Assessing Software Quality in Ada Based Products with
the Objectives, Principles, Attributes Framework

by

Gary Neal Bundy

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirement for the degree of
Master of Science
in
Computer Science

APPROVED:

[Signatures]

Dr. James D. Arthur, Chairman

Dr. Richard E. Nance
Dr. Sallie M. Henry

September, 1990
Blacksburg, Virginia
Assessing Software Quality in Ada Based Products with the Objectives, Principles, Attributes Framework

by

Gary Neal Bundy

Dr. James D. Arthur, Chairman

Computer Science

(ABSTRACT)

This thesis describes the results of a research effort focusing on the validation of a procedure for assessing the quality of an Ada-based product. Starting with the identification of crucial Ada constructs, this thesis outlines a seven-step process for defining metrics that support software quality assessment within a framework based on linkages among software engineering objectives, principles, and attributes. The thesis presents the impact of the use of crucial Ada constructs on the software engineering attributes and describes measurement approaches for assessing that impact. This thesis also outlines a planned research effort to develop an automated analyzer for the assessment of software quality in Ada-based products and plans for validating the assessment procedure.

* This work has been partially supported by the Joint Logistics Commanders through the Systems Research Center under contract number N60921-89-D-A239 B004.
Acknowledgements

I would like to thank my advisor, Dr. James Arthur, for his guidance and direction during my research and the actual writing of this thesis. I would also like to thank the other two members of my committee, Dr. Richard Nance and Dr. Sallie Henry. Dr. Nance provided intellectual stimulation and interest in my research when I needed it the most and Dr. Henry gave me advice and encouragement during the early stages of my research and graduate education.

During the final eight months of my research, the Joint Logistics Commanders through the Systems Research Center provided financial support. Their support enabled me to concentrate on the research effort.

Finally, one other person deserves a special thanks. Sandra Griffith provided editorial assistance and friendship during the final stages of writing this thesis.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction and Background</td>
<td>1</td>
</tr>
<tr>
<td>Problem Statement</td>
<td>3</td>
</tr>
<tr>
<td>Solution Approach</td>
<td>4</td>
</tr>
<tr>
<td>Novelty of Problem Solution</td>
<td>5</td>
</tr>
<tr>
<td>Summary of Previously Developed Code Metrics</td>
<td>6</td>
</tr>
<tr>
<td>Halstead's Software Science</td>
<td>7</td>
</tr>
<tr>
<td>McCabe's Cyclomatic Number</td>
<td>9</td>
</tr>
<tr>
<td>Henry and Kafura's Information Flow</td>
<td>10</td>
</tr>
<tr>
<td>The OPA Framework</td>
<td>12</td>
</tr>
<tr>
<td>The Software Metric Development Procedure</td>
<td>17</td>
</tr>
<tr>
<td>Classifying Programming Language Components by Functionality</td>
<td>22</td>
</tr>
<tr>
<td>The Relationship Between Pascal &amp; Ada</td>
<td>24</td>
</tr>
<tr>
<td>Issues That Complicate Metric Development</td>
<td>25</td>
</tr>
<tr>
<td>Lines of Code</td>
<td>25</td>
</tr>
<tr>
<td>Overloading</td>
<td>26</td>
</tr>
<tr>
<td>Functionality</td>
<td>28</td>
</tr>
<tr>
<td>Concurrency</td>
<td>28</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Software Complexity</td>
<td>29</td>
</tr>
<tr>
<td>Assessing the Impact of Crucial Ada Components on the Software Engineering Attributes</td>
<td>30</td>
</tr>
<tr>
<td>Data Types</td>
<td>31</td>
</tr>
<tr>
<td>Built-In Data Types</td>
<td>31</td>
</tr>
<tr>
<td>Data Aggregates</td>
<td>33</td>
</tr>
<tr>
<td>User Defined Types</td>
<td>33</td>
</tr>
<tr>
<td>Abstract Data Types</td>
<td>35</td>
</tr>
<tr>
<td>Statement Level Control Structures</td>
<td>37</td>
</tr>
<tr>
<td>Sequencing Structures</td>
<td>39</td>
</tr>
<tr>
<td>Selection Structures</td>
<td>39</td>
</tr>
<tr>
<td>Looping Constructs</td>
<td>39</td>
</tr>
<tr>
<td>Block Structures</td>
<td>41</td>
</tr>
<tr>
<td>Unit Level Control Structures</td>
<td>44</td>
</tr>
<tr>
<td>Subprograms</td>
<td>45</td>
</tr>
<tr>
<td>Parameters</td>
<td>47</td>
</tr>
<tr>
<td>Exception Handlers</td>
<td>51</td>
</tr>
<tr>
<td>Concurrent Routines: Tasking</td>
<td>52</td>
</tr>
<tr>
<td>Definitional Units</td>
<td>54</td>
</tr>
<tr>
<td>Packages</td>
<td>55</td>
</tr>
<tr>
<td>Generics</td>
<td>58</td>
</tr>
<tr>
<td>Summary</td>
<td>60</td>
</tr>
</tbody>
</table>

Developing Indicators, Measures, and Metrics for Crucial Ada Components  | 61   |
Data Types: Discriminants                                              | 62   |
Statement Level Control Structures: Block Statement                     | 64   |
Examples of Methods of Using Ada Exception Handlers .............. 107

Property/Attribute Pairs for Identified Indicators ................. 110

Measures and Metrics for Ada Specific Indicators ............... 135

Metrics for Ada Constructs by Software Engineering Attribute .... 175

Vita .............................................................................. 195
List of Illustrations

Figure 1. Objectives, Principles, Attributes Evaluation Procedure ............... 13
Figure 2. Strings declaration examples from Gilpin .............................. 34
Figure 3. Discriminant examples .................................................. 36
Figure 4. Array slicing examples from Booch .................................... 40
Figure 5. Named loop and loop exit examples from Gilpin ....................... 42
Figure 6. Block and overloading example from Gilpin ............................ 43
Figure 7. Default parameter examples ............................................ 48
Figure 8. Name and positional parameter examples ................................ 49
Figure 9. Property/Attribute Indicators for Ada Parameters ..................... 67
Figure 10. Property/Attribute Indicators for Ada Packages ....................... 69
Figure 11. Property/Attribute Indicators for Ada Generics ....................... 73
Figure 12. Property/Attribute Indicators for Ada Tasking Constructs .......... 76
Figure 13. Number of OPA Framework Metrics by Attribute .................... 83
Figure 14. The Automatic Analyzer .............................................. 85
List of Tables

Table 1. Software Engineering Objectives ......................................... 14
Table 2. Software Engineering Principles ........................................... 15
Table 3. Software Engineering Attributes .......................................... 16
Table 4. Analysis of Ada and Pascal Data Types ............................... 32
Table 5. Analysis of Ada and Pascal Statement-Level Control Structures ... 38
Table 6. Analysis of Ada and Pascal Unit-Level Control Structures ........ 46
Chapter 1

Introduction and Background

Assessing the quality of software has been a continuous problem in computer science and software engineering. As program size and complexity have grown, so has the task of assessing software quality. Manual assessments can be completed in a timely and accurate manner for small programs. However, as program size grows, this task becomes prohibitively large.

Another problem with attempting to assess software quality is deciding upon which metric or metrics to use. A literature survey of metrics reveals that many metrics are available for measuring software. One of the oldest metrics available is lines of code. Although this measure is extremely intuitive, it is also riddled with problems. One of the most prevalent problems is the extreme lack of standardization of
measurement approaches [FIRED88]. Other metrics in use today include Halstead's Software Science [HALSM77], McCabe's Cyclomatic Number [MCCAT76], and Henry and Kafura's Information Flow [HENRS81]. Each of these metrics is discussed in greater detail later in this chapter.

A major criticism of many of these metrics is the lack of a "clear specification of what is being measured" [KEARJ86, p. 1050]. Another author notes that a desirable attribute, missing from most metrics, is that software metrics should "empirically and intuitively describe software behavior" [EJIOC87, p. 61].

A contrasting approach to code metrics is the use of a quality assessment framework. An early framework, developed by McCall [MCCAJ77] and extended under the direction of the Rome Air Development Center [RADC89] relates quality factors and software engineering principles. Unfortunately, some quality factors such as safety and efficiency are not objectives of software engineering. In fact, the achievement of efficiency is usually at the detriment of objectives such as maintainability and reliability. In addition, McCall's framework does not address the concerns raised by Kearney and Ejiogu because of the weak link between software engineering principles and software products.

In an attempt to address the concerns raised by Kearney and Ejiogu, Arthur and Nance have developed the Objectives, Principles, Attributes (OPA) framework which indicates precisely what is being measured. Additionally, the OPA framework defines a set of linkages which relate the achievement of software engineering objectives to the use of principles, and the use of principles to the presence of certain code attributes. Code properties are then used to measure the extent to which attributes are either present or absent in the code. The code
properties provide simple, intuitive measures for assessing the impact of a property on an attribute. Each property and affected attribute are linked to form property/attribute pairs. Because of the confirming and contrasting nature of the attribute/property pairs, a code property can have a positive effect on one attribute, and a negative effect on another [ARTHJ87]. Effectively, the attribute/property pairs, in tandem with the linkages among software engineering objectives, principles, and attributes, provide a sound basis for assessing the achievement of desirable software engineering objectives.

