report form was used to gather data about

1) the faults found during the testing phase,
2) classes changed to correct such faults, and
3) the effort in correcting them.

The latter was not used in this study. Further details can be found in [5].

A component origination form was used to record information that characterizes each class under development in the project at the time it goes into configuration management. First, this form was used to capture whether the class has been developed from scratch or has been developed from a reused class. In the latter case, we collected the amount of modification needed to meet the system requirements and design: none, slight (less than 25 percent of code changed), or extensive (more than 25 percent of code change) as well as the name of the reused class. Classes reused without modification were labeled: *verbatim reused*.

In addition, the name of the sub-system to which the class belonged was also collected. In our study, we had two types of sub-systems: user interface (UI) and database processing (DB).

2.7.2 Data Collected

Chidamber and Kemerer's OO design metrics were collected for each of the 180 classes across the eight systems under study. In addition, all faults detected during testing

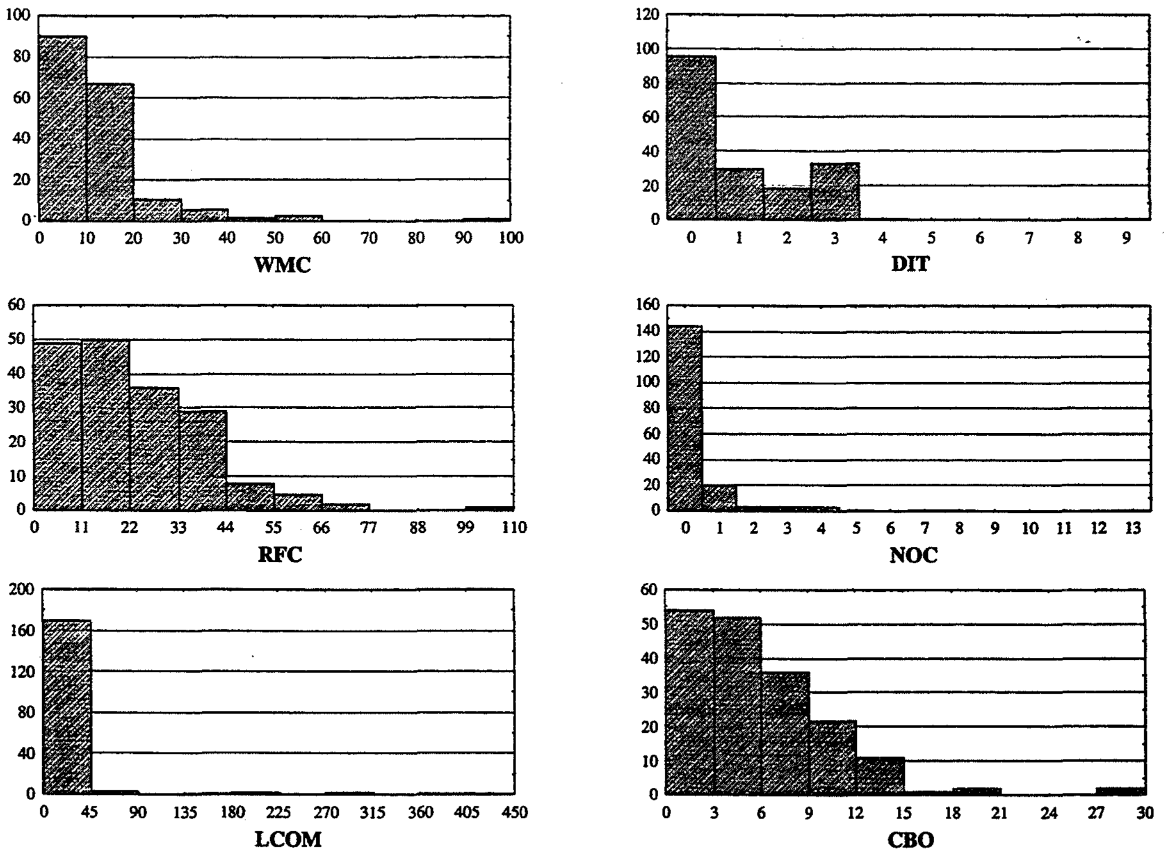


Fig. 1. Distribution of the analyzed OO metrics. The X axes represents the values of the metric. The Y axes represents the number of class.

activities were located in the systems and, therefore, associated with one or several of their classes.

3 Data Analysis

In this section, we will assess empirically whether the OO design metrics defined in [13] are useful predictors of fault-prone classes. This will help us assess these metrics as quality indicators and how they compare to common code metrics. We intend to provide the type of empirical validation that we think is necessary before any attempt to use such metrics as objective and early indicators of quality is made [9]. Section 3.1 shows the descriptive distributions of the OO metrics in the studied sample whereas Section 3.2 provides the results of univariate and multivariate analyses of the relationships between OO metrics and fault-proneness.

3.1 Distribution and Correlation Analyses

Fig. 1 shows the distributions of the analyzed OO metrics based on 180 classes present in the studied systems. Table 1 provides common descriptive statistics of the metric distributions. These results indicate that inheritance hierarchies are somewhat flat (DIT) and that classes have, in general, few children (NOC) (this result is similar to what was found in [13]). In addition, most classes show a lack of cohesion (LCOM) near zero. This latter metric does not seem to differentiate classes well and this may stem from its definition which prevents any negative measure. This issue will be discussed further in Section 3.2.

TABLE 1
Descriptive Statistics of the 180 Studied C++ Classes

	WMC	DIT	RFC	NOC	LCOM	CBO
Maximum	99.00	9.00	105.00	13.00	426.00	30.00
Minimum	1.00	0.00	0.00	0.00	0.00	0.00
Median	9.50	0.00	19.50	0.00	0.00	5.00
Mean	13.40	1.32	33.91	0.23	9.70	6.80
Std Dev	14.90	1.99	33.37	1.54	63.77	7.56

Descriptive statistics will be useful to help us interpret the results of the analysis in the remainder of this section. In addition, they will facilitate comparisons of results from future similar studies.

TABLE 2
Correlation Analysis

	R^2 Values					
	WMC	DIT	RFC	NOC	LCOM	CBO
WMC	1	0.02	**0.24**	0	**0.38**	0.13
DIT		1	0	0	0.01	0
RFC			1	0	0.09	**0.31**
NOC				1	0	0
LCOM					1	0.01

Table 2 shows very clearly that linear Pearson's correlations (R^2: Coefficient of determination) between the studied OO metrics are, in general, very weak. Three coefficients of determination appear somewhat more significant (bold coefficients in Table 2). However, when looking at the scatterplots, only the relationship between CBO and RFC seems not to be due to outliers. We conclude that these metrics are mostly statistically independent and, therefore, do not capture a great deal of redundant information.

3.2 The Relationships Between Fault Probability and OO Metrics

3.2.1 Analysis Methodology

The response variable we use to validate the OO design metrics is binary, i.e., was a fault detected in a class during testing phases? We used logistic regression, a standard technique based on maximum likelihood estimation, to analyze the relationships between metrics and the fault-proneness of classes. Currently, logistic regression is a standard classification technique [25] used in experimental sciences. It has already been used in software engineering to predict error-prone components [8], [29], [32].

Other classification techniques such as classification trees [34], Optimized Set Reduction [8], or neural networks [28] could have been used. However, our goal here is not to compare multivariate analysis techniques (see [8] for a comparison study) but, based on a suitable and standard technique, to validate empirically a set of metrics.

We first used univariate logistic regression, to evaluate the relationship of each of the metrics in isolation and fault-proneness. Then, we performed multivariate logistic regression, to evaluate the predictive capability of those metrics that had been assessed sufficiently significant in the univariate analysis. This modeling process is further described in [25].

A multivariate logistic regression model is based on the following relationship equation (the univariate logistic regression model is a special case of this, where only one variable appears):

$$\pi(X_1, X_2, \ldots, X_n) = \frac{e^{(C_0 + C_1 \bullet X_{i1} + \ldots + C_n \bullet X_{in}) \bullet Y_i}}{1 + e^{(C_0 + C_1 \bullet X_{i1} + \ldots + C_n \bullet X_{in})}} \qquad (*)$$

where π is the probability that a fault was found in a class during the validation phase, and the X_is are the design metrics included as explanatory variables in the model (called *covariates* of the logistic regression equation). The curve between π and any single X_i—i.e., assuming that all other X_js are constant—takes a flexible S shape which ranges between two extreme cases:

1) when a variable is not significant, then the curve approximates a horizontal line, i.e., π does not depend on X_i, and
2) when a variable entirely differentiates error-prone software parts, then the curve approximates a step function.

Such a S shape is perfectly suitable as long as the relationship between X_is and π is monotonic, an assumption consistent with the empirical hypotheses to be tested in this study. Otherwise, higher degree terms have to be introduced in equation (*).

The coefficients C_is will be estimated through the maximization of a likelihood function, built in the usual fashion, i.e., as the product of the probabilities of the single observations, which are functions of the covariates (whose values are known in the observations) and the coefficients (which are the unknowns). For mathematical convenience, $l = ln[L]$, the loglikelihood, is usually the function to be maximized. This procedure assumes that all observations are statistically independent. In our context, an observation is the (non)detection of a fault in a C++ class. Each (non) detection of a fault is assumed to be an event independent from other fault (non)detections. Each data vector in the data set describes an observation and has the following components: An event category (fault, no fault) and a set of OO design metrics (described in Section 2.1) characterizing either the class where the fault was detected or a class where no fault was detected.

The global measure of goodness of fit we will use for such a model is assessed via R^2—not to be confused with the least-square regression R^2—they are built upon very different formulae, even though they both range between zero and one and are similar from an intuitive perspective. The higher R^2, the higher the effect of the model's explanatory variables, the more accurate the model. However, as opposed to the R^2 of least-square regression, high R^2s are rare for logistic regression. For this reason, the reader should not interpret logistic regression R^2s using the usual heuristics for least-square regression R^2s. (The interested reader may refer to [21] for a detailed discussion of this issue.). Logistic regression R^2 is defined by the following ratio:

$$R^2 = \frac{LL_S - LL}{LL_S}$$

where

- LL is the loglikelihood obtained by Maximum Likelihood Estimation of the model described in formula (*)
- LL_S is the loglikelihood obtained by Maximum Likelihood Estimation of a model without any variables, i.e., with only C_0. By carrying out all the calculations, it can be shown that LL_S is given by

$$LL_S = m_0 \ln\left(\frac{m_0}{m_0 + m_1}\right) + m_1 \ln\left(\frac{m_1}{m_0 + m_1}\right)$$

where m_0 (resp., m_1) represents the number of observations for which there are no faults (resp., there is a fault). Looking at the above formula, $LL_S/(m_0 + m_1)$ may be interpreted as the uncertainty associated with the distribution of the dependent variable Y, according to Information Theory concepts. It is the uncertainty left when the variable-less model is used. Likewise, $LL/(m_0 + m_1)$ may be interpreted as the uncertainty left when the model with the covariates is used. As a consequence, $(LL_S - LL)/(m_0 + m_1)$ may be interpreted as the part of uncertainty that is explained by the model. Therefore, the ratio $(LL_S - LL)/LL_S$ may be interpreted as the proportion of uncertainty explained by the model.

Tables 3 and 4 contain the results we obtained through, respectively, univariate and multivariate logistic regression on all of the 180 classes. We report those related to the metrics that turned out to be the most significant across all eight development projects. For each metric, we provide the following statistics:

Coefficient (appearing in Tables 3 and 4), the estimated regression coefficient. The larger the coefficient in absolute value, the stronger the impact (positive or negative, according to the sign of the coefficient) of the explanatory variable on the probability p of a fault to be detected in a class.

TABLE 3
UNIVARIATE ANALYSIS–SUMMARY OF RESULTS

Metrics	Coefficient	$\Delta\psi$	p-value	R^2	Classes
WMC (1)	0.022	2%	0.0607	0.007	ALL
WMC (2)	0.086	9%	0.0003	0.024	New-Ext
WMC (3)	0.027	3%	0.0656	0.015	DB
WMC (4)	0.094	10%	0.0019	0.047	UI
DIT (1)	0.485	62%	0.0000	0.065	ALL
DIT (2)	0.868	138%	0.0000	0.131	New-Ext
DIT (3)	0.475	60%	0.043	0.019	DB
DIT (4)	0.29	34%	0.024	0.017	UI
RFC (1)	0.085	9%	0.0000	0.065	ALL
RFC (2)	0.087	8%	0.0000	0.248	New-Ext
RFC (3)	0.077	8%	0.0000	0.188	DB
RFC (4)	0.108	11%	0.0000	0.362	UI
NOC (1)	–3.3848	–96%	0.0000	0.143	ALL
NOC (2)	–3.62	–97%	0.0011	0.362	New-Ext
NOC (3)	–2.05	–77%	0.0000	0.083	DB
CBO (1)	0.142	15%	0.0000	0.068	ALL
CBO (2)	0.079	8%	0.017	0.020	New-Ext
CBO (3)	0.086	9%	0.006	0.034	DB
CBO (4)	0.284	33%	0.0000	0.170	UI

ALL means all the classes. New-Ext stands for classes which have been created from scratch or extensively modified. DB labels classes implementing database manipulations. UI labels classes implementing user interface functions.

TABLE 4
MULTIVARIATE ANALYSIS WITH OO DESIGN METRICS

	Coefficient	p-value
Intercept	3.13	0.0000
DIT	0.50	0.0004
RFC	0.11	0.0000
NOC	–2.01	0.0178
CBO	0.13	0.0072
Class Origin	1.84	0.0000

- $\Delta\psi$ (appearing in Table 3 only), which is based on the notion of odd ratio [25], and provides an evaluation of the impact of the metric on the response variable. More specifically, the odds ratio $\psi(X)$ represents the ratio between the probability of having a fault and the probability of not having a fault when the value of the metric is X. As an example, if, for a given value X, $\psi(X)$ is two, then it is twice as likely that the class does contain a fault than that it does not contain a fault. The value of $\Delta\psi$ is computed by means of the following formula:

$$\Delta\psi = \frac{\psi(X+1)}{\psi(X)} \qquad (2)$$

  Therefore, $\Delta\psi$ represents the reduction/increase in the odds ratio (expressed as a percentage in Table 3) when the value X increases by one unit. This is designed to provide an intuitive insight into the impact of explanatory variables.

- The statistical significance (p-value, appearing in Tables 3 and 4) provides an insight into the accuracy of the coefficient estimates. It tells the reader about the probability of the coefficient being different from zero by chance. Historically, a significance threshold of $\alpha = 0.05$ (i.e., 5 percent probability) has often been used to determine whether an explanatory variable was a significant predictor. However, the choice of a particular level of significance is ultimately a subjective decision and other levels such as $\alpha = 0.01$ or 0.1 are common. Also, the larger the level of significance, the larger the standard deviation of the estimated coefficients, and the less believable the calculated impact of the explanatory variables. The significance test is based on a likelihood ratio test [25] commonly used in the framework of logistic regression.

3.2.2 Univariate Analysis

In this section, we analyze the relationships between six OO metrics introduced in [13] (though slightly adapted to our context) and the probability of fault detection in a class during test phases. Thus, we intend to test the hypotheses stated in Section 2.2.

- Weighted Methods per Class (WMC) was shown to be somewhat significant (p-value = 0.06) overall. For new and extensively modified classes and for UI (Graphical and Textual User Interface) classes, the results are more significant: p-value = 0.0003 and p-value = 0.001, respectively. Therefore, the H-WMC hypothesis is supported by these results: The larger the WMC, the larger the probability of fault detection. These results can be explained by the fact that the internal complexity does not have a strong impact if the class is reused verbatim or with very slight modifications. In that case, the class interface properties will have the most significant impact.
- Depth of Inheritance Tree of a class (DIT) was shown to be very significant (p-value = 0.0000) overall. The H-DIT hypothesis is supported by the results: The larger the DIT, the larger the probability of fault detection. Again, the strength of the relationship increases (R^2 goes from 0.06 to 0.13) when only new and extensively modified classes are considered.
- Response For a Class (RFC) was shown to be very significant overall (p-value = 0.0000). The H-RFC hypothesis is supported by the results: The larger the RFC, the larger the probability of fault detection. Again, R^2 improved significantly for new and extensively modified classes and UI classes (from 0.06 to 0.24 and 0.36, respectively). Reasons are believed to be the same as for WMC for extensively modified classes. In addition, UI classes show a distribution which is significantly different from that of DB classes: The mean and median are significantly higher. This, as a result, may strengthen the impact of RFC when performing the analysis.
- Number Of Children of a Class (NOC) appeared to be very significant (except in the case of UI classes) but the observed trend is contrary to what was stated by

the H-NOC hypothesis: The larger the NOC, the lower the probability of fault detection. This surprising trend can be explained by the combined facts that most classes do not have more than one child and that verbatim reused classes are somewhat associated with a large NOC. Since we have observed that reuse was a significant negative factor on fault density [5], this explains why large NOC classes are less fault-prone. Moreover, there is some instability across class subsets with respect to the impact of NOC on the probability of detecting a fault in a class (see $\Delta\psi$s in Table 3). This may be explained in part by the lack of variability on the NOC measurement scale (see descriptive analysis in Table 1 and distribution in Fig. 1).

- Lack of Cohesion on Methods (LCOM) was shown to be insignificant in all cases (this is why the results are not shown in Table 3) and this should be expected since the distribution of LCOM shows a lack of variability and a few very large outliers. This stems in part from the definition of LCOM where the metric is set to zero when the number of class pairs sharing variable instances is larger than that of the ones not sharing any instances. This definition is definitely not appropriate in our case since it sets cohesion to zero for classes with very different cohesions and keeps us from analyzing the actual impact of cohesion based on our data sample.
- Coupling Between Object classes (CBO) is significant and more particularly so for UI classes (p-value = 0.0000 and R^2 = 0.17). No satisfactory explanation could be found for differences in pattern between UI and DB classes.

It is important to remember, when looking at the results in Table 3, that the various metrics have different units. Some of these units represent "big steps" on each respective measurement scale while others represent "smaller steps." As a consequence, some coefficients show a very small impact (i.e., $\Delta\psi$s) when compared to others. This, however, is not a valid criterion to evaluate the predictive usefulness of such metrics.

Most importantly, aside from NOC, all metrics appear to have a very stable impact across various categories of classes (i.e., DB, UI, New-Ext, etc.). This is somewhat encouraging since it tells us that, in that respect, the various types of components are comparable. If we were considering different types of faults separately, the results might be different. Such a refinement is, however, part of our future research plans.

3.2.3 Multivariate Analysis

The OO design metrics presented in the previous section can be used early in the life cycle (high- or low-level design) to build a predictive model of fault-prone classes. In order to obtain an optimal model, we included these metrics into a multivariate logistic regression model. However, only the metrics that significantly improve the predictive power of the multivariate model were included through a stepwise selection process. Another significant predictor of fault-proneness is the level of reuse of the class (called "Class origin" in Table 4). This information is available at the end of the design phase when reuse candidates have been identified in available libraries and the amount of change required can be estimated. Table 4 describes the computed multivariate model. Using such a model for classification, the results shown in Table 5 are obtained by using a classification threshold of π(Fault detection) = 0.5, i.e., when $\pi > 0.5$, the class is classified as faulty and, otherwise, as nonfaulty. As expected, classes predicted as faulty contain a large number of faults (250 faults on 48 classes) because those classes tend to show a better classification accuracy.

TABLE 5
CLASSIFICATION RESULTS WITH OO DESIGN METRICS

Actual	Predicted: No Fault	Predicted: Fault
No Fault	90	32
Fault	10 (18)	48 (250)

The figures before parentheses in the right column are the number of classes classified as faulty. The figures within the parentheses are the faults contained in those classes.

We now assess the impact of using such a prediction model by assuming, in order to simplify computations, that inspections of classes are 100 percent effective in finding faults. In that case, 80 classes (predicted as faulty) out of 180 would be inspected and 48 faulty classes out of 58 would be identified before testing. If we now take into account individual faults, 250 faults out of 258 would be detected during inspection. As mentioned above, such a good result stems from the fact that the prediction model is more accurate for multiple-faults classes. To summarize, results show that the studied OO metrics are useful predictors of fault-proneness.

In order to evaluate the predictive accuracy of these OO design metrics, it would be interesting to compare their predictive capability and that of usual code metrics even though they can only be obtained later in the development life cycle. Three code metrics, from the set provided by the Amadeus tool[2] [2], were selected through a stepwise logistic regression procedure. Table 6 shows the resulting parameter estimations of the multivariate logistic regression model where: *MaxStatNext* is the maximum level of statement nesting in a class, *FunctDef* is the number of function declarations, and *FunctCall* is the number of function calls. It should be noted that other multivariate models can be generated using different metrics provided by Amadeus and yield results of similar accuracy. The model in Table 6 happens to be, however, the one resulting from the use of a standard, stepwise logistic regression analysis procedure.

TABLE 6
MULTIVARIATE ANALYSIS WITH CODE METRICS

	Coefficient	p-value
Intercept	0.39	0.0384
MaxStatNest	–0.286	0.0252
FunctDef	0.166	0.0010
FunctCall	–0.0277	0.0000

In addition to being collectable only later in the process, code metrics appear to be somewhat poorer as predictors of class fault-proneness (see Table 7). In this case, 112 classes

2. The Amadeus tool provides 35 code metrics, e.g., lines of code with and without blank, executable statements, declaration statements, function declaration, function definitions, function calls, cyclomatic complexity, loop statements, maximum class depth and width in a file, number of method declarations, definitions, and average number of methods.

(predicted as faulty) out of 180 would be inspected and 51 faulty classes out of 58 would be detected. If we now take into account individual faults, 231 faults out of 268 would be detected during inspection. Three more faulty classes would be corrected (51 versus 48) but 32 more classes would have to be inspected (112 versus 80) resulting in a significant extra effort. Moreover, the OO design metrics are better predictors of classes containing large numbers of faults since 19 more faults (250 versus 231) would be detected in that case. Therefore, predictions based on code metrics appear to be poorer.

TABLE 7
CLASSIFICATION RESULTS BASED ON CODE METRICS SHOWN IN TABLE 6

	Predicted	
Actual	No fault	Fault
No Fault	61	61
Fault	7 (37)	51 (231)

Table 8 confirms that result by showing the values of correctness (percentage of classes correctly predicted as faulty) and completeness (percentage of faulty classes detected). Values between parentheses present predictions' correctness and completeness values when classes are weighted according to the number of faults they contain (classes with no fault are weighted one).

TABLE 8
CLASSIFICATION ACCURACIES BASED ON OO AND CODE METRICS SHOWN IN TABLE 3 AND TABLE 6

Model Accuracy	OO metrics	Code metrics
Completeness	88% (93%)	83% (86%)
Correctness	60% (92%)	45.5% (86%)

3.2.4 *Threats to Validity*

Several threats to the external validity of our study may limit the generalizability of our results:

- The programs developed lie between five KSLOC and 14 KSLOC. Those programs are small as compared to large industry systems. The relationships between the studied OO design metrics and the fault introduction probability are the results of a complex psychological phenomenon and they may look very different in larger programs.
- The conceptual complexity of these systems was rather limited. Again, many different problems may arise in more complex systems.
- It is likely that the study participants were not as well trained and as experienced as average professional programmers. However, this was partially addressed as discussed in Section 2.4.

4 RELATED WORK

In [10], metrics for measuring abstract data type (ADT) cohesion and coupling are proposed and are validated as predictors of faulty ADTs. The main differences and similarities between the work here and [10] are as follows (see Table 9). They did not empirically validate their metrics on OO programs in a context of inheritance but they used a similar validation approach. In both cases, statistical model were built to predict component (i.e., ADTs and classes, respectively) fault-proneness (i.e., probability of fault detection) by using multiple logistic regression.

In [30], a validation of Chidamber and Kemerer's OO metrics studying the number of changes performed in two commercial systems implemented with an OO dialect of Ada was conducted. They show that Chidamber and Kemerer's OO metrics appeared to be adequate in predicting the frequency of changes across classes during the maintenance phase. They provided a model to predict the number of modifications in a class, which they assume is proportional to change effort and is representative of class maintainability.

The work described in [30] is comparable to our work in the following ways (see Table 9). Li and Henry [30] used the same suite of OO metrics we used. They also used data from products implemented in an OO language which provides multiple inheritance, overloading, and polymorphism. On the other hand, we used the probability of fault detection as the dependent variable of our statistical model. Thus, our goal was to assess whether Chidamber and Kemerer's OO metrics were useful predictors of fault-prone classes. In addition, in [30] (multivariate) least-square linear regression was used to build a predictive model whereas we used logistic regression (i.e., a classification technique for binary dependent variables). The nature of our dependent variable (i.e., (non)occurrence of fault detection) has led us to use logistic regression [25].

TABLE 9
SOME DIFFERENCES AND SIMILARITIES BETWEEN [10], [30], AND OUR WORK

	VALIDATION WORK		
CRITERIA	**Briand et al.** [10]	**Li and Henry** [30]	**Our work**
Suite of Metrics	ADT Cohesion and Coupling	CK metrics	CK metrics
Type of products	Ada	OO dialect of Ada	C++
Dependent variable	fault occurrence in Ada packages	number of changes in component's	fault occurrence in C++ classes
Statistical technique	logistic regression	least-square regression	logistic regression

5 CONCLUSIONS AND FURTHER WORK

In this study, we collected data about faults found in object-oriented classes. Based on these data, we verified how much fault-proneness is influenced by internal (e.g., size, cohesion) and external (e.g., coupling) design characteristics of OO classes. From the results presented above, five out of the six Chidamber and Kemerer's OO metrics appear to be useful to predict class fault-proneness during the high- and low-level design phases of the life-cycle. In addition, Chidamber and Kemerer's OO metrics show to be better predictors than the best set of "traditional" code metrics, which can only be collected during later phases of the software development processes.

This empirical validation provides the practitioner with some empirical evidence demonstrating that most of Chidamber and Kemerer's OO metrics can be useful quality indicators. Furthermore, most of these metrics appear to be complementary indicators which are relatively independent from each other. The results we obtained provide motivation for further investigation and refinement of Chidamber and Kemerer's OO metrics.

Finally, results seem to show that one would likely be able to make inspections of design or code artifacts more efficient if they were driven by models such as the one we built in Section 3.2.3, based on Chidamber and Kemerer's OO metrics. However, how to help focus inspections on error-prone parts in large programs is still an important issue to be further investigated. Our results should be interpreted as maximum possible gains and not as expected gains.

Our future work includes:

- Replicating this study in an industrial setting: A sample of large-scale projects developed in C++ and Ada95 in the framework of the NASA Goddard Flight Dynamics Division (Software Engineering Laboratory). This work should help us better understand the prediction capabilities of the suite of OO metrics described in this paper. Replication should help us achieve the following objectives:
    - Build models and provide guidance to improve the allocation of resources with respect to test and verification efforts.
    - Gain a better understanding of the impact of OO design strategies (e.g., single versus multiple inheritance) on different types of defects and rework. In this study, because the data collection process was not fully adequate, we were unable to analyze the relationships of OO design metrics with rework and different defect categories. With regard to rework, we believe that this drawback could be overcome by refining our data collection process to capture the amount of effort spent debugging each class individually. With regard to defect categories, we would need to collect additional information about defect origin (e.g., specification, design, implementation, previous change), defect type (e.g., omission/commission), defect class (e.g., external interface, internal interface, etc.), etc.
    - Investigating the prediction usefulness of Chidamber and Kemerer's OO design metrics with regard to different types of faults, e.g., fault severity. The fault-proneness prediction capabilities of any suite of OO may be different depending on the type of fault used.
- Studying the variations, in terms of metric definitions and experimental results, between different OO programming languages. The fault-proneness prediction capabilities of the suite of OO metrics discussed in this paper can be different depending on the programming language used. Work must be undertaken to validate this suite of OO design metrics across different OO languages, e.g., Ada95, Smalltalk, C++, etc.
- Extending the empirical investigation to other OO metrics proposed in the literature and develop improved metrics, e.g., more language specific, based on more sophisticated hypotheses.

Acknowledgments

We want to thank (1) Dr. Gianluigi Caldiera for helping us teach the OMT method; (2) A. Belkhelladi, K. El Emam, C. Kemerer, J. Kontio, C. Seaman, F. Shull, and R. Tesoriero for their suggestions that helped improve both the content and the form of this paper; and (3) the students of University of Maryland for their participation on this study. This work will not be possible without the help of Prem Devanbu (AT&T Labs). Finally, we wish to thank the anonymous reviewers and Hausi Muller, the editor of this paper, for their valuable comments. This work was supported, in part, by the National Aeronautics and Space Administration under Grant No. NSG-5123, the National Science Foundation under Grant No. 01-5-24845, Fraunhofer Gesellschaft, UMIACS, and Westinghouse Corporation.

References

[1] F.B. Abreu and R. Carapuça, "Candidate Metrics for Object-Oriented Software within a Taxonomy Framework," *J. System and Software*, vol. 26, no. 1, pp. 87–96, Jan. 1994.

[2] Amadeus Software Research, *Getting Started With Amadeus*, Amadeus Measurement System, 1994.

[3] V. Basili and D. Hutchens, "Analyzing a Syntactic Family of Complexity Metrics," *IEEE Trans. Software Eng.*, vol. 9, no. 6, pp. 664–673, June 1982.

[4] V. Basili, R. Selby, and T.-Y. Philips, "Metric Analysis and Data Validation Across Fortran Projects," *IEEE Trans. Software Eng.*, vol. 9, no. 6, pp. 652–663, June 1983.

[5] V. Basili, L. Briand, and W. Melo, "Measuring the Impact of Reuse on Software Quality and Productivity," *Comm. ACM*, vol. 39, no. 10, pp. 104–116, Oct. 1996.

[6] J.M. Bieman and B.-K. Kang, "Cohesion and Reuse in an Object-Oriented System," *Proc. ACM SIGSOFT Symp. Software Reusability*, Seattle, Wash., pp. 259–262, 1995.

[7] B. Boehm, *Software Eng. Economics*, Prentice-Hall, 1981.

[8] L. Briand, V. Basili, and C. Hetmanski, "Developing Interpretable Models with Optimized Set Reduction for Identifying High Risk Software Components," *IEEE Trans. Software Eng.*, vol. 19, no. 11, pp. 1,028-1,044, Nov. 1993.

[9] L. Briand, K. El Emam, and S. Morasca, *Theoretical and Empirical Validation of Software Product Measures*, ISERN Technical Report 95-03, 1995.

[10] L. Briand, S. Morasca, and V. Basili, *Defining and Validating High-Level Design Metrics*, Technical Report CS-TR-3301, Univ. of Maryland, Dept. of Computer Science, College Park, Md., 1994.

[11] L. Briand, S. Morasca, and V. Basili, "Property Based Software Engineering Measurement," *IEEE Trans. Software Eng.*, vol. 22, no. 1, pp. 68-86, Jan. 1996.

[12] I. Brooks, "Object-Oriented Metrics Collection and Evaluation with a Software Process," *Proc. OOPSLA '93 Workshop Processes and Metrics for Object-Oriented Software Development*, Washington, D.C., 1993.

[13] S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object-Oriented Design," *IEEE Trans. Software Eng.*, vol. 20, no. 6, pp. 476–493, June 1994.

[14] S.R. Chidamber and C.F. Kemerer, "Authors Reply," *IEEE Trans. Software Eng.*, vol. 21, no. 3, p. 265, Mar. 1995.

[15] N.I. Churcher and M.J. Shepperd, "Comments on 'A Metrics Suite for Object-Oriented Design,'" *IEEE Trans. Software Eng.*, vol. 21, no. 3, pp. 263–265, Mar. 1995.

[16] S.D. Conte, H.E. Dunsmore, and V.Y. Shen, *Software Eng. Metrics and Models*, Benjamin/Cummings, 1989.

[17] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood, "The Effect of Inheritance Depth on the Maintainability of Object-Oriented Software," *Empirical Software Eng.: An Int'l J.*, vol. 1, no. 2, Feb. 1996.

[18] P. Devanbu, "GENOA/GENII—A Customizable Language and Front-End Independent Code Analyzer," *Proc. 14th Int'l Conf. Software Eng.*, Melbourne, Australia, 1992.

[19] P. Devanbu, S. Karstu, W. Melo, and W. Thomas, "Analytical and Empirical Evaluation of Software Reuse Metrics," *Proc. 18th Int'l Conf. Software Eng.*, pp. 189-199, Berlin, Germany, 1996.
[20] R.B. Grady, *Practical Software Metrics for Project Management and Process Improvement*, Prentice Hall, 1992.
[21] W. Harrison, "Using Software Metrics to Allocate Testing Resources," *J. Management Information Systems*, vol. 4, no. 4, pp. 93–105, Apr. 1988.
[22] W. Harrison, "Software Measurement: A Decision-Process Approach," *Advances in Computers*, vol. 39, pp. 51–105, 1994.
[23] G. Heller, J. Valett, and M. Wild, *Data Collection Procedure for Software Eng. Laboratory (SEL) Database*, SEL Series, SEL-92-002, 1992.
[24] M. Hitz and B. Montazeri, "Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective," *IEEE Trans. Software Eng.*, vol. 22, no. 4, pp. 267-271, Apr. 1996.
[25] D. Hosmer and S. Lemeshow, *Applied Logistic Regression*, Wiley-Interscience, 1989.
[26] N.E. Fenton, *Software Metrics: A Rigorous Approach*, Chapman & Hall, 1991.
[27] C.M. Judd, E.R. Smith, and L.H. Kidder, *Research Methods in Social Relations*, Harcourt Brace Jovanovich College Publishers, 1991.
[28] T.M. Khohgoftaar, A.S. Panday, and H.B. More, "A Neural Network Approach for Predicting Software Development Faults," *Proc. Third Int'l IEEE Symp. Software Reliability Eng.*, North Carolina, 1992.
[29] F. Lanubile and G. Visaggio, "Evaluating Predictive Quality Models Derived from Software Measures: Lessons Learned," to appear in the *J. Software and Systems;* also available as Technical Report CS-TR-3606, Univ. of Maryland, Computer Science Dept., College Park, Md., 1996.
[30] W. Li and S. Henry, "Object-Oriented Metrics that Predict Maintainability," *J. Systems and Software*, vol. 23, no. 2, pp. 111–122, Feb. 1993.
[31] F. McGarry, R. Pajersk, G. Page, S. Waligora, V. Basili, and M. Zelkowitz, *Software Process Improvement in the NASA Software Eng. Laboratory*. Carnegie Mellon Univ., Software Eng. Inst., Technical Report CMU/SEI-95-TR-22, Dec. 1994.
[32] J. Munson and T. Khoshgoftaar, "The Detection of Fault-Prone Programs," *IEEE Trans. Software Eng.*, vol. 18, no. 5, May 1992.
[33] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.
[34] R. Selby and A. Porter, "Learning from Examples: Generation and Evaluation of Decision Trees for Software Resource Analysis," *IEEE Trans. Software Eng.*, vol. 14, no. 2, pp. 1,743-1,747, Feb. 1988.
[35] N. Schneidewind, "Methodology for Validating Software Metrics," *IEEE Trans. Software Eng.*, vol. 18, no. 5, pp. 410-422, May 1992.
[36] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley Series in Computer Science, second edition, 1991.
[37] D.A. Young, *Object-Oriented Programming with C++ and OSF/MOTIF*, Prentice Hall, 1992.

Victor R. Basili is a professor in the Institute for Advanced Computer Studies and the Computer Science Department at the University of Maryland, College Park, Maryland, where he has served as chairman for six years.

He is one of the founders and principals in the Software Engineering Laboratory, a joint venture between NASA Goddard Space Flight Center, the University of Maryland, and Computer Sciences Corporation, established in 1976. He is a recipient of the first Process Improvement Achievement Award by the IEEE and the SEI, and the NASA Group Achievement Award.

He has served as editor-in-chief for the *IEEE Transactions on Software Engineering*; general chair of the 15th International Conference on Software Engineering in 1993, in Baltimore, Maryland; program chair for the Sixth International Conference on Software Engineering in 1982, in Japan; and general/program chair for several other conferences. He is an IEEE fellow and member of the IEEE Computer Society and Golden Core.

Lionel C. Briand received the BS degree in geophysics and the MS degree in computer science from the University of Paris VI, France, in 1985 and 1987, respectively. He received the PhD degree, with high honors, in computer science from the University of Paris XI, France, in 1994. Dr. Briand started his career as a software engineer at CISI Ingénierie, France. Between 1989 and 1994, he was a research scientist at the NASA Software Engineering Laboratory, a research consortium: NASA Goddard Space Flight Center, University of Maryland, and Computer Science Corporation. He then held the position of lead researcher of the software engineering group at CRIM, the Computer Research Institute of Montreal, Canada. He is currently the head of the Quality and Process Engineering Department at the Fraunhofer Institute for Experimental Software Engineering, an industry-oriented research center located in Rheinland-Pfalz, Germany. His current research interests and industrial activities include measurement and modeling of software development products and processes, software quality assurance, domain-specific architectures, reuse, and reengineering.

Walcélio L. Melo received the following degrees in computer science: BSc in 1983 from the University of Brasilia, Brazil; MSc in 1988 from the Federal University of Rio Grande do Sul, Brazil; and a PhD in 1993 from Joseph Fourier University, Grenoble, France. He is now the lead researcher in software engineering at the Centre de Recherche Informatique de Montréal (CRIM), Canada. At CRIM, he develops research and technology transfer projects related to software measurement, software reuse, object-oriented technology, software architecture, and software process modeling. He is also an adjunct professor at the McGill University School of Computer Science and a former member of the NASA Software Engineering Laboratory. From 1983 to 1989, he worked as a system analyst for the Brazilian Federal Bureau of Data Processing, where he developed several business-oriented software applications. From 1989 to 1993, he developed research related to software process programming languages at the Software Engineering Laboratory of Grenoble, France. From 1994 to 1996, he worked as a research associate at the University of Maryland Institute for Advanced Computer Studies, where he developed research projects related to software maintenance, software metrics, and object-oriented technology. He is a member of the IEEE Computer Society.

OmIT

Comments on "Towards a Framework for Software Measurement Validation"

Sandro Morasca, *Member, IEEE Computer Society*, Lionel C. Briand, Victor R. Basili, *Fellow, IEEE*, Elaine J. Weyuker, Member, *IEEE Computer Society*, and Marvin V. Zelkowitz, *Senior Member, IEEE*

Abstract—A view of software measurement that disagrees with the model presented in a recent paper by Kitchenham, Pfleeger, and Fenton, is given. Whereas Kitchenham, Pfleeger, and Fenton argue that properties used to define measures should not constrain the scale type of measures, we contend that that is an inappropriate restriction. In addition, a misinterpretation of Weyuker's properties is noted.

Index Terms—Software measurement, measurement theory, measurement scales, axiomatic approaches, software complexity properties.

1 Introduction

KITCHENHAM, PFLEEGER, AND FENTON [3] questioned the way properties have been used in the literature to assess software measures. We have two main comments on their criticisms.

2 Attribute Properties and Measurement Scales

First, the authors propose that properties that imply or exclude any particular measurement scale in the definition of a measure cannot be used. This is stated clearly in the following paragraph ([3] p. 932, last paragraph):

"2) Since an attribute may be measured in many different ways, attributes are independent of the unit used to measure them. Thus, any definition of an attribute that implies a particular measurement scale is invalid. Furthermore, any property of an attribute that is asserted to be a *general* property but implies a specific measurement scale must also be invalid."

Adherence to this model will seriously impede the appropriate definition of attributes, particularly when there is a well understood intuition. There is no problem defining properties that at least permit ordering over the set of entities. Without such properties, we end up abstracting away all relevant structure from our model, limiting our ability to say anything of interest. It is true that such properties would only be relevant for measures of the attribute that are defined on an ordinal scale or higher. Nevertheless, this does not make such properties invalid, as stated in [3]. Experimental physics has successfully relied on attributes such as temperature that imply measurement scales in the definition of their properties. Other examples will help clarify our point.

Consider the notion of the size of an object. Our intuitive understanding about the concept of size is that when one "puts together" two objects O_1 and O_2 to obtain a third object O_3, then the size of O_3 is not smaller than the size of O_1 or O_2. (Although the operation of "putting together" may be formally defined, for brevity's sake, we do not provide such a definition here.) Since this simple property is not appropriate for nominal scale size measures, it would be considered invalid for any size attribute, according to [3]. However, the usual understanding of the attribute size can be formalized through properties of size measures requiring at least the possibility of comparing objects' sizes. Our understanding of the attribute size may go even further. In fact, in [2], we propose simple and widely-acceptable properties that imply the ratio scale.

As a second example, consider the notion of distance between two elements of a set S. According to the standard definition, distance is defined as a function:

$$d: \quad S \times S \rightarrow R_+$$

(R_+ is the set of nonnegative real numbers) that satisfies the following three axioms:

Axiom 1.	$\forall x, y \in S$	$(d(x, y) \geq 0$ **and** $(d(x, y) = 0 \Leftrightarrow x = y))$
Axiom 2.	$\forall x, y \in S$	$(d(x, y) = d(y, x))$
Axiom 3.	$\forall x, y, z \in S$	$(d(x, y) \leq d(x, z) + d(z, y))$

These axioms exclude

- nominal scales, since they contain the "≤" operator;
- ordinal scales, since they contain the "+" operator;
- interval scales, since Axiom 3's truth value is not invariant under the admissible transformation for interval scales, i.e., Axiom 3 does not imply the following formula (R is the set of real numbers):

$$\forall x, y, z \in S, \forall a > 0, \forall b \in R\ (a \bullet d(x, y) + b \leq a \bullet d(x, z) + b + a \bullet d(z, y) + b)$$

Axiom 3's truth value is invariant under the admissible transformation for the ratio scale, i.e., Axiom 3 implies the following formula

$$\forall x, y, z \in S, \forall a > 0\ (a \bullet d(x, y) \leq a \bullet d(x, z) + a \bullet d(z, y)).$$

Therefore, Axiom 3 implies the ratio scale, and hence, according to [3] the three axioms usually provided for distance are invalid. This view of meaurement is so narrow and restrictive that it limits our ability to define properties that adequately characterize attributes, even for very well-understood attributes.

We therefore conclude that acceptance of the perspective proposed in [3] has important consequences, including:

1) If we discard some properties, we may be discarding a good deal of relevant information about the attribute. Therefore, our modeling of the attribute will not be as accurate as it could be.
2) If we discard some properties, we will have a less powerful mechanism for checking whether a function that is proposed as a measure for an attribute actually is a measure for that attribute.

It is certainly not true that all attributes can be appropriately defined by properties that imply the ratio, interval, or even ordinal scale (e.g., the color of a physical object). However, as argued above, this does not imply that we should forbid any attributes from being defined by properties that do imply a particular measurement scale or prevent some measurement scales.

Although some attributes used in software engineering (e.g., complexity, cohesion, coupling) are not as well-understood as distance or size, it does not follow that we should prohibit the use of properties that constrain the scale type of a measure. Indeed, an

- *S. Morasca is with the Dipartimento di Elettronica e Informazione, Politecnico di Milano, Piazza Leonardo da Vinci 32, I-20133 Milano, Italy. E-mail: morasca@elet.polimi.it.*
- *L.C. Briand is with the Fraunhofer Institute for Experimental Software Engineering (FhG IESE), Sauerwiesen 6, D-676761, Kaiserslautern, Germany. E-mail: briand@iese.fhg.de.*
- *V.R. Basili and M.V. Zelkowitz are with the Computer Science Department, University of Maryland, College Park, MD 20742. E-mail: {basili, mvz}@cs.umd.edu.*
- *E J. Weyuker is with AT&T Research-Labs, Room 2A-429, 600-700 Mountain Ave., P.O. Box 636, Murray Hill, NJ 07974. E-mail: weyuker@research.att.com.*

Manuscript received Aug. 12, 1996.
Recommended for acceptance by D.R. Jeffery.
For information on obtaining reprints of this article, please send e-mail to: transse@computer.org, and reference IEEECS Log Number S96132.

important purpose of using properties as a means of defining measures is to help codify intuition and make underlying assumptions explicit. In fact that is exactly why Euclid introduced the axiomatic method for geometry more than 2,000 years ago.

3 Weyuker's Properties

Another point of this paper involves criticisms of the properties Weyuker proposed in [4]. First, the authors repeat Zuse's statement [5] that Weyuker's axioms are inconsistent from a Measurement Theory point of view ([3] p. 932, last paragraph):

> "Thus, while Zuse criticises Weyuker's complexity measure properties as contradictory because one (property 5) implies a ratio scale and another (property 6) explicitly excludes a ratio scale ..."

They describe Weyuker's properties as "disputed" and "caution researchers to avoid justifying measures on the basis of either disputed properties or ..." As argued in [1], a careful reading of Zuse's book demonstrates that Zuse's criticisms are unfounded. Concisely, in [1] we show that Zuse's criticisms only prove that Weyuker's properties are not compatible with the fact that the underlying empirical system of a measure assumes an extensive structure. However, the fact that the underlying empirical system of a measure assumes an extensive structure is a sufficient condition to obtain a ratio scale measure, but is by no means a necessary one. Although Zuse refers to Weyuker's properties as contradictory, they are not contradictory in the usual mathematical sense of being incapable of being satisfied at the same time. Some of the properties do require the ratio scale, but there is nothing inappropriate about this.

In addition to Zuse's criticism, another erroneous criticism is introduced in [3], p. 939.

> "3) *Each unit of an attribute contributing to a valid measure is equivalent*. This seems to be standard measurement practice. Weyuker's property 7 relates to this issue. She, in fact, asserts the *converse* of this assumption by claiming that program complexity should be responsive to the order of statements in a program. It seems here that Weyuker is confusing the attributes program correctness and/or psychological complexity with structural complexity. It is unlikely that a random re-ordering of program will be correct or understandable, but a re-ordering would not necessarily be more structurally complex."

Weyuker's property 7 asserts that there exist two programs P and Q, where Q is a re-ordering of P, such that the complexity of P is *different* from the complexity of Q [4]. This property does not assert that, by re-ordering a program, one obtains a new program which would necessarily be *more* or *less* structurally complex than the original one. Weyuker's property 7 states that program complexity may be responsive to the order of statements. It does not contradict the statement made by Kitchenham, Pfleeger, and Fenton:

> "a re-ordering would not necessarily be more structurally complex."

Just as Zuse's criticism in [5] with respect to Axioms 6, 7, and 9 was caused by a misinterpretation or misrepresentation, Kitchenham, Pfleeger, and Fenton have misinterpreted Weyuker's axiom 7.

References

[1] L.C. Briand, K. El Emam, and S. Morasca, "On the Application of Measurement Theory in Software Engineering," *Empirical Software Eng.: An Int'l J.*, vol. 1, no. 1, pp. 61-88, 1996.

[2] L.C. Briand, S. Morasca, and V.R. Basili, "Property Based Software Engineering Measurement," *IEEE Trans. Software Eng.*, vol. 22, no. 1, pp. 68–86, Jan. 1996.

[3] B. Kitchenham, S.L. Pfleeger, and N. Fenton, "Towards a Framework for Software Validation Measures," *IEEE Trans. Software Eng.*, vol. 21, no. 12, pp. 929–944, Dec. 1995.

[4] E.J. Weyuker: "Evaluating Software Complexity Measures," *IEEE Trans. on Software Eng.*, vol. 14, no. 9, pp. 1,357–1,365, Sept. 1988.

[5] H. Zuse, *Software Complexity: Measures and Methods*. De Gruyter, 1991.

omit

Response to: Comments on "Property-Based Software Engineering Measurement: Refining the Additivity Properties"

Lionel C. Briand, Sandro Morasca, *Member, IEEE Computer Society*, and Victor R. Basili, *Fellow, IEEE*

1 Introduction to the Response

Poels and Dedene have correctly identified a few inconsistencies related to the use of the union operator for modules of a modular system we defined in [1]. We gratefully acknowledge their comments, which show the increasing interest in the laying of theoretical bases for Software Engineering measurement.

We first show how the inconsistencies identified can be easily fixed without any conceptual change or addition to the axiomatic measurement framework proposed in [1]. Then, we discuss the more complex alternative proposed by Poels and Dedene.

The problems in [1] are listed below. Points 2 and 4 are actual inconsistencies, while points 3 and 5 are just redundancies. In what follows, we first report the original text of [1]; then we show how it should be corrected.

2 Explanation of Fig. 1—p. 70

Original Text

Union. The union of modules $m_i = <E_{mi}, R_{mi}>$ and $m_j = <E_{mj}, R_{mj}>$ (notation: $m_i \cup m_j$) is the module $<E_{mi} \cup E_{mj}, R_{mi} \cup R_{mj}>$. In Fig. 1, the union of modules m_1 and m_3 is module $m_{13} = <\{a, b, c, d, e, f, g, i, j, k, m\}, \{<b, a>, <b, f>, <c, d>, <c, g>, <d, f>, <e, g>, <f, i>, <f, k>, <g, m>, <i, j>, <k, j>\}$ (area filled with ▨ or ▦ or ▧).

Modifications

Relationship $<c, b>$ does not belong to the set of relationships of module m_{13}, the union of modules m_1 and m_3.

3 Property Cohesion.4—p. 77

Original Text

Property Cohesion.4: **Cohesive Modules.** Let $MS' = <E, R, M'>$ and $MS'' = <E, R, M''>$ be two modular systems (with the same underlying system $<E, R>$) such that $M'' = M' - \{m'_1, m'_2\} \cup \{m''\}$, with $m'_1 \in M'$, $m'_2 \in M'$, $m'' \notin M'$, and $m'' = m'_1 \cup m'_2$. (The two modules m'_1 and m'_2 are replaced by the module m'', union of m'_1 and m'_2.) If no relationships exist between the elements belonging to m'_1 and m'_2, i.e., $InputR(m'_1) \cap OutputR(m'_2) = \emptyset$ **and** $InputR(m'_2) \cap OutputR(m'_1) = \emptyset$, then

$[\max\{Cohesion(m'_1), Cohesion(m'_2)\} \geq Cohesion(m'') \mid Cohesion(MS') \geq Cohesion(MS'')]$ (Cohesion.IV □

Modifications

The additional condition "If no relationships exist between the elements belonging to m'_1 and m'_2, i.e., $InputR(m'_1) \cap OutputR(m'_2) = \emptyset$ **and** $InputR(m'_2) \cap OutputR(m'_1) = \emptyset$" is redundant. It is already implied by the fact that $m'' = m'_1 \cup m'_2$.

4 Property Coupling.4—p. 78

Original Text

Property Coupling.4: **Merging of Modules.** Let $MS' = <E', R', M'>$ and $MS'' = <E'', R'', M''>$ be two modular systems such that $E' = E''$, $R' = R''$, and $M'' = M' - \{m'_1, m'_2\} \cup \{m''\}$, where $m'_1 = <E_{m'1}, R_{m'1}>$, $m'_2 = <E_{m'2}, R_{m'2}>$, and $m'' = <E_{m''}, R_{m''}>$, with $m'_1 \in M'$, $m'_2 \in M'$, $m'' \notin M'$, and $E_{m''} = E_{m'1} \cup E_{m'2}$ and $R_{m''} = R_{m'1} \cup R_{m'2}$. (The two modules m'_1 and m'_2 are replaced by the module m'', whose elements and relationships are the union of those of m'_1 and m'_2.) Then

$[Coupling(m'_1) + Coupling(m'_2) \geq Coupling(m'') \mid Coupling(MS') \geq Coupling(MS'')]$ (Coupling.IV) □

Modifications

The condition must be modified as follows.

Let $MS' = <E', R', M'>$ and $MS'' = <E'', R'', M''>$ be two modular systems such that $E' = E''$, $R' = R''$, and $M'' = M' - \{m'_1, m'_2\} \cup \{m''\}$, where $m'_1 = <E_{m'1}, R_{m'1}>$, $m'_2 = <E_{m'2}, R_{m'2}>$, and $m'' = <E_{m''}, R_{m''}>$, with $m'_1 \in M'$, $m'_2 \in M'$, $m'' \notin M'$, and $E_{m''} = E_{m'1} \cup E_{m'2}$ and $R_{m''} = R_{m'1} \cup R_{m'2} \cup \{<e_1, e_2> \in R \mid (e_1 \in E_{m1}$ *and* $e_2 \in E_{m2})$ *or* $(e_1 \in E_{m2}$ *and* $e_2 \in E_{m1})\}$. (The two modules m'_1 and m'_2 are replaced by the module m'', whose elements and relationships are the union of those of m'_1 and m'_2.)

If $m'' = m'_1 \cup m'_2$, then there would be no relationships in m'' connecting elements that originally were in m'_1 and m'_2.

5 Property Coupling.5—p. 79

Original Text

Property Coupling.5. **Disjoint Module Additivity.** Let $MS' = <E, R, M'>$ and $MS'' = <E, R, M''>$ be two modular systems (with the same underlying system $<E, R>$) such that $M'' = M' - \{m'_1, m'_2\} \cup \{m''\}$, with $m'_1 \in M'$, $m'_2 \in M'$, $m'' \notin M'$, and $m'' = m'_1 \cup m'_2$. (The two modules m'_1 and m'_2 are replaced by the module m'', union of m'_1 and m'_2.) If no relationships exist between the elements belonging to m'_1 and m'_2, i.e., $InputR(m'_1) \cap OutputR(m'_2) = \emptyset$ **and** $InputR(m'_2) \cap OutputR(m'_1) = \emptyset$, then

$[Coupling(m'_1) + Coupling(m'_2) = Coupling(m'') \mid Coupling(MS') = Coupling(MS'')]$ (Coupling.V) □

• *L.C. Briand is with the Fraunhofer Institute for Experimental Software Engineering (IESE), Sauerwiesen 6, D-67661 Kaiserslautern, Germany. E-mail: briand@iese.fhg.de.*
• *S. Morasca is with the Politecnico di Milano, Sede di Como, Piazzale Gerbetto 6, I-22100 Como, Italy. E-mail: morasca@elet.polimi.it.*
• *V.R. Basili is with the Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, MD 20742. E-mail: basili@cs.umd.edu.*

Manuscript received Feb 10, 1997; revised Mar. 18, 1997.
Recommended for acceptance by S.H. Zweben.
For information on obtaining reprints of this article, please send e-mail to: transse@computer.org, and reference IEEECS Log Number S96154.

1997 IEEE. Reprinted, with permission, from IEEE Transactions on Software Engineering, vol. 23, no. 3, pp. 196-197; March 1997

Modifications

The additional condition "If no relationships exist between the elements belonging to m'_1 and m'_2, i.e., $\text{InputR}(m'_1) \cap \text{OutputR}(m'_2) = \varnothing$ **and** $\text{InputR}(m'_2) \cap \text{OutputR}(m'_1) = \varnothing$" is redundant. It is already implied by the fact that $m'' = m'_1 \cup m'_2$.

6 Discussion

As Poels and Dedene point out, it is important that inconsistencies be identified and removed. This will allow for a better understanding and refinement of the axiomatic framework proposed in [1]. In turn, it will lead to a more rigorous definition of software attributes and better measurement.

In their comments, Poels and Dedene have explored additivity, one of the most important and studied property in measurement. They propose the introduction of a new union operator for modules. They substantiate their idea by the fact that it is important to discriminate between modules that are disjoint and modules that, in addition to being disjoint, are not connected.

However, it is our position that we need to keep the set of operators as small as possible, since this will make it easier for researchers and practitioners to understand and discuss the properties proposed in [1] for different software attributes. That is why we introduced only a few operators (union, intersection, empty module, etc.) for modules, whose syntax and semantics were intentionally kept close to the syntax and semantics for sets, for which these operators are usually applied. One more union operator would be redundant, i.e., it would not add much expressiveness to the module "algebra" defined in [1]. If needed, the new union operator proposed by Poels and Dedene can be defined based on the other module operators and set operators.

Reference

[1] L.C. Briand, S. Morasca, and V.R. Basili, "Property-Based Software Engineering Measurement," *IEEE Trans. Software Eng.*, vol. 22, no. 1, pp. 68–86, Jan. 1996.

S3-61
415737
360916
12P

Analytical and Empirical Evaluation of Software Reuse Metrics*

Prem Devanbu, Sakke Karstu, Walcélio Melo and William Thomas

Abstract

How much can be saved by using existing software components when developing new software systems? With the increasing adoption of reuse methods and technologies, this question becomes critical. However, directly tracking the actual cost savings due to reuse is difficult. A worthy goal would be to develop a method of measuring the savings *indirectly* by analyzing the code for reuse of components. The focus of this paper is to evaluate how well several published software reuse metrics measure the "time, money and quality" benefits of software reuse. We conduct this evaluation both *analytically* and *empirically*. On the *analytic* front, we introduce some properties that should arguably hold of any measure of "time, money and quality" benefit due to reuse. We assess several existing software reuse metrics using these properties. *Empirically*, we constructed a toolset (using GEN++) to gather data on all published reuse metrics from C++ code; then, using some productivity and quality data from "nearly replicated" student projects at the University of Maryland, we evaluate the relationship between the known metrics and the process data. Our empirical study sheds some light on the applicability of our different analytic properties, and has raised some practical issues to be addressed as we undertake broader study of reuse metrics in industrial projects.

1 Introduction

Software reuse is considered to be one of the most promising approaches for increasing productivity. By re-using existing software, in addition not having to re-implement it, one can avoid downstream costs of maintaining additional code, and (if the re-used artifact has been thoroughly tested) increase the overall quality of the software product. Several industrial and governmental initiatives are underway to increase the reuse of software, involving both adjustments to process, and the adoption of new technologies. As these efforts mature, it is very important to demonstrate to management and funding agencies that reuse makes good business sense; to this end, it is necessary to have methods to gather and furnish clear financial evidence of the benefits of reuse in real projects. Thus, we need to define good *metrics* that capture these benefits, and develop tools and processes to allow the effective use of these metrics.

*Devanbu is with the Software & Systems Research Laboratory, AT&T Bell Laboratories, 600 Mountain Av., Murray Hill NJ 07974, USA. Karstu is with Michigan Technological University, Houghton, MI. Melo and Thomas are with the University of Maryland, Institute for Advanced Computer Studies and Computer Science Dept., College Park, MD 20742 USA.

We can think of *reuse benefit* of a project or system, as being the normalized (percentage) financial gain due to reuse. This is an example of an *external* process attribute (see [8]), concerned with an external input (money) into the software development process. Unfortunately, the direct measurement of the actual financial impact of reuse in a system can be difficult. The project as a whole may not have the machinery in place to gather financial data. There are also other difficulties associated with measuring the financial impact of reuse. There are different types of reuse—reuse of specifications, of design, and code. Specification and design processes often have informal products (such as natural language documents) which can be quite incommensurate. Even in reuse of code, there are different *modus operandi*, from the primitive "cut, edit, and paste", to the formal, controlled language based approaches provided in languages such as C++ and ML. In any case, to determine cost savings, one may have to ask individual developers to estimate the financial benefit of the code that they reused. This information may be unreliable and inconsistent.

Fortunately, one of the key approaches to reuse is the use of features such as functions and modules in modern programming languages. In this context, one can find evidence of (some kinds of) reuse directly in the code; thus, it may be possible to find an *indirect* measure of the benefits of software (code) reuse *directly in the code*. Measures derivable directly from the code are *internal* measures. Several such measures of software reuse have been proposed in the literature [2, 4, 9, 11, 14]. This paper is concerned with the evaluation of how well various *indirect, internal measures* of software reuse actually measure the relevant *external process attribute*: reuse benefit.

The rest of the paper is organized as follows. First, following the lead of Weyuker [16] in the field of complexity measures, we develop some *general properties* or axioms that (we argue) should apply to any measure of reuse benefit. Although (for reasons discussed above) it is difficult to develop a direct, external measure of reuse benefit, these axioms give us a yardstick to evaluate candidate *internal* measures. We then look at the internal measures of reuse reported in the literature and analytically examine their relationship to these properties. Finally, we describe an empirical evaluation of these metrics. We have constructed tools to gather the internal metrics, and methods to gather corresponding process data. We use statistical methods to assess the relationship of the various internal metrics with the corresponding process data. The results suggest some possible improvements to the published internal measures of software reuse. This paper is aimed at establishing a broad framework to assist in the study of

1996 IEEE. Reprinted, with permission, from Proceedings of the 18th International Conference on Software Engineering (ICSE-18); March 1996; pp. 189-199

reuse metrics, covering: a) the formulation of analytic properties, b) analytic evaluation of published metrics, c) construction of metrics gathering tools, and d) empirical evaluation which in turn sheds some light on the analytic properties.

2 Indirect Measurement of Reuse Benefit

Fenton [8] categorizes software measures along two orthogonal axes. The first is the process/product axis: a metric may measure an attribute of software *product*, (*e.g.,* quality of code), or an attribute of software *process* (*e.g.,* cost of design review meetings). Another, orthogonal axis is the internal/external axis. A metric may measure an *internal* attribute (*e.g.,* the number of loops in a module), or an *external* attribute (*e.g.,* maintainability of a module). Our goal is to develop a reasonable way of measuring the actual financial impact of reusing software. By Fenton's categorization, this is an external process attribute. We would like to measure reuse benefit as a normalized measure of the degree of cost savings achieved by adopting software reuse. Thus, we define R_b, the reuse benefit of a system S, in terms of development cost (C) as follows:

$$R_b(S) = \frac{C(S\ without\ reuse) - C(\ S\ with\ reuse)}{C(S\ without\ reuse)} \tag{1}$$

It is important to note here that we are really concerned with the cost of development, which is quite different from the incremental benefit to revenue from the product. It may be possible that by doing reuse, we bring out the product to market earlier, and with greater functionality. This may well increase revenue. Our model ignores this: R_b is solely concerned with the effect on coding costs.

For reasons given in the introduction, it can be difficult to get a reasonable direct measure of R_b. When direct measurement is difficult, *indirect* measures have been used. For example, the external process attribute of *maintainability* is often measured indirectly by internal product measures of *complexity* such as cyclomatic complexity. Likewise, the internal product measure of software *size* (in units of NCSL) is considered to be a reasonable indirect measure of the external process attribute of *development cost*. Following this approach, we are concerned with the development of an indirect internal measurement of R_b, the reuse benefit of a system S, from the product, by searching the source code of S for instances of language-based reuse such as subroutine calls.

With such an indirect measure, there is a risk that we are not really measuring what we seek to measure; we would therefore like to validate our indirect measure in some way. One approach to validating indirect measures is to perform empirical studies, whereby one gathers statistical data about both the indirect and direct measures of the attribute in question, and tries to show that there are some correlations between the direct and indirect measures, and perhaps construct a regression model. A parallel (or perhaps preceding) approach, proposed by Weyuker [16] and others is to enumerate some formal properties that should hold of any measure (direct or indirect) of the attribute in question. Then, given a candidate measure, one can evaluate whether these properties apply to it. Weyuker used this approach to evaluate several internal measures of complexity. Of course, we are using this approach differently than Weyuker: she "axiomatized[1]" properties of a complexity internal measure, and evaluated several internal complexity measures against these properties. We are seeking to "axiomatize" an external measure—reuse benefit—and use these "axioms" to evaluate and develop indirect internal measures of reuse benefit. In addition, measuring reuse benefit is quite different from measuring complexity; thus many of her axioms aren't relevant in our context. However, her Property 4 (implementation dependence) is critically important in measuring reuse, and in fact, we reformulate and strengthen Property 4 in several ways applicable particularly to measures of reuse benefit.

We begin with some notation, and present some "axioms", moving from the simple to the more complex.

2.1 Notation

Some definitions of the terminology that will be used in this paper:

S_i = A software system or a subsystem whichever is appropriate, subscript i is to distinguish the systems from one another.

c_j = A software component (module, class, function, subsystem). With a superscript e (e.g.,) "c_j^e" refers to an *e*xternal component, which existed independently of the system in which it is being used.

$Cu(S_1, c_1)$ = The number of times component c_1 is used in a system S_1

$Cost(X)$ = The cost of developing system or component X. it may often be hard to determine the actual cost; we use size as an indirect measure of cost.

$Function(S)$ = The "meaning" of the system S, from the customer's point of view. Two systems S_1 and S_2 are equivalent for a customer if

$$Function(S_1) = Function(S_2) \tag{2}$$

We also use (2) to denote equivalence of components.

Before we present our "axioms" of reuse benefit, it is important to emphasize that our goal here is precisely *not* to claim that our properties are the final and complete word on reuse benefit measures; we simply offer them as a candidate set for further additions/modifications.

[1] We use the quotation marks here because these are not necessarily axioms in the formal mathematical sense, but rather a list of properties that would appear to most people to hold of the measures in question.

2.2 Minimal and Maximal R_b

To begin with, we'd like to postulate what the maximum and minimum possible values of reuse benefit are. First, consider the system which uses no external components, and uses each internal component at most once. Such a system does not derive any cost savings from reuse, and should have a reuse benefit of zero. It is certainly possible (if silly) to construct such a system $\hat{S}$ which gives us the minimal possible value of R_b, when:

$$i.e., Cu(\hat{S}, c_j^e) = 0 \ \& \ Cu(\hat{S}, c_k) \leq 1$$

for all internal components c_k and all external components c_j^e. In this case,

$$R_b(\hat{S}) = 0$$

This is a little optimistic: it is also possible that there is actually a negative R_b. We might have a case where component provides only a very trivial functionality, and/or is very difficult to locate and understand, and/or involves a great deal of set-up or "glue" code to use. For the purposes of this paper, we assume that we only have "rational" re-use, and that there is actually a net positive benefit to every re-used component, perhaps after some number of re-uses.

Now, for the maximal value (or upper bound) we consider a system that is built in its entirety by reusing external components. Such a system would still need some "glue" to tie all the external components together; writing the "glue" would involve some (possibly very small) additional cost. So the maximal value of reuse benefit would be strictly less than one[2]. Thus, we have, for any system S:

Property 1 $\forall S, \ 0 \leq R_b(S) < 1$

2.3 Implementation Dependence

Weyuker's Property 4 [16] asserts that there are systems with the same function, but different complexity measures (based on the implementation style). This *implementation dependence* is a crucial aspect that we demand of any good measure of reuse benefit. Clearly, it is possible to produce the same functionality with and without reuse. Our measure *must* be able to distinguish between one that enjoys a great deal benefit of from reuse and one that doesn't. Thus, we insist that:

Property 2
$\exists$ $S1, S2$ *such that* $Function(S_1) = Function(S_2)$ *but* $R_b(S_1) \neq R_b(S_2)$

Property 2 simply states that R_b's may be different for different implementations; we need to make a stronger requirement for a reuse benefit measure. We want to be able to compare different implementations, and see which one is better or worse with respect to reuse. For example, given a system S with a nonzero reuse benefit, we should be able to find a way to syntactically perturb S, eliminate some reuse, and create a system $\tilde{S}$ that is functionally identical, but has less reuse.

[2] If it were one, that would mean that we are simply using an entire existing system.

Property 3 $\forall$ S *such that* $R_b(S) > 0$,
$\exists$ $\tilde{S}$ *such that* $Function(S) = Function(\tilde{S})$ *and* $R_b(S) > R_b(\tilde{S})$

Property (3) is fundamentally important. It says that by changing the implementation, you can increase (or reduce) reuse while maintaining functionality. Using this property, we can successively consider different implementation techniques that increase reuse in a system, and demand that each of these show a corresponding increase in any good measure of reuse benefit. However, in the ensuing discussion, we always perturb an existing system by *eliminating* some reuse, while leaving the functionality untouched. We then demand that this perturbation reduce the R_b. This approach simplifies the analysis of the desired impact on the reuse benefit. The rest of this section considers different kinds of reuse implementation techniques in turn and develops a specialization of (3) for each technique.

First, we can expect that a reuse benefit measure will be sensitive to the number of times a component is reused. Thus, suppose we have a system S where a component c is reused n times (for $n \geq 2$, in case it is an internal component: it must be used at least twice to be considered reused). We denote this system by S_c^n. Now suppose we create a mutation of this system, with functionality identical to it: S_c^{n-1}, by eliminating one reuse of the component c, and re-implementing the functionality by "open-coding" c; we also assume that the usage of the other components is unaffected. We can now demand the following axiom of a candidate R_b measure[3]

Property 4 $\forall S, c \ \ R_b(S_c^n) > R_b(S_c^{n-1})$

Reuse benefit measures should also be sensitive to the cost of the component being reused. Reusing a more expensive component is more beneficial than reusing a cheaper component. Consider a system S, which reuses two components C and c each at least once; also assume that $Cost(C) > Cost(c)$. Now consider two perturbations of S, S_C^- and S_c^-. S_C^- (respectively, S_c^-) is created from S by *eliminating* one reuse of C (respectively, c) and re-implementing its functionality. Now we can say:

Property 5 *If* $Cost(C) > Cost(c)$ *then*
$R_b(S_c^-) > R_b(S_C^-)$

It should also be the case that reusing external components is in general better than reusing internal components. Thus consider a system which uses an external (pre-existing) component c^e for a certain functionality (irrespective of how often it is reused). We denote this by S_{c^e}. Now consider a perturbation of S, which replaces c by a custom-implemented, (for this system) equivalent component c. Call this new system S_c, which we will assume has the same functionality. In this case, we demand that:

[3] This axiom doesn't account for initial difficulties (during the first several reuses) involved in learning about an external component (or implementing it in a re-usable fashion, if it is an internal component).

Property 6 $R_b(S_{c^e}) > R_b(S_c)$

Consider another system $S_{c^e,n}$, where the external component c^e is used n times. Now we eliminate the n^{th} reuse of an external component c^e, and replace it with a use of a different, identical external component $\acute{c}^e$, thereby yielding system $S_{c^e,n-1,\acute{c}^e}$ This often happens in large systems: a careless developer, unaware of a previously incorporated external component that performs a certain function, incorporates a distinct, but functionally identical one again from an external repository or library. The incorporation of this new code involves needless additional work to identify, procure, and validate the component; therefore, the added extra component should not increase the benefit from reuse:

Property 7 $R_b(S_{c^e,n}) \geq R_b(S_{c^e,n-1,\acute{c}^e})$

Finally, we have an axiom that relates to "cut & paste" reuse. For this, consider a system S with three variants that are functionally identical: S_ϕ S_{c^m} and S_{c^v}. S_ϕ is implemented by simply adding custom-crafted code to S. S_{c^m} is implemented by obtaining a component c^m from somewhere (internal or external) modifying it "slightly" (see below) and linking it into S. S_{c^v} is created in a similar manner to S_{c^m}, except that an additional verbatim use c^v has been included to implement it. In this case, we should expect that verbatim reuse is better than "cut & paste" reuse which is better than no reuse at all:

Property 8 $R_b(S_{c^v}) > R_b(S_{c^m}) > R_b(S_\phi)$

Since the term "slightly modified" is hard to define, Property 8 can be particularly difficult to measure in a repeatable way; perhaps for this reason, most published measures ignore this property, with the exception of [2]. However, as discussed below, our empirical study suggests that this is an important property.

Most existing measures of reuse benefit turn out to be not strictly consistent with one or more of the properties listed above; in fact, as we shall see below, there are some inherent difficulties in any approach to measuring reuse.

3 Analytic Evaluation of Reuse Metrics

There are many models and metrics [3, 4, 5, 10, 11, 14] in the literature that try to evaluate the degree of reuse in a software system. Most of these measures are concerned with estimating the actual financial benefits due to reuse. Bieman [3] suggests a range of measures of various reuse occurrences in object-oriented software. Our theoretical framework, as well as the empirical study, is concerned more with measures that yield a single number that could potentially estimate the savings due to reuse. In this section we will compare some of these models to our proposed set of properties of reuse benefit measures.

3.1 Producer/Consumer Models

Several researchers [5, 11, 10, 4, 14] seek to evaluate the benefits of reuse in a corporation. They use different models, but essentially, they all comprise a producer-consumer framework. Reusable artifacts are created by the *producer* (*e.g.,* a domain engineering group which produces reusable software) and re-used by several *consumers.* The producer groups have to undertake extra cost burdens to create high-quality reusable assets. Consumers benefit by avoiding re-implementation costs. The return on the asset producer's investment is proportional to use by consumers. Business-case oriented models of reuse metrics seek to measure the overall benefit to the corporation of re-use practices: thus they include measurements of code size, relative cost of producing re-usable software, number of reuses etc, into a unified model that can combine all these numbers into a figure for overall cost benefit of reuse. Gaffney *et al* have investigated different models for computing the financial benefits of reuse [11, 10]. Poulin *et al* [14] have developed and institutionalized a comprehensive reuse program that incorporates a producer/consumer financial model of reuse benefits. Bollinger and Pfleeger [4] propose financial and accounting practices to motivate multi-project reuse, based on the producer/consumer model.

A key component of all these efforts is a model for the amount of savings during the coding phase, directly attributable to reuse. However, the methods used for computing coding-phase savings in [4, 14, 11, 10] do not necessarily conform to the properties presented in § 2.3. For example Poulin [14] gives reuse benefit credit only for external components, and for each reused component just once, regardless of the number of times it is called. His argument is that the cost of implementing the component is saved only once; after that each additional use should not get additional credit. Programmers should be expected to use components that are in the system as a matter of course, and should not get credit for that. Since larger components are given more credit, their treatment of *external component* is consistent with the Property 5. However, the "credit for one use only" assumption is not consistent with our Property (4). For his computation of the cost savings due to re-use, he uses a *product reuse level* number, which is a normalized ratio of the number of lines of reused source instructions (RSI) to the total number of lines. To estimate the actual cost savings (Reuse Cost Avoidance, or RCA) he multiplies the RSI number by a per-line cost savings. Chen *et al* [5] use a very similar computation, but have constructed a repeatable, tool-based measurement apparatus.