The OPA framework as well as how it is used as a foundation for developing a well-defined, objective approach for assessing and predicting software quality will be discussed in greater detail in a subsequent section of this chapter. Following the discussion of the OPA framework, a seven step metric development procedure used to analyze Ada and develop metrics relative to the OPA framework is discussed. To support the metric development procedure, two schemes for identifying, categorizing, and classifying Ada components are discussed. Finally, several complicating issues that developed during the course of this research are considered.

**Problem Statement**

Currently, attribute/property pairs have been developed for a basic block-structured language such as Pascal [FARNM87]. The programming language Ada (ANSI/MIL-STD-1815 A) is a derivation of Pascal, but provides significant extensions in support of separate compilation, information hiding, exception handling, and concurrency [BOOCG83]. To date, however, the aforementioned extensions have not been considered within the OPA framework. Because of Ada's
widespread use in mission critical Department of Defense applications, and because Ada is said to promote "better" software engineered products through its language facilities, examining Ada relative to the OPA framework provides a basis for assessing the quality of mission critical software and empirically evaluating Ada products with respect to software engineering. The focus of this research is to examine Ada relative to the OPA framework. This problem decomposes into three major tasks:

- the classification and categorization of Ada constructs,
- the identification of new attribute/property pairs, and
- the development of metrics for identified attribute/property pairs.

This thesis describes the research initiative that has evolved to solve this problem.

Solution Approach

The solution approach for examining Ada relative to the OPA framework extends the research efforts of Dandekar [DANDA87] and Faman [FARNM87]. Among other results, these authors provide property/attribute pairs (indicators) for the conventional language constructs found in a language such as Pascal. Recognizing this result leads to the solution approach proposed in this research effort.

The first step in the solution approach is to identify each of the new components (or extensions) in Ada that is not found in Pascal. Secondly, each of these new components is carefully studied to determine how it will actually be (or is being) used. This results in a list of uses which can have beneficial and detrimental impacts on the software quality. These beneficial and detrimental uses are referred
to as uses and abuses. The uses and abuses are used to identify code properties indicative of each use and abuse. Each of these code properties is characterized as having either a positive or negative effect on one or more of the attributes. The resulting attribute/property pairs are termed "code indicators."

Once these pairs are identified, the final step of developing measures and metrics for each indicator can be completed. One application of the research results is an automatic analyzer that uses the metrics previously developed for a language similar to Pascal [DANDA87, FARNM87] and the new metrics developed specifically for Ada. The automated set of metrics, embodied within the OPA framework, provides an assessment tool that can determine the extent to which (a) software engineering objectives are achieved, (b) principles are used, and (c) desirable attributes are present or absent in the Ada-based products.

**Novelty of Problem Solution**

Over the past decade several systems of metrics have been developed in an attempt to capture various measures of software products. More recently a set of complexity metrics was extended to include Ada [CHAPB89, CHAPB90]. Using the OPA framework to assess the quality of Ada-based products provides a novel approach that is yet to be utilized. The OPA framework integrates well with the assessment of Ada-based products because of the design rationale for Ada. The OPA framework is based upon the software engineering objectives, principles, and attributes [ARTHJ87] while Ada was designed to support the utilization of software engineering principles [BOOOG83].
To facilitate relating the OPA framework to Ada the author has developed a seven step procedure for developing metrics. The results of applying this procedure include:

1. The categorization of Ada components relative to their presence or absence in Pascal;
2. the categorization of Ada components relative to functionality; a set of property/attribute pairs (code indicators) relating Ada code properties to the presence or absence of code attributes; and
3. a quantitative, objective set of metrics assessing the effect of a property on an attribute for each property/attribute pair defined.

Additionally, the research relating the OPA framework to Ada includes a review and modification (if appropriate) to the property/attribute pairs and metrics developed for conventional languages like Pascal. The application of the metric development procedure and the review of the original property/attribute pairs and metrics with respect to Ada and the Objectives, Principles, Attributes framework is the focus of this thesis.

Summary of Previously Developed Code Metrics

In current literature several proposed complexity metrics are given, and new metrics continue to appear. Three popular well documented metrics are: (1) Halstead's Software Science, (2) McCabe's Cyclomatic Number, and (3) Henry and Kafura's Information Flow [KEAR86]. Each is representative of a style or direction in the complexity metric field. Although other popular metrics are available, these three are sufficient for a discussion of the approaches to current complexity metrics. The...
three metrics are discussed as to what is being measured and what is not being measured.

**Halstead's Software Science**

Halstead's approach to software assessment is based on the tokens found in a program [HALSM77]. In his book, Halstead defines two types of tokens, operands and operators, and from them the basic variables of Halstead's Software Science are developed. The unit of measure for these variables is the bit. The reasoning is that given any language with a finite set of tokens, a finite number of bits can be used to represent each unique token. The basic variables of the vocabulary are:

- $n_1$: number of unique operators,
- $n_2$: number of unique operands,
- $N_1$: the total number of operators, and
- $N_2$: the total number of operands.

Halstead builds on these basic variables by defining the length of a program as the sum of $N_1$ and $N_2$. This value is intuitive since the length of a program would be the sum of the total number of operators and the total number of operands. In order to define a measure for volume, the size of the program independent of the character set used to express the program, Halstead notes that the vocabulary of a language is the set of unique operators and unique operands. The size of this vocabulary is the sum of $n_1$ and $n_2$. In general, the minimum length of bits needed to express each element of the vocabulary is $\log_2 (n_1 + n_2)$. Utilizing the program
length and minimum number of bits needed to express elements of the language, program volume - $V$, can be defined as

- $V = (N_1 + N_2) \log_2(n_1 + n_2)$.

The calculation for program volume is objective, succinct, and justifiable. However, as Halstead continues to define the elements of his software science, the calculations become based on estimations, not absolutes. For example, the next element defined is program level. Program level refers to the level of the language in which the program is written, e.g., a higher-level language as opposed to assembly language. According to Halstead, the program level is a function of the program volume and program potential volume. He estimates the program level - $L$, as:

- $L = (2 \cdot n_2) / (n_1 \cdot N_2)$.

Halstead then defines the program difficulty as the inverse of program level ($D = 1/L$). The rationale for this calculation is that as the program level increases, the difficulty of writing a program decreases. From program difficulty and program volume, Halstead defines the effort - $E$, associated with a program to be equal to the difficulty of the program multiplied by the volume of the program. This calculation is more succinctly expressed by the formula

- $E = D \times V = \frac{(n_1 \cdot N_2(N_1 + N_2) \log_2(n_1 + n_2))}{2 \cdot n_2}$.

Halstead states intuitive arguments as to why variables such as unique operators, operands, and so forth have an effect on software complexity. In fact, most of the
formula that Halstead proposes are intuitive. What is not clear is the relationship between Halstead's program effort and the software engineering objectives, for example maintainability and reliability. Moreover, if one considers a language that has a package feature, e.g. Ada, capturing the relationship between added volume and the potential for significant complexity reduction is not clear.

McCabe's Cyclomatic Number

McCabe approaches measuring complexity in a manner different from Halstead [MCCAT76]. McCabe claims to have developed a metric which directly correlates with the program's testability and maintainability. McCabe notes that a 50 line program containing 25 consecutive "IF THEN" constructs is highly untestable due to the 33.5 million possible distinct control paths. In fact, McCabe contends that the major factor in determining program complexity is control paths in the program. To express this relationship McCabe utilizes graph theory to represent control structures in a program. Each vertex in the graph represents a control structure and arcs represent the paths between the control structure. The three variables used in McCabe's Cyclomatic Number are:

- e - number of edges in the control structure graph,
- n - number of vertices in a control structure graph, and
- p - number of connected components in the control structure graph.

The cyclomatic number is given by:

- \( v(G) = e - n + p \).
Noting special cases and conditions McCabe is able to simplify the calculation of the cyclomatic number. With these simplifications the number can be used to note modules that potentially could benefit from further decomposition. One problem with using McCabe's Cyclomatic Number, and also Halstead's effort metric, is noted by Kearney [KEARNJ86]. Kearney states that McCabe's and Halstead's metrics are highly correlated, but may give improved measures for lesser quality programs [KEARNJ86, p. 1047]. Another problem is that the number of functional statements does not affect McCabe's Cyclomatic Number.

Some language constructs are included in the Ada language for improving testability and maintainability. For example, generics aid both testability and maintainability by providing a parameterized template for defining multiple objects [BOOOCG83]. Utilization of generics requires the testing and maintenance of one algorithm instead of multiple algorithms. The improvement is not reflected by McCabe's Cyclomatic Number.

Henry and Kafura's Information Flow

Henry and Kafura focus on another aspect of the program complexity in developing their information flow metric [HENRS81]. The information flow metric concentrates on the interconnectivity of program components. Existing metrics which concentrate on analyzing lexical tokens (Halstead and McCabe are two of the authors they note) do not capture this measure for program complexity according to Henry and Kafura [pp. 510-511]. They also note that a system structure with an overly complex information flow among components presents large maintenance problems.
One important characteristic of Henry and Kafura's information flow metric is a description of it in terms of global flows of information among modules as well as indirect and direct local information flows among modules. Also, fan-in, the number of distinct flows into a procedure and fan-out, the number of distinct flows out of a procedure are derived in terms of local information flow and data structure accesses. The formula for procedural complexity is:

\[ \text{length} \times (\text{fan-in} \times \text{fan-out})^2. \]

The \((\text{fan-in} \times \text{fan-out})\) term represents the combinations of input-output destinations. Henry and Kafura weight this term by squaring it because they believe its effect on complexity is more than linear.