Given a project where all the programmers can always be expected to be aware of and likely to use all the re-usable components, Poulin's argument for giving credit only once, to just the linecount of the external components, seems applicable. But in many large, long-lived software systems, with frequent personnel turnover, programmers may be unaware of reusable components, whether internal or external. Conversations with developers have revealed cases where the same function had been re-implemented dozens of times in a very large project. Such practices complicate the calculation of the reuse benefit. As a specific example, consider a 1,000,000 line system S with 400,000 lines of RSI (including a 2000-line component c_1) Now, assume that subsequently, a programmer (unaware of the existence of c_1) creates S_1, with some new functionality, by retrieving and using a component c_2 (with functionality identical to c_1, but implemented differently) of the same length (2000 lines) from an external repository. Now suppose a more careful programmer, creates S_2 from S_1, by adding another reuse of the component c_1. By Property (7), S_1 should be assigned a higher reuse benefit. However, using the RSI count, S_2 would be assigned a higher normalized reuse benefit. Even if the existing component was hard to find (because of poor retrieval support), it is unclear whether the needless introduction of a new external component predicates a greater benefit from reuse.

This kind of needless re-use, by "re-discovering" external components, might inflate the RSI count and thus complicate the return on investment computations. This would appear to present difficulties for both [11] and [14]. Intuitively, the problem seems to arise from the exclusive focus on the reused code (RSI) rather than the *manner in which it is reused in the rest of the code.* Thus simply by inflating RSI, without re-using it effectively, one can get an inflated relative benefit number. On the other hand, consider a system that is implemented without any external components at all, but which incorporates a highly modularized and parametrized architecture which allows a high degree of reuse of internal (custom crafted) components. Such a system would have an RSI of zero, but might well realize high levels of reuse benefit. Our empirical data (See § 4) includes some student projects that illustrate this possibility.

Some of the other measures discussed in this section, notably the measures of Frakes and Terry, and the R_{sf} measure, don't focus solely on the RSI, but give credit for each reuse of a component. Poulin gives examples of spuriously inflated reuse benefit resulting from such measures. Thus both methods are subject to anomalies, albeit in different contexts.

Finally, the RSI measure, (like all measures discussed in this section with the exception of RR from Section § 3.4) does not give any credit for non-verbatim reuse, *i.e,* the reuse of components that have been adapted somewhat; RSI is thus not consistent with Property 8. Our case study suggests that there is a significant degree of benefit from re-using partially modified components.

3.2 Reuse Level Models

Unlike the work described in the previous chapter, which is concerned exclusively with *how much* code is being reused, Frakes and Terry [9] focus on *how* code is being reused. Their *reuse level* and *frequency* measures are concerned with how frequently components are being used. They distinguish between internal and external reuse; total reuse is the sum of these two.

In calculation of their reuse level and reuse frequency, Frakes and Terry use *threshold levels* to determine when a component is considered being reused. A threshold is a value that determines when a module will be reused. If a threshold is 2 then an item that has been used more than two times is considered to be reused. Different threshold values (respectively, ETL and ITL) can be used for external reuse and internal reuse. Given these numbers, the number of internal and external components (resp., IU and EU) which are used more than the threshold can be counted; the total number of components is given by T. Frakes and Terry also count the *frequency* of reuse: the number of references to internal and external items (which are reused more than the threshold) are counted by IUF and EUF, and the total number of references is denoted by TF. Given these numbers, the overall reuse level (RL) and reuse frequency (RF) measures are computed thus:

$$RL = \frac{IU + EU}{T}$$

$$RF = \frac{IUF + EUF}{TF}$$

The RL & RF measures are two different measures of reuse level, which could both be used as indirect measures of reuse benefit. For this purpose, these measures differ from the RSI measure used by [14]; here, there is actually a focus on *how* the reusable components are used, rather than just the total line count of reused code. In addition Frakes and Terry give credit for *both* internal and external components. However, RL and RF are different. After a given threshold value, RL is not sensitive to the number of uses of a particular component; therefore, it does not strictly conform to Property (4). RF, on the other hand, is usage sensitive.

However, these measures are insensitive to the cost of the modules being reused; thus, they don't incorporate Property (5). However, [9] does describe a simple method to weight these measures based on computation of certain ratios of the average sizes of reused modules. While this "size weighting" method accounts for the size to some extent, it is not sensitive to the level of reuse of modules of various sizes. According to Property(5), it is better to reuse larger modules (if size is taken as a good proxy for cost).

Finally, RL and RF only count verbatim reuse; if a slightly modified version of an existing component is used again, it would be treated as a use of a new component; depending on the level at which the threshold is set, this may not be recognized as being re-used. Thus, RL and RF may not always conform to Property 8.

3.3 Size and Frequency Metric - R_{sf}

In this section, we describe another normalized indirect measure of reuse benefit, R_{sf}, first described in [7]. This measure tries to account for both *how much* code is being reused, as well as in *what manner* it is being reused (sf stands for size and *f*requency) It uses a notion of *expanded code size* $Size_{sf}$, which indicates how much code would have to be written to implement the system, had there not been any reuse. The actual code size is denoted by $Size_{act}$. We model our measures in general thus:

$$R_{sf}(S) = \frac{Size_{sf} - Size_{act}}{Size_{sf}} \qquad (3)$$

The form of this equation is almost identical with the form of the equation (1) on page 2. In fact, equation (3) follows directly from equation (1) using a simple two step argument. First, we take the size of a system to be a good indicator of the effort taken to implement (and thus the cost of) the system. Second, we take the expanded size $Size_{sf}$ of the system as a proxy for the cost of the system without reuse, and $Size_{act}$ be a proxy of the actual cost of implementing the system. $Size_{act}$ is simply the number of statements in the newly written functions of implemented system (not counting reused pre-existing code from external repositories). This is a fixed number, computed in the usual way. It should be immediately clear (since $Size_{act}$ is a positive non zero number) that if $Size_{sf} \geq Size_{act}$, the indirect measure defined above conforms strictly to Property (1) on page 3.

The definition of R_{sf} makes use of the function call graph of a program:

Definition 1 *A callgraph $CG(S)$ for a system S is a connected, directed graph rooted at the main procedure, and described by a pair $\langle N_S, E_S \rangle$ where the nodes N_S represent the functions in the system, and the edges E_S represent the function invocations. For each node n in N_S, the in-degree (the number of calls to n) of n is denoted by $calls(n)$, and the code size of n by $size(n)$. $EXT(S)$ is the set of nodes in N_S that represent functions from external libraries, and $INT(S)$ is the rest.*

$$Size_{act}(S) = \sum_{n \in INT(S)} size(n) \qquad (4)$$

$$Size_{sf}(S) = \sum_{n \in N_S} size(n) * calls(n) \qquad (5)$$

$$R_{sf}(S) = \frac{Size_{sf}(S) - Size_{act}(S)}{Size_{sf}(S)} \qquad (6)$$

With this definition, it's easy to see that R_{sf} satisfies Property (1). In the case where there is no external component use, and each internal component is used only once, we get $Size_{sf} = Size_{act}$; in all other cases, $Size_{sf} > Sizeact$, as desired.

The $Size_{sf}$ measure is sensitive both to the size of the function being reused, and the number of times it is being used. It is easy to see that it conforms to Properties (4) and (5) provided we assume that size is a good proxy for cost. We remind the reader here that Properties (2 & 3) are weaker preliminaries to Property (4).

Now consider Property (6). Suppose we have an external function component c^e in S, of size $size(c^e)$ which is used i times ($i > 1$). Now suppose we create $\tilde{S}$ by removing one use of c^e, and re-implementing c^e as a component c^{int} (internal to S); we also make the reasonable assumption that the size of c^e is much larger than the *difference* between $size(c^e)$ and $size(c^{int})$, (*i.e.,*):

$$size(c^e) >> |\, size(c^{int}) - size(c^e)\, | \qquad (7)$$

Under this assumption, we can easily show (the details are omitted here for brevity, and may be found in [7]) that

$$R_{sf}(S) > R_{sf}(\tilde{S})$$

as specified by Property (6).

Now we turn to property (7). Assume that we have a system S with an external function c_1^e, invoked i times ($i > 1$). Now we create a mutation $\hat{S}$, where one use of c_1^e is replaced by a functionally identical new external function c_2^e.

In the case where $size(c_1^e) \geq size(c_2^e)$ we can show a result consistent with Property (7):

$$R_{sf}(S) \geq R_{sf}(\hat{S})$$

Thus, unlike the purely size-sensitive metrics described in § 3.1, R_{sf} doesn't get fooled by the inclusion of a functionally identical component of the same or smaller size. Unfortunately, if the new component is larger, this measure is also fooled, and reports a gain in reuse ! In general, such phenomena as needlessly large components are likely to pose difficulties of any practical tool that derives an indirect reuse benefit measure from the code.

Finally, R_{sf} only counts verbatim reuse. Use of a slightly modified component is not given any reuse credit; it can be easily shown that R_{sf} does not conform to Property 8. We now describe a measure that actually accounts for non-verbatim reuse.

3.4 Reuse Ratio

The reuse ratio has been used for many years in the NASA Software Engineering Laboratory [13]. Recently this metric has been further investigated on object-oriented systems developed in C++ and Ada [2, 15]. It is the only measure examined here that addresses Property 8. This measure is defined for a system S, with components C_i, $i \ldots n$. For each component C_i, we use a $Size(C_i)$, as before. But we now also have a *change ratio* $Change_i$ (where $0 \leq Change_i \leq 1$) which measures what portion of the component has been hand-crafted (added, modified or deleted) for inclusion into S. Thus, for a component C_i drawn from a library and used verbatim, $Change_i$ would be zero, and for a component for which exactly 50% of the code has been rewritten $Change_i$ would be 0.5. In practice, it is difficult to account precisely for the degree of custom coding in a reused component. In [2, 15] this problem has been handled by asking the reuser if 25% or more of a component had been changed; then, the value of $Change_i$ is thresholded as follows (IR is a binary value standing for <u>*is*</u> <u>*reused*</u>)

$$IR(i) = 1 \; if \; Change_i < 0.25 \;, \; 0 \; otherwise$$

Using these, Melo *et al* define *RR*, the reuse ratio measure, thus:

$$RR(S) = \frac{\sum_{C_i \in S} IR(i) * Size(C_i)}{\sum_{C_i \in S} Size(C_i)} \qquad (8)$$

The computation shown in equation 8 is very similar to that used by Poulin *et al* in the product reuse level number. Indeed, if the $IR(i)$'s were all set to zero, except for the components which were reused verbatim, the computation is identical. Thus, the analytical evaluation here is identical to the discussion in § 3.1,

Property	R_{sf}	RL	RF	RSI	RR
P1	X	*	*	*	*
P2	X	X	X	X	X
P3	X	X	X	X	X
P4	X	*	X	-	-
P5	X	-	-	*	*
P6	X	*	*	X	X
P7	*	X	X	-	-
P8	-	-	-	-	*

Table 1: Summary of Reuse Measure Conformance to Reuse Benefit Properties. An "X" indicates that the measure conforms to the property, a "-" indicates non-conformance, and a "*" indicates partial conformance.

except for one vital difference: *RR* is the only measure discussed in this paper that actually conforms to Property 8. Of course, it conforms only for components which are modified 25% or less. This deficiency stems from the difficulty of identifying the "degree of cutting and pasting" in modified components. However, we are experimenting with some new algorithms due to Baker [1] which might lead to repeatable, analytic approaches to quantifying the level of modification.

3.5 Discussion

Table 1 provides a summary of the examined reuse measures in terms of their conformance to the properties listed in section 2.

While all of the examined reuse measures satisfy properties 2 and 3, none of the measures conform to all properties. Property 1 requires that reuse benefit be greater than zero and strictly less than one, however, the definitions of RR and RSI do not preclude a system composed entirely of reused components, so it is possible to have RR or RSI equal to 1. There also are some similarly unusual cases where RL and RF can be equal to 1. So in light of such unusual cases, these measures are listed as only partially conforming to property 1, as they can equal 1, but can never exceed it. The two measures that do not consider internal reuse, (RSI and RR), do not satisfy the property associated with internal reuse, the sensitivity to multiple reuses (Property 4). These also do not satisfy Property 7. In addition, they only partially conform to Property 5, since the size of reused *internal* components is ignored. RL and RF combine internal and external reuse; if ERL and ERL were used, they would conform to Property 6. However, they do not strictly account for the size of the reused components (Property 5). Moving to Property 7: *RL* and *RF* are only affected by the frequency of reuse of components, and are thus not fooled by the needless introduction of new external components, as are *RSI* and *RR*. R_{sf}, can be fooled in some cases, as discussed in Section 3.3, page 6. R_{sf} satisfies all properties except for Property 8, which accounts for the benefit from modifying an existing component. This property is not fully satisfied by any of the measures, and only partially satisfied by RR.

These results suggest that there is room for improvement of these measures. Since there is significant variation in the set of properties satisfied by each reuse measure, we would expect similar variation in the amount and type of benefit that they predict. We re-emphasize here that this is an *a-priori* property formulation. When a large, diverse set of reuse metrics data (with associated process data) becomes available, the validity of these different assumptions can be evaluated. As we shall see, our initial empirical study using student data indicates that some of these properties appear to be quite critical; it also indicates that there are some practical difficulties to be overcome while using some of the metrics listed in table 1.

4 Experimental Validation

In order to experimentally validate the metrics discussed in the previous sections, we examined the degree to which these metrics show an impact on software productivity and quality. To do so, we used the data gathered in study performed at the University of Maryland [2]. Section 4.1 describes the product and process measures that were collected in the study, and Section 4.2 provides a summary of the metrics collected for each of the programs in the study. In section 4.3 we present and interpret results obtained from the statistical analysis performed on the data.

4.1 Data Collected

Both product and process data were gathered as part of this study. We describe here only the product and process data that are relevant to help us validate the suite of reuse metrics presented in this paper. For further detail about how these data were gathered and validated see [2].

4.1.1 Product Data

We have built the software tool infrastructure to gather data about 4 different reuse measures: our R_{sf} metrics, the RSI metric used by Poulin and others, and the RL and RF metrics of Frakes and Terry.

Our tools have 3 elements. First, we have a static analyzer, built with the GEN++ [6] analyzer generator, which analyzes C++ programs and generates call graph and function size information. This information is generated into flat files. These are then processed by a relational database system (Daytona [12]) which supports such features as transitive closure (which is needed to identify a connected call graph), and aggregate queries (which are needed to compute the different summary metrics).

Unfortunately, we did not have a software tool to calculate reuse ratio. We used a form, the component origination form [2], to capture whether a component has been developed from scratch or has been developed from a reused component. In the latter case, we asked the developers to tell us if more or less than 25 percent of a component had been changed. In the former case, the component was labeled: ***Extensively modified*** and in the latter case: ***slightly modified***. If the component was inserted into the system without any modification it was labeled: ***verbatim reuse***. Only ***verbatim reuse*** and ***slightly modified*** have been used to calculate reuse ratio [2].

4.1.2 Effort

Here we are interested in estimating the effort breakdown for development phases, and for error correction. Again, we used forms filled out by the developers to track person-hours expended across development activities. These activities include:

- Analysis. The number of hours spent understanding the concepts embedded in the system before any actual design work. This activity includes requirements definition and requirements analysis. It also includes the analysis of any changes made to requirements or specifications, regardless of where in the life cycle they occur.

- Design. The number of hours spent performing design activities, such as high-level partitioning of the problem, drawing design diagrams, specifying components, writing class definitions, defining object interactions, etc. The time spent reviewing design material, such as walk-throughs and studying the current system design, was also taken into account.

- Implementation. The number of hours spent writing code and testing individual system components

- Rework. This includes the number of hours spent on isolating errors, as well as correcting them.

4.1.3 Number of Defects

Here we analyze the number of defects found for each system/component. We will use the term defect as a generic term, to refer to either an error or a fault. Errors and faults are two pertinent ways to count defects, thus they were both considered in this study. Errors are defects in the human thought process made while trying to understand given information, to solve problems, or to use methods and tools. Faults are concrete manifestations of errors within the software. One error may cause several faults and various errors may cause identical faults. In our study, an error is assumed to be represented by a single error report form; a fault is represented by a physical change to a component.

4.2 Overview of the Projects

Table 2 provides descriptive measures of the projects included in the study, showing the project ID, project size (source lines of code (SLOC)), total lifecycle productivity (SLOC/Hour), fault density (Faults/KSLOC), and error density (Errors/KSLOC).

Table 3 shows for each project the reuse measures discussed in the previous sections: R_{sf}, reuse level, reuse frequency, RSI, and reuse ratio. As one can see, RSI shows very little variation across the projects: most of the projects have RSI equal to zero. Given that, we will not analyze the impact of RSI on productivity and quality, since the poor distribution in our sample can easily bias the statistical analysis.

ID	SLOC	Prod.	Fault Dens.	Error Dens.
1	5105	18.23	8.23	6.46
2	11687	32.02	3.76	3.59
3	10390	34.30	3.95	3.17
4	8173	51.40	8.20	3.18
5	8216	31.12	3.41	3.04
6	9736	69.54	1.64	1.54
7	5255	19.91	14.27	8.37

Table 2: Size, Productivity, Fault Density, and Error Density in the Examined Projects

ID	R_{sf}	RL	RF	RSI	RR
1	0.45	0.52	0.79	0.00	0.02
2	0.86	0.37	0.78	0.08	0.26
3	0.45	0.28	0.64	0.00	0.15
4	0.93	0.52	0.92	0.00	0.40
5	0.74	0.38	0.76	0.11	0.38
6	0.83	0.38	0.76	0.11	0.43
7	0.51	0.45	0.81	0.00	0.00

Table 3: Reuse Measures in the Examined Projects

4.3 Results

To provide some evidence of the usefulness of the measure of reuse benefit, we examined the relationship between reuse benefit and the quality factors of productivity, defect density, and rework effort. The coefficients of correlation between these quality measures and the measures of reuse benefit are shown in table 4. The following sections describe our observations on the relationship between these quality factors and the various reuse measures.

4.3.1 Productivity

Productivity is typically calculated as size of the system divided by cost spent to develop it, for some measure of size and cost. Keeping the size of a system constant, increasing productivity will result in a reduction in cost. There are many ways to measure both of these quantities, so as a result, there are many different measures of productivity. We used the total number of hours spent across development phases (analysis, design, implementation, testing) and rework as our measure of cost. Size was calculated as the total source lines of code (SLOC).

Using this measure of productivity, we first examined the correlations between the various reuse measures and productivity. As shown in table 4, the reuse ratio measure clearly has the best correlation with this measure of productivity. The only other measure that has a significant correlation with productivity is R_{sf}, with a correlation of 0.66.

A model can be developed to quantify the impact of reuse benefit on productivity. Since both reuse benefit (R) and productivity (Π) are non-negative real valued

Measure	R_{sf}	RL	RF	RR
Productivity	0.66	-0.16	0.12	0.82
Fault Density	-0.39	0.62	0.47	-0.67
Error Density	-0.62	0.49	0.20	-0.79
Percent Rework	0.09	0.62	0.69	-0.24

Table 4: Experimental Results: Correlations with Product Quality Factors

variables, we can model their relationship as:

$$\Pi = a(1+R)^b,$$

for some coefficients a and b. When there is no reuse, productivity is a. As reuse benefit increases, productivity increases, with the maximum reuse benefit of 1 resulting in productivity of $a * 2^b$. Taking the natural logarithm of both sides of the equation and simplifying yields the following:

$$\ln(\Pi) = ln(a) + b\ln(1+R).$$

With this form of the model, we can use a standard least squares regression to estimate the coefficients a and b.

Table 5 shows models of this form developed using the two reuse measures best correlated with productivity. The table shows the calculated coefficients for the intercept ($\ln(a)$) and the explanatory variable R (b), as well as their standard error and level of significance. The models are:

$$\ln(\Pi) = 3.96 + 1.07\ln(1+R_{sf})$$

$$\ln(\Pi) = 2.94 + 2.78\ln(1+RR)$$

Using R_{sf}, the R^2 for the model is .51, indicating that R_{sf} explains half the variation in productivity. The model developed using RR is stronger, with an R^2 of 0.77. The intercept for this model is 2.94, so when $RR = 0$, $ln(\Pi) = 2.94$, and thus productivity without reuse is $e^{2.94}$, or 18.94 SLOC/Hour. As RR increases, productivity increases. For example, an increase in reuse ratio from 0.20 to 0.30 would result in an increase in productivity from 31.4 to 39.2 SLOC per hour. As there are no projects in this sample with RR greater than 50%, any conclusion about productivity for very high levels of RR would be purely speculative.

4.3.2 Product Quality

We examined the relationship of the reuse measures to the product quality measures of fault and error density. As with productivity, we used standard definitions of fault and error density, Faults per KSLOC and Errors per KSLOC, resp. The expected effect is that as reuse increases, these measures of fault and error density will decrease. The coefficients of correlation of these defect density measures with the measures of reuse benefit are shown in table 4.

Term	R_{sf}	RR
Intercept	3.96	2.94
std. err.	0.25	0.16
p-value	0.00	0.00
ln(R)	1.07	2.78
std. err.	0.48	0.69
p-value	0.07	0.01
R^2	0.51	0.77

Table 5: Comparison of Reuse Measures in Models of Productivity

Term	Reuse Ratio
Intercept	1.96
std. err.	0.19
p-value	0.00
ln(R)	-3.24
std. err.	0.78
p-value	0.01
R^2	0.77

Table 6: A Model of Error Density Based on Reuse Ratio

For fault density, the RR again has the best correlation (r=-0.67) followed by RL (r=0.62). However, RL had a correlation in the opposite direction, i.e., as RL increases, fault density increases. This is the opposite of the result for RR and R_{sf}, which shows the expected relationship that as reuse increases, fault density decreases. One reason that RL (and RF) are correlated in this direction is that RL is defined as a measure of the density of subprogram calls. Such measures have been identified as indicators of an increased error density. Another way of looking at this is that given a function f that is needed by the developer, if he can call an existing function g, there will be an increase of a single line of code in the total project SLOC. On the other hand, if the developer prefers to create a new function g' by copying the code from g, the change in project size will be an increase of the SLOC of g. The increase with the latter option will be greater than for the former, resulting in a smaller defect density for the case where code is copied, and a larger defect density when the function is called.

The reuse ratio had the strongest correlation with Error Density, showing the expected result, namely, that as reuse increased, error density decreased. R_{sf} also had a high negative correlation with Error Density (Pearson r = -0.62). Again RL and RF had a positive correlation with Error Density showing that as the frequency increases the quality did not increase. Based on these results it appears that property 4 (which says as frequency increases the benefit should also increase) may not be applicable to measure the reuse benefit in terms of software quality.

Using an approach similar to that described for productivity, models for defect density can be developed. Again, we used a logarithmic form of the model, and used a standard least squares regression to obtain estimates of the model coefficients. The model using reuse ratio to explain error density stands out as the best model. This model is described in table 6, which shows the calculated coefficients for the intercept and explanatory variable (RR), their associated standard errors and p-values, and the model R^2. The model is

$$\ln(ED) = 1.96 - 3.24 \ln(1 + RR)$$

Both terms are significant at the 0.01 level. The intercept term of 1.96 indicates that with no reuse, error density is $e^{1.96}$, or 7.1 errors per KSLOC, and as reuse increases, error density decreases. There appears to be a decreasing impact of RR on error density as RR increases (i.e., the reduction in error density is greater for a change in reuse ratio from 0 to 0.25 than from 0.25 to 0.50), suggesting that the initial benefits of reuse in terms of error density are likely to be greater than susequent incremental increases. This is the opposite of what we see in the productivity model based on RR (i.e., as RR increases, the incremental change in productivity also increases.)

4.3.3 Rework Effort

We also looked at a measure of rework, the percentage of the effort that was spent in correcting errors, which is simply rework hours divided by total hours. This measure quantifies the inefficiency in the development process due to development errors, and is independent of how the size of the system is computed.

As indicated in table 4, R_{sf} and RR did not correlate well with this measure.

RL and RF had correlations of similar strength, however, again they indicate a negative effect, as RL and RF increase, the percentage of rework also increases. This is in part due to the correlation with defect density discussed in the previous section.

5 Conclusion

This paper is concerned with an evaluation of ***indirect*** measurement of the benefit of software reuse. Five metrics proposed in the literature have been analytically and empirically assessed with regard to their capabilities to predict productivity and quality in object-oriented systems. To analytically evaluate the metrics, we have proposed a set of desirable properties of reuse benefit measures, and evaluated these metrics in terms of their compliance with these properties.

None of the metrics satisfied all the properties, as all had strengths in some areas and weaknesses in others. RL and RF fall short in terms of the sensitivity to the cost of the reused object and the additional benefit from external reuse. RSI and RR do not cover the benefit of internal reuse. R_{sf} appears to provide a good balance, accounting for the benefit of both internal and external reuse. However, it does not account for reuse via modification, a weakness of all the measures except for RR.

To empirically evaluate the metrics, we have (1) constructed a set of tools to extract these metrics from C++ programs, (2) collected process data on the development of a set of small object-oriented systems, and then, based on the product and process data collected on these systems, (3) verified statistically the correlations between these metrics and the quality factors of productivity and defect density. Finally, for those metrics that correlated well with productivity and defect density, we also developed predictive models.

RR is well correlated with productivity, fault density and error density, but, not with the percentage of rework effort. R_{sf} has significant correlations with productivity and error density, but not with fault density of the percentage of rework effort. RL and RF appear correlated with fault and error density, and the percentage of rework effort, but interestingly, in the opposite direction. As RL and RF increase, we see an increase in fault density, rework density, and the percentage of rework effort.

A major difference between R_{sf} and RL/RF is that R_{sf} accounts for component size. This important difference may be the reason for the markedly different results found with these measures, with R_{sf} showing some correlation with the quality factors, and RL/RF showing either no correlation, or a significant correlation, but in the opposite direction.

Another interesting point raised with this work is the fact that the modified components also appear to have a significant effect in terms of increasing productivity and quality, and, thus, should also be considered in a comprehensive definition of a reuse metric. Nevertheless, this raises some questions. For instance, how can we accurately verify the extent to which a component has been changed? What should the modification threshold be? In this work we assumed that only components changed less than 25 percent should be counted as reused. This threshold may be domain dependent, i.e., different organizations should conduct empirical work in order to determine which threshold is most significant in their environment. In addition, tools must be built in order to determine automatically how much a component has been changed. This can, in fact, reduce the human error introduced in the analysis, thus increasing the accuracy and reliability of the results.

Finally, our empirical study has highlighted a practical difficulty in using RSI. In four out of the seven student projects used in our data, there was ***no verbatim reuse*** of components from external libraries. For this reason, RSI was zero in four out of seven data points. This precludes any useful analysis of the predictive power of the RSI data; however, our experience indicates that RSI may not provide helpful data in projects where a significant number of external components are used only after modifications. From our experience, it appears that other metrics offer some ability to explain the variation in productivity and quality even in such cases; this suggests that internal reuse may be an important factor, and that RSI may be taking too strict a view of what constitutes reuse.

The results indicate that different reuse metrics can be used as predictors of different quality attributes. For example, reuse ratio and size/frequency reuse metric each appeared to be well correlated with productivity and error density, but this size/frequency metric did not show any significant result with regard to fault density. Further empirical validation is, thus, still necessary in order to evaluate these metrics in actual software organizations. Empirical work is (in general) hobbled by the difficulty in obtaining sufficient process data to allow for empirical validation of the metrics. This work provides a framework by which reuse metrics can be analytically and empirically evaluated prior to their use: the analytical properties, software tools and data collection programs developed in the framework of this study can be used in other studies, thus facilitating the replication of this work in academia and industry. As a continuation of this work, we intend to:

- Perform case studies at the Software Engineering Laboratory (SEL) to assess the feasibility of automated methods for determining the amount of modification in a component and to further identify what is an appropriate threshold of modification to still achieve a reuse benefit,
- Evaluate the set of metrics analyzed in this paper using the product and process data extracted from object-oriented systems under development at the SEL,
- Evaluate the capabilities of prediction of these metrics with regard to fault density, rework and maintainability,
- Continue the empirical analyses to better understand the importance of the proposed properties of reuse measures identified in this paper.

Acknowledgements

We would like to thank the anonymous referrees for their insightful comments; we are also grateful to Dr. Vic Basili of the University of Maryland (who generously provided research resources which helped in the realization of the empirical study) and to Dr. Rick Greer of AT&T Bell Laboratories (for help with the Daytona database system).

References

[1] B. Baker. A theory of parametrized pattern matching: algorithms and applications. *Journal of Comput. Sys. Sci.*, to appear, 1995.

[2] V. Basili, L. Briand, and W. Melo. Measuring the impact of reuse on quality and productivity in object-oriented systems. Technical Report CS-TR-3395, University of Maryland, Computer Science Department, 1995.

[3] J. M. Bieman. Deriving measures of softwre reuse in object oriented systems. In T. Denvir, R. Herman, and R. W. Whittey, editors, *Formal Aspects of Measurement*, pages 79–82. Springer–Verlag, 1992.

[4] T. Bollinger and S Pfleeger. Economics of reuse: issues and alternatives. *Information and Software Technology*, 32(10):643–652, 1990.

[5] Y-F. Chen, B. Krishnamurthy, and K-P. Vo. An Objective Reuse Metric: Model and Methodology. In *Fifth European Software Engineering Conference*, 1995.

[6] P. Devanbu. GENOA a customizable, language and front–end independent code analyzer. In *Proc. of 14th Int'l Conf. on Software Engineering (ICSE)*, pages 307–317. IEEE Press, 1992.

[7] P. Devanbu and S. Karstu. Measuring the benefits of software reuse. Technical report, AT&T Bell Laboratories, 1994.

[8] N. E. Fenton. *Software Metrics: A Rigorous Approach.* Chapman & Hall, 1991.

[9] W. Frakes and C. Terry. Reuse level metrics. Technical Report TR 94-03, Virginia Polytechnic Institute and State University, 1991.

[10] J. E. Gaffney and R. Cruickshank. A general economic model for software reuse. In *Proc. of the 14th Int'l Conf. on Software Engineering.* IEEE Press, 1992.

[11] J. E. Gaffney and T. A. Durek. Software reuse — key to enhanced productivity: some quantitative models. *Information and Software Technology*, 31(5):258–267, 1989.

[12] R. Greer. All about daytona. Technical report, AT&T Bell Laboratories, 1994.

[13] F. McGarry, R. Pajerski, G. Page, S. Waligora, V. Basili, and M. Zelkowitz. Software process improvement in the NASA software engineering laboratory. Technical Report CMU/SEI-95-TR-22, Carniege-Mellon Unv., S/W Eng. Institute, Dec. 1994.

[14] J. Poulin, J. Caruso, and D. Hancock. The business case for software reuse. *IBM Systems Journal*, 32(4):567–594, 1993.

[15] W. Thomas, A. Delis, and V. Basili. An analysis of errors in a reuse-oriented development environment. Technical Report CS-TR-3424, Dept. of Computer Science, University of Maryland, College Park, MD, 20742, Feb. 1995.

[16] E. J. Weyuker. Evaluating software complexity metrics. *IEEE Transacations on Software Engineering*, 14(9):1357–1365, 1988.

54-61
415788
360718 4P

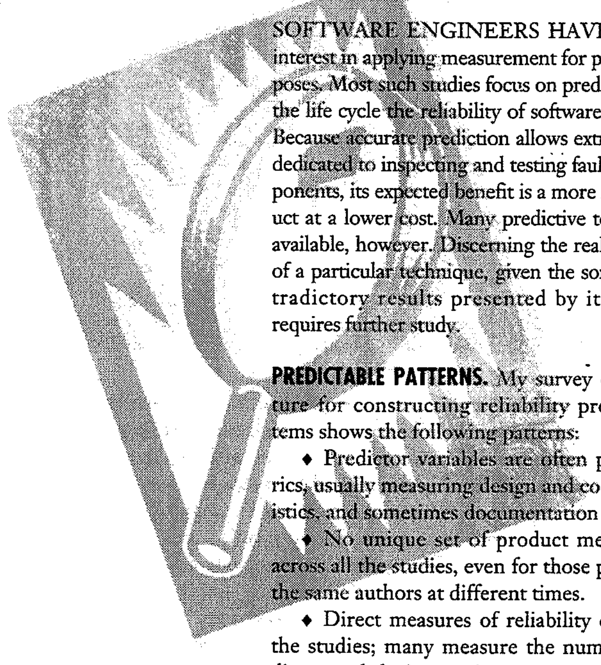
FILIPPO LANUBILE, UNIVERSITY OF MARYLAND

Why Software Reliability Predictions Fail

New views of mature ideas on software and quality productivity.

Software reliability reflects a customer's view of the products we build and test, as it is usually measured in terms of failures experienced during regular system use. But our testing strategy is often based on early product measures, since we cannot measure failures until the software is placed in the field. This issue, Filippo Lanubile shows us that such measurement is not effective at predicting the likely reliability of the delivered software.

—Shari Lawrence Pfleeger



SOFTWARE ENGINEERS HAVE A GREAT interest in applying measurement for predictive purposes. Most such studies focus on predicting early in the life cycle the reliability of software components. Because accurate prediction allows extra effort to be dedicated to inspecting and testing fault-prone components, its expected benefit is a more reliable product at a lower cost. Many predictive techniques are available, however. Discerning the real effectiveness of a particular technique, given the sometimes contradictory results presented by its advocates, requires further study.

PREDICTABLE PATTERNS. My survey of the literature for constructing reliability prediction systems shows the following patterns:

- ♦ Predictor variables are often product metrics, usually measuring design and code characteristics, and sometimes documentation attributes.
- ♦ No unique set of product metrics is used across all the studies, even for those performed by the same authors at different times.
- ♦ Direct measures of reliability differ among the studies; many measure the number of faults discovered during testing, some count the number of failures during operation, and others track the number of repairs made during maintenance.
- ♦ Prediction problems are often reduced to classification problems by choosing a categorical variable as an outcome that may be predicted from the predictor variables, for example, grouping all components with at least one fault under the high-risk category and all components with no faults under the low-risk category.
- ♦ Many study authors advocate a modeling technique derived from statistical analysis, machine learning, or neural networks—or a combination of the three—aiming to show that their own approach is "good," without comparing it to any others.
- ♦ Other study authors show that their modeling technique is "better than others," by comparing it with just one or two different techniques selected to showcase their own technique's advantages.
- ♦ Studies use different criteria when comparing various prediction systems, use different definitions even when using the same criteria names, risk ambiguity when they use informal criteria, and fail to capture all aspects of the prediction systems studied.
- ♦ Few studies try to define criteria so that the capability of the model to actually predict real behavior can be determined.
- ♦ All the studies claim to have been successful at showing the superiority of the advocated modeling technique; when studies include a comparison, the advocated technique always scores highest.

Few studies define criteria so that the capability to actually predict behavior can be determined.

EMPIRICAL STUDY. Amazed by previous studies' high success rate when using predictive techniques, but conscious of existing methodological limits, I started a research project with Giuseppe Visaggio to externally replicate past studies. Scientists perform replications to increase their ability to generalize their results in different settings and times. External replications—those independently conducted by different researchers—are needed because empirical observations in support of a hypothesis may be in error or be biased by the original researcher. Our replication effort exhibited the following characteristics.

We used our own data, collected during three

Editor:
Shari Lawrence Pfleeger
Systems/Software
4519 Davenport St. NW
Washington, DC
20016-4415
s.pfleeger@ieee.org

TABLE 1
RESULTS FROM COMPARING REAL AND PREDICTED RISK

Modeling technique	Predictive Validity (Probability)	Proportion of Type 1 Errors	Proportion of Type 2 Errors	Proportion of Type 1 + 2 Errors	Completeness	Overall Inspection	Wasted Inspection
Discriminant analysis	*p*=0.621	28%	26%	54%	42%	46%	56%
Principal-component analysis + discriminant analysis	*p*=0.408	15%	41%	56%	68%	74%	55%
Logistic regression	*p*=0.491	28%	28%	56%	42%	49%	58%
Principal component analysis + logistic regression	*p*=0.184	13%	46%	59%	74%	82%	56%
Logical classification Model	*p*=0.643	26%	21%	46%	47%	44%	47%
Layered neural network	*p*=0.421	28%	28%	56%	42%	49%	58%
Holographic network	*p*=0.634	26%	28%	54%	47%	51%	55%
Heads or tails?	*p*=1.000	25%	25%	50%	50%	50%	50%

years of a project-intensive software engineering course held at the University of Bari. Small teams of students developed 27 Pascal programs in the IS domain from the same specification. Other independent student teams drawn from an advanced software engineering course tested different groups of components, randomly selected from each program.

We chose the most-used dependent and independent variables. The dependent variable was the risk class of software components. We defined as high-risk any software component whose faults were detected during testing, and as low-risk any component with no discovered faults. The two classes contained an approximately equal number of components. The independent variables were 11 product metrics covering design attributes such as fan-in, fan-out, and information flow; implementation attributes such as cyclomatic complexity, number of unique operands, total number of operands, total lines of code, number of noncomment lines of code, and Halstead's program length and volume; and the documentation attribute of comment density.

We included all the classification techniques already used for predicting software reliability: principal-component analysis, discriminant analysis, logistic regression, logical classification models, layered neural networks, and holographic networks. Principal-component analysis served as a preprocessing step to obtain a smaller number of orthogonal domain metrics. We built two models each for discriminant analysis and logistic regression. The first model was based on the 11 original complexity measures; the second used three domain metrics that had been generated from the principal-component analysis.

We improved the evaluation criteria. All criteria were formalized using a two-way contingency table as the underlying model with one row for each level of the variable real risk and one column for each level of the variable predicted risk. We assessed the absolute worth of a predictive model by testing a null hypothesis of no association between the real risk and the predicted risk with an alternative hypothesis of general association.

Among the criteria to compare the performance of the prediction systems, we measured the proportions of the two misclassifications: Type 1 errors, in which a high-risk component is classified as low-risk, and Type 2 errors, in which a low-risk component is classified as high-risk. We measured the completeness of the prediction, defined as the percentage of high-risk components that actually have been classified as such by the model. We also considered the cost of identifying a component as needing more verification effort. We measured both the overall inspection cost, defined as the percentage of components that have been flagged as high-risk, and the wasted inspection, defined as the percentage of verified components that have been incorrectly classified.

RESULTS. We built a training set, including two-thirds of the 118 tested components, to create and tune the predictive models. We used the remaining third of the components, which comprised the testing set, to assess the models and compare their performances, as shown in Table 1. Despite the variegated selection of techniques available, no model satisfied the criterion of predictive validity by being able to discriminate between components with faults and components without faults. All the significance probabilities are too high with respect to the most common values—0.01, 0.05, 0.1—used to reject a null hypothesis. No model shows a significant departure from the performance of a random prediction, except for discriminant analysis and logistic regression applied in conjunction with the principal-component analysis. These two models have good percentages of Type 1 error and completeness but also bad percentages of Type 2 error and overall inspection. The proportion of Type 1 + Type 2 errors and the wasted inspection do not vary with respect to the other models. Instead of producing more stable models, principal-component analysis built models that were biased toward classifying components as high-risk.

LESSONS LEARNED. Our experience indicates that the future behavior of software products cannot always be predicted successfully. Although this sounds obvious, much of the scientific literature reports successful results only in the identification of fault-prone components from product measures. Publishing only empirical studies with positive findings can give practitioners unrealistic expectations that are quickly followed by equally unrealistic disillusionment.

Although the study did not specifically set out to do so, it shows that predictive-modeling techniques are only as good as the data on which they are based. All the prediction systems failed because they assumed a relationship between software product measures and software faults. This is not always true. On the contrary, a predictive model, from the simplest to the most complex, is worthwhile only if used with a local process to select metrics that are valid as predictors. Recently, methods to validate measures of internal software attributes have been proposed, based on iterative verification of locally collected data. Unfortunately, these methods cannot guarantee *a priori* that a significant relationship will be found between some internal product metric and the software attribute of interest. ◆

Filippo Lanubile is a senior researcher of the Experimental Software Engineering Group at the University of Maryland, College Park (http://www.cs.umd.edu/projects/SoftEng/ESEG/). He is currently on sabbatical from the University of Bari, Italy, where he is an assistant professor of computer science. He can be reached at lanubile@cs.umd.edu.

S5-61

416789

360822

12p

Defining Factors, Goals and Criteria for Reusable Component Evaluation

Presented at the CASCON '96 conference, Toronto, Canada, November 12-14, 1996.

Jyrki Kontio, Gianluigi Caldiera and Victor R. Basili
University of Maryland
Department of Computer Science
A.V.Williams Building
College Park, MD 20742, U.S.A.
Emails: [jkontio | caldiera | basili]@cs.umd.edu

Abstract:

This paper presents an approach for defining evaluation criteria for reusable software components. We introduce a taxonomy of factors that influence selection, describe each of them, and present a hierarchical decomposition method for deriving reuse goals from factors and formulating the goals into an evaluation criteria hierarchy. We present some highlights from two case studies in which the approach was applied. The approach presented in this paper is a part of the OTSO[1] method that has been developed for reusable component selection process.

1. Introduction

Software reuse is considered an important solution to many of the problems in software development. It is credited with improving the productivity and the quality of software development [1,4,15,24,30,33], and many organizations have claimed significant benefits from it [13,23].

Some organizations have implemented systematic reuse programs [13], which have resulted in in-house libraries of reusable components. Other organizations have supported their reuse with component-based technologies and tools. The increased commercial availability of embeddable software components, standardization of basic software environments (such as Microsoft Windows, Unix), and the explosive popularity of the Internet have resulted in a new situation for reusable software consumers: there are many more accessible reuse candidates. Consequently, many organizations are spending much time in reusable component selection since the choice of the appropriate components has a major impact on the project and resulting product.

Despite their importance, the issues and problems associated with the selection of suitable reusable components have rarely been addressed in the reuse community. Poulin et al. present an overall selection process [23] and include some general criteria for assessing the suitability of reuse candidates [32]. Some general criteria have been proposed to help in the search for potential reusable components [24,25]. Boloix and Robillard recently presented a general framework for assessing the software product, process and their impact on the organization [6]. However, none of this work is specific to off-the-shelf (OTS[2]) software selection, and the issue of how to define the evaluation criteria is not addressed. Furthermore, most of the reusable component literature does not seem to emphasize the sensitivity of such criteria to each situation.

We have developed a method that addresses the selection process of packaged, reusable software, or OTS as we refer to it in this paper. The method, called OTSO, supports the search, evaluation and selection of reusable software, and provides specific techniques for defining the evaluation

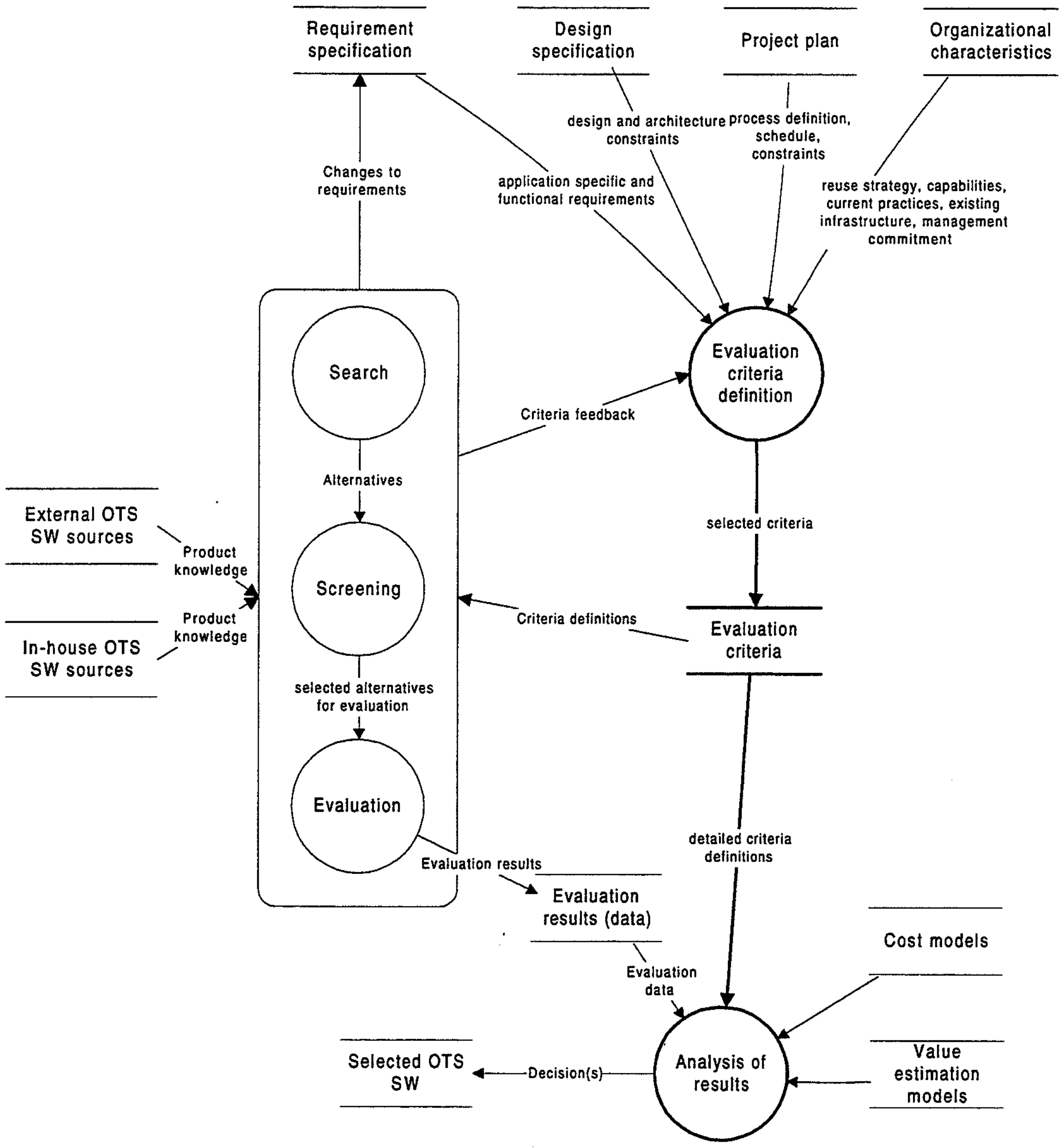
[1] OTSO stands for Off-The-Shelf Option. The OTSO method represents a systematic approach to evaluate such an option.

[2] "OTS" stands for "off-the-shelf". The term originates from the term "COTS software", i.e., commercial off-the-shelf software. In this paper OTS refers to both commercial and in-house source code, executables, design, documentation, test cases, etc.

criteria, comparing the costs and benefits of alternatives, and consolidating the evaluation results for decision making [18,19,21]. The main characteristics of the OTSO method are as follows:

- A defined, systematic process that covers the whole reusable component selection process.
- A method for estimating the relative effort or cost benefits of different alternatives.
- A method for comparing the "non-financial" aspects of alternatives, including situations involving multiple criteria.
- a predefined template for product quality characteristics to be tailored and used in each instance of the selection process.

Figure 1 shows the main activities in the OTSO reusable component selection process using a dataflow diagram notation. Each activity in presented as a process symbol – a circle – and artifacts produced or used are presented as data storage symbols in Figure 1. In the *search* phase, the goal is to identify potential candidates for further study. The *screening* phase selects the most promising candidates for detailed *evaluation*. In the *analysis* phase, the results of product evaluations are consolidated, and a decision about reuse is made. As the selected alternative is used (*deployed*), the effectiveness of the reuse decision, eventually, can be *assessed.*



Requirement specification
Design specification
Project plan
Organizational characteristics
design and architecture constraints
process definition, schedule, constraints
Changes to requirements
application specific and functional requirements
reuse strategy, capabilities, current practices, existing infrastructure, management commitment
Search
Evaluation criteria definition
Criteria feedback
Alternatives
selected criteria
External OTS SW sources
Product knowledge
Screening
Product knowledge
Criteria definitions
Evaluation criteria
In-house OTS SW sources
selected alternatives for evaluation
Evaluation
detailed criteria definitions
Evaluation results
Evaluation results (data)
Cost models
Evaluation data
Selected OTS SW
Decision(s)
Analysis of results
Value estimation models

Figure 1: The main phases in the OTSO process

Reuse candidates are evaluated in different ways in all phases. The OTSO method is based on incremental, evolutionary definition and use of the evaluation criteria so that the criteria set can be gradually refined to support each phase. While Figure 1 presents the overall OTSO process, this paper presents the goal-driven criteria definition process of the method that has not been described publicly so far. Details about other aspects of the method are available in separate reports and publications [18,19,21].

The structure of this paper is as follows. Section 2 presents the factors that influence the OTS software selection and how the reuse goals can be formulated. Section 3 presents how the evaluation criteria can be defined and decomposed. Section 4 presents applicable results from the two case studies with the OTSO method. Finally, the conclusion section discusses the relevance of our results.

2. Factors in Reusable Software Selection

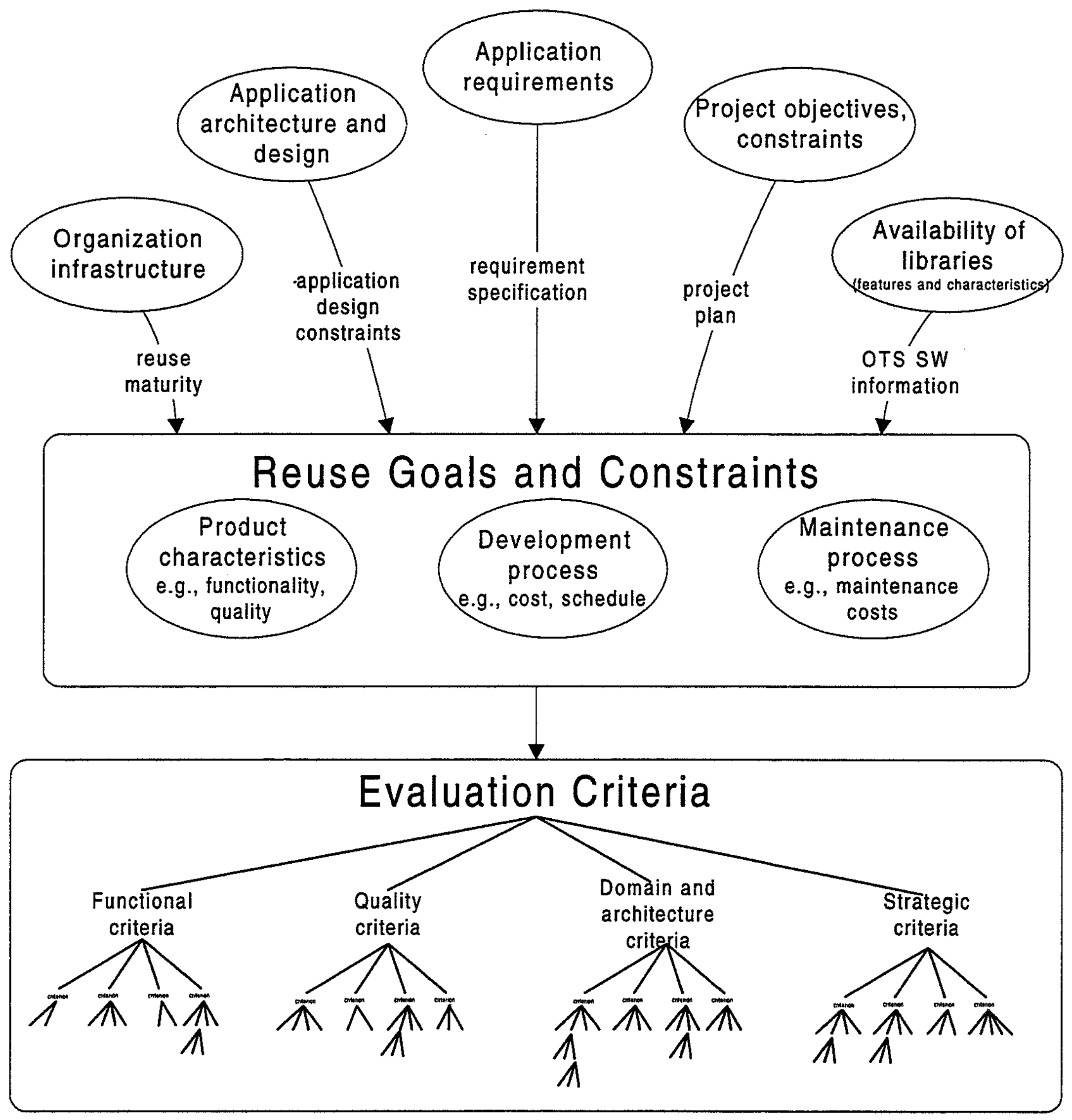
The overall relationships among influencing factors, reuse goals and evaluation criteria are presented in Figure 2. The first main task in reusable software evaluation is to define the reuse goals. This must be based on a careful analysis of the influencing factors. We identified five groups of factors that primarily influence the OTS software selection. In the following sections we discuss each of these groups.

Application requirements are likely to be the most important factor in evaluating reusable software. Such requirements can include functional requirements (such as the ability to manage and display graphical geographical data) and non-functional requirements (such as available memory or speed of operations). The requirement specification, if available, should be used as a basis for interpreting such requirements.

The requirement specification, however, can only give partial support for the interpretation of software reuse goals for two reasons. First, the requirement specification typically does not define how the system should be implemented or what components could be implemented through OTS software. Considering the use of OTS software in a system means making some assumptions about the architecture of the system, and requires some decisions on which system features should be covered by the OTS software. Second, the requirement specification may not be detailed enough for evaluating OTS software alternatives. In both of these cases the formulation of software reuse goals requires interpretation and further refinement of requirements, and some design concepts.

Application and domain architecture introduce additional elements that need to be evaluated. The architecture, in this context, provides a set of constraints deriving from how particular applications are built: this includes, for instance, components and design patterns used or assumed, communication and interface standards, platform characteristics. All of this introduces a set of constraints that may make integration of some alternatives impractical or costly. Some kind of application mediator, or "middleware", may be used to overcome such problems. This mediator, however, needs to be either developed or acquired from another source and this provides, for instance, a cost increment that needs to be estimated.

The application domain may also have some specific characteristics that are not addressed by OTS software developed for other domains (for example, real-time applications vs. batch processing). Sometimes the architecture is a given and acts as a constraint in the OTS software selection; sometimes the selection of the right OTS software may determine or influence the system architecture.



Application requirements
Application architecture and design
Project objectives, constraints
Organization infrastructure
Availability of libraries (features and characteristics)
application design constraints
requirement specification
project plan
reuse maturity
OTS SW information
Reuse Goals and Constraints
Product characteristics e.g., functionality, quality
Development process e.g., cost, schedule
Maintenance process e.g., maintenance costs
Evaluation Criteria
Functional criteria
Quality criteria
Domain and architecture criteria
Strategic criteria

Figure 2: Factors influencing the selection of reusable off-the-shelf software

Project objectives and constraints may influence the library selection through the schedule or the budget of the project. For instance, early deadlines or low personnel budget[3] may require the use of externally produced software. Project objectives may also imply the use of external software if, for example, these external libraries may provide a better way to comply with standards or may be proven reliable in implementing some aspects of the library. There may also be some organizational constraints that are set for the project, such as availability of personnel with specific skills.

The availability of features in software reuse candidates also affects the evaluation criteria definition. This works in two ways. On one hand, it is important to check that the evaluation criteria are based on realistic expectations. That is, the criteria set should not assume characteristics that are not provided by any OTS software alternative. On the other hand, it may be useful to know about the possibilities that OTS software alternatives

[3] Note that this example is not meant to imply that the use of OTS software necessarily results in lower overall costs. The example highlights the usual situation where the use of OTS software changes the cost structure of a project, e.g., development costs may be lowered but software acquisition costs are higher.

offer but that may not have been included in the requirement specification.

Finally, an organization's reuse infrastructure and reuse maturity should also be considered when defining reuse goals. Reuse maturity comprises several issues: an organization's experience in reuse, its commitment and interest in continuing systematic reuse, the knowledge and skills of personnel responsible for reuse, the availability of specific tools for supporting reuse (configuration management tools, information databases, etc.), and the existing software development environment [9,17]. Reuse maturity is particularly important for the in-house production of reusable components. Also, if an organization has no experience in OTS software reuse, it may have a limited ability to integrate OTS software and to estimate the effort required for OTS software integration.

The main point of this discussion is that the *evaluation criteria should be developed with full awareness of all these factors*. In most cases, this requires that each factor be explicitly analyzed, and documented and used as input in the final definition of the evaluation criteria.

Each OTS software reuse situation is different, and so are the reuse goals associated with it. Based on the analysis of factors as described in the previous section, the reuse goals for the project need to be stated explicitly. The OTS software reuse goal statement essentially should describe the following:

- Where and how OTS software is to be used in the application;
- The expected benefits of OTS software reuse, such as functionality, quality, schedule impact, or effort savings;
- Possible constraints for OTS software reuse;
- Cost budget for the use of OTS software.

The reuse goal statement should be documented explicitly, although initially the goal statement may seem abstract and simple. Our experience indicates that it will be revised and become more detailed as the OTS evaluation progresses.

Reuse objectives can be divided into development process goals, maintenance process goals and product characteristics goals. Development process goals relate to the cost, effort and schedule of the development project. The maintenance process goals deal with issues such as the ease or cost of maintenance and who will be responsible for maintenance. Product characteristics goals refer to product functionality and product quality.

3. Evaluation Criteria

3.1 Classes of evaluation criteria

The factors and goals described in Figure 2 determine the reuse goals for the system. The content and priorities of these goals determine which characteristics must be considered in the of the OTS software selection process. The evaluation criteria themselves can be categorized into four main areas: (i) functional requirements, (ii) product quality characteristics, (iii) strategic concerns, and (iv) domain and architecture compatibility.

Functional requirements: These refer to identifiable, functional features or characteristics that are specific to the particular situation. These criteria are derived from the requirement or design specification and are expressed in the form of requirements. Here are two examples from an application dealing with geographical data:

- Display ocean bathymetry data
- Show political boundaries.

Product quality characteristics are common to a broader set of reuse situations. Typically the structure and relationships of these characteristics remain the same but their acceptable values may vary from case to case. Three examples are:

- Defect rate
- Compliance to the project user interface guidelines
- Clarity of documentation.

Strategic concerns: These are the short-term and the long-term effects of the reuse candidate, the cost-benefit issues and the organizational issues beyond the scope of the project in question. These help to consolidate information for decision making. Three examples are:

- Acquisition costs
- Effort saved
- Vendor's future plans.

Domain and architecture compatibility: An application domain or the software architecture may also require specific characteristics from reuse candidates. For instance, all flight-control software must be very reliable and must be developed with time-sensitive and reactive issues in mind. A reuse candidate originally developed

Attribute of GQM	Explanation	Examples	
Object (entity)	The entity being analyzed, e.g., OTS product, OTS vendor	OTS product	OTS vendor
Focus (issue)	The attributes that are of interest, e.g, cost, reliability, or efficiency.	Cost	Viability
Purpose	Evaluate: evaluate the characteristics of the entity w.r.t. a relevant benchmark. This attribute is typically the same in all reuse evaluation cases.	Evaluate	Evaluate
Point of view (perspective)	Whose interest is being expressed, e.g., project manager, corporation, customer, developer, etc.	Customer	Development organization

Table 1: GQM-based evaluation criteria definition template

for accounting software may have fundamental design and performance characteristics that make it unsuitable for such an application area.

Domain compatibility refers to how well the reuse candidate and its features map into the domain terminology and concepts. In the case of object oriented reuse candidates, this can refer to a match between domain objects and object definitions in the reuse candidate. Architecture compatibility refers to software or hardware architecture requirements that are common to the application area.

Examples are listed below:

Domain compatibility:
- system states can be modeled and represented
- geographical data manipulation capability

Architecture compatibility:
- supports or is compatible with CORBA
- compatible with Microsoft Windows OLE.

The evaluation criteria must be customized for each selection situation. The functional requirements, which often are central to the selection process, are often unique to each application. For the product quality characteristics and strategic concerns, it is possible to define some templates that have stable elements accross applications. As an input to product quality characteristics, there are several possible sources [5,7,10,16]. Figure 3 shows an example of the product quality factors we defined for one of our case study projects.

3.2 Hierarchical decomposition of evaluation criteria

The evaluation criteria are derived from the factors and goals discussed in the previous sections. The first step in this process is to define the *evaluation goals* using the GQM approach, as it provides a well-defined template for documenting such evaluation goals [2,3]. Table 1 presents the template used for GQM goals. The *object* attribute and *focus* attributes can be derived often directly from reuse goals. For example, if a reuse goal is to reduce development cycle time, we are evaluating the process (object) and its duration (focus).

The *purpose* attribute can range from simple characterization to understanding, evaluation and even prediction [2]. However, most often the purpose is evaluation. The point of view attribute is relevant when there are different stakeholders interested in the results and their views need to be considered. For example, developer and user perspectives may be different in terms of required functionality of the product.)

The basic steps of criteria decomposition are the following [18]:

1. Identify and formulate evaluation goals using the template given in Table 1. For example, an evaluation goal could be stated as follows: object/entity:
2. For each evaluation goal, define a set of high-level criteria or questions that characterize it.
3. For each criterion, write down an unambiguous definition of it.
4. If the value for the criterion can be determined with an objective measurement, observation or judgment, call it an evaluation attribute, and continue to decompose and define other criteria. If the criterion is too abstract to be measured with a single metric, if it has too many aspects to be assessed through observation or if it cannot be judged objectively, continue decomposing it.

Type of Evaluation Attribute	Examples
Measurement	• memory used when loaded • time to initialize • number of bugs found during evaluation • support of incremental image loading
Description	• list of reported bugs • description of the "undo" function
Subjective assessment	• estimate of vendor's growth potential • look and feel of GUI

Table 2: Examples of evaluation attributes

The number of items at each level should be less than 10, preferably around 3 to 5.

The objective of the criteria-definition process is to decompose criteria into a set of concrete, measurable, observable or testable *evaluation attributes*. An evaluation attribute can be an observation, a measurement, or a piece of information to be obtained. Table 2 lists examples of possible types of evaluation attributes.

Once the criteria have been defined, the OTSO method relies on the use of the Analytic Hierarchy Process (AHP) for consolidating the evaluation data for decision-making purposes. The AHP technique was developed by Thomas Saaty for multiple-criteria decision-making situations [26,27]. The technique has been widely and successfully used in several fields [28], including software engineering [11] and software selection [14,22]. It has been reported effective in several case studies and experiments [8,12,28,31]. Due to the hierarchical treatment of our criteria, AHP fits well into our evaluation process as well. AHP is supported by a commercial tool that supports the entering of judgments and performs all the necessary calculations [29].

The AHP is based on the idea of decomposing a multiple-criteria decision-making problem into a criteria hierarchy. At each level in the hierarchy the relative importance of factors is assessed by comparing them in pairs. Finally, the alternatives are compared in pairs with respect to the criteria. Rephrasing the AHP approach in the OTSO framework, the evaluation proceeds as follows [11,27]:

1. Define the importance of factors on each level.
2. Define the preferences of alternatives over the lowest level factors in the criteria tree.
3. Check the consistency of rankings and revise them if necessary.
4. Present the results of the evaluation, the alternative with the highest priority being the one that is recommended as the best alternative.

The rankings obtained through paired comparisons between the alternatives are converted to normalized rankings using the "eigenvalue" method, preferably using a software tool that automates the calculation process [27].

This process can be illustrated with a simple example. Assume that one needs to decide which Web browser to use, Internet Explorer or Netscape (alternatives). Assume that the evaluation criteria decomposition process has resulted in just criteria, price and popularity. According to the AHP method we would first determine the relative importance of factors, resulting in weights for each. New both alternatives (Internet Explorer and Netscape) are compared against these two criteria and their relative rankings (weights) are obtained. Based on this information, the relative preferences of alternatives can be calculated and expressed as numbers totaling one. More information about the details of the AHP method or the Expert Choice tool is available separate publications [26,27,29].

From our perspective, the main advantage of AHP is that it provides a systematic, validated approach for consolidating information about alternatives using multiple criteria. AHP can be used to "add up" the characteristics of each alternative. Furthermore, an additional benefit of AHP is that we can choose the level of consolidation. We recommend that consolidation be carried out only to a level that is possible without sacrificing important information. On the other hand, some consolidation may avoid overwhelming the decision makers with too much detailed, unstructured information.

The weighting of alternatives is done using the AHP method, preferably using a supporting tool [29]. Preferences are collected and consolidated to the level stakeholders prefer. The AHP allows the consolidation of all qualitative information and financial information into a single ranking of alternatives. However, we believe that this would condense valuable information too much. Instead, we recommend that information about the evaluation be consolidated to a level where a few main items remain so that stakeholders can discuss their impact and preferences. The full consolidation can be done at the end as a sanity check, if desired.

4. Case Studies

We carried out two case studies using the OTSO method. The results of these case studies are reported separately [18,19,21]. The first case study assessed the overall feasibility of the method and the second one focused on the comparison of analysis methods. Both case studies took place in the NASA's Earth Orbiting System (EOS) program with Hughes Information Technology Corporation and were dealing with real software development projects facing a COTS selection problem.

Our first case study dealt with the selection of a library that would be used to develop an interactive, graphical user interface for entering location information on Earth's surface areas. This case study used the OTSO method's hierarchical and detailed criteria definition approach. Part of the criteria hierarchy is presented in Figure 3. The main conclusion was that the OTSO method was a feasible approach in COTS selection and its overhead costs were marginal [18].

The first case study also showed that OTS package features can change the application requirements: one of the OTS alternatives was able to display ocean bathymetry data graphically. Although this was not initially specified as a requirement, the application designers considered it a valuable feature and proposed it to be included in the requirements specification. This important feedback loop is characterized by the arrow from the search/screening/evaluation contour in Figure 1.

The second case study dealt with the selection of a hypertext browser for the EOS information service. This case study included a comparison between two analysis methods, the AHP method and a weighted scoring method.

A total of over 48 tools were found during the search for possible tools. Based on the screening criteria, four of them were selected for hands on evaluation. The evaluation criteria were derived from existing, broad requirements. However, as in the first case study, the requirements had to be elaborated and detailed substantially during this process.

This case study further supported our conclusion of the low overhead of the OTSO method. Furthermore, this case study involved several evaluators, and our criteria definition approach improved the efficiency and consistency of the evaluation. We also found an unexpected result when comparing the two analysis methods: they yielded different rankings of the COTS alternatives even though they were based on the same data [19,20]. In our opinion, this highlights the importance of appropriate analysis and data consolidation techniques in such evaluations.

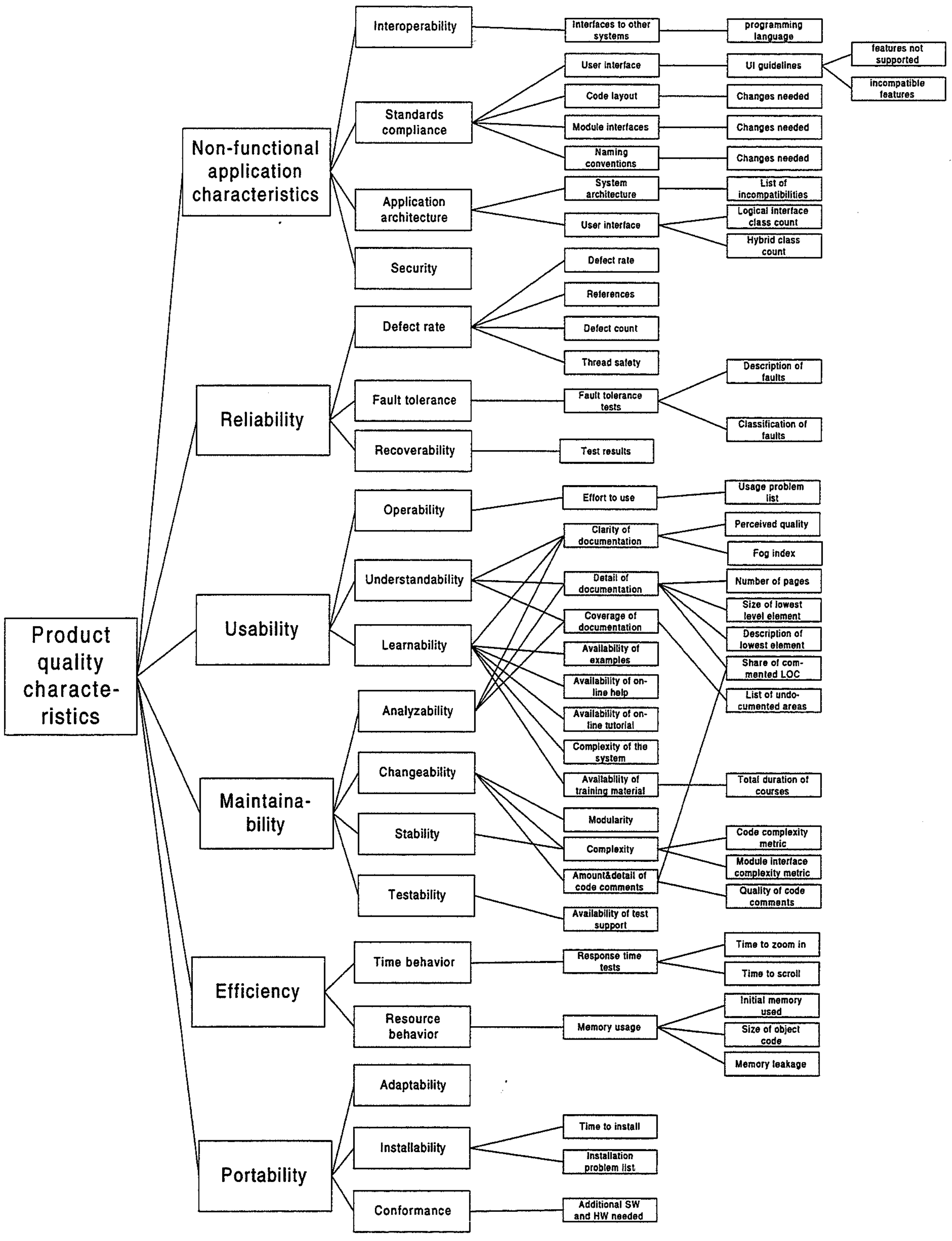


Figure 3: Example of a product evaluation criteria hierarchy

5. Conclusions

The OTSO method was developed to consolidate some of the best practices we have been able to identify for OTS software selection. The experiences from our case studies indicate that our method is feasible in an operational context: it improves the efficiency and consistency of evaluations, it has low overhead costs, and it makes the COTS selection decision rationale explicit in the organization. The detailed evaluation criteria also contribute to the refinement of application requirements.

The evaluation criteria definition approach presented in this paper is a central element of the OTSO method. The underlying assumption of our approach is that as each situation is different, the factors, goals and evaluation criteria will need to be defined for each situation separately. By formalizing this criteria definition process, it is possible to reuse the OTS software selection experiences better, leading to a more efficient and reliable selection process.

Although our case studies were both performed in the same application domain, we have not encountered any domain specific characteristics that would limit the applicability of the method in other domains. Also, while the case studies themselves were relatively small, the evaluation processes, and the resulting criteria, were quite extensive. This leads us to suggest that the method may be able to scale up to larger situations as well. However, further validation is necessary to determine this with more confidence.

6. Acknowledgments

This work has been supported by the Hughes Information Technology Corporation and the EOS Program, as well as by the Software Engineering Laboratory, a joint software process improvement organization between NASA, Computer Sciences Corporation and University of Maryland.

7. About the Authors

Jyrki Kontio is a researcher at the Department of Computer Science at University of Maryland. His research interests include risk management, process improvement, process modeling, technology management, and software reuse. He is on a leave-of-absence from Nokia Research Center, completing his Ph.D. on software risk management. He can be reached at jkontio@cs.umd.edu.

Gianluigi Caldiera is a Research Manager at the Department of Computer Science at University of Maryland. His research and professional activities are in software engineering, with special focus on software quality assurance and management, software metrics, software reuse and software factories, software process improvement, and quality standards. In his more than 15 years of professional and academic experience, he has consulted for Government and industry in both Europe and United States. He can be reached at gcaldiera@cs.umd.edu.

Victor R. Basili is a professor in the Institute for Advanced Computer Studies and the Department of Computer Science. He is a co-founder and a director of the Software Engineering Laboratory. Professor Basili is also co-editor-in-chief of the International Journal of Empirical Software Engineering. He can be reached at basili@cs.umd.edu.

8. References

[1] B. Barnes, T. Durek, J. Gaffney, and A. Pyster. "A Framework and Economic Foundation for Software Reuse,". In: *Tutorial: Software Reuse: Emerging Technology*, ed. W. Tracz. Washington: IEEE Computer Society, 1988.pp. 77-88.

[2] V. R. Basili, "Software Modeling and Measurement: The Goal/Question/Metric Paradigm," CS-TR-2956, 1992. Computer Science Technical Report Series. University of Maryland. College Park, MD.