The design of the information flow metric is to encourage modularization and to simplify communication between procedures. The smallest (best) value for the information flow metric can be obtained through the anomaly condition of a program with no procedures resulting in no information flow among procedures [KEARJ86].

Henry and Kafura's Information Flow metric does expose procedures with a disproportionately large information flow [KEARJ86]. However, additional factors contribute to code complexity that Henry and Kafura do not consider. For instance, a module with a good complexity rating and an extremely poor structure would still receive a good information flow rating. In a later paper, Henry recommends using structure metrics to complement the information flow metric [HENRS90].
The OPA Framework

What is missing from the measurement approaches discussed so far is a method of assessing software quality based upon the many aspects of code structure and interconnectivity which can be related to the attainment of code objectives such as testability, maintainability, and reliability. The Objectives, Principles, Attributes (OPA) framework addresses the criticisms concerning complexity metrics stated in this thesis and also those enumerated by Kearney [KEANJ86]. The OPA framework was originally designed to evaluate software development methodologies and refined for the evaluation of associated products [ARTHJ87]. The approach of the OPA framework is based on sets of linkages among software engineering objectives, principles, and attributes. More specifically,

- The goals of a methodology or product are realized through the achievement of software engineering objectives,
- The objectives are achieved through the use of software engineering principles, and
- The use of certain principles induces desirable attributes in the software product.

For a list and definitions of the software engineering objectives, principles, and attributes as defined by Arthur and Nance [ARTHJ87], consult Table 1 on page 14, Table 2 on page 15, and Table 3 on page 16, respectively. For a graphical representation of the OPA framework [ARTHJ87], see Figure 1 on page 13.

Throughout the defined linkages among the objectives, principles, and attributes, the presence or absence of desirable code attributes is propagated to reflect the
Figure 1. Objectives, Principles, Attributes Evaluation Procedure

Introduction and Background
Table 1. Software Engineering Objectives

<table>
<thead>
<tr>
<th>Objective</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maintainability</td>
<td>The ease with which maintenance of a functional unit can be performed in accordance with prescribed requirements.</td>
</tr>
<tr>
<td>Correctness</td>
<td>The extent to which software is free from design defects and coding defects; that is, fault free.</td>
</tr>
<tr>
<td>Reusability</td>
<td>The extent to which a module can be used in multiple applications.</td>
</tr>
<tr>
<td>Testability</td>
<td>The extent to which software facilitates both the establishment of test criteria and the evaluation of the software with respect to those criteria.</td>
</tr>
<tr>
<td>Reliability</td>
<td>The extent to which software can be expected to perform its intended function with the required precision.</td>
</tr>
<tr>
<td>Portability</td>
<td>The ease with which software can be transferred from one computer system or environment to another.</td>
</tr>
<tr>
<td>Adaptability</td>
<td>The ease with which software allows differing system constraints and user needs to be satisfied.</td>
</tr>
<tr>
<td>Principle</td>
<td>Definition</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Hierarchical Decomposition</td>
<td>A method of designing a system by breaking it down into its components through a series of top-down refinements.</td>
</tr>
<tr>
<td>Functional Decomposition</td>
<td>A method of designing a system by breaking it down into its components along functional boundaries.</td>
</tr>
<tr>
<td>Information Hiding</td>
<td>The technique of encapsulating software design decisions in modules in such a way that the module interfaces reveal as little as possible about the module's inner workings.</td>
</tr>
<tr>
<td>Stepwise Refinement</td>
<td>A system development methodology in which data definitions and processing steps are defined broadly at first and then with increasing detail.</td>
</tr>
<tr>
<td>Structured Programming</td>
<td>A disciplined approach to software design which adheres to a specific set of rules based on principles such as top-down design, stepwise refinement, and data flow analysis.</td>
</tr>
<tr>
<td>Concurrent Documentation</td>
<td>The management of documents which may include the actions of synthesis and modifications to reflect current product status, document identification and acquisition, and document processing, storing, and dissemination.</td>
</tr>
<tr>
<td>Life Cycle Verification</td>
<td>Verifying requirements throughout the design, development, and maintenance phases of the life cycle.</td>
</tr>
</tbody>
</table>
Table 3. Software Engineering Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coupling</td>
<td>A measure of the interdependence among modules in a computer program.</td>
</tr>
<tr>
<td>Cohesion</td>
<td>The degree to which the tasks performed by a single program module are functionally related.</td>
</tr>
<tr>
<td>Complexity</td>
<td>The degree or complication of a system or system component, determined by such factors as the number and intricacy of interfaces, the number and intricacy of conditional branches, the level of nesting, the types of data structures, and other system characteristics.</td>
</tr>
<tr>
<td>Well-Defined Interface</td>
<td>The definitional clarity and completeness of a shared boundary between a pair of software components.</td>
</tr>
<tr>
<td>Readability</td>
<td>The difficulty in understanding a software component.</td>
</tr>
<tr>
<td>Ease of Change</td>
<td>The ease with which software accommodates enhancements and extensions.</td>
</tr>
<tr>
<td>Traceability</td>
<td>The ease in retracing the complete history of a software component from its current status to its design inception.</td>
</tr>
<tr>
<td>Visibility of Behavior</td>
<td>The provision of a review process for error checking.</td>
</tr>
<tr>
<td>Early Error Detection</td>
<td>An indication of fault in the requirement specifications and design prior to implementation.</td>
</tr>
</tbody>
</table>
achievement of software engineering objectives. Attributes, however, cannot be
directly measured. Code properties are used to obtain measures for code attributes.
For example, "Use of Parentheses around Conditions" is indicative of a positive
impact on the attribute of readability and "Use of Excessive Number of Parameters"
has a negative impact on the attribute of well-defined interfaces. The benefit of
using property/attribute pairs (indicators) is that the linkage between the property
and attribute is undeniable. With a sufficient number of objective indicators, an
unbiased analysis of the software product can be performed. To facilitate
automation, all of the properties can be statically detected in an automated manner.
Metrics have been developed that relate the impact of each property on associated
attributes. Requirements for indicators and supporting metrics used in the OPA
framework state that they be simple, understandable, objective, intuitive, and easy
to relate to the attributes [ARTH87].

The major factor separating the OPA framework from the other metrics described in
the literature is that the OPA framework uses many confirming and contrasting
metrics as opposed to one major metric. The many confirming and contrasting
metrics allow for a more complete coverage of the entire spectrum of program
characteristics. Among these are metrics that evaluate interfaces, metrics that
evaluate control structures, and metrics that evaluate the utilization of special
language components to aid software quality.

The Software Metric Development Procedure

As previously noted, metrics such as those offered by Halstead, McCabe, and
Henry and Kafura, tend to concentrate on one program characteristic or component.
The OPA framework differs in that all significant program structures are assessed.
Because of the diversity of Ada components and structures, a procedural approach is needed to relate Ada to the OPA framework. The procedure that was developed gives structure to the research approach and partitions the level of effort into manageable increments. The seven step procedure begins with the identification of crucial language components and results in a set of metrics for the language that assesses the impact of language properties on the software engineering attributes. The following paragraphs outline that procedure.

1. Identifying, Categorizing, and Classifying Crucial Language Components:

The categorization and classification of individual language components support and encourage independent analysis with respect to each component. The analysis is performed by first identifying the language components [ADARM83]. Next, the components are categorized along functional boundaries using a scheme presented by Ghezzi and Jazayeri [GHEZC82]. Finally, the components are further partitioned based upon their relationship to existing Pascal constructs using an Ada-relative-to-Pascal set notation described by Wichmann [WICHB84b]. An example of this step is the identification, categorization, and classification of Ada's exception handler. The component is identified in the Ada Reference Manual. According to Ghezzi and Jazayeri's functional boundaries, it is an implicitly called unit level control structure; using Wichmann's scheme exception handlers are a component of Ada that would require a major redesign of Pascal to add them. Hence, exception handlers are a crucial language component that must be considered within the OPA framework.
2. Understanding the Rationale for Component Inclusion.

Language component rationales often motivate the necessity for including the component in the language definition as well as provide insight into the theoretical uses of the components. For Ada, the language designers have written *Rationale for the Design of the Ada Programming Language* [ADARF84]. Using the example of Ada exception handlers again, this component is defined in the Ada language to support user detection and response to error and terminal situations [ADARF84].


From a software engineering perspective, the impact of using a particular language construct or component is significant within the OPA framework. Such information, found in language rationales and language reviews from literature, provides insight into a component's contribution toward achieving software engineering objectives, supporting accepted software engineering principles and inducing desirable product attributes. An example is name parameter association with respect to Ada subprograms. This component is included in Ada to improve the code readability of subprogram calls with long parameter lists [ADARF84, BOOCG83].