[3] V. R. Basili, G. Caldiera, and H. D. Rombach. "Goal Question Metric Paradigm,". In: *Encyclopedia of Software Engineering*, ed. J. J. Marciniak. New York: John Wiley & Sons, 1994.pp. 528-532.

[4] T. Birgerstaff and C. Richter, "Reusability Framework, Assessment, and Directions," *IEEE Software*, vol. 4, March. pp. 41-49, 1987.

[5] B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative Evaluation of Software Quality," pp. 592-605, 1976. Pcodeedings of the Second International Conference on Software Engineering. IEEE.

[6] G. Boloix and P. N. Robillard, "A Software System Evaluation Framework," *IEEE Computer*, vol. 28, 12. pp. 17-26, 1995.

[7] J. P. Cavano and J. A. McCall, "A Framework for the Measurement of Software Quality," *ACM SIGSOFT Software Engineering Notes*, vol. 3, 5. pp. 133-139, 1978.

[8] A. T. W. Chu and R. E. Kalaba, "A Comparison of Two Methods for Determining the Weights Belonging to Fuzzy Sets," Journal of Optimization Theory and Applications, vol. 27, 4. pp. 531-538, 1979.

[9] T. Davis, "Toward a reuse maturity model," eds. M. L. Griss and L. Latour. pp. Davis_t-1-7, 1992. Proceedings of the 5th Annual Workshop on Software Reuse. University of Maine.

[10] M. S. Deutsch and R. R. Willis. "*Software Quality Engineering - A total Technical and Management Approach*," Englewood Cliffs: Prentice-Hall, 1988. 317 pages.

[11] G. R. Finnie, G. E. Wittig, and D. I. Petkov, "Prioritizing Software Development Productivity Factors Using the Analytic Hierarchy Process," *Journal of Systems and Software*, vol. 22, pp. 129-139, 1995.

[12] E. H. Forman, "Facts and Fictions about the Analytic Hierarchy Process," *Mathematical and Computer Modelling*, vol. 17, 4-5. pp. 19-26, 1993.

[13] M. L. Griss, "Software reuse: From library to factory," *IBM Systems Journal*, vol. 32, 4. pp. 548-566, 1993.

[14] S. Hong and R. Nigam. "Analytic Hierarchy Process Applied to Evaluation of Financial Modeling Software,". In: *Proceedings of the 1st International Conference on Decision Support Systems, Atlanta, GA*, Anonymous1981.

[15] J. W. Hooper and R. O. Chester. "*Software Reuse: Guidelines and Methods*," R.A. Demillo (Ed). New York: Plenum Press, 1991.

[16] ISO. "*Information technology - Software product evaluation - Quality characteristics and quidelines for their use, ISO/IEC 9126:1991(E)*," Geneve, Switzerland: International Standards Organization, 1991.

[17] P. Koltun and A. Hudson, "A Reuse Maturity Model," ed. W. B. Frakes. pp. 1-4, 1991. Proceedings of the 4th Annual Workshop on Software Reuse. University of Maine. Department of Computer Science.

[18] J. Kontio, "OTSO: A Systematic Process for Reusable Software Component Selection," CS-TR-3478, 1995. University of Maryland Technical Reports. University of Maryland. College Park, MD.

[19] J. Kontio, "A Case Study in Applying a Systematic Method for COTS Selection," 1996. Proceedings of the 18th International Conference on Software Engineering.

[20] J. Kontio and S. Chen, "Hypertext Document Viewing Tool Trade Study: Summary of Evaluation Results," 441-TP-002-001, 1995. ECS project Technical Paper. Hughes Corporation, ECS project.

[21] J. Kontio, S. Chen, K. Limperos, R. Tesoriero, G. Caldiera, and M. S. Deutsch, "A COTS Selection Method and Experiences of Its Use," 1995. Proceedings of the 20th Annual Software Engineering Workshop. NASA. Greenbelt, Maryland.

[22] H. Min, "Selection of Software: The Analytic Hierarchy Process," *International Journal of Physical Distribution & Logistics Management*, vol. 22, 1. pp. 42-52, 1992.

[23] J. S. Poulin, J. M. Caruso, and D. R. Hancock, "The business case for software reuse," *IBM Systems Journal*, vol. 32, 4. pp. 567-594, 1993.

[24] R. Prieto-Díaz, "Implementing faceted classification for software reuse," *Communications of the ACM*, vol. 34, 5.1991.

[25] C. V. Ramamoorthy, V. Garg, and A. Prakash, "Support for Reusability in Genesis," pp. 299-305, 1986. Proceedings of Compsac 86. Chicago.

[26] T. L. Saaty. "*Decision Making for Leaders*," Belmont, California: Lifetime Learning Publications, 1982. 291 pages.

[27] T. L. Saaty. "*The Analytic Hierarchy Process*," New York: McGraw-Hill, 1990. 287 pages.

[28] T. L. Saaty. "Analytic Hierarchy,". In: *Encyclopedia of Science & Technology*, Anonymous McGraw-Hill, 1992.pp. 559-563.

[29] T. L. Saaty, Expert Choice software 1995, ver. 9, rel. 1995. Expert Choice Inc. IBM. Windows 95.

[30] W. Schäfer, R. Prieto-Díaz, and M. Matsumoto. "*Software Reusability*," W. Schäfer, R. Prieto-Díaz, and M. Matsumoto (Eds). Hemel Hempstead: Ellis Horwood, 1994.

[31] P. J. Schoemaker and C. C. Waid, "An Experimental Comparison of Different Approaches to Determining Weights in Additive Utility Models," *Management Science*, vol. 28, 2. pp. 182-196, 1982.

[32] W. Tracz, "Reusability Comes of Age," *IEEE Software*, vol. 4, July. pp. 6-8, 1987.

[33] W. Tracz. "Software Reuse: Motivators and Inhibitors,". In: *Tutorial: Software Reuse: Emerging Technology*, ed. W. Tracz. Washington: IEEE Computer Society, 1988.pp. 62-67.

Note: Some of the technical reports and papers describing the OTSO method are available through "http://www.cs.umd.edu/users/jkontio/".

omit
this
page

SECTION 3—SOFTWARE MODELS

The technical papers included in this section were originally prepared as indicated below.

- "Characterizing and Modeling the Cost of Rework in a Library of Reusable Software Components," V. R. Basili, S. E. Condon, K. El Emam, R. B. Hendrick and W. L. Melo, International Conference on Software Engineering (ICSE-19), May 1997, pp. 282-291

56-61
415790

Characterizing and Modeling the Cost of Rework in a Library of Reusable Software Components

360737
12p

Victor R. Basili
University of Maryland
Computer Science
College Park, MD
20742 USA
basili@cs.umd.edu

Steven E. Condon
CSC
10110 Aerospace Rd.
Lanham-Seabrook,
MD
20706 USA
scondon@csc.com

Khaled El Emam
Fraunhofer IESE
Sauerwiesen 6
D-67661
Kaiserslautern
Germany
elemam@iese.fhg.de

Robert B. Hendrick
CSC
10110 Aerospace Rd.
Lanham-Seabrook
MD
20706 USA
bhendric@cscmail.csc.com

Walcelio Melo
CRIM
1801 McGill College
Montreal, Quebec
Canada
H3A 2N4
wmelo@crim.ca

ABSTRACT
In this paper we characterize and model the cost of rework in a Component Factory (CF) organization. A CF is responsible for developing and packaging reusable software components. Data was collected on corrective maintenance activities for the Generalized Support Software reuse asset library located at the Flight Dynamics Division of NASA's GSFC. We then constructed a predictive model of the cost of rework using the C4.5 system for generating a logical classification model. The predictor variables for the model are measures of internal software product attributes. The model demonstrates good prediction accuracy, and can be used by managers to allocate resources for corrective maintenance activities. Furthermore, we used the model to generate proscriptive coding guidelines to improve programming practices so that the cost of rework can be reduced in the future. The general approach we have used is applicable to other environments.

Keywords
Software Process Improvement, cost of rework, software metrics, classification models, prediction models.

INTRODUCTION

Previous research has shown that software reuse has a great potential to improve software development productivity and product quality [6][25][19]. For example, effective reuse of knowledge, processes and products from previous experience can decrease software development cost, reduce project delivery time and improve software quality [5][13].

However, reuse will not just happen—rather, components must be designed for reuse, and organizational elements must be created to enable projects to take advantage of the reusable software artifacts [2][11][26].

To facilitate the packaging and reuse of software development experience, an infrastructure called the Component Factory (CF) has been proposed [4]. The CF is a separate entity from the organization that produces applications. The CF is responsible for developing and packaging reusable software components. It creates and maintains a software component repository for future reuse and supplies reusable components to the development organization upon demand.

Several studies have empirically examined the characteristics of reusable components. For example, [22] investigated new versus reused code in a large collection of FORTRAN projects to analyze the pros and cons of creating a component from scratch versus modifying an existing component. Also in [25], eight medium scale Ada projects were assessed with respect to the defects found in newly developed and reused components. However, none of these works were concerned with software components that were developed exclusively for reuse. As far as we know, studies of reuse have focused on the side of the project organization, which reuses the components, rather than on the side of the CF, which creates the components. The primary reason for this different focus appears to be that not many software companies have a CF set up to develop and maintain reusable software components. Another potential explanation is that the few existing CFs have not collected sufficient data allowing them to evaluate the different aspects of the development and maintenance of reusable components.

In this paper we present a study that characterizes and models the cost of rework for a library of reusable components. This library, known as the Generalized Support Software (GSS) reuse asset library, is located at the Flight Dynamics Division (FDD) of NASA's Goddard Space Flight Center (GSFC). Component development began in 1993. Subsequent efforts focused on generating new components to populate the library and on implementing specification changes to satisfy mission requirements. The first application using this library was developed in early 1995.

The asset library currently consists of 921 Ada83 components totaling approximately 515 KSLOC. Based on a review of the first 58 GSS error correction reports, 102 of these 921 components have required error correction one or more times. We first characterize the 58 error correction reports in terms of source of error, class of error (both defined below), effort required to isolate the error, and effort required to correct the error. We then use a machine learning algorithm, C4.5 [21], to construct a model for approximate prediction of the cost of rework ("high" or "low"), using internal source code metrics of the components that are changed. The prediction model can help managers of the GSS asset library in the decision-making process by providing them with guidelines for predicting where corrective maintenance resources will most be needed. The model also consists of a set of easily interpretable coding guidelines that can be applied in improving current practices in order to reduce the cost of future rework. We expect that the process used to model rework in the GSS environment can be used in other environments to provide equally effective prediction models and coding guidelines that are appropriate for those environments.

In [16], various modeling techniques were used to predict maintenance productivity. In that article, the only product metric that was considered was a software size measure based on LOC. In [8], a machine learning algorithm was also used to predict the cost of rework in an Ada environment using internal product metrics. Unlike the components we have studied, however, the components analyzed in [8] were developed to satisfy specific application requirements. The current paper is, to our knowledge, the first that applies machine learning techniques to help manage the maintenance of reusable components, and to improve the way these components are produced in order to reduce maintenance costs within a CF.

The paper is organized as follows. It first presents the framework in which this study was conducted: the FDD, the Software Engineering Laboratory (SEL), and the GSS domain engineering and application deployment processes. The paper then presents the method for data collection and analysis. Then, the results of our analysis, including descriptive statistics that characterize the components and a predictive model of rework effort, are presented. We conclude the paper with a summary and directions for future work.

ENVIRONMENT OF THE STUDY

The FDD

GSFC manages and controls NASA's Earth-orbiting scientific satellites and also supports human space flight. For fulfilling flight dynamics responsibilities for both of these complex missions, the FDD developed and now maintains over 100 different software systems, ranging in size from 10 thousand source lines of code (KSLOC) to 300 KSLOC, and totaling approximately 4.5 million SLOC. This software covers three separate subdomains of the FDD mission: mission planning, orbit determination, and attitude[1] determination.

To increase the amount and type of reuse, and at the same time to drastically reduce the cycle time needed to develop and test new software systems, the FDD embarked on the GSS Domain Engineering Process in 1993. This process achieves rapid deployment by utilizing an object-oriented architecture in which the reusable assets are the generalized specifications for the reusable software components, as well as the reusable software components themselves (written in Ada83). Adopting this architecture and process results in a paradigm shift from *developing* software applications to *configuring* software applications. The GSS reuse asset library is the software component repository examined in this paper.

The SEL

The Software Engineering Laboratory began in 1976 with the goals of understanding the software process and product in the FDD, determining the impact of available technologies, and infusing the identified/refined methods, techniques, and products back into the environment. The approach has been to identify technologies with potential, apply them, and study their effect, based on studying the impact of the changes on such issues as cost, reliability, and quality. The participating organizations are the FDD, the University of Maryland, and Computer Sciences Corporation.

Over the years, the SEL has investigated numerous techniques and methods in over a hundred projects to understand and improve the software development process and product in their environment [20]. The result of this legacy is an organization and personnel that are quite interested in experimentation with new technologies and not averse to change. They are also a part of an environment that is quite successful at the type of work in which they are involved.

The approaches used for learning include the concept of the Experience Factory (EF). The focus of the EF in the SEL is on collecting metrics and lessons learned from standard projects and from special experiments, and then analyzing these data and packaging them into guide books, models, and training courses that can be spread to all areas of the development organization. The EF is different from the Project Organization (PO) which focuses on the development and maintenance of applications. Their relationship is depicted in Figure 1. The SEL EF has developed and packaged:

- resource models and baselines (e.g., local cost models, resource allocation models)
- change and defect baselines and models (e.g., defect prediction models, types of defects expected for the application)
- project models and baselines (e.g., actual vs. expected product size)
- process definitions and models (e.g., process models for Cleanroom, Ada waterfall model)
- method and technique evaluations (e.g., best method for finding interface faults)
- products and product parts (e.g., Ada generics for simulation of satellite orbits)
- quality models (e.g., reliability models, defect slippage models, ease of change models), and
- lessons learned (e.g., risks associated with an Ada development).

1. The term "attitude" refers to a spacecraft's orientation in space.

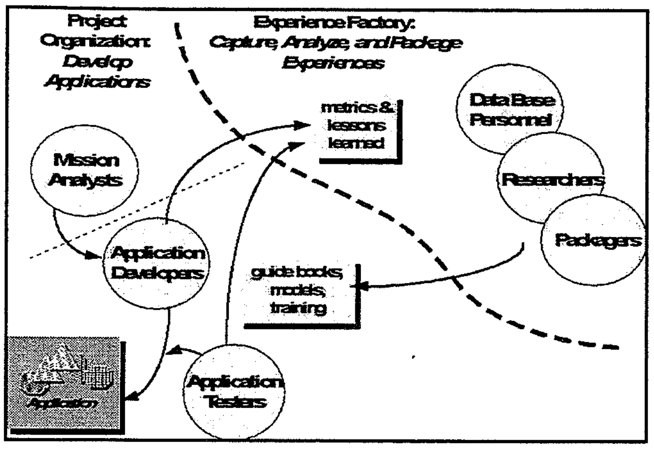

Project Organization: Develop Applications
Experience Factory: Capture, Analyze, and Package Experiences
metrics & lessons learned
Data Base Personnel
Mission Analysts
Researchers
Application Developers
Packagers
guide books, models, training
Application
Application Testers

Figure 1: The relationship between the Experience Factory and the Project Organization.

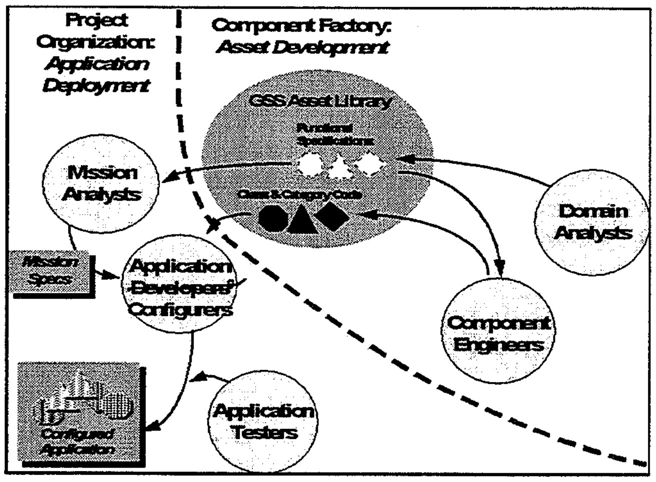


Figure 2: The relationship between the Component Factory and the Project Organization.

These models are built to understand the local environment, identify areas for improvement, attempt improvement via change, and form bases for evaluating that change against goals.

The Component Factory (CF) organization is a sub-organizational structure of the EF—an addition to the traditional EF. The CF focuses on generating a configuration architecture and reusable components, based on learning over time. This learning is in the form of analysis and synthesis of what is most effective for reuse (as well as what is expected to be needed for configuring applications) for the future development of products in a certain class. To staff a CF, some members of the PO functionally become members of the CF, although they may continue to think of themselves still as PO members. (See Figure 2.) That is, some mission analysts and application developers become domain analysts for the CF, and some application developers become component engineers for the CF. The domain analysts design the architecture and class specifications of the reuse asset library. The component engineers then construct the reusable class components. The PO takes advantage of this architecture and asset library to configure new systems. The PO's mission analysts now compare mission requirements to the asset library's functional specifications and produce a *mission specification* document that tells the PO's application configurers—application developers are no longer needed—how to configure the desired system from the reuse library assets. The traditional elements of the EF, together with the CF staff, then study how effective this process and the asset library have been for future improvement

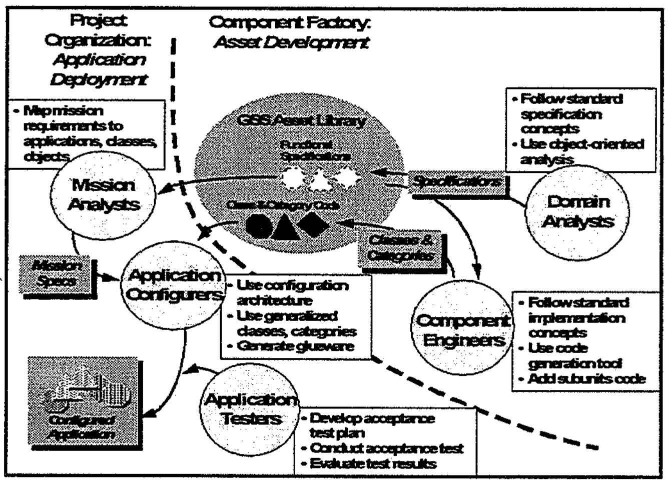

Project Organization: Application Deployment
Component Factory: Asset Development
• Map mission requirements to applications, classes, objects
GSS Asset Library
Functional Specifications
• Follow standard specification concepts
• Use object-oriented analysis
Mission Analysts
Specifications
Domain Analysts
Class & Category Code
Classes & Categories
Mission Specs
Application Configurers
• Use configuration architecture
• Use generalized classes, categories
• Generate glueware
Component Engineers
• Follow standard implementation concepts
• Use code generation tool
• Add subunits code
Configured Application
Application Testers
• Develop acceptance test plan
• Conduct acceptance test
• Evaluate test results

Figure 3: The GSS domain engineering and application deployment process.

The GSS Process

The activites of the CF and the PO in the GSS domain engineering and application deployment process are shown in more detail in Figure 3. The process relies on five functionally distinct teams, although some personnel may overlap between teams (particularly between the component engineers and the application configurers). The domain analysts write the class and category specifications. The component engineers code the classes and categories that, together with the specifications, make up the GSS reuse asset library. The mission analysts analyze the mission requirements and specify which classes need to be used for a given mission and how they should be configured. The application configurers configure the desired mission applications from the available classes and categories in the GSS reuse asset library, instantiate the generics, and perform integration testing of the application. The application testers cónduct acceptance testing of the configured mission application.

DATA COLLECTION AND ANALYSIS METHOD

Definitions

Errors are defects in the human thought process that are made while trying to understand and communicate given information, solve problems, or use methods and tools. Faults are concrete manifestations of errors within the software.

In this study, an *error* is represented by a single software Change Request Form (CRF) [15] filled by developers and configurers to institute and document a change to one or more components. A CRF results in modifications to one or more components in the reuse asset library. CRFs are also generated for enhancements, requirements changes, and adaptation. The current paper examines only error correction CRFs.

A *fault* pertains to a single component and is evidenced by the physical change of that component in response to a particular error CRF. In this study, we define a *component* as as an Ada file in configuration management. A *faulty* component version becomes a *fixed* component version after it is corrected. We are only interested in the faulty component versions.

Data Collection
We collected data on: (1) error identification and error correction (which follow initiation of a CRF), including the names and version numbers of the source code components that had faults in them, and (2) source code metrics characterizing these particular components.

Between 9th March 1994 and 21st September 1995, a total of 58 GSS error correction CRFs were generated, meaning 58 errors were identified. (In addition, 96 additional GSS CRFs were generated for requested enhancements, adaptations, and requirements changes.) Most of the GSS error correction CRFs were initiated by configurers, who uncovered problems during instantiation of the Ada generics and integration testing of the configured application, prior to turning over the configured application to acceptance testing. A very small minority of the CRFs—perhaps ten percent—were initiated by a maintainer of the reuse asset library following the report of a failed application test item by the independent tester group during conductance of acceptance testing of the application..

The CRF data analyzed by our study consisted of (1) the classification of errors by source and class, (2) the names of components changed to correct the errors, (3) the effort expended to isolate all faults associated with the error, and (4) the effort required to correct all of these faults. Each of these is described below.

Isolation and correction effort was measured on a 4-point ordinal scale: 1 hour, from 1 hour to 1 day, from 1 to 3 days, and more than 3 days. In addition, the maintainer provides the source of the error (requirements, functional specification, design, code, or previous change). Once an error is found during configuration and testing, the maintainer finds the cause of the error, locates where the modifications are to be made, and determines that all effects of the change are accounted for. Then the maintainer modifies the design (if necessary), code, and documentation to correct the error. Once the maintainer fixes the error, the maintainer provides the names of the components changed (in our case the faulty components). The maintainer also specifies the class of the error (initialization, internal/ external interface, user interface, database, algorithm, etc.).

The Amadeus tool [1] was used to extract source code metrics from all faulty component versions. A description of the source code metrics that were found useful is given in the results section of this paper. If after the extraction of some metrics it was found that they had zero variation (e.g., the number of Goto's), we excluded these metrics from further analysis.

Data Analysis: Characterization
The first data analysis task was to characterize or describe the errors. The objective of this characterization is to understand better the nature of the errors and how they are distributed. For this, basic pie charts were used. Furthermore, basic bivariate analysis using contingency tables and chi-square tests [24] was conducted to identify if there were any relationships between the source and class of errors and the rework effort.

Since the contingency tables tended to be sparse in some instances (i.e., cell frequencies approaching zero), we dichotomized each of the isolation and correction cost variables. We therefore considered isolation or correction effort of 1 hour as *Low*, and effort greater than 1 hour to be *High*.

Data Analysis: Modeling
A cost of rework model should allow: (1) the prediction of which components are likely to be associated with costly rework, and (2) provide programming guidelines that can be used to prevent costly rework in the future. The cost of rework is measured as the *total* effort taken to isolate and correct an error.

Unit of Analysis
The unit of analysis for developing the model is a faulty component version. During rework, a total of 118 changes were made to 102 components to fix these 58 errors. Four of the components were changed three times (i.e., on three different CRFs), 8 components were changed twice, and the remaining 90 were changed only once.

Approximately 75% of the components in the library are generated using a code generator. When software changes are necessary, maintainers do not make changes directly to the outputs of the code generator. Instead, the inputs to the code generator are changed, and new versions of the output components are generated. Given that rework effort is only directly affected by the characteristics of the component versions that are actually changed by the maintainers, component versions that are automatically regenerated by the code generator should not be included in our analysis. Where the components associated with a CRF include the input to the code generator as well as the output component, we excluded the modified output versions in our analysis. This leaves a total of 76 faulty component versions which are the basis of our analysis.

Model Specification
The model that we developed identifies component versions that are associated with costly rework rather than trying to predict the exact effort for reworking a component version. We therefore use the characteristics of a faulty component version as input into the model, and the total rework effort for the error as the output of the model. Given that the model

we developed is a classification model, it classifies a component version into ones of two rework cost categories: *Low Cost* and *High Cost.* (Note that these categories are different from the one described in the "Characterization" paragraph above because, for the model we are interested in *total* rework effort, while in characterization we look at isolation and correction separately.) This allows the model to predict whether a component version is associated with a costly, or otherwise, error.

Modeling Technique
The modeling technique that we used is a machine learning algorithm called C4.5 [21]. The C4.5 algorithm partitions continuous attributes, in our case the internal product metrics, finding the best threshold among the set of training cases to classify them on the dependent variable. As well as being useful for prediction, the generated tree provides decision rules characterizing component versions that fall into each one of the two rework cost categories.

We chose this technique because the models are straightforward to build and are also easy to interpret. In addition, this class of modeling techniques has been used in the software engineering literature to build prediction models [23], and therefore there already is some familiarity with it. Of course, other classification techniques, e.g., Optimized Set Reduction [9] or logistic regression [6], could have been used. However, our goal here is not to compare classification techniques.

Potential Application of the Model
A prediction identifying component versions that are going to be associated with costly errors can help managers allocate resources for the maintenance activities. The availability of rules as part of the model can help prevent high rework cost in the maintenance environment. For example, rules that characterize high rework cost can be treated as *proscriptive* programming guidelines for developing future components. It is on proscriptive rules that we focus in this study.

It should be noted, however, that the model does *not* identify which component versions in the asset library are likely to have faults, only which of the faulty versions should be more or less expensive to isolate and correct. Application of such predictions assumes that the manager knows *beforehand* which components are likely to contain a fault. Models for the prediction of fault-prone Ada components in the SEL environment have been developed in the past [9]. Once a component version has been identified as potentially fault-prone, then it is possible to predict the cost of rework category when fixing an error that leads to faults in that version. Using this additional information, a manager can improve the resource allocation for maintenance.

Dependent Variable
To build a classification model, we dichotomize our dependent variable, which is the total cost of rework. We converted the four effort categories into average values following [3]. We assumed an 8 hour day, and took the average value for each of the categories of rework. Therefore, the category of "1 Hour" was changed to 0.5 hours, the category of "1 hour to 1 Day" was changed to 4.5 hours, the category of "from 1 to 3 Days" was changed to 16 hours, and the category of "more than 3 Days" was changed to 32 hours. We then summed up these values for isolation and correction costs. This gives us an average overall rework cost. The median of total rework cost per CRF was 5 hours, and we used that as the cutoff point for dichotomization. Based on this dichomotomization, we have 33 component versions that were associated with errors requiring a low cost of rework and 43 that required a high cost of rework.

Independent Variables
Internal product metrics have been widely used to predict quality attributes such as productivity and software quality [14]. Here, we are interested in studying the use of internal product metrics of the faulty GSS component versions to predict the cost of rework. Previous research investigated the use of the characteristics of the change as the basis for the prediction of correction effort [10], however, the characteristics of the change are usually not available before the change is actually made (or at least not before isolation of the error). We only wanted to use information that would be available before isolation in order to develop a model for predicting total rework effort.

Evaluation of the Model
To evaluate the model, we need criteria for evaluating the overall model accuracy and for evaluating the strength of the rules. Evaluating model accuracy tells us how good the model is expected to be as a predictor. Evaluating the strength of the rules tells us the extent to which we can trust these rules as programming guidelines.

Evaluating Prediction Accuracy
Three criteria for evaluating the accuracy of predictions are the predictive validity criterion, and measures of correctness and completeness. These are defined below with reference to Table 1. Table 1 shows symbols for frequencies.

A criterion of prediction validity has been presented in [17]. This basically involves laying out the frequencies as in Table 1, and calculating the chi-square statistic. If the value is larger than a critical value then it is claimed that the model has predictive validity. The authors state that a model that does not meet the criterion of predictive validity should be rejected. This does not necessarily mean that a model that meets the predictive validity criteria should be accepted (it would be easy to demonstrate that if the classification model predicted all High Cost components as Low Cost and vice versa - i.e., very high misclassification - it would still have high predictive validity). We use this criterion to determine whether there is any association between the real rework cost of a component and its actual rework cost.

		Predicted Rework Cost	
		Low Cost	**High Cost**
Real Rework Cost	**Low Cost**	n_{11}	n_{12}
	High Cost	n_{21}	n_{22}

Table 1: Evaluating the accuracy of predicted classifications.

Correctness is defined as the percentage of component versions that were predicted to be costly to rework and were actually costly to rework. We want to maximize correctness because if correctness is low, then the model is identifying more component versions as being costly to rework when they really are not costly to rework, which could lead to an over-allocation of resources to making changes (i.e., wastage).

$$\text{Correctness} = \left(\frac{n_{22}}{n_{12}+n_{22}}\right) \times 100$$

Completeness is defined as the percentage of those component versions costly to rework and were predicted to be costly to rework. We want to maximize completeness because as completeness decreases, more versions that were costly to rework are mis-identified as not costly to rework, which would lead to a shortage of resources for making changes..

$$\text{Completeness} = \left(\frac{n_{22}}{n_{21}+n_{22}}\right) \times 100$$

In order to calculate values for correctness and completeness, we used a V-fold cross-validation procedure [7]. For each observation X in the sample, a model was developed based on the remaining observations (sample - X). This model was then used to predict whether observation X will have high rework or low rework. This validation procedure is commonly used when data sets are small.

Evaluation of Rules

The generated model from all 76 versions is also useful for providing proscriptive guidelines to programmers. The guidelines inform the programmers of the characteristics of faulty components that tend to require costly rework. By producing components that do not have these characteristics, there is a greater chance that components will be produced that are not costly to rework. There are two ways for evaluating such rules. First by measuring the number of cases that a rule classified correctly. Second, by appeal to the intuition of programmers in the environment (i.e., do the rules make sense to them).

RESULTS

Characterizing Errors

Distribution of Errors by Error Source

Figure 4 shows the overall distribution of errors (the 58 errors) by error source. Requirements and functional specification errors are those triggered by a misunderstanding of user requirements, and are introduced into the system by the process of transforming user requirements into project requirement specifications. Design errors are those introduced in the process of transforming requirements and specifications into detailed (component-level) design. Coding errors are those that occur when transforming the detailed design to code, such as mistyping a variable name, incorrectly coding an assignment statement, or incorrectly coding the exit criteria of a loop. Finally, errors resulting from a previous change are those that were not in the system until some other change was implemented (in which case the implementation of the previous change did not consider all of its possible effects, or the change was simply implemented incorrectly).

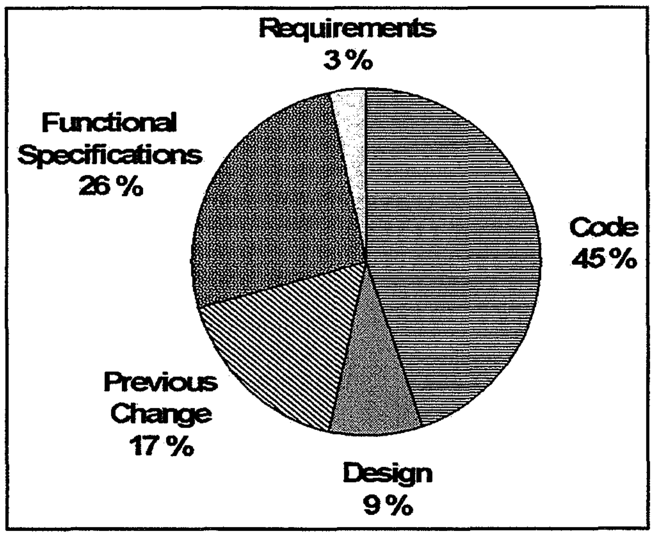


Figure 4: Distribution of errors by source.

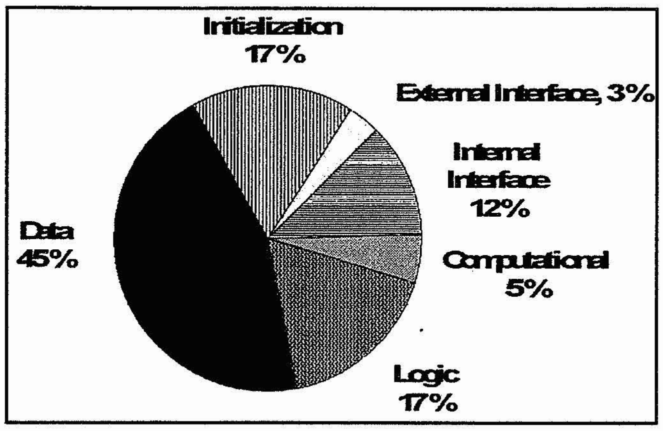


Figure 5: Distribution of errors by class.

Coding errors are responsible for approximately half of the errors found during acceptance testing (45%), followed by errors from requirements and functional specifications (29%), previous changes (17%), and finally design (9%).

It is interesting to note the small amount of design errors compared with requirements, specification, and coding errors. In part, this stems from the fact that most of the "design" of the GSS library is done during the specification phase. The object classes and the relationship between such classes of the three types of applications developed in the FDD (orbit, attitude and mission support) are, in fact, defined during the requirements analysis phase. The description of the methods of GSS classes are also done during the analysis.

Distribution of Errors by Error Class
The components in the library are based on generalizations of existing algorithms that were previously used in earlier systems. Therefore logic and computational errors are expected to be low (17% and 5% respectively as seen in Figure 5).

Initialization errors are responsible for 17% of the errors found during acceptance testing. (Initialization errors are those which result from an incorrectly initialized variable, failure to reinitialize a variable, or because a necessary initialization was missing; failure to initialize or reinitialize a data structure properly upon a component's entry/exit is also considered an initialization error). Once an application is created using the component library, a minimal set of integration tests are run. Particularly for an initial version of an application, this can result in a large number of initialization errors since this would be the first time the components have been configured in this fashion.

Data (value or structure) are responsible for the largest proportion of errors caught by the configurers and testers (see Figure 5). Data errors are those provoked by any error resulting from an incorrect use of a data structure. Examples of data errors are the use of incorrect subscripts for an array, the use of the wrong variable, the use of the wrong unit of measurement, or the inclusion of an incorrect declaration of a variable local to the component. One potential explanation for the large incidence of Data errors is that the Ada compiler catches a large proportion of the errors that would fall in the other categories, but many common Data errors will pass through compilation. This could be, for example, specifying a variable as POSITIVE instead of NATURAL.

Characterizing the Cost of Rework

Distribution of Errors By Cost of Isolation and Correction
Most of the GSS errors had a low isolation cost (60%) and a low correction cost (64%). It can be hypothesized that the design of the GSS architecture and the use of coding standards help reduce the time necessary to isolate errors, as well as the application of object-oriented design principles. Another explanation for the relatively low rework costs in general is that the people responsible for correcting errors in the GSS components have participated in the development of these components. They have, therefore, a good understanding of the design and realization strategies implemented into the code.

It should be noted that the median number of components changed for each CRF is 1 (maximum is 6), and the median number of other components examined is zero (with a maximum of 5). To test the hypothesis that the number of changed and examined components is related to the cost of isolation and correction, we used the Mann-Whitney U test [24]. No difference was found for the number of components examined when isolation cost was considered. When considering correction cost, it was found that more components are changed for high correction cost CRFs compared to low cost CRFs (at an alpha level of 0.05). No difference was found for number of components examined and correction cost.

Impact of Error Source on Rework Effort
Table 2 shows the distribution between the categories of error isolation cost and the error source. The contingency table contains the frequency of CRFs in each cell and the percentage of the total. We combined the Requirements and Functional Specification sources together into one "Analysis" category to avoid having expected frequencies less than one in the table. Likewise, Table 3 shows the distribution between the categories of error correction cost and the source of error.

Observation of the table indicates that for analysis sources, the isolation and correction costs tend to be low. We used the Pearson chi-square statistic to determine if there is a general association between source and rework cost. The probability values for both the isolation cost and the correction cost table were not significant at the 0.05 alpha level.[1]. Therefore, there is no association between source of error and isolation nor correction cost.

1. The approximation of the X^2 statistic to the chi-square distribution assumes that expected frequencies are not too small. This is usually interpreted to mean having at least 20% of expected frequencies greater than 5 and no cell having an expected frequency less than 1 for tables with degrees of freedom greater than 1 [12]. However, it has been suggested that the conventional chi-square statistic may be used for 2xc tables where all expected frequencies are as low as 1 [18].

	Code	Design	Analysis	Previous Change	Total
HIGH Isolation Cost	13 22.4%	2 3.45%	4 6.9%	4 6.9%	23
LOW Isolation Cost	13 22.4%	3 5.17%	13 22.41%	6 10.34%	35
Total	26	5	17	10	58

Table 2: Relationship between error source and isolation cost.

	Code	Design	Analysis	Previous Change	Total
HIGH Correction Cost	9 15.52%	3 5.17%	5 8.62%	4 6.9%	21
LOW Correction Cost	17 29.31%	2 3.45%	12 20.69%	6 10.34%	37
Total	26	5	17	10	58

Table 3: Relationship between error source and correction cost.

	Computational	Data	Initialization	Interface	Logic	Total
HIGH Isolation Cost	1 1.72%	11 19%	3 5.17%	2 3.45%	6 10.34%	23
LOW Isolation Cost	2 3.45%	15 25.86%	7 12%	7 12%	4 6.9%	35
Total	3	26	10	9	10	58

Table 4: Relationship between error class and isolation cost.

	Computational	Data	Initialization	Interface	Logic	Total
HIGH Correction Cost	1 1.72%	11 18.97%	2 3.45%	4 6.9%	3 5.17%	21
LOW Correction Cost	2 3.45%	15 25.86%	8 13.79%	5 8.62%	7 12%	37
Total	3	26	10	9	10	58

Table 5: Relationship between error class and correction cost.

		Predicted Rework Cost		
		Low Cost	High Cost	
Real Rework Cost	Low Cost	23	10	33
	High Cost	12	31	43
		35	41	76

Table 6: Predicted versus real rework categories.

Impact of Error Class on the Cost of Rework
Table 4 shows the distribution between the class of error and the isolation cost. We combined the Internal and External Interface categories to avoid having cells with expected frequencies less than one. The relationship between source and correction cost is depicted in Table 5. It can be observed from the tables that interface errors tend to cost less to isolate, and initialization errors tend to cost less to correct. Chi-square tests however do not identify any statistically significant association for either of the two tables.

Modeling the Cost of Rework
Table 6 shows the relationship between real and predicted rework. The predictive validity criterion for the contingency table presented in Table 6 is met at a one-tailed alpha level of 0.05. The values of correctness and completeness are shown in Figure 6. We found that correctness was 76% and completeness 72%. These values were perceived to be sufficient for decision making, especially when combined with expert judgment.

In this paper we are concerned with rules that characterize component versions that are costly to rework. The proportion of components that match the rule and are classified correctly by the rule give us a measure of how accurate a particular rule is. The model we developed had three interpretable rules for classifying high rework cost component versions. These are shown in Figure 7. For engineers involved with the GSS asset library, the rules were perceived to be intuitive in the sense that they express the fact that "more complicated things are more likely to cost more to correct." Moreover, the rules formalize the characteristics of the more complicated component versions.

The three rules can be used as maximal thresholds when developing new components. In some cases, there may be good design reasons for a component to exceed the threshold(s). Therefore the rules ought not be interpreted as strictly proscriptive. If a new component matches one or more of the rules, then the developer can decide whether it needs to be changed to reduce its potential for being associated with an error that is costly to isolate and correct.

Figure 8 shows the 3 internal product metrics that were found useful in developing this model. These 3 metrics were automatically selected by C4.5 from the set of metrics provided by Amadeus.

Correctness	76% (31/41)
Completeness	72% (31/43)

Figure 6: Correctness and completeness results for the prediction model.

Rule(s)	Accuracy
FunctionCalls > 38	100%
DeclarationStatements > 59	90%
ProgrammerExceptionsUsed > 2	83%

Figure 7: Proscriptive coding rules and their accuracy.

Metric Name	Brief Description
FunctionCalls	The number of function calls.
DeclarationStatements	The number of declaration statements, including those with and without initialization.
ProgrammerExceptionsUsed	The number of exceptions used in the file.

Figure 8: Description of the metrics that were found useful for building the model.

The proscriptive guidelines provided in Figure 7 were found from error data for a specific reusable components library. Caution should be exercised in attempting to generalize these rules beyond this context and applying them in a different environment. The overall approach we have used, however, can easily be generalized to other contexts. For example, after collecting the appropriate data, another organization could develop models for prediction and for producing coding guidelines to manage and reduce rework effort.

CONCLUSIONS
In this paper we reported on a study to model and understand the cost of rework in a library of reusable software components. We described how rework costs are distributed during the error correction process, and developed a model to predict the component versions that are associated with errors that are costly to rework. The model was also used to develop proscriptive coding rules that can be used by programmers as guidelines to reduce the cost of rework in the future.

Extensions of this work would include developing models for predicting components that have a high risk of faults (to help managers focus testing and inspections) and that can also be used to provide guidelines to programmers. We have used a specific set of internal product metrics for developing the model. These metrics tended to be counts of elements of a component. A different set of metrics that better characterize the structure and design of components may improve the predictive quality of the model, and also would provide guidelines for improving design practices.

Furthermore, it would be informative to compare models where cost of rework is the dependent variable with models where risk of fault is the dependent variable to determine if the derived guidelines from the two models are complementary or contradictory.

ACKNOWLEDGEMENTS
The authors wish to thank Lionel Briand, Steve Burke, Minna Makarainen, Mark Nicholson, Mari Sykes, and Carolyn Seaman for their help in data collection and preparation, comments on this work and reviews of earlier versions of this paper. The work reported in this paper was partially funded through NASA grant 01-5-26393 and NASA contract NAS5-31000 (subcontractor no. HQ-001057). Walcelio Melo was sponsored by Bell Canada, Software Quality Group, for the work reported in this paper.

REFERENCES

1. Amadeus Software Research Inc.: "Getting Started with Amadeus." Amadeus Measurement System. 1994.
2. R. Banker, R. Kauffman and D. Zweig: "Repository Evaluation of Software Reuse." In *IEEE Transactions on Software Engineering*, 19(4):379-389, April 1993.
3. V. Basili and B. Perricone: "Software Errors and Complexity: An Empirical Investigation." In *Communications of the ACM*, 27(1):42-52, January 1984.
4. V. Basili, and H.D. Rombach: "The TAME Project: Toward an Improvement-oriented Software Environment." In *IEEE Transactions on Software Engineering*, 14(6):758-773, June 1988.
5. V. Basili, and H.D. Rombach: "Support for Comprehensive Reuse." In *Software Engineering Journal*, 6(5):303-316, September 1991.
6. V. Basili, L. Briand and W. Melo: "A Validation of Object-Oriented Design Metrics as Quality Indicators." In *IEEE Transactions on Software Engineering*, December 1996.
7. L. Breiman, J. Friedman, R. Olshen and C. Stone: *Classification and Regression Trees*. Published by Wadsworth, 1984.
8. L. Briand, W. Thomas, and C. Hetmanski: "Modeling and Managing Risk Early in Software Development". In *Proceedings of the International Conference on Software Engineering*, pages 55-65, 1993.
9. L. Briand, V. Basili and C. Hetmanski: "Developing Interpretable Models with Optimized Set Reduction for Identifying High Risk Software Components." In *IEEE Transactions on Software Engineering*, SE-19 (11):1028-1044, 1993.
10. L. Briand and V. Basili: "A Classification Procedure for the Effective Management of Changes during the Maintenance Process". In *Proceedings of the International Conference on Software Maintenance*, pages 328-336, 1992.
11. G. Caldiera, and V. Basili: "Identifying and qualifying reusable software components." In *IEEE Software*, pp.61-70, February 1991, .
12. W. Cochran: "Some Methods for Strengthening the Common X^2 Tests". In *Biometrics*, 10:417-451, 1954.
13. W. Frakes, and S. Isoda: "Success factors of systematic reuse." In *IEEE Software*, pp.15-19, September 1994.
14. W. Harrison: "Software measurement: a decision-process approach." *Advances in Computers*, 39:51–105. 1994.
15. G. Heller, J. Valett and M. Wild: *Data Collection Procedure for the Software Engineering Laboratory (SEL) Database*. Technical Report SEL-92-002, Software Engineering Laboratory, 1992.
16. M. Jorgensen: "Experience with the Accuracy of Software Maintenance Task Effort Prediction Models". In *IEEE Transactions on Software Engineering*, 21(8):674-681, August 1995.
17. F. Lanubile and G. Visaggio: "Evaluating Predictive Quality Models Derived from Software Measures: Lessons Learned". Technical Report ISERN-96-03, International Software Engineering Research Network, 1996.
18. R. Lewontin and J. Felsenstein: "The Robustness of Homogenity Tests in 2xN Tables". In *Biometrics*, 21:19-33, 1965.
19. W. Lim: "Effects of Reuse on Quality, Productivity, and Economics." In *IEEE Software*, pp.23-30, September 1994.
20. F. McGarry, R. Pajerski, G. Page, S. Waligora, V. Basili, and M. Zelkowitz: *An Overview of the Software Engineering Laboratory*. Technical Report SEL-94-005, Software Engineering Laboratory, 1994.
21. J. R. Quinlan: *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, Sao Mateo, CA, 1993.
22. R. Selby: "Empirically Analyzing Software Reuse in a Production Environment". In W. Traca (ed.) *Software Reuse: Emerging Technology*. IEEE Press, 1988,
23. R. Selby and A. Porter: "Learning from Examples: Generation and Evaluation of Decision Trees for Software Resource Analysis." In *IEEE Transactions on Software Engineering*, 14(2):1743-1747. 1988.
24. S. Siegel and N. Castellan: *Nonparametric Statistics for the Behavioral Sciences*. McGraw Hill, 1988.
25. W. Thomas, A. Delis, and V. Basili: "An Analysis of Errors in a Reuse-oriented Development Environment." Technical Report, University of Maryland, 1995.
26. C. Wohlin, and P. Runeson: "Certification of Software Components." In *IEEE Transactions on Software Engineering*, 20(6):494-499, June 1994.

OMIT THIS PAGE

SECTION 4—TECHNOLOGY EVALUATION

The technical papers included in this section were originally prepared as indicated below.

- "A Knowledge-Based Approach to the Analysis of Loops," S. K. Abd-El-Hafiz and V. R. Basili, *IEEE Transactions on Software Engineering*, vol. 22, no. 5, S. May 1996, pp. 339-360
- *Experiences from an Exploratory Case Study with a Software Risk Management Method*, J. Kontio, H. Englund and V. R. Basili, Computer Science Technical Report, CS-TR-3705, UMIACS-TR-96-75, August 1996
- "An Empirical Study of Communication in Code Inspection," C. B. Seaman and V. R. Basili, *Proceedings of 19th International Conference on Software Engineering (ICSE-19),* May 1997, pp. 96-106
- "How Reuse Influences Productivity in Object-Oriented Systems," *Communications of the ACM*, V. R. Basili, L. C. Briand and W. L. Melo, vol. 39, no. 10, October 1996, pp. 104-116

A Knowledge-Based Approach to the Analysis of Loops

S7-61

115791

3 60741

Salwa K. Abd-El-Hafiz, *Member, IEEE Computer Society*, and Victor R. Basili, *Fellow, IEEE*

22p

Abstract—This paper presents a knowledge-based analysis approach that generates first order predicate logic annotations of loops. A classification of loops according to their complexity levels is presented. Based on this taxonomy, variations on the basic analysis approach that best fit each of the different classes are described. In general, mechanical annotation of loops is performed by first decomposing them using data flow analysis. This decomposition encapsulates closely related statements in events, that can be analyzed individually. Specifications of the resulting loop events are then obtained by utilizing patterns, called plans, stored in a knowledge base. Finally, a consistent and rigorous functional abstraction of the whole loop is synthesized from the specifications of its individual events. To test the analysis techniques and to assess their effectiveness, a case study was performed on an existing program of reasonable size. Results concerning the analyzed loops and the plans designed for them are given.

Index Terms—First order predicate logic, formal specifications, knowledge base, loops, program understanding, reverse engineering.

1 Introduction

PROGRAM understanding plays an important role in nearly all software related tasks. It is vital to the maintenance and reuse activities. Such activities cannot be performed without a deep and correct understanding of the component to be maintained or reused. Program understanding is also indispensable for improving the quality of software development activities such as code reviews, debugging, and some testing approaches. All these development activities require programmers to read and understand programs.

Due to the importance of program understanding, there has been considerable research on techniques and tools for analyzing and understanding computer programs. Within these efforts, substantial interest is usually directed towards the specific topic of analyzing loops. This interest stems mainly from inherent reasoning difficulties involving repeated program state modifications and the fact that loops have a major effect on program understandability [42].

To analyze loops and reason about their properties, some approaches define heuristics that can be used to guide a search for a loop invariant [19] or function [32]. However, heuristic techniques in general are not always useful. After applying the heuristics a considerable number of times, one may or may not succeed in finding a correct invariant or function. Other approaches focus on developing algorithmic techniques for finding the invariants or functions of specific simple classes of loops. The research performed by Basu and Misra [8], Dunlop and Basili [12], Katz and Manna [26], and Wegbreit [49] is representative of these loop analysis approaches. These algorithmic approaches analyze loops through the use of formal, semantically sound, and unambiguous notation. Although some of them provide guidelines on how to mechanically generate loop invariants or functions, no algorithmic techniques were actually used to implement automatic analysis systems. A different approach, that analyzes loops by mechanically decomposing them into smaller fragments, was adopted by Waters [47]. Even though Waters' approach does not address the issue of how to use this decomposition to mechanically annotate loops, it is especially interesting because of its practicality.

To analyze complete programs, the knowledge-based approaches utilize a knowledge base of plans in providing intelligent analysis results. Plans are defined as units of knowledge representing, or necessary for identifying, abstract programming concepts [15], [16], [24], [37], [38], [48]. These approaches are inspired by the cognitive studies [31], [41], [43] which suggest that the understanding process is one in which programmers make use of stereotyped solutions to problems in making sophisticated high-level decisions about a program. These knowledge-based approaches are all implemented, to varying degrees, in automatic analysis systems. Some of these approaches are: graph-parsing [38], [50]; top-down analysis using the program's goals as input [23], [24]; top-down analysis using a functional representation of programs that relates the program code and goals to a proof of correctness [6], [33]; heuristic-based object-oriented recognition [15], [16]; transformation of a program into a semantically equivalent but more abstract form with the help of plans and transformation rules [27], [29], [46]; and decomposition of a program into smaller more tractable parts using control flow analysis [17] or program slicing [18]. Even though these approaches demonstrate the feasibility and usefulness of the automation of program understanding, they lack some important features.

Most of the knowledge-based program analysis and understanding approaches produce program documentation

• *S.K. Abd-El-Hafiz is with the Engineering Mathematics Department, Faculty of Engineering, Cairo University, Giza, Egypt.
E-mail: elhafiz@cairo.eun.edu.*
• *V.R. Basili is with the Department of Computer Science, University of Maryland, College Park, MD 20742. E-mail: basili@cs.umd.edu.*

*Manuscript received Mar. 11, 1994; revised Jan. 19, 1996.
Recommended for acceptance by D. Wile.
For information on obtaining reprints of this article, please send e-mail to: transse@computer.org, and reference IEEECS Log Number S95491.*

1996 IEEE. Reprinted, with permission, from IEEE Transactions on Software Engineering, vol. 22, no. 5, pp. 339-360; May 1996

that is generally in the form of structured natural language text [9], [15], [16], [17], [24], [36], [38], [50]. Such informal documentation gives expressive and intuitive descriptions of the code. However, there is no semantic basis that makes it possible to determine whether or not the documentation has the desired meaning. This lack of a firm semantic basis makes informal natural language documentation inherently ambiguous.

Some of the knowledge-based approaches rely on real-time user-supplied information that might not be available at all times. For instance, goals a program is supposed to achieve [6], [24] or transformation rules that are appropriate for analyzing a specific code fragment [27], [46] are not always clear to the user. Others have difficulty in analyzing nonadjacent program statements [29]. In addition, a significant amount of program analysis and understanding research used toy programs that are less than 100 lines of code to validate proposed approaches. Realistic evaluations of these approaches, which give quantifiable results about recognizable and unrecognizable concepts in real and existing programs, are needed. Such evaluations can also serve as a basis for empirical studies and future comparisons with other approaches [40].

To address the above-mentioned drawbacks, we present a knowledge-based approach to the automation of program analysis. It combines and builds on the strengths of a practical program decomposition method [47], the axiomatic correctness notation [19], and the knowledge-based analysis approaches. It mechanically documents programs by generating first order predicate logic annotations of their loops. The advantages of predicate logic annotations are that they are unambiguous and have a sound mathematical basis. This allows correctness conditions to be stated and verified, if desired. Another advantage is that they can be used in assisting formal development of software using such languages as VDM and Z [25], [45].

A family of analysis techniques has been developed and tailored to cover different levels of program complexity. This complexity is determined by classifying while loops along three dimensions. The first dimension focuses on the control computation of the loop. As defined by Pratt [35], the control computation for a loop is that part concerned with the initialization, modification, and testing of the variables which determine the flow of control into, through, and out of the loop. The second dimension focuses on the complexity of the loop condition as determined by the number of clauses it has. The third dimension focuses on the complexity of the loop body. Based on this taxonomy, the analysis techniques that can be applied to the different loop classes are described.

In general, we annotate loops with predicate logic assertions in a step-by-step process as depicted in Fig. 1 [1]. The analysis of a loop starts by decomposing it into fragments, called *events*. Each event encapsulates the loop parts that are closely related, with respect to data flow, and separates them from the rest of the loop. The resulting events are then analyzed, using plans stored in a knowledge base, to deduce their individual predicate logic annotations. Finally, the annotation of the whole loop is synthesized from the annotations of its events.

This study tests several hypotheses related to the presented analysis approach:

- A loop complexity dimensions are indicators of its amenability to analysis.
- The loop decomposition and plan design methods can make the plans applicable to many loops that are different in their designs and functions. This, in turn, can increase plan utilization.
- The analysis techniques can be automated.

To test the first two hypotheses and to characterize the practical limits of the analysis approach, a case study on a set of 77 loops in an existing Pascal program for scheduling university courses has been performed. The program has 1,400 executable lines of code and the loops analyzed have the usual programming language features such as pointers, procedure and function calls, and nested loops. However, the loops analyzed do not involve recursive function and procedure calls. Recursion is not currently being handled by our analysis approach. To test the third hypothesis, a prototype tool, which annotates loops with predicate logic annotations, was developed.

Section 2 of this paper gives some of the definitions used. Section 3 introduces the loop taxonomy. Sections 4 and 5 describe the techniques used for analyzing flat and nested loops, respectively. Section 6 discusses the approach presented and highlights its advantages and limitations. Section 7 describes how the case study was performed and gives the results of the analysis. Section 8 briefly explains the design and structure of the implemented prototype tool. Finally, conclusions and future research directions are given in Section 9. Appendices A and B give the notation and acronyms used throughout the rest of the paper.

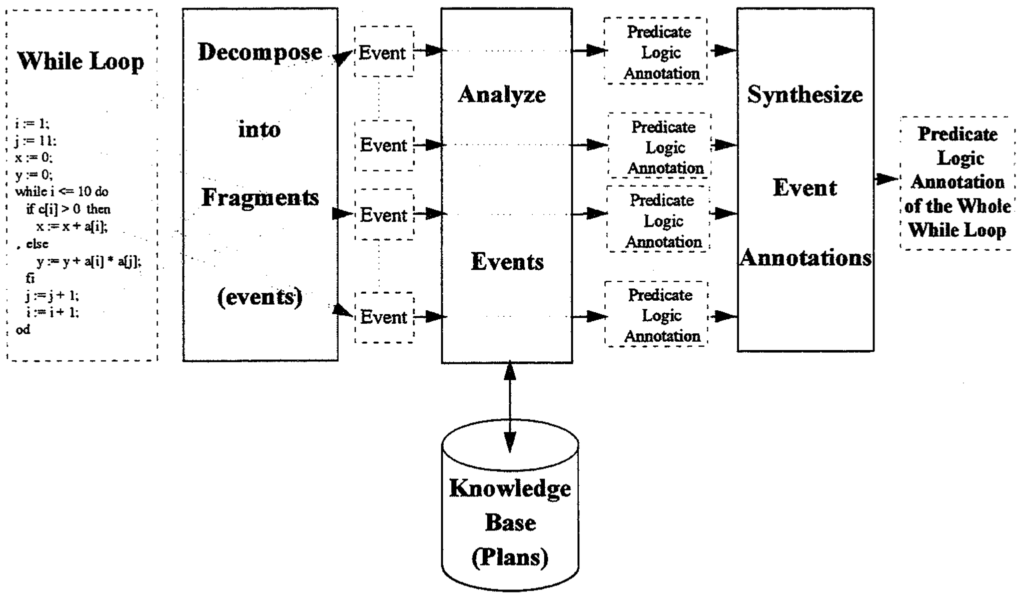
2 Definitions

We start by defining some of the notation used throughout this paper. First, we give the definitions related to the representation of while loops.

A *control-flow graph* is a directed graph that has one node for each simple statement and one node for each control predicate. There is an edge from node I to node J if an execution of J can immediately follow that for I [21].

Let the *abstract representation of the while loop* be *while B do S* where the condition *B* has no side effects and the statements *S* are representable by a single-entry single-exit control-flow graph. This representation abstracts from the syntax of the specific imperative programming language being used. Though the approach described here applies to all loops having this abstract representation, examples and illustrations are given using Pascal. Using this abstract representation, a *control variable* of the while loop is a variable that exists in the condition *B* and is modified in the body *S*. The sequence of values *scanned* by a control variable are these values that get assigned to the control variable and actually used in the loop body.

Now, we give some definitions that introduce the language and terminology used in the analysis. A *concurrent assignment* is a statement in which several variables can be assigned simultaneously. We use the form $v_1, v_2, \ldots, v_n := e_1,$



```
i := 1;
j := 11:
x := 0;
y := 0;
while i <= 10 do
   if c[i] > 0 then
      x := x + a[i];
   else
      y := y + a[i] * a[j];
   fi
   j := j + 1;
   i := i + 1;
od
```

While Loop
Decompose into Fragments (events)
Event
Event
Event
Event
Analyze Events
Predicate Logic Annotation
Predicate Logic Annotation
Predicate Logic Annotation
Predicate Logic Annotation
Synthesize Event Annotations
Predicate Logic Annotation of the Whole While Loop
Knowledge Base (Plans)

Fig. 1. Overview of the analysis approach.

$e_2, \ldots, e_n$ to assign every ith expression from the right hand list to its corresponding ith variable from the left hand list [14], [32]. A *conditional assignment* is a set of one or more guarded concurrent assignments separated by commas ','. When the guard (i.e., the Boolean expression), of a concurrent assignment is satisfied, the modifications performed on a variable are given by the concurrent assignment [14], [32]. Similar to Gries' definition of the alternative command, all the guards must be well defined [14]. However, it is possible that all guards evaluate to false. In this case, no variable is modified (i.e., the conditional assignment evaluates to a skip command [14]). It should also be noted that because we are only analyzing deterministic programs, all the guards are mutually exclusive.

Any variable assigned in a conditional assignment defines the *data flow out* of the statement. Any variable referenced by a conditional assignment defines the *data flow into* the statement. Two conditional assignments are said to be *circularly dependent* if some variable is responsible for data flow out of one statement and into the other, either directly or indirectly, and vice versa.

3 A Loop Taxonomy

To design the analysis techniques that best fit different levels of program complexity, we classify while loops along three dimensions. The first dimension focuses on the control computation part of the loop. The other two dimensions focus on the complexity of the loop condition and body. Along each dimension, a loop must belong to one of two complementary classes as shown in Table 1. In this classification, the loops in the middle column are expected to be more amenable to analysis than the corresponding ones in the right column.

Within the first dimension, we differentiate between simple and general loops. *Simple loops* have a behavior similar to that of *for* loops. They are defined by imposing two restrictions: the loop has a unique control variable, and the modification of the control variable does not depend on the values of other variables modified within the loop body. Loops that do not satisfy these conditions are called *general loops*.

Along the second dimension, the complexity of the loop condition can vary between two cases. In the *noncomposite* case, *B* is a logical expression that consists of one clause of the conjunctive normal form [39]. In the *composite* case, more than one clause exists. Along the third dimension, the complexity of the loop body varies between *flat* and *nested* loop structures. In flat loop structures, the loop body cannot include other loops. In nested structures, however, the loop body includes one or more loops.

TABLE 1
THE THREE DIMENSIONS USED FOR CLASSIFYING LOOPS

Dimension	Complementary classes	
1. Control computation	Simple loop	General loop
2. Complexity of condition	Noncomposite condition	Composite condition
3. Complexity of body	Flat loop	Nested loop

4 Analysis of Flat Loops

As depicted in Fig. 2, the analysis of flat loops is performed in a step-by-step process divided into four main phases. Descriptions of these phases and their application to the example shown in Fig. 3 are given in the remainder of this section [3]. In this example, a simple loop with a noncomposite condition scans a segment of the array *capacity* searching for its minimum.

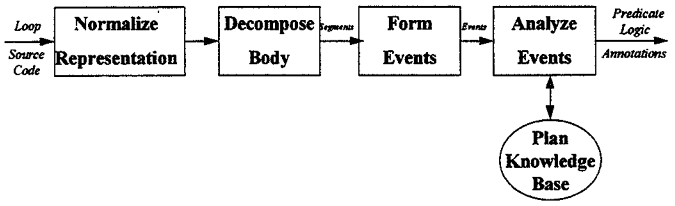


Fig. 2. Analysis of flat loops.

```
j, index, min, num_of_rooms: integer;
capacity: array[1 .. max_rooms] of integer;
:
while j < num_of_rooms + 1 do begin
    if capacity[j] < min then begin
        index := j;
        min := capacity[j];
    end;
    j := j + 1
end;
```

Fig. 3. Analysis of flat loops.

4.1 Normalization of the Loop Representation

The purpose of this phase is to make the loop representation independent of the programming language and the implementation specific details.

Normalization of the Loop Condition. The loop condition is converted into a standard normal form, which is the *conjunctive normal form*. This normal form represents a well-formed formula (wff) in predicate logic as a conjunction of clauses where a clause is defined to be a wff in conjunctive normal form but with no instances of the *and* connector [39]. For example, the loop condition $x < a$ *or* $(y < b$ and $z < c)$ is transformed to the conjunction of two clauses. The first clause is $(x < a$ *or* $y < b)$ and the second is $(x < a$ *or* $z < c)$.

Normalization of the Loop Body. A single unwinding of the loop body is performed by symbolic execution [4] that gives the net modification performed on each variable in one iteration of the loop, if any [7]. We use the conditional assignment notation to represent the result of this symbolic execution.

After converting the loop condition and body into the aforementioned standard forms, they are further normalized by performing some simplifications. Arithmetic expressions are simplified by converting them into an internal canonical form for polynomials, manipulating them, and converting them back to their external form [34]. Predicate simplifications are performed using rule-based transformations. Since the simplification details are dependent on our specific prototype implementation, they are not discussed during the description of the analysis phases.

For the loop given in Fig. 3, the condition is already in conjunctive normal form containing the one clause $j < num\_of\_rooms + 1$. The symbolic execution does not change the body of the loop. However, the net modification performed on each variable is given in the form of a conditional assignment as follows:

Name	Conditional Assignment
C_1	$capacity[j] < min \Rightarrow index := j,$
C_2	$capacity[j] < min \Rightarrow min := capacity[j],$
C_3	$true \Rightarrow j := j + 1$

4.2 Decomposition of the Loop Body

To facilitate the mechanical generation of loop annotations, the symbolic execution result is uniquely decomposed into *segments* of code that can be analyzed separately. Each segment encapsulates the statements that are interdependent with respect to data flow. The *loop segments* are partitions of the loop body symbolic execution result. Each segment consists of a maximal set of conditional assignments such that any two conditional assignments in the set are circularly dependent.

To obtain the loop segments, we assume that the conditional assignments of the symbolic execution result correspond to the nodes of a directed graph. An edge from node C_i to node C_k exists if and only if there is data flowing out of C_i into C_k and C_i and C_k are distinct. The strongly connected components of this graph correspond to the loop segments [10].

For the loop shown in Fig. 3, the three conditional assignments of the symbolic execution result form a directed graph G with three edges: two from C_3 to C_1 and C_2, and one from C_2 to C_1. Since there are no cycles in G, its strongly connected components correspond to its nodes. Thus, the loop segments correspond to C_1, C_2, and C_3.

Because the analysis of a segment might be dependent on the analysis results of other segments, a segment analysis result should be obtained before analyzing the segments dependent on it. That is why we need to order the segments according to their data flow dependencies [47]. Assuming that S is the set of segments in the loop body, the order of each segment is determined by the following algorithm:

```
1) Set $m$ to 1.
2) While the number of segments in $S$ is $\geq 1$ do
    a) Identify the maximal subset of $S$ such that each of
       its segments does not have data flowing out into
       other segments of $S$.
    b) Let the order of the identified segments be $m$.
    c) Remove the identified segments from $S$.
    d) Increment $m$.
3) Let the final order of each segment be ($m$ – old order).
```

Step 2 of the above algorithm assigns unique orders to the segments such that order of S_i > order of S_k if and only if there is data flowing, either directly or indirectly, from segment S_i to segment S_k. Step 3 produces an irreflexive partial order of the segments. The resulting ordering relation 'analyzed before,' is denoted by '$\rightarrow$'. It is irreflexive because it is meaningless for a segment to be analyzed before itself. It satisfies the antisymmetric property because any two distinct segments, by definition, have no circular dependencies. The design of the above algorithm ensures the satisfaction of the transitive property. Moreover, it is possible for two segments to be unrelated (i.e., they can have the same order).

In the example given in Fig. 3, let the segments of the loop be S_1, S_2, and S_3 that correspond to C_1, C_2, and C_3, respectively. The orders assigned to these segments using the above algorithm are:

Order	Name	Segment
1	S_3	$j := j + 1$
2	S_2	$capacity[j] < min \Rightarrow min := capacity[j]$
3	S_1	$capacity[j] < min \Rightarrow index := j$

Notice that the segment that defines j, S_3, has the lowest order because the other two segments, S_1 and S_2, reference j (i.e., $S_3 \rightarrow S_1$ and $S_3 \rightarrow S_2$). Similarly, $S_2 \rightarrow S_1$ because *min* is defined in S_2 and referenced in S_1. Since the premise of the conditional assignment that modifies j is *true*, it is removed.

4.3 Formation of the Loop Events

To represent the abstract concepts in a loop, we use the loop body segments and the clauses of the loop condition to form the loop *events*. We define two categories of loop events: *basic events* and *augmentation events*.

Basic Events (*BEs*) are the fragments that constitute the control computation of the loop. A BE consists of three parts: the *condition*, the *enumeration*, and the *initialization*. The *condition* consists of only one clause from the loop condition. The *enumeration* is a segment responsible for the data flow into the *condition* (i.e., the variables assigned in the *enumeration* are referenced by the *condition*). The *initialization* is the initialization of the variables defined in the *enumeration*.

To form BEs, each clause of the loop condition is used as the condition of a unique BE. Then, the enumeration of each BE is constructed from the highest order segment(s) having data flow into the condition. If a clause has no segment responsible for the data flow into it, this means that this clause is redundant and should be removed from the loop condition. If a segment is responsible for the data flow into the loop condition but remains with no clause associated with it, this segment is used as the enumeration of a new BE whose condition is set to *true*. The initializations of the control variables defined in a BE are included in the initialization part.

The BE of the loop given in Fig. 3 is formed by combining the unique condition clause, ($j < num\_of\_rooms + 1$), with the only segment that is responsible for the data flow into it, S_3. Since the loop under consideration has no initializations, we use the notation $j?$ to denote the initial value of the variable j. As a result, the BE has the following form:

condition: $j < num\_of\_rooms + 1$
enumeration: $j := j + 1$
initialization: $j := j?$

Augmentation Events (AEs) are the fragments that constitute loop computations other than the control computation. An AE consists of two parts: the *body* and the *initialization*. The *body* is one segment of the loop body that is not responsible for the data flow into the loop condition. The *initialization* is the initialization of the variables defined in the *body*.

After identifying the BEs, the AEs bodies are formed from the segments of the loop that did not get used in BEs. The initialization of each variable defined in an AE is then included in it.

For the loop shown in Fig. 3, the remaining segments S_2 and S_1 constitute the bodies of two AEs given below. The notation *min?* and *index?* are used to denote the initial values of the variables *min* and *index*

1) AE 1
   body: $capacity[j] < min \Rightarrow min := capacity[j]$
   initialization: $min := min?$
2) AE 2
   body: $capacity[j] < min \Rightarrow index := j$
   initialization: $index := index?$

Finally, we give each event (basic or augmentation) the same order as the segment it utilizes. This enforces the condition that the variables referenced in an event are either defined in a lower order event or not modified within the loop at all. As mentioned in the previous subsection, this makes it possible to propagate the results of analyzing an event to the analysis of other events dependent on it.

The three events of the loop shown in Fig. 3 are thus ordered as follows.

1) BE (order 1)
   condition: $j < num\_of\_rooms + 1$
   enumeration: $j := j + 1$
   initialization: $j := j?$
2) AE (order 2)
   body: $capacity[j] < min \Rightarrow min := capacity[j]$
   initialization: $min := min?$
3) AE (order 3)
   body: $capacity[j] < min \Rightarrow index := j$
   initialization: $index := index?$

4.4 A Knowledge Base of Plans

To analyze the loop events, we utilize plans stored in a knowledge base. We use the term 'plan' to refer to a unit of knowledge required to identify an abstract concept in a program. Our plans are used as inference rules [15], [16]. Their basic structure is divided into two parts: the antecedent and the consequent. When a loop event matches a plan antecedent, the plan is fired. The instantiation of the information in the consequent represents the contribution of this plan to the loop specifications. To guarantee the accuracy of the predicate logic specifications included in the consequents, no partial matches with antecedents are allowed (i.e., the antecedent has to be completely matched).

The knowledge base is designed so that any two plans do not have similar antecedents. Thus, a loop event can only match the antecedent of a unique plan. It should also be noted that the possibility of designing as many plans as the number of loop events in a specific program is reduced because the loop events encapsulate abstract concepts that can occur in different loops. Section 7 will examine this issue of the knowledge base size in more detail.

Corresponding to the two event categories, we have two plan categories: *Basic Plans* (*BPs*) and *Augmentation Plans* (*APs*). BPs analyze BEs and APs analyze AEs. Plans are further classified according to the kind of loops they analyze.

In case of simple loops, the sequences of values scanned by the control variable during and after the execution of a simple loop can be easily written because the control computation is isolated from the rest of the loop. The loop condition, the control variable's initial value, and the net modification performed on the control variable in one loop iteration, if any, provide sufficient information for writing these sequences. This specific information about the control computation of the loop can be used to produce equally specific loop specifications. The plans that analyze simple loops can include these sequences and utilize them in writing the loop specifications. The loop specifications produced for simple loops are the preconditions, invariants, and postconditions. The formal approach used for deriving the invariants is the axiomatic approach [14], [19], [20]. In this approach, if we assume that B, S, S_0, I, P, and Q are the loop condition, body, initialization, invariant, precondition, and postcondition, respectively, then the relations between them are given in the following rules. In these rules, the notation $P\{S\}Q$ means that if the predicate P is true before executing the first statement of the program part S, and if S terminates, then the predicate Q will be true after execution of S is complete.

I {while B do S} I *and* $\neg B$,
(I *and* $\neg B \Rightarrow Q$), and
($P \Rightarrow T$), where T is deduced from $T\{S_0\}I$.

The analysis of *general loops* is not as straightforward as that of simple ones. In many cases, it might not be easy, or even possible, to obtain such specific knowledge because the control computation of the loop is not as determinate and isolated as in the case of simple loops. The sequences of values scanned by the control variable(s) and the program state at the end of the loop are usually dependent on the combined indeterminate effects of several events and the values of some program variables. As a result, the plans that analyze general loops neither include the aforementioned sequences nor utilize them in writing the loop specifications. The loop postcondition can only be deduced after the synthesis of the loop invariant. The postcondition is formed by taking the conjunction of the loop invariant with the negation of the loop condition [14], [19]. Using this method to obtain the loop postcondition yields predicates that might not be as informative and concise as those of simple loops. As a result, additional simplifications might be needed to reduce the complexity and improve the readability of general loops postconditions.

For instance, consider the simple loop shown in Fig. 3. The sequence scanned by the control variable at any point during the loop execution is $j?$ to $j - 1$. This sequence is needed to write the part of the invariant:

$$min = MIN(\{min?\} \cup \{capacity[j? \,..\, j-1]\}),$$

where $MIN(s)$ is the minimum of the set s and $\cup$ is the set union operator. The final sequence of values scanned by the control variable in this loop is $j?$ to *num_of_rooms*. This sequence is needed to write the part of the postcondition:

$$min = MIN(\{min?\} \cup \{capacity[j? \,..\, num\_of\_rooms\,]\}).$$

In the general loop given in Fig. 4, however, there is no guarantee that the final sequence scanned by the control variable j will be $j?$ to *num_of_rooms*. The value of the final sequence is dependent on the interaction of the two events that modify *flag* and j, and the contents of the variables *capacity* and *limit*. As a result of this generality of the control computation, the sequences of values scanned by the control variable(s) and, consequently, the postcondition parts of the individual events cannot be written.

```
while (j <= num_of_rooms + 1) and (flag = false) do begin
    if capacity[j] < limit then begin
        index := j;
        flag := true
    end;
    j := j + 1
end
```

Fig. 4. Example of a general loop.

To accommodate the differences between simple and general loops, we have two categories of BPs. Determinate BPs (DBPs) contain in their consequents information regarding the postcondition and the sequences of values scanned by the control variable. Indeterminate BPs (IBPs), on the other hand, do not contain such information. We also have two categories of APs. Simple APs (SAPs) utilize the above sequences in writing the loop specifications, including its postcondition. General APs (GAPs) do not include the loop postcondition part or utilize the above sequences. These plan categories are shown in Fig. 5. It should be noticed that because the information contained in the consequents of IBPs is a subset of that contained in the consequents of DBPs, DBPs can be used in analyzing general loops. In such cases, we neglect the information regarding the control sequences and the postcondition in the DBPs consequents. However, because IBPs consequents do not contain such specific information, IBPs cannot be used in analyzing simple loops.

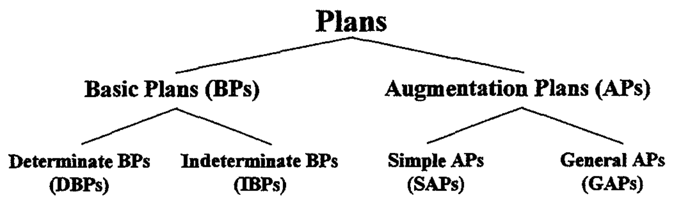


Fig. 5. Plan categories.

In general, The information included in a plans antecedent and consequent are described below. In this description, the words printed in **bold** correspond to fields in the plans (see Figs. 6 and 7).

An antecedent contains the following information:

1) An individual listing of the control variables, in the **control-variables** part, which serves to underscore their importance and to facilitate the design, readability, and comprehension of the plan.
2) Generic patterns of BEs and AEs that are used to match stereotyped loop events.
3) Knowledge needed for the correct identification of the plans such as data taype informaiton and the results of analyzing previous events. This knowledge is given in the **firing-condition**.

A consequent includes the following information:

1) Knowledge necessary for the annotation of loops with their Hoare-style [19] specifications. The **precondition** and **invariant** have the usual meaning [19]. The **postcondition** part gives information, in case of simple loops, about the variables values after the loop execution ends. It is correct provided that the loop executes at least once. If the loop does not execute, no variable gets modified.
2) In case of DBPs, knowledge about the sequence of values scanned by the control variables at any point during and after the loop execution is captured in **sequence** and **final-sequence**, respectively.

Fig. 6 and Fig. 7 show two example plans of the categories DBP and SAP, respectively. To convey the basic analysis ideas within a reasonable space limit, we only show simplified versions of the plans. The suffix '#' is used to indicate terms in the antecedent (or consequent) that must be matched (or instantiated) with actual values in the loop events.

plan-name	DBP$_1$ (ascending enumeration)
antecedent	
control-variables	*var#*
condition	*var# R# exp#*
enumeration	*var# := SUCC(var#)*
initialization	*var# := var?#*
firing-condition	(*R#* is relational operator that equals ≤ or <) *and* (*var#* is of a discrete ordinal type) *and* (Noncomposite or general loop condition)
consequent	
precondition	*PRED(var?#) R# exp#*
invariant	*var?# ≤ var# R# SUCC(exp#)*
postcondition	*var# = SUCC(SHIFT(exp#))*
sequence	*var?# .. PRED(var#)*
final-sequence	*var?# .. SHIFT(exp#)*
inner-addition	*var?# ≤ var# R# exp#*
where,	
i .. j	Sequence of integers from *i* up to *j inclusive*.
SUCC (x)	The successor of *x*.
PRED (x)	The predecessor of *x*.
SHIFT	The identity function if *R#* equals ≤. Equals *PRED* otherwise.

Fig. 6. A determinate basic plan.

plan-name	SAP$_5$ (find minimum)
antecedent	
control-variables	*v#*
body	*a#[exp#] R# lhs# ⇒ lhs# := a#[exp#]*
initialization	*lhs := lhs?#*
firing-condition	(*R#* equals ≤ or <)
consequent	
precondition	*true*
invariant	$lhs = MIN(\{lhs?\#\} \cup \{a\#[exp\# \overset{v\#}{\text{sequence}}]\})$
postcondition	$lhs = MIN(\{lhs?\#\} \cup \{a\#[exp\# \overset{v\#}{\text{final-sequence}}]\})$
inner-addition	Same as invariant.
where,	
MIN(s)	The minimum of the set *s*.

Fig. 7. A simple augmentation plan.

The plan DBP$_1$ (Fig. 6) represents an enumeration construct that goes over a sequence of values of a discrete ordinal type in an ascending order with a unit step. In the case where the loop has a composite condition, the **sequence**, **final-sequence** and **postcondition** of this plan are written in a more general form that enables deducing the corresponding **sequence**, **final-sequence** and **postcondition** of the loop from the multiple BEs it contains. The plan SAP$_5$ (Fig. 7) searches for the minimum of a segment of the array *a#* and stores it in the variable *lhs#*.

The knowledge base in a specific application domain should be created by an expert in both formal specifications and this domain. The expert should analyze the commonly used events in this domain and create new plans or improve on already existing ones. In creating this knowledge base, its size should be controlled by increasing the utilization of the designed plans. The loop decomposition method was designed for this purpose; to reveal the common algorithmic constructs that can be incorporated in many different loops. The hypothesis is that this decomposition can have a positive effect on plan utilization and, hence, on the size of the knowledge base. Improvements on the structure and/or the knowledge represented in the plans can also make the plans applicable to a larger set of events.

Knowledge representation improvements, called *abstractions*, involve replacing some of the terms in a plan with more abstract ones that make the plan capable of analyzing more cases. For example, replacing the addition operator, +, in a plan that analyzes an accumulation by summation event by a more abstract one that denotes either addition or multiplication represents an abstraction of this plan. The new plan can analyze both accumulation by summation and accumulation by multiplication events.

Structural improvements to a plan modify the basic structure into a tree structure that allows the inclusion of several similar plans in one tree-structured plan. The root of the tree corresponds to an antecedent part that should match loop events. The edges of the tree correspond to local **firing-conditions** that control the selection of the appropriate consequents given in the remaining tree nodes. In other words, a tree-structured plan consists of a single antecedent and several consequents organized into one or more tree structures as shown in Fig. 8. The consequents are organized into one tree if the default consequent exists. Otherwise, they are organized into more than one tree (forest). In order to select a specific tree-structured plan, the event under consideration should satisfy the antecedent first. Within the plan, local **firing-conditions** guide the search for the suitable consequent. The more general the consequent, the closer it is to the root of its tree (e.g., consequent 1 of Fig. 8 is more general than consequent 1.1). **Firing-conditions** located at the same level are mutually exclusive. This means that only forward search is needed and no backtracking is required. When the event satisfies the antecedent, the search for the appropriate consequent starts at the appropriate root going down in the tree as far as possible. The edge between a parent and a child can only be taken if the local **firing-condition** associated with this edge is satisfied.

Tree-structured plans can be used to detect special cases and output loop specifications that are simple and concise. They can also be used to analyze similar events whose specifications vary depending on their environment (e.g., data types, control computation of the loop, ..., etc.).

For instance, the plan SAP$_5$ (Fig. 7) can be structurally improved as shown in Fig. 9. The antecedent is similar to that shown in Fig. 7 except for the firing condition. The antecedent **firing-condition** now allows *R#* to be matched with more relational operators. Three local **firing-conditions** and the consequents cover three different variations. Consequent 1, which is similar to the consequent of the basic plan in Fig. 7, is for finding the minimum. Consequent 1.1 further simplifies the resulting annotations based on special values of *lhs#* and the analysis information of the control variable *v#*. Consequent 2 is for finding the maximum.

Using the tree-structured plans can lead to a reduction in the size of the knowledge base since several plans can be

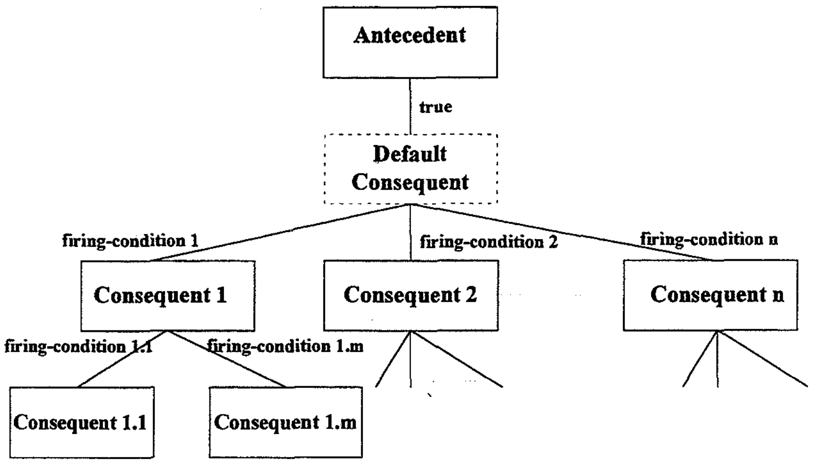


Fig. 8. The tree structure of a plan.

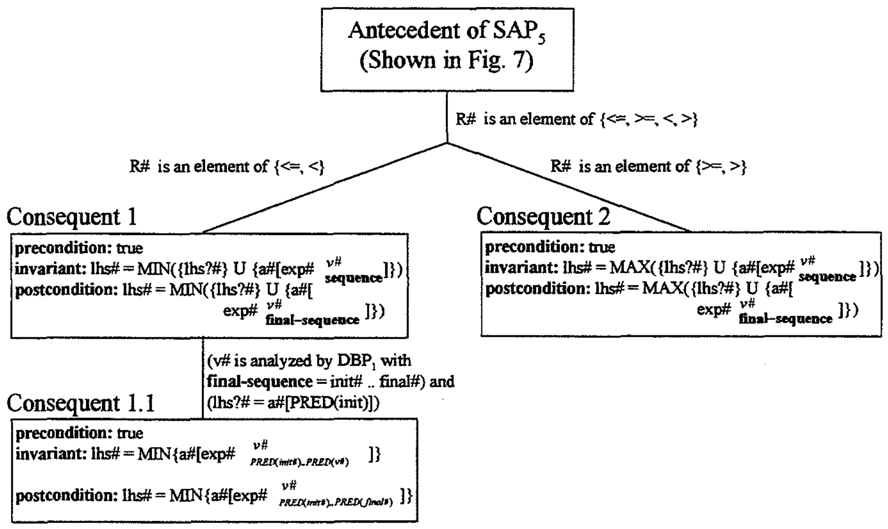

Antecedent of SAP5 (Shown in Fig. 7)
R# is an element of {<=, >=, <, >}
R# is an element of {<=, <}
R# is an element of {>=, >}
Consequent 1
precondition: true
invariant: lhs# = MIN({lhs?#} U {a#[exp# v# sequence]})
postcondition: lhs# = MIN({lhs?#} U {a#[exp# v# final-sequence]})
Consequent 2
precondition: true
invariant: lhs# = MAX({lhs?#} U {a#[exp# v# sequence]})
postcondition: lhs# = MAX({lhs?#} U {a#[exp# v# final-sequence]})
(v# is analyzed by DBP1 with final-sequence = init# .. final#) and (lhs?# = a#[PRED(init)])
Consequent 1.1
precondition: true
invariant: lhs# = MIN{a#[exp# v# PRED(init#)..PRED(v#)]}
postcondition: lhs# = MIN{a#[exp# v# PRED(init#)..PRED(final#)]}

Fig. 9. Structural improvement to the plan SAP_5.

combined together into a larger one having a unique antecedent. However, the identification of the proper consequent becomes more complicated due to the required tree search.

4.5 Analysis of the Events

The events are analyzed by trying to match them with the antecedents of the knowledge base plans. When an event satisfies the antecedent of a plan, the appropriate consequent of the matched plan is instantiated giving the contribution of the event to the loop specification. The precondition, invariant, and postcondition of the loop are formed by taking the conjunction of the corresponding parts of the event analysis results. When some event(s) do not match any library plans, the analysis only generates partial specifications of the loop.

To represent the results of matching loop events with plan antecedents, we define the *Analysis Knowledge* notation. The *Analysis Knowledge*, *AK*(*v*), of a variable *v* modified by a certain loop event consists of an n-tuple where n is dependent on the specific matched plan. The first term of the tuple is the name of the matched plan. The remaining (n – 1) terms are the results of matching the # terms with the actual values in the event.

The resulting AK tuples for the events of the loop given in Fig. 3 are shown in Fig. 10. The first line of Fig. 10 shows that the event that modifies the variable *j* is matched with the antecedent of plan DBP_1 (Fig. 6). The plan variables *var#* and *var?#* are matched with the event variables *j* and *j?*, respectively. The plan relational operator *R#* and expression *exp#* are matched with < and *num_of_rooms* + 1, respectively. The remaining two lines of Fig. 10 can be similarly interpreted. This AK information is used to instantiate the consequents of identified plans. The instantiation results are given in Fig. 11. In this figure, the event and plan responsible for the production of each predicate are shown to its left. The first two events are analyzed by the plans DBP_1 (Fig. 6) and SAP_5 (Fig. 7), respectively. The plan, SAP_{n1}, which analyzes the third event is not shown here because it is similar to the plan SAP_5. It searches for the location of the minimum instead of the minimum. Finally, the synthesized loop specifications are shown in Fig. 12.

AK(*j*) = (DBP_1, *var#*: *j*, *var?#*: *j?*, *R#*: <, *exp#*: *num_of_rooms* + 1)
AK(*min*) = (SAP_5, *v#*: *j*, *a#*: *capacity*, *exp#*: *j*, *lhs#*: *min*, *lhs?#*: *min?*)
AK(*index*) = (SAP_{n1}, *v#*: *j*, *a#*: *capacity*, *exp#*: *j*, *rhs#*: *min*, *rhs?#*: *min?*, *lhs#*: *index*, *lhs?#*: *index?*)

Fig. 10. The AK tuples for the events of the loop given in Fig. 3.

Precondition:

Event	Plan	Predicate
1	DBP_1	$j? - 1 < num\_of\_rooms + 1$
2	SAP_5	*true*
3	SAP_{n1}	$capacity[index?] = min?$

Invariant:

Event	Plan	Predicate
1	DBP_1	$j? \le j < num\_of\_rooms + 2$
2	SAP_5	$min = MIN(\{min?\} \cup \{capacity[j? .. j - 1]\})$
3	SAP_{n1}	$capacity[index] = min$

Postcondition:

Event	Plan	Predicate
1	DBP_1	$j = num\_of\_rooms + 1$
2	SAP_5	$min = MIN(\{min?\} \cup \{capacity[j? .. num\_of\_rooms]\})$
3	SAP_{n1}	$capacity[index] = min$

Fig. 11. The instantiations for the events of the loop given in Fig. 3.

Precondition:
$(j? - 1 < num\_of\_rooms + 1)$ *and*
$(capacity[index?] = min?)$

Invariant:
$(j? \le j < num\_of\_rooms + 2)$ *and*
$(min = MIN(\{min?\} \cup \{capacity[j? .. j - 1]\}))$ *and*
$(capacity[index] = min)$

Postcondition:
$(j = num\_of\_rooms + 1)$ *and*
$(min = MIN(\{min?\} \cup \{capacity[j? .. num\_of\_rooms]\}))$ *and*
$(capacity[index] = min)$

Fig. 12. The synthesized specifications of the loop given in Fig. 3.

5 Analysis of Nested Loops

To rigorously analyze nested loops using Hoare's axiomatic approach [19], [20], the following problems need to be solved:

1) **How to represent and utilize the analysis results of inner loops?** A technique for analyzing flat loops has been described in Section 4. Can the same basic technique be used for outer loops (loops containing other loops)? What modifications, if any, need to be performed on the basic analysis technique to utilize the results of analyzing inner loops in the analysis of outer loops?
2) **How to modify the resulting specifications to facilitate Hoare-style verification?** [19], [20] This problem can be further divided into two subproblems, which are explained using the nested construct shown in Fig. 13. In this nested construct, let I_i and I_o be the invariants of the inner and outer loops, respectively.
   a) Can the above invariants be used to satisfy Hoare verification conditions that connect the specifications of inner and outer loops in the nested construct? In other words, is it possible to prove the following rules:

$$(I_i \text{ and } \neg (B_i)) \{S_2\}\ I_o \qquad (1)$$

$$(I_o \text{ and } B_o) \{S_1\}\ I_i \qquad (2)$$

   b) If the above invariants use the notation *var?* to denote the initial value of a variable *var*, does this notation consistently refer to the value of *var* before the start of the outermost loop in the nested construct? If not, how can this inconsistency be removed?

```
while B₀ do begin ← Location L₁
    S₁
    while Bᵢ do begin ← Location L₂

    end;
    S₂
end;
```

Fig. 13. A nested structure of while loops.

To solve these problems, the analysis of nested loops is performed by recursively analyzing the innermost loops and replacing them with sequential constructs that represent their functional abstraction. The functional abstraction of an outer loop depends on the functional abstraction of the inner ones and not on the details of their implementation or structure.

Since this recursive analysis approach is performed bottom-up, complete knowledge of inner loop functions is available during the analysis of an outer loop. Thus, the invariant of an outer loop can be directly designed to satisfy the verification rules that are similar to rule (2) listed above. Despite the fact that inner loops are likely to contain references to variables defined in the outer loops, inner loops are analyzed in isolation of the outer ones enclosing them. As a result, a complete proof of nested constructs requires adapting the inner loop specifications to the context and initializations provided by the outer loop. More specifically, inner loop invariants and, consequently, postconditions might not be strong enough to satisfy the verification rules that are similar to rule (1). Some predicates might need to be added to the inner loop invariants and postconditions to enable the verification of such rules. The *context adaptation* phase derives these predicates and adds them to the inner loop specifications. Moreover, the consistency of using the notation *var?* to denote the initial value of a variable *var* is ensured using the *initialization adaptation* phase.

We start in Section 5.1 with some definitions that explain how we extract the initialization of a loop in a nested construct, whether it is the outermost loop or an inner one. Sections 5.2–5.4 present solutions to the two research problems mentioned above. Sections 5.2 and 5.3 offer a solution to the first research problem. Section 5.4 presents a partial solution to the second problem. In these sections, the descriptions of the analysis steps are interspersed with their application on the selection sorting example given in Fig. 14. In this example, a simple nested loop repeatedly scans an array segment searching for its minimum. It interchanges the minimum with the first element in the segment. It stops after the array *capacity*[1 .. *num_of_rooms*] has been sorted in ascending order. The inner loop of this example is the same one given in Fig. 3.

5.1 Definitions

In the following definitions, we limit the initialization of a loop to assignment statements. Conditional statements are not considered as initializations to reduce the complexity of the resulting loop specifications. Thus, resulting loop specifications are representative of the loop function without composing it with the function of preceding conditional statements.

```
i, j, index, min, num_of_rooms: integer;
capacity: array[1 .. max_rooms] of integer;
:
i := 1;
while i ≤ num_of_rooms – 1 do begin
    index := i;
    min := capacity[i];
    i := i + 1;
    j := i ;
    while j < num_of_rooms + 1 do begin
        if capacity[j] < min then begin
            index := j;
            min := capacity[j];
        end;
        j := j + 1
    end;
    capacity[index] := capacity[i – 1];
    capacity[i – 1] := min
end;
```

Fig. 14. Example of a nested loop.

The *initialization of a loop that is not enclosed by another loop* is assumed to be a set of assignment statements of the form *identifier* := *expression*, which are immediately placed before its start. These statements give initial values for identifiers that get modified within the loop body. If this assumption cannot be satisfied or, equivalently, the loop initialization is unavailable, the notation *v?* is used to denote the initial value of a variable *v* just before the start of the loop.

If we have two nested while loops, the *adaptation path* of the inner loop is a sequence of statements extracted from their control-flow graph representation. This sequence contains all the statements, simple or compound, that are completely located along the paths starting from the outer loop control predicate node and ending at the inner loop control predicate node. In this path, the relative order of the statements is kept unchanged.

The *initialization of an inner loop* in a nested construct is obtained by, first, symbolically executing its adaptation path to produce the net modification performed on each variable, if possible. Statements of the form *identifier* := *expression* are, then, extracted from the symbolic execution result. Statements are extracted if they satisfy the following two conditions:

1) The *identifier* is one of the variables modified within the inner loop body.
2) The *expression* does not reference any of the variables modified along the adaptation path.

If the initialization of a variable *v* that gets modified within the loop body is not given by the extracted statements, the notation *v?* is used to denote its initial value just before the start of the loop.

The first condition, in the above definition, ensures that the initialization statements are utilized by the inner loop events. The second condition ensures that the values of *identifier* and *expression*, just before the start of the inner loop, are equal. For example, if the adaptation path is $i := i + 1; j := i$, then its symbolic execution gives the concurrent assignment $i, j := i + 1, i + 1$. Taking $j := i + 1$ as an initialization statement is not allowed because the values of j and $i + 1$, just before the start of the loop, are not equal (the values of j and i are equal). The second condition also prevents using statements of the form, say, $x := x + 1$ as initializations.

To extract the initialization of the inner loop given in Fig. 14, we use the above definitions. First, we need to symbolically execute the adaptation path of the inner loop. Since there is only one path between the start of the outer loop and the start of the inner one, the adaptation path includes all the statements completely located on this path. The adaptation path is: *index* := *i*; *min* := *capacity*[*i*]; $i := i + 1$; $j := i$.

The symbolic execution of the adaptation path yields the concurrent assignment: *index*, *min*, *i*, *j* := *i*, *capacity*[*i*], $i + 1$, $i + 1$.

Then, we need to extract initialization statements of the form *identifier* := *expression* from the symbolic execution result. The variables modified within the inner loop body are: *index*, *min*, and *j*. Thus, the statements that satisfy the first condition of the above definition are: *index* := *i*, *min* := *capacity*[*i*], and $j := i + 1$. However, these statements are not valid initialization statements because their right hand sides reference the variable *i* that gets modified along the adaptation path. In other words, these statements do not satisfy the second condition of the above definition. As a result, the initialization statements of the inner loop are written by using the notation *v?* to denote the initial value of a variable *v* as follows: *index* := *index?*, *min* := *min?*, and *j* := *j?*.

5.2 Analysis of Inner Loops and Representation of Their Analysis Results

The analysis of inner loops is performed using the same four phases described, in Section 4, for flat loops. To analyze an outer loop in a nested construct, the analysis results of its inner loops must be represented in a way that reveals the functionality of the inner loops and the flow of data into and out of the inner loops. The data flow information is needed to perform the decomposition of the outer loop body.

Though the resulting AK tuples or predicate logic annotations can be used to represent the inner loop analysis results, they either include too much detail or the deduction of the required information is difficult, respectively. Hence, the solution is to use a formalism that is similar to function calls; the name encapsulates the functionality while the arguments indicate the data flow information. The formalism used for this purpose is called an *Abstraction Class* (*AC*).

An AC is a knowledge base object that transforms the detailed analysis results of an inner loop to a more abstract representation that facilitates the analysis of outer loops. It groups AK tuples based on some common functionality and ignores the unnecessary implementation specific details. The common functionality is documented to explain the purpose of designing the AC and to enhance its modifiability. Furthermore, the definition of an AC offers an abstract representation of its elements that specifies the data flow information. This abstract representation facilitates the mechanical manipulation of ACs. An Abstraction Class (AC) consists of three parts:

1) The **elements** part consists of generic AK tuples that are separated by the symbol '|'.
2) The **common-function** describes the functionality that the elements of this class share by using common instantiated **final-sequence, postcondition,** or **invariant** parts of the matched plans.
3) The *representation* is a unique abstract representation that gives the class name, followed by the following arguments (separated by semicolons and enclosed between two parentheses): the list of expressions responsible for the data flow into this AC, the list of variables defined by the AC, the control variables of the loop under consideration, and a unique number identifying the loop being analyzed.

The representation part contains the class name that is an arbitrary and unique name. It also contains the arguments responsible for the data flow into and out of the AC so that they can be used during the data flow analysis. The control variables and unique number of the loop are used in the design of some plan consequents. To simplify the presentation, the last two arguments are only listed when needed.

The AK of some variable belongs to a specific AC if it matches an AK tuple existing in the **elements** part. The symbol '*' is used to denote irrelevant information. An expression, *exp*, enclosed between two brackets in the **elements** part implies that the expression should be matched with the corresponding instantiated element of the actual AK to deduce the value of the variables defined in it. Some of these variables are utilized in forming the AC arguments.

The AK of the variable j analyzed in the inner loop of Fig. 14 has the following form:

$$AK(j) = (DBP_1, var\#: j, var?\#: j?, R\#: <, exp\#: num\_of\_rooms + 1).$$

This AK belongs to the AC in Fig. 15 because it matches the first AK tuple of the **elements** part. If we had implemented this loop with the condition $j \le num\_of\_rooms$ instead of $j < num\_of\_rooms + 1$, it would have belonged to the same AC. This is because it matches the second AK tuple of the **elements** part. These two different implementations belong to the same AC because they have the common function of going over the integer sequence $j? .. num\_of\_rooms$ in an ascending order.

elements	$(DBP_1, var\#: [v], var?\#: *, R\#: \le, exp\#: [final])$
	|
	$(DBP_1, var\#: [v], var?\#: *, R\#: <, exp\#: [SUCC(final)])$
common-function	The instantiated **final-sequence** of the plan is: $v? .. final$
representation	$AC_{DB_1}(v, final; v)$

Fig. 15. An abstraction class for ascending enumeration.

Using similar analysis, the AK of the variable *min* is found to belong to AC_{SA_5} (Fig. 16). The AK of the variable *index* belongs to $AC_{SA_{n1}}$. Because $AC_{SA_{n1}}$ is similar to AC_{SA_5}, it is not shown here. AC_{SA_5} includes the AK tuples that have the common function of finding the minimum of an array segment irrespective of the enumeration direction (ascending or descending) and the index of the array element being checked (v, $PRED(v)$, or $SUCC(v)$). It should be mentioned that AC_{DB_2} is similar to AC_{DB_1} but for descending enumeration. The ACs of the variables modified in the inner loop of Fig. 14 are, thus, as follows:

$AK(j) \in AC_{DB_1}(j, num\_of\_rooms; j)$

$AK(min) \in AC_{SA_5}(capacity, j, num\_of\_rooms, min; min)$

$AK(index) \in AC_{SA_{n1}}(capacity, j, num\_of\_rooms, min, index; index)$

elements	$(SAP_5, v\#: [v], a\#: [a], exp\#: [PRED(v)], lhs\#: [lhs], lhs?\#: *)$, where $AK(v) \in AC_{DB_2}([SUCC(final)], [SUCC(init)])$
	|
	$(SAP_5, v\#: [v], a\#: [a], exp\#: [v], lhs\#: [lhs], lhs?\#: *)$, where $AK(v) \in AC_{DB_2}([final], [init])$ *or* $AK(v) \in AC_{DB_1}([init], [final])$
	|
	$(SAP_5, v\#: [v], a\#: [a], exp\#: [SUCC(v)], lhs\#: [lhs], lhs?\#: *)$, where $AK(v) \in AC_{DB_1}([PRED(init)], [PRED(final)])$
common-function	The instantiated **postcondition** of the plan is: $lhs = MIN(\{lhs?\} \cup \{a[init .. final]\})$
representation	$AC_{SA_5}(a, init, final, lhs; lhs)$

Fig. 16. An abstraction class for finding the minimum.

After analyzing an inner loop, we replace it with the concurrent assignment that assigns to the list of variables modified by it the result of their analysis. If the AK of a variable belongs to a predefined AC, its abstract representation, as deduced from the identified AC, is assigned to it. If the AK of the variable, *var*, does not belong to a predefined AC, we assign the form *UAC(ak-list; var)* to it, where *UAC* stands for Unknown AC and *ak-list* is a list representing the AK data. The *ak-list* and *var* are used, during automatic analysis, to provide information on the unanalyzed parts of the loop.

Conceptually, the described replacement is equivalent to replacing the inner loop with a set of function calls that assign to each variable changed in the inner loop the desired value. This replacement preserves the control flow dependencies because the concurrent assignment is placed at the same relative location within the outer loop body. It also preserves the data flow dependencies between the variables because the ACs clearly state what variables are responsible for the data flow into and out of it.

Replacing the inner loops given in Fig. 14 with the described concurrent assignment gives the following modified outer loop:

```
$i := 1;$
while $i \le num\_of\_rooms - 1$ do begin
    $index := i;$
    $min := capacity[i];$
    $i := i + 1;$
```

```
        $j := i;$
        $j, min, index := AC_{DB_1}(j, num\_of\_rooms; j),$
        $AC_{SA_5}(capacity, j, num\_of\_rooms, min; min),$
        $AC_{SA_{n1}}(capacity, j, num\_of\_rooms, min, index; index);$
        $capacity[index] := capacity[i - 1];$
        $capacity[i - 1] := min$
    end;
```

5.3 Analysis of Outer Loops

After modifying an outer loop body, we analyze it using the previously described method for analyzing flat loops (Section 4), as if it does not contain any other loops inside it. This can be done since the inner loop(s) have been replaced by ordinary sequential constructs. The only difference, in this case, is that high-level plans are used in addition to the usual (low-level) ones. High-level plans are those that utilize ACs.

Adding another classification level, based on whether the plan is low-level or high-level, to the four plan categories shown in Fig. 5, we get eight plan categories. These new plan categories are shown in Fig. 17. The advantage of this plan classification scheme is that it indexes plans for rapid access given the loop and event types.

The strength of this approach for analyzing nested constructs is that it can scale up to handle more than two nested loops. This is because the inner loops can be recursively analyzed and replaced by sequential constructs. Any outer loop can thus be analyzed by using the high-level plans in addition to the low-level ones. If we are unable to analyze one of the inner loops, the analysis of the outer loop proceeds as far as possible. That is, we can only analyze outer loop events that are independent of the unanalyzed inner loop events. In such cases, partial analysis results are produced. An outline of the application of the analysis steps on the modified outer loop of Fig. 14 is given below.

The ordered events of the modified outer loop are as follows:

1) BE (order 1)
   - condition: $i \leq num\_of\_rooms - 1$
   - enumeration: $i := i + 1$
   - initialization: $i := 1$
2) AE (order 2)
   - *body:* $capacity[i]$, $capacity[AC_{SA_{n1}}(capacity, i + 1, num\_of\_rooms, capacity[i], i; index)] := AC_{SA_5}(capacity, i + 1, num\_of\_rooms, capacity[i]; min), capacity[i]$
   - initialization: $capacity := capacity?$
3) AE (order 2)
   - body: $j := AC_{DB_1}(i + 1, num\_of\_rooms; j);$
   - initialization: $j := j?$
4) AE (order 3)
   - body: $min :(AC_{SA_5}(capacity, i + 1, num\_of\_rooms, capacity[i]; min)$
   - initialization: min := min?
5) AE (order 3)
   - body: $index := (AC_{SA_{n1}}\ capacity, i + 1, num\_of\_rooms, capacity[i], i; index)$
   - initialization: index := index?

The first event is matched with the antecedent of the plan DBP$_1$ (Fig. 6). The second event is matched with a Simple High-level AP (SHAP) that represents the selection sorting concept. Because the variables j, *min* and *index* do not explicitly contribute to the outer loop specifications, the last three events are matched with SHAPs that produce *true* predicates. These variables implicitly affect the outer loop specifications through their abstraction classes that get used by the second event. For details concerning the plans used and the event analysis results, refer to [1]. The final synthesized analysis results are given below. The first event is responsible for the production of the first conjugate of each predicate. The second event is responsible for the production of the rest of the specifications.

Precondition:

$(0 \leq num\_of\_rooms - 1)$

Invariant:

$(1 \leq i \leq num\_of\_rooms)$ *and*
(FORALL ind: $1 \leq ind \leq i - 1$: $capacity[ind] =$
$MIN(\{capacity[ind .. num\_of\_rooms]\})$ *and*
PERM(capacity, capacity?)

Postcondition:

$(i = num\_of\_rooms)$ *and*
(FORALL ind: $1 \leq ind \leq num\_of\_rooms - 1$: $capacity[ind] =$
$MIN(\{capacity[ind .. num\_of\_rooms]\})$ *and*
PERM(capacity, capacity?)

The resulting predicate logic annotations produced for the inner and outer loops can be used to assist the understanding of the nested construct. An understanding of the sorting algorithm can be formed using the predicate

$(min = MIN(\{min?\} \cup \{capacity[j? .. num\_of\_rooms]\}))$ *and*
$(capacity[index] = min)$

of the inner loop postcondition and the predicate

(FORALL *ind:* $1 \leq ind \leq num\_of\_rooms - 1$: $capacity[ind] =$
$MIN(\{capacity[ind .. num\_of\_rooms]\})$ *and*
PERM(capacity, capacity?)

of the outer loop postcondition. However, such specifications cannot be proved using Hoare-style [19] axiomatic correctness. To be able to prove the outer loop invariant, the predicate

$(1 \leq i - 1 \leq num\_of\_rooms - 1)$ *and*
(FORALL ind: $1 \leq ind \leq i - 2$: $capacity[ind] =$
$MIN(\{capacity[ind .. num\_of\_rooms]\})$ *and*
PERM(capacity, capacity?)

should be added to the invariant of the inner loop. This predicate provides information about the context of the inner loop, which is needed to prove rule (1) of the second research problem that is given at the beginning of this section. In addition, *j?, min?,* and *index?* in the inner loop specifications should be replaced with *i*, *capacity*[*i* – 1], and *i* – 1, respectively.

5.4 Adaptation of Inner Loop Specifications

To be able to prove that the implementations of nested constructs satisfy their specifications, the specifications of inner

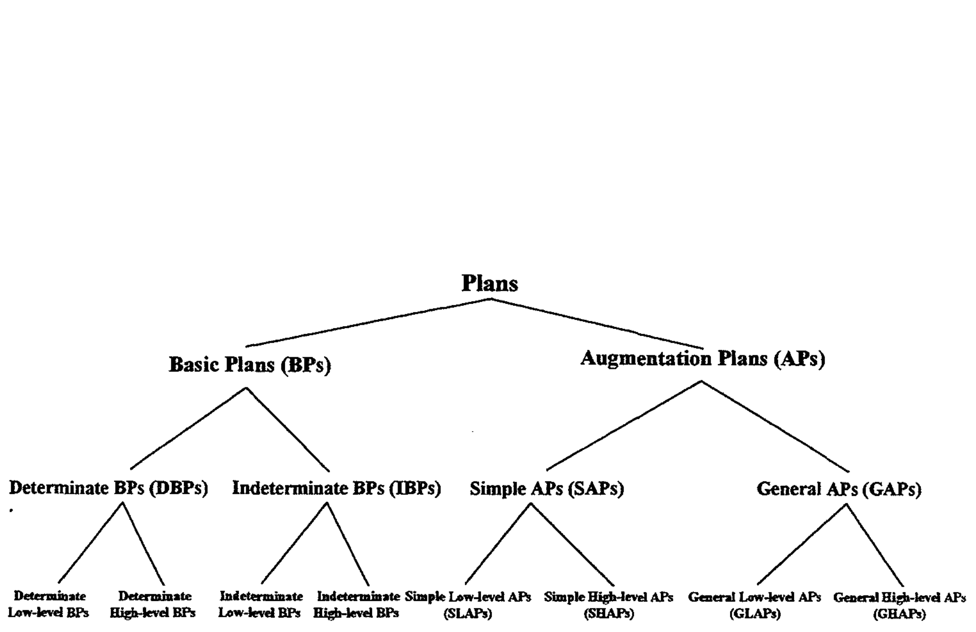

Plans
Basic Plans (BPs)
Augmentation Plans (APs)
Determinate BPs (DBPs)
Indeterminate BPs (IBPs)
Simple APs (SAPs)
General APs (GAPs)
Determinate Low-level BPs
Determinate High-level BPs
Indeterminate Low-level BPs
Indeterminate High-level BPs
Simple Low-level APs (SLAPs)
Simple High-level APs (SHAPs)
General Low-level APs (GLAPs)
General High-level APs (GHAPs)

Fig. 17. New plan categories.

loops need to be strengthened to include information about the context of outer loops enclosing them. To ensure that the notation *var?* is consistently used to denote the initial value of a variable *var* before the start of the outermost loop, variables of the form *var?* in specifications of inner loops need to be replaced by their actual values. These tasks are performed in the context and initialization adaptation phases. The remainder of this subsection describes how to perform these adaptations. In this description, it is assumed that the adaptation path of an inner loop, that is defined in Section 5.1, only includes assignment and conditional statements. The cases in which this assumption is not satisfied are discussed in Section 6.

Context Adaptation. While analyzing the outer loop, we have complete knowledge of an inner loop function. Thus, this is the best time to generate a context related predicate *inner-addition*, which strengthens inner loop invariants. By studying the differences between the current outer loop **invariant** part and the generated inner loop **invariant** part, we design and add an **inner-addition** field to the consequents of the knowledge base plans. This field provides any predicates that should be added to the invariants of inner loops to enable the verification of rules similar to: $(I_i$ *and* $\neg B_i)$ $\{S_2\}$ I_0. After analyzing an outer loop, the instantiated **inner-addition** fields are synthesized, by conjunction, to form the predicate *inner-addition*.

For instance, assume that the plan DBP_1 (Fig. 6) is used to analyze an ascending enumeration construct of an outer loop having the control variable *var#*. While analyzing the inner loop in isolation, no knowledge exists about *var#* being an outer loop control variable that scans a specific sequence of values. Hence, the **inner-addition** filed of DBP_1 should provide this information in the form of the predicate: *var?#* $\leq$ *var# R# exp#*.

Analyzing the BE of the outer loop given in Fig. 14 using DPB_1 yields the following instantiated **inner-addition**:

$(1 \leq i \leq num\_of\_rooms - 1)$

Similarly, when the inner loop of this sorting example is analyzed in isolation, its invariant does not include any information about the sorted segment of the array *capacity*. Thus, the **inner-addition** part of the outer loop selection sorting plan should provide the following predicate:

(FORALL ind: $1 \leq ind \leq i - 1$: *capacity*[*ind*] =
MIN({*capacity*[*ind* .. *num_of_rooms*]}) *and*
PERM(*capacity, capacity?*)

By taking the conjunction of the two instantiated **inner-addition** parts, the *inner-addition* of the example given in Fig. 14 is:

$(1 \leq i \leq num\_of\_rooms - 1)$ *and*
(FORALL ind: $1 \leq ind \leq i - 1$: *capacity*[*ind*] =
MIN({*capacity*[*ind*.. *num_of_rooms*]}) *and*
PERM(*capacity, capacity?*)

However, the synthesized *inner-addition* is designed to be correct at a fixed reference point which is location L_1 (see Fig. 13). This is because during the design of the library plans there is no knowledge, a priori, of the statements physically located along the adaptation path. The effect of the statements along the adaptation path should be taken into consideration to get the corresponding correct predicate, *inner-addition*$_2$, at location L_2.

By comparing the *inner-addition* produced for the loop given in Fig. 14 to the predicate that should be added to the inner loop specifications (given at the end of Section 5.3), it is clear that they are not exactly the same. This is because the effect of the statements along the adaptation path has not been taken into consideration yet.

The context adaptation uses *inner-addition* and the adaptation path to find *inner-addition*$_2$.The predicate *inner-addition*$_2$ is deduced by reversing the effect of the statements along the adaptation path on the variables in *inner-addition* [14]. For example, if the adaptation path changes i to $i - 1$, then all the free occurrences of i in *inner-addition* are replaced by $i + 1$ to generate *inner-addition*$_2$.

This reversing (or inversion) is performed, mechanically, by introducing a set of auxiliary variables that replace all the free occurrences, in *inner-addition*, of the variables modified along the adaptation path. Conceptually, the auxiliary variables denote the state of the corresponding original ones at location L_1.

For the example shown in Fig. 14, the auxiliary variable i_1 replaces the variable i in *inner-addition*. Since the variable *capacity* is not modified along the adaptation path, no corresponding auxiliary variable is introduced for it. The modified *inner-addition*, which is called *inner-addition*$_1$, has the form:

$(1 \leq i_1 \leq num\_of\_rooms - 1)$ and
(FORALL ind: $1 \leq ind \leq i_1 - 1$: *capacity*[*ind*] =
MIN({*capacity*[*ind*.. *num_of_rooms*]}) *and*
PERM(*capacity, capacity?*)

We then form a predicate, *aux-values*, that represents the relation between the auxiliary variables, used at location L_1, and the corresponding original ones, used at location L_2. This predicate is formed using the symbolic execution result of the adaptation path. First, the introduced auxiliary variables should replace their corresponding actual ones that are responsible for the data flow into the symbolic execution result. The predicate equivalent of the statements that modify the original variables are, then, generated and conjunctioned together.

The predicate equivalent of an assignment statement is produced by replacing the assignment sign with an equal sign. Conditional assignments can be converted into assignment statements of the form: *var* := *choice* (*condition1, value1, condition2, value2, ..., etc.*), where the right hand side is equal to *value1* if *condition1* is *true*, *value2* if *condition2* is *true*, and so on. The resulting assignment statement is converted into a predicate as described before.

In the example shown in Fig. 14, the symbolic execution result of the adaptation path is:

index, min, i, j := *i, capacity*[*i*], *i* +1, *i* +1.

The context adaptation replaces *i* by i_1 in the right hand side to produce:

index, min, i, j := i_1, *capacity*[i_1], i_1+1, i_1+1.

The statement that modifies the original variable *i* is $i := i_1 + 1$. The predicate equivalent of this statement, $i = i_1 + 1$, is the predicate *aux-values*.

The required correct predicate *inner-addition*$_2$ is the conjunction of *aux-values* and *inner-addition*$_1$. The predicate *inner-addition*$_2$, which is actually added to the inner loop invariant, has the form:

($1 \le i_1 \le$ *num_of_rooms* – 1) *and*
(*FORALL ind*: $1 \le ind \le i_1 - 1$: *capacity*[*ind*] =
MIN({*capacity*[*ind* .. *num_of_rooms*]}) *and*
PERM(*capacity, capacity?*) *and*
$i = i_1 + 1$

Initialization Adaptation. The initialization adaptation replaces each variable of the form *var?*, in an inner loop specification, with its value as deduced from its adaptation path and the invariant of the enclosing loop. After this replacement, the notation *var?* is reserved for referring to the state of a variable *var* before the start of the outermost loop. The notation var_{outer} is used to refer the value of *var* as deduced from the invariant of the loop enclosing it.

The initial value of a variable *var* is extracted from the symbolic execution result of the adaptation path. If the symbolic execution result assigns the value var_{adapt} to *var*, then var_{adapt} is the needed initial value. However, var_{adapt} needs to be modified so that it is expressed in terms of the program state at location L_2 and not location L_1. This modification is performed in the same way we modified *inner-addition*. That is, var_{adapt} is modified to $\mathrm{var}_{adapt}{}^{original\ variables}_{auxilliary\ variables}$. However, if *var* itself occurs in var_{adapt}, it should, first, be replaced by var_{outer} to avoid a circular definition of the initial value of *var*. In short, every *var?* in the inner loop specification is replaced by $((var_{adapt})^{var}_{var_{outer}})^{original\ variables}_{auxiliary\ variables})$.

For instance, the symbolic execution result of the adaptation path of the example shown in Fig. 14 is:

index, min, i, j := *i, capacity*[*i*], *i* + 1, *i* + 1.

The variable *j?* in the inner loop specification is replaced by $((i+1)^{j}_{j_{outer}})^{i}_{i_1})$, where $i = i_1 + 1$. So, *j?* is effectively replaced by *i*. Similar analysis shows that *min?* and *index?* should be replaced by *capacity*[*i* – 1] and *i* – 1, respectively.

In summary, the specification of the inner loop shown in Fig. 14 is adapted by adding the predicate *inner-addition*$_2$ that is simplified to:

($1 \le i - 1 \le$ *num_of_room* – 1) and
(*FORALL ind*: $1 \le ind \le i - 2$: *capacity*[*ind*] =
MIN({*capacity*[*ind* .. *num_of_rooms*]}) *and*
PERM(*capacity, capacity?*)

The initial variables *j?*, *min?*, and *index?* are replaced with *i*, *capacity*[*i* – 1], and *i* – 1, respectively. These adaptation results are exactly the ones described at the end of Section 5.3.

6 Discussion

In this paper, a knowledge-based program understanding approach has been described. The resulting predicate logic annotations are unambiguous and have a sound mathematical basis that allows correctness conditions to be stated and verified, if desired. The analysis approach does not rely on real-time user-supplied information and can analyze nonadjacent loop parts.

However, there are limitations to this approach. These are:

- Practical limitations related to the effort and ingenuity needed to design the plans.
- Theoretical limitation related to the generation of concise postconditions for general loops.
- Theoretical limitation related to the adaptation of inner loop specifications in some nested loops.

The practical limits stem from the plan designers inability to formally analyze complicated loops and find their invariants despite the fact that these invariants exist theoretically. The resulting specifications are as accurate, readable, and correct as the plans are. That is why the tasks of designing plans and managing the knowledge base, for a specific application domain of interest, should be performed by an expert in both the desired domain and formal specifications.

The first theoretical limit was discussed in Section 4.4. In case of general loops, we cannot produce loop postconditions as intelligently and concisely as for simple loops because it was not possible to include postcondition parts in the plans designed for analyzing individual events of general loops. Thus, additional simplifications of the postconditions that transforms them into more readable ones might be required.

The second theoretical limit occurs in nested structures having the following characteristic: the adaptation path of an inner loop contains statements other than assignment and conditional statements (e.g., loops or procedure calls).

The context and initialization adaptations cannot, in general, be performed for such cases. The reason for this limitation is that the context and initialization adaptations are based on the fact that assignment statements and, to a lesser extent, conditional statements can be easily inverted in a mechanical way [14]. However, if there are loops, procedure calls, or function calls, this inversion cannot be performed mechanically. Performing such an inversion is equivalent to finding the specifications of arbitrary program fragments containing nonsequential constructs and representing their analysis results in terms of equational specifications that can be easily inverted. The presented approach can perform symbolic execution of sequential constructs and can produce first order predicate logic specifications of loops. However, these two different capabilities have not been integrated to produce invertible equational specifications of arbitrary program fragments.

The second theoretical limitation only affects the ability to prove that the loop implementations satisfy the resulting specifications. It does not affect the ability to assist the understanding of nested loops. This is because the approach still produces meaningful specifications of the whole construct. For instance, it has been shown that an understanding of the sorting algorithm in our example was possible before performing the adaptation steps. In addition, the context and initialization adaptation can be performed in some special cases. One special case occurs when the variables used in the *inner-addition* do not get modified along the adaptation path. Another special case happens when variables, whose initial values need to be replaced, do not get modified along the adaptation path. In the first case, the context adaptation does not need to modify the predicate *inner-addition*. In the second case, the initialization adaptation directly replaces *var?*, if any, with its value as deduced from the outer loop invariant. A third special case occurs when the loops located on the adaptation path are simple ones. In this case, the adaptation of an inner loop specification can be performed using postcondition parts of its preceding loop, which are in equational form, instead of its outer loop invariant. It should be noted that the first theoretical limit partly affects the second one. If we were able to include equational postconditions in the plans that analyze general loops, they could have been used in the adaptation steps.

7 Case Study

The program chosen as a case study of our loop analysis process deals with scheduling a set of university courses. It has about 1,400 lines of executable Pascal source code. There are a total of 39 modules (functions and procedures). A complete listing of the requirements, specifications, design, and code documents is given elsewhere [22]. In this program, there are 77 loops that cover all the classes in our taxonomy. Many of these loops involve sorting, searching, and scheduling algorithms. Because of the interactive nature of this program, it contains several other loops that perform input error detection as well as warning and error messages generation.

7.1 Objectives

The main objective of this case study was to test our analysis approach and to assess its effectiveness when applied to a fixed set of loops in a real and pre-existing program of some practical value. To this effect, we collected the data needed for performing the following validations and characterizations:

- Test the hypothesis that a loop complexity dimensions are valid indicators of its amenability to analysis.
- Test the hypothesis that the loop decomposition and plan design methods of our approach can make the plans applicable in many different loops and, hence, increase their utilization.
- Characterize the practical limits of the analysis approach.

7.2 Method

The case study was performed, manually, prior to the implementation of the prototype tool. Case study results are, thus, not affected by the limits of the implementation that are given at the end of Section 8. The set of 77 loops in the described program were extracted along with their initializations. This set included 25 *for* loops, which were transformed to their equivalent while loops. The loops analyzed had the usual programming language features such as pointers, procedure and function calls, and nested loops. To design, and prove, assertions of loops containing pointer variables, the notation and techniques of Luckham and Suzuki [30] were used. Procedures that were called from within loops had to be formally analyzed, using Hoare techniques [20], to obtain rigorous descriptions of their functionality and data flowing into and out of them.

During the study, every loop under consideration was first decomposed into its basic and augmentation events. Then, every event was analyzed in order to design a plan suitable for it. If no plan was available in the knowledge base to match the event under consideration, or a similar event, a new plan was developed with designer defined, candidate specifications. The plan was then modified and tailored to give correct specifications by trying to prove the loop invariant using Hoare techniques [19]. If a plan that matched a similar event, but not the exact one under consideration, existed in the knowledge base, improvements on the structure and/or the knowledge represented in the existing plan were considered.

As the number of analyzed loops increased, the experience gained led to the evolution of the knowledge base. The monitored usage of the knowledge base served to improve some of the plans in terms of their structure, knowledge representation, number, and naming conventions. As a result, the knowledge base was more suitable for the domain under consideration.

The designed plans (BPs and APs) were not only limited to those which provided functional specifications but also included plans that discarded unnecessary detail about temporary variables and plans that provided warning and error messages. It should also be mentioned that the resulting formal specifications were not formulated in terms of concepts specific to the application domain. Even though such domain independent specifications can increase the chance of reusing the plans, they sometimes have the disadvantage of being more difficult to read [7].

We decided not to specifically design plans for the analysis of 12 loops (15.6%) in the case study. The unique and complex nature of these loops suggested that the effort needed to design their analysis plans highly outweighs advantages that could be gained from their expected extent of utilization in this specific application domain. That is, the partial analysis of the 12 loops in this case study is mainly attributed to the practical limitation discussed in the previous section. These 12 loops were arbitrarily numbered from p1 through p12. They were analyzed using the available set of plans to determine whether useful partial specifications could be obtained.

7.3 Results and Analysis

Tables 2 and 3 give the data collected to test the hypothesis that a loop complexity dimensions are indicators of its amenability to analysis. Table 2 gives the number of loops completely analyzed in each class defined by our taxonomy. Along the first dimension, the available and analyzed numbers of Simple (S) and General (G) loops are given. In the second dimension, the available and analyzed numbers of loops with Noncomposite (N) and Composite (C) conditions are given. Finally, the available and analyzed numbers of Flat (F) and Nested (N) loops are given along the third dimension. Using the three classification dimensions, any loop must belong to one of the 8 (2^3) equivalence classes given in Table 3. In this table, the available and analyzed numbers of loops in each of these equivalence classes are shown. The table also gives the total numbers of events and their averages for the analyzed loops in each class.

The results given in Tables 2 and 3 support the hypothesis that the classification taxonomy helps in predicting a loop amenability to analysis. Table 2 shows that the presumably more complex classes always have lower percentages of completely analyzed loops than the presumably less complex ones. For example, the percentages of completely analyzed flat and nested loops are 98 and 54, respectively. All flat loops were completely analyzed except for one loop (loop p10) that contained a call to a procedure with a partially analyzed nested loop (loop p9). This percentage variation is even more notable when further investigated along the five available equivalence classes of Table 3. Percentages range from 100% for SNF and SCF to 22% for GCN. The numbers of events in the analyzed loops further support the interpretation that the classification of a loop is an indicator of its complexity and, correspondingly, its amenability to analysis. For example, while SNF loops (Flat) have an average of 2.4 events/loop, SNN loops (Nested) have an average of 5.6 events/loop.

Table 4 summarizes the data collected to examine the plan utilization issue. It shows the number of events analyzed by each of the designed plans. It also shows the total utilization of the plans in each of the six available categories. Since only one high-level basic plan (IBP_6) was designed, we do not differentiate between low and high-level BPs. During the iterative process of designing the plans, some of them underwent abstractions and others were structurally improved into tree structures. The * or + superscript is used to denote those plans that underwent abstractions or structural improvements, respectively. For example, plan DBP_1 was used 45 times and had a tree-structured design.

The 48 plans designed were utilized in analyzing a total of 235 events. A closer examination of the results in Table 4 shows that a set of 27 plans (56%) analyzed 214 events (91%). The remaining 21 plans were only used once. These results indicate that if we focus on a specific application domain, there is bound to be a kernel of events that can be captured by a relatively reasonable number of plans. On the other hand, there will also be plans that, as in our study, may be used just once. The emphasis should be on the design of the plans that cover the kernel.

TABLE 2
NUMBER OF COMPLETELY ANALYZED LOOPS ALONG THE THREE DIMENSIONS

Analysis statistics	Dimension					
	1		2		3	
	Simple loop	General loop	Noncomposite condition	Composite condition	Flat body	Nested body
Available number	52	25	46	31	53	24
Number analyzed	48	17	42	23	52	13
Percentage analyzed	92	68	91	74	98	54

TABLE 3
NUMBER OF COMPLETELY ANALYZED LOOPS IN THE AVAILABLE CLASSES

Analysis statistics	Equivalence class							
	SNF	SCF	SNN	SCN	GNF	GCF	GNN	GCN
Available number	31	6	15	0	0	16	0	9
Number analyzed	31	6	11	—	—	15	—	2
Percentage analyzed	100	100	73	—	—	94	—	22
Number of events	75	18	61	—	—	51	—	8
Average events/loop	2.4	3	5.6	—	—	3.4	—	4

TABLE 4
UTILIZATION OF THE DESIGNED PLANS

Name (subscript)	Plan category					
	DBP	IBP	SLAP	GLAP	SHAP	GHAP
1	45^{*}	4	23^{*+}	4	3^{+}	3
2	6^{*}	15^{*}	19^{+}	13^{*+}	13^{*+}	2
3	8	1	3^{*}	1	1	—
4	9^{*}	2	1	2	1	—
5	1	2	1	1	1	—
6	—	2	1	1	1	—
7	—	—	1	1	2	—
8	—	—	20	1	2	—
9	—	—	3	—	2	—
10	—	—	—	—	1	—
11	—	—	—	—	2	—
12	—	—	—	—	1	—
13	—	—	—	—	1	—
14	—	—	—	—	1	—
15	—	—	—	—	2	—
16	—	—	—	—	1	—
17	—	—	—	—	3	—
18	—	—	—	—	1	—
Total	69	26	72	24	39	5

The 10 plans that underwent improvements to their structure and knowledge representation (21%) analyzed 149 events (63%). The average number of utilization of the plans vary from 4.9 (with standard deviation of 8) for all 48 plans to 14.9 (with standard deviation of 11.8) for the 10 improved plans that are marked with the * and + superscripts. These numbers support the argument that commonly used plans get more chances to be revised and adapted and this, in turn, leads to their higher utilization.

We also notice, from Table 4, that even though nine SLAPs analyzed 72 events, double the number of SHAPs (19) only analyzed 39 events. This indicates that simple 'low-level' blocks of code are more frequently utilized than the more complex 'high-level' ones.

In general, the results in Table 4 show that the events/plan ratio is high (4.9), especially in case of the plans that underwent structural and knowledge representation improvements (14.9). This indicates that the decomposition and plan design methods tend to have a positive effect on plan utilization and, consequently, on the size of the knowledge base. However, since our main objective was to validate and evaluate the analysis approach, we designed many plans (21) that were only used once. These plans helped us in evaluating the analysis approach in loops with, say, high nesting level or a large number of procedure calls. Since these plans were designed to handle single specific events, they are probably not fully developed. The analysis of more loops in the same application domain should either eliminate or improve them.

Tables 5 and 6 summarize the data collected to determine which kinds of loops are more appropriately analyzed by the approach. Table 5 provides some insight into the practical limits of the approach. It gives different characteristics of the partially analyzed loops. Table 6 compares some of these characteristics to the corresponding ones of the completely analyzed loops. To provide a more detailed insight into the analyzed loops, some loop source codes are given in Appendix C.

The second theoretical limitation, described in Section 6, only occurred in loop p9. That is, the partial analysis of the 12 loops in this case study was mainly because of practical limitations. Analyzing loops p1–p6 and p8–p9 using the current set of plans yielded no partial results. Loop p10, whose characteristics are compatible with those of the completely analyzed loops, was almost completely analyzed; four out of five events were analyzed. The fifth event was not analyzed because it contained a call to a procedure with a partially analyzed nested loop (loop p9). Loops p7 and p12 yielded some minor partial analysis results. Loop p11 gave considerable partial analysis results.

It is clear from Table 5 that almost all of the partially analyzed loops are nested (11 out of 12) and contain procedure calls (10 out of 12). They have an average size of 43.2 executable source lines of code and an average of 12.4 modified variables. Table 6 shows that some of these characteristics are considerably different from the corresponding ones for the completely analyzed loops. For example, the completely analyzed loops have an average size of 10.5 executable source lines of code and an average of 3.4 modified variables. While the average number of events in the completely analyzed loops is 3.3, the partially analyzed loops have 11.9 events on the average. This case study has given us the impression that loops of up to five events were more easily analyzed than others.

However, we noticed in some loops (p7, p8, p11, and p12) that some events closely match some of the designed plans. A larger domain of study could have improved those plans or resulted in designing similar ones that can contribute more to the specifications of such loops.

Even though the results of the case study are encouraging, further experimentation is, in our opinion, needed to investigate the generality and efficiency of the presented approach with respect to various application domains. This experimentation can serve to characterize the cases in which this approach can work best.

TABLE 5
CHARACTERISTICS OF THE 12 PARTIALLY ANALYZED LOOPS

Loop #	Characteristics								
	Class	Events	Executable SLOC	Modified variables		Pointer variables	Procedure calls	Function calls	Inner loops
				control	non-control				
p1	GCN	13	48	3	10	0	5	4	1
p2	GCN	9	30	3	6	0	2	2	1
p3	GCN	13	46	3	10	0	3	2	2
p4	GCN	9	32	3	6	0	2	2	1
p5	GCN	13	49	3	10	0	4	2	2
p6	SNN	17	53	1	16	2	7	2	1
p7	SNN	20	53	1	19	4	0	1	1
p8	GCN	8	36	2	7	2	4	0	1
p9	SNN	5	29	1	4	3	0	0	2
p10	GCF	5	13	2	4	0	1	2	0
p11	GCN	12	52	3	11	4	1	4	2
p12	SNN	19	77	1	20	4	1	4	3

TABLE 6
COMPARISON BETWEEN THE COMPLETELY AND PARTIALLY ANALYZED LOOPS

Characteristics (in terms of average numbers)	Completely analyzed loops	Partially analyzed loops
Events	3.3 (SD = 2.1)	11.9 (SD = 4.8)
Executable SLOC	10.5 (SD = 8.3)	43.2 (SD = 15.7)
Modified variables	3.4 (SD = 2.5)	12.4 (SD = 4.9)

8 IMPLEMENTATION

To demonstrate the feasibility of automating our knowledge-based analysis approach, a prototype tool, which annotates loops with predicate logic annotations, has been designed [2]. LANTeRN, which stands for "Loop ANalysis Tool for Recognizing Natural-concepts," has been developed using Lisp. The input to the current version of LANTeRN is in the form of a loop to be analyzed, and its declarations, written in a subset of Pascal. It is assumed that the input Pascal program has been previously compiled successfully. LANTeRN's output includes the loop classification, loop events along with names of the plans they match, individual event analysis results, and synthesized and adapted final results. Its knowledge bases contain plans and ACs from the case study. The test cases were also used from the case study. It should also be mentioned that the specifications presented in this paper were generated by LANTeRN.

In the current implementation, construction of the plans and ACs is not automated. That is, we manually populated the knowledge bases. However, no human interaction is needed during the utilization of the plans and ACs during analyzing loops. The construction of the tree-structured plans, especially in case of large knowledge bases, can be facilitated by the design of automated techniques that assist in their acquisition and development. For instance, several knowledge base plans might have the same antecedent parts except for the **firing-conditions**. Other plans might have antecedents that represent special cases of a more general antecedent. Automatically identifying such plans and combining them into more sophisticated tree structures is an interesting topic for future study.

The first phase in the implementation is a translation phase that converts the input into a language independent form. The loop initialization and body are converted into a set of lisp function calls. The loop condition, however, is left in its predicate form. Data type information is also extracted. The translation results are stored in a global data base so that they can be easily accessed by all analysis phases. After the translation phase, the rest of the prototype can be used to analyze loops independent of the imperative programming language used.

Starting from the innermost loop(s), all input loops are recursively analyzed. If the loop body contains inner loop(s), the AK tuples of the inner loop(s) are used to search for the matching ACs in the AC knowledge base. The inner loop(s) are then replaced by a concurrent assignment in terms of the found ACs as explained in Section 5.2. After this replacement, the four main phases of the loop analysis approach are implemented by following the descriptions given in Section 4. Two kinds of simplifications are performed in LANTeRN. The simplification of arithmetic expressions is performed by converting input expressions into an internal canonical form for polynomials, manipulating them, and converting them back to their external form [34]. Predicate simplifications, however, are limited. They are performed using rule-based translation with a set of logical identities serving as rules.

Because LANTeRN was designed for the specific purpose of demonstrating that our approach can be automated, its user interface is primitive and the only structured type currently being handled is the array type. Because of the second limitation, all loops considered in our case study were analyzed by LANTeRN except for those which included pointers.

9 Conclusion

In this paper, a knowledge-based loop analysis approach has been described. This approach mechanically generates rigorous unambiguous predicate logic annotations of computer programs. It is a bottom-up analysis approach that does not rely on real-time user-supplied information that might not be available at all times (e.g., the goals a program is supposed to achieve). In addition, it enables partial recognition and analysis of stereotyped, nonadjacent program parts.

A case study was performed on a real and existing program of some practical value. This case study served to partially validate the analysis approach and to characterize its practical limits. To demonstrate the feasibility of automating our knowledge-based analysis approach, a prototype tool, which annotates loops with predicate logic annotations, has been designed and implemented [2].

The approach can assist in the maintenance and reuse activities by producing semantically sound and expressive predicate logic annotations of programs. Since many programs are undocumented, underdocumented, or misdocumented, a major part of the maintenance task is spent in recognizing and understanding abstract programming concepts [5], [28]. Automation of program analysis and understanding can, thus, contribute to maintenance tools and methods and provide support for various maintenance activities. Program analysis and understanding is also crucial for code reuse since the reuser must be aware of what a code component does. Understanding reusable code components can be achieved by augmenting them with a precise and clear description of their functionality [7]. If these descriptions are in the form of formal specifications, they can be further used in generating test cases and assessing the correctness of the implementation. Automation of program understanding is needed to facilitate the quick and efficient population of a reuse repository with well documented components [4], [11].

However, when annotating complicated and large program parts, these formal specifications can become hard to read. The readability of such specifications can be enhanced if they are further abstracted. This abstraction can be performed by replacing a formal statement with another one that is formulated in terms of a more widely known and understood concept [13]. Domain abstractions can further abstract the formal specifications with concepts specific to the application domain. The domain specific replacements can be explicitly performed by producing the abstract and then the domain specific ones. Otherwise, they can be implicitly performed by designing the plans such that their consequents are directly written in terms of the domain specific terms. In the former case, the knowledge base plans are more general and can be used in several different domains. The last stage that performs the higher level abstractions can be tailored to the needs of different domains and thus enhances the portability of the system. The latter approach, however, is easier to implement mechanically but reduces the generality of the plans.

With respect to software development, predicate logic plays an important role in development of software using such languages as VDM and Z [25], [44], [51]. Since our loop analysis technique produces predicate logic annotations, it can assist such formal development methods. Our reverse engineering approach can provide assistance in the last development stage that moves from operation specifications to imperative programming language implementations. That is, the presented loop analysis technique can help in showing that proof obligations generated during the operation refinement process are satisfied. It should be noted, however, that the mathematical notations used in VDM, Z, and our plans are not the same. To transform one mathematical notation to another, simple syntactic variations need to be performed. For a detailed description of how our approach can assist in program development with VDM and Z, refer to [2].

There are some practical and theoretical limits to the presented approach. The practical limits are due to the difficulty of designing the knowledge base plans. The theoretical limits occur in nested structures with adaptation paths that contain statements other than assignment and conditional statements. They also occur while deriving the postconditions of general loops.

Future work includes extensions and improvements of the analysis approach, experimenting with the techniques in various application domains, and improvements on the prototype tool. The analysis approach needs to be expanded to perform an intelligent analysis of complete program modules that include nonalgorithmic constructs such as stacks and queues. We need to investigate the utilization of additional information and knowledge in the source code (e.g., comments, variable names) to assist in plan recognition. Performing empirical studies in various application domains can serve to address and investigate several issues related to the acquisition and development of plans and the generality and efficiency of the presented approach with respect to different application domains. Finally, the developed tool served to demonstrate that the analysis techniques can be automated [2]. For practical utilization of such a tool, it needs to be enhanced to support additional programming language features and improve the user interface.

APPENDIX A – NOTATION

$\neg$	The negation operator
$\Rightarrow$	The implication operator
$x \in y$	x is an element of y
$x \cup y$	Union of the sets x and y
$*$	Denotes an irrelevant information
var?	Value of *var* before an operation or a loop
var_{outer}	Value of *var* as deduced from the outer loop invariant
var_{adapt}	Value of *var* as deduced from the adaptation path of the current inner loop
$i \,..\, j$	Sequence of integers from i up to j inclusive
(*FORALL* x: *p1*: *p2*)	For all x values that satisfy *p1*, *p2* is true
and	The logical conjunction operator
B	While loop condition
MIN s	The minimum of the set (or sequence) *s*
MAX s	The maximum of the set (or sequence) *s*
or	The logical disjunction operator
P^x_y	The result of substituting y for each free occurrence of x in P
P{S}Q	If the predicate P is true before executing the first statement of the program part S, and if S terminates, then the predicate Q will be true after the execution of S is complete
PERM(a, b)	Array a is a permutation of the array b
PRED(x)	The predecessor of x
SUCC(x)	The successor of x

APPENDIX B – ACRONYMS

AC	Abstraction Class
AC_{DB}	AC that abstracts AK tuples whose first term is a DBP
AC_{SA}	AC that abstracts AK tuples whose first item is a SAP
AE	Augmentation Event
AK	Analysis Knowledge
AP	Augmentation Plan
BE	Basic Event
BP	Basic Plan
DBP	Determinate Basic Plan
GAP	General Augmentation Plan
GCF	General loop, Composite condition, Flat
GCN	General loop, Composite condition, Nested
GHAP	General High-level Augmentation Plan
GLAP	General Low-level Augmentation Plan
GNF	General loop, Noncomposite condition, Flat
GNN	General loop, Noncomposite condition, Nested
IBP	Indeterminate Basic Plan
LANTeRN	Loop ANalysis Tool for Recognizing Natural-concepts
SCF	Simple loop, Composite condition, Flat
SCN	Simple loop, Composite condition, Nested
SHAP	Simple High-level Augmentation Plan
SLAP	Simple Low-level Augmentation Plan
SNF	Simple loop, Noncomposite condition, Flat
SNN	Simple loop, Noncomposite condition, Nested
UAC	Unknown Abstraction Class

APPENDIX C – EXAMPLE LOOPS

The following four figures provide a more detailed insight into the analyzed loops. The first two figures (Figs. 18 and 19) show two of the completely analyzed loops. The last two figures (Figs. 20 and 21) demonstrate two of the partially analyzed loops. These two loops were referenced as p1 and p9, respectively.

```
i := 1;
course_i := 0;
flag := false;
while (i <= num_of_courses) and not flag do begin
  if course_no = course_no_db[i] then begin
    course_i := i;
    flag := true
  end else
    i := i + 1
end
```

Fig. 18. First example of a completely analyzed loop.

```
num_of_pref := 0;
valid_pref_list := nil;
while pref_list <> nil do begin
  num_of_pref := num_of_pref + 1;
  i := 1;
  flag := false;

  while (i <= num_of_times) and not flag do begin
    if pref_list^.p_index = time_slot_db[i] then begin
      new(temp_list);
      temp_list^.p_index := i;
      if num_of_pref = 1 then
        temp_list^.p_ptr := nil
      else
        temp_list^.p_ptr := valid_pref_list;
      valid_pref_list := temp_list;
      flag := true
    end else
      i := i + 1
  end;

  if i > num_of_times then begin
    writeln('line # ', line_no: 3, ' ** ', msgbuf);
    writeln('      no such preference in the db : ', pref_list^.p_index);
    num_of_pref := num_of_pref - 1
  end;
  pref_list := pref_list^.p_ptr
end
```

Fig. 19. Second example of a completely analyzed loop.

```
error := false;
num_of_rooms := 0;
get_next_line(1, buffer);
line_no := line_no + 1;
msgbuf := buffer;
flag := false;
while (buffer <> 'eof') and (num_of_rooms < maxrooms)and not flag do begin
  token := get_token([';', ':'], buffer);
  if token = '; ' then begin
    flag := true
  end;
  if not flag then begin
    if not chk_fmt_rm_no(token) then begin
      error := true
    end;

    {The following loop was completely analyzed.}
    i := 1;
    while i <= str_length do begin
      room_no[i] := token[i];
      i := i + 1
    end;
    token := get_token([';', ':'], buffer);
    if token = ': ' then
      token := get_token([';', ':'], buffer);
    cap := string_to_int(token);
    if not chk_range_cap(cap) then
      error := true;
    if not error then
      if not chk_dup(room_no) then begin
        num_of_rooms := num_of_rooms + 1;
        classroom_db[num_of_rooms].room_no := room_no;
        classroom_db[num_of_rooms].capacity := cap
      end else begin
        writeln('line #', line_no: 3, ' ** ', msgbuf);
        writeln('   classroom entry specified more than once :', room_no, '** ignored **')
      end;
    token := get_token([';', ':'], buffer);
    if token = '; ' then
      flag := true
    else
      if num_of_rooms <> maxrooms then begin
        get_next_line(1, buffer);
        line_no := line_no + 1;
        msgbuf := buffer
      end
  end
end;
```

Fig. 20. Partially analyzed loop number p1.

```
slot_full := true;
temp_pg_res := pg_reserve;
while temp_pg_res <> nil do begin

  {The following loop was completely analyzed.}
  temp_timeslots := temp_pg_res^.timeslots;
  while (temp_timeslots<>nil)and(temp_timeslots^.timeslot<>time) do
    temp_timeslots := temp_timeslots^.t_ptr;

  if temp_timeslots <> nil then begin
    temp_room_list := temp_timeslots^.roomlist;
    if (temp_room_list <> nil) and (temp_room_list^.r_index = room) then begin
      temp_timeslots^.roomlist := temp_timeslots^.roomlist^.r_ptr;
      if temp_timeslots^.roomlist <> nil then
      slot_full := false
    end else begin
      slot_full := false;
      if temp_room_list <> nil then begin

        {The following loop was completely analyzed.}
        flag := false;
        while not flag and (temp_room_list^.r_ptr <> nil) do
          if temp_room_list^.r_ptr^.r_index <> room then begin
            temp_room_list := temp_room_list^.r_ptr
          end else
            flag := true;

        if temp_room_list^.r_ptr <> nil then
          temp_room_list^.r_ptr := temp_room_list^.r_ptr^.r_ptr;
      end
    end
  end;
  temp_pg_res := temp_pg_res^.res_next
end;
```

Fig. 21. Partially analyzed loop number p9.

ACKNOWLEDGMENT

We thank Lionel Briand, Gianluigi Caldiera, Walcelio Melo, Carolyn Seaman and Barbara Swain for their helpful contributions to a variety of aspects presented in this paper. This research was supported in part by the Office of Naval Research under Grant No. NOOO14-87-k-0307 to the University of Maryland.