4. Identifying the Impact of Component Usage on Software Engineering Attributes.

The fourth step entails the identification of all possible uses (abuses) for each language component and recognition of the impact that such uses (abuses) can have on product attributes. References used in steps two and three (*Rationale for the
Design of the Ada Programming Language, books, and journal articles) contain this information. Booch [BOOCG83], Shumate [SHUMK88], and others provide considerable detail in their discussions of proposed package uses and possible misuses. For example, the use of Ada packages to define abstract data types has a positive impact on the attributes of cohesion, well-defined interfaces, and ease of change.

5. Identifying Product Properties Reflective of Presence (or Absence) of Attributes.

The important activity in the fifth step of the metric development process is to determine how to detect various uses (abuses) of each component in the code. For each use (abuse), code properties that attest to the attribute impact determined in step four must be identified. An example of applying this step is captured in the mixing of name parameter notation with positional parameter notation in the same subprogram call. This use has a negative impact on readability. The code property that is indicative of this negative impact is "use of both positional and name notation in one subprogram call." Identifying this product property was relatively easy. Other code properties, however, require significantly more work.

6. Defining Product/Attribute Indicators.

This step formally links the code properties identified in step five to the affected attributes identified in step four and justifies such linkages. Through the justifications, the assessment process substantiates each property/attribute indicator. For example, the previous property of "use of both positional and name notation in one subprogram call" has a negative impact on readability. The justification for
such a statement is that using both forms of parameter notation in one subprogram call is confusing to the reader and as a result, reduces the overall readability of the program.

7. Defining the Measure and Supporting Metric for Each Indicator.

The seventh and final step requires an approach to be identified for measuring the indicator defined in step six, and the definition of a metric whose value reflects the justification linking each property/attribute pair. To continue with the example discussed in steps five and six, the measurement approach for assessing the impact on readability of both positional and name notation in one subprogram call is based on an examination of the number of subprogram calls using mixed notation relative to the total number of subprogram calls. The actual metric for this indicator utilizes the following formula:

- Proposed Metric = 5 - 5 * (Percentage of Subprogram Calls Using Both Parameter Notations).

A value of zero for the proposed metric indicates the worst possible impact while a value of zero indicates that no impact can be determined. Although values above five are not possible for this metric, the valid range of values for metrics is from zero to ten. A value above five indicates a positive impact with ten indicating the best possible impact. The procedure presented above provides a guide to developing metrics. During the application of the procedure, the successful completion of one step required the revision of results from a previous step. However, the use of the procedure greatly aided the formulation of metrics within the OPA framework.
Classifying Programming Language Components by Functionality

The first step of the software metric development procedure is to identify, categorize, and classify crucial language components. The identification of the Ada components was through the *Ada Reference Manual*. As mentioned in the discussion of step 1, an organizational scheme is needed to categorize the Ada components. Since the *Ada Reference Manual* presents the Ada language components according to their functionality, a categorization scheme that partitions components in a functional manner appears to be most appropriate. Several schemes were studied for possible use, but the one that proved most advantageous was presented by Ghezzi and Jazayeri in [GHEZC82]. This scheme partitions language components among functional boundaries in a layered approach. The categories in the top level are: (1) data types, (2) statement-level control structures, and (3) unit-level control structures. Each category is discussed in further detail below.

The category, data types, includes all language components used to define variables and type declarations. A few classic examples include the type integer, the array constructor, and the record constructor. Data types can be further divided into a series of subcategories:

- built-in types,
- data aggregates,
- user defined types, and
- abstract data types.
This subdivision breaks down the category of data types to a sufficient level of detail so that they can easily be studied.

The next category suggested by Ghezzi and Jazayeri, statement-level control structures, contains those components which comprise the "meat" of the code. Some examples are loops, decision statements, and assignment statements. Similar to the category data types, the category of statement-level control structures can be broken down into a series of subcategories for study. The subcategories for statement-level control structures are:

- sequencing,
- selection,
- repetition, and
- user defined control structure.

This categorization scheme breaks down statement-level control structures so that the crucial details among structures can be easily distinguished.

The third category of unit-level control structures contains those components that are typically, but not always, thought of at the module level. Some well known examples are subprograms, functions, exception handlers, and concurrency constructs. This category is also decomposed into a series of subcategories. They are:

- explicitly called subordinate units,
- implicitly called units (exception handlers),

Introduction and Background
- symmetrical units - coroutines, and
- concurrent units.

This subdivision, however, does not totally cover unit-level control structures in Ada. In particular, Ada has units such as packages and generics that are unique to Ada and hence, are not contained in any of the previously listed unit-level control structure subcategories. To handle this problem, a fifth subcategory, definition units, is added. This subcategory contains all units which provide a mechanism for simply defining other units.

The Relationship Between Pascal & Ada

As previously noted, the OPA framework has been used to assess products from a block structured language with components similar to Pascal. To effectively utilize the previous research results, any relationship between Pascal and Ada must be noted and exploited during the analysis of Ada. Since the design of Ada is a derivation of Pascal [BOOCG83], many authors have made analogies between Ada and Pascal. One such author, Wichmann, is an Ada design team member and places Ada components into three sets [WICHB84b]. The first set is the Pascal subset of Ada. Elements of this set can utilize attribute/property pairs and metrics presented by Dandekar [DANDA87]. The second set is the elements of Ada that can easily be added to Pascal. Two examples of this set are dynamic arrays and floating point precision. Elements in this set may require slight modifications to the attribute/property pairs and metrics presented by Dandekar. The third set contains elements of Ada that would require a major language revision if added to Pascal. Three examples from this set are tasking, separate compilation, and packages. This particular set of Ada components is the thrust of this research and requires the
development of property/attribute pairs and metrics. References to this set
categorization will also be useful in Chapter 2 when it is necessary to identify all
components in Ada that are not integrated into the OPA framework [ARTHJ87].

Issues That Complicate Metric Development

During the course of analyzing Ada relative to the OPA framework several
complicating issues have risen. For example, obtaining an accurate measure for
lines of code is a non-trivial matter due to multiple definitions for an Ada line of
code. Because most of these complicating issues require a resolution, a summary
of the considerations of each issue and a proposed resolution is given below.

Lines of Code

Counting lines of code accurately is important because this measure is used in the
calculation of many of the metrics resulting from this research effort. Also,
research results often discuss the effort in terms of lines of code. For software
engineers to effectively use this information, an accurate, stable, and uniform
definition of lines of code is needed. Firesmith identifies six current major
definitions [FIRED88]:

1. Physical lines (i.e. carriage returns or line feeds),
2. Non-comment, non-blank physical lines,
3. Semicolons,
4. Terminal semicolons (i.e., all semicolons except for those in strings and
   comments),
5. Limited terminal semicolons (i.e., all terminal semicolons except those in formal parameter lists), and


Using generic instantiations, reused software, and modified software also complicate the issue of deciding on a line count definition [FIRED88].
To illustrate the potential diversity of values that the definitions can spawn, Firesmith uses statistics gathered from the Advanced Field Artillery Tactical Data System (AFATDS). The method of counting physical lines produced a line count of 2,517,594. Similarly, the method of non-comment, non-blank lines produced a line count of 1,175,498. Finally, the method of limited terminal semicolons produced a count of 409,982. For specific details of the implementation of each method see [FIRED88].

In using a method for counting lines of code for this research, two factors are of importance. The first is that it be used consistently so the results of analyzing one product can be compared to another. The second is to ensure the selected definition is reasonable. Of the six alternatives that Firesmith notes, all are reasonable methods for evaluating lines of code, although each can be manipulated. The lines of code measure for this research is the limited terminal semicolon definition.
Moreover, generics units contribute to the definition of lines of code only once.

Overloading

A second complicating issue is overloading. Overloading is an important issue because Ada allows identifiers to be easily overloaded and also permits type, variable, and subprogram names to be overloaded. Overloading exists in many
languages. For example, in Pascal the "+" symbol is the same for both integer and real addition. Selective overloading in a language can aid in promoting simplicity and usability of a language while overuse can have an adverse effect on readability [GHEZC82]. Because of the potential impact on attributes such as readability, overloading is a significant issue in the software engineering world as well as Ada. Generic instantiations are a major source of overloading in Ada. The operator "use" allows the instantiated name to be shortened to the common name declared in the generic. The utilization of the "use" operator is a well debated topic in publications such as Ada Letters. Mendal gives three reasons against utilizing the "use" clause [MENDDG88]. They are:

1. The semantics of the "use" clause are not well known,
2. Visibility changes occur when changes are made to the system, and
3. Utilization of the "use" clause leads to less reliable systems.

In a response article, Racine notes problems with each of the examples that Mendal uses to support his reasoning. According to Racine, the major benefit of the "use" clause is "the potential access to readable, understandable, operations" [RACIR88]. Rosen claims and supports that the "use" clause improves readability, abstraction, reliability, and maintainability [ROSEJ87].

Overloading is considered as it impacts subprograms, operator overloading, independent of overloading created through generic instantiations. Overloading through generic instantiations will be factored in with measures resulting from the analysis of generics.
Functionality

Functionality is another complicating research issue that has an important impact on software engineering development. Statically measuring functionality is a difficult (or even intractable) problem. Most metrics, including those used in the OPA framework, are applied to the static source code. Many uses and abuses of language components are directly related to functionality. Whenever possible, an attempt is made to use a property that indicates a certain functionality. However, some uses and abuses cannot be included in the OPA framework or any other static analysis because no static property indicative of the functionality exists.

Concurrency

The concept of concurrency introduces other complicating problems in the research effort. One problem is that concurrent problems are, by nature, complex and difficult to understand [DIETH84]. Secondly, languages do not generally provide multiple mechanisms for handling concurrency, and many do not even provide a single mechanism. A third problem is that many uses and abuses of concurrency are directly related to functionality which has previously been discussed.

In this research, program concurrency is assumed to be a requirement. No attempt is made to determine whether or not concurrency is really necessary. The impact that concurrency has on the solution is measured in the Ada tasking indicators. To measure the effects that the use of concurrent constructs has on the software product, the synchronization and communication between the concurrent processes are examined. Further details on relating concurrency to the software engineering attributes can be found in Chapters 2 and 3.
Software Complexity

The final complicating issue that results from analyzing Ada relative to Pascal is problems with the use of the attribute "complexity." Kearney, et al. describe many problems and ambiguities with the multiple definitions for complexity found in the literature [KEARJ86]. One major reason for the confusion is the preponderance of "complexity" metrics. Although many of the metrics are measuring some distinct entity, they are all referred to as complexity metrics. For example, McCabe assesses complexity based on the number of decision statements, while Halstead assesses complexity based on the tokens in a program. Zuse and Bollman have also noted that no "generally accepted criteria and methods [currently exist] as to what these software complexity measures should really measure" [ZUSEH87]. Unless otherwise specified, the meaning for complexity in this thesis will be as specified by Arthur and Nance. Complexity is "an abstract measure of work associated with a software component" [ARTH87, p. 46].
Chapter 2

Assessing the Impact of Crucial Ada Components on the Software Engineering Attributes

Chapter 1 provides a background overview and discusses a seven step procedure for developing metrics reflecting the software engineering contribution of crucial Ada components. Crucial Ada components are identified as being important Ada components that must be considered during the quality assessment of Ada products relative to the OPA framework. This chapter considers the major Ada language components, determines those that are crucial (relative to the OPA framework for software quality assessment), and discusses their relationship to software engineering.

The discussion follows a logical progression that reflects the language decomposition suggested by Ghezzi and Jazayeri [GHEZC82] compounded by a further partition described by Wichmann [WICHB84b]. That is, the desirable software engineering...
qualities of the Ada components are presented based on a language concepts/components partitioning within functional boundaries.

Data Types

The first category of components to be studied is data types. Data types have one common characteristic: operations are performed on them; data types do not perform operations on other objects. The subcategories of data types, with a few examples are:

- built-in types - integers, reals, and characters,
- data aggregates - arrays,
- user defined types - records and variant records, and
- abstract data types - stacks, queues, and lists.

With the completed subdivision of language components along functional boundaries, Wichmann's scheme of analyzing the Ada components relative to Pascal components can now be applied to the category of data types. Table 4 on page 32 illustrates the results of this analysis. Each data type subcategory is discussed following the order presented in the table.

Built-In Data Types

Built-in data types are indivisible data representation forms provided by the language. Examples include enumerated types, characters, booleans, integers, and real types. The Pascal set of built-in data types is similar to the Ada set except for one small distinction. Ada allows both floating point and fixed point numbers of user specified precision.
Table 4. Analysis of Ada and Pascal Data Types

<table>
<thead>
<tr>
<th>Data Type Category</th>
<th>Ada</th>
<th>Pascal</th>
</tr>
</thead>
<tbody>
<tr>
<td>built-ins</td>
<td>enumerated</td>
<td>enumerated</td>
</tr>
<tr>
<td></td>
<td>character</td>
<td>character</td>
</tr>
<tr>
<td></td>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td></td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>floating point</td>
<td>real</td>
</tr>
<tr>
<td></td>
<td>fixed point</td>
<td></td>
</tr>
<tr>
<td>aggregates</td>
<td>access</td>
<td>pointers</td>
</tr>
<tr>
<td></td>
<td>arrays</td>
<td>arrays</td>
</tr>
<tr>
<td></td>
<td>strings</td>
<td>set</td>
</tr>
<tr>
<td></td>
<td></td>
<td>file</td>
</tr>
<tr>
<td>user defined types</td>
<td>records</td>
<td>records</td>
</tr>
<tr>
<td></td>
<td>variant records</td>
<td>variant records</td>
</tr>
<tr>
<td></td>
<td>discriminants</td>
<td></td>
</tr>
<tr>
<td>abstract data types</td>
<td>packages</td>
<td></td>
</tr>
</tbody>
</table>
[ADARM83] while Pascal has the one type, real. Because the floating and fixed points
types are not substantially different from the one Pascal type, real, they are not considered
in the metric development discussion.

Data Aggregates

Data aggregates include types such as pointers, arrays, and sets. Within this category Ada
and Pascal aggregates differ in three ways. First, Ada has the additional type "strings."
Strings are one dimensional arrays of characters, hence they are simply a special case of
arrays. This special case of arrays was added to Ada to facilitate string manipulations
[ADARF84]. Strings are not considered in the metric development process because they
are merely a specialized case. Figure 2 on page 34 provides three examples of string
declarations and default initializations.

The other two differences between Ada and Pascal data aggregates arise in the Pascal types
sets and files. Ada contains no equivalent data type for these two aggregates. Instead, files
are implemented with the Ada library generic package, "TEXT_IO" [ADARM83].
Habermann shows that Pascal sets can be implemented with a user defined Ada package
[HABEA83]. Therefore, these two distinctions do not need to be considered in the metric
development process.

User Defined Types

The third category of data types is user defined types. User defined types are those types
that are typically defined in terms of other types. Ada user defined types are defined using
records, variant records, and discriminants while Pascal user defined types are defined with
FIVE_CHARS : STRING(1..5); -- no initial value
MESSAGE : STRING(1..25) := (MESSAGE'range => '');
STRING_CHEESE : STRING(1..6) := ('C','h','e','e','s','e');

Figure 2. String declaration examples from Gilpin [GILPG86].
only records and variant records. In the Ada domain, a discriminant is a record with a
 discrete component that is a parameter to the record [ADARM83]. Variable declarations of
discriminants can be either constrained or unconstrained. A constrained discriminant is
given a value in the variable declaration and cannot be changed during execution. An
unconstrained discriminant uses the default value (a constrained discriminant is allowed
only if there is a default value) for the initial value and can be changed during execution.

Figure 3 on page 36 gives a few examples of discriminant type declarations. Discriminants
are included in the Ada language to allow a more precise modeling of data dependencies
than can be achieved through Cartesian records [ADARM83].

Discriminant records are parameterized records, allowing exact specification and
instantiation to occur at run-time. Exploiting this dynamic nature of discriminants can have
a significant impact on the software engineering attributes. Using discriminants as
repositories for data with dynamic requirements is a less complex solution than "standard"
records [PRATT84]. A "standard" record uses extra fields and maximum sizes to handle
potential requirements. A discriminant is defined in terms of its dependencies.

Furthermore, discriminants, when used as parameters, add definitional clarity to the
interface because of the precise data modeling potential of discriminant records
[ADARF84]. Discriminants also have a potential impact on early error detection by
encouraging abstractions to be made earlier in the life cycle, thereby exposing potential
weaknesses before implementation. The effects of discriminants are summarized in
"Property/Attribute Pairs for Identified Indicators" on page 110.

Abstract Data Types

The final subcategory in data types is abstract data types. Pascal does not have any special
components for the implementation of abstract data types. One proposed use for Ada
-- Below is a type declaration with a default value.
type BUFFER(SIZE : BUFFER_SIZE := 100) is
  record
    POS    : BUFFER_SIZE := 0;
    VALUE  : STRING(1..SIZE);
  end record;

-- Below is a type declaration without a default value.
type SQUARE(SIDE : INTEGER) is
  record
    MAT : MATRIX(1..SIDE, 1..SIDE);
  end record;

-- Below are declaration examples.
LARGE  : BUFFER(200); -- constrained, always 200
MESSAGE : BUFFER;     -- unconstrained, initially 100

BASIS   : SQUARE(5);  -- constrained, always 5 by 5
ILLEGAL : SQUARE;      -- illegal, a SQUARE must be constrained

Figure 3. Discriminant examples from the Ada Reference Manual [ADARM83].
packages is the implementation of abstract data types [ADARF84]. Packages are considered in detail in the section "Unit Level Control Structures" on page 43.

An analysis of data types reveals one crucial language component in Ada that is not found in Pascal, record discriminants. Record discriminants potentially impact the software engineering attributes of early error detection, well-defined interface, and complexity in a positive manner. No adverse effects for record discriminants have been identified. Record discriminants are considered in the next chapter when formulating an approach to assess their impact on the identified software engineering attributes.

Statement Level Control Structures

According to Ghezzi and Jazayeri [GHEZC82], the second category of language components is statement level control structures. Statement level control structures are characteristically the "meat" of the code which handle everything from decisions to assignments and sequencing. Below is a list of the subcategories of statement level control structures:

- sequencing,
- selection,
- repetition, and
- user defined control structures.