REFERENCES

[1] S.K. Abd-El-Hafiz and V.R. Basili, *A Knowledge-Based Approach to Program Understanding*. Kluwer Academic Publishers, 1995.
[2] S.K. Abd-El-Hafiz and V.R. Basili, "A Tool for Assisting the Understanding and Formal Development of Software," *Proc. Sixth Int'l Conf. Software Engineering and Knowledge Engineering*, Jurmala, Latvia, pp. 36-45, 1994.
[3] S.K. Abd-El-Hafiz and V.R. Basili, "Documenting Programs Using a Library of Tree Structured Plans," *Proc. Conf. Software Maintenance,*" Montreal, Canada, pp. 152-161, 1993.
[4] S.K. Abd-El-Hafiz, V.R. Basili, and G. Caldiera, "Towards Automated Support for Extraction of Reusable Components," *Proc. Conf. Software Maintenance*, Sorrento, Italy, pp. 212-219, 1991.
[5] A. Abran and H. Nguyenkim, "Analysis of Maintenance Work Categories Through Measurement," *Proc. Conf. Software Maintenance*, Sorrento, Italy, pp. 104-113, 1991.
[6] D. Allemang and B. Chandrasekaran, "Functional Representation and Program Debugging," *Proc. Sixth Ann. Knowledge-Based Software Engineering Conf.*, Syracuse, N.Y., pp. 167-178, 1991.
[7] V.R. Basili and S.K. Abd-El-Hafiz, "A Method for Documenting Code Components," *The J. of Systems and Software*, to appear.
[8] S.K. Basu and J. Misra, "Proving Loop Programs," *IEEE Trans. Software Engineering*, vol. 1, no. 1, pp. 76-86, 1975.
[9] K. Bertels, P. Vanneste, and C. De Backer, "A Cognitive Approach to Program Understanding," *Proc. Working Conf. Reverse Engineering*, Baltimore, Md., pp. 1-7, 1993.
[10] G. Brassard and P. Bratley, *Algorithmics: Theory Practice*. Prentice Hall, 1988.
[11] G. Caldiera and V.R. Basili, "Identifying and Qualifying Reusable Software Components," *Computer*, vol. 24, no. 2, pp. 61-70, 1991.
[12] D.D. Dunlop and V.R. Basili, "A Heuristic for Deriving Loop Functions," *IEEE Trans. Software Engineering*, vol. 10, no. 3, pp. 275-285, 1984.
[13] R.B. France and V.R. Basili, "A Pattern-Driven Approach to Code Analysis for Reuse," Tech. Report CS-TR-2802, Dept. of Computer Science, Univ. of Maryland, College Park, Md. 1991.
[14] D. Gries, *The Science of Programming*. Springer-Verlag, 1981.
[15] M.T. Harandi and J.Q. Ning, "PAT: A Knowledge-Based Program Analysis Tool," *Proc. Conf. Software Maintenance*, Phoenix, Ariz., pp. 312-318, 1988.
[16] M.T. Harandi and J.Q. Ning, "Knowledge-Based Program Analysis," *IEEE Software*, vol. 7, no. 1, pp. 74-81, 1990.
[17] J. Hartman, "Understanding Natural Programs Using Proper decomposition," *Proc. 13th Int'l Conf. Software Engineering*, pp. 62-73, Austin, Tex., 1991.
[18] P.A. Hausler, M.G. Pleszkoch, R.C. Linger, and A.R. Hevner, "Using Function Abstraction to Understand Program Behavior," *IEEE Software*, vol. 7, no. 1, pp. 55-63, 1990.
[19] C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," *Comm. ACM*, vol. 12, no. 10, pp. 576-580, 583, 1969.

[20] C.A.R. Hoare, "Procedures and Parameters: An Axiomatic Approach," *Symp. Semantics of Algorithmic Languages*, pp. 102-116. Springer-Verlag, 1971.
[21] C.S. Hsieh, "Slice, Chunk, and Dataflow Anomaly as Datalog Rules," *The J. of Systems and Software, vol.* 16, no. 3, pp. 197-203, 1991.
[22] P. Jalote, *An Integrated Approach to Software Engineering*. Springer-Verlag, 1991.
[23] W.L. Johnson, *Intention-Based Diagnosis of Novice Programming Errors*, Morgan Kaufmann, 1986.
[24] W.L. Johnson and E. Soloway, "PROUST: Knowledge-Based Program Understanding," *IEEE Trans. Software Engineering*, vol. 11, no. 3, pp. 267-275, 1985.
[25] C.B. Jones, *Systematic Software Development Using VDM*. Prentice Hall Int'l, 1990.
[26] S. Katz and Z. Manna, "Logical Analysis of Programs," *Comm. ACM*, vol. 19, no. 4, pp. 188-206, 1976.
[27] K. Lano and P. Breuer, "From Programs to Z Specifications," *Z User Workshop*, *J.E.* Nicholls , pp. 46-70. Springer-Verlag, 1989.
[28] B.P. Leintz and E.B. Swanson, *Software Maintenance Management*. Addison-Wesley, 1980.
[29] S. Letovsky, "Program Understanding with the Lambda Calculus," *Proc. 10th Int'l Joint Conf. on AI*, pp. 512-514, 1987.
[30] D.C. Luckham and N. Suzuki, "Verification of Array, Record, and Pointer Operations in Pascal," *TOPLAS*, vol. 1, no. 2, pp. 226-244, Oct. 1979.
[31] K.B. McKeithen, J.S. Reitman, H.H. Rueter, and S.C. Hirtle, "Knowledge Organization and Skill Differences in Computer Programmers," *Cognitive Psychology*. vol. 13, pp. 307-325, 1981.
[32] H.D. Mills, V.R. Basili, J.D. Gannon, and R.G. Hamlet, *Principles of Computer Programming: A Mathematical Approach*. Allyn and Bacon, 1987.
[33] W.R. Murray, *Automatic Program Debugging for Intelligent Tutoring Systems*. Morgan Kaufmann, 1988.
[34] P. Norvig, *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, 1992.
[35] T.W. Pratt, "Control Computations and the Design of Loop Control Structures," *IEEE Trans. Software Engineering*, vol. 4, no. 2, pp. 81-89, 1978.
[36] A. Quilici, "A Hybrid Approach to Recognizing Programming plans," *Proc. Working Conf. Reverse Engineering*, pp. 126-133, Baltimore, Md., 1993.
[37] C. Rich, "A Formal Representation for Plans in the Programmer's Apprentice," *Proc. Seventh Int'l Joint Conf. on AI*, pp. 1,044-1,052, Aug. 1981.
[38] C. Rich and L.M. Wills, "Recognizing a Program's Design: A Graph-Parsing Approach," *IEEE Software*, vol. 7, no. 1, pp. 82-89, 1990.
[39] E. Rich and K. Knight, *Artificial Intelligence*. McGraw-Hill, 1991.
[40] P.G. Selfridge, R.C. Waters, and E.J. Chikofsky, "Challenges to the field of Reverse Engineering," *Proc. Working Conf. Reverse Engineering*, Baltimore, Md., pp. 144-150, 1993.
[41] E. Soloway, "Learning to Program = Learning to Construct Mechanisms and Explanations," *Comm. ACM*, vol. 29, no. 9,pp. 850-858, 1986.
[42] E. Soloway, J. Bonar, and K. Ehrlich, "Cognitive Strategies and Looping Constructs: an Empirical Study," *Comm. ACM*, vol. 26, no. 11, pp. 853-860, 1983.
[43] E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge," *IEEE Trans. Software Engineering*, vol. 10, no. 5, 1984.
[44] J.M. Spivey, "An Introduction to Z and Formal Specifications," *Software Engineering J.*, pp. 40-50, Jan. 1989.
[45] J.M. Spivey, *The Z Notation: a Reference Manual*. Prentice Hall Int'l, 1992.
[46] M. Ward, F.W. Calliss, and M. Munro, "The Maintainer's Assistant," *Proc. Conf. Software Maintenance*, Miami, Fla., pp. 307-315, 1989.
[47] R.C. Waters, "A Method for Analyzing Loop Programs," *IEEE Trans. Software Engineering*, vol. 5, no. 3, pp. 237-247, 1979.
[48] R.C. Waters, "The Programmer's Apprentice: A Session with KBEmacs," *IEEE Trans. Software Engineering*, vol. 11, no. 11, pp. 1,296-1,320, Nov. 1985.
[49] B. Wegbreit, "The Synthesis of Loop Predicates," *Comm. ACM*, vol. 17, no. 2, pp. 102-112, 1974.
[50] L.M. Wills, "Flexible Control for Program Recognition," *Proc. the Working Conf. Reverse Engineering*, Baltimore, Md., pp. 134-143, 1993.
[51] J.C.P. Woodcock, "Structuring Specifications in Z," *Software Engineering J.*, pp. 51-66, Jan. 1989.

Salwa K. Abd-El-Hafiz received the BS degree in electrical engineering from Cairo University, Egypt, in 1986 and the MS and PhD degrees in computer science from the University of Maryland, College Park, Maryland, in 1990 and 1994, respectively. She is currently an assistant professor at the Engineering Mathematics Department, Faculty of Engineering, Cairo University, Giza, Egypt. Her primary research interests in software engineering are program understanding, the application of artificial intelligence to software analysis, and software specification. Other interests include software maintenance, reuse, and measurements. Dr. Abd-El-Hafiz is a member of the IEEE Computer Society.

Victor R. Basili ((M'83–SM'84–F'90) is a professor in the Institute for Advanced Computer Studies (UMIACS) and the Computer Science Department at the University of Maryland, College Park, Maryland, where he served as chairman for six years. He was involved in the design and development of several software projects, including the SIMPL family of programming languages. He is currently measuring and evaluating software development in industrial and government settings and has served as a consultant to many agencies and organizations, including IBM, Motorola, HP, Boeing, Xerox, NRL, NSWC, and NASA.

Dr. Basili works on the development of quantitative approaches for software management, engineering, and quality assurance, using models and metrics for improving the software development process and product. He helped found and is one of the principals of the Software Engineering Laboratory, a joint venture between the NASA Goddard Space Flight Center, the University of Maryland, and the Computer Sciences Corporation that was established in 1976. He received the first Process Improvement Achievement Award for the GRO Ada experiment in 1989 and the NASA/GSFC Productivity Improvement and Quality Enhancement Award for the Cleanroom project in 1990.

Dr. Basili has authored more than 100 journal and refereed conference papers. In 1982, he received the Outstanding Paper Award from *IEEE Transactions on Software Engineering* for his paper on the evaluation of methodologies. He has served as editor-in-chief of *IEEE Transactions on Software Engineering*; general chair of the 15th International Conference on Software Engineering held in 1993 in Baltimore, Maryland; program chair of the sixth International Conference on Software Engineering in 1982 in Japan; and general or program chair of several other conferences. He was treasurer of the IEEE Computing Society Board of Governors and a member of the Board of Directors of Verdix Corp. He serves on the editorial board of the *Journal of Systems and Software* and is an IEEE fellow. He is a member of the Engineering Excellence Council for the Xerox Corp. He serves as a co-editor-in-chief of a new journal, *Empirical Software Engineering, An International Journal*, to be published by Kluwer Academic Publishers beginning in April 1996.

58-61
415792
360745
42P

CS-TR-3705
UMIACS-TR-96-75

August 1996

Experiences from an Exploratory Case Study with a Software Risk Management Method

Jyrki Kontio[*], *Helena Englund*[†] *and Victor R. Basili*[*]

[*] Institute for Advanced Computer Studies and
Department of Computer Science
University of Maryland
A.V.Williams Building
College Park, MD 20742, U.S.A.
Email: {jkontio, basili}@cs.umd.edu

[†] Software Metrics Laboratory
Department of Computer & Systems Sciences
Kungliga Tekniska Hogskolan
Electrum 230
S-164 40 Kista, Sweden
Email: d91-hen@nada.kth.se

Version 1.00
Status: Final
Release date: November 1, 1996

Abstract:

This paper presents the results of an exploratory case study in which a risk management method was used and compared with a method currently used by the organization. The goal of the case study was to obtain feedback on an early version of a risk management method, called *Riskit*, that has been developed at the University of Maryland. This paper also presents an overview of Riskit method version 0.10 and describes the comparison method currently used by the case study organization, as well as the case study design.

Table of Contents

List of Figures

List of Tables

1. Introduction

This paper presents the results of an exploratory case study in which a recently developed risk management method was used and compared with a method currently used by the case study organization. The primary goal of this work was to obtain feedback on an early version of a risk management method that had been developed at the University of Maryland. The method, called *Riskit*, is based on a graphical modeling technique that supports qualitative analysis of risks. This case study used an early version of the method (version 0.10) and the results of the case study were used to improve the method.

This paper presents the Riskit method version 0.10, the comparison method currently used by the case study organization, as well as the results of the case study that was performed.

2. Acknowledgments

We would like to thank the Software Engineering Laboratory for the opportunity to carry out the case study described in this report. Sharon Waligora (CSC) recommended the study and lead to Jean Liu (CSC) who was instrumental in finding a suitable project to work on. Sharon also gave us valuable comments on earlier versions of this report. We are also grateful to Scott Green (NASA) who supported the experiment from NASA's side and provided feedback on this report. Filippo Lanubile helped us to discuss our experimental design and arrangements more thoroughly. The biggest contribution, however, to the project was given by Thomas Gwynn of the Computer Sciences Corporation. He made much of his time and expertise available for the case study and his comments and insights proved to be essential in the execution and analysis of the study.

3. Motivation for Risk Management

Software development is often plagued with unanticipated problems that cause projects to miss deadlines, exceed budgets, or deliver less than satisfactory products. While these problems cannot be eliminated totally, some of them can be controlled better by taking appropriate preventive action. Risk management is an area of project management that deals with these threats before they occur. Organizations may be able to avoid a large number of problems if they use systematic risk management procedures and techniques early in projects.

Several risk management approaches have been introduced during the past decade (Boehm, 1989; Charette, 1989; Carr et al. 1993; Karolak, 1996) and while some organizations, especially in the U.S. defense sector (Boehm, 1989; Edgar, 1989), have defined their own risk management approaches, most organizations do not manage their risks explicitly and systematically (Ropponen, 1993). Risk management based on intuition and individual initiative alone is seldom effective.

When risk management methods are used, they are often simplistic and users have little confidence in the results of their risk analysis results. We believe that the following factors contribute to the low usage of risk management methods in practice:

- Risk is an abstract and fuzzy concept and users lack the necessary tools to define risk more accurately for deeper analysis.
- Many current risk management methods are based on quantification of risks for analysis and users are rarely able to provide accurate enough estimates for probability and loss for the analysis results to be reliable.
- Risks have different implications to different stakeholders. Few existing methods provide support for dealing with these different stakeholders and their expectations.
- Each risk may affect a project in more than one way. Most existing risk management approaches focus on cost, schedule or quality risks, yet their combinations or even other characteristics (such as future maintenance effort or company reputation) may be important factors that influence the real decision making process.
- Many current risk management methods are perceived as complex or too costly to use. A risk management method should be easy to use and require a limited amount of time to produce results, otherwise it will not be used.

Given the increasing interest in risk management in the industry, we believe that for risk management methods to be applied more widely, the risk management community will need to address the above issues. Furthermore, risk management methods should also provide comprehensive support for risk management in projects, they should provide practical guidelines for application, they should support communications between participants, and they should be credible. The Riskit method was developed to address these issues.

4. The Riskit Method, version 0.10

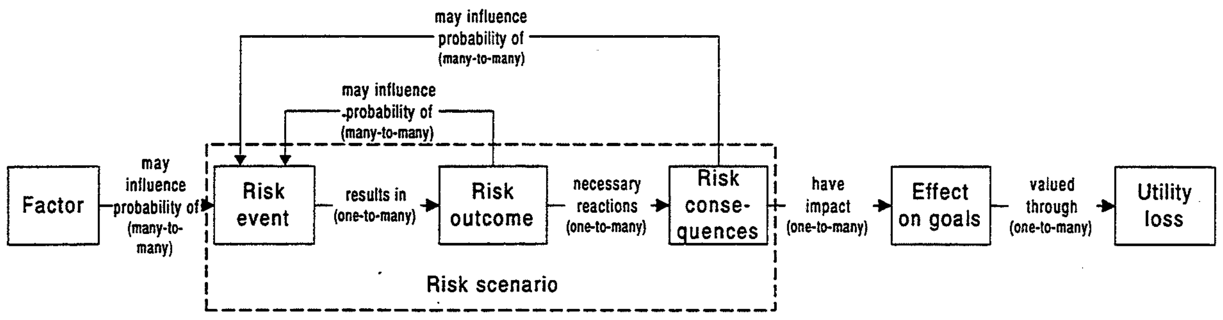
This section presents an overview of the Riskit method as it was defined when the case study was carried out (Kontio, 1995). It is important to point out that a new version of the method is being released at the time of writing of this report (Kontio, 1996), largely based on the feedback obtained during the case study described in this report.

4.1 Decomposing Risk: The Risk Analysis Graph

In everyday language risk can mean various things, it can refer to a possibility of loss, it can mean events that cause loss, it can refer to objects, characteristics or factors that usually are associated with danger or loss (Anonymous, 1992). Clearly, the range of different meanings associated to the word risk is too broad for accurate discussion and analysis.

The *Riskit analysis graph* is a graphical formalism that is used to define the different aspects of risk more formally. The Riskit analysis graph can be seen both as a conceptual template for defining risks, as well as a well-defined graphical modeling formalism. In both cases, it can be used as a communication tool during risk management.

When we use the term risk on its own, we are using it in its original, somewhat fuzzy meaning: risk is defined as *a possibility of loss or any characteristic, object or action that is associated with that possibility.* The two important characteristics of risk are loss and uncertainty. Despite the obvious disadvantages of such broad definition, we have noticed that in the early stages of risk identification and analysis it is beneficial to have such a "fuzzy" concept to facilitate discussion.



may influence probability of (many-to-many)
may influence probability of (many-to-many)
Factor
may influence probability of (many-to-many)
Risk event
results in (one-to-many)
Risk outcome
necessary reactions (one-to-many)
Risk consequences
have impact (one-to-many)
Effect on goals
valued through (one-to-many)
Utility loss
Risk scenario

Figure 1: Definition of the Riskit analysis graph[1]

The Riskit analysis graph is used during the Riskit process to decompose risks into clearly defined components, *risk elements*, as we call them in this document. The components of the Riskit analysis graph are presented in Figure 1. Each rectangle in the graph represents a risk element and each arrow describes the possible relationship between risk elements. The relationship arrow is read in the direction of the arrow, that is, "*[a] factor may influence the probability of [a] risk event*". We have also defined the allowed cardinalities[2] of these relationships, written in parenthesis on each relationship arrow, read in the direction of the arrow. We will define the components of the graph in the following paragraphs.

Risk element	Software Engineering Examples	General Examples
Risk factor	• inexperience of personnel • use of new methods • use of new tools • unstable requirements[3]	• a high cholesterol diet • living near a fault line of earth's plates (e.g., San Francisco) • wet (slippery) driving conditions
Risk event	• a system crashes • a key person quits • extra time spent on learning a method • A major requirements change	• a doctor's diagnosis of a patients heart problem • an earthquake • a car accident
Risk outcome	• the system out of service for a time, some data lost • knowledge is lost, effort shortage • less time spent on actual development	• awareness of the heart problem • buildings collapse, injuries to humans • car demolished, passenger injuries
Risk consequence	• system operational after delay, back up data restored • recruiting process initiated, staff reassigned	• treatment of heart problem • reconstruction of roads and building • treatment of injuries, purchase new car
Risk Effect	• added cost $50K • two calendar-month delay • some functionality lost • reputation as a reliable vendor damaged	• hospital stay, cost of medical care • cost and inconvenience of reconstruction, loss of human life, medical expenses • medical costs, permanent injury effects, raised insurance premiums

Table 1: Examples of risk elements

[1] Note that in the later versions of the method this has been modeled differently, i.e., "event" and "outcome" are combined and "consequence" is replaced by a "reaction".

[2] In this context cardinality refers to the number of allowed connections between risk elements. E.g., in Figure 1 the one-to-many relationship between "risk outcome" and "risk consequence" indicates that each risk outcome can have more than one risk consequence but each risk consequence can only have one risk outcome associated with it.

[3] Note that this is different from "a change in requirements", which would be a risk event. When defined as a factor, "unstable requirements" refers to the characteristics of the situation.

A risk factor is a characteristic that affects the probability of a negative event (that is, risk event) occurring. A risk factor describes the characteristics of an environment, it is not an event itself. Examples of risk factors are listed in Table 1. Risk factors that are documented typically increase the probability of risk events occurring, but they may also reduce them (e.g., "the development team recently developed a similar application").

The purpose of risk factors is not to document all possible characteristics that may influence a risk event as there may be an infinite number of such factors. Instead, a risk factor is relative to the general assumptions madė for the situation (e.g., project), that is, it documents aspects that are somehow different from the "normal" situation. As the arrow in Figure 1 shows, each risk factor can influence one or more risk events.

A *risk event* represents the occurrence of a negative incident – or a discovery of information that reveals negative circumstances. Risk event is a stochastic phenomenon, that is, it is not known for certain whether it will happen or not. This uncertainty can be characterized by a probability estimate associated to the risk event. Examples of risk events are listed in Table 1.

As the arrow in Figure 1 shows, each risk event can be influenced by many risk factors. However, a risk event does not have to have a risk factor associated with it. Each risk event results in one or more risk outcomes. In case there are more than one risk outcome associated with the risk event, the different outcomes represent stochastic relationships.

The *risk outcome* describes the state of the project domain[4] after the risk event has occurred and before any corrective reaction is taken. Risk outcome essentially describes the immediate situation after the risk event. Examples of risk outcomes are listed in Table 1. Risk outcomes can influence the probabilities of other risk events. If the influence is stochastic, they are have a similar relationship as a risk factor has to a risk event. In case of a deterministic relationship (that is, a risk outcome directly results in another risk event) the outcome of the resulting deterministic risk event should be included in the original risk outcome.

If a risk event occurs, the resulting outcome is rarely accepted as such. Instead, organizations take some corrective reaction[5] that reduces the negative impact of the risk event. The *risk consequences* represent the state of project domain after corrective reaction has been taken. Examples of risk consequences are listed in Table 1. These corrective reactions are an important part of understanding what is the overall impact of the risk event to the project domain. Each risk outcome is associated with one or more risk consequences, as shown by the one-to-many relationship arrow the corresponding arrow in Figure 1. Risk consequences may also influence the probabilities of other risk events, as indicated by the arrow in Figure 1.

The *risk effect* represents the impact of risk scenario to project goals after risk has occurred and corrective reactions have been carried out. The effects are stated for all goals that are affected. For instance, it could be that a risk event was a loss of a key person in a project. Corrective reaction includes search for a new person and training of that person. The final effect on project goals could be a delay (search and training took time) and added cost (search and training costs and reduced productivity). Examples of different effects on goals are listed in

[4] Project domain refers to all relevant characteristics of the project and organization.
[5] Note that we use the term "corrective reaction" to action that is taken <u>after</u> the risk event occurs, as opposed to controlling actions that are taken <u>before</u> risk events occur.

Table 1. Each risk consequence can have one or more risk effects associated with it (see Figure 1).

Symbol	Definition
Factor <name>	**Risk factor.** Represents risk factors. Risk factors name is entered in the symbol. May be connected from the right-hand side to one or more risk events.
Event <name> Prob:	**Risk event.** Represents risk events. Event name in entered in the symbol and the probability of the event entered in the "Prob:" field. Need to be connected to one or more outcomes.
Outcome <name> Desc:	**Outcome.** Represents the outcome of the risk event. Descriptive name of the outcome entered in the symbol. Description of the outcome can be included if required. Need to be connected to one or more consequences.
Consequence <name> Desc:	**Consequence.** Represents the consequences and actions that may be taken after the risk event has resulted in an outcome. Descriptive name of the consequence entered in the symbol. Additional description of the consequence or actions included in it. Need to be connected to one or more effects.
Effect <goal1>: <goal2>: <goal3>: <etc.> Impact: <impact>/ <stakeholder>	**Effect.** Effect of a scenario to project goals. Each goal is listed and the scenarios effect on it is described. The effect on goals is expressed using the same metric or description as were used when the goal was defined. The effect is entered as a positive or negative value on each goal and the unit of measure must be included. A zero ("0") is used to indicate that there is no impact for a given goal. Thus, the format is: <sign> <effect> <unit> Below are some examples: Sched: + 2 mo — two month increase in project duration Cost: -$100 K — a $100,000 decrease in project cost Func: -undo feature — the "undo" function will not be available in the system The field "Impact" indicates the total effect on stakeholders' utility. If more than one stakeholder are included in the analysis, the stakeholders are each listed separately.
Action: <description>	**Action.** Risk reducing actions that are planned. The targeted impact on Riskit analysis graph entities is marked by arrows. Actions can be expressed in three ways. Potential actions that have been considered but whose decision whether to implement them or not has not been taken are marked with dashed ovals. Actions that should be taken are marked with solid border. Actions that have already been implemented are marked with a checkmark attached to the action symbol.
——————→	**Deterministic connector.** Represents a certain relationship between risk elements in the Riskit analysis graph.
- - - - - - - - →	**Stochastic connector.** The causality between risk elements is either probabilistic or can be decided.
- - - +/- - - →	**Factor-event connector.** A stochastic connector between risk elements. A positive sign represents an increase in the probability of an event, a negative sign a decrease in the probability.

Table 2: Riskit analysis graph symbols

While the effect on goals represents the impact the risk had on each goal, the concept of *utility loss* captures how severe the loss has been to different stakeholders. The concept of utility loss is based on the utility theory[6], a concept used in economics and decision theory (Von Neumann and Morgenstern, 1944; French, 1989). Increased costs that are within the limits of project contract may not have any meaningful utility loss associated with the project manager. However, the customer paying the bill will consider this loss higher. Also, analyzing utility loss separately allows more appropriate consideration for non-linear and discontinuous utility functions[7].

The utility loss is estimated for each relevant stakeholder. Thus, each risk effect has at least one utility loss estimate associated with it.

We use the term *risk scenario* for any unique event-outcome-consequence combination. Risk scenario is marked in Figure 1 with a named rectangle. Each such scenario can be associated with risk effect and, correspondingly, a set of utility losses. Examples of risk elements can be seen in Figure 3.

The risk elements can be visually represented in the *Riskit analysis graph.* The Riskit analysis graph is based on a graphical modeling formalism developed to support the modeling of risk elements and risk scenarios. The definition of Riskit analysis graph symbols is given in Table 2.

4.2 The Riskit Risk Management Process

This section presents an overview of the Riskit method as it was used during the case study (i.e., version 0.10 of the method). More details are available in a separate report (Kontio, 1995). The updated method has been documented separately (Kontio, 1996).

The risk management cycle in a project can be viewed as consisting of some basic activities: review and definition of goals; risk identification and monitoring; risk analysis; risk control planning; and controlling of risks. The flow of information between these activities is represented in Figure 2. The activities in Figure 2 are represented by circles (process symbols in the dataflow diagram notation used) and the arrows represent information flows between entities. Each of the activities can be instantiated several times during the project duration and they may be enacted concurrently. However, the most critical instances of the risk management cycle are the ones enacted in the beginning of the project.

The risk management approach used in the Riskit method aims at proactive risk management, it attempts to identify actions that can be taken before risks occur, including making contingency plans (that is, the action of planning for reactions should the risks occur). Strictly speaking, once a risk occurs, it is no longer a risk but a problem that needs attention.

[6] The utility theory states that people make relative comparisons between alternatives based on the utility (or utility loss) that they cause. The utility is the level of satisfaction, pleasure or joy that a person feels or expects.
[7] There are strong reasons to assume that utility functions are both non-linear (Friedman and Savage, 1948; Boehm, 1981) and there are points of discontinuity in it.

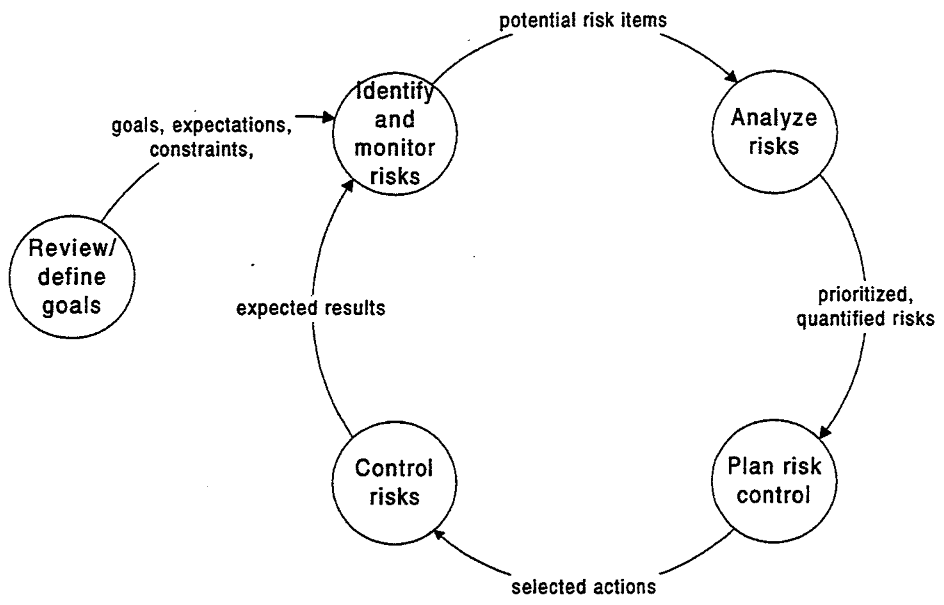


Figure 2: The Riskit risk management cycle

Riskit step	Description	Output
Review and definition of goals	Review the stated goals for the project, refine them and define implicit goals and constraints explicitly. Recognize all relevant stakeholders and their associations with the goals and constraints.	Explicit goal and constraint definition
Risk identification and monitoring	Identify all potential threats to the project using multiple approaches. Monitor the risk situation.	An unanalyzed list of potential risks
Risk analysis	Classify identified risks into risk factors and risk events. Complete risk scenarios for all risk events. Estimate risk effects for all risk scenarios Estimate probabilities and utility losses of risk scenarios.	Completed risk analysis graphs for all identified risks.
Risk control planning	Select the most important risks for risk control planning. Propose risk controlling actions for most important risks. Select the risk controlling actions to be implemented.	Selected risk controlling actions
Controlling of risks	Implement the risk controlling actions.	Reduced risks.

Table 3: Overview of outputs and exit criteria of the Riskit process

4.2.1 Review and Define Goals

Risks do not exist without a reference to goals, expectations or constraints that are associated with a project. If goals are not recognized, the risks that may affect them may be ignored totally or, in the best case, they cannot be analyzed in any detail as the reference level is not defined. Some of a project's goals typically have been explicitly defined but many relevant aspects that influence management decisions may be implicit. Therefore, it is necessary to begin the risk management process of a project by a careful review, definition and refinement of goals and expectations that are associated with a project.

A goal is a general statement of purpose, direction or objective. We have used the term goal in a broad meaning in this text. When defined more accurately, there are three types of possible goals:

Objective:	A goal that has an achievable, well-defined target level of achievement, e.g., "drive from A to B in one hour".
Driver:	A goal that indicates a "direction" of intentions without clearly defined criteria for determining when the "goal" has been reached, e.g., "drive from A to B as fast as you can".
Constraint:	A limitation or rule that must be respected, e.g., "... while obeying all traffic laws".

The Riskit process is initiated by a review of project's goals, which often leads to definition of some additional, previously implicit objectives, drivers and constraints. The purpose of this step is to produce formal definitions of these issues for the stakeholders that the project manager must satisfy. The goals and constraints are expressed using the template presented in Table 4.

Goal attribute	Description
Name	Name of the goal.
Type of goal	Objective / driver / constraint
Description	Description of the goal.
Stakeholder(s)	Names of the stakeholders for the goal.
Measurement unit	Measurement unit used for the goal (e.g., $, date, or person-month).
Target value	Target value for the goal. Relevant for objectives and possibly for constraints.
Direction of increasing utility	Definition of whether an increase or decrease in goal value increases the utility near the target. I.e., whether an increase in goal value is good or bad. Stated as "growing" or "decreasing".
Required value range	Minimum or maximum value required for the goal.

Table 4: Goal and constraint definition template

As Table 4 indicates, goals are linked to different stakeholders that are affiliated with a project. This information will later be used in risk analysis to compare and rank risks. The stakeholders also determine the scope of a project's risk management mandate: which stakeholders are to be defended by the project's risk management activities and which are beyond the risk management mandate of the project. This needs to be explicitly defined for the project, possibly including a prioritization of stakeholder interests.

The goals and constraints are often defined in the project plan or the project contract. However, all the goals and, especially, constraints may not be in these documents. For instance, efficient resource utilization may be an important consideration for the contractor but this typically is not considered a project goal. However, if these goals are real for some of the stakeholders in the project, they must be included in the risk management process. Goals and constraints can typically be found in the following areas:

- schedule
- resources used, most often personnel time
- cost of development
- product requirements, which can include both functional and other quality characteristics.
- resource utilization
- technical constraints, such as hardware platforms, operating systems and use of particular software tools

The goal review can be considered completed when project manager and stakeholders have agreed on the goals and they are formally defined. However, the goal definition process may often need to be re-initiated as new goals are identified during the risk analysis process.

4.2.2 Identify and Monitor Risks

The identify and monitor activity is enacted more thoroughly in the beginning of the project and repeated frequently later in the project as the risk situation is monitored.

The goal of the initial identify and monitor activity is to identify all possible risks that the project may face. It produces a gross list of potential risk factors and risk events for the project, possibly some risk outcomes as well. There are various techniques that can be used to facilitate effective risk element identification, such as brainstorming, checklists (Boehm, 1989; Carr et al. 1993; Karolak, 1996), critical path analysis, and review of goals.

The later instances of the identify and monitor process rely on the results of the initial identification process. The goal of these later process instances is to identify any changes in the risk situation. Changes can include identification of new risks, changes in the risk factor or event information or the consequences of the risk events. The Riskit analysis graph is used as a supporting tool to discuss possible changes.

The risk identification and monitoring activity can be considered completed when the participants have agreed that the produced risk list is comprehensive enough for the project's purposes. The output of the activity is a "raw" list of risks, i.e., each risk has been briefly described.

4.2.3 Risk Analysis

Risk analysis is a process where the information from the identify and monitor process is used and risks are analyzed in detail. The purpose of this activity is to provide detailed descriptions of project's risks so that highest risk elements and appropriate risk controlling action can be planned and implemented in the next step of the Riskit cycle.

The Riskit analysis process consists of the following steps:

- Classify identified risks into risk factors and risk events.
- Complete risk scenarios for all risk events.
- Estimate risk effects for all risk scenarios.
- Estimate probabilities and utility losses of risk scenarios.

The first step, classifying risks into risk factors and risk events, is based on the risk list produced during the identification and monitoring step. The categorization is based on the definitions given in section 4.1 and results are documented in the Riskit analysis graph (Table 2). An example of a Riskit analysis graph is given in Figure 3.

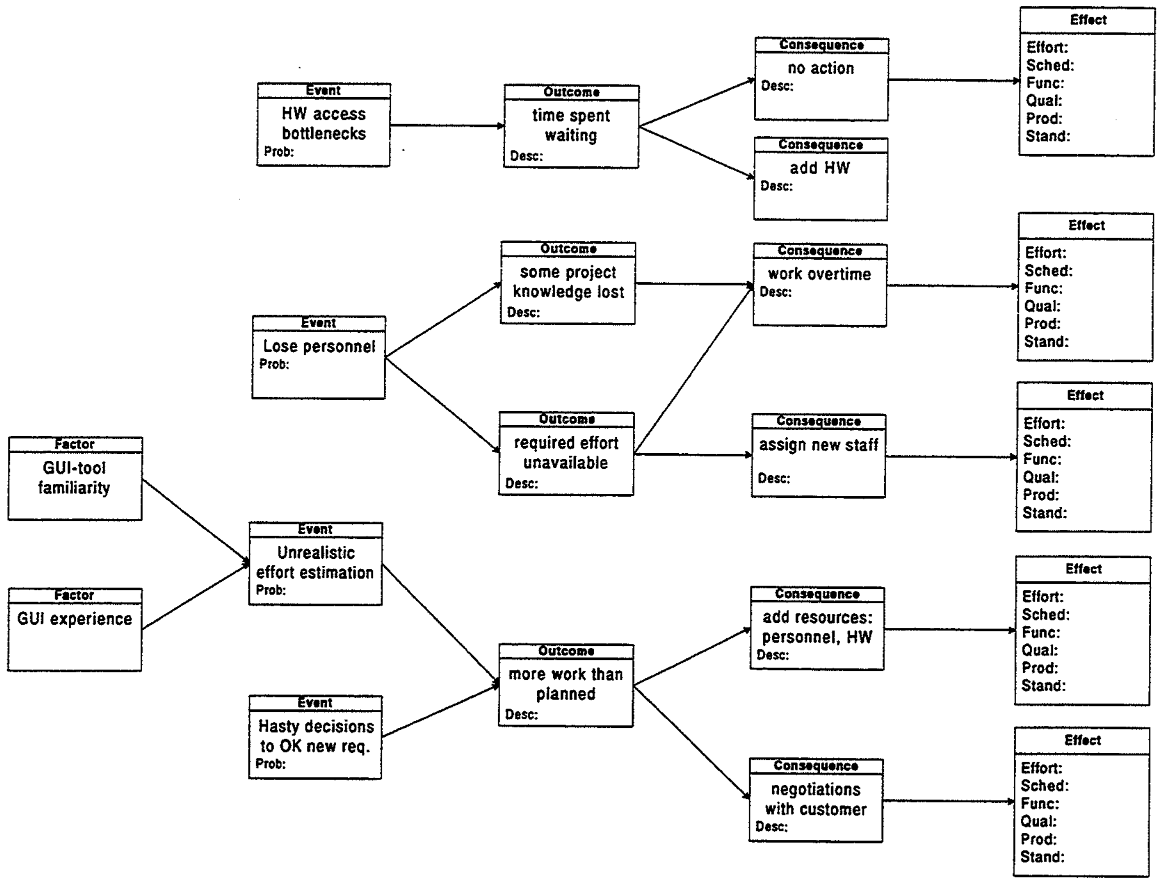


Figure 3: The Riskit analysis graph example

The classification of risks into factors and events is supported by two templates that augment and formalize the graphical presentation of the Riskit analysis graph. Table 5 and Table 6 present these two templates.

Risk factor attributes	Description
Name	Name of the risk factor to be used as an identifier.
Description	Description of the risk factor.
Normal/assumed reference level	Description of the "normal" level for the risk factor.
Project's risk factor state	Description of the risk factors state for the project

Table 5: Risk factor attribute table

Risk event attributes	Description
Name	Name of the risk event to be used as an identifier.
Description	Description of the risk event.
Probability of occurrence	Assessment of the probability of the event occurring.
Uncertainty of the estimate	Assessment of the uncertainty in the probability assessment.
Information source	Description of sources of information about the risk event for monitoring the changes in the probability or event occurrence.

Table 6: Risk event attribute table

As factors and events are being reviewed and positioned on the Riskit analysis graph, the relationships between the two are documented by "influence" arrows Table 2.

The classification process also reviews the listed risks and, when necessary, combines, decomposes or even deletes risks as they are discussed. It is also likely that new risk factors or events may be recognized during the classification process.

The next step in the analysis is to define risk scenarios for all risk events, i.e., define risk outcomes and risk consequences in a scenario. Each risk event has at least one risk outcome, in which case there is a deterministic "result in" relationship between the event and the outcome. Sometimes the outcome may be probabilistic and more than one possible outcome needs to be defined. In such a case, a stochastic connector is used to indicate "may result in" relationship (see Table 2). Similarly, there is at least one risk consequence for each risk outcome (marked by a deterministic relationship) but sometimes alternative lines of action need to be considered and they are marked with a stochastic relationship connector. Templates for risk outcome and risk consequences are presented in Table 7 and Table 8, respectively.

Risk outcome attributes	Description
Name	Name of the risk outcome to be used as an identifier.
Description	Description of the outcome after the risk occurrence. Describes the project state after the event before any other action is taken and this does not need to be directly linked to project goals.
Certainty of the outcome	Assessment of the probability of the outcome if the risk event occurs (when not deterministic).

Table 7: Risk outcome attributes

Risk consequence attributes	Description
Name	Name of the risk consequence to be used as an identifier.
Description	Description of the risk consequence, i.e., the results of possible set of actions that may be required to correct the situation. Note that some of the consequences should be mutually exclusive.

Table 8: Risk consequence attributes

After the risk scenarios have been completed, the risk effects on goals are estimated. Depending on the estimation methods and tools available, the effects can be stated qualitatively (e.g., as textual descriptions or classifications high/medium/low) or quantitatively. Ranges can be expressed as well, if participants consider this necessary.

Not all goals are affected by all risk scenarios and sometimes the effects may be positive for some goals (e.g., loss of personnel may reduce costs while delaying schedule and limiting functionality). Effects on goals are documented in the Riskit analysis graph with the dedicated symbol (see Table 2).

The final step in risk analysis is to rank or estimate the probabilities and utility losses for each risk scenario. The Riskit method itself does not dictate how accurate these estimates are. They may be estimates based on historical data and expressed in ratio scale metrics (e.g., probabilities of events) or they may be ordinal scale rankings of items (Fenton, 1991). As a general rule we suggest that estimates are done using the type metrics that can be supported by the available data or experience. If relevant, reliable historical data exist on probabilities of the events, probabilities may be stated in percentage points. If reliable methods are used to elicit utility loss estimates (e.g., (Saaty, 1990)), they may be expressed in ratio scale preference values. However, it may often be more practical to use ordinal scale rankings or classification categories for this purpose. The goal of risk management is primarily to *identify* the most important risks to be controlled. This identification does not require precise *quantification* of risks.

The utility losses should be estimated separately for all different stakeholders that are to be defended against risk under the risk management mandate of the project.

The probabilities and utility losses are marked in the appropriate risk element symbols in the Riskit analysis graph (see Table 2).

4.2.4 Plan Risk Control

Once the risks have been analyzed and ranked, possible controlling action is planned. The goal of this activity is to determine which risk control activities are necessary to take. This involves three main steps:

- Select the high risk scenarios to be considered for risk control.
- Define possible preventive risk management action for each high risk scenario.
- Select cost-effective actions for all high risk scenarios.

The Riskit method does not advocate any strict rule in determining what are the highest risk scenarios to be controlled. Traditionally, *risk exposure* (i.e., probability * loss) has been used as a metric for risk. If scenario probabilities and utility losses were quantitatively estimated, risk exposures of different scenarios can be used to select highest risk scenarios.

If either probability or loss has been estimated using an ordinal scale, high risk scenarios must be selected using a more qualitative approach, i.e., ranking scenarios into pareto optimal[8] sets, considering scenarios that are in the highest sets, and continuing selection into lower sets until risk scenarios become so insignificant that they do not require any further consideration.

[8] A choice *a* is considered pareto optimal over *b* when $\forall\, i\ a_i >= b_i$ and $\exists\, i\ a_i > b_i$ (French, 1986; Keeney and Raiffa, 1976).

Once the high risk scenarios have been selected, possible controlling actions are proposed for each of them. Identifying possible controlling actions is a creative process and can be carried out in a free format manner. We have also used a simple taxonomy of risk controlling actions as a checklist to verify that no obvious categories of actions are ignored. This taxonomy is presented in Figure 4.

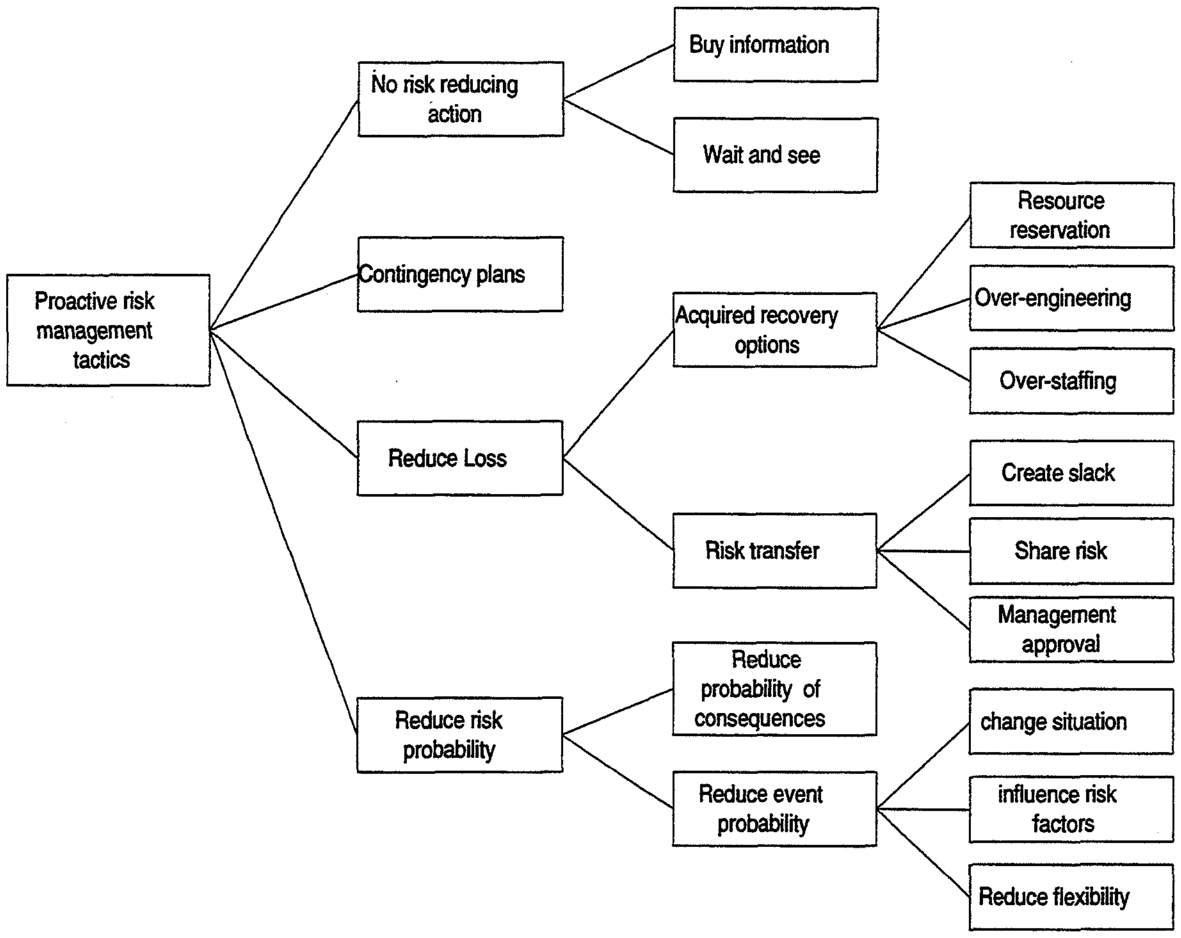


Figure 4: Options for risk management decision making

The first set of options in Figure 4, *no risk reducing action* means that an organization does not take any immediate action to prepare for risk or to reduce risk. *Buying information* is an option that is used when the management does not have enough information to decide what to do about a risk and there is a possibility to obtain more information. In principle, it is only a temporary option that results in a new decision as the information becomes available. After additional information becomes available, some of the other options are selected. Buying information can take many forms. Sometimes information can be literally bought from outside sources, such as market research organizations or by hiring a consultant that knows about the area that risk is relevant to. However, more typical way of buying information is to develop prototypes, run simulations, initiate feasibility studies or conduct, e.g., performance tests.

The *wait and see* option can be used in two situations. First, it is a good option for all risks that are considered to be small enough not to require any other action. Second, it can also be considered when there are no inexpensive ways of obtaining additional information and a major part of the risk is in the uncertainty of the magnitude of the risk itself. In other words, the ranges of estimates of risk are wide and management has no special reason to believe that higher risk estimates are probable. This option, in fact, would be the same as the reactive strategy we discussed earlier. Clearly, using this option to cover high uncertainty risks is, to say it simply, risky. A conservative approach would be to use some of the other options for high uncertainty risks.

Contingency planning means that recovery plans are made for a risk. These plans should describe the actions that will be taken if the risk occurs. Note that this option does not imply that any other preparations are made. Plans are written and approved and they are put on the side and used only if risks occur. Contingency plans do not reduce risk, or loss, to be exact. They help organization to make sure that there is a way to recover from the risk. Contingency plans, in effect, are a way to detail the size of loss.

The options under the term *Reduce loss* build upon recovery plans and include some additional actions that reduce the loss that would result if the risk were to occur. The *Acquired recovery options* refers to a set of actions that buy options that can be used to limit the loss. They typically have a cost associated with them. The *Resource reservation* option refers to a situation where some resources are reserved for limiting the impact of risk if the risk occurs. Resources can be human, computer or financial. *Over-engineering* mean implementing some features in the product or design so that there will be alternative ways of action if the risk occurs. For instance, Over-engineering could mean that extra effort is spent during design or coding to make sure that alternative system architecture or compilers can be used. *Over-staffing* may be introduced to make sure that more than one person knows enough about each area in the project. All these actions buy different options that can be started if risks occur.

Risk transfer can include three different options. The most straight-forward way is to *create slack* in the aspects of project are threatened, i.e., relax objectives of constraints. In other words, lengthen the schedules, make more memory available, or increase budget. Due to competitive situations this may be often difficult. However, if risks are analyzed and communicated well to the management and customers of the project, this option is likely to work better than without risk management.

It is also possible to *share risks*. Sharing can happen, e.g., with customers or subcontractors of the project. Again, a critical issue is to analyze the risks well and communicate their significance to all stakeholders. This typically requires, sometimes lengthy, contractual negotiations.

It is also possible to obtain a *management approval* for some risks. In such a case the management accepts the risk and takes the responsibility for it. Project is still responsible for monitoring the risk but additional actions are not taken. This option may be used when a project is very important for the organizations and there are no available resources for reducing risk.

Reducing the probability of risk can take many forms and is dependent on the type of risk that is to be managed. We have divided this into two categories, *reduce event probability* and *reduce probability of negative consequences* if the risk occurs. For instance, personnel unavailability probabilities can be reduced, to some degree, by financial incentives (project reward on completion) or by addressing the causes that may result in personnel unavailability. Good engineering or design practices can diminish the probability of performance or memory problems.

Once the potential risk controlling actions have been identified, their costs and estimated impacts need to be estimated. The selection of appropriate actions is based on the available resources for risk control and risk reduction effectiveness of the proposed actions. In principle, actions with highest risk reduction leverage[9] (Boehm, 1989) while monitoring that the risk control budget is not exceeded (e.g., some risk controlling action may have a very high risk reduction leverage but the overall cost may be too high for the available budget).

The controlling actions can be presented in the Riskit analysis graph to document their intended and estimated impact. This is done by a specific symbol, an oval, that has arrows pointing to the entities that are targeted (see Table 2). This will highlight how each risk reducing action is intended to influence the risks.

4.2.5 Risk Control

The control process implements the risk controlling actions. From the perspective of the Riskit method this is a project management activity that is not explicitly supported by the Riskit method. However, as risk controlling actions are implemented, they are marked with a checkmark in the Riskit analysis graph. As new information about risks becomes available, the identify and monitor activity may be initiated.

5. Case Study Design

5.1 Case Study Organization

This case study was carried out at the Software Engineering Laboratory (NASA, 1995). The SEL is a partnership organization that was established in 1976 at NASA Goddard Space Flight Center (GSFC) by its Flight Dynamics Division (FDD), Computer Sciences Corporation (CSC) and the department of Computer Science at University of Maryland. The SEL was established for understanding and improving the software products and development process in the FDD. The SEL has a consistent and long track record of systematic process improvement and in 1994 it was awarded the first IEEE Computer Society Software Process Achievement Award to "recognize its outstanding achievements in software process improvement" (McGarry et al. 1994).

The SEL has also been a working example of the Experience Factory and Quality Improvement Paradigm in practice (Basili et al. 1992). The software product and process improvement in the SEL have been improved over the years based on systematic data collection, analysis and organizational learning (Basili and Green, 1994).

The SEL supports the software development within the FDD. Software developed by the FDD is mainly scientific applications that process data received from earth orbiting satellites in the areas of orbit, attitude and mission analysis. The total FDD software development staff,

[9] Risk reduction leverage is defines as $\dfrac{\text{Risk Exposure}_{before} - \text{Risk Exposure}_{after}}{\text{Risk Reduction Cost}}$

including contractor support, is approximately 250-275, and about half of this is allocated to software maintenance. Typical project involves between 5 to 25 staff members and results in system size of 100-300 KSLOC. The SEL itself has a staff of 10-15 analysts (McGarry et al. 1994).

The project selected for study was a small utility that was part of the Flight Dynamics Support System (FDSS) developed by the FDD in support of the Tropical Rainfall Measuring Mission (TRMM). The utility, known as the Maneuver Command Utility (MCU), produces spacecraft maneuver command sheet for use by mission operators. The project had been estimated to be approximately 5 person months in effort and was scheduled to take place between October 1995 and January 1996, including independent system testing. Two people had been assigned to the project along with the project manager.

The project manager that participated in our case study had been using the comparison risk management method for about three years and had used it in close to ten projects.

5.2 The Comparison Method

The project organization in our case study used a systematic risk management approach that was supported by a tool. Based on our assessment, the case study organization's risk management was more mature than what the industry average seems to be (Ropponen, 1993).

The case study organization has provided most managers with training on risk management, primarily focusing on the risk management tool that is used. Risk management is a required activity in all projects and risks are discussed with the management and customer frequently. Risk estimates are normally updated monthly.

The risk management approach is supported by a spreadsheet-based tool that guides risk analysis and helps in quantifying and ranking the risks. This internally developed tool has been in use since 1992 and it has been updated and improved during its usage. This risk management tool seems to be the driver of the risk management process in projects.

The comparison risk management tool collects the following information about each risk:

- Risk title, i.e., the name of the risk
- Risk description, i.e., a textual description of the risk
- Risk source, i.e., list of causes or factors that contribute to the risk
- Risk impact, i.e., a description of the impact the risk would have on the project
- Importance to the customer, i.e., ranking of risk's impact on the customer (expressed as Hi / Med / Lo)
- Current status, i.e., what has been done to the risk item (open / closed / in mitigation)
- Probability of occurrence, i.e., estimated probability of risk occurring, expressed as a probability percentage

The tool also collects information about the impact of risk if no mitigation action is taken, estimating the impact on quality (using a scale of Hi / Med / Lo / None), schedule impact (in weeks) and cost impact (in $K). The weight of these impacts can be set for each risk.

Once each risk has been identified, information about risk mitigation plans is entered into the tool:

- a description of the risk mitigation approach
- the trigger that is used to initiate the risk mitigation
- quality impact of the risk mitigation
- schedule impact of risk mitigation, i.e., the time delay caused by risk mitigation, regardless of whether mitigation is successful or not
- cost impact of risk mitigation, i.e., the additional cost caused of risk mitigation action is taken, regardless of whether mitigation is successful or not
- probability of risk mitigation success

The above information is used to calculate the risk analysis results using three scenarios (i) risk does not occur and no mitigation is done, (ii) risk occurs and mitigation is done but fails, and (iii) risk occurs, mitigation is done and it succeeds. These scenarios and the attributes used are presented in Table 9.

	Risk does not occur	risk occurs	
	no mitigation is done	mitigation is done but fails	mitigation is succesful
Probability	80%*	2%	18%
Quality factor	None	Med	None
Schedule	10 weeks	17 weeks	12 weeks
Cost	$16 K	$26 K	$18 K

* the values do not necessarily represent actual data

Table 9: Results of the comparison method's risk management tool

The decision of the appropriate risk mitigation action is left to decision makers evaluating the risk analysis data.

We interviewed the participating project manager after he had completed the risk analysis using the comparison method. According to him, the main benefit of the method is that it forces projects to think about risks frequently, every month. The approach also gives a quantitative indication of whether risk mitigation should be done. The results are often used in the decision making with management.

When inquired about the usage experiences and possible problems with the comparison risk management approach, the project manager pointed out that probability values are difficult to obtain and there is little support for estimating them, yet they play a critical role in the risk analysis process. "The risks associated with the estimation errors and assumptions used when making these estimates may contain some risks", he pointed out.

5.3 Case Study Goals and Metrics

The objectives of the case study were to assess the feasibility of the Riskit method in an industrial project, investigate the cost and time effectiveness of the method, evaluate the credibility of the method, and compare the Riskit method with the method currently used by the project. Furthermore, the case study was used to provide practical feedback on the use of the method.

Before our case study we initially formulated our evaluation goals and case study metrics in detail using the GQM method (Basili et al. 1994; Basili, 1992). These GQM-based metrics are presented in appendix A. Even though we used most of these metrics in our questionnaire and interviews, they did not result in useful data for our analysis. As we anticipated this problem, we documented the case study in detail so that different types of analyses could be done after the case study, i.e., exploring data or issues that were not necessarily identified in advance. This was done by taking detailed notes during the interviews and observation sessions, storing all the artifacts produced during the case study and writing synthesis reports shortly after the sessions.

Our first evaluation goal, expressed using the format GQM method (Basili et al. 1994; Basili, 1992) was as follows:

Analyze	the Riskit method
in order to	characterize it
with respect to	its feasibility
from perspective of	project manager
in the context of	an industrial project.

We considered the Riskit method feasible, which was our hypothesis, if it meets the following criteria:

- The method produces intended results, i.e., is able to list and rank potential risks and is able to produce a list of controlling action.
- The method can be applied within reasonable time and effort. We are using the recommendations from Ropponen's survey as a guideline: effort allocation between two and eight percent of the project total is considered reasonable (Ropponen, 1993).
- The users of the method give a positive opinion of its feasibility.

In order to evaluate this goal and hypothesis we collected all the output the method produced, including intermediate ones, collected effort data, and interviewed the method user after the use of the method.

Our second goal was to investigate the cost and time effectiveness of the method. This was also described as a GQM goal:

Analyze	the Riskit method
in order to	characterize it
with respect to	its cost-effectiveness
from perspective of	project manager
in the context of	an industrial project.

This goal attempted to measure the effort required to use the method, relative to various aspects of the method, such as number of risks identified and number of risk controlling actions proposed.

Our third goal was to evaluate the credibility of the method. This was also described as a GQM goal:

Analyze	the Riskit method
in order to	characterize it
with respect to	its credibility
from perspective of	project manager
in the context of	an industrial project.

We define a risk management method's credibility as the level of confidence its users have in the results, i.e., the degree to which the output of the method is believable (Garrabrants et al. 1990; Kontio, 1994). This was assessed through asking about the level of confidence directly from the method user as well as monitoring whether the proposed risk controlling actions were actually implemented.

Our fourth goal was to compare the Riskit method with the method currently used by the project. This was defined as the following GQM goal:

Analyze	the Riskit method and the comparison method
in order to	compare them
with respect to	effort, granularity, coverage, accuracy and effectiveness
from perspective of	project manager
in the context of	an industrial project.

As we intended to discover qualitative differences between the methods we did not specify specific metrics for this goal in advance. Instead, we planned to use the data collected to identify possible differences and compare the methods qualitatively.

5.4 Case Study Arrangements

We arranged our case study so that we were able to compare the two risk management methods used in the project, the Riskit method and the comparison method. As Figure 5 shows, the case study started by a joint session where project goals were reviewed and risks identified. Using the list of risks produced the project manager used the comparison method to carry out risk analysis the way he normally does it. After this the risk analysis using the Riskit method was carried out. After both analyses the project manager decided on which risk controlling actions he should actually take.

The project manager performed the first risk analysis on his own and documented the results of his analysis, including the risk controlling action he was planning to take.

The Riskit method was applied in a session where the method expert (i.e., the method author, J. Kontio) facilitated the session. This was done for two reasons. First, the project manager's time was not available for training him well enough in the method so that he could have reliably applied it on his own. Second, by facilitating the Riskit risk analysis we hoped that we would be able to avoid the effect caused by having applied the comparison method first.

Figure 5 also shows where and how we collected the case study data. A dashed line to the vertical line from a case study activity indicates whether we used observation or interviews and questionnaire to obtain relevant data. A connector appearing after an activity box indicates that the information was obtained after the activity was completed.

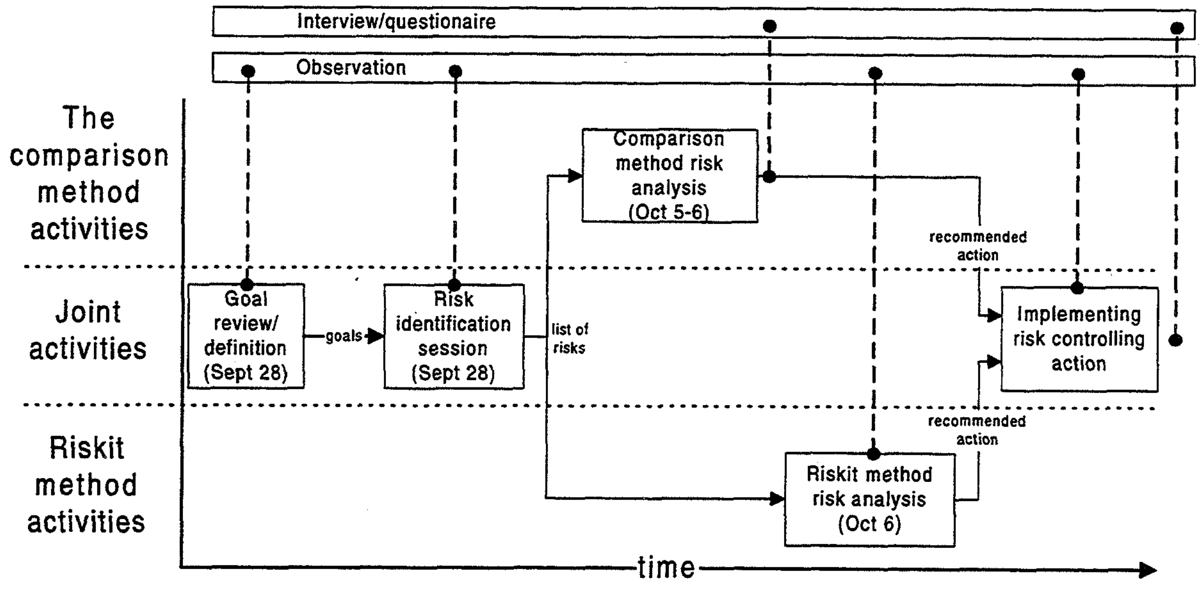

Interview/questionaire
Observation
The comparison method activities
Joint activities
Riskit method activities
Comparison method risk analysis (Oct 5-6)
Goal review/ definition (Sept 28)
goals
Risk identification session (Sept 28)
list of risks
recommended action
Implementing risk controlling action
recommended action
Riskit method risk analysis (Oct 6)
time

Figure 5: The timeline of case study activities

5.5 Validity Threats

In this section we discuss the limitations that our case study design had with respect to validity of the results.

As we had only a single project in the study we were forced to apply the methods in sequence and this may have lead to some maturation effects (Campbell and Stanley, 1963; Judd et al. 1991), i.e., the accumulated time spent on risk management may have increased participant's awareness and knowledge about risks. We tried to minimize this effect by taking two specific actions. First, even though the dedicated risk identification session is a characteristic of the Riskit method and not of the comparison method, we decided to conduct a joint risk identification session for both methods. We reasoned that risk identification would be especially vulnerable to maturation effect and could seriously bias the results. As risk identification is not a main aspect of the Riskit method we did not consider this a serious compromise in the method comparison. Second, we avoided analyzing risks in the identification session. We simply listed candidate risks and tried not to analyze or discuss them in any detail.

The sequential application of methods may also have caused a multiple treatment effect: the latter, Riskit method application may have been influenced by earlier analysis done using the comparison method. We tried to control this threat by carrying out the latter risk analysis as independently from the comparison method analysis as possible: we asked the project manager not to think about the results of the comparison method, we used the original list of risks as a starting point, and we facilitated the Riskit risk analysis session according to the Riskit method. Two observations lead us to believe that multiple treatment effect did not occur or was minimal: the risks selected for analysis were different and the method user clearly indicated that the analysis processes were so different that he himself did not observe any effect, the Riskit method seemed to have immersed the user so that he "forgot" his previous analysis.

The interviews and associated questions may have posed some construct validity and instrumentation threats in the study. As the Riskit method sessions were observed and the session notes reviewed shortly after each session, the Riskit observations were not affected by this threat. The interview sessions potentially may have been affected by this threat. As we discussed in section 5.3, the main research constructs were explicitly defined we have reported the resulting data in detail later in this report. It should be noted that many of the original metrics and questions turned out not to be applicable in the study or produced no responses from the method user. In retrospect, these questions and metrics seemed to have been the result of our attempts to "over-measure" the study.

The fact that we facilitated the Riskit risk analysis session may have caused a different kind of bias in the results, i.e., a construct validity threat similar to the Hawthorne effect (Cook and Campbell, 1979). It is plausible that the facilitator may have contributed to the analysis or that the mere presence of a facilitator and a scribe may have improved the performance of the project manager. We tried to minimize these effects by maintaining a strictly facilitating role in the analysis (we refrained from actually making any judgments or conclusions) and by strictly following the Riskit method. However, we cannot rule out the possibility that either our participation or unconscious contributions might have affected the analysis.

As the method developer was involved in the execution of the study and in the analysis of the results the experimenter expectancies may have influenced the results. We tried to control this threat by involving an experimenter whose sole research interest was in the experimental design and by documenting the case study results and outputs in detail in this report. This way outside, objective readers can evaluate possible bias independently.

Overall, we believe that our study design and arrangements prevented any significant validity threats to our results. The two most important validity threats relate to constructs used: the Riskit method changed two important parameters in risk analysis: the amount of effort spent and number of people participating. With the Riskit method more time was spent on risk analysis and risk control planning than with the comparison method. With the Riskit method there also was a member of the technical staff present in the analysis session present. While these factors quite likely had an effect on the results, they are also characteristics of the Riskit method. In other words, they were part of the control variable that we wanted to study.

6. Case Study Results

The following sections describe the progress of the risk analysis in the case study.

6.1 Goal Review

The goal review session was organized jointly for the two methods in the September 28 meeting (Figure 5), even though it is a step specified for the Riskit method (ver 0.10). The goals were listed in the session and the necessary information, as defined by the Riskit (ver 0.10) templates (Kontio, 1995) was documented. The resulting goal definitions are presented in Table 10. The goals were not formally articulated in the session in the format given in Table 10, however. This was done intentionally in order to minimize the possible influence to the comparison method that did not call for an explicit review of goals.

Goal/constraint	Stakeholders	Measurement unit	Target value	Direction of increasing utility
Schedule	CSC NASA	calendar date	Dec. 15, 1995	earlier is better
Effort	CSC NASA	staff months before testing	2.5	less is better
Functionality	CSC NASA	number of functions	satisfy the specification	more functionality is better
Quality	CSC NASA	number of errors in testing	7 (3.3/KLOC)	fewer errors is better
Productivity	CSC	LOC/hr	2.8	higher is better
Standards compliance	CSC	N/A	compliance to standards	N/A

Table 10: MCU project goals

The goal review was done in the beginning of a meeting that continued as a risk identification session.

6.2 Risk Identification

The project manager and two members of the technical staff participated in the risk identification session, as well as the experiment organizers, J. Kontio acting as a facilitator and

1. Unstable requirements
2. Mismatch between specification and actual requirements
3. Mismatch between user interface (UI) tool and required functionality. May have to change design to use the user interface tool.
4. Not familiar with tool (UI or other)
5. Not experienced in GUI development
6. Compatibility with AMPT, reuse. AMPT-- Automated Maneuver Planning Tool. Long-term support utility, in planning phase. AMPT will have, among other things, same functionality as the MCU. They may want to reuse MCU, and MCU will likely be replaced by AMPT in the future.
7. Platform familiarity. No longtime experience with the platform: UNIX and C
8. External interface problems. The tools and programs providing the input to MCU are changing. This may cause change of file format.
9. Staff reassigned. Customer may give directions to shift priorities, e.g., to the mainframe rehosting project. In this case all the goals will be changed.
10. Lose personnel.
11. Bottlenecks resources. Workstations and network may be occupied since more and more of the work move from mainframes.
12. Customer contact availability. If customer contact person changes or he is not available, important decisions may be postponed or have to be made by project manager.
13. Personnel turnover. Personnel may be relocated to other tasks (rehosting project) and be replaced by people with less experience.
14. Overhead of experiment Lots of time spent in meetings and doing extra tasks due to the experiment.
15. Unrealistic effort estimation. Effort estimation is not so accurate in preliminary design phase.
16. Not following standards. Not meeting company project standards
17. Different acceptance criteria between customer and vendor.
18. Unanalyzed acceptance of requirements changes
19. TBDs in the specification. Things "to be defined", requirements that are left unspecified.

Table 11: Risks identified

H. Englund as the observer and scribe.

We used three approaches in the risk identification session. First, we carried out a free-format brainstorming session where participants were allowed to name any risk and it was recorded on the white board. There was little discussion on the items. This session identified the first 14 risks listed in Table 11. These risks were identified in about 25 minutes.

After the free brainstorming step the facilitator asked participants to look at the project goals and consider possible threats to them. This goal-driven analysis produced the risks 15 and 16 (Table 11) and lasted less than ten minutes.

Finally, we used the Taxonomy-Based Questionnaire (TBQ) of the SEI (Carr et al. 1993) and went through the relevant questions to check whether they would prompt participants to recognize any additional risks. This yielded risks 17-19 and one additional goal that was not identified in the initial goal review session. The TBQ session lasted one hour.

The session concluded with a list of identified risks. They were not classified into the risk analysis graph as this would have had an unintended effect on the comparison method.

6.3 The Comparison Method

The project manager participating in the project carried out the comparison method risk analysis on his own. He was given the list of identified risks and he selected three risks to work on:

- UI tool integration (composite of risks 3 and 4 in Table 11)
- AMPT compatibility (risk 6 in Table 11)
- inadequate staffing (composite of risks 9 and 10 in Table 11)

Note that he consolidated some risks together for his analysis. These risks, in his judgment, were the most important ones to consider, based on their impact, probability and possibilities for control. The risks and their selected risk controlling actions produced by the comparison method are listed in Table 12.

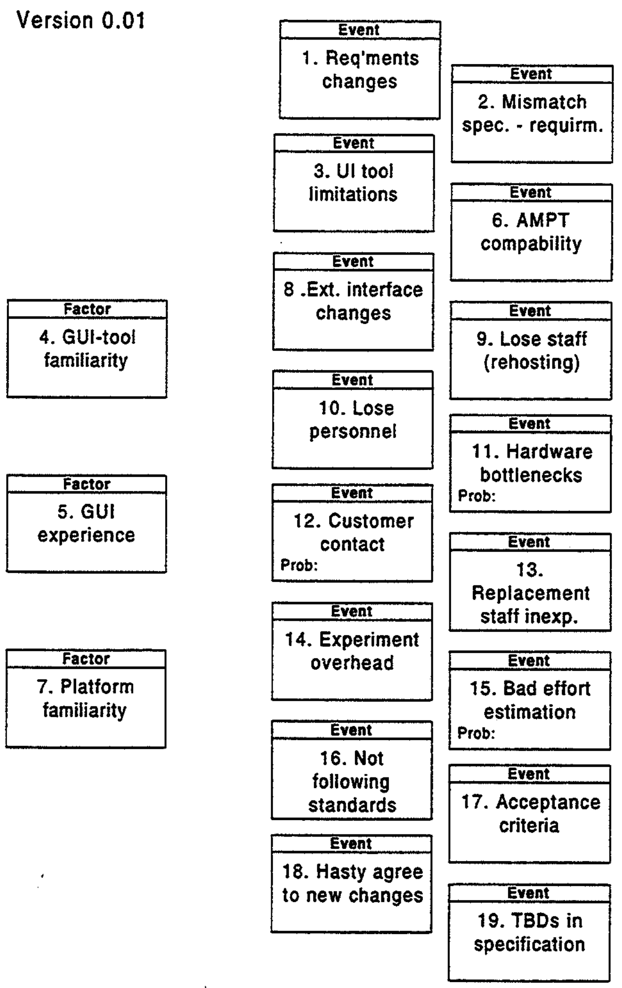


Figure 6: Results of the risk element classification

Risk	Selected controlling actions	Proposed but not implemented controlling actions
UI tool integration (containing risk 3)	A_{C1}: Apply lessons learned from previous UI tool integration (unique) A_{C2}: Have UI personnel review design and implementation products (overlapping A_{R6})	A_{C5}: Add staff or negotiate for more time (same as A_{R8})
AMPT compatibility (unique)	A_{C3}: Present detailed design walkthrough to analysts to ensure a consistent understanding of design approach (same as A_{R2})	A_{C6}: Estimate cost and schedule impact and provide to ATR (unique)
Inadequate staffing (overlapping)	A_{C4}: Finish all unit designs before coding (unique)	A_{C2}: <repeated> A_{C5}: <repeated> A_{C7}: Have available staff work extra hours (unique)

Note that actions A_{C2} and A_{C5} appear twice in the table but are each counted as one action.

Table 12: Risk controlling actions produced by the comparison method

The method user spent two hours on risk analysis using the spread-sheet based tool, which is normal for the type of projects he has been involved with. An example of the comparison method's output is presented in Table 9.

6.4 Riskit Method

6.4.1 Risk Analysis

After the risk identification session we grouped the identified risks (Table 11) into risk factors and risk events and placed them on the Riskit analysis graph, resulting in a graph presented in Figure 6. This was done without project manager's participation and was a relatively straight-forward task, taking approximately an hour to complete. Note that the names and meanings of some risks were slightly modified during this process to avoid ambiguity and overlapping of risks. The numbering used in Figure 6 refers to numbering used in Table 11 to maintain traceability of elements.

After the initial classification of risk elements into the Riskit analysis graph, we extended the graph by adding the other elements that belong to the graph. An initial version of this positioning was done without project manager to save his time. The results of this analysis are presented in Figure 7.