Similar to the approach discussed in the previous section, the author applies Wichmann's partitioning scheme on top of Ghezzi and Jazayeri's scheme to assist in identifying crucial statement level control structures defined in Ada (see Table 5 on page 38). A discussion of
Table 5. Analysis of Ada and Pascal Statement-Level Control Structures

<table>
<thead>
<tr>
<th>Statement-Level Control Structures</th>
<th>Ada</th>
<th>Pascal</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequencing</td>
<td>line oriented</td>
<td>line oriented</td>
</tr>
<tr>
<td></td>
<td>begin..end</td>
<td>begin..end</td>
</tr>
<tr>
<td></td>
<td>assignment</td>
<td>assignment</td>
</tr>
<tr>
<td></td>
<td>array slicing</td>
<td></td>
</tr>
<tr>
<td>selection</td>
<td>if statement</td>
<td>if statement</td>
</tr>
<tr>
<td></td>
<td>case statement</td>
<td>case statement</td>
</tr>
<tr>
<td>looping</td>
<td>for loop</td>
<td>for loop</td>
</tr>
<tr>
<td></td>
<td>while loop</td>
<td>while loop</td>
</tr>
<tr>
<td></td>
<td>goto statement</td>
<td>goto statement</td>
</tr>
<tr>
<td></td>
<td>exit loop</td>
<td></td>
</tr>
<tr>
<td></td>
<td>named loop</td>
<td></td>
</tr>
<tr>
<td>block</td>
<td>block structure</td>
<td></td>
</tr>
<tr>
<td>user defined</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Assessing the Impact of Crucial Ada Components on the Software Engineering Attributes 38
each subcategory identified in Table 5 on page 38 is presented below. One exception to the presentation format involves user defined control structures. This subcategory is excluded from the discussion because neither Ada nor Pascal have user defined control structures.

Sequencing Structures

Ada and Pascal both provide sequencing structures: fundamental code structures and methods of ordering code. Ada and Pascal both provide code sequencing through line orientation and support through explicit constructs such as "begin ... end" blocks and assignment statements. Unique to Ada, array slicing allows a portion of one array to be assigned to an array section of the same size and compatible type. Array slicing was included in Ada to facilitate assignments of arrays of differing sizes. Figure 4 on page 40 contains examples of array slicing. Array slicing can be easily simulated with a loop and assignment statements; this component is not a crucial language component.

Selection Structures

The second subcategory in statement level control structures is selection. The flow of control is stated using selection structures. Examples include the "if statement" and the "case statement." Except for minor differences in syntax, both Ada and Pascal selection structures are similar. Therefore, no selection structures are included in the crucial components for further metric development study.

Looping Constructs

The third category of statement level control structures are looping constructs. Looping constructs block groups of statements for repetitious execution. Examples include "for
VALVE_RECORD(1..20) := VALVE_RECORD(21..40);
VALVE_RECORD(1..10) := VALVE_RECORD(6..15);

Figure 4. Array slicing examples from Booch [BOOCG83].
loops," "while loops," and "goto" statements. Ada loop structures have two additional features not found in Pascal: loop exits and loop names. Using loop exits and loop names, loops can be constructed that have multiple exits and exits that can jump one or more nested levels. An example of loop exits and loop names is provided in Figure 5 on page 42.

The loop exit statement adds the capability to exit from a loop at any point. When the loop exit construct includes a loop name, the exit statement can be used to jump out of nested loops. The loop exit statement and loop names can produce multiple exits from loops. Arthur and Nance state the detrimental effects of multiple exit points from loops in [ARTHJ87]. To summarize Arthur and Nance's findings, complexity increases and readability decreases due to the discontinuity of code with multiple exits. The discontinuity also lessens code cohesion. Because multiple exit points from loops is an indicator in the OPA Framework as discussed by Dandekar [DANDA87] and Farnan [FARNM87], loop exits and names need not be re-assessed relative to the current metric development research for Ada.

Block Structures

The fourth subcategory of statement level control structures deals with block structures. A block structure defines a sequence of statements with an optional declarative part and optional exception handlers [ADARM83]. No equivalent to the block structure exists in Pascal. Figure 6 on page 43 provides an example of block structures and variable scopes.

The block structure provides the capability to move declarations closer to where they are needed or localize them [WATTD87], supporting code cohesion. As the name of the block statement suggests, the block statement binds individual code elements into a functionally cohesive unit. The significance of this code binding is similar to "if statements" and "for
\[
X := 10; \\
\text{COUNT\_DOWN:} \\
\text{loop} \\
\quad \text{put (X);} \\
\quad X := X - 1; \\
\quad \text{exit COUNT\_DOWN when } X = 0; \quad \text{-- exit when } X = 0; \\
\quad \text{NEW\_LINE;} \\
\text{end loop COUNT\_DOWN;} \\
\text{put ("Blast off!");}
\]

Figure 5. Named loop and loop exit example from Gilpin [GILPG86].
OUTER:
    declare
    x : integer;
begin
BLOCK1:
    declare
    X : integer;
begin
    OUTER.X := 100;
    BLOCK1.X := 150;
    X := 200; -- refers to BLOCK1.X
end BLOCK1;
end OUTER;

Figure 6. Block and overloading example from Gilpin.
loops." The impact is an improvement of code cohesion by binding statements together, and an improvement to readability and complexity by presenting the reader smaller "blocks" of code to read. The OPA framework classifies structures such as "if statements" and "for loops" as control structures. Because this naming scheme is inappropriate for the block statement, control structures will henceforth be referred to as block structures and will include the block statement.

The above analysis reveals two crucial components of the Ada language that have not previously been considered within the OPA framework: (1) loop exits and loop names and (2) block structures. The analysis reveals one crucial Ada component for further study. The block statement needs to be integrated into the existing OPA framework metrics for blocking structures.

Unit Level Control Structures

The last category suggested by Ghezzi and Jazayeri, unit level control structures, provides structures for grouping other language components. The four subcategories described by Ghezzi and Jazayeri do not include such important Ada components as packages and generics. Packages and generics are structures only to group other structures into one definitional unit. With the addition of definitional units, the five subcategories of unit level control structures are:

- explicitly called subordinate units - subprograms,
- implicitly called units - exception handlers,
- symmetrical units - coroutines,
- concurrent units - tasks, and
- definitional units - packages and generics.
Using these five subcategories and Wichmann's analysis of Ada relative to Pascal leads to
the decomposition outlined in Table 6 on page 46. Note that this table contains an
additional subcategory, "parameters." This subcategory was factored out of the two
subcategories, subprograms and concurrent routines, to prevent duplication. With the
exception of coroutines, which are omitted because neither Pascal nor Ada have coroutines,
unit level control structures are presented below utilizing the order of Table 6 on page 46.

Before the actual discussion of unit level control structures can begin, however,
terminology relating to the development of indicators must be discussed. An indicator is a
product property with an undeniable linkage to a specific software engineering attribute. At
this stage of the analysis, preliminary indicators exist in the form of factors. A factor is a
use or concept associated with a language component. A factor differs from an indicator in
that its associations have yet to be established. The purpose of discussing factors is to
discover relationships between factors and attributes that potentially evolve into indicators.
The discussion of unit level control structures in this chapter notes significant component
factors and attributes that impact these factors.

Subprograms

Ada and Pascal both offer equivalent subprogram forms, procedures and functions.
Functions differ only in the trivial detail of how data is returned and the types of data that
can be returned. One factor relating to subprograms, however, is the overloading of
subprogram names. Pascal does not permit the user to overload a subprogram name while
Ada does allow the user to overload any subprogram name [ADARM83]. Overloading of
subprogram names reduces readability as the number of overloaded names increases. One
subprogram name that refers to two or more subprogram definitions causes
Table 6. Analysis of Ada and Pascal Unit-Level Control Structures

<table>
<thead>
<tr>
<th>Unit-Level Control Structure</th>
<th>Ada</th>
<th>Pascal</th>
</tr>
</thead>
<tbody>
<tr>
<td>subprograms</td>
<td>procedures</td>
<td>procedures</td>
</tr>
<tr>
<td></td>
<td>functions</td>
<td>functions</td>
</tr>
<tr>
<td>parameters</td>
<td>in, out, in out</td>
<td>reference, value</td>
</tr>
<tr>
<td></td>
<td>defaults</td>
<td></td>
</tr>
<tr>
<td></td>
<td>positional</td>
<td>positional</td>
</tr>
<tr>
<td></td>
<td>name</td>
<td></td>
</tr>
<tr>
<td>exception handlers</td>
<td>exception handlers</td>
<td></td>
</tr>
<tr>
<td>coroutines</td>
<td></td>
<td></td>
</tr>
<tr>
<td>concurrent routines</td>
<td>tasking</td>
<td></td>
</tr>
<tr>
<td>definition</td>
<td>packages</td>
<td></td>
</tr>
<tr>
<td></td>
<td>generics</td>
<td></td>
</tr>
</tbody>
</table>
understandability problems in two main areas: (1) differences in the parameters of the subprograms and (2) differences in the semantics and functionality of the subprograms. For example, the symbol "X" can represent both the multiplication and dot product of matrices. Overloading is a crucial concept and needs to be considered for a quality assessment of Ada products.

Parameters

The second subcategory of unit level control structures deals with parameter issues. As noted earlier the parameter issues are common to both subprograms and tasks. Ada offers three parameter passing modes (in, out, in out); only two are offered by Pascal (value, reference). Ada, however, permits default values for parameters and Pascal does not. Default parameters are a useful feature for values which remain relatively stable over several calls [BOOCG83]. An example of a subprogram specification with defaults defined and example calls to it is provided in Figure 7 on page 48.

Another parameter consideration is the use of parameter notations. The two prevalent notations are name association (the formal parameter and actual parameter are both shown in the call) and positional notation (the actual parameter's position in the call determines which formal parameter it relates to). Ada permits both name and positional parameter notation while Pascal permits only positional notation. The chief benefit to be gained by using name parameter notation is that the readability of the subprogram call is improved by the clear association of the formal and actual parameters [ADARM83]. See Figure 8 on page 49 for a sample subprogram specification and calls to it using both positional and name parameter notation.
-- Below is a subprogram specification for the example calls.
procedure ACTIVATE( PROCESS : in PROCESS_NAME;
                   AFTER  : in PROCESS_NAME :=
                   NO_PROCESS;
                   WAIT   : in DURATION    := 0.0;
                   PRIOR  : in BOOLEAN     := FALSE);

-- Below are the example calls with use of default parameters.
ACTIVATE(X);
ACTIVATE(X, AFTER => Y);
ACTIVATE(X, WAIT => 60.0, PRIOR = TRUE);
ACTIVATE(X, Y, 10.0, FALSE);

Figure 7. Default parameter examples from Ada Reference Manual [ADARM83].
-- Below is a subprogram specification for the example calls.
procedure PRINT_HEADER(PAGES : in NATURAL;
    HEADER : in LINE   := (1..LINE'LAST => ' ');
    CENTER : in BOOLEAN := TRUE);

-- Below are example calls, all are equivalent.
PRINT_HEADER(128, TITLE, TRUE);
PRINT_HEADER(128, HEADER => TITLE, CENTER => TRUE);
PRINT_HEADER(HEADER => TITLE, CENTER => TRUE,
    PAGES => 128);

Figure 8. Name and positional parameter examples from Ada Reference Manual [ADARM83].
Most of the uses (and abuses) relating to default parameters and name parameter notation overlap. For example, name parameter notation is instrumental in the effective use of default parameters [BOOCG83, WICHB84a, HARDW76]. Related factors that have a significant software engineering impact are:

- Use of Default Parameters with Positional Notation,
- Definition and Use of Default Parameters for Stable Values,
- Use of Parameters with Name Notation,
- Mixing the Order of Parameter Lists, and
- Use of both Positional and Name Notation in a Single Subprogram Call.

These factors primarily affect the same three attributes: complexity, readability, and well-defined interface. As discussed below some of the factors have positive impacts and others have negative impacts. The commonality stems from the close relationship that these two crucial language components have with respect to their rationale. Both are included in the Ada language to improve the readability or clarity of interfaces (both subprogram and task) [BOOCG83, HARDW76, WICHB84a]. Understanding a subprogram call is directly related to the readability of that call; if readability is made easier, then the call benefits from both improved readability and decreased complexity. Additionally, the clarity of a call relates to the quality of the interface. Since improved readability and decreased complexity are indicative of improved clarity, the attribute, well-defined interface, benefits as well.

A similar argument can be made for cases that the abuse (or non-use) of default parameters and name notation results in a negative impact on three attributes: complexity, readability, and well-defined interfaces. A reduced ability to decipher a call hinders readability and increases complexity. As the clarity of the call is reduced, the clarity of the interface is also
reduced. For example, using default parameters without name parameter notation produces calls that appear to have missing parameters. This reduces the clarity or understandability of the call, producing a negative impact on the three identified attributes. Similar arguments can be made for (a) using name parameter notation to mix the order of parameter lists and (b) using both name parameter notation and the traditional positional parameter notation in one call.

Using default parameters with name parameter notation provides a clear, concise, and readable call. This use benefits readability, complexity, and well-defined interfaces. Similarly, use of name parameter lists, especially for longer parameter lists, improves readability, reduces complexity, and aids well-defined interfaces. A tabular format for the information presented in this section can be found in "Property/Attribute Pairs for Identified Indicators" on page 110.

Exception Handlers

Exception handlers, the third subcategory in unit level control structures, are found in Ada but not in Pascal. Exception handlers are included in the Ada language because of the need to handle and respond to erroneous conditions that would otherwise cause program termination [ADARF84,BOOCG83]. The proper handling of exceptions improves program reliability, a necessity for one of Ada's target applications - real-time systems.

The design rationale of Ada's exception handling implementation stems from the need for termination code because an error causes the need for abandoning the code. This type of error handler offers four options: (1) abandon the execution of the unit, (2) try the operation again, (3) use an alternative approach, and (4) repair the cause of the error [BOOCG83]. Examples of these options are included in
"Examples of Methods of Using Ada Exception Handlers" on page 107.

Because all of the methods listed above relate to functionality, it is not feasible to attempt to distinguish each of them in the code. Instead, the approach to analyzing exception handlers is to identify abuses (one significant abuse was found) and to integrate the impact that the general use of exception handlers has on a program. The two resulting factors are:

- Use of Exception Handling and
- Relying on Upper Level Subprograms to Handle Errors.

To identify the attributes that are affected by "use of exception handling," the OPA framework is used in a top-down evaluation manner from the objective, reliability. The primary attributes shown to benefit from "use of exception handling" are coupling, cohesion, and complexity. An intuitive justification to support this premise is that use of exception handlers makes the code more independent of the calling module (coupling), binds the code together (cohesion), and that exception handlers are easier to use than simulating them in the code (complexity).

The next factor, "relying on upper level subprograms to handle errors," relates to modules that raise an error but do not have an exception handler to deal with the error. This causes the module to rely on the exception handler in one of the upper level calling modules which in turn, creates a dependence, hence increased coupling.

**Concurrent Routines: Tasking**

Concurrent units, the fifth subcategory in unit level control structures, execute simultaneously. Ada achieves concurrency through its tasking mechanism; Pascal has no
mechanism to support concurrency. The Ada tasking model is included in the Ada
definition because of concurrency demands in real-time systems [ADARF84]. Because the
designers of Ada felt that semaphores, signals, and events are too low-level to be easily
understood and that monitors are too high-level, they attempted to develop a mechanism
which was a good mix of both high-level and low-level constructs. Tasking is the resulting
mechanism [ADARF84]. The Ada tasking model utilizes rendezvous between units to
exchange information. For an example of tasking, see "Tasking Demonstration to Find
Prime Numbers" on page 103.

Substantial literary evidence supports the conclusion that tasking adversely affects
complexity and readability. Deitel writes that "concurrent programs are normally more
difficult to write, debug, and prove correct than sequential programs" [DEITH84].

Intuitively, requiring the programmer and program reader to mentally comprehend multiple,
simultaneous processes is a difficult task. Programming and understanding concurrent
processes leads to two factors: (1) use of tasking and (2) definition of tasks. These two
factors relate the use and definition of tasks to the adverse impact on the attributes,
complexity and readability.

The next concurrency related factor deals with the communication process during tasking,
the "rendezvous." Ada tasks pass information in a manner similar to subprograms.
Communication can be "in" only, "out" only, or both "in" and "out." Nielsen argues that
the latter form, both "in" and "out," needlessly couples the processes together [NIELK86].
Communication which is unidirectional (i.e. in or out only), claims Nielsen, is best because
the purpose of a tasking rendezvous is to either give or receive a piece of information and
then continue processing. Therefore, the factor "Use of Rendezvous with In Out or In and
Out Parameters" creates excessive coupling.
Global variables coupled with concurrency produce an extremely undesirable side effect. In addition to the normal detriment caused by using global data, there is the additional problem of indeterminate access that results from concurrent processes updating unprotected global data areas. Ease of Change is impaired because repeating the exact conditions that lead to the erroneous results may be difficult to do. Coupling is increased due to the multiple alliances created dynamically between units. Hence, the attributes of ease of change and coupling are both negatively affected by the factor, "Use of Unprotected Global Data Areas."

As illustrated in "Property/Attribute Pairs for Identified Indicators" on page 110, the significant factors related to Ada tasking having a measurable impact on software engineering qualities are:

- Use of Tasking,
- Definition of Tasks,
- Use of Rendezvous with In Out or In and Out Parameters, and
- Use of Unprotected Global Data Areas.

These are discussed further in the next chapter.

Definitional Units

The sixth and final subcategory of unit level control structures is definitional units. Definitional units are those units through which subprograms and objects are defined; definitional units are not, however, executable. Ada has two definitional units: (1) packages and (2) generics. Pascal does not have an equivalent structure for either of these.
Packages

The *Rationale for the Design of the Ada Programming Language* states three proposed uses of packages [ADARF84]:

- Named Collections of Entities (Named Collections of Declarations),
- Groups of Related Subprograms (Groups of Related Program Units), and
- Encapsulated Data Types (Abstract Data Types).

Booch [BOOCG83] adds a fourth use, "Abstract State Machines." Using packages for these purposes "directly supports the principles of modularity, abstraction, localization, and information hiding" [BOOCG83]. An example of an Ada package used to implement the abstract data type "queue" can be found in the "Package Implementation of Abstract Data Type Queue Example" on page 95. The major distinction between Ada and many other programming languages (Pascal for example) is that Ada contains components (like packages) to enforce and encourage the use of software engineering principles [ADARF84, BOOCG83, LAWLP88].