The resulting graph was used as a starting point in the Riskit risk analysis session with the project manager and one member of the technical staff. The graph was first reviewed and changes were made to correspond to project manager's perception of the situation. This resulted in the following changes:

- Risk event "AMPT compatibility" (risk number 6 in Table 11) was dropped because it

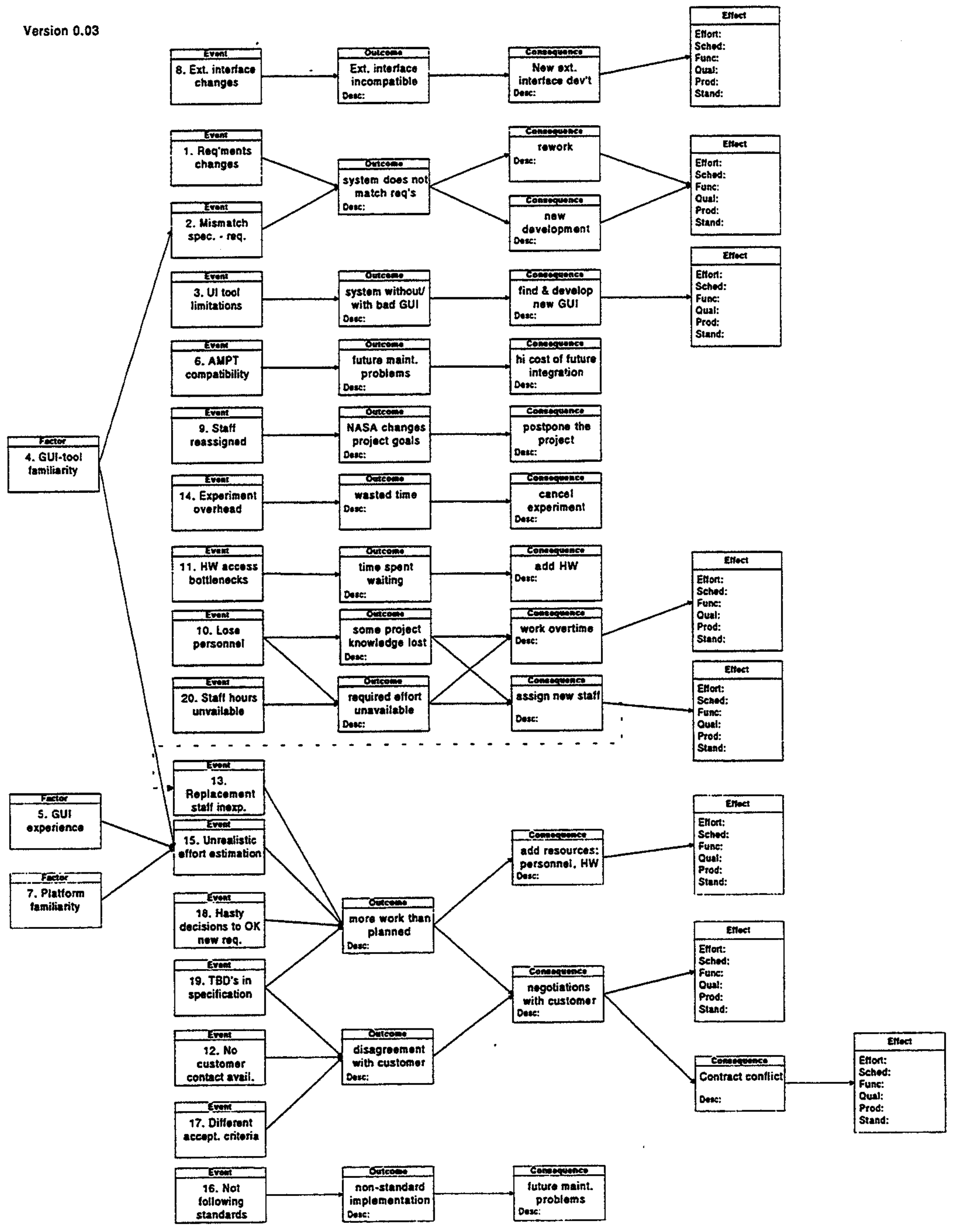

Version 0.03
Event
8. Ext. interface changes
Outcome
Ext. interface incompatible
Desc:
Consequence
New ext. interface dev't
Desc:
Effect
Effort:
Sched:
Func:
Qual:
Prod:
Stand:
1. Req'ments changes
2. Mismatch spec. - req.
system does not match req's
rework
new development
3. UI tool limitations
system without/ with bad GUI
find & develop new GUI
6. AMPT compatibility
future maint. problems
hi cost of future integration
9. Staff reassigned
NASA changes project goals
postpone the project
Factor
4. GUI-tool familiarity
14. Experiment overhead
wasted time
cancel experiment
11. HW access bottlenecks
time spent waiting
add HW
10. Lose personnel
some project knowledge lost
work overtime
20. Staff hours unavailable
required effort unavailable
assign new staff
13. Replacement staff inexp.
5. GUI experience
15. Unrealistic effort estimation
7. Platform familiarity
add resources: personnel, HW
18. Hasty decisions to OK new req.
more work than planned
19. TBD's in specification
negotiations with customer
12. No customer contact avail.
disagreement with customer
Contract conflict
17. Different accept. criteria
16. Not following standards
non-standard implementation
future maint. problems

Figure 7: The result of initial risk analysis using the Riskit analysis graph

was not any of the identified goals or stakeholders in the project.

- Risk event "staff reassigned" (risk number 9 in Table 11) was dropped because this would occur as a customer requirement and is therefore not a risk to any stakeholder.
- Risk event "no customer contact available" (risk number 12 in Table 11) was dropped because this would be an unrealistic event (i.e., having infinitely small probability and if occurred, not being a risk to the project contractor).
- Risk event "replacement staff inexperience" (risk number 13 in Table 11) was dropped because of dropping risk 9 ("staff reassigned") due to their causal relationship.
- Risk event "TBDs in the specification" (risk number 19 in Table 11) was dropped as the specification document had been reviewed and there had not been any TBDs.
- A new risk event was added: "staff hours not available". This was done to separate the scenarios where staff members leave the project (risk event 10 in Table 11) and when their time becomes unavailable, e.g., because of the prioritization of other projects. This risk event was numbered as risk 20 in Figure 7.

Risk event	Classification (High/Medium/Low)	Ranking	
		Staff member	Project manager
3. UI tool limitations	High	2	1
1. Requirements changes	High	1	2
2. Mismatch spec. - req.	High	3	3
8. Ext. interface changes	Medium	7	5
15. Unrealistic effort estimation	Medium	6	6
6. AMPT compatibility	Medium	8	7
20. Staff hours unavailable	Medium	9	8
11. HW access bottlenecks	Medium	5	9
17. Different acceptance criteria	Low	12	10
10. Lose personnel	Low	10	11
19. Hasty decisions to OK new req.	Low	14	12
14. Experiment overhead	Low	11	13
16. Not following CSC standards	Low	13	14

Table 13: Risk event probability classification and rankings

Probabilities of risk events were estimated next. This was done using the following approach:

- Each risk event was categorized into as "high", "medium" or "low" using a discussion and consensus opinion of the project manager and the member of the technical staff.
- Both project manager and the member of the technical staff independently ranked risks from most likely to least likely.
- Rankings of the two individuals as well as the results of the classification approach were compared to spot any inconsistencies.

The results of this estimation process are presented in Table 13. As the Table 13 shows, all three estimation approaches yielded results that are reasonably close to each other. Thus, we assumed that we had obtained a reliable ranking of risks and used the results of the high-medium-low classification in the remainder of the analysis.

The next step was to review and refine each risk scenario and estimate the impact of each scenario to project goals. The impacts were quantified or described verbally as they affected the project goals. We then asked project manager to classify the "pain", i.e., utility loss, of each scenario into "High", "Medium" and "Low". The results of this activity are presented in Figure 8, together with other results of the analysis. The pain rankings are marked as the last attribute in the boxes representing the effects of each scenario (the right-most boxes in Figure 8). Scenarios with high pain and events with high probability have been highlighted by darkening the banner of the corresponding boxes.

6.4.2 Plan Risk Control

The final step in the Riskit method was to identify risks that should be controlled and propose some risk controlling actions. For risk control planning activity we selected the event-scenario combinations that met any of the following conditions:

- the event-scenario combination had both high probability and high pain;
- the event-scenario combination had high probability associated with medium pain; or
- the event-scenario combination had high pain associated with medium probability.

Note that we used our judgment in interpreting the above criteria and also reviewed all other scenarios to determine whether they would deserve further consideration even though they did not meet the above criteria.

The risk scenarios that were selected for risk control planning are listed in the left hand column of Table 14. For each risk scenario we tried to identify possible risk controlling action that could be taken. As a tool in this process we used the risk controlling action taxonomy (Kontio, 1995) to act as a checklist for proposing controlling actions.

The possible actions and their impacts on the risks were also documented in the Riskit analysis graph, as shown in Figure 9. The risk controlling actions are marked as ovals in Figure 9.

Risk	Selected controlling actions	Proposed but not implemented controlling actions
8. Ext. interface changes (unique)	A_{R1}: Show designs to the customer for approval (unique)	
1. Requirements changes (unique) 2. Mismatch spec. - req. (unique)	A_{R2}: Verify that walk-through reviews are done well (same as A_{C3})	A_{R11}: Document all requirements changes in detail (unique)
3. UI tool limitations (subsumed to "UI tool integration")	A_{R3}: Use the alternative UI tool (unique) A_{R4}: Train somebody in the alternative UI tool immediately (unique) A_{R5}: Make sure alternative UI tool experts are available (unique) A_{R6}: Consult current UI tool experts to check whether it satisfies the project needs (overlapping A_{C2})	
15. Unrealistic effort estimation (overlapping)	A_{R7}: Review estimates at walk-through review (unique) A_{R8}: Create slack (effort and schedule) with customer (same as A_{C5})	
10. Lose personnel (unique) 20. Staff hours unavailable (unique)	A_{R9}: Agree on project priority with other managers (unique) A_{R10}: Coordinate staff allocation with other managers (unique)	A_{R12}: Document well (unique)

Table 14: Risk scenarios selected for risk control planning and corresponding actions

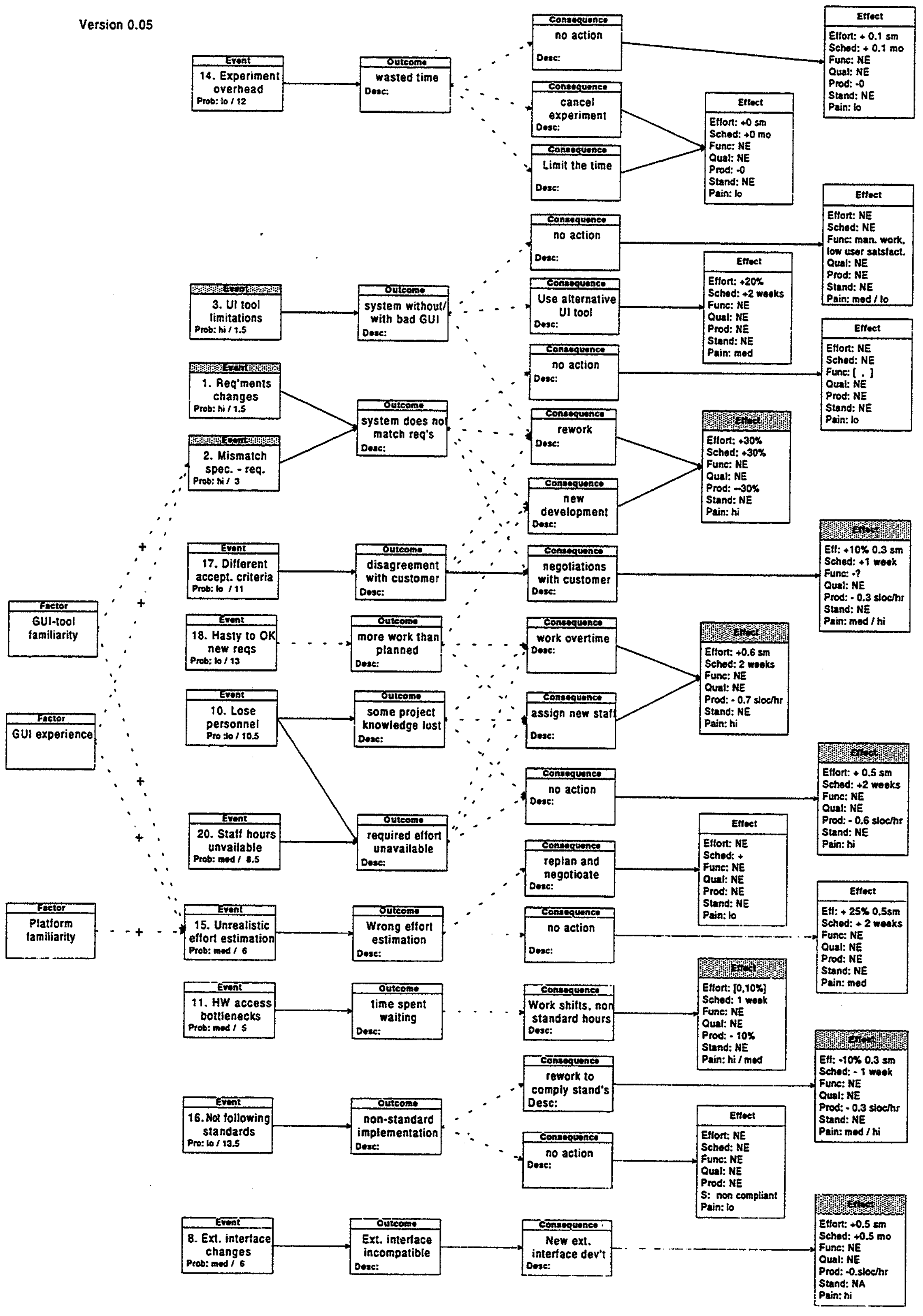
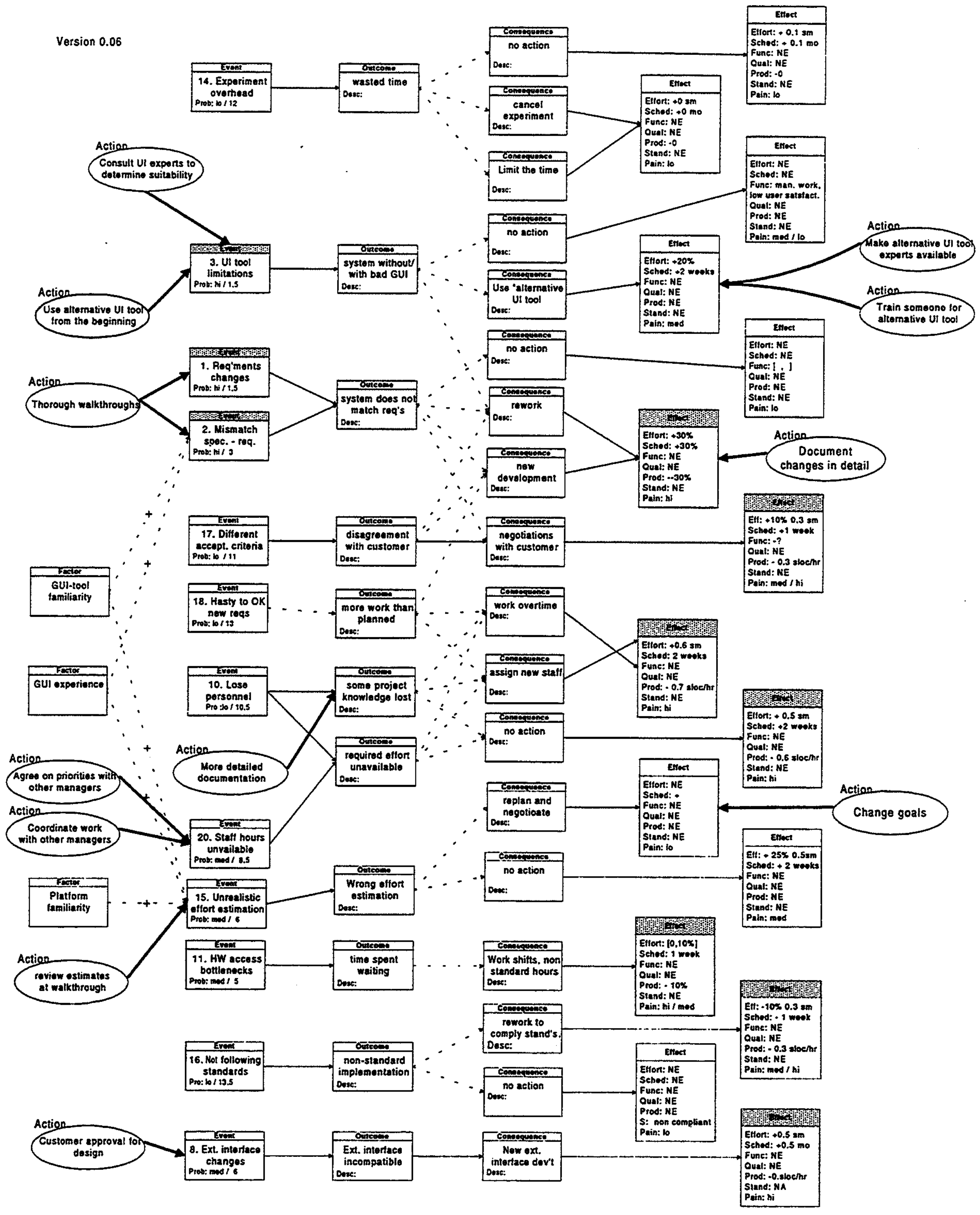


Figure 8: Results of risk analysis step



Version 0.06
Event 14. Experiment overhead Prob: lo / 12
Outcome wasted time Desc:
Consequence no action Desc:
Consequence cancel experiment Desc:
Consequence Limit the time Desc:
Effect Effort: + 0.1 sm Sched: + 0.1 mo Func: NE Qual: NE Prod: -0 Stand: NE Pain: lo
Effect Effort: +0 sm Sched: +0 mo Func: NE Qual: NE Prod: -0 Stand: NE Pain: lo
Action Consult UI experts to determine suitability
Event 3. UI tool limitations Prob: hi / 1.5
Action Use alternative UI tool from the beginning
Outcome system without/ with bad GUI Desc:
Consequence no action Desc:
Consequence Use 'alternative UI tool Desc:
Effect Effort: NE Sched: NE Func: man. work, low user satisfact. Qual: NE Prod: NE Stand: NE Pain: med / lo
Effect Effort: +20% Sched: +2 weeks Func: NE Qual: NE Prod: NE Stand: NE Pain: med
Action Make alternative UI tool experts available
Action Train someone for alternative UI tool
Action Thorough walkthroughs
Event 1. Req'ments changes Prob: hi / 1.5
Event 2. Mismatch spec. - req. Prob: hi / 3
Outcome system does not match req's Desc:
Consequence no action Desc:
Consequence rework Desc:
Consequence new development Desc:
Effect Effort: NE Sched: NE Func: [,] Qual: NE Prod: NE Stand: NE Pain: lo
Effect Effort: +30% Sched: +30% Func: NE Qual: NE Prod: --30% Stand: NE Pain: hi
Action Document changes in detail
Event 17. Different accept. criteria Prob: lo / 11
Outcome disagreement with customer Desc:
Consequence negotiations with customer Desc:
Effect Eff: +10% 0.3 sm Sched: +1 week Func: -? Qual: NE Prod: - 0.3 sloc/hr Stand: NE Pain: med / hi
Factor GUI-tool familiarity
Event 18. Hasty to OK new reqs Prob: lo / 13
Outcome more work than planned Desc:
Consequence work overtime Desc:
Consequence assign new staff Desc:
Effect Effort: +0.6 sm Sched: 2 weeks Func: NE Qual: NE Prod: - 0.7 sloc/hr Stand: NE Pain: hi
Factor GUI experience
Event 10. Lose personnel Pro :lo / 10.5
Outcome some project knowledge lost Desc:
Consequence no action Desc:
Effect Effort: + 0.5 sm Sched: +2 weeks Func: NE Qual: NE Prod: - 0.6 sloc/hr Stand: NE Pain: hi
Action More detailed documentation
Outcome required effort unavailable Desc:
Action Agree on priorities with other managers
Action Coordinate work with other managers
Event 20. Staff hours unvailable Prob: med / 8.5
Consequence replan and negotiate Desc:
Effect Effort: NE Sched: + Func: NE Qual: NE Prod: NE Stand: NE Pain: lo
Action Change goals
Factor Platform familiarity
Event 15. Unrealistic effort estimation Prob: med / 6
Outcome Wrong effort estimation Desc:
Consequence no action Desc:
Effect Eff: + 25% 0.5sm Sched: + 2 weeks Func: NE Qual: NE Prod: NE Stand: NE Pain: med
Action review estimates at walkthrough
Event 11. HW access bottlenecks Prob: med / 5
Outcome time spent waiting Desc:
Consequence Work shifts, non standard hours Desc:
Effect Effort: [0,10%] Sched: 1 week Func: NE Qual: NE Prod: - 10% Stand: NE Pain: hi / med
Event 16. Not following standards Pro: lo / 13.5
Outcome non-standard implementation Desc:
Consequence rework to comply stand's. Desc:
Consequence no action Desc:
Effect Eff: -10% 0.3 sm Sched: - 1 week Func: NE Qual: NE Prod: - 0.3 sloc/hr Stand: NE Pain: med / hi
Effect Effort: NE Sched: NE Func: NE Qual: NE Prod: NE S: non compliant Pain: lo
Action Customer approval for design
Event 8. Ext. interface changes Prob: med / 6
Outcome Ext. interface incompatible Desc:
Consequence New ext. interface dev't Desc:
Effect Effort: +0.5 sm Sched: +0.5 mo Func: NE Qual: NE Prod: -0.sloc/hr Stand: NA Pain: hi
+

Figure 9: Final results of the Riskit process – risk controlling actions

7. Case Study Analysis

In the following sections we present the case study data and analyze the data with respect to the case study goals we had (presented in section 5.3). As we indicated earlier by Figure 5, the information about the methods was collected through observation, analysis of the artifacts produced, and interviews.

7.1 Qualitative Characterization of the Methods

We used questionnaires and interviews to inquire the method user's experiences and opinions about the two methods. All questions were sent to him in advance by email, he replied to the questions and we held an interview session to discuss his responses in more detail. The following represents the method user's responses to the main questions asked from him[10]:

Are the methods easy to understand and use?

"Riskit is easier to get started with – method of identifying risks is better defined. [Comparison method] provides better risk summary. Riskit follows a more scientific way of determining a risk's likelihood of occurrence. [Using the comparison method] we guess at the probability.
Although the Riskit method had a better defined process, it would have been difficult to apply without facilitation.

"Comparison method is easier to use, it has a simple, well-defined input format."

Comment the output format of the methods.

"[The comparison method] quantifies risks and provides a good textual summary of them – good for individual risks but does not provide a high-level analysis of all risks, as Riskit does. Riskit has a complex and busy graph, but ranks risks well and presents them in a good summary table."

"[The comparison method] cannot highlight the most important risks, Riskit does this clearly and effectively – perhaps its greatest asset."

What is your opinion of the usability and practical value of the Riskit method?

"The method is usable and practical, it is a better risk management method, a more complete one. It may be better utilized in longer, riskier projects.

"Riskit is certainly more thorough, [the comparison method] may find too few risks."

"Riskit takes more resources. The [Riskit analysis] graph was too big."

How much confidence did you have in the risk analysis results produced by the Riskit method and why?

"I did have confidence in what it produced because of the process that was used, because of its more complete analysis of risks and because of the risk ranking process it used."

[10] While most of the answers are verbatim quotes from the email responses, some the answers have been combined from more than one question, as they were addressed in different parts of the follow-up interview.

Which method, or which combination of them, would you recommend for use?

> *"Apply the brainstorming [risk identification] and risk ranking approach, as these do not increase the costs by much. Try out the complete Riskit method on selected projects. Use the [comparison method] for documenting each risk."*

We have evaluated the qualitative responses from three perspectives: ease of use, input and out formats, and practical value of the method.

It is difficult to compare the ease of learning and ease of use of the methods. While the Riskit method has some underlying, more complex principles in it, it is better documented and its application was facilitated in the case study. On the other hand , the comparison method had been used by the method users for several years and they had initially received training on it. However, given that the method user was able to apply and understand the method without any training in a facilitated session leads us to suggest that there are no significant differences in the ease of learning and ease of use between the methods.

Regarding the input and output formats of the methods, the comparison method seems to have an advantage in entering information in it – it has clearly defined items that need to be entered into the tool. It also seems to provide good summaries of each individual risk, although this observation may be largely due to the method user's familiarity of the output. The Riskit method seems to provide a better overview of the risk situation in the project and highlights most important risks well.

The method user expressed clearly more confidence in the results produced by the Riskit method. He saw it as a more thorough and complete method. In particular, he valued its risk analysis and ranking approach. He also indicated an interest in applying or experimenting with the method, or its components, in future projects.

7.2 Cost and Time

The cost and effectiveness of the method was analyzed based on the data presented in Table 15. As we discussed earlier, the risk identification session was shared between the methods. Thus, it is not straight-forward to sum up the effort used by the methods as a separate risk identification session is not normally part of the comparison method. If the risk identification session is included in the totals of both methods, the comparison method consumed 11 person hours and the Riskit method 20 person hours. If the risk identification step is excluded, corresponding figures are 3 and 12 hours, as Table 15 shows. It would be even plausible to compare the comparison method's 3 hours against the total of Riskit method's 20 hours, as they actually represent approximations of what "normally" would have happened without the experimental arrangements.

	Study management	Risk identification	Comparison method	Riskit method	**Total**
MCU project manager	6	2	3	3.5	14.5
MCU technical staff	0	4	0	3.5	7.5
UMD study personnel	NA	2	0	5	7
Total	6	8	3	12	29

Table 15: Study effort distribution in person-hours[11]

7.3 Granularity, Coverage and Accuracy

We have analyzed the granularity and coverage of the two methods by defining a set of specific metrics for risks and controlling actions that were produced. We realized that a mere counting of risk or controlling actions fails to account for the granularity and coverage of respective items. Thus, we use the following additional metrics to characterize the methods:

- Number of *same risks/actions* produced by the method, i.e., risks/actions that are judged to be same or very similar to a risk described by the other method.
- Number of *unique risks/actions* produced by the method, i.e., risks/actions that have not been identified by the other method and which do not overlap or are subsumed by other method's risks/actions.
- Number of *subsumed risks/actions*, i.e., risks/actions that are subsets of risks/actions identified by the other method.
- Number of *containing risks/actions*, i.e., risks/actions that include one or more of the risks/actions identified by the other method.
- Number of *overlapping risks/actions*, i.e., risks/actions that have some similarities but do not belong to any of the previous categories.

We used the above definitions to classify the risks selected for risk control planning and the controlling actions that were produced. Table 16 presents the metrics produced by the analysis of coverage and granularity of risks that were selected for risk control planning for each method.

When analyzing the risks we chose to compare the risks that were selected to risk control planning. The list of identified risks could not be used because the identification session was a joint session for both methods. We have marked the classification of each risk in Table 12 and Table 14 in parenthesis in the right-hand column, e.g., the text "(unique)" indicates that the risk thus marked was a unique risk for the method.

As Table 16 shows, the Riskit method analyzed more risks than the comparison method. However, direct count of analyzed risks is not a meaningful indicator of the differences between

[11] Some clarifications are necessary in order to interpret the data in Table 15 correctly. First, the item "study management" includes preparation and planning for the study, data collection and creating additional documentation for the purposes of the study. Consequently, we have estimated the editing work on the Riskit Analysis Graphs to have taken 1.5 hours. Second, the UMD personnel's time for the study management task was not accurately measured (thus the "NA" item in the corresponding cell).

methods. The difference between the number of unique risks produced by the methods is more interesting: Riskit analyzed five unique risks compared to one of the comparison method's.

Metric	Comparison Method	Riskit Method
same risks	0	0
unique risks	1	5
subsumed risks	0	1
containing risks	1	0
overlapping risks	1	1
Total	3	7

Table 16: Coverage and granularity metrics for risks analyzed

The comparison method's "UI tool integration" was a containing risk to Riskit method's subsumed risk "UI tool limitations". As there was only one pair of containing/subsumed risks we cannot make any conclusions from this particular data. In general, however, a high number of subsumed risks indicates finer granularity and, if the subsumed risks cover all or most of the containing risk, this can be considered more precise description of the risks in a situation.

Risks "Inadequate staffing" (comparison method) and "Unrealistic effort estimation" (Riskit) were considered overlapping.

Given the data about the analyzed risks in the case study, the risk management methods seem to differ in their coverage. If we assume that the union of analyzed risks represents the "real" risks in the situation and count same, subsumed, containing and overlapping risks as one instance each, the *risk coverage ratios* for each method can be calculated as follows:

- comparison method: 3/8 = 38%
- Riskit method: 7/8 = 88%

We would like to emphasize that due to the assumptions and interpretations made during the above analysis, the above figures should be interpreted conservatively.

We repeated a similar process for risk controlling actions that were produced. Table 17 presents this data. The classification of actions into our categories have been marked in Table 12 and Table 14 in parenthesis in the middle and left-hand columns.

As Table 17 shows, the Riskit method proposed more controlling actions than the comparison method. It also produced a higher number of unique controlling actions. Using the same principle as above, the coverage ratios for risk controlling actions are as follows:

- comparison method: 7/16 = 44%
- Riskit method: 12/16 = 75%

The above figures suggest that the coverage of actions proposed by the Riskit method is higher, i.e., it proposed a wider range of actions to be considered for implementation.

Metric	Comparison Method	Riskit Method
same controlling actions	2	2
unique controlling actions	4	9
subsumed controlling actions	0	0
containing controlling actions	0	0
overlapping controlling actions	1	1
Total	7	12

Table 17: Coverage and granularity metrics for controlling actions analyzed

We assess the accuracy of the methods indirectly through the risk controlling actions that were actually taken in the project, vs. the actions that were planned. The rationale for this metric is that we assume that the project manager, as a rational decision maker, will take the necessary cost efficient action in the project as further information about the project becomes available. Any action that was planned but not implemented indicates that (i) risk situation changed after the action was planned, (ii) the action did not address a big enough risk to justify it, or (iii) the action was not considered effective enough to justify its costs.

According to the project manager, there were no recognizable changes in the risk situation after the risk control planning and taking the action. Thus, we are using the ratio

Risk controlling action accuracy ratio = number implemented actions / number of planned actions

as an indicator of the accuracy of the results produced. Below are the corresponding ratios for the two methods:

- comparison method: 4/9 = 44%
- Riskit method: 10/12 = 83%

These figures lead us to suggest that the Riskit method was more effective in proposing accurate risk controlling actions, i.e., it proposed actions that were considered worth implementing in the project.

It is also noteworthy to highlight that the Riskit method addressed a risk that actually realized: the UI tool was considered unsuitable for the project and an alternative tool was used. The risk controlling action that was taken mitigated the potential negative impact of this risk in advance. The comparison method addressed a containing risk ("UI tool integration") for the same risk but did not recognize the controlling actions that directly mitigated the risk.

7.4 Feasibility

Our first goal was to investigate the feasibility of the Riskit method in industrial context (page 21). The criteria we defined for determining feasibility were met. First, the method produced intended results (identified risks, ranked them and proposed controlling action). Second, the overall effort spent on the use of the method was 12 hours. This is 20% of the management time of the project, and 2% of the total effort in the project, i.e., well within the effort limit proposed by Ropponen's survey (Ropponen, 1993). Third, as we reported in section 7.1, the method user gave a positive assessment of the method with respect to its thoroughness, indicated a higher level of confidence in its results and considered its risk ranking approach more sound.

Based on these findings we conclude that the Riskit method was a feasible approach in the case study project. We would like to point out that the validity threats described in section 5.5 prevent us from generalizing this conclusion outside this project with confidence. However, none of the validity threats directly contradicts such generalization, either.

7.5 Efficiency

The evaluation of the efficiency of the method was based on the data obtained in the characterization process described in sections 7.1 to 7.3. We Defined two derived metrics to characterize the efficiency. The first one, *risk coverage efficiency index*, utilizes *the risk coverage ratio*, defined in section 7.3, and the effort used for risk management using the method. The rationale for this metric is that the risk coverage ratio represents the best available information of the coverage of all relevant risks in a situation. Dividing this by the effort expended to reach that coverage gives an indication of a method's efficiency in risk analysis.

The second metric, *risk controlling action efficiency index*, utilizes the concept *risk controlling action accuracy ratio*, defined in section 7.3, and effort for the method. The rationale for this method is that the total of implemented actions represent the best available information about the correct action to take in a situation. As the *risk controlling action accuracy ratio* numerically describes how well the method was able to produce the ideal set of actions, normalizing the *risk controlling action accuracy ratio* by effort expended gives an indication of risk controlling action efficiency.

The effort used in these calculations was the method's total effort without the shared risk identification session (see Table 15). The two metrics and corresponding data are presented in Table 18.

Metric	Comparison Method	Riskit Method
risk coverage efficiency index = risk coverage ratio / risk management effort	38% / 3 = 13%	88% /12 = 7%
risk controlling action efficiency index = risk controlling action accuracy ratio / risk management effort	44% / 3 = 15%	83% / 12 = 7%

Table 18: Efficiency metrics used in the case study analysis for the two methods

As the results of Table 18 show, the comparison method is more efficient in analyzing risks and proposing actions. This is not surprising, since the comparison method analyzed fewer risks and proposed fewer actions. It is quite likely that the most obvious risks and actions are the least costly to produce. The relative efficiency decreases as more risks and actions are analyzed and proposed. Consequently, we do not think that efficiency is an effective metric to evaluate a risk management method.

7.6 Effectiveness

The evaluation of effectiveness of the methods is to consider whether the unique risks produced by the methods resulted in actions that were actually implemented and whether these actions were unique. From this perspective, the comparison method produced one unique risk ("AMPT compatibility") whose controlling action (A_{C3}) was the same as one of Riskit method's implemented actions (A_{R2}). Riskit, on the other hand, produced five unique risks and seven unique risk controlling actions (see Table 14). This seems us to suggest that while the marginal efficiency if the Riskit method was lower, its overall effectiveness was higher.

8. Conclusions

The purpose of exploratory case studies is to provide real-world data, experience and feedback to in order to identify problems, interesting relationships or concepts, or simply to provide a basis for ideas and innovation. From this perspective the case study was an exploratory one – it gave us insights to the issues in risk management and how the Riskit method addresses these issues. A secondary goal was to investigate the feasibility of the method.

The case study had a major impact on the further development of the method. The Riskit Analysis Graph was simplified and revised, the Riskit process description subsequently detailed, and several application guidelines were identified.

The case study also served to characterize and evaluate the method. Based on the analysis of our experiences we have concluded that Riskit is a feasible method in an industrial context. The Riskit method seems to cover risks comprehensively and propose risk controlling actions accurately. Furthermore, it seems to provide a good overall view of risks and its results seems to be credible. However, it seems to consume more resources than the default method. It seems that Riskit may be a method to be applied when projects are large or when risks are high. Small, low risk projects may be better off with simpler and less costly risk management approaches.

Given the limited size of the case study and limited number of data points available, it is too early to generalize these findings with any confidence. However, they indicate that the method has several potentially significant benefits.

9. References

Anonymous (1992) *The American Heritage Dictionary of the English Language*, 3rd edn. U.S.A. Microsoft Bookshelf/Houghton Mifflin Company.

Basili, V.R. (1992) Software Modeling and Measurement: The Goal/Question/Metric Paradigm. CS-TR-2956, College Park, MD: University of Maryland.

Basili VR, Caldiera G, McGarry F, Pajerski R, Page G, and Waligora S. The Software Engineering Laboratory - an Operational Software Experience Factory. 370 p. -381.(1992)

Basili, V.R., Caldiera, G. and Rombach, H.D. (1994) Goal Question Metric Paradigm. In: Marciniak, J.J. (Ed.) *Encyclopedia of Software Engineering*, pp. 528-532. New York: John Wiley & Sons

Basili, V.R. and Green, S. (1994) Software Process Evolution at the SEL. *IEEE Software* **11**, 58-66.

Boehm, B.W. (1981) *Software Engineering Economics*, Englewood Cliffs, N.J. Prentice Hall.

Boehm, B.W. (1989) *Tutorial: Software Risk Management*, IEEE Computer Society Press.

Campbell, D.T. and Stanley, J.C. (1963) *Experimental and Quasi-Experimental Designs for Research*, Boston: Houghton Mifflin Co.

Carr, M.J., Konda, S.L., Monarch, I.A., Ulrich, F.C. and Walker, C.F. (1993) *Taxonomy-Based Risk Identification, SEI Technical Report SEI-93-TR-006*, Pittsburgh, PA: Software Engineering Institute.

Charette, R.N. (1989) *Software Engineering Risk Analysis and Management*, New York: McGraw-Hill.

Cook, T.D. and Campbell, D.T. (1979) *Quasi-Experimentation: Design & Analysis Issues for Field Settings*, Chicago: Rand McNally College Pub. Co.

Edgar, J.D. (1989) Controlling Murphy: How to Budget for Program Risk (originally presented in Concepts, summer 1982, pages 60-73). In: Boehm, B.W. (Ed.) *Tutorial: Software Risk Management*, pp. 282-291. Washington, D.C. IEEE Computer Society Press

Fenton, N.E. (1991) *Software Metrics A Rigorous Approach*, London: Chapman & Hall.

French, S. (1986) *Decision Theory: An Introduction to the Mathematics of Rationality*, Chichester: Ellis Horwood.

French, S. (1989) *Readings in Decision Analysis*, London: Chapman and Hall.

Friedman, M. and Savage, L.J. (1948) The Utility Analysis of Choices Involving Risk. *Journal of Political Economy* **56**, 279-304.

Garrabrants, W.M., Ellis III, A.W., Hoffman, L.J. and Kamel, M. (1990) **CERTS**: A Comparative Evaluation Method for Risk Management Methods and Tools. In: Anonymous *Proceedings of the Sixth Annual Computer Security Applications Conference*, pp. 251-257. Los Alamitos: IEEE Computer Society Press

Judd, C.M., Smith, E.R. and Kidder, L.H. (1991) *Research Methods in Social Relations*, 6th edn. Fort Worth: Harcourt Brace Jovanovich College Publishers.

Karolak, D.W. (1996) *Software Engineering Risk Management*, Washington, DC: IEEE.

Keeney, R.L. and Raiffa, H. (1976) *Decision with Multiple Objectives: Preferences and Value Tradeoffs*, New York: John Wiley & Sons.

Kontio, J. (1994) Software Engineering Risk Management: A Technology Review Report. PI_4.1, Helsinki, Finland: Nokia Research Center.

Kontio, J. (1995) Riskit: An Analytical Model for Risk Management. working paper.

Kontio, J. (1996) The Riskit Method for Software Risk Management, version 1.00. College park, MD: University of Maryland.

McGarry, F., Pajerski, R., Page, G., Waligora, S., Basili, V.R. and Zelkowitz, M.V. (1994) Software Process Improvement in the NASA Software Engineering Laboratory. CMU/SEI-94-TR-22, Pittsburgh, PA: Software Engineering Institute.

NASA (1995) Software Engineering Laboratory World Wide Web home page: http://fdd.gsfc.nasa.gov/seltext.html.

Ropponen, J. (1993) Risk Management in Information System Development. TR-3, Jyväskylä: University of Jyväskylä, Department of Computer Science and Information Systems.

Saaty, T.L. (1990) *The Analytic Hierarchy Process*, New York: McGraw-Hill.

Von Neumann, J. and Morgenstern, O. (1944) *Theory of Games and Economic Behavior*, Princeton: Princeton University Press.

An Empirical Study of Communication in Code Inspections

S9-61
415723
12p
360746

Carolyn B. Seaman
Institute for Advanced Computer Studies
Computer Science Department
University of Maryland
College Park, MD, USA
(301) 405-2721
cseaman@cs.umd.edu

Victor R. Basili
Institute for Advanced Computer Studies
Computer Science Department
University of Maryland
College Park, MD, USA
(301) 405-5497
basili@cs.umd.edu

ABSTRACT
This paper describes an empirical study which addresses the issue of communication among members of a software development organization. In particular, data was collected concerning code inspections in one software development project. The question of interest is whether or not organizational structure (the network of relationships between developers) has an effect on the amount of effort expended on communication between developers. Both quantitative and qualitative methods were used, including participant observation, structured interviews, generation of hypotheses from field notes, some simple statistical tests of relationships, and interpretation of results with qualitative anecdotes. The study results show that past and present working relationships between inspection participants affect the amount of meeting time spent in different types of discussion, thus affecting the overall meeting length. Reporting relationships and physical proximity also have an effect, as well as the point in the project that an inspection occurs. All but the last of these factors are organizational structure relationships. The contribution of the study is a set of well-supported hypotheses for further investigation.

Keywords
empirical study, communication, process, organizational structure, inspections

INTRODUCTION
Many factors which impact the success of software development projects still defy our efforts to control, predict, manipulate, or even identify them. One factor that has been identified [3] but is still not well understood is information flow. It is clear that information flow impacts productivity (because developers spend time communicating) as well as quality (because developers need information from each other in order to carry out their tasks effectively) [12]. It is also clear that efficient information flow is affected by the relationship between development processes and the organizational structure in which they are executed. A process requires that certain types of information be shared between developers and other process participants, thus making information processing demands on the development organization. The organizational structure, then, can either facilitate or hinder the efficient flow of that information. These relationships between general concepts are pictured in Figure 1.

The study described in this paper addresses the productivity aspects of communication. In particular, it empirically studies how process communication effort (the effort associated with the communication required by a development process) is influenced by the organizational structure (the network of relationships between developers) of the development project. In this paper, we examine the organizational structure of one particular project, the code inspection process used, and the time and effort associated with inspection meetings. We found that organizational attributes are significantly related to the amount of time inspection participants spend in different types of discussions. The aim of this study is not to test or validate hypotheses about relationships between these variables, but to explore what relationships might exist and try to explain those relationships. Our contribution, then, is a set of *proposed* hypotheses, along with an argument, in the form of supporting evidence, for their further examination.

Although the importance of efficient communication, and its relationship to organizational structure, is well supported in the organization theory literature [11, 5], it has not been adequately addressed for software development organizations. Communication has been identified as an important factor in how developers spend their time [12], and some organizational characteristics which affect its efficiency have been suggested [3, 8]. Some, but surprisingly little, of the "process" work in software engineering has dealt with information flow or organizational structure [1, 2, 13]. It has been postulated that informal communication is usually more

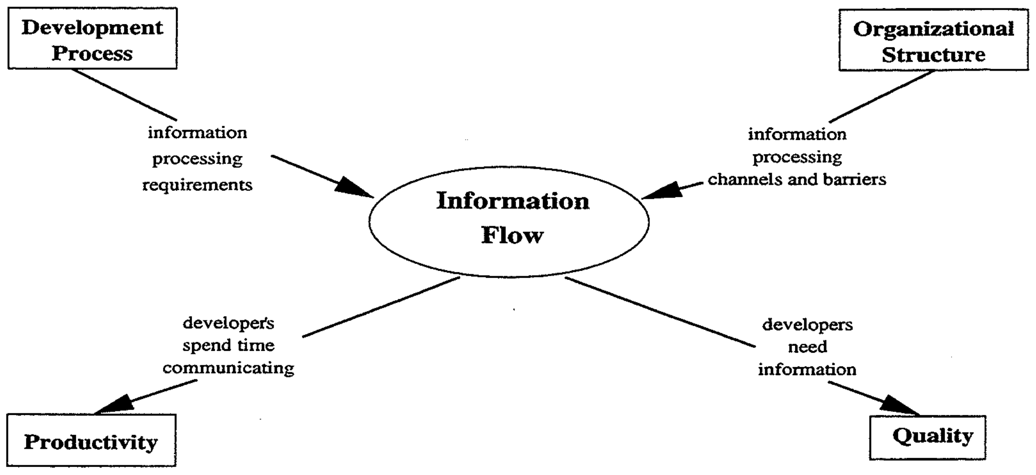

Development Process
information processing requirements
Organizational Structure
information processing channels and barriers
Information Flow
developer's spend time communicating
Productivity
developers need information
Quality

Figure 1: Relationships between concepts relevant to this work.

valuable than formal, interpersonal communication [9] (which includes what we have termed process communication). However, there is still a need for focused studies of the latter because, unlike informal communication, formal communication can be planned for and controlled, if we know the factors which can be manipulated to make it more efficient. Studies of human communication must, by definition, be empirical studies because they deal with non-analytical entities (i.e., people) which have few universally applicable laws or theories governing their behavior. Concepts such as communication, process, and organization must be studied where they occur, in real software development projects.

The study combines quantitative and qualitative research methods. Qualitative data is data represented as words and pictures, not numbers [6]. Qualitative methods are especially useful for generating, rather than testing, hypotheses. Quantitative methods are generally targeted towards numerical results, and are often used to confirm or test previously formulated hypotheses. They can be used in exploratory studies such as this one, but only where well-defined quantitative variables are being studied. We have combined these paradigms in order to flexibly explore an area with little previous work, as well as to provide compelling evidence to support the hypotheses we present.

STUDY SETTING

The project used for this study involved the development of mission planning software for NASA/Goddard Space Flight Center's Flight Dynamics Division (FDD). Much of the development was contracted to Computer Sciences Corporation (CSC). About 20 technical leads and developers (most from CSC) participated in the inspection process during the course of the study, although more participated in the project. The project began in early 1995, and the first release was scheduled for the summer of 1996 (as of this writing, it has not yet been delivered).

The two aspects of the project which are of interest are the development processes used (in particular the communication required by those processes), and the organizational relationships between the process participants.

This study focuses on the project's code inspection process. We relied initially on a written document, the *Code Inspection Procedure*, which defined the tailored inspection process for this project, including the relevant steps and roles. Throughout the study, however, we updated our understanding of the inspection process through observation and interviews. Inspections were conducted after unit test, before submitting the code to configuration control. Both code and unit test products (test plan and results) were inspected. It should be noted that some of the code inspected was produced by a code generator, which was used to write skeletons for all the classes developed, and some of the supporting code. The inspection meeting (the unit of analysis for this study) was one step in the inspection process. Inspection meeting participants included the "author", who had implemented and unit tested the C++ classes being inspected, the "moderator", a "code inspector", and a "test inspector". In some cases, more developers were assigned to inspect the code or test. All the observed inspections occurred at CSC, and involved mostly CSC personnel. The objective of the inspection meeting was to record defects which had been found by the inspectors during their preparation.

We have defined organizational structure as a network of organizational relationships. These relationships include management, or reporting, relationships and physical proximity. Information about these types of relationships came mainly from organizational documents, and was validated through interviews. Other organizational relationships we studied were past and present working relationships, which existed between many of the CSC and FDD personnel.

The organization and process information described above was modeled using a formalism called *Actor-Dependency models* (AD models) [15]. This model also included the actual data collected, and allowed us to automate some of the data analysis.

The process used to produce the AD model is known as *prior ethnography* [10], the practice of taking some time before data collection begins to become familiar with the study setting. In November and early December of 1995, the researcher attended team meetings, conducted open-ended interviews with several developers and managers, observed several inspection meetings (without recording data), and was introduced to all project participants. The goal was to become familiar with the setting, produce the AD model, and choose the relevant dependent and independent variables.

RESEARCH METHODS

The data and methods employed in this study are both quantitative and qualitative. The data collection phases were largely qualitative. The qualitative part of the data analysis began about halfway through data collection, and resulted in the generation of initial hypotheses. Quantitative analysis began with the coding of the data into numeric values corresponding to the study variables. Then various statistical techniques were used to discover relationships between the variables. This analysis was guided initially by the hypotheses generated by the qualitative analysis. Finally, qualitative analysis was also used after the initial statistical results were generated in order to help clarify and explain them. All of these techniques are discussed briefly in the following sections and are elaborated during the discussion of results.

Data Collection

The data for this study was collected between December, 1995, and April, 1996. The data collection procedures included gathering official documents, participant observation [14], and structured interviews [10].

The official documents of an organization are valuable sources of information because they are relatively available, stable, rich, and non-reactive, at least in comparison to human data sources [10]. Some of the documents which provided data for this study were:

- organizational charts
- process descriptions
- inspection data collection forms
- online newsgroup

Much of the data for this study was collected during participant observation of 23 inspection meetings. During the observations, the observer collected data on the lengths and topics of discussions, but did not play a direct role in the inspection process.

The other important data source was a set of interviews conducted with inspection participants. These interviews were semi-structured; each interview started with a specific set of questions, the answers to which were the objective of the interview. However, many of these questions were open-ended and were intended for (and successful in) soliciting other information not foreseen by the interviewer.

Data Analysis

Initial qualitative analysis on the data began about halfway through data collection. The first analysis was similar to the "constant comparison method" described by Glaser and Strauss [7] and the comparison method suggested by Eisenhardt in [4]. The method consisted of a case-by-case (meeting-by-meeting) comparison in order to reveal patterns among the characteristics of inspection meetings. The goal of this initial analysis was to suggest possible relationships between variables. These suggested relationships would then be further explored quantitatively where appropriate.

The quantitative variables chosen for analysis fall into three categories. First are the dependent variables, all of which have to do with the time or effort spent in the inspection process. Secondly, there is a set of independent variables which represent the issues of interest for this study, i.e., organizational issues. Finally, there are two intervening variables, size and complexity of the inspected material. These variables must be taken into account so that the relationships between independent and dependent variables will not be masked by them.

The quantitative analysis used in this study was fairly simple and straightforward. We began by looking at descriptive statistics (mean, minimum, maximum, median, standard deviation) for each of the variables. This helped to form an overall picture of the scope and shape of the data. Then we calculated Spearman correlation coefficients to determine which variables were statistically related (especially which organizational characteristics were related to measures of communication effort).

Qualitative data and findings were also used to help illuminate and explain the statistical findings. This was done in a more ad hoc way, by simply searching the field notes for anecdotes or quotes which shed some light on a particular finding. As well, after the initial quantitative results were generated, they were presented to several key developers on the project. This technique is called *member checking* [10]. The developers' responses to and explanations for those results were recorded qualitatively and also helped illuminate the statistical results. There are several examples of this in the next section.

RESULTS

Figure 2 depicts the network of relationships between factors that affect meeting length, according to the findings of this study. Each box represents a study variable or some other factor that became relevant during the course of the analysis. Each arrow represents some sort of relationship between two variables (e.g. a correlation) that was found in the data. We will discuss these variables and relationships in detail in the following subsections.

Components of Meeting Length

Although this study used dependent variables reflective of all parts of the inspection process, this paper presents results relating only to the inspection meeting itself, in particular the length of the meeting. Besides the actual meeting length, other relevant variables break the meeting length down into the time spent in various types of discussion. All of these variables are measures of communication effort because they describe the effort expended during the meeting, which was entirely a communication activity.

The *defect* discussion time associated with an inspection consists of time taken by raising, recording, and discussing actual defects. *Global* discussion time includes discussion of issues that are applicable to other parts of the system as well as the code being inspected. Since this includes the raising and discussing of "global" defects, this category overlaps with the defect discussion category. *Unresolved* discussion time refers to discussion of issues which could not be resolved during the meeting. *Administrative* time includes time spent in administrative activities as well as the discussion of administrative or process issues. *Miscellaneous* discussion time includes miscellaneous discussions of a technical nature, including raising and discussing questions about the code being inspected which are not determined to be defects. Aside from the overlap between "defect" and "global" discussion time, the categories are mutually exclusive.

The meeting time devoted to each discussion type is strongly correlated with meeting length, but defect discussion time is the most strongly correlated. Not only does the amount of time spent discussing defects increase for longer meetings, but so does the percentage of time spent in discussion of defects. In other words, much of what makes a long meeting long is due to extra time spent discussing defects.

However, other discussion types also play a role in determining the length of an inspection meeting. The amount of time spent discussing unresolved and global issues increases for longer meetings, as does the percentage of meeting time devoted to such discussions. Miscellaneous discussion time also seems to account for a considerable amount of the extra time spent in longer meetings.

Relatively speaking, the time spent in administrative tasks in an inspection meeting stays fairly constant and is nearly independent of the meeting length.

Organizational and Other Factors

To determine which organizational characteristics are relevant with respect to the amount of time spent in different types of discussion (and thus to the overall length of the meeting), we examined relationships between variables statistically using Spearman correlation coefficients. This test was chosen because it is non-parametric and does not require that the underlying distributions of the variables be normal. To examine the effect of the intervening variables, we also conducted the same tests between variables after subsetting the data by size and complexity. This was done to see whether or not certain relationships existed, not for all the inspections, but only for inspections of material of a certain size or complexity.

The objective of this study was to generate theory, not test it. The presentation of results below is summarized periodically with the hypotheses generated by the study findings.

Defect Discussion Time

The amount of time spent discussing defects during an inspection meeting is usefully broken down into two components. First of all, as might be expected, the defect discussion time is closely tied to the number of defects reported (Spearman coefficient 0.93, $p<.001$). However, there is some variation in the "defect discussion duration", which is the average amount of time spent discussing each defect raised in a meeting. It is useful to separate these two factors because the data shows that they are affected by different variables.

Data on the number of defects reported came from copies of the *Inspection Data Collection Form* for each inspection observed. These forms included a lot of other information about the inspection, most of which had already been collected during observations, so the forms served as a validation instrument.

It is somewhat surprising that the number of defects reported in a meeting is statistically unrelated to the size of the material being reviewed. *Size*, one of the two intervening variables used in this study, was coded into a three-level ordinal variable for analysis purposes. Size information was also provided on the Inspection Data Collection Forms.

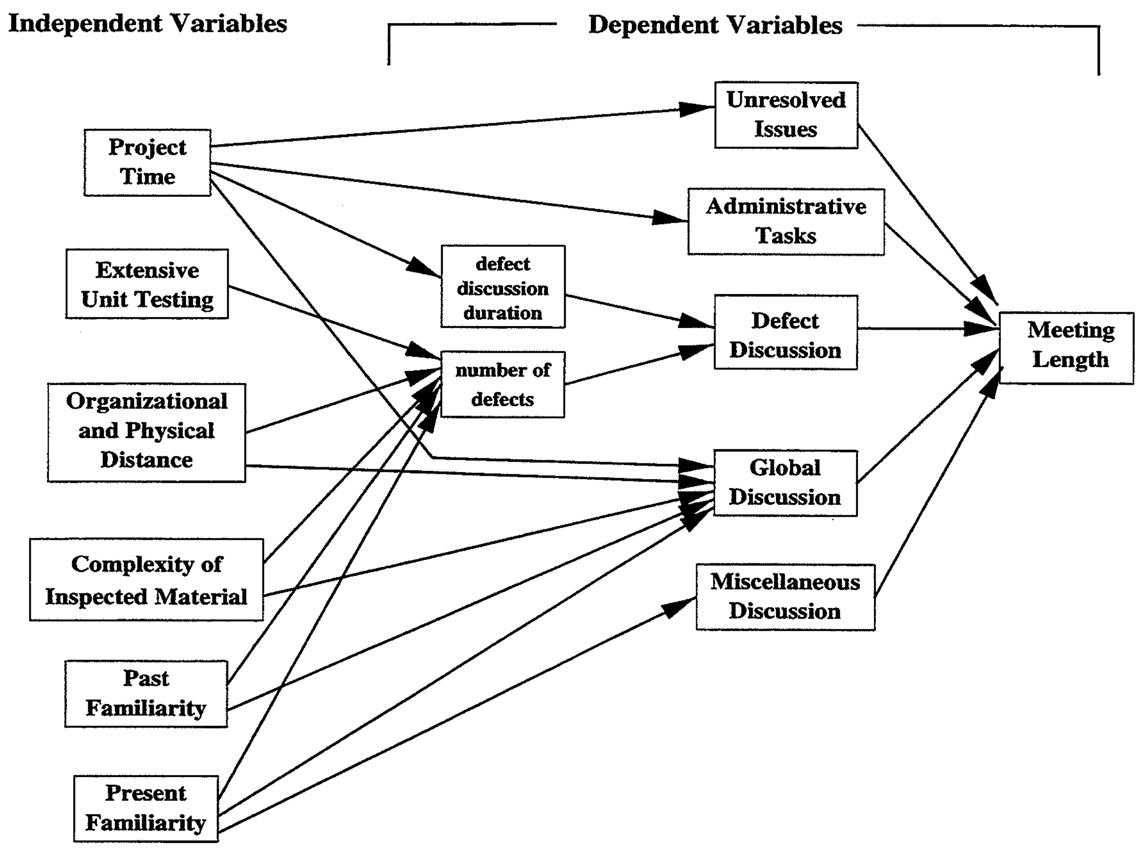


Figure 2: The network of relationships between variables.

The other intervening variable in this study, *complexity* of the inspected material, did seem to have a moderate effect on the number of defects reported. Fewer defects were reported when complexity was high (Spearman coefficient -0.5, p<.05). It might be reasonable to assume that material of high complexity actually contained fewer defects (maybe because it was assigned to more skilled developers). However, another explanation is that complex code was not inspected as carefully as less complex code. Fewer defects may have been reported because inspectors had to spend more time understanding the code and thus did not have adequate time to search for defects. Complexity was originally coded on a five-point subjective scale (based on interview data, described below), but was collapsed down to three levels for analysis.

Hypothesis: The more complex the material is, the fewer defects will be reported.

Under certain conditions, fewer defects tended to be reported when the inspection participants were more familiar with each other. Two measures of familiarity were used in this study, both based on pairs of inspection participants, and both ratio-valued. *Present familiarity* reflects the degree to which the participants in an inspection interact with each other on a regular basis, and thus share common internal representations of the work being done. The value of this variable is the proportion of *pairs* of participants in the set of inspection participants who interact with each other on a regular basis. *Past familiarity* reflects the degree to which a set of inspection participants have worked together on past projects. This is assumed to contribute to a shared internal representation, not of the current work, but of the application domain in general, and a shared vocabulary. Past familiarity represents the percentage of pairs of participants who have worked together on past projects. Both types of familiarity measures are based on information gathered during interviews with each project member.

When the material being inspected was of low complexity, fewer defects were reported when the inspection participants were very familiar with each other, based on either past or present working relationships (Spearman coefficients -0.95 and -1, respectively, p<.1). Also, no matter what the complexity, fewer defects were reported when the inspection participants were familiar based on past working relationships and the material inspected was small in size (Spearman coefficient -0.87, p<.1). So, for some types of material, closer past or present working relationships between the inspectors results in fewer defects reported. This may indicate that developers are reluctant to report all the defects they find in material authored by close colleagues, or that they tend not to inspect such material as carefully as that authored by developers who are less familiar to them. Yet another possible explanation is that the familiarity measures also reflect the average experience of the inspection participants. That is, people who have been working (in the company or on the project) longer will be more familiar with more people. Thus the fewer number of defects is actually a result of experience, not familiarity. However, no significant correlations were found between the familiarity measures and a rough measure of experience which was formulated for this purpose.

Hypothesis: The more familiar the inspection participants are with each other, the fewer defects will be reported.

Familiarity information was collected through interviews. Each interview used a tailored interview form, or guide [14], which included the questions and topics to be covered. These included information missing from the data form for a particular inspection, questions about organizational relationships, data on inspection activities other than the meeting, and information on the code inspected. These forms were not shown to the interviewee, but were used as a checklist and for recording answers and comments. In some cases, the more straightforward questions were asked via email. This was requested by the project management to reduce the amount of time the project personnel had to spend in interviews. Most interviews were audiotaped in their entirety. Extensive field notes were written immediately after each interview. The tapes were used during the writing of field notes, but they were not transcribed verbatim.

Another indication of familiarity is whether or not the author was in the "core group" which, for the purposes of this analysis, is defined as the eight developers who interact with other developers the most. This group consisted entirely of CSC developers, including the two CSC technical leads. All of the inspections included participants in the core group, but very few inspections involved exclusively core group members.

Inspections with a core-group author had less than half the number of reported defects than those with non-core group authors. And when the author was one of the technical leads, the number of reported defects was less than a third than in other inspections. One developer explained this latter result by explaining that one of the technical leads is very experienced, and the other, although not very experienced, is "a whiz". The classes assigned to the technical leads also tended to be lower in complexity than other classes as well.

There is also some evidence that extensive unit testing prior to the inspection reduces the number of defects reported in the meeting, which is intuitively logical. In two inspections, such extensive testing took place. In one of the inspections, one defect was reported, and two were reported in the other (much lower than the average of about 9). The low defect level cannot be explained by size or complexity. Because there were so few defects, there was very little defect discussion time, and the meetings themselves were correspondingly short. This result was especially satisfying to the developers to whom it was presented. Two developers, who both had leadership roles on the project, expressed the opinion that unit testing was a vital part of the development process, and this result was an indication that it was effective. However, since we do not know the actual defect densities of these classes, we might conclude that the inspectors may not have inspected as carefully because they knew that the classes had had extensive unit testing.

> **Hypothesis:** When more unit testing is performed prior to the inspection, fewer defects are reported.

There is also evidence that organizational and physical distance have an effect on the number of defects reported. *Organizational distance* refers to the degree of management hierarchy between members of the organization. In this study, each inspection was either organizationally "close" (all the participants reported to the same CSC manager) or organizationally "distant" (at least one participant from FDD was present). *Physical distance* reflects the number of physical boundaries between the inspection participants. In this study, physical distance takes on three values, corresponding to a set of inspection participants with offices on the same corridor, in the same building, or in separate buildings. The data used to evaluate the distance measures was collected during the prior ethnography phase, and was stored in the AD model built during that phase.

Both organizational and physical distance played a role in one particular inspection with respect to the number of defects reported. This inspection meeting was an outlier, the longest in the data set, at 100 minutes. The author was an FDD developer, while all the inspectors were from CSC (this was an unusual situation in the data set). Consequently, the inspectors were not very familiar with the class before they had inspected it in preparation for the meeting. This meeting also had the highest number of defects reported in the data set, 42. This may have been partly a direct result of the high organizational and physical distance between the participants, particularly the author. Fourteen (compared to an average of 2) of the defects were global in nature, meaning that they were defects that had been raised in previous inspections. However, the author of the outlier inspection was physically and organizationally removed from the participants in those previous inspections. This may have contributed to a lack of communication about the global defects. This is consistent with remarks from developers, who described developers in other parts of the organization as "isolated".

> **Hypothesis:** The closer the inspection participants are, either physically or in the reporting structure, the fewer defects will be reported.

The other factor contributing to the amount of time spent discussing defects, besides the number of defects reported, is the defect discussion duration, or the average amount of time spent discussing each defect. The defect discussion duration, surprisingly, is unrelated to either size or complexity of the inspection material. In fact, it is not correlated, in general, with any of the study variables. Significant correlations were found only under certain conditions. For example, for material of medium complexity, the duration of *global* defect discussions decreased over time. That is, the later in the project that the inspection occurred, the less time was spent discussing each global defect (Spearman coefficient -0.81, $p<.05$). As discussed in the next section, this most likely has more to do with the global nature of those defects than any property of defects in general.

In summary, a large part of the variation in meeting length is accounted for by the amount of time spent discussing defects, which in turn is largely dependent on the number of defects reported. This finding is somewhat comforting because the main purpose of an inspection meeting is, usually, to discuss defects. The number of defects is related to nearly all of the study variables, under different circumstances. However, the defect discussion duration also plays an important part in the amount of meeting time spent discussing defects. Unlike the number of reported defects, the defect discussion duration does not seem to be affected in general by any of the organizational variables, but under certain conditions it seemed to decrease over the course of the project. It should be noted that defect data was not available for 6 of the inspection meetings observed. The findings related to the number of defects or the defect discussion duration are based on only 17 inspection meetings, instead of the 23 that comprise the whole dataset.

Global Discussion Time

The time spent discussing global issues (including global defects) in an inspection meeting was strongly affected by a number of factors, as can be seen from the proliferation of arrows pointing to it in Figure 2.

First of all, global discussion time tended to be lower when the inspection participants were very familiar with each other, based on past working relationships. This correlation was not particularly strong in general (Spearman coefficient -0.38, p<.1), but was stronger for inspections of small amounts of material or material of low complexity. Also for material of low complexity, there was a strong tendency for global discussion time to be low when the inspection participants currently worked together a great deal (i.e. when present familiarity was high, Spearman coefficient -0.9, p<.05). In other words, people who interact on a regular basis spend less time discussing global issues only when the material being inspected is not very complex, but past working relationships have a more general effect. One developer addressed the latter result by observing that coding standards (which were the subject of many of the global discussions) are similar on all projects at CSC. So people who have worked together on past projects have most likely worked through some of these global issues together before, and thus it takes them less time to discuss them in the present. Also, it may be that developers are likely to discuss such issues outside the meeting with inspectors with whom they have worked before, thus reducing the need to discuss them during the meeting.

> **Hypothesis:** The more familiar the inspection participants are with each other, the less time they will spend discussing global issues.

There were some very specialized relationships between global discussion time and organizational and physical distance in some parts of the data.

For material of low complexity, there was strong tendency for more time to be spent discussing global issues when the inspection participants were organizationally or physically distant (Spearman coefficient 0.87, p<.1). However, the effect of organizational distance on global discussion time is very different when we restrict the data to inspections of large amounts of material. For such inspections, there was *less* global discussion time when the participants were organizationally distant (Spearman coefficient -0.64, p<.1). That is, more organizationally distant inspection participants spent less time on global issues when inspecting large amounts of material. These results are contradictory, and they imply that any effect that organizational distance has on the amount of global discussion time is overshadowed by the size and complexity of the material to inspect. It may be that large size, at least in some cases, leaves little time for inspectors to spend on "cosmetic" defects, which are often global. On the other hand, low complexity may allow inspectors to spend more time on such defects.

> **Hypothesis:** The closer the workspaces of the inspection participants, physically, the less time they will spend discussing global issues.

> **Hypothesis:** The distance between inspection participants in the reporting structure has an effect on the time they will spend discussing global issues, but depends on the size and complexity of the material being inspected.

This low complexity case is illustrated with the outlier meeting mentioned earlier (the longest meeting, at 100 minutes). The distance measures for this meeting were high, and it also included a large amount of global discussion. Global discussion constituted 18 minutes of the inspection meeting, which was much higher than the average of about 4 minutes. The complexity of the material was low, and it was small in size. The major factor seemed to be the organizational and physical distance of the author. Below is an excerpt from the field notes:

> One of the reasons this inspection was so long was that every "global" issue that had been hashed over in previous inspections was hashed out here as well, even a lot of things that had already been taken care of in [the code generator]. However, they all seemed to be a surprise to [the author], who hadn't gotten any of this presumably because he's at [FDD].

In some of the results above, the complexity of the material being inspected played a role by determining the conditions under which some results held. But complexity also had a direct relationship with global discussion time in the dataset as a whole. In general, the more complex the material, the less time was spent discussing global issues (Spearman coefficient -0.58, p<.005). This may indicate that, with highly complex material, the available time was spent discussing weightier issues than global defects, which are often "cosmetic".

> **Hypothesis:** The more complex the material being inspected, the less time will be spent discussing global issues.

Global discussion time also decreased over time to some extent, especially for material which was large or highly complex (Spearman coefficient -0.53, p<.001). This was explained by one developer as largely due to the role of the code generator, which was being developed concurrently. Many of the defects which were raised repeatedly in different inspections (i.e. global defects) were eventually remedied by implementing the fixes into the code generator. So, early in the project, a lot of effort was made to specify these problems and solutions carefully for the developers of the code generator, so that they would be implemented correctly.

Hypothesis: The later in the project the inspection occurs, the less time will be spent discussing global issues.

Other Discussion Types
Miscellaneous discussion time does not decrease significantly over time, nor is it significantly related to size or complexity. However, one component of miscellaneous discussion time (the amount of time spent asking and answering questions about the code being inspected) tends to be lower when the inspection participants are familiar, based on present working relationships (Spearman coefficient -0.65, $p<.1$). As explained by one developer, people who work together a lot are simply used to communicating, so can relay ideas very quickly. They also tend to discuss many issues outside the meeting, so less time is spent on them in the meeting.

As mentioned earlier, the time spent in administrative tasks in an inspection meeting is relatively constant, regardless of the meeting length. However, time spent in administrative tasks did decrease over time (Spearman coefficient -0.52, $p<.05$), especially for inspection material of low complexity. This is largely due to the fact that much of the administrative time in early inspection meetings was spent in asking and discussing questions about the inspection process itself. Inspections were just beginning on this project, the inspection process document had just been released, and inspections were being performed differently for this project in several ways. Inspection process questions consumed up to 5 minutes of each inspection meeting of the first 10 (out of 23) inspections observed. After that, process questions did not arise, and the administrative procedures became a "habit", as one developer put it. Even with the extra time in the early inspections, however, differences in administrative time between inspection meetings does not account for very much of the variance in meeting length.

In general, more meeting time was spent on unresolved issues early in the project than later (Spearman coefficient -0.49, $p<.05$). This was because, as one developer explained, developers at first made an effort to resolve every issue during the meeting, even if they eventually found they couldn't. However, they later came to recognize more quickly which issues were best referred to someone else.

CONCLUSIONS

This paper describes an empirical study of code inspection meetings in a NASA-sponsored software development project. The relevant variables in this study were process communication effort (in particular the effort expended in inspection meetings, in general and in discussions of different types) and characteristics of the organizational structure (reporting relationships, familiarity, physical proximity). We found that several organizational characteristics have an effect on the amount of time spent in different types of discussions during inspection meetings. Below, we present our findings in the form of testable hypotheses, which are the main contribution of this work (these are also presented graphically in Figure 2).

First, we presented results that showed that two of the major factors that make longer inspection meetings longer are the time spent discussing defects and the time spent discussing global issues. Furthermore, the time spent discussing defects is mostly determined by the number of defects reported during the meeting. The following hypotheses (similar to those presented previously) represent the study findings which relate to factors affecting the number of defects reported:

- **H1** The more the inspection participants interact with each other on a regular basis, the fewer defects will be reported.
- **H2** The more the inspection participants have worked together in the past, the fewer defects will be reported.
- **H3** The more closely related the inspection participants are in the reporting structure, the fewer defects will be reported.
- **H4** The closer the workspaces of the inspection participants are, physically, the fewer defects will be reported.
- **H5** The more complex the material is, the fewer defects will be reported.
- **H6** When more unit testing is performed prior to the inspection, fewer defects are reported.

Except for the last two of the above hypotheses, all of these point to the conclusion that developers will report fewer defects in material authored or inspected by other developers with whom they are "close" (in terms of organizational distance, physical distance, or familiarity). Unfortunately, this finding cannot be fully interpreted without knowing more about the error histories of the classes inspected. That is, we cannot know whether those classes which had fewer reported defects actually had fewer defects, or whether the closeness of the inspection participants influenced the inspectors to find or report fewer defects than actually existed. A follow-up study of testing data could provide the insight necessary to address this question. It is important to look at this issue closely because the number of reported defects appears to have a very strong influence on meeting length. In fact, aside from the various types of discussion times, it is the only variable that is strongly and directly associated with meeting length. Thus it is important to know what factors affect the number of defects reported, besides the actual number of defects in the code.

For example, suppose we extrapolate the above general conclusion (close inspection participants report fewer defects) to imply that close inspection participants report a lower percentage of the defects that actually exist in the code. This is as reasonable a statement as any, as we have no reason to assume that the distribution of defects in classes inspected by a close group is any different from that of other classes. This would indicate that, while choosing a close set of inspection participants would seem to make the inspection meeting more efficient, it would seriously degrade its effectiveness.

Another factor in the length of inspection meetings is the time spent discussing global issues, or those issues that arise repeatedly and are relevant to the system as a whole, not just the code being inspected. This study indicated that the time spent discussing such issues is strongly related to the organizational relationships between inspection participants, as detailed by these hypotheses:

- **H7** The more the inspection participants have worked together in the past, the less time they will spend discussing global issues.
- **H8** The more the inspection participants interact with each other on a regular basis, the less time they will spend discussing global issues.
- **H9** The distance between inspection participants in the reporting structure has an effect on the time they will spend discussing global issues, but depends on the size and complexity of the material being inspected.
- **H10** The closer the workspaces of the inspection participants, physically, the less time they will spend discussing global issues.
- **H11** The later in the project the inspection occurs, the less time will be spent discussing global issues.
- **H12** The more complex the material being inspected, the less time will be spent discussing global issues.

In general, it can be concluded that inspection participants who are "close" spend less time discussing global issues. This is likely due to several factors, including the amount of discussion which goes on outside the inspection meeting, the shared vocabulary that arises from familiarity which facilitates communication, and a shared understanding of the actual issues that come up repeatedly. Because less time is spent in global discussion, a close group of participants also results in a shorter meeting. This says nothing, however, about the effectiveness of such a meeting.

These hypotheses could all be tested in carefully controlled experiments that are designed for that purpose. The study described here provides some evidence of their validity.

This study peels back just one layer of understanding about the role organizational structure plays in the efficiency of inspection meetings. Many other, deeper, questions remain, however. For example, what makes an inspection efficient? Is an efficient inspection meeting necessarily shorter? Does it have less discussion of some types and more of another? The answers to questions like these lie, at least in part, on the goals and objectives of inspection meetings, which vary from project to project. If they can be answered for a particular project, then studies like the one described here can provide guidance as to the organizational factors which can be manipulated to meet the project goals.