Packages are a mechanism provided by Ada for grouping constants, type declarations, variables, and/or subprograms. The goal is that the programmer will use packages to group logically related items [ADARF84]. Packages are also a powerful tool for providing abstractions. The ability to localize implementation details and to group related collections of information is a prerequisite for developing abstract data types in a language. As discussed by Shaw [SHAWM80] Ada provides this capability through the use of packages. In addition to developing abstract data types, Ada packages can be used to group named collections of declarations, provide subprogram libraries, and implement abstract state machines [BOOCG83] [GANNJ86] [ADARF84] [SHUMK88].
An extensive analysis of the four proposed uses of packages reveals two interesting observations. The first observation concerns the exporting of subprograms. Declaration packages export type and constant declarations (i.e. do not export any subprograms or functionality). The other three proposed uses of packages have one major commonality, the exporting of subprogram units. A close analysis of packages reveals that the distinction of exporting subprograms is an important factor in assessing packages with respect to the software engineering attributes. The second observation distinguishes between the actual definition of a package and another unit "withing" the package. The impact of a package varies depending on the perspective. For example, the actual definition of packages usually impacts attributes such as ease of change and cohesion. However, the utilization of a package by another unit impacts the attributes of complexity and readability. The two significant observations of packages combined with two abuses of packages produces the following package factors:

- Definition of Declaration Packages,
- Insufficient Decomposition of Declaration Packages,
- Definition of Packages that Export Subprograms,
- Units that "with" Packages that Export Subprograms, and
- Definition of Packages that are Never "withed."

The definition of declaration packages has a significant, positive impact on two attributes, cohesion and ease of change. Booch justifies the relationship noted above by stating that "one of the simplest uses of packages is for the logical grouping of objects and types. This application benefits maintainability by factoring out common data, objects, and types and placing their definition in one location" [BOOCG83, p. 193]. Similarly, Ross notes that it is a particularly good use for data structure design methodology to group similar
declarations together as one cohesive unit. Moreover, such a grouping also benefits ease of change [ROSSD86].

One can, however, overdo the grouping of declarations within a package. In particular, the grouping together of too many declarations or loosely cohesive declarations together leads to insufficient decomposition of declaration packages. A change to an insufficiently decomposed declaration package necessitates verifying whether or not each "withing" unit actually uses the changed declaration. The lower the level of decomposition, the more adverse the impact on ease of change. Hence declaration packages should be kept relatively small. Declaration packages with large numbers of weakly cohesive constants and types should be decomposed at least one more level. In addition, Booch notes that smaller packages aid ease of change by decreasing unnecessary recompilations [BOOCG83].

The definition of packages that export subprograms positively affects the attributes of well-defined interface, ease of change, and cohesion. Grouping related subprograms together provides a tight, implementation independent interface while enhancing ease of change [BOOCG83]. Yourdon notes (before Ada existed) a need to group logically related units in cohesive units [YOURE78]. Another software engineering improvement that results from the definition of packages that export subprograms is the increased modularity, thereby improving ease of change [GANNJ86].

The next factor is units which "with" packages that export subprograms. Since the user of a package need never see the body, abstractions can be made that improve code readability and reduce the complexity of the code [BOOCG83]. Similar to the previous factor, the modularity achieved by using packages reduces the complexity of the code [GANNJ86].

The final factor relating to packages is the definition of packages that are never "withed."
Recognizing this factor reveals the existence of code without any additional functionality. This adds to the work associated with understanding the program or the program complexity.

**Generics**

In addition to packages, generics are another definitional structure. The generic is a template for defining packages and subprograms. Using Ada generics provides a further level of abstraction, thereby extending the power of the language [ADARM83, ADARF84, BOOOG83]. For example, the abstract data type package example in "Package Implementation of Abstract Data Type Queue Example" on page 95 is extended to a generic package in "Generic Package Implementation of Abstract Data Type Queue Example" on page 99. Comparing the code of the two packages illustrates that a few modifications create a generic package with increased usability potential.

The ability to make abstractions is one of the greatest contributions of packages. Generics are similar in that they allow a further factorization to occur. A generic consists of taking one of the program units (subprograms or packages) and factoring out one or more parameters. The purpose of factorization is to reduce source code and hence the size of the program text [ADARF84]

According to the *Rationale for the Design of the Ada Programming Language* the software engineering reasons for the inclusion of generics in Ada are to improve maintainability, readability, and efficiency [ADARF84]. Although it is not explicitly stated, one can infer that the goal is to maximize the efficiency of the source code with respect to size and duplications. The study of Ada generics reveals the following significant factors:
• Definition of Parameterless Generics,
• Use of Subprograms as Generic Parameters,
• Definition of Generic Units, and
• Instantiation of Generic Units.

The first generic factor is the definition of parameterless generics. A parameterless generic can be used to provide a separate set of variables, types, and/or procedures for a module. Definition of parameterless generics leads to multiple packages with no distinction between them. Non-differing duplications cause confusion and hence, increase complexity [WATTD87].

Pascal subprograms have the option of a procedure or function as a parameter. Ada does not have an explicitly defined capability. Using a subprogram as a generic parameter, however, is one way of simulating a similar functionality. Nonetheless, this use can increase complexity and reduce readability through use of a nondescriptive subprogram name in the code. Booch states that the use of subprogram parameters to generics reduces program clarity [BOOCG83].

The factor, definition of generic units, aids the attributes of complexity and readability. The improvements occur due to an additional layer of abstraction a generic can provide. The abstraction allows the use of generics without detailed knowledge of their implementation [WATTD87] which aids both complexity and readability. Booch supports the relationship between this factor and complexity with "generic program units are powerful tools that can assist the programmer in managing the complexity of his or her software solution" [BOOCG83, p. 217].

The final factor relating to generics recognizes the benefit of instantiating multiple copies of
a generic unit. The benefit to ease of change with regard to generics increases as the number of generic instantiations increases. A generic with only one instantiation has no benefit to ease of change; whereas a generic with five instantiations achieves beneficial results with respect to ease of change. Only one program unit need be changed instead of five.

Summary

With the discussion of the subcategory definition structures, the analysis of unit level control structures is complete. The analysis identifies the following crucial components of Ada that are not found in Pascal: (1) default parameters, (2) name parameter notation, (3) exception handlers, (4) tasking, (5) packages, and (6) generics. Unique to the crucial components identified in unit level control structures, each results in numerous factors stemming from one component. For example, packages resulted in five factors, each impacting one or more attributes. The discussion given in this chapter for each of these factors and affected attributes is continued in Chapter 3. Factors evolve into indicators through a process of formalizing the factor into an identifiable product property linked to a specific software engineering attribute. The linkage between the product property and the attribute is a justification that is intuitive, and substantiated with a preponderance of literary evidence. Along with evolution, Chapter 3 describes the measures and supporting metric for each identified property/attribute indicator.
Chapter 3

Developing Indicators, Measures, and Metrics for Crucial Ada Components

The previous chapter, Chapter 2, discusses the application of steps one through four of the software metric development procedure. The results of applying steps one through four are a set of crucial components delineated along functional boundaries. For each crucial component, an analysis is performed that gives the component's language rationale, impact on software engineering, and effect on the software engineering attributes. To complete the presentation of the results of the software metric development procedure, Chapter 3 discusses the property/attribute indicators, measures, and supporting metrics developed in steps five through seven of the procedure.

The development of the attribute/property indicators does not require any new analysis or justification. This work has been completed in the identification of "factors" in Chapter 2.
This step of the procedure simply takes the factors and the affected attributes described in Chapter 2 and molds them into property/attribute indicators. Additionally, the justification for each of these property/attribute indicators is given in "Property/Attribute Pairs for Identified Indicators" on page 110.

Once formal property/attribute indicators are formed, a measure and supporting metric must be identified. The measure must be an identifiable product property indicative of the impact that a factor has on an attribute. Additionally, the OPA framework stresses that the supporting metric for each measure must be simple and intuitive. The discussion for this chapter follows the order presented by Ghezzi and Jazayeri for language components. The subsections to be discussed are data types, statement level control structures, and unit level control structures.

Data Types: Discriminants

The application of steps one through four of the software metric development procedure in Chapter 2 reveals one crucial data type component of Ada, discriminants. Recall that discriminants are parameterized records that allow a precise modeling of structured data types. The precise modeling capabilities potentially induce a positive effect on the attributes of complexity and early error detection. The formal property/attribute indicators for this factor are:

- Use of Record Discriminants
  - Complexity (+) and
  - Early Error Detection (+).
The developing of a measure for an indicator necessitates an understanding of what properties are indicative of the indicator. In this case record discriminants can be assessed from (1) the number of discriminant types defined, (2) the number of discriminants used in parameter lists, (3) the number of discriminant variable declarations, and (4) the number of discriminants used in assignment statements. Note that this is not a complete list and that the reader can probably supply further measures for discriminants.

The requirements of measures and metrics developed for the OPA framework is that they be simple, intuitive, and directly related to an attribute. The use of all four proposed measures for discriminants would surely result in a loss of simplicity and intuitiveness. Therefore, the measure uses the most significant factor; occasionally