Some of the qualitative data in this study indicated the complexity of these underlying questions. In the outlier meeting mentioned earlier, for example, recall that the number of defects reported was very high and the author was organizationally and physically distant from the other participants. He had not interacted with the inspectors during implementation of that class. This suggests the following argument. Different developers may be sensitive to different types of code errors, depending on their experience. The developers with whom an author consults during development, then, will help to eliminate certain types of errors from that author's code. If those same developers are those who inspect that code, they may not find many errors because those they are most aware of have already been eliminated. But if a different set of developers inspects the class, then they may bring different sensitivities to the inspection and thus find other errors (although they may take longer to do it). This may be what happened during the long outlier inspection. One developer addressed this very issue during an interview:

> She can imagine that if the inspectors are the same people who helped craft the code, then they're not likely to find anything wrong with it. So this may be a reason to choose inspectors that are not that familiar with the code.

The above anecdote is meant simply to underscore the fact that the work described in this paper helps to enable a whole area of research. Further work in the effects of organizational structure on the productivity of development processes has potential for profoundly influencing the success of software development projects. This study not only illustrates one effective way of conducting such investigations, but also provides some hypotheses with which to begin.

ACKNOWLEDGEMENTS
The authors would like to thank the members of the development team on the project used for this study for their patient cooperation. We would also like to thank the Experimental Software Engineering Group at the University of Maryland for their input at various stages of this work, including review of drafts of this paper. Also helpful were the comments from anonymous ICSE reviewers on the original submitted version of this paper.

REFERENCES

[1] B. Curtis, H. Krasner, and N. Iscoe. "A Field Study of the Software Design Process for Large Systems". *Communications of the ACM*, 31(11), Nov. 1988.

[2] K. M. Eisenhardt. "Building Theories from Case Study Research". *Academy of Management Review*, 14(4):532–550, 1989.

[3] J. R. Galbraith. *Organization Design*. Addison-Wesley, 1977.

[4] J. F. Gilgun. "Definitions, Methodologies, and Methods in Qualitative Family Research". In *Qualitative Methods in Family Research*. Sage, 1992.

[5] B. G. Glaser and A. L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine Publishing Company, 1967.

[6] H. Krasner, B. Curtis, and N. Iscoe. "Communication Breakdowns and Boundary Spanning Acitivities on Large Programming Projects". In G. Olsen, S. Sheppard, and E. Soloway, editors, *Empirical Studies of Programmers, second workshop*, chapter 4, pages 47–64. Ablex Publishing, New Jersey, 1987.

[7] R. E. Kraut and L. A. Streeter. "Coordination in Software Development". *Communications of the ACM*, 38(3):69–81, Mar. 1995.

[8] Y. S. Lincoln and E. G. Guba. *Naturalistic Inquiry*. Sage, 1985.

[9] J. March and H. A. Simon. *Organizations*. John Wiley, New York, 1958.

[10] D. E. Perry, N. A. Staudenmayer, and L. G. Votta. "People, Organizations, and Process Improvement". *IEEE Software*, July 1994.

[11] S. J. Taylor and R. Bogdan. *Introduction to Qualitative Research Methods*. John Wiley and Sons, 1984.

[12] E. Yu and J. Mylopoulos. "Understanding "why" in software process modeling, analysis, and design". In *Proceedings of the 16th IEEE International Conference on Software Engineering*, Sorrento, Italy, 1994.

S10-61

115474

14P

360951

Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo

HOW REUSE INFLUENCES PRODUCTIVITY IN OBJECT-ORIENTED SYSTEMS

ALTHOUGH reuse is assumed to be especially valuable in building high-quality software, as well as in OO development, limited empirical evidence connects reuse with productivity and quality gains. The authors' eight-system study begins to define such benefits in an OO framework, most notably in terms of reduced defect density and rework, as well as in increased productivity.

1997 ACM. Reprinted, with permission, from Communications of the ACM, vol. 39, no. 10, pp. 104 - 116

THIS article presents the results of a study conducted at the University of Maryland in which we assessed the impact of reuse on quality and productivity in object-oriented (OO) systems. Reuse is assumed to be an effective strategy for building high-quality software. However, there is currently little empirical information about what to expect from reuse in terms of productivity and quality gains.

The study is one step toward a better understanding of the benefits of reuse in an OO framework in light of currently available technology. Data was collected for four months—September through December 1994—on the development of eight small (less than 15,000 source lines of code [KSLOC]) systems with equivalent functional requirements. All eight projects were developed using the Waterfall-style Software Engineering Life Cycle Model, an OO design method, and the C++ programming language. The study found significant benefits from reuse in terms of reduced defect density and rework as well as increased productivity. These results can also help software organizations assess new reuse technologies against a quantitative and objective baseline of comparison.

Software reuse can help produce quality software more quickly. Software reuse is the process of using existing software artifacts instead of building them from scratch [18]. Broadly speaking, the reuse process involves three steps:

- Selecting a reusable artifact
- Adapting it to the purpose of the application
- Integrating it into the software product under development

The major motivation for reusing software artifacts is to decrease software development costs and cycle time by reducing the time and human effort required to build software products. Some research [3, 11, 21] suggests that software quality can be improved by reusing quality software artifacts. Some work has also hypothesized that software reuse is an important factor in reducing maintenance costs because, when reusing quality objects, the time and effort required to maintain software products can be reduced [4, 19]. Thus, the reuse of software products, software processes, and other software artifacts is considered the technological key to enabling the software industry to achieve required levels of productivity and quality [7].

This article assesses the impact of product reuse on software quality and productivity in the context of OO systems. OO approaches are assumed to make reuse more efficient from both financial and technical perspectives. However, there is little empirical evidence that high efficiency is actually achieved with current technology. Therefore, what's needed is a better understanding of the potential benefits of OO for reuse—as well as current OO limitations. We view several quality attributes as dependent variables, including rework effort and number/density of defects found during the testing phases.

Validating the Hypotheses

Participants in our empirical study were the students of a graduate-level class offered by the Department of Computer Science at the University of Maryland. The class's objective was to teach OO software analysis and design. The students were not required to have previous experience or training in the application domain or in OO methods. All students had some experience with C or C++ programming and relational databases and therefore had the basic skills needed for the study.

To control for differences in skills and experience, the students were randomly grouped into eight teams of three students per team. To ensure the teams were comparable with respect to the ability of their members, the following two-step procedure (known as blocking [17]) was used to assign students to teams:

- Each student's level of experience was characterized. We used questionnaires and performed interviews. We asked the students about their previous working experience, their student status (part-time or full-time), their computer science degree (B.S., M.S., Ph.D.), their previous experiences with analysis/design methods, and their skill in various programming languages.
- Each of the eight most experienced students was randomly assigned to a different team. Students considered most experienced were computer science Ph.D. candidates who had already implemented large (less than or equal to 10 KSLOC) C or C++ programs and those with industrial experience of more than two years in C programming. None of the students had experience in OO software analysis and design methods. Similarly, each of the eight next most experienced students was randomly assigned to different groups; this random assigning was repeated for the remaining eight students.

Each team was asked to develop a management information system supporting the rental/return process of a hypothetical video rental business and the maintenance of customer and video databases. Such an application domain had the advantage of being easily comprehensible; therefore, we could make sure that system requirements could be easily interpreted by students regardless of their educational background.

The development process was performed according to a sequential software engineering lifecycle model derived from the Waterfall model and including the following phases: analysis, design, implementation, testing, and repair. A document was delivered at the end of each phase—analysis document, design document, code, error report, and modified code. Analysis and design documents were checked to verify they matched the system requirements. Errors found in these first two phases were reported to the students. This verification and error checking maximized the chances that implementation would begin with a correct OO analysis/design. Acceptance testing was performed by an independent group. During the repair phase, the students were asked to correct their system based on the errors found by the independent test group.

The Object Modeling Technique (OMT), an OO analysis and design method, was used during the analysis and design phases [20]. The C++ programming language, the GNU software development environment, and OSF/MOTIF were used during implementation. Sun Microsystems Sparc workstations were used as the implementation platform—a development environment and technology representative of what is currently used in industry and academia. Our results are thus more likely to be generalizable to other development environments.

Rework seems to be lower in high-reuse categories, but there is no statistically significant evidence that faults are easier to detect and correct.

We provided the students with three libraries:

- **MotifApp.** This public-domain library includes C++ classes on top of OSF/MOTIF for manipulating windows, dialogs, and menus [22]. The MotifApp library provides a way to use the OSF/Motif widgets in an OO programming/design style.
- **GNU library.** This public-domain library is in the GNU C++ programming environment and contains functions for manipulation of strings, files, lists, and more.
- **C++ database library.** This library gives a C++ implementation of multi-indexed B-Trees.

We also provided a specific domain application library to make our study more representative of industrial conditions. This library implemented a graphical user interface (GUI) for insertion and removal of customer records and was implemented in such a way that the main resources of the OSF/Motif widgets and MotifApp library were used. Therefore, the library contained a small part of the implementation required for developing the rental system.

No special training was provided to teach the students how to use these libraries. However, the students received a tutorial describing how to implement OSF/Motif applications. In addition, a C++ programmer familiar with OSF/Motif applications was available to answer questions about the use of OSF/Motif widgets and the libraries. A hundred small programs exemplifying how to use OSF/Motif widgets were also provided. In addition, the code sources and the complete documentation of the libraries were provided. Finally, it should be noted that the students were not required to use the libraries and that, depending on the particular design they adopted, different reuse choices were expected.

To define the metrics to be collected during the experiment, we used the Goal/Question/Metric (GQM) paradigm [5, 7]. The study's goal was to analyze reuse in an OO software development process for evaluation with respect to rework effort, defect density, and productivity from an organizational point of view. In other words, our objective was to assess the following assumptions in the context of OO systems developed under currently available technology:

- A high reuse rate results in a lower likelihood of defects.
- A high reuse rate results in lower rework effort, that is, less effort to repair software products.
- A high reuse rate results in higher productivity.

According to the GQM paradigm, we had to define a set of questions pertinent to the defined experimental goal and a set of metrics allowing us to devise answers to these questions. We do not present the complete GQM here, only the metrics we derived. However, the metrics described in the next section were derived by following the GQM methodology [6].

Independent and Dependent Variables

Here we define the study's independent variables (e.g., size, amount of reuse) and dependent variables (e.g., productivity, defect density). We intend to make the underlying assumptions and models clear, so a precise terminology is used in the rest of the article. A thorough and formal discussion of these issues can be found in [10].

The size of a system S is a function *Size(S)* characterized by several properties, including the following:

- Size cannot be negative (property Size.1).
- We expect size to be null when a system does not contain any component (property Size.2).
- More important, when components do not have elements in common, we expect *Size* to be additive (property Size.3).

From these simple properties, other properties can be derived, as discussed in [10].

Let us assume an operator called *Components*, which, when applied to a system S, gives the distinct components of S, so that:

Components(S) = $\{C_1, ..., C_n\}$, such that if $C_i=C_j$ then $i=j$, where $i, j=1,...,n$.

The size of a system S is given by the following function:

$$\mathrm{Size}(S) = \sum_{c \in \mathit{Components}(S)} \mathrm{Size}(c)$$

where Size(c) can be defined as, for instance, the number of source lines of code of the component c. However, as discussed later, measuring size in the context of reuse raises difficult measurement issues related to such OO mechanisms as inheritance and aggregation of classes [20].

THE amount of reused code in a system S is a function *Reuse(S)*, also characterized by the properties Size.1–Size.3; that is, *Reuse(S)* is an instance of a size metric. Therefore, *Reuse* cannot be negative (property Size.1), and we expect it to be null when a system does not contain any reused element (property Size.2). When reused components do not have reused elements in common, we expect *Reuse* to be additive (property Size.3).

The way we define *Reuse(S)* must take into account specific OO concepts, such as classes and inheritance. For instance, consider a class C, which is included in a system S. There are five cases:

1. Class C belongs to the library LC. In this case we have verbatim reuse, that is, an existing class is included in the system S without being modified. Therefore:

   $\mathrm{Reuse}(C) = \mathrm{Size}(C)$

   As we are dealing with an OO language allowing inheritance, all ancestors of C, as well as all classes aggregated by C, also have to be included in S. As all C ancestors and all classes aggregated by C also belong to the library, including a library class may trigger an apparent large amount of verbatim reuse.

2. Class C is a new class created by specializing, through inheritance, a library class LC. This case is a variation of the first case, that is, the class LC and all its ancestors and subclasses (aggregated classes) will be included in S and will be dealt with in a way similar to verbatim reuse.
3. Class C is a new class that aggregates a library class LC. This case is also a variation of the first case, that is, the class LC and all its ancestors and subclasses will be included in S and will be considered in a way similar to verbatim reuse.
4. Class C has been created by changing the existing class EC. Reuse can be estimated as:

   $\mathrm{Reuse}(C) = (1 - \%\mathrm{Change}) \times \mathrm{Size}(C)$

   where %Change represents the percentage of C added to or modified from EC.

However, the percentage of change is difficult to obtain. As a simplification, we asked the developers to tell us if more or less than 25% of a component had been changed. In the former case, the class was labeled as extensively modified; in the latter case, the class was labeled as slightly modified. Therefore, reuse rates were computed based on the following approximations:

- Extensively modified: $\mathrm{Reuse}(C) = 0$
- Slightly modified: $\mathrm{Reuse}(C) = \mathrm{Size}(C)$

We show later that slightly modified and verbatim reused components are quite similar from the point of view of defect density and rework. Thus, the approximation appears to be reasonable.

5. Class C was created from scratch. In this case, the amount of reuse of the class C is 0:

   $\mathrm{Reuse}(C) = 0$

   Now assume a function called *Classes*, which when applied to a system S, yields all classes of the system S, so that:

   $\mathrm{Classes}(S) = \{C_1, \ldots, C_n\}$, such that if $C_i = C_j$, then $i=j$, where $i,j=1,\ldots,n$.

   Reuse of a system S is given by the following function:

   $$\mathrm{Reuse}(S) = \sum_{c \in \mathrm{Classes}(S)} \mathrm{Reuse}(c)$$

We are also particularly interested in knowing the reuse rate in a particular system. Reuse rate is measured by the following function:

$\mathrm{ReuseRate}(S) = \mathrm{Reuse}(S)/\mathrm{Size}(S)$

This metric has the property of being normalized: $0 \le \mathrm{ReuseRate}(S) \le 1$.

The first three cases show that the size measures of systems can be artificially inflated. Only a more detailed static analysis of the code would permit more precise size measurement by distinguishing what is actually used from what is inherited. This issue is addressed when defining measures of productivity and defect density.

Here we are interested in estimating the effort breakdown for development phases and for error correction:

- Person-hours per development activity, including:
  Analysis. The number of hours spent understanding the concepts embedded in the system before any actual design work, including requirements definition and requirements analysis, as well as analysis of changes made to requirements or specifications, regardless of where in the life cycle they occur.
  Design. The number of hours spent performing design activities, such as high-level partitioning of

the problem, drawing design diagrams, specifying components, writing class definitions, and defining object interactions. The time spent reviewing design material, such as doing walk-throughs and studying the current system design, was also taken into account.

Implementation. The number of hours spent writing code and testing individual system components, including: person-hours per error (referred to as rework), such as number of hours spent isolating an error and correcting it. We are also interested in rework efficiency, that is, how easily modifiable a class or a system is. To measure such an attribute, we normalize rework effort by the size of classes and the number of faults, or changes, respectively.

Here we are interested in measuring the productivity of each team. The measure used was the amount of code delivered by each project vs. the effort to develop such code, so:

$$\text{Productivity}(S) = \text{Size}(S)/\text{DE}(S)$$

where, in the study:

- Size(S) is first operationally defined as the number of lines of code delivered in the system S. Other size measures, such as function points, could have been used, but lines of code fulfilled our requirements and could be collected easily. More important, we were looking at the relative sizes of systems addressing similar requirements and therefore of similar functionality. However, because of the effect of inheritance on size measurement, we also measured size by excluding verbatim reused classes. This exclusion is not fully satisfactory because it underestimates the size of systems with a large amount of verbatim reuse classes. Nevertheless, it provides an additional insight on productivity complementary to our first measure. In addition, since all systems are supposed to be functionally equivalent, we also measured productivity by assuming that systems all have an equivalent size; therefore, effort was assumed to be inversely proportional to productivity. Again, this relationship is an approximation and is another interesting way to look at productivity.
- DE(S) (development effort) is defined as the total number of hours a group spent analyzing, designing, implementing, and repairing the system S.

Here we analyze the number and density of defects for each system component. We use the term *defect* generically to refer to either an error or a fault. Errors and faults are two pertinent ways to count defects, and both were considered in the study. Errors are defects in the human thought process made while trying to understand given information, to solve problems, or to use methods and tools. Faults are concrete manifestations of errors in the software. One error may cause several faults, and various errors may cause identical faults. Density is defined as:

$$\text{Density}(S) = \#\text{Defects}(S)/\text{Size}(S),$$

where #Defects(S) is defined as the total number of defects detected in the system S during test phases.

In the study, an error is assumed to be represented by a single error report form filled out by the independent tester group; a fault is represented by a physical change to a component, that is, in this particular context, a C++ class. Error density is first operationally defined as the number of errors found in a system over the number of KSLOC contained in the system. As for productivity, and for the same reasons, we also used a size measure excluding verbatim reused classes. Again we assumed that system sizes are roughly equivalent. In this case, defect counts were assumed to capture defect density.

Now assume that, in order to correct error E1, two classes—C1 and C2—have been modified, whereas, in order to correct E2, only class C2 was modified. In this case, the *fault* density of S is three faults per KSLOC.

To apportion errors to specific classes, we have to account for the fact that one error could result in changes to several classes. In this case, we follow a procedure illustrated by the following example:

- The error weight affecting C1 will be equal to 0.5 because two classes were modified to correct E1.
- The error weight of C2 will be equal to 1.5 because two classes were modified to correct E1, and only C2 was modified to correct E2.

This procedure is formally represented as:

$$|\,\text{ErrorWeight}(C_i)\,| = \sum_{E_{ij} \in \{E_{ij} \ldots E_{in}\}} |\,\text{Classes\_affected}(E_{ij})\,|$$

where:

- $|\,\text{ErrorWeight}(C_i)\,|$ is the error weight associated with the class C_i;
- $\{E_{ij} \ldots E_{in}\}$ is the set of errors in which the class C_i was affected; and
- $|\,\text{Classes\_affected}(E_{ij})\,|$ is the number of classes affected by the error E_{ij}.

We used the approach in [5], which proposes using forms for collecting data and gives guidelines for checking the accuracy of the information gathered. We used three different types of forms tailored from those used by the Software Engineering Laboratory [5]:

- **Personnel Resource Form.** This form is used to gather information about the amount of time the software engineers spent on each software development phase.

- **Component Origination Form.** This form is used to record information characterizing each component in the project under development at the time it gets into configuration management. This form is also used to capture whether the component has been developed from scratch or from a reused component. In the latter case, we collected the amount of modification—none, small, or large—needed to meet the system requirements and design, as well as the name of the reused component. By small/large, we mean less/more than 25% of the original code has been modified.
- **Error Report Form.** This form was used to gather data about (1) the errors found during the testing phase, (2) the components changed to correct such errors, and (3) the effort expended in correcting it. The last item includes:
  - Determining precisely what change was needed
  - Understanding the change or finding the cause of the error
  - Locating the point where the change was to be made
  - Determining that all effects of the change were accounted for
  - Implementing the correction, including design changes, code modifications, and regression testing

Validity

The study's validity can be analyzed from two perspectives: internal (What threats to the conclusions can we draw from the study?) and external (How generalizable are these results?) [17]. With respect to internal validity, we can say that subjects were classified according to their ability and assigned randomly to form "equivalent" teams. Therefore, we are less likely to obtain biased results due to differences in ability across teams. However, performing a formal controlled experiment would have required assignment of random levels of reuse to projects and classes—not feasible in practice. And because of this inability to assign random levels of reuse, reuse rates might be associated with other factors.

Concerning external validity, we can say that even though students are not industrial programmers, we have trained them thoroughly and we used an application domain intuitive enough to avoid misunderstandings when interpreting the requirements. In addition, we used a development environment representative of what is available in industry for OO software development. Another possible threat to external validity is that our systems were relatively small, so their conceptual complexity may be limited when compared to large software development applications. However, the inherent limitation of such empirical studies can't be avoided.

Software Product Reuse and Software Quality

We analyzed the impact of code reuse on software quality, investigating two aspects of quality: defect density and rework. We present the results in two ways:

- Assessing the differences in quality across reuse categories—new, extensively modified, slightly modified, verbatim
- Computing an approximate project reuse rate and assessing its statistical association with project quality

In the first form of analysis, projects are considered as separate entities; in the second, trends across projects are analyzed in a way that assumes the projects are comparable. In addition, the first type helps us justify the definition of the reuse metric we used for the study.

Note that during the analysis we did not distinguish between "horizontal" and "vertical" reuse; that is, the code reused from the generic libraries and the code reused from the domain-specific libraries have been combined. Even though comparing the benefits of these two kinds of software product reuse would be interesting, it is beyond the scope of this article.

The study found significant benefits from reuse in terms of reduced defect density and rework as well as increased productivity.

Reuse Vs. Defect Density

The first analysis compared reused and newly created code from the perspective of defect density. We looked at defects according to two definitions: errors and faults. At the class level, we used a simple measure of size—lines of code—but other size measures could have been used for the same purpose. However, since we were comparing systems developed based on identical requirements and of similar functionalities, we think this simple and convenient size measure is at least precise as a relative measure between projects. At the system level, three different measures of defect densities were investigated.

We first examined the relationship between classes and defect density to see if reused classes are less prone to defects. In addition, we used this analysis as an opportunity to evaluate the value of our ordinal reuse measure and assess its effect on defect density.

Table 1 shows the error and fault densities (errors and faults per thousand lines of code) observed in each of the four categories of class origin. Apparently, fewer defects were found in reused code. For example, error density was found to be only 0.125 in the code reused verbatim, 1.50 in the slightly modified code, 4.89 in the extensively modified code, and 6.11 in the newly developed code.

However, these differences should be assessed statistically; the significance of these trends should be calculated. To perform this statistical analysis, defect densities were computed for each project and each reuse category; the results were combined to derive defect densities for each project's subset of classes

The way we measure reuse and size needs to be refined to obtain more accurate measurement of what is actually used by the system as opposed to what is inherited.

belonging to each reuse category. When comparing reuse categories, we are in fact comparing sets of defect density values, each corresponding to a given project. Each observation in each reuse category therefore matches one observation in each of the other reuse categories, since they correspond to the same system and have been developed by an identical team. In addition, classes across reuse categories have similar complexity and comparable functionalities. Conceptually, it is almost like looking at the characteristics of identical sets of classes developed with different reuse rates.

Considering that eight systems were developed for the study, eight independent observations are available at the system level. The data set is rather small; consequently, we adopted a data analysis strategy following several steps:

Table 1. Error/fault densities and rework in each class origin category for all projects

Class Origin	No. of Comp.	No. of SLOC	No. Faults	Fault Density	No. of Errors	Error Density	Rework
New	177	25,642	247	9.63	157	6.11	336.35
Extensively Modified	79	15,165	93	6.13	74	4.89	160.04
Slightly Modified	45	6,685	11	1.57	10	1.50	22.5
Reused Verbatim	92	16,015	6	0.37	2	0.12	3
All Classes	**393**	**63,537**	**356**	**4.88**	**243**	**3.82**	**521.89**

- First, we used a nonparametric test (Wilcoxon matched-pairs signed rank test or Wilcoxon T test [12]) to determine whether significant differences could be observed between reused, modified, and newly developed classes. The rationale underlying this test is straightforward. We are comparing a set of pairs of scores (in this case, defect densities). Suppose the score for the first member of the pair is DD_1, and the score for the second member of the pair is DD_2. For each pair, we calculate the difference between the scores as $DD_2 - DD_1$. The null hypothesis we wish to test is that there is no difference between pairs of scores for the population from which the sample of pairs is drawn; that is, that there is no difference with respect to defect density between reuse categories. If this is true, we would expect similar numbers of negative and positive differences and similar magnitudes of differences. Therefore, the Wilcoxon T test takes into account both the direction and the magnitude of differences between scores, or defect densities, to determine whether the null hypothesis is reasonable. Thus, by using this test, we can obtain a statistical comparison of any project characteristic, such as defect density, across class reuse categories. Such a test does not assume all projects are comparable, does not require more than five observations per reuse category, and is robust to outliers, or extreme differences in scores between pairs. All these properties are important in our study.
- Once it has been determined that there is a significant difference between reuse categories, our goal is to quantify to the best extent possible the impact of reuse on defect density. To do this, we perform a linear least-squares regression between reuse rate and defect density. It might be argued that the number of data points we are working with is too small to allow such an analysis. However, there is common agreement that the number of independent observations per explanatory variable could be as low as five [14].

Another analysis strategy would have been to work at the class level by, say, comparing defect densities of classes across reuse categories, but this strategy presented problems:

- Some of the projects included a large percentage of all reused classes. Therefore, it would have been difficult to determine whether the observed trends could be attributed to skill differences between teams or to reuse.
- Our defect density and productivity measures can be considered suitable at the system level but are too rough at the class level.

We first looked at *fault* densities. Figure 1 shows the distribution and mean of project fault densities across reuse categories. Each diamond schematically represents the mean for each class reuse category. The line across each diamond represents the category mean. The height of each diamond is proportional to the 95% confidence interval for each category, and its width is proportional to the cate-

Figure 1. Distribution and mean of project fault densities across reuse categories

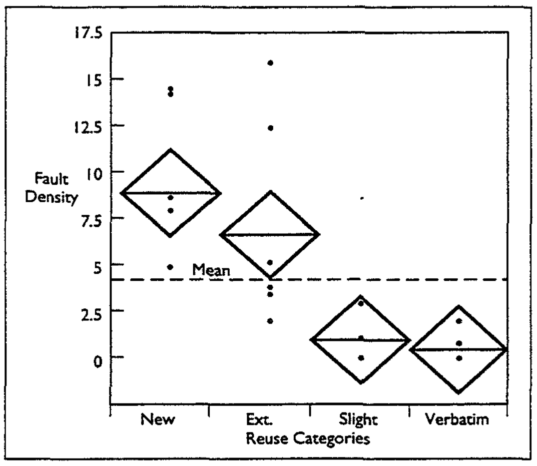
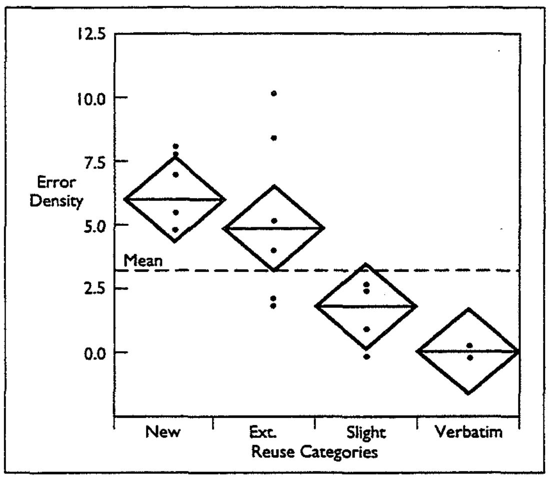


Figure 2. Distribution and mean of project error density across reuse categories



gory sample size. Recall that we want to know whether reuse reduces defect-proneness; in other words, the null hypothesis is: Reused and nonreused classes are, on average, of "similar defect-proneness." However, to gain confidence in the results, we have to check that the internal structure of the classes (in each reuse category) did not play a role in the outcome. We ran an analysis using various code metrics (e.g., cyclomatic complexity, nesting level, and function calls) and determined that the distributions across reuse categories were not statistically different. (These measures were extracted using the Amadeus tool [2].)

Table 2. Levels of significance (fault density per reuse category)

p-values	Slight	Ext.	New
Verbatim	0.46	0.012	0.012
Slight		0.012	0.012
Ext.			0.26

Table 2 shows the paired statistical comparisons of fault densities between reuse categories. We assumed significance at the 0.05 α-level; that is, if the p-value is greater than 0.05, we assume there is no observable difference. Recall that p-values are estimates of the probabilities that differences between reuse categories (in this case, in terms of project fault densities) are due to chance. According to these results, there is no support for the fact that there is an observable difference between verbatim reuse and slightly modified code, or between extensively modified and new code. This means that from the perspective of fault density, extensively modified code does not bring much benefit and that slightly modified code is nearly as good as code reused verbatim.

We used the same approach to obtain a statistical comparison of class error density per class category. Error density is more complicated to compute with respect to reuse categories, as one error may trigger changes in several classes from different categories. We calculated the error weight per class; then, for each class category, we computed the sum of the classes' error weights for each project and divided it by the sum of the sizes of those classes.

This assumption is not so strong, since (1) in general, each error generates only one fault, and (2) when an error generates many faults, in most cases all classes affected belonged to the same reuse category.

Table 3. Levels of significance (error density per reuse category)

p-values	Slight	Ext.	New
Verbatim	0.08	0.012	0.012
Slight		0.0136	0.025
Ext.			0.26

Therefore, all errors were considered with equal weight. The distributions are shown in Figure 2, and the Wilcoxon T-test results are shown in Table 3.

Again there is no observable difference between verbatim-reused and slightly modified classes (even though the significance improved compared to fault analysis results, it is still greater than 0.05) or between extensively modified and newly created classes. This lack of difference means that from the perspective of error density, extensively modified code does not bring much benefit and slightly modified code is as good as code reused verbatim. These results confirm the results we obtained using fault density as a quality measure.

We also wanted to verify the hypothesis that the higher the project reuse rate, the lower the number of project errors. For the sake of simplification, only

Project	No. of SLOC	No. of Errors	Reuse Rate	Error Density (with verbatim reuse code)	Error Density (without verbatim reuse code)	Rework
1	13,981	24	47.29	1.72	3.48	51
2	5,068	33	2.23	6.51	6.60	71
3	9,735	42	31.44	4.31	5.42	92
4	8,543	33	18.08	3.86	3.86	72
5	8,173	26	40.05	3.18	4.78	59
6	6,368	25	48.67	3.93	6.15	51
7	6,571	15	64.01	2.28	2.96	31
8	5,068	44	0.00	8.68	9.36	93

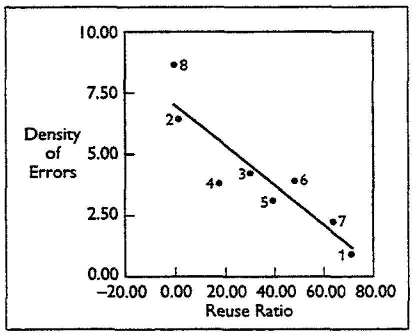
Table 4. Overview of the projects' data

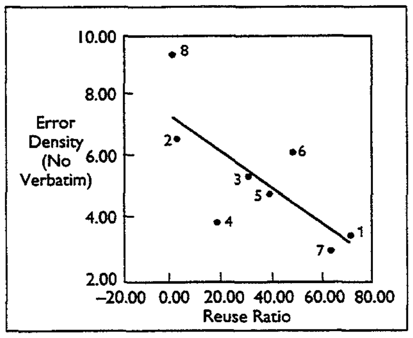
verbatim-reused and slightly modified classes were considered "reused classes" for computing the reuse rate per project. This approximation was to some extent justified by the results discussed earlier showing extremely different trends for slightly and extensively modified classes. We see this analysis as complementary to the earlier analysis, since the determination of the relationship between reuse rate and error density allows us to quantify the impact of reuse. However, the drawbacks of this new analysis are that all projects were assumed to be comparable and that the results could easily have been biased by outliers. Table 4 provides an overview of the projects' data, including number of lines of code delivered at the end of the implementation phase, reuse rate per project, error density (including and excluding verbatim reuse), and rework hours.

The average rate of reuse is approximately 31%, with a maximum of 64%. Based on Figure 3a, there appears to be a strong linear relationship between reuse rate and project error density when verbatim reuse is included in system size. This relationship is statistically significant (p-value = 0.0051 when performing an F-test) and shows a high coefficient of determination (R^2= 0.755). The estimated intercept and slope are 7.02 and -0.086, respectively. That means that when there is no reuse, error density should be expected to be around 7, and each additional 10 percentage points in the reuse rate decrease this density by nearly 1 (the estimate is 0.86) within the range covered by the data set (we limit ourselves to interpolation). No outlier seems to be the cause of a spurious correlation; therefore, this result should be meaningful (see Figure 3a).

Figures 3b and 3c show the relationships between reuse rate and, respectively, error density without verbatim reused classes (R^2= 0.54 and p-value = 0.04) and number of errors (R^2= 0.66 and p-value = 0.01). In both cases, a significant negative relationship can be observed, confirming our interpretation of the relationship identified in Figure 3a. In other words, we obtained consistent results using three different measures of error density as a dependent variable. Since these measures are based on very different assumptions, we are pretty confident in saying that reuse has a strong and positive impact on error-proneness.

These results support the assumption that reuse in OO software development yields a lower defect density. For example, the participants in Project 8 decided to implement everything from scratch; reuse did not have an impact on error density in their case. All the participants in the other projects performed better, and their error densities appear





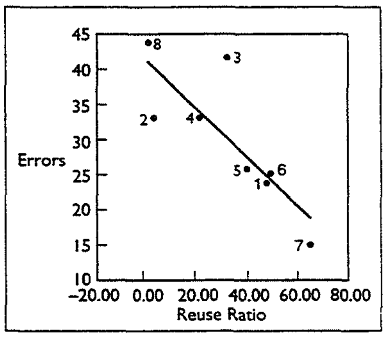


Figure 3a. Linear relationship between error density and reuse rate
Figure 3b. Linear relationship between error density without verbatim reuse code and reuse rate
Figure 3c. Linear relationship between a project's number of errors and its reuse rate

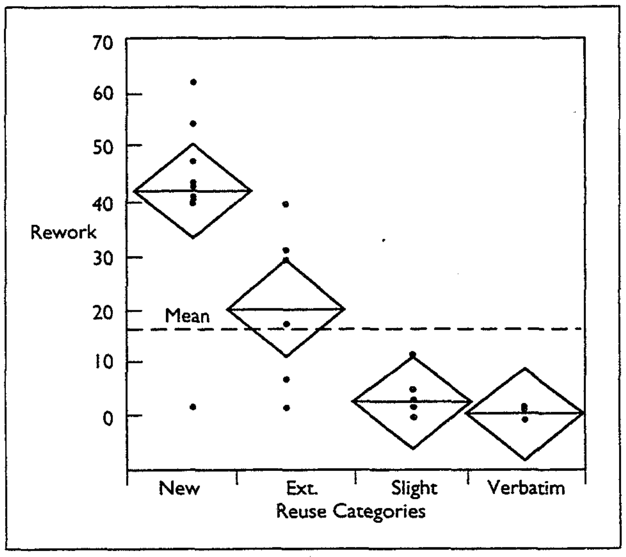


Figure 4. Distribution and mean of project rework per reuse category

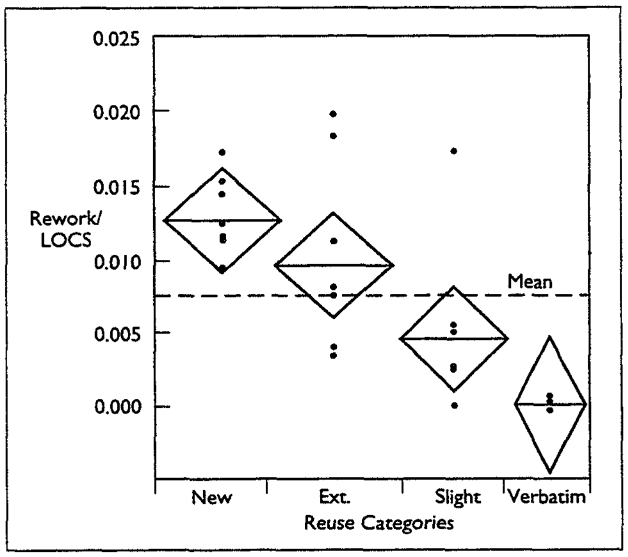


Figure 5. Distribution and mean of project rework density per reuse category

to decrease linearly with the reuse rate. This is strong evidence that reuse helped improve quality across the covered reuse rate range, that is, 0% to 64%.

In [16], rework is identified as a major cost factor in software development. Rework on the average accounts for more than 50% of the effort for large projects [16]. Reuse of previously developed, reviewed, and tested classes could result in easy-to-maintain classes and consequently should decrease the rework effort. Here, we first compare rework effort on reused and newly created classes. We then check whether the total amount of reuse per project is related to a reduction in the project rework effort.

Reuse Vs. Rework

We are interested in seeing whether the effort needed to repair reused classes is lower than the effort needed to repair classes created from scratch or extensively modified. We looked at three different measures to answer this question:

1. Total amount of rework in each class reuse category
2. Rework normalized by the size of the classes belonging to each reuse category
3. Rework normalized by the number of faults detected in the class of each reuse category

Distributions and means are shown in Figures 4, 5, and 6.

These metrics allowed us to look at rework from various perspectives:

- Capturing the total cost of rework, expected to be somewhat associated with the size of the classes and the number of faults, or changes, in each reuse category.
- Allowing us to look at rework without considering the relative amount of code in each reuse category, which gives us a more accurate insight into the relative cost of debugging and perfecting code in each reuse category.
- Allowing us to look at the expected difficulty of repairing a single fault across the various reuse categories, which gives us a more accurate insight into the modifiability of classes independent of their fault-proneness.

Before any thorough statistical analysis, a look at the distributions seems to indicate measures 1 and 2 are significantly different across reuse categories. To test the significance of this difference, we ran a Wilcoxon T test [12]. Instead of using defect density per reuse category as scores, we used total amount of rework

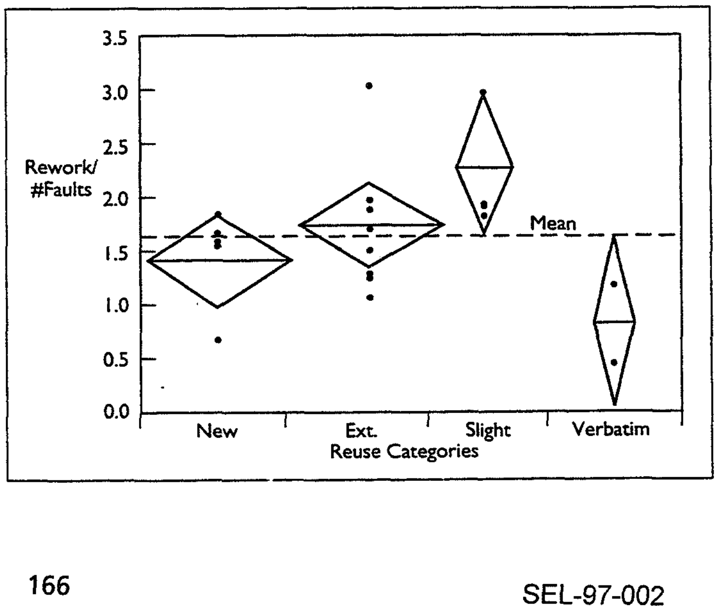


Figure 6. Distribution and mean of project rework difficulty per reuse category

per reuse category. The results for the first metric are shown in Table 5.

Based on Table 5, we can conclude that reuse reduces the amount of rework, even when the code is extensively modified. Again there is no observable difference between the verbatim-reused and slightly modified classes. These results show that from the perspective of total rework, extensively modified code might still bring benefits and, once again, slightly modified code is nearly as good as code reused verbatim.

Table 5. Rework per reuse category

p-values	Slight	Ext.	New
Verbatim	0.138	0.018	0.012
Slight		0.025	0.017
Ext.			0.05

As for defect density and rework, we used the Wilcoxon T test to analyze the variations of the second metric, that is, rework normalized by the size of the classes belonging to each reuse category (referred to as rework density) across reuse categories. Results are presented in Table 6.

Based on Table 6, we conclude that reuse reduces rework density except when the code is extensively modified. There is no observable difference between the verbatim-reused and slightly modified classes. These results show that from the perspective of rework density, reuse brings benefits and slightly modified code is nearly as good as code reused verbatim.

Some projects had no faults in their verbatim or slightly reused classes. The number of data points in these categories thus became too small for applying a Wilcoxon T test to the third metric. In such cases, we could look only at the difference between new and extensively modified classes. No significant difference could be observed in this case. As a last attempt to look at change difficulty (the third metric), we performed an analysis at the class level, where rework effort per class was normalized by the number of faults detected and corrected in these classes. No significant differences in distribution could be observed across reuse categories. If these results were confirmed by further studies, it would mean that differences in rework effort across reuse categories would be mainly due to differences in fault-proneness and not to differences in ease of modification.

To conclude, rework seems to be lower in high-reuse categories, but there is no statistically significant evidence that faults are easier to detect and correct.

To complement these results, we would like to verify whether larger project reuse rate is associated with lower project total rework effort. Even though the number of data points available is small, we can observe a strong linear relationship between rework and project reuse rates that is statistically significant ($p = 0.015$ on F-test), with a coefficient of determination R^2 of 0.65. The estimated intercept and slope are 88.52 and -0.748, respectively. That means that, where there is no reuse, rework effort should be expected to be around 88 person-hours for each project and that for each additional 10 percentage points in reuse rate (within the reuse rate interval covered by our data set) rework effort will decrease by nearly 7.5 person-hours. These results are, of course, specific to the system requirements implemented in the study, but they could be generalized as follows: Each additional 10 percentage points in reuse rate, within the reuse rate interval covered by our data set, decreases rework by nearly 8.5%. Again, no outlier seems to be causing any spurious correlation (see Figure 7).

Table 6. Rework/SLOC per reuse category

p-values	Slight	Ext.	New
Verbatim	0.144	0.043	0.043
Slight		0.028	0.042
Ext.			0.16

Table 7. Overview of the projects' data collected after the errors have been fixed

Project	SLOC Delivered	Reused	Reuse Rate	Effort	Productivity	Productivity (without verbatim reused code)
1	14,222	6,611	46.48	155	91.75	47.55
2	5,105	113	2.21	280	18.23	17.70
3	11,687	3,061	26.19	365	32.01	18.28
4	10,390	1,545	14.87	303	34.3	23.10
5	8,173	3,273	40.04	159	51.4	30.82
6	8,216	3,099	37.71	264	31.12	12.38
7	9,736	4,206	43.2	140	69.54	16.89
8	5,255	0	0	264	19.9	19.20

To better capture the concept of rework, it would be better to look at rework normalized by the size of the changes that occurred during the repair phase. Unfortunately, we could not capture this information accurately with the data collection procedures we had in place. As a rough approximation, we looked at rework normalized by the number of faults. However, no significant differences were observed between reuse categories. This result was confirmed when we attempted to investigate rework normalized by the number of faults at the class level.

In conclusion, the results support the assumption that reuse in OO software development results in lower rework effort.

Software Product Reuse and Software Productivity

Reuse has been advocated as a means of reducing development cost. For example, in [9], reuse of classes is identified as one of the most attractive strategies for improving productivity. As productivity is often considered an exponential function of software size, a reduction in the amount of software to be created could provide a dramatic savings in development costs [8]. The question now is, to what extent does reuse improve productivity—despite change and integration costs?

Table 7 shows, for each project analyzed:

- Number of lines of code delivered at the end of the lifecycle
- Number of lines of code reused (verbatim reused and slightly modified classes)
- Reuse rate
- Effort
- Productivity, including verbatim reused code
- Productivity, excluding verbatim reused code

Note that the data in the reuse rate and SLOC delivered column are different from the data in Table 4. This difference stems from the fact that Table 8 presents the results at the end of the lifecycle, that is, after the errors have been fixed, whereas Table 4 presents the data collected at the end of the implementation phase.

BASED on data in Figure 8a, we can conclude that there is also a strong linear relationship ($R^2 = 0.666$), which is statistically significant (p-value = 0.013 ON an F-test) between productivity (including verbatim reuse) and reuse rate. The estimated intercept and slope are 14.04 and 1.11, respectively. When there is no reuse, productivity should be expected to be around 14 SLOC per hour and each additional 10 percentage points in the reuse rate should increase productivity by 11 SLOC per hour. Figures 8b and 8c show the relationships between reuse rate and, respectively, productivity without verbatim reused classes ($R^2 = 0.45$ and p-value= 0.067) and effort ($R^2 = 0.38$ and p-value = 0.099). In Figure 8b, a weaker positive relationship (significant at the a = 0.1 level) can be observed, confirming our interpretation that productivity has improved. Figure 8c shows a weak negative trend (expected to be negative because the dependent variable is effort), also supporting our claim about productivity improvement. However, the latter figure is graphically and statistically not as clear as the other two figures, due to the third observation, which is clearly an outlier.

To explain outliers on these scatterplots, we performed some qualitative analysis of the process used, the teams involved, and the design strategies adopted in each project. For example, the team in Project 3 had no previous experience with respect to GUIs, and

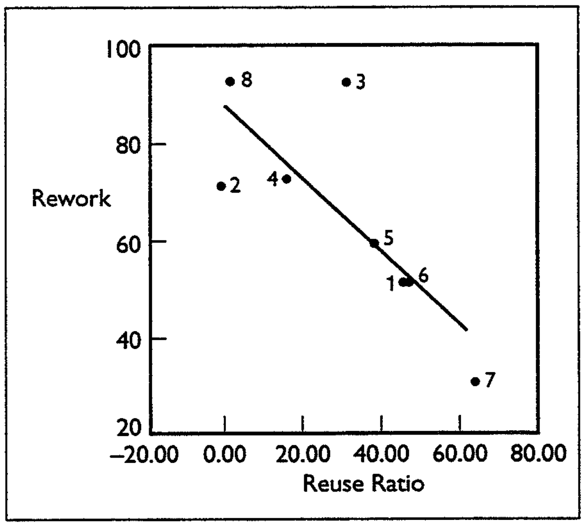
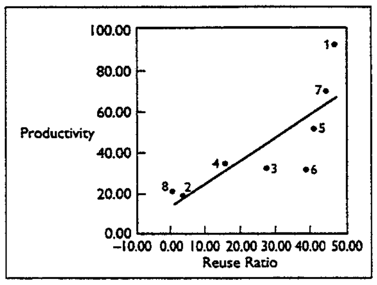


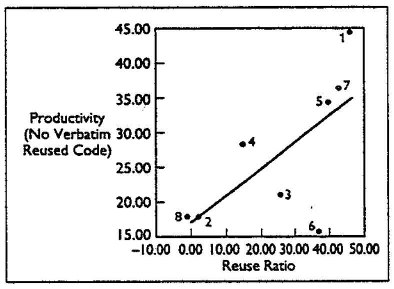
Figure 7. Linear relationship between rework and reuse rate

learning the basics was perceived as a significant effort. Similarly, Project 6 appears to have had lower productivity than expected in Figures 8a, 8b, and 8c when considering reuse rate. Lower productivity was explained by the particularly sophisticated GUI this group designed. In the context of the requirements we provided to the students, the GUI could be considered gold-plating.

Conclusion

This article offers significant results showing the strong impact of reuse on product productivity and,





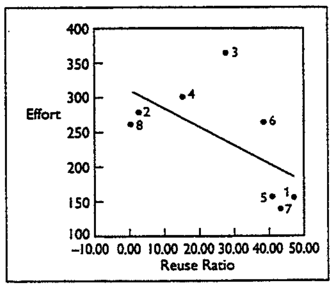


Figure 8a. Linear relationship between productivity and reuse rate
Figure 8b. Linear relationship between productivity (without verbatim reuse code) and reuse rate
Figure 8c. Linear relationship between development effort and reuse rate

especially, on product quality, or defect density and rework density, in the context of OO systems. In addition, these results were obtained in a common and representative OO development environment using standard OO technology. Such results can be used as rough estimates by managers and as a baseline of comparison in future reuse studies for the purpose of evaluating reuse processes and technologies.

This study can and must be replicated in industry and academia. In industry, replicating this study can, for example, help managers decide whether it is worth investing in particular OO technologies to improve software quality and productivity. In academia, replicating this study can help test OO methods or compare the advantages of such methods against those of traditional development methods. In any case, replication is necessary to confirm the results we obtained and refine the models we built.

Future work includes refinement of the information collected during the repair phase with regard to the size and complexity of the changes. This would allow us to better estimate the impact of reuse on rework. However, it is likely to require better automation of the change data and therefore the design of tools for monitoring the changes to code and design documents. In addition, the way we measure reuse and size needs to be refined to obtain more accurate measurement of what is actually used by the system as opposed to what is inherited. Thus, we should be able to measure productivity and defect density more precisely.

We also intend, in future replications of this experiment, to assess independently the impact of horizontal (non-domain-specific) and vertical (domain-specific) software reuse on software quality and productivity. We will compare the advantages and drawbacks of using these two types of software libraries. Finally, it would be interesting to refine our comparison of the internal class characteristics across reuse categories by using more specific OO metrics [1]. We need to better characterize the impact of reuse on system size, complexity, coupling, and cohesion. ◘

Acknowledgments
We want to thank Gianluigi Caldiera for helping us teach the OMT method; Khaled El Emam, Jyrki Kontio, Carolyn Seaman, and Barbara Swain for their suggestions, which helped improve both the content and the form of this article; and the students of the University of Maryland for participating in this study.

This work was supported, in part, by NASA Grant NSG-5123, NSF Grant 01-5-24845, Fraunhofer Gesellschaft, UMIACS, and Westinghouse Corp.

References

1. Abreu, F.B., and Carapuça, R. Candidate metrics for object-oriented software within a taxonomy framework. *J. Syst. Software 26*, 1 (1994), 87–96.
2. Amadeus Software Research, Inc. *Getting Started with Amadeus.* Amadeus Measurement System, Irvine, Calif., 1994.
3. Agresti, W.W., and McGarry, F. The Minnowbrook workshop on software reuse: A summary report. In *Software Reuse: Emerging Technology*, W. Tracz, Ed., IEEE Press, 1987.
4. Basili, V. Viewing maintenance as reuse-oriented software development. *IEEE Software 7*, 1 (Jan. 1990), 19–25.
5. Basili, V., and Weiss, D.M. A methodology for collecting valid software engineering data. *IEEE Trans. Software Eng. 10*, 6 (Nov. 1984) 728–738.
6. Basili, V., and Rombach, H.D. Support for comprehensive reuse. IEEE Software Engineering Journal (Sept. 1991), 303–316.
7. Basili, V., and Rombach, H.D. The TAME project: Towards improvement-oriented software environments. *IEEE Trans. on Software Eng. J. 14*, 6 (June 1988), 758–773.
8. Boehm, B.W. *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, N.J., 1981.
9. Boehm, B.W., and Papaccio, P.N. Understanding and controlling software costs. *IEEE Trans. Software Eng. 14*, 10 (Oct. 1988), 1462–1476.
10. Briand, L., Morasca, S., and Basili, V. Property-based software engineering measurement. *IEEE Trans. Software Eng. 22*, 1 (Jan. 1996), 68–86.
11. Brooks, F.P. No silver bullet: Essence and accidents of software engineering. *Computer 20*, 4 (Apr. 1987).
12. Devore, J. *Probability and Statistics for Engineering and Sciences.* Brooks Cole Publishing Co. 1991.
13. Fenton, N.E. *Software Metrics: A Rigorous Approach.* Chapman & Hall, London, 1991.
14. Ferguson, G., and Takane, Y. *Statistical Analysis in Psychology and Education.* McGraw-Hill, New York, 1989.
15. Heller, G., Valett, J., and Wild, M. Data Collection Procedure for the Software Engineering Laboratory (SEL) Database. SEL Series, SEL-92-002. NASA/GSFC, Greenbelt, Md., 1992.
16. Jones, T.C. *Programming Productivity.* McGraw-Hill, New York, 1986.
17. Judd, C.M., Smith, E.R., and Kidder, L.H. *Research Methods in Social Relations.* Harcourt Brace Jovanovich College Publishers, 1991.
18. Krueger, C.W. Software reuse. *ACM Comput. Surv. 24*, 2 (1992), 131–183.
19. Rombach, H.D. Software reuse: A key to the maintenance problem. *Inf. Software Technol. J. 33*, 1 (Jan./Feb. 1991), 86–92.
20. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, N.J., 1991.
21. Thomas, W., Delis, A., and Basili, V. An evaluation of Ada source code reuse. In *Proceedings of the Ada-Europe International Conference* (Zandvoort, The Netherlands, June 1992).
22. Young, D.A. *Object-Oriented Programming with C++ and OSF/MOTIF.* Prentice-Hall, Englewood Cliffs, N.J., 1992.

VICTOR R. BASILI is a professor in the University of Maryland's Institute for Advanced Computer Studies and Computer Science Department. He can be reached at basili@cs.umd.edu.

LIONEL C. BRIAND is head of the quality and process engineering department at the Fraunhofer-Institute for Experimental Software Engineering in Germany. He can be reached at briand@iese.fhg.de.

WALCÉLIO L. MELO is Lead Researcher in the Software Engineering Group in the Centre de Rescherche Informatique de Montréal in Canada. He can be reached at wmelo@crim.ca.

OMIT THIS PAGE

SECTION 5—ADA TECHNOLOGY

The technical papers included in this section were originally prepared as indicated below.

- "Using Applet Magic (tm) to Implement an Orbit Propagator: New Life for Ada Objects," M. Stark, *Proceedings of the 14th Annual Washington Ada Symposium (WAdaS97),* June 1997
- "The Generalized Support Software (GSS) Domain Engineering Process: An Object-Oriented Implementation and Reuse Success at Goddard Space Flight Center," S. Condon, R. Hendrick, M. Stark, W. Steger, *Addendum to the Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 96*), San Jose, California, U.S.A., October 1996

5/1-61
415795
360754

Using AppletMagic(tm) to Implement an Orbit Propagator: New Life for Ada Objects

16p

Michael Stark

Flight Dynamics Division/ Code 551
Goddard Space Flight Center
Greenbelt, MD 20771

michael.e.stark@gsfc.nasa.gov
Phone: 301-286-5048
Fax: 301-286-0245

Introduction

This paper will discuss the use of the Intermetrics AppletMagic tool to build an applet to display a satellite ground track on a world map. This applet is the result of a prototype project that was developed by the Goddard Space Flight Center's Flight Dynamics Division (FDD), starting in June of 1996. Both Version 1 and Version 2 of this applet can be accessed via the URL http://fdd.gsfc.nasa.gov/Java.html. This paper covers Version 1, as Version 2 did not make radical changes to the Ada part of the applet.

This paper will briefly describe the design of the applet, discuss the issues that arose during development, and will conclude with lessons learned and future plans for the FDD's use of Ada and Java. The purpose of this paper is to show examples of a successful project using AppletMagic, and to highlight some of the pitfalls that occurred along the way. It is hoped that this discussion will be useful both to users of AppletMagic and to organizations such as Intermetrics that develop new technology.

The Orbit Applet

Figure 1 shows the design of the Orbit Propagator applet.

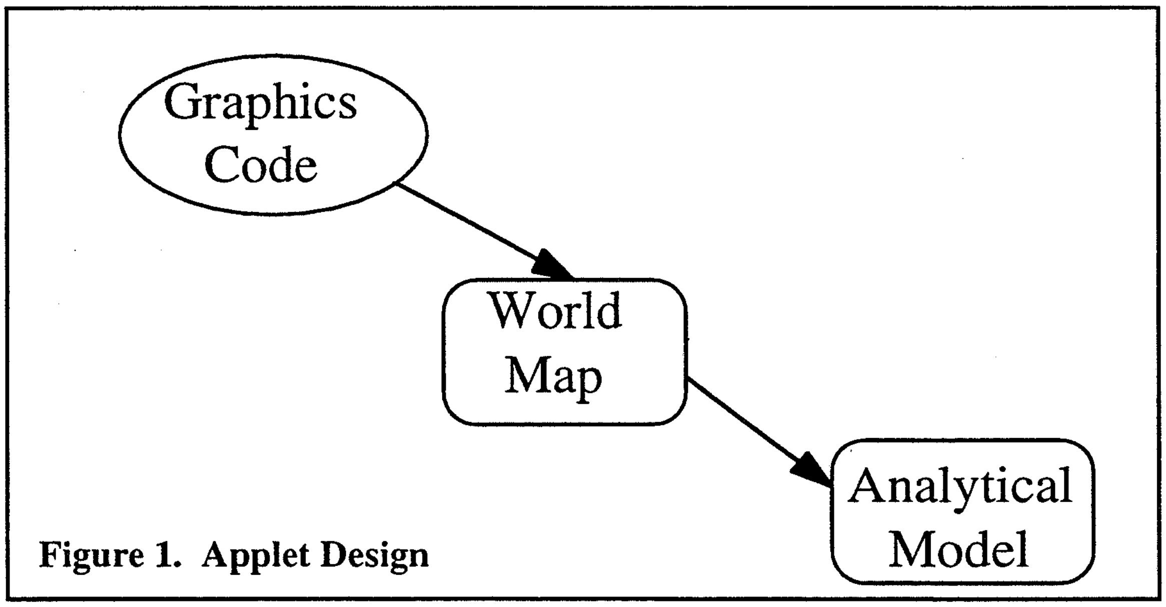


Figure 1. Applet Design

This applet is built from the following components:

Package Analytical_Model is reused from the Generalized Support Software (GSS) library class of the same name [1]. It implements the GSS class Analytical Model with the abstract data type Analytical_Model.Instance. This package was modified to simplify the design, by adding parameters (such as Earth radius) to the package that in GSS are retrieved via dependencies on other objects. The operation Propagate is used to compute the satellite's position and velocity at a requested time. Selectors are then used to access these data.

Package World_Map is responsible for controlling the propagation of the orbit and for computing longitude and latitude. This package is implemented as an abstract state machine, instead of exporting a private type it exports subprograms and simple numeric types and stores state data as package body variables.

For each point to be plotted package updates the current (simulated) clock by adding the user input step size to the simulated time, propagates the orbit to that time, then computes the longitude and latitude. The longitude and latitude is computed by converting the position vector (x,y,z) to spherical coordinates (right ascension, declination, height), then converting right ascension to longitude by accounting for the Earth's rotation (latitude and declination mean the same thing in this case). The latitude and longitude can then be selected by any client of the World_Map package.

Graphics code is written in Java as an applet which can be started from a Web browser. This code retrieves the latitude and longitude from the World_Map package, and then plots a point at the appropriate location over a Mercator projection map of the Earth.

The total size of the system is shown in the table below. The system also references a utility library of approximately 35,000 lines of code (carriage returns), but uses only a small proportion of these utilities.

Component	# lines
Analytical_Model package (specification & body)	895
World_Map package (specification and body)	265
Ada test driver (main.ada)	47
Java test driver (driver.java)	102

The size of each of these components is measured in carriage returns. Listing of World_Map, main.ada and driver.java are contained in the appendix to this paper. A full set of Ada source code will be available at the FDD Java Web site.

The approach to developing the applet was a three step approach

1) Develop the Ada code and test an application that uses an Ada test driver.
2) Replace the Ada test driver with a Java test driver to test the interface.
3) Deliver the compiled code for integration with the graphics portion of the applet.

The integration of the Ada components with the applet did not involve the use of AppletMagic, and the interaction of the graphics code with the propagator code is closely modeled by the Java test driver. Thus we will use the Java test driver to show how the code compiled by AppletMagic was used within the applet.

The Development Process

This project was developed with one part-time Ada developer (the author), with a second developer to help with the initial porting of our utilities to AppletMagic, and one Java developer who was responsible for the graphics. The team initially had little experience with either Ada 95 or Java. We started our work with Version 1.3 of AppletMagic under Solaris on a Sun Ultra machine, and are currently using Version 2.0.1.

The main issues encountered were that the AppletMagic documentation contains more information about Ada systems using Java libraries than on Java code calling Ada, and that the compiler used was a beta version. The parts below will discuss these issues in chronological order. The section on lessons learned will draw conclusions from the results of this project.

Getting Started

The approach taken to the world map application was to initially develop and test using our existing Ada 83 development environment (VADS under HP/UX 9.05), then to rebuild and test using AppletMagic. The development under VADS involved simplifying the Analytical_Model package to remove dependencies, developing the World_Map package, and writing the test driver Main. Having done this, we were ready to start with AppletMagic.

The first step was to recompile the system, using a compilation order generated by the VADS compiler and substituting invocations of AppletMagic's "adajava" command for the invocations of VADS. We found that AppletMagic produced clear messages and well formatted listings, so the initial compilation went smoothly. We encountered only minor problems getting the AppletMagic version of Main running.

The first was the simple fact that Version 1.3 of AppletMagic did not implement Text_IO. This problem was circumvented by importing java.io.Printstream and using the println() function. Intermetrics added the Text_IO package to a later release of AppletMagic. This was requested by several users, because Java's print functions lack the formatting capabilities of the Ada.Text_IO package or of C's printf() function.

In addition to Text_IO, there were some predefined functions that were not yet implemented. The most notable of these were "mod" and "**", which are heavily used in our mathematically oriented domain.

The second issue was that the VADS compiler (at least at optimization level 0) generated a compilation order that compiled generic package instantiations before the generic package body. This does not cause any problems within VADS, but would not compile with AppletMagic. As we were more interested in producing a running application and applet than in investigating the cause of the problem, we simply changed the order and pressed on.

Once these issues were resolved, all the Ada code was compiled and executed, and the results were identical to the VADS compilation of the same code. We were then ready to go on and write a Java test driver, which would be used by the graphics developer as an example of how to use the Ada code.

Java Calling Ada

To test the ability to call Ada code from Java, we replaced the Ada test driver Main with the Java class Driver. To determine what calls Driver would make to World_Map, we needed to execute the Java disassembler via the "javap World_Map" command. This command provides Java source code for the interface to the Ada package World_Map. The results of javap are usually redirected to a file that is used to document the interface.

The only difficulty encountered was in converting Java strings to Ada strings. We initially found the problem by examining the output of javap. The operations with string arguments had an extra parameter that (presumably) contained array size information, and returned a byte[] array rather than the expected char [] array. When we examined the problem further, we found that AppletMagic provides the operation Interfaces.java."+", which converts an Ada string to a Java string (represented by the Ada type java.lang.String_Ptr), but does not provide the inverse operation. This allows a call of the Java operation (on class Graphics)

```
    g.drawstring ("Ada rules!");
```

to be implemented as

```
    drawstring (g, +"Ada rules");
```

in Ada packages.

Thus we looked at the string representations in more detail. Ada strings are the familiar array of 8-bit characters, but Java implements strings in class String, which is accessed by reference, rather than declared as an array. The Java strings also contain 16-bit Unicode characters, rather than the usual 8-bit character.

These difficulties were overcome by using pragma Convention (Java, *subprogram_name*) to eliminate the array size information, and by using type Wide_String instead of String for the subprogram argument. These modifications to package World_Map were the first use of features that are new in Ada 95. The type Wide_String in Ada directly maps to type char[] in Java, so to call the Ada function, a Java string S would be converted to Wide_String by passing the argument S.toCharArray() to the Ada subprogram.

The use of pragma Convention is briefly discussed in the applet writer's guide [2], but was not discussed in the context of passing array parameters between Ada and Java. Since this information applies to *passing any array type*, not just strings, it probably deserves to be highlighted in its own applet writer's guide section.

This approach corrected the interface. The only remaining problem was to convert Wide_String parameters to and from type String, so that our existing utility library could be used. The source code for these functions, which were implemented in the package body of World_Map, is shown below:

```
-- Hidden functions to handle strings of 16-bit characters
-- these function allow reuse of 8-bit character string utilities

   function To_Wide_String ( S: in String) return Wide_String is
      W : Wide_String(1..S'length);
      J : Positive := S'first;
      Pos : Natural;

   begin
      for I in W'range loop
         Pos := Character'pos (S(J) );
            W(I)  := Wide_Character'val(Pos);
         J    := J+1;

      end loop;
      return W;
   end To_Wide_String;

--
   function To_String (W : in Wide_String ) return String is
      S : String (1..W'length);
      J : Positive := W'first;
      Pos : Natural;

   begin
      for I in S'range loop
         Pos := Wide_Character'pos (W(J) );
         S(I) := Character'val(Pos);
         J := J+1;

      end loop;
      return S;

   end To_String;
```

The conversion function To_String worked well, but To_Wide_String raised the Java exception "java.lang.IndexOutOfRangeException" when the boldfaced line was executed. The conversion to Wide_String was intended for use in the Current_Time_Of selector function, which was used in testing, but not in the full scale applet. The testing work around was to write a second time selector that returns type String, and modifying the Ada test driver to use that function. The Java test driver does not write time data as part of its output. This was another case of a problem that was not pursued to resolution, since it did not stop the development of the applet.

Once the issues related to string parameters were resolved, the application produced the same results with the Java test driver as with the Ada test driver. The next steps were to integrate the Ada packages with the Java graphics code, and to prototype the possible interactions of Java classes with Ada tagged types. We will first address the implementation of classes in Ada 95 and Java.

Implementing Classes in Ada 95

The World_Map package described above met the goal of producing applet code with minimal effort. However, we also needed to investigate the interoperability of Ada 95 and Java code. To do this, we needed to demonstrate the following:

1) Objects can be allocated in Java code from classes implemented in Ada 95
2) Objects can be allocated in Ada 95 from classes implemented in Java
3) Classes implemented in Ada 95 can be extended in Java
4) Classes implemented in Java can be extended in Ada 95.

The fourth objective was met during the prototyping of the first, by developing Ada classes as explicit extensions of java.lang.Object. The result of the prototyping was that once the first objective was met the second and third were straightforward to verify.

The first step of our protoyping was to rewrite World_Map to meet Intermetric's convention for classes. The new version of the World_Map package is shown below in outline form:

```
package World_Map is
  type World_Map_Obj is tagged private;
  type World_Map_Ptr is access all World_Map_Obj'class;
   ...
  procedure Initialize (wm : access World_Map_Obj);
  function Current_Time_Of (wm: access World_Map_Obj) return Wide_String;
   ...
private
  type World_Map_Obj is tagged record
    -- state data are defined as record components
  end record;

end World_Map;
```

The convention of having the package name match the Java class name, a tagged record with "_Obj" appended to the package name, and the access type with the "_Ptr" suffix allows AppletMagic to create the file World_Map.class in the same directory. If one does not follow the convention, the package name becomes a subdirectory of the current directory, and the tagged type name becomes the name of the class file in that directory. AppletMagic also makes all the directory names lower case to follow the Java style convention for package names.

Either naming approach is a reasonable option depending on a project's requirements. It was not clear from AppletMagic's documentation that the naming convention is optional, not mandatory. We discovered this through trial and error.

Our initial attempts to compile the new World_Map package ran into a more fundamental documentation problem. The applet writer's guide did not provide guidance on calling Ada code from Java. This was not an issue with the abstract state machine implementation. AppletMagic generated the file World_Map.class, and when this file was disassembled with javap the interface contained static functions corresponding to the Ada subprograms.. However, when the code was changed so that the subprograms have access parameters, disassembling World_Map.class produced *no* functions corresponding to Ada subprograms.

We initially thought that we would need to explicitly extend java.lang.Object, but when we changed the declaration of type World_Map_Obj to

```
    type World_Map_Obj is new java.lang.Object with private;
```

we had the same unsatisfying result. The problem was that when a subprogram has an access parameter as an argument, one needs to use pragma Export

```
    procedure Initialize (wm: access World_Map_Obj);
    pragma Export (Java, Initialize);
```

to produce the corresponding Java function

```
    public void Initialize ();
```

in the javap output.

We figured this out after Intermetrics suggested that we write and compile the desired Java class (with member functions stubbed out), run their java2ada tool to generate an Ada interface to that class, and then to change all the Import pragmas to Export in the package specification. This solved the problem. We did not do any further investigation of what conditions demand the use of pragma Export.

The use of pragma Export is probably obvious to experienced Ada 95 programmers, but not all AppletMagic users will be experienced Ada 95 programmers. An example such as the one above would be a useful addition to the applet writer's guide. The use of the convention Java_Constructor is already well documented, but more in the context of extending Java classes than in making Ada code callable by Java.

Once the Export pragmas were added, we were able to allocate an object within a new version of the Java test class driver.java and produce the same results as the abstract state machine implementation. We tested both a version that defined type World_Map_Obj directly and one that defined it as an extension of java.lang.Object, which we tested with both an Ada and a Java test driver.

As stated above, the extension of java.lang.Object met our fourth prototyping objective, to show that a Java class could be extended in Ada 95. We will not actually implement classes this way, so that our code can be used outside the Java context. All Java classes are extensions of class java.lang.Object in any case, and the byte code generated from AppletMagic does this whether or not this is explicitly done in the Ada source.

After implementing the World_Map class, we developed a Java subclass WorldMapDeg that extends WorldMap so that all angles would be expressed in degrees rather than radians. The Java source code follows:

```
public class WorldMapDeg extends World_Map {
  private final double RTD = 57.295;
  private final double DTR = 1.0 / RTD;

// setup functions -- convert values to radians for processing

  public void Set_Inclination (double i) {
    super.Set_Inclination (DTR*i);
  };

// A few similar modifiers removed for compactness

// selectors -- convert to degrees for output

  public double Longitude_Of () {
    return RTD * super.Longitude_Of();
  };

  public double Latitude_Of () {
    return RTD * super.Latitude_Of();
  };

};  // class WorldMapDeg
```

We then modified the test driver to allocate an object of class WorldMapDeg and to express all angles in degrees. Again, we obtained the same results as before.

The final test was to create an interface to a Java class and allocate an object in Ada. We had no real doubts that this could be done, because Intermetrics had already done that with the Java API packages such as java.awt. Nonetheless, it was useful to verify this for ourselves, and especially useful to learn the process needed to make the Java code visible to the Ada compiler.

The class we wrote was a simple class to write latitude and longitude as debug output, as shown below:

```
class MapData {
   // instance variable
   private boolean debugFlag;
   // constructor
   public MapData () {debugFlag = false;}

   // member functions
   public void initialize () {System.out.println ("Initialized!"); }
   public void finalize   () {System.out.println ("Finalized!");}
   public void debugOn  () {debugFlag = true;}
   public void debugOff () {debugFlag = false;}
   public void write (String time, double longitude, double latitude) {
      if (debugFlag) {
         System.out.println (time + " " + longitude + " " + latitude);
      }
   }  // write
}  // MapData
```

The sequence of UNIX commands for entering this into the Ada library is

```
%javac MapData.java
%java2ada MapData.class > MapData.ada
%adareg MapData.ada
```

The file MapData.ada provides the Ada package specification that corresponds to the Java class, and accesses the Java code using the Import pragma. The class WorldMap was then successfully modified to contain a reference to a MapData object in the tagged record defining its state, and to call the constructor for MapData.

The javac command invokes the Java compiler. The command java2ada is provided by AppletMagic to create Ada interfaces to Java code. The command adareg is used in the place of adajava when an interface to an existing class file is being entered into the Ada library. In this case, all the pieces of the command sequence needed to enter class MapData into the Ada library are documented. However, and end-to-end example like the one above would make the process clearer.

The additional work described in this section was not delivered as part of the applet, but was instrumental in convincing us that Ada 95 and Java would interact well. The only minor issue that remains is that record components defined in the tagged record, which is defined in the private part of the Ada package specification, map to public instance variables in the Java class. Since the Ada type should only be accessible in the package body, or in child library units, it would be more appropriate to produce protected instance variables. However, this is an issue that can be addressed by coding standards that discourage the direct use of such public variables in the Java code.

Verifier Problems

After the Ada code was compiled and tested as an application, the byte code files were delivered for integration with the graphics code. The applet was smoothly integrated and run under Sun's

Appletviewer, and with minor corrections produced the correct ground check. When the applet was first run from Netscape, however, it refused to run due to verifier errors.

Both the java command that runs applications and Sun's Appletviewer allowed code to be executed whether or not it passed byte-code verification. Intermetrics suggested running the Java command

```
%javap -verify -v -verify-verbose <classname>
```

on each class to determine if it would pass verification. We started by manually running this command for each class, but quickly decided to write a script that would test all the class files in a directory and all its subdirectories. This script is also included in the appendix to this paper. When we ran this script, we ran into three main errors.

First, the utilities package Standard_Types includes a type Various that is declared as an array of bytes, and uses Unchecked_Conversion to convert to and from standard scalar types such as Integer. This type was intended to contain spacecraft telemetry, and was made a byte array rather than a variant record because the data was passed to and from a user interface implemented in C, and the UI developers were too understaffed to design an interface that would handle Ada variant records. Since the applet was not producing or using telemetry, the Unchecked_Conversion instantiations were removed from the body of Standard_Types and the conversion functions were revised to return constant values.

Second, the code fragment

```
    when Numeric_Error | Constraint_Error =>
```

failed verification wherever it occurred. Removing Numeric_Error from this line solved the problem. However, Intermetrics fixed the compiler error before we needed to change every occurrence of this clause.

Finally, many of the utility functions contained uninitialized local variables, many of which caused verifier errors. Executing the verification script and redirecting the output to a file gave us a list of classes that failed verification. The only drawback of this script is that the Java verifier halts after the first failure, rather than scanning the whole class file. This forced several iterations for some files, so the corrections were both straightforward and tedious. If one is using AppletMagic to generate Java byte-codes, the style guideline of initializing local variables at the point of declaration should be strictly enforced.

After these problems, the orbit propagator applet was runnable from Netscape on the Sun or on a Macintosh, but not on a PC with Windows 95. In addition, the applet also encountered verifier errors on all our platforms when Internet Explorer was used as the browser. However, the error message from the verifier indicated that the failure was in the AppletMagic class interfaces.java.Ada_Exceptions. When Intermetrics made the corrections, the applet ran under all browsers except Internet Explorer on Windows machines. This problem still has not been resolved. The lesson here is that all Java virtual machine implementations are not created equal. We found it disturbing that the different verifiers were inconsistent, but the solution to this problem is beyond the control of both Goddard and Intermetrics.

Lessons Learned

1) *Establish a good relationship between user and vendor.* The project described in this paper would not have been successful without the timely answers to e-mail questions and response to bug reports provided by Intermetrics. While this seems to be an obvious point, we will nonetheless state it, since it was so critical to our success.

2) *Ada 83 was easily modified to use Java.* This implies that existing Ada code need not be replaced should an organization want to transition to Java, saving the organization money and freeing resources to do the interesting user interface and distributed system development in Java.

3) *Documentation should not assume expertise in Ada 95 and object-oriented programming.* As stated above, there is a great potential for use of AppletMagic with Ada 83. Also, there are still organizations using Ada that are just beginning to transition to Ada 95. Our organization has done some prototyping using Ada 95, but has not used it for full-scale development. Learning Ada 95 features was part of the process of learning how to use AppletMagic

4) *Upgrade documentation* to show how to use Ada code form Java, as well as Java from Ada. The FDD is probably not the only organization that would design a Java user interface to interact with Ada application code. It would also be useful to have step-by-step instructions in some parts of the applet writer's guide, such as importing Java code into an Ada library. It is easier for the new user to follow step by step instructions than to gather information from several different parts of the documentation.

5) *Provide more frequent interim releases of a beta product.* Bug fixes need to be made available to users as fast as they can be produced. The Internet user tends to be a very impatient person.

6) *When experimenting with new technology, start small.* We built a small applet and got it running for demonstration purposes. Once we had the basic version running, we were able to create several prototypes through a series of modifications that allowed a step-by-step verification of assumptions and proof of several important concepts for a low price. In the past, our organization has developed massive prototypes, which tend to take on too many new ideas at once and to take too long to produce tangible results. Concepts need to be proven on a small scale first.

7) *Don't try to use Java style for Ada code, or vice versa.* It is tempting to have a single programming style, but keeping the style language specific has the advantage of making it obvious when a call is being made to code in a different language. For example, the Java test driver would call World_Map.Initialize () to initialize the world map package, rather than worldMap.Initialize(). This would be a visual cue that the Java code is calling Ada. This is a minor point, but we have concluded that this is a reasonable approach, so we are documenting it here.

Future Directions

The success of the first version of the orbit propagator applet led to an expansion of the FDD's use of Java. This included upgrading the applet to read real satellite data from the Flight Dynamics Product Center (A server containing files with satellite position information in them) and propagate the orbit from the last saved position and velocity of the actual satellite. As stated in the introduction to this paper, this new version is available from the World Wide Web.

The second project that has started is the implementation of a larger system in Java. Two versions of a Real-Time Attitude Determination System (RTADS) are being developed for Java. The first is to be built as a new system, with scaled back specifications. The second is to take an existing RTADS and to rebuild it using AppletMagic. These two systems will share the same user interface, which is being implemented in Java. Thus there will be some adaptation needed to fit the Ada code into a different user interface. However, the bulk of the RTADS code should be recompilable without modification. Both of these projects are scheduled to finish in September of 1997.

Appendix: Source Files

This appendix contains the source code for the Ada package World_Map that is included in the orbit propagator applet, the Ada and Java test drivers used to test the package before integrating the Ada code with the Java graphics code, and the shell script "fix" that runs the verifier on all the compiled class files. The specification and body of World_Map have been shortened for presentation purposes. The full source code can be found on the FDD Java Web page at http://fdd.gsfc.nasa.gov/Java.html.

World Map

Package specification:

```
with Real_Types; use Real_Types;
package World_Map is
  -- types for time input
  subtype Year    is Integer range 1901..2099;
  subtype Month   is Integer range 1..12;
  subtype Day     is Integer range 1..31;
  subtype Hour    is Integer range 0..23;
  subtype Minute  is Integer range 0..59;
  subtype Seconds is Real    range 0.0..60.0;

  -- parameter setting operations
  -- time string in format yyyymmdd.hhmmssmmm
  procedure Set_Start_Time (To : in Wide_String);
  pragma Convention (Java, Set_Start_Time);  --****
  procedure Set_End_Time   (To : in Wide_String);
  pragma Convention (Java, Set_End_Time);    --****
  procedure Set_Stepsize   (To : in Real); -- seconds

  -- set parameters for orbit (some are removed to shorten example)
  procedure Set_Semimajor_Axis  (To : in Real ); -- kilometers
  procedure Set_Eccentricity    (To : in Real );
  procedure Set_Inclination     (To : in Real ); -- radians
  procedure Set_Right_Ascension (To : in Real ); -- radians
  procedure Set_Argument_of_Periapsis
                                (To : in Real ); -- radians
  procedure Set_Mean_Anomaly    (To : in Real ); -- radians

  -- modifiers
  procedure Initialize;
  procedure Propagate;
  -- selectors (some are removed to shorten example
  function Longitude_Of return Real;
  function Latitude_Of  return Real;
end World_Map;
```

Package Body:

```
with Math_Constants;     use Math_Constants;
with Analytical_Model;   use Analytical_Model;
with Time_Utilities;     use Time_Utilities;
with Time_IO;            use Time_IO;
with Real_Utilities;     use Real_Utilities;
with Linear_3D_Algebra;  use Linear_3D_Algebra;
with Utility_Exceptions; use Utility_Exceptions;
package body World_Map is

   Orbit        : Analytical_Model.Instance;
   Start_Time       : Time_Utilities.Time;
   Current_Time
                : Time_Utilities.Time;
   End_Time     : Time_Utilities.Time;
   Increment        : Time_Utilities.Elapsed_Time;
   V            : Vector3;
   R            : Real;
   Longitude         : Real;
   Latitude     : Real;
   GHA          : Real;

  -- World map functions
  procedure Set_Start_Time  (To : in Wide_String) is
  begin
    Start_Time := Time_Of (To_String(To) );
  end Set_Start_Time;

  procedure Set_End_Time  (To : in Wide_String) is
  begin
    End_Time := Time_Of (To_String(To) );
  end Set_End_Time;

  procedure Set_Stepsize  (To : in Real) is
  begin
    Increment := To ;
  end Set_Stepsize ;

  procedure Set_Semimajor_Axis  (To : in Real) is
  begin
    Set (Orbit, Initial_Semimajor_Axis, To) ;
  end Set_Semimajor_Axis ;

  -- some procedure bodies removed to shorten example
  procedure Set_Mean_Anomaly  (To : in Real) is
  begin
    Set (Orbit, Initial_Mean_Anomaly, To) ;
  end Set_Mean_Anomaly ;

  function GHA_Of (Current_Time : Time_Utilities.TIME) return Real is

    Sec_since_0hrs_UTC : Elapsed_Time := UTC_Seconds_of_Day_of_Time
                                             (Current_Time) ;
    GHA : Real ;
  begin
    GHA := (Pi/43200.0) * Sec_since_0hrs_UTC ;
    return GHA ;
  end GHA_Of ;
```

```
procedure Propagate (To_Time : in Time) is
    AD : RA_DEC ;
  begin
    Propagate (Orbit, To_Time) ;

    GHA        := GHA_Of (Current_Time) ;
    V          := Position_in_GCI_Of (Orbit, Current_Time);
    AD         := Cartesian_to_Ra_Dec (V) ;
    R          := AD.R ;

    Longitude := AD.Right_Ascension - GHA ;
    while Longitude < -Pi loop
      Longitude := Longitude + Two_Pi;
    end loop;
    Latitude  := AD.Declination ;
  exception
    when Argument_Error => raise Is_Zero_Vector;
  end Propagate;

  procedure Initialize is
  begin
    Initialize (Orbit) ;
    Current_Time := Start_Time ;
    Propagate (To_Time => Current_Time) ;
  end Initialize;

  procedure Propagate is
  begin
    Current_Time := Current_Time + Increment ;
    Propagate (To_Time => Current_Time) ;
  end Propagate ;

  function Longitude_Of return Real is
  begin
    return Longitude ;
  end Longitude_Of;

  function Latitude_Of return Real is
  begin
    return Latitude;
  end Latitude_Of;

end World_Map;
```

javap command output:

```
Compiled from world_map_.ada
public final class World_Map extends java.lang.Object {
    static void <clinit>();
    public static void Set_Start_Time(int,int,int,int,int,double);
    public static void Set_Start_Time(char []);
    public static void Set_End_Time(int,int,int,int,int,double);
    public static void Set_End_Time(char []);
    public static void Set_Stepsize(double);
    public static void Set_J2_Flag(boolean);
    public static void Set_Initial_Epoch(int,int,int,int,int,double);
    public static void Set_Initial_Epoch(char []);
    public static void Set_Semimajor_Axis(double);
    public static void Set_Eccentricity(double);
    public static void Set_Inclination(double);
    public static void Set_Right_Ascension(double);
    public static void Set_Argument_of_Periapsis(double);
    public static void Set_Mean_Anomaly(double);
    public static void Initialize();
    public static void Propagate();
    public static double Longitude_Of();
    public static double Latitude_Of();
    public static byte Current_Time_String(int [])[];
    public static char Current_Time_Of()[];
    public static boolean End_Time_Reached();
}
```

driver.java

This file contains the Java test code for package World_Map:

```
// test driver for world map application -- M. Stark
//      This application generates 3 hours worth of orbit data at a
//      60 second interval. The approach here is the starting point
//      for developing an applet to generate world map graphics.
import World_Map;
class driver {
    public static void main (String args[]) {
// variables used in computations
        double RTD = 57.295;
        double DTR = 1.0 / RTD;
        double latitude, longitude;
// strings for times
        String epoch = "19950502.0";
        String t0    = "19950502.0900";
        String tf    = "19950502.120000";
// start executable code
// set up epoch elements for orbit,
//     angles input in degrees by user, converted to radians before calls
        World_Map.Set_Initial_Epoch (epoch.toCharArray() );
        World_Map.Set_Semimajor_Axis (7000.0);  // kilometers
        World_Map.Set_Eccentricity (0.0);
        World_Map.Set_Inclination  (DTR * 28.5);
        World_Map.Set_Right_Ascension (DTR * 0.0);
        World_Map.Set_Argument_of_Periapsis (DTR * 0.0);
        World_Map.Set_Mean_Anomaly (DTR * 0.0);
// set up world map propagation
        World_Map.Set_Start_Time (t0.toCharArray() );
        World_Map.Set_End_Time   (tf.toCharArray() );
        World_Map.Set_Stepsize   (60.0);  // seconds
        World_Map.Set_J2_Flag    (false);
//  initialize and execute
        World_Map.Initialize();
        longitude = World_Map.Longitude_Of();
        latitude  = World_Map.Latitude_Of();
        System.out.println (RTD*longitude + "   " + RTD*latitude);
        while (!World_Map.End_Time_Reached() )
            {
            World_Map.Propagate();
//            ttag = timeTag();
            longitude = World_Map.Longitude_Of();
            latitude  = World_Map.Latitude_Of();
            System.out.println
              (ttag + RTD*longitude + "   " + RTD*latitude);
            }   //loop
}               // class driver
```

main.ada

This file contains the Ada test driver:

```
with World_Map; use World_Map;
with Real_Types;
use Real_Types;
with Text_IO; use Text_IO;
procedure Main is
  package Real_IO is new Text_IO.Float_IO(Real_Types.REAL);
  RTD : constant := 57.295;
  DTR : constant := 1.0 / RTD;
begin
  -- set up initial conditions
  Set_Initial_Epoch    (To => "19960502.0");
  Set_Semimajor_Axis   (To => 7000.0);
  Set_Eccentricity     (To => 0.0);
  Set_Inclination      (To => 28.5 * DTR);
  Set_Right_Ascension (To => 0.0);
  Set_Argument_Of_Periapsis (To => 0.0);
  Set_Mean_Anomaly     (To => 0.0);
  Set_Start_Time (To => "19960502.0900");
  Set_End_Time   (To => "19960502.1200");
  Set_Stepsize   (To => 60.0);
  Set_J2_Flag    (To => FALSE);

  -- initialize & execute
  Initialize;
  Text_IO.Put_Line ("Initial time is = " & Current_Time_String);
  Text_IO.New_Line;
  Text_IO.Put (Current_Time_String & "   ");
  Real_IO.Put (RTD * Longitude_Of, Fore =>4, Aft=>4, Exp=>0);
  Text_IO.Put ("   ");
  Real_IO.Put (RTD * Latitude_Of, Fore => 4, Aft=>4, Exp => 0);
  Text_IO.New_Line;
```

```
 while not End_Time_Reached loop
    Propagate;
    Text_IO.Put (Current_Time_String & "   ");
    Real_IO.Put (RTD * Longitude_Of, Fore =>4, Aft=>4, Exp=>0);
    Text_IO.Put ("   ");
    Real_IO.Put (RTD * Latitude_Of, Fore => 4, Aft=>4, Exp => 0);
    Text_IO.New_Line;
  end loop;
end Main;
```

script fix

This script searches the current directory and all subdirectories for files with extension ".class" and uses the javap command to determine if the class will pass byte-code verification within an applet.

```
#!/bin/sh

recurse() {
  for FILE in $1; do
    if [ "$FILE" != "$1" ]; then
      if [ -d $FILE ]; then
        recurse $FILE/"*"
      else
        OUTPUT=`echo $FILE | grep ".class" | wc -l`
        if [ $OUTPUT != "0" ]; then
          #PREFIX=`echo $FILE | sed 's/^.*\///'`
          PREFIX=`echo $FILE`
          PREFIX=`echo $PREFIX | sed 's/\.class$//'`
          PREFIX=`echo $PREFIX | sed 's/\//./'`
          #echo "Updating $FILE ... $PREFIX"
          javap -verify -v -verify-verbose $PREFIX
        fi
      fi
    fi
  done
}

recurse "*"
```

References

1. Flight Dynamics Distributed Systems (FDDS) Generalized Support Software (GSS) Functional Specification, Revision 1, Update 4. Document 553-FDD-93/046R1UD4, September 1996.

2. file appletwriters_guide.txt, documentation provided on line with AppletMagic, August 1996.

360752
8P
S12-61
415 776
CLOSE

The Generalized Support Software (GSS) Domain Engineering Process: An Object-Oriented Implementation and Reuse Success at Goddard Space Flight Center

Steven Condon[1], Robert Hendrick[1], Michael E. Stark[2], Warren Steger[3]

Abstract

The Flight Dynamics Division (FDD) of NASA's Goddard Space Flight Center (GSFC) recently embarked on a far-reaching revision of its process for developing and maintaining satellite support software. The new process relies on an object-oriented software development method supported by a domain specific library of generalized components. This Generalized Support Software (GSS) Domain Engineering Process is currently in use at the NASA GSFC Software Engineering Laboratory (SEL). The key facets of the GSS process are (1) an architecture for rapid deployment of FDD applications, (2) a reuse asset library for FDD classes, and (3) a paradigm shift from developing software to configuring software for mission support. This paper describes the GSS architecture and process, results of fielding the first applications, lessons learned, and future directions

[1] Computer Sciences Corporation, 10110 Aerospace Rd., Lanham-Seabrook, MD 20706; scondon@csc.com, rhendri@cscmail.csc.com
[2] NASA Goddard Space Flight Center, Greenbelt, MD 20771; michael.e.stark@gsfc.nasa.gov
[3] Formerly with Computer Sciences Corporation, now with Bell Atlantic; warren.l.steger@bell-atl.com

Introduction

The FDD began, about 3.5-4 years ago, to effect a shift from *developing* applications to *configuring* applications out of generalized, reusable assets, which include code components, specifications, tools, and standards. These efforts are an outgrowth of 20 years of software experimentation at the FDD that have been guided, studied, documented, and nurtured by the Software Engineering Laboratory (SEL). The SEL is a virtual organization which consists of FDD civil servants, CSC contractors supporting the FDD, and representatives from the Computer Science Department of the University of Maryland at College Park.

The effort to move toward configuring applications resulted in the Generalized Support Software (GSS) Domain Engineering Process. This has involved creating an architecture for designing and developing the reusable assets and for configuring applications from them, as well as evolving a process for the team members to follow in doing this work.

This experience report shares the FDD's motivations and goals on this effort, explains how the GSS Domain Engineering Process operates today, points out some of the benefits of the process, provides some lessons learned, and leaves you with an idea of how the GSS process is evolving.

The FDD and SEL

Over the past decade, the FDD has usually consisted of about 100 civil servants supported by 300-400 CSC and subcontractor personnel.

The mission of the FDD is to build, deploy, and maintain space ground systems for NASA science missions, with emphasis on earth orbiting satellites.

The particular domain of the FDD systems is spacecraft flight dynamics. Flight dynamics applications are essentially scientific data processing systems: some are institutional (support multiple missions) and others are mission-specific (we need to build a new one for each spacecraft).

The GSS was developed *first* to support the attitude determination subdomain (*attitude* refers to a spacecraft's orientation in space). These applications process real-time and non-real-time sensor measurements from telemetry data for determining spacecraft attitude. This report focuses on this GSS experience. (More recently the GSS has been broadened to provide interactive tools that help analysts plan mission profiles and on-orbit maneuvers. Even more recently the GSS has begun expanding to provide embedded applications that perform onboard autonomous navigation using GPS signals.)

Approximately 40% of FDD personnel are *analysts,* trained in physics and mathematics, who write requirements for all FDD software. Another 40% of FDD personnel make up the *project organization* which develops, tests, delivers, and maintains software systems for the FDD. The SEL interface with the project organization and is composed of 15-20 government, CSC, and University of Maryland personnel. The SEL receives software metrics and experience data from the project organization, stores these in its database, analyzes portions of the data, and packages the results as study reports, estimation models, best practices, and instructional courses, which serve to benefit the project organization. The SEL also plays a large role in deciding which new software practices to test out in the FDD. The SEL has been in existence at the FDD for over 20 years.

Reuse History

When the FDD began work on the GSS process, we had a fairly solid foundation of experience in reuse. Our efforts were focused largely on the class of mission-specific applications, where reuse would have the most impact.

A little over 10 years ago, the FDD was building these systems in a FORTRAN mainframe environment, achieving a modest level of reuse of very low level utilities.

In the mid-1980s, we began exploring object-oriented (OO) design and Ada with the goal of increasing reuse levels and cutting down cost and cycle time. We learned a great deal about using OO and Ada generics for one particular type of application, a simulation test tool that we began developing on an Ada-friendly platform—the DEC VAX.

The bulk of our mission-specific applications, the AGSSs, however, were still FORTRAN mainframe. We were unable to transfer our Ada practices to the mainframe because we could not find adequate Ada tools for the mainframe environment. In lieu of this, we tried to apply some domain engineering and OO concepts to this environment. We had some success with this approach, but the results were not truly "generalized" and the systems grew with each new mission and became cumbersome to maintain. Nonetheless, these were all valuable experiences on which we were able to build.

Motivation for GSS

Despite our prior achievements in reuse (and the cost reductions that came with them), in the early 1990s the FDD was continuing to feel increasing pressure to do things "better, faster, and cheaper."

At the same time, advances in technology were driving FDD customers away from mainframe solutions. The FDD decided to move all of its flight dynamics systems (both institutional and mission-specific—approximately 6 million lines of code) to a distributed workstation environment.

Because COTS solutions were generally not available in this specialized domain, we hit on the approach of building our own library of COTS-like objects that would be generalized and reusable across the entire flight dynamics domain.

We also saw the move to the workstations as an opportunity to eliminate much of the duplication of functionality that had built up in 30 years' worth of the legacy systems. These forces provided the motivation and goals for the GSS process.

GSS Architecture Hierarchy

Examples of GSS classes are:

- three-axis stabilized attitude model

- sun-pointing coordinate system model
- V-slit sun sensor model

A GSS class is implemented as multiple Ada83 generics. Thus it is not the same as the class construct in C++. Similar GSS classes are grouped into *categories,* along with rules for using member classes for mission support. A category defines the minimum functional interface for each of its classes, making its classes plugable and swapable. Several categories are grouped into a *subdomain,* which is a group that specifies the functionality in a specific high-level area of the overall problem domain.

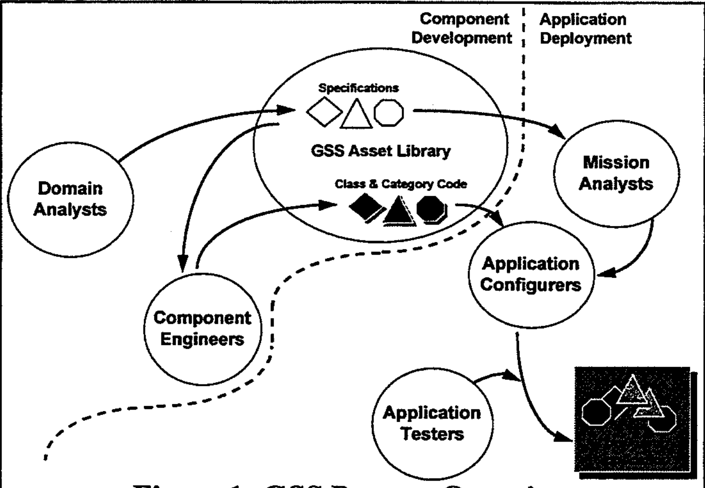


Figure 1: GSS Process Overview

GSS Overview

Figure 1 shows an overall model of the GSS component development and application deployment process.

Central to the model is the asset library. It contains not only the generalized software components, but also their specifications, as well as tools that are used at various points in the process—a code generator for producing the generalized components and various tools for automating the application configuration process. It also contains the standards that the teams follow for writing the generalized specifications and for implementing the generalized components.

On the left side of the dashed line in Figure 1, we have producers of assets (the domain analysts and component engineers). On the right we have the consumers of assets (mission analysts and application configurers) plus the testers.

The GSS has evolved and continues to operate through a series of overlapping phases. Each of the five teams shown in Figure 1 is primarily responsible for the activities of one such phase. These phases are:

- Domain Engineering
- Component Engineering
- Application Definition
- Application Configuration
- Application Testing

The first two of these phases have a startup portion, an initial portion, and a sustaining portion. These two phases are responsible for stocking the reuse asset library. The other three phases (making up *application deployment*) are repeated once for each application that is configured from the library assets.

Domain Engineering Phase

The GSS process begins with the domain engineering phase. In domain engineering, the main goals of the startup stage are to create a high-level model of the domain and to generate standards for documenting the architecture and the specifications.

This is done by defining the boundaries of the domain, in terms of a superset of requirements that encompasses all of the applications one intends to build from the generalized components, anticipating, where possible, requirements for future applications.

The engineers then define, at least partially, several of the subdomains. This often takes several iterations of looking at different ways to group the generalized functionality, sometimes going back to redefine the domain boundaries. In the process, the overall architecture is defined and standards for specifying the architecture components (categories and classes) are evolved.

Core subdomains are used across all applications in the domain. *Application subdomains* are used by a given application area.

Once the domain has been partitioned into subdomains and specification standards have been developed, the domain engineers "flesh out" the specifications for a particular class of applications (e.g., telemetry simulators). They define the categories and classes in both core and application subdomains needed to implement the application.

This is not strictly a top-down process and often involves considerable refinement of the subdomain and category boundaries. As each successive application type is worked, however, the amount of refinement begins to decrease.

Once all applications have been tackled, the sustaining domain analysis phase supports the continued refinement of the architecture. It also supports the addition of new functionality as new applications require development of new generalized components or further generalization of existing components (e.g., in our domain, when a spacecraft with a new type of sensor needs applications built).

Component Engineering Phase

After the domain engineering phase is well underway, the component engineering phase can begin. As in domain engineering, the preliminary stages of the component engineering phase focus on developing standards to be followed in implementing the library assets (classes and categories).

This involves creating a design for implementing the assets as specified by the domain engineers, along with a software architecture to support the configuration of the assets into applications. (Recall that classes are implemented with multiple generic packages in Ada.)

The standard model for interfacing with a GUI is also defined during this phase, along with specific coding standards that the developers will follow during implementation.

Although much of the "design" of the library assets and applications is embodied in the classifications and specifications produced by the domain engineers, their work can be thought of more as creating a *logical* design, whereas the design work of the component engineers can be thought of as creating the *physical* design of how assets and applications are implemented in terms of language units, data interfaces, and GUI interfaces.

The definition of rigorous specification and implementation standards makes it possible to use code generation technology to create much of the structural code when implementing the assets.

As the domain engineers are refining their partitioning of the domain by specifying assets for the first target application, the component engineers begin implementing the assets and populating the library. Going through this process for the initial application helps the component engineers refine the physical design.

The code generator mentioned earlier is also refined and validated during this phase. On GSS, the code generator produces approximately 75% of the code required to implement a class. The component engineers translate the specifications into a code generator input language, generate the structural code, and then implement the detailed algorithms from the equations in the functional specifications. Classes and categories are inspected and unit tested, at least initially (more on this in lessons learned).

Once a critical mass of components has been created, the component engineers prototype an application (or a subset of an application) to validate the design. This allows further refinement of the architecture before full-scale production of assets gets underway.

Once the library is populated with assets for all target applications, the sustaining component engineering phase begins. This is the counterpart to sustaining domain engineering—implementing specification changes, error corrections, new requirements, etc.

Application Definition Phase

The next three phases are repeated once for each application that is configured from GSS assets. First among these is the application definition phase. The first step in this phase, defining system operations concepts and requirements, is the same as in traditional software engineering.

Once requirements are defined, however, the mission analysts then determine what types of applications they need to meet them. For example, if they need attitude determination, do they need a real-time system, a non-real-time system, or both?

Once an application type has been chosen, the set of objects required to build the application for the spacecraft's particular configuration is identified.

Spacecraft-unique default parameter values are then specified for all of the identified objects, as well as any objects required by tracing dependencies.

Finally, the contents and layout of user interface displays and reports are defined.

One other consideration in the application definition phase is whether the application requires any specific functionality that is not available from the library assets. If so, the domain engineers determine whether that functionality should be implemented as a new, generalized library asset, or whether unique code should be built for use on this application only.

Application Configuration Phase

We now come to the payoff in the GSS process, the application configuration phase. The speed, efficiency, and repeatability of this phase is one of the most important measures of the success of the GSS Domain Engineering Process.

These are the basic steps in the configuration process.

- Build object table
- Generate "glueware"
- Configure driver
- Create database of parameter default values for initializing the application
- Create database of display formats used by GUI

Although we began doing these steps manually for the first few configurations, it quickly became clear that we could develop tools to automate many of them.

In addition to the steps listed above, any required mission-specific functionality is developed during the configuration phase.

Application Testing Phase

The last phase is the application testing phase. The main activities are developing the acceptance test plan, conducting the acceptance test, and evaluating the test results. We found that testing GSS-based applications required some changes in how we carried out these activities.

Our testers first tried a "white box" approach, similar to what they had used for testing structured FORTRAN systems. Using this approach, the first two GSS applications for the first mission were tested against low-level specifications. Each application required about 1200 test items. Testers used a debugger tool to examine specific algorithms and verify intermediate values. The testing proved to be very tedious, very expensive, and—because the code contained so few errors—very wasteful.

After these two applications were tested , the SEL led a workshop to develop alternative testing approaches. From this effort came the GSS "black box" testing approach. Now applications are tested against the user's requirements (about 200 test items per application). Initial feedback reveals that the testers are much happier with this approach and the cost is greatly diminished. We are continuing to study this testing process.

Results

To date, we have used the GSS assets to configure applications for two missions—5 applications for the first mission and 2 applications for the second mission. We'll use these applications to illustrate the results we're seeing.

With GSS we are reusing the same generalized classes across multiple applications supporting the same mission. For the first GSS mission more than half of the classes in each of the five applications came from the core subdomain, with the rest drawn primarily from the various application subdomains.

The second mission to use GSS components required two attitude applications, each of which required a very small amount of functionality not already available in the library. We were able to meet the mission requirements with systems that, when considered in total, consisted of 98% reused code.

Increased reuse usually suggests reduced production costs. An average AGSS produced during the era of low code reuse (i.e., prior to 1985) cost about 41,000 hours to develop and test. During the era from 1985 to 1993—when AGSSs were still coded in FORTRAN, but a code reuse library was actively maintained and utilized for AGSSs—the average cost per AGSS was reduced about 65% , to 14,000 hours. With the first GSS-based AGSS, this new cost has been reduced about 65% again, to 4600 hours. For the second GSS-based AGSS the results were even better:

cost was only about 1500 hours, a 90% reduction from the FORTRAN reuse era.

In order to achieve these cost reductions in the 2nd and 3rd eras, it was necessary to invest development and testing hours in creating the reuse libraries. The FORTRAN reuse library, created for AGSSs required 95,000 hours, about twice the cost of an AGSS in the 1st era. The Ada reuse library, created for telemetry simulators (at about the same time), required 15,000 hours. The GSS library, which supports both AGSSs and simulators, required 40,000 hours, about 3 times the cost of an AGSS in the 2nd era.

In addition, we know that the GSS library required 36,000 hours from domain analysts, to define and write the GSS functional specifications; these are shown as a white bar. Because of limitations in the data collection methods in earlier times, we do not know how many specifications hours went into the earlier FORTRAN or Ada reuse libraries.

Lessons Learned

The following actions proved very beneficial to us.

1. Planning for iterations during the startup subphases of domain engineering and component engineering until convergence on a workable design.
2. Prototyping the design concepts along the way.
3. Planning extra schedule in the early builds of component development for tweaking the concepts, refining the code generator, etc.
4. Freezing changes until we got to the point where we could begin to build something, in order to avoid spinning our wheels with continual rework.
5. Scheduling a separate build to go back and bring all the assets up to the latest version of the design and standards.
6. Setting rigorous specification and implementation standards and developing a code generator to cut the costs of asset production considerably.
7. Automating many of the configuration steps.
8. Dropping unit testing in favor of unit inspections. (Standalone unit testing of generalized classes is very expensive. We found inspections to be more efficient at finding errors in generalized classes. Once we started using the classes in applications, we found testing the classes in context to be effective and efficient for finding the remaining errors.)

With the benefit of hindsight, we wish we had done the following:

The FDD decided to use a home-grown GUI, which was under development in parallel with the GSS. Consequently, so we didn't have anything to get in front of the users early on. In our environment, end users are computer literates who are exposed to slick GUIs on various COTS products. Look and feel is very important to them. Our end users have been very slow to warm up to this GUI.

We didn't do a very good job of "selling" the technology to the mission analysts whom we expected to perform application configuration. The fact that we were using terms like classes, subdomains, and behavior models didn't help—it was a foreign language to them.

We noted above that we automated the configuration process. This was more of an afterthought after finding out how painful it was to do manually the first time around. If we had it to do again, we would have planned and developed configuration tools earlier. Ideally, we would create GUI-based tools that would allow the application definition and configuration phases to be combined, while at the same time bridging the gap between our object terminology and the functional way of thinking to which our mission analysts are accustomed. This would have helped with the "selling."

Another group to whom objects and classes were foreign words was our independent application testers. Working with them early on to establish more of a "black box" testing philosophy and to leverage the reuse of previously validated assets would have saved us a lot of heartache.

Future Directions

Our immediate plans are to continue populating the library to expand the use of the assets into other application domains. The move into the domain of mission and maneuver planning is well under way. For this we are developing classes in C++. We are also beginning to expand into the orbit & navigation domain.

As discussed under lessons learned, we would like to create a GUI-based interface so that users can create applications by dragging and dropping icons in a more intuitive and automated fashion.

Omit to End

STANDARD BIBLIOGRAPHY OF SEL LITERATURE

The technical papers, memorandums, and documents listed in this bibliography are organized into two groups. The first group is composed of documents issued by the Software Engineering Laboratory (SEL) during its research and development activities. The second group includes materials that were published elsewhere but pertain to SEL activities. The *Annotated Bibliography of Software Engineering Laboratory Literature* contains an abstract for each document and is available via the SEL Products Page at http://fdd.gsfc.nasa.gov/selprods.html.

SEL-ORIGINATED DOCUMENTS

SEL-76-001, *Proceedings From the First Summer Software Engineering Workshop,* August 1976

SEL-77-002, *Proceedings From the Second Summer Software Engineering Workshop,* September 1977

SEL-78-005, *Proceedings From the Third Summer Software Engineering Workshop,* September 1978

SEL-78-006, *GSFC Software Engineering Research Requirements Analysis Study,* P. A. Scheffer and C. E. Velez, November 1978

SEL-78-007, *Applicability of the Rayleigh Curve to the SEL Environment,* T. E. Mapp, December 1978

SEL-78-302, *FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 3),* W. J. Decker, W. A. Taylor, et al., July 1986

SEL-79-002, *The Software Engineering Laboratory: Relationship Equations,* K. Freburger and V. R. Basili, May 1979

SEL-79-004, *Evaluation of the Caine, Farber, and Gordon Program Design Language (PDL) in the Goddard Space Flight Center (GSFC) Code 580 Software Design Environment,* C. E. Goorevich, A. L. Green, and W. J. Decker, September 1979

SEL-79-005, *Proceedings From the Fourth Summer Software Engineering Workshop,* November 1979

SEL-80-002, *Multi-Level Expression Design Language-Requirement Level (MEDL-R) System Evaluation,* W. J. Decker and C. E. Goorevich, May 1980

SEL-80-005, *A Study of the Musa Reliability Model,* A. M. Miller, November 1980

SEL-80-006, *Proceedings From the Fifth Annual Software Engineering Workshop,* November 1980

SEL-80-007, *An Appraisal of Selected Cost/Resource Estimation Models for Software Systems,* J. F. Cook and F. E. McGarry, December 1980

SEL-80-008, *Tutorial on Models and Metrics for Software Management and Engineering*, V. R. Basili, 1980

SEL-81-011, *Evaluating Software Development by Analysis of Change Data*, D. M. Weiss, November 1981

SEL-81-012, *The Rayleigh Curve as a Model for Effort Distribution Over the Life of Medium Scale Software Systems*, G. O. Picasso, December 1981

SEL-81-013, *Proceedings of the Sixth Annual Software Engineering Workshop*, December 1981

SEL-81-014, *Automated Collection of Software Engineering Data in the Software Engineering Laboratory (SEL)*, A. L. Green, W. J. Decker, and F. E. McGarry, September 1981

SEL-81-101, *Guide to Data Collection*, V. E. Church, D. N. Card, F. E. McGarry, et al., August 1982

SEL-81-110, *Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics*, G. Page, F. E. McGarry, and D. N. Card, June 1985

SEL-81-305, *Recommended Approach to Software Development*, L. Landis, S. Waligora, F. E. McGarry, et al., June 1992

SEL-81-305SP1, *Ada Developers' Supplement to the Recommended Approach*, R. Kester and L. Landis, November 1993

SEL-82-001, *Evaluation of Management Measures of Software Development*, G. Page, D. N. Card, and F. E. McGarry, September 1982, vols. 1 and 2

SEL-82-004, *Collected Software Engineering Papers: Volume 1*, July 1982

SEL-82-007, *Proceedings of the Seventh Annual Software Engineering Workshop*, December 1982

SEL-82-008, *Evaluating Software Development by Analysis of Changes: The Data From the Software Engineering Laboratory*, V. R. Basili and D. M. Weiss, December 1982

SEL-82-102, *FORTRAN Static Source Code Analyzer Program (SAP) System Description (Revision 1)*, W. A. Taylor and W. J. Decker, April 1985

SEL-82-105, *Glossary of Software Engineering Laboratory Terms*, T. A. Babst, M. G. Rohleder, and F. E. McGarry, October 1983

SEL-82-1306, *Annotated Bibliography of Software Engineering Laboratory Literature*, D. Kistler, J. Bristow, and D. Smith, November 1994

SEL-83-001, *An Approach to Software Cost Estimation*, F. E. McGarry, G. Page, D. N. Card, et al., February 1984

SEL-83-002, *Measures and Metrics for Software Development*, D. N. Card, F. E. McGarry, G. Page, et al., March 1984

SEL-83-003, *Collected Software Engineering Papers: Volume II*, November 1983

SEL-83-007, *Proceedings of the Eighth Annual Software Engineering Workshop*, November 1983

SEL-83-106, *Monitoring Software Development Through Dynamic Variables (Revision 1)*, C. W. Doerflinger, November 1989

SEL-84-003, *Investigation of Specification Measures for the Software Engineering Laboratory (SEL)*, W. W. Agresti, V. E. Church, and F. E. McGarry, December 1984

SEL-84-004, *Proceedings of the Ninth Annual Software Engineering Workshop*, November 1984

SEL-84-101, *Manager's Handbook for Software Development (Revision 1)*, L. Landis, F. E. McGarry, S. Waligora, et al., November 1990

SEL-85-001, *A Comparison of Software Verification Techniques*, D. N. Card, R. W. Selby, Jr., F. E. McGarry, et al., April 1985

SEL-85-002, *Ada Training Evaluation and Recommendations From the Gamma Ray Observatory Ada Development Team*, R. Murphy and M. Stark, October 1985

SEL-85-003, *Collected Software Engineering Papers: Volume III*, November 1985

SEL-85-004, *Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics*, R. W. Selby, Jr., and V. R. Basili, May 1985

SEL-85-005, *Software Verification and Testing*, D. N. Card, E. Edwards, F. McGarry, and C. Antle, December 1985

SEL-85-006, *Proceedings of the Tenth Annual Software Engineering Workshop*, December 1985

SEL-86-001, *Programmer's Handbook for Flight Dynamics Software Development*, R. Wood and E. Edwards, March 1986

SEL-86-002, *General Object-Oriented Software Development*, E. Seidewitz and M. Stark, August 1986

SEL-86-003, *Flight Dynamics System Software Development Environment (FDS/SDE) Tutorial*, J. Buell and P. Myers, July 1986

SEL-86-004, *Collected Software Engineering Papers: Volume IV*, November 1986

SEL-86-005, *Measuring Software Design*, D. N. Card et al., November 1986

SEL-86-006, *Proceedings of the Eleventh Annual Software Engineering Workshop*, December 1986

SEL-87-001, *Product Assurance Policies and Procedures for Flight Dynamics Software Development*, S. Perry et al., March 1987

SEL-87-002, *Ada® Style Guide (Version 1.1)*, E. Seidewitz et al., May 1987

SEL-87-003, *Guidelines for Applying the Composite Specification Model (CSM)*, W. W. Agresti, June 1987

SEL-87-004, *Assessing the Ada® Design Process and Its Implications: A Case Study*, S. Godfrey, C. Brophy, et al., July 1987

SEL-87-009, *Collected Software Engineering Papers: Volume V*, November 1987

SEL-87-010, *Proceedings of the Twelfth Annual Software Engineering Workshop*, December 1987

SEL-88-001, *System Testing of a Production Ada Project: The GRODY Study*, J. Seigle, L. Esker, and Y. Shi, November 1988

SEL-88-002, *Collected Software Engineering Papers: Volume VI*, November 1988

SEL-88-003, *Evolution of Ada Technology in the Flight Dynamics Area: Design Phase Analysis*, K. Quimby and L. Esker, December 1988

SEL-88-004, *Proceedings of the Thirteenth Annual Software Engineering Workshop*, November 1988

SEL-88-005, *Proceedings of the First NASA Ada User's Symposium*, December 1988

SEL-89-002, *Implementation of a Production Ada Project: The GRODY Study*, S. Godfrey and C. Brophy, September 1989

SEL-89-004, *Evolution of Ada Technology in the Flight Dynamics Area: Implementation/Testing Phase Analysis*, K. Quimby, L. Esker, L. Smith, M. Stark, and F. McGarry, November 1989

SEL-89-005, *Lessons Learned in the Transition to Ada From FORTRAN at NASA/Goddard*, C. Brophy, November 1989

SEL-89-006, *Collected Software Engineering Papers: Volume VII*, November 1989

SEL-89-007, *Proceedings of the Fourteenth Annual Software Engineering Workshop*, November 1989

SEL-89-008, *Proceedings of the Second NASA Ada Users' Symposium*, November 1989

SEL-89-103, *Software Management Environment (SME) Concepts and Architecture (Revision 1)*, R. Hendrick, D. Kistler, and J. Valett, September 1992

SEL-89-301, *Software Engineering Laborary (SEL) Database Organization and User's Guide (Revision 3)*, L. Morusiewicz, February 1995

SEL-90-001, *Database Access Manager for the Software Engineering Laboratory (DAMSEL) User's Guide*, M. Buhler, K. Pumphrey, and D. Spiegel, March 1990

SEL-90-002, *The Cleanroom Case Study in the Software Engineering Laboratory: Project Description and Early Analysis*, S. Green et al., March 1990

SEL-90-003, *A Study of the Portability of an Ada System in the Software Engineering Laboratory (SEL)*, L. O. Jun and S. R. Valett, June 1990

SEL-90-004, *Gamma Ray Observatory Dynamics Simulator in Ada (GRODY) Experiment Summary*, T. McDermott and M. Stark, September 1990

SEL-90-005, *Collected Software Engineering Papers: Volume VIII*, November 1990

SEL-90-006, *Proceedings of the Fifteenth Annual Software Engineering Workshop*, November 1990

SEL-91-001, *Software Engineering Laboratory (SEL) Relationships, Models, and Management Rules*, W. Decker, R. Hendrick, and J. Valett, February 1991

SEL-91-003, *Software Engineering Laboratory (SEL) Ada Performance Study Report*, E. W. Booth and M. E. Stark, July 1991

SEL-91-004, *Software Engineering Laboratory (SEL) Cleanroom Process Model*, S. Green, November 1991

SEL-91-005, *Collected Software Engineering Papers: Volume IX*, November 1991

SEL-91-006, *Proceedings of the Sixteenth Annual Software Engineering Workshop*, December 1991

SEL-91-102, *Software Engineering Laboratory (SEL) Data and Information Policy (Revision 1)*, F. McGarry, August 1991

SEL-92-001, *Software Management Environment (SME) Installation Guide*, D. Kistler and K. Jeletic, January 1992

SEL-92-002, *Data Collection Procedures for the Software Engineering Laboratory (SEL) Database*, G. Heller, J. Valett, and M. Wild, March 1992

SEL-92-003, *Collected Software Engineering Papers: Volume X*, November 1992

SEL-92-004, *Proceedings of the Seventeenth Annual Software Engineering Workshop*, December 1992

SEL-93-001, *Collected Software Engineering Papers: Volume XI*, November 1993

SEL-93-002, *Cost and Schedule Estimation Study Report*, S. Condon, M. Regardie, M. Stark, et al., November 1993

SEL-93-003, *Proceedings of the Eighteenth Annual Software Engineering Workshop*, December 1993

SEL-94-001, *Software Management Environment (SME) Components and Algorithms*, R. Hendrick, D. Kistler, and J. Valett, February 1994

SEL-94-003, *C Style Guide*, J. Doland and J. Valett, August 1994

SEL-94-004, *Collected Software Engineering Papers: Volume XII*, November 1994

SEL-94-005, *An Overview of the Software Engineering Laboratory*, F. McGarry, G. Page, V. R. Basili, et al., December 1994

SEL-94-006, *Proceedings of the Nineteenth Annual Software Engineering Workshop*, December 1994

SEL-94-102, *Software Measurement Guidebook (Revision 1)*, M. Bassman, F. McGarry, R. Pajerski, June 1995

SEL-95-001, *Impact of Ada in the Flight Dynamics Division at Goddard Space Flight Center*, S. Waligora, J. Bailey, M. Stark, March 1995

SEL-95-003, *Collected Software Engineering Papers: Volume XIII*, November 1995

SEL-95-004, *Proceedings of the Twentieth Annual Software Engineering Workshop*, December 1995

SEL-96-001, *Collected Software Engineering Papers: Volume XIV*, October 1996

SEL-97-002, *Collected Software Engineering Papers: Volume XV*, October 1997

SEL-RELATED LITERATURE

[10]Abd-El-Hafiz, S. K., V. R. Basili, and G. Caldiera, "Towards Automated Support for Extraction of Reusable Components," *Proceedings of the IEEE Conference on Software Maintenance-1991 (CSM 91)*, October 1991

[15]Abd-El-Hafiz, S. K., V. R. Basili, "A Knowledge-Based Approach to the Analysis of Loops," *IEEE Transactions on Software Engineering*, May 1996

[4]Agresti, W. W., V. E. Church, D. N. Card, and P. L. Lo, "Designing With Ada for Satellite Simulation: A Case Study," *Proceedings of the First International Symposium on Ada for the NASA Space Station*, June 1986

[2]Agresti, W. W., F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," *Program Transformation and Programming Environments*. New York: Springer-Verlag, 1984

[1]Bailey, J. W., and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," *Proceedings of the Fifth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1981

[8]Bailey, J. W., and V. R. Basili, "Software Reclamation: Improving Post-Development Reusability," *Proceedings of the Eighth Annual National Conference on Ada Technology*, March 1990

[10]Bailey, J. W., and V. R. Basili, "The Software-Cycle Model for Re-Engineering and Reuse," *Proceedings of the ACM Tri-Ada 91 Conference*, October 1991

[1]Basili, V. R., “Models and Metrics for Software Management and Engineering,” *ASME Advances in Computer Technology*, January 1980, vol. 1

Basili, V. R., *Tutorial on Models and Metrics for Software Management and Engineering.* New York: IEEE Computer Society Press, 1980 (also designated SEL-80-008)

[3]Basili, V. R., “Quantitative Evaluation of Software Methodology,” *Proceedings of the First Pan-Pacific Computer Conference*, September 1985

[7]Basili, V. R., *Maintenance = Reuse-Oriented Software Development*, University of Maryland, Technical Report TR-2244, May 1989

[7]Basili, V. R., *Software Development: A Paradigm for the Future*, University of Maryland, Technical Report TR-2263, June 1989

[8]Basili, V. R., “Viewing Maintenance of Reuse-Oriented Software Development,” *IEEE Software*, January 1990

[13]Basili, V. R., "The Experience Factory and Its Relationship to Other Quality Approaches," *Advances in Computers*, vol. 41, Academic Press, Incorporated, 1995

[1]Basili, V. R., and J. Beane, “Can the Parr Curve Help With Manpower Distribution and Resource Estimation Problems?,” *Journal of Systems and Software*, February 1981, vol. 2, no. 1

[13]Basili, V. R., L. Briand, and W. L. Melo, *A Validation of Object-Oriented Design Metrics*, University of Maryland, Computer Science Technical Report, CS-TR-3443, UMIACS-TR-95-40, April 1995

[13]Basili, V. R., and G. Caldiera, *The Experience Factory Strategy and Practice*, University of Maryland, Computer Science Technical Report, CS-TR-3483, UMIACS-TR-95-67, May 1995

[9]Basili, V. R., G. Caldiera, and G. Cantone, “A Reference Architecture for the Component Factory,” *ACM Transactions on Software Engineering and Methodology,* January 1992

[10]Basili, V. R., G. Caldiera, F. McGarry, et al., “The Software Engineering Laboratory—An Operational Software Experience Factory,” *Proceedings of the Fourteenth International Conference on Software Engineering (ICSE 92)*, May 1992

[1]Basili, V. R., and K. Freburger, “Programming Measurement and Estimation in the Software Engineering Laboratory,” *Journal of Systems and Software*, February 1981, vol. 2, no. 1

[12]Basili, V. R., and S. Green, "Software Process Evolution at the SEL," *IEEE Software*, July 1994, pp. 58–66

[3]Basili, V. R., and N. M. Panlilio-Yap, “Finding Relationships Between Effort and Other Variables in the SEL,” *Proceedings of the International Computer Software and Applications Conference*, October 1985

[4]Basili, V. R., and D. Patnaik, *A Study on Fault Prediction and Reliability Assessment in the SEL Environment*, University of Maryland, Technical Report TR-1699, August 1986

[2]Basili, V. R., and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Communications of the ACM*, January 1984, vol. 27, no. 1

[1]Basili, V. R., and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory," *Proceedings of the ACM SIGMETRICS Symposium/Workshop: Quality Metrics*, March 1981

[3]Basili, V. R., and C. L. Ramsey, "ARROWSMITH-P—A Prototype Expert System for Software Engineering Management," *Proceedings of the IEEE/MITRE Expert Systems in Government Symposium*, October 1985

Basili, V. R., and J. Ramsey, *Structural Coverage of Functional Testing*, University of Maryland, Technical Report TR-1442, September 1984

Basili, V. R., and R. Reiter, "Evaluating Automatable Measures for Software Development," *Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity, and Cost*. New York: IEEE Computer Society Press, 1979

[5]Basili, V. R., and H. D. Rombach, "Tailoring the Software Process to Project Goals and Environments," *Proceedings of the 9th International Conference on Software Engineering*, March 1987

[5]Basili, V. R., and H. D. Rombach, "TAME: Tailoring an Ada Measurement Environment," *Proceedings of the Joint Ada Conference*, March 1987

[5]Basili, V. R., and H. D. Rombach, "TAME: Integrating Measurement Into Software Environments," University of Maryland, Technical Report TR-1764, June 1987

[6]Basili, V. R., and H. D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," *IEEE Transactions on Software Engineering*, June 1988

[7]Basili, V. R., and H. D. Rombach, *Towards A Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment*, University of Maryland, Technical Report TR-2158, December 1988

[8]Basili, V. R., and H. D. Rombach, *Towards A Comprehensive Framework for Reuse: Model-Based Reuse Characterization Schemes*, University of Maryland, Technical Report TR-2446, April 1990

[9]Basili, V. R., and H. D. Rombach, "Support for Comprehensive Reuse," *Software Engineering Journal*, September 1991

[3]Basili, V. R., and R. W. Selby, Jr., "Calculation and Use of an Environment's Characteristic Software Metric Set," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

Basili, V. R., and R. W. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, December 1987

[3]Basili, V. R., and R. W. Selby, Jr., "Four Applications of a Software Data Collection and Analysis Methodology," *Proceedings of the NATO Advanced Study Institute*, August 1985

[5]Basili, V. R., and R. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, December 1987

[9]Basili, V. R., and R. W. Selby, "Paradigms for Experimentation and Empirical Studies in Software Engineering," *Reliability Engineering and System Safety*, January 1991

[4]Basili, V. R., R. W. Selby, Jr., and D. H. Hutchens, "Experimentation in Software Engineering," *IEEE Transactions on Software Engineering*, July 1986

[2]Basili, V. R., R. W. Selby, and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," *IEEE Transactions on Software Engineering*, November 1983

[2]Basili, V. R., and D. M. Weiss, *A Methodology for Collecting Valid Software Engineering Data*, University of Maryland, Technical Report TR-1235, December 1982

[3]Basili, V. R., and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Transactions on Software Engineering*, November 1984

[1]Basili, V. R., and M. V. Zelkowitz, "The Software Engineering Laboratory: Objectives," *Proceedings of the Fifteenth Annual Conference on Computer Personnel Research*, August 1977

Basili, V. R., and M. V. Zelkowitz, "Designing a Software Measurement Experiment," *Proceedings of the Software Life Cycle Management Workshop*, September 1977

[1]Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," *Proceedings of the Second Software Life Cycle Management Workshop*, August 1978

[1]Basili, V. R., and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," *Computers and Structures*, August 1978, vol. 10

Basili, V. R., and M. V. Zelkowitz, "Analyzing Medium Scale Software Development," *Proceedings of the Third International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1978

[13]Basili, V. R., M. Zelkowitz, F. McGarry, G. Page, S. Waligora, and R. Pajerski, "SEL's Software Process-Improvement Program," *IEEE Software*, vol. 12, no. 6, November 1995, pp. 83–87

[14]Basili, V. R., S. Green, O. Laitenberger, F. Shull, S. Sorumgard, and M. Zelkowitz, "*The Empirical Investigation of Perspective-Based Reading,*" University of Maryland, Computer Science Technical Report, CS-TR-3585, UMIACS-TR-95-127, December 1995

[14]Basili, V. R., "Evolving and Packaging Reading Technologies," P*roceedings of the Third International Conference on Achieving Quality in Software, Florence, Italy*, January 1996

[14]Basili, V. R., "The Role of Experimentation in Software Engineering: Past, Current, and Future," *Proceedings of the Eighteenth Annual Conference on Software Engineering (ICSE-18)*, March 1996

[14]Basili, V. R., G. Calavaro, G. Iazeolla, "Simulation Modeling of Software Development Processes," *7th European Simulation Symposium (ESS '95),* October 1995

[15]Basili, V. R., L. C. Briand and W. L. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering,* October 1996

[15]Basili, V. R., S. E. Condon, K. El Emam, R. B. Hendrick and W. L. Melo, "Characterizing and Modeling the Cost of Rework in a Library of Reusable Software Components," International Conference on Software Engineering (ICSE-19), May 1997

[15]Basili, V. R., L. C. Briand, and W. L. Melo, "How Reuse Influences Productivity in Object-Oriented Systems," *Communications of the ACM,* October 1996

[12]Bassman, M. J., F. McGarry, and R. Pajerski, *Software Measurement Guidebook*, NASA-GB-001-94, Software Engineering Program, July 1994

[9]Booth, E. W., and M. E. Stark, "Designing Configurable Software: COMPASS Implementation Concepts," *Proceedings of Tri-Ada 1991*, October 1991

[10]Booth, E. W., and M. E. Stark, "Software Engineering Laboratory Ada Performance Study—Results and Implications," *Proceedings of the Fourth Annual NASA Ada User's Symposium*, April 1992

[10]Briand, L. C., and V. R. Basili, "A Classification Procedure for the Effective Management of Changes During the Maintenance Process," *Proceedings of the 1992 IEEE Conference on Software Maintenance (CSM 92)*, November 1992

[10]Briand, L. C., V. R. Basili, and C. J. Hetmanski, "Providing an Empirical Basis for Optimizing the Verification and Testing Phases of Software Development," *Proceedings of the Third IEEE International Symposium on Software Reliability Engineering (ISSRE 92)*, October 1992

[11]Briand, L. C., V. R. Basili, and C. J. Hetmanski, *Developing Interpretable Models with Optimized Set Reduction for Identifying High Risk Software Components*, University of Maryland, Technical Report TR-3048, March 1993

[12]Briand, L. C., V. R. Basili, Y. Kim, and D. R. Squier, "A Change Analysis Process to Characterize Software Maintenance Projects," *Proceedings of the International Conference on Software Maintenance*, Victoria, British Columbia, Canada, September 19–23, 1994, pp. 38–49

[9]Briand, L. C., V. R. Basili, and W. M. Thomas, *A Pattern Recognition Approach for Software Engineering Data Analysis*, University of Maryland, Technical Report TR-2672, May 1991

[13]Briand, L., W. Melo, C. Seaman, and V. R. Basili, "Characterizing and Assessing a Large-Scale Software Maintenance Organization," *Proceedings of the 17th International Conference on Software Engineering*, Seattle, Washington, U.S.A., April 23–30, 1995

[11]Briand, L. C., S. Morasca, and V. R. Basili, "Measuring and Assessing Maintainability at the End of High Level Design," *Proceedings of the 1993 IEEE Conference on Software Maintenance (CSM 93)*, November 1993

[12]Briand, L., S. Morasca, and V. R. Basili, *Defining and Validating High-Level Design Metrics*, University of Maryland, Computer Science Technical Report, CS-TR-3301, UMIACS-TR-94-75, June 1994

[13]Briand, L., S. Morasca, and V. R. Basili, *Property-based Software Engineering Measurement*, University of Maryland, Computer Science Technical Report, CS-TR-3368, UMIACS-TR-94-119, November 1994

[13]Briand, L., S. Morasca, and V. R. Basili, *Goal-Driven Definition of Product Metrics Based on Properties*, University of Maryland, Computer Science Technical Report, CS-TR-3346, UMIACS-TR-94-106, December 1994

[11]Briand, L. C., W. M. Thomas, and C. J. Hetmanski, "Modeling and Managing Risk Early in Software Development," *Proceedings of the Fifteenth International Conference on Software Engineering (ICSE 93)*, May 1993

[14]Briand, L., Y. Kim, W. Melo, C. B. Seaman, V. R. Basili, "Qualitative Analysis for Maintenance Process Assessment," University of Maryland, Computer Science Technical Report, CS-TR-3592, UMIACS-TR-96-7, January 1996

[14]Briand, L., V. R. Basili, S. Condon, Y. Kim, W. Melo and J. D. Valett, "Understanding and Predicting the Process of Software Maintenance Releases," P*roceedings of the Eighteenth Annual Conference on Software Engineering (ICSE-18)*, March 1996

[15]Briand, L., S. Morasca, and V. R. Basili, "Property-Based Software Engineering Measurement," *IEEE Transactions on Software Engineering*, January 1996

[15]Briand, L., S. Morasca, and V. R. Basili, Response to: Comments on "Property-Based Software Engineering Measurement: Refining the Additivity Properties", *IEEE Transactions on Software Engineering*, March 1997

[5]Brophy, C. E., W. W. Agresti, and V. R. Basili, "Lessons Learned in Use of Ada-Oriented Design Methods," *Proceedings of the Joint Ada Conference*, March 1987

[6]Brophy, C. E., S. Godfrey, W. W. Agresti, and V. R. Basili, "Lessons Learned in the Implementation Phase of a Large Ada Project," *Proceedings of the Washington Ada Technical Conference*, March 1988

[2]Card, D. N., "Early Estimation of Resource Expenditures and Program Size," Computer Sciences Corporation, Technical Memorandum, June 1982

[2]Card, D. N., "Comparison of Regression Modeling Techniques for Resource Estimation," Computer Sciences Corporation, Technical Memorandum, November 1982

[3]Card, D. N., "A Software Technology Evaluation Program," *Annais do XVIII Congresso Nacional de Informatica*, October 1985

[5]Card, D. N., and W. W. Agresti, "Resolving the Software Science Anomaly," *Journal of Systems and Software*, 1987

[6]Card, D. N., and W. W. Agresti, "Measuring Software Design Complexity," *Journal of Systems and Software*, June 1988

[4]Card, D. N., V. E. Church, and W. W. Agresti, "An Empirical Study of Software Design Practices," *IEEE Transactions on Software Engineering*, February 1986

Card, D. N., V. E. Church, W. W. Agresti, and Q. L. Jordan, "A Software Engineering View of Flight Dynamics Analysis System," Parts I and II, Computer Sciences Corporation, Technical Memorandum, February 1984

Card, D. N., Q. L. Jordan, and V. E. Church, "Characteristics of FORTRAN Modules," Computer Sciences Corporation, Technical Memorandum, June 1984

[5]Card, D. N., F. E. McGarry, and G. T. Page, "Evaluating Software Engineering Technologies," *IEEE Transactions on Software Engineering*, July 1987

[3]Card, D. N., G. T. Page, and F. E. McGarry, "Criteria for Software Modularization," *Proceedings of the Eighth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1985

[1]Chen, E., and M. V. Zelkowitz, "Use of Cluster Analysis To Evaluate Software Engineering Methodologies," *Proceedings of the Fifth International Conference on Software Engineering*. New York: IEEE Computer Society Press, 1981

[4]Church, V. E., D. N. Card, W. W. Agresti, and Q. L. Jordan, "An Approach for Assessing Software Prototypes," *ACM Software Engineering Notes*, July 1986

[15]Condon, S., R. Hendrick, M. E. Stark, W. Steger, "The Generalized Support Software (GSS) Domain Engineering Process: An Object-Oriented Implementation and Reuse Success at Goddard Space Flight Center," *Addendum to the Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 96)*, San Jose, California, U.S.A., October 1996

[15]Devanbu, P., S. Karstu, W. L. Melo and W. Thomas, "Analytical and Empirical Evaluation of Software Reuse Metrics," *Proceedings of the 18th International Conference on Software Engineering (ICSE-18),* March 1996

[2]Doerflinger, C. W., and V. R. Basili, "Monitoring Software Development Through Dynamic Variables," *Proceedings of the Seventh International Computer Software and Applications Conference*. New York: IEEE Computer Society Press, 1983

Doubleday, D., *ASAP: An Ada Static Source Code Analyzer Program*, University of Maryland, Technical Report TR-1895, August 1987 (NOTE: 100 pages long)

[6]Godfrey, S., and C. Brophy, "Experiences in the Implementation of a Large Ada Project," *Proceedings of the 1988 Washington Ada Symposium*, June 1988

[5]Jeffery, D. R., and V. R. Basili, *Characterizing Resource Data: A Model for Logical Association of Software Data*, University of Maryland, Technical Report TR-1848, May 1987

[6]Jeffery, D. R., and V. R. Basili, "Validating the TAME Resource Data Model," *Proceedings of the Tenth International Conference on Software Engineering*, April 1988

[15]Kontio, J., G. Caldiera and V. R. Basili, "Defining Factors, Goals and Criteria for Reusable Component Evaluation," *CASCON '96 Conference*, November 1996

[15]Kontio, J., H. Englund and V. R. Basili, *Experiences from an Exploratory Case Study with a Software Risk Management Method,* Computer Science Technical Report, CS-TR-3705, UMIACS-TR-96-75, August 1996

[15]Lanubile, F., "Why Software Reliability Predictions Fail," *IEEE Software*, July 1996

[11]Li, N. R., and M. V. Zelkowitz, "An Information Model for Use in Software Management Estimation and Prediction," *Proceedings of the Second International Conference on Information Knowledge Management*, November 1993

[5]Mark, L., and H. D. Rombach, *A Meta Information Base for Software Engineering*, University of Maryland, Technical Report TR-1765, July 1987

[6]Mark, L., and H. D. Rombach, "Generating Customized Software Engineering Information Bases From Software Process and Product Specifications," *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, January 1989

[5]McGarry, F. E., and W. W. Agresti, "Measuring Ada for Software Development in the Software Engineering Laboratory (SEL)," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988

[7]McGarry, F., L. Esker, and K. Quimby, "Evolution of Ada Technology in a Production Software Environment," *Proceedings of the Sixth Washington Ada Symposium (WADAS)*, June 1989

[13]McGarry, F., R. Pajerski, G. Page, et al., *Software Process Improvement in the NASA Software Engineering Laboratory,* Carnegie-Mellon University, Software Engineering Institute, Technical Report CMU/SEI-94-TR-22, ESC-TR-94-022, December 1994

[3]McGarry, F. E., J. Valett, and D. Hall, "Measuring the Impact of Computer Resource Quality on the Software Development Process and Product," *Proceedings of the Hawaiian International Conference on System Sciences*, January 1985

[15]Morasca, S., L. C. Briand, V. R. Basili, E. J. Weyuker and M. V. Zelkowitz, Comments on "Towards a Framework for Software Measurement Validation," *IEEE Transactions on Software Engineering*, March 1997

[3]Page, G., F. E. McGarry, and D. N. Card, "A Practical Experience With Independent Verification and Validation," *Proceedings of the Eighth International Computer Software and Applications Conference*, November 1984

[12]Porter, A. A., L. G. Votta, Jr., and V. R. Basili, *Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment*, University of Maryland, Technical Report TR-3327, July 1994

[5]Ramsey, C. L., and V. R. Basili, "An Evaluation of Expert Systems for Software Engineering Management," *IEEE Transactions on Software Engineering*, June 1989

[3]Ramsey, J., and V. R. Basili, "Analyzing the Test Process Using Structural Coverage," *Proceedings of the Eighth International Conference on Software Engineering.* New York: IEEE Computer Society Press, 1985

[5]Rombach, H. D., "A Controlled Experiment on the Impact of Software Structure on Maintainability," *IEEE Transactions on Software Engineering*, March 1987

[8]Rombach, H. D., "Design Measurement: Some Lessons Learned," *IEEE Software*, March 1990

[9]Rombach, H. D., "Software Reuse: A Key to the Maintenance Problem," *Butterworth Journal of Information and Software Technology*, January/February 1991

[6]Rombach, H. D., and V. R. Basili, "Quantitative Assessment of Maintenance: An Industrial Case Study," *Proceedings From the Conference on Software Maintenance*, September 1987

[6]Rombach, H. D., and L. Mark, "Software Process and Product Specifications: A Basis for Generating Customized SE Information Bases," *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, January 1989

[7]Rombach, H. D., and B. T. Ulery, *Establishing a Measurement Based Maintenance Improvement Program: Lessons Learned in the SEL*, University of Maryland, Technical Report TR-2252, May 1989

[10]Rombach, H. D., B. T. Ulery, and J. D. Valett, "Toward Full Life Cycle Control: Adding Maintenance Measurement to the SEL," *Journal of Systems and Software*, May 1992

[14]Seaman, C. B., V. R. Basili, "*Communication and Organization in Software Development: An Empirical Study,*" University of Maryland, Computer Science Technical Report, CS-TR-3619, UMIACS-TR-96-23, April 1996

[15]Seaman, C. B, V. R. Basili, "An Empirical Study of Communication in Code Inspection," *Proceedings of 19th International Conference on Software Engineering (ICSE-19),* May 1997, pp. 96-106

[6]Seidewitz, E., “Object-Oriented Programming in Smalltalk and Ada,” *Proceedings of the 1987 Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1987

[5]Seidewitz, E., “General Object-Oriented Software Development: Background and Experience,” *Proceedings of the 21st Hawaii International Conference on System Sciences*, January 1988

[6]Seidewitz, E., “General Object-Oriented Software Development with Ada: A Life Cycle Approach,” *Proceedings of the CASE Technology Conference*, April 1988

[9]Seidewitz, E., “Object-Oriented Programming Through Type Extension in Ada 9X,” *Ada Letters*, March/April 1991

[10]Seidewitz, E., “Object-Oriented Programming With Mixins in Ada,” *Ada Letters*, March/April 1992

[12]Seidewitz, E., “Genericity versus Inheritance Reconsidered: Self-Reference Using Generics,” *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, October 1994

[4]Seidewitz, E., and M. Stark, “Towards a General Object-Oriented Software Development Methodology,” *Proceedings of the First International Symposium on Ada for the NASA Space Station*, June 1986

[9]Seidewitz, E., and M. Stark, “An Object-Oriented Approach to Parameterized Software in Ada,” *Proceedings of the Eighth Washington Ada Symposium*, June 1991

[8]Stark, M., “On Designing Parametrized Systems Using Ada,” *Proceedings of the Seventh Washington Ada Symposium*, June 1990

[11]Stark, M., “Impacts of Object-Oriented Technologies: Seven Years of SEL Studies,” *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, September 1993*

[7]Stark, M. E. and E. W. Booth, “Using Ada to Maximize Verbatim Software Reuse,” *Proceedings of TRI-Ada 1989*, October 1989

[5]Stark, M., and E. Seidewitz, “Towards a General Object-Oriented Ada Lifecycle,” *Proceedings of the Joint Ada Conference*, March 1987

[13]Stark, M., and E. Seidewitz, "Generalized Support Software: Domain Analysis and Implementation," *Addendum to the Proceedings OOPSLA '94*, Ninth Annual Conference, Portland, Oregon, U.S.A., October 1994, pp. 8–13

[15]Stark, M., “Using Applet Magic (tm) to Implement an Orbit Propagator: New Life for Ada Objects,” *Proceedings of the 14th Annual Washington Ada Symposium (WAdaS97),* June 1997

[10]Straub, P. A., and M. V. Zelkowitz, "On the Nature of Bias and Defects in the Software Specification Process," *Proceedings of the Sixteenth International Computer Software and Applications Conference (COMPSAC 92)*, September 1992

[8]Straub, P. A., and M. V. Zelkowitz, "PUC: A Functional Specification Language for Ada," *Proceedings of the Tenth International Conference of the Chilean Computer Science Society*, July 1990

[7]Sunazuka, T., and V. R. Basili, *Integrating Automated Support for a Software Management Cycle Into the TAME System*, University of Maryland, Technical Report TR-2289, July 1989

[13]Thomas, W. M., A. Delis, and V. R. Basili, *An Analysis of Errors in a Reuse-Oriented Development Environment*, University of Maryland, Computer Science Technical Report, CS-TR-3424, UMIACS-TR-95-24, February 1995

[10]Tian, J., A. Porter, and M. V. Zelkowitz, "An Improved Classification Tree Analysis of High Cost Modules Based Upon an Axiomatic Definition of Complexity," *Proceedings of the Third IEEE International Symposium on Software Reliability Engineering (ISSRE 92)*, October 1992

Turner, C., and G. Caron, *A Comparison of RADC and NASA/SEL Software Development Data*, Data and Analysis Center for Software, Special Publication, May 1981

[10]Valett, J. D., "Automated Support for Experience-Based Software Management," *Proceedings of the Second Irvine Software Symposium (ISS _92)*, March 1992

[5]Valett, J. D., and F. E. McGarry, "A Summary of Software Measurement Experiences in the Software Engineering Laboratory," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988

[14]Waligora, S., J. Bailey, and Mike Stark, "The Impact of Ada and Object-Oriented Design in NASA Goddard's Flight Dynamics Division," July 1996

[3]Weiss, D. M., and V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data From the Software Engineering Laboratory," *IEEE Transactions on Software Engineering*, February 1985

[5]Wu, L., V. R. Basili, and K. Reed, "A Structure Coverage Tool for Ada Software Systems," *Proceedings of the Joint Ada Conference*, March 1987

[1]Zelkowitz, M. V., "Resource Estimation for Medium-Scale Software Projects," *Proceedings of the Twelfth Conference on the Interface of Statistics and Computer Science.* New York: IEEE Computer Society Press, 1979

[2]Zelkowitz, M. V., "Data Collection and Evaluation for Experimental Computer Science Research," *Empirical Foundations for Computer and Information Science* (Proceedings), November 1982

[6]Zelkowitz, M. V., "The Effectiveness of Software Prototyping: A Case Study," *Proceedings of the 26th Annual Technical Symposium of the Washington, D.C., Chapter of the ACM*, June 1987

[6]Zelkowitz, M. V., “Resource Utilization During Software Development,” *Journal of Systems and Software*, 1988

[8]Zelkowitz, M. V., “Evolution Towards Specifications Environment: Experiences With Syntax Editors,” *Information and Software Technology*, April 1990

[14]Zelkowitz, M. V., "Software Engineering Technology Infusion Within NASA," *IEEE Transactions On Engineering Management*, vol. 43, no. 3, August 1996

NOTES:

[1]This article also appears in SEL-82-004, *Collected Software Engineering Papers: Volume I*, July 1982.

[2]This article also appears in SEL-83-003, *Collected Software Engineering Papers: Volume II*, November 1983.

[3]This article also appears in SEL-85-003, *Collected Software Engineering Papers: Volume III*, November 1985.

[4]This article also appears in SEL-86-004, *Collected Software Engineering Papers: Volume IV*, November 1986.

[5]This article also appears in SEL-87-009, *Collected Software Engineering Papers: Volume V*, November 1987.

[6]This article also appears in SEL-88-002, *Collected Software Engineering Papers: Volume VI*, November 1988.

[7]This article also appears in SEL-89-006, *Collected Software Engineering Papers: Volume VII*, November 1989.

[8]This article also appears in SEL-90-005, *Collected Software Engineering Papers: Volume VIII*, November 1990.

[9]This article also appears in SEL-91-005, *Collected Software Engineering Papers: Volume IX*, November 1991.

[10]This article also appears in SEL-92-003, *Collected Software Engineering Papers: Volume X*, November 1992.

[11]This article also appears in SEL-93-001, *Collected Software Engineering Papers: Volume XI*, November 1993.

[12]This article also appears in SEL-94-004, *Collected Software Engineering Papers: Volume XII*, November 1994.

[13]This article also appears in SEL-95-003, *Collected Software Engineering Papers: Volume XIII*, November 1995.

[14]This article also appears in SEL-96-001, *Collected Software Engineering Papers: Volume XIV*, October 1996.

[15]This article also appears in SEL-97-002, *Collected Software Engineering Papers: Volume XV*, October 1997.

REPORT DOCUMENTATION PAGE			*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** October 1997	**3. REPORT TYPE AND DATES COVERED** Technical Memorandum

4. TITLE AND SUBTITLE Software Engineering Laboratory Series Collected Software Engineering Papers: Volume XV	**5. FUNDING NUMBERS** Code 551
6. AUTHOR(S) Flight Dynamics Systems Branch	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS (ES) Goddard Space Flight Center Greenbelt, Maryland 20771	**8. PEFORMING ORGANIZATION REPORT NUMBER** SEL-97-002
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS (ES) National Aeronautics and Space Administration Washington, DC 20546-0001	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER** TM—1998–208614
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION / AVAILABILITY STATEMENT Unclassified–Unlimited Subject Category: 82 Report available from the NASA Center for AeroSpace Information, 7121 Standard Drive, Hanover, MD 21076-1320. (301) 621-0390.	**12b. DISTRIBUTION CODE**

13. ABSTRACT *(Maximum 200 words)*

The Software Engineering Laboratory (SEL) is an organization sponsored by NASA/GSFC and created to investigate the effectiveness of software engineering technologies when applied to the development of application software.

The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document.

14. SUBJECT TERMS Software Engineering Laboratory, Application software, Documentation	**15. NUMBER OF PAGES** 214
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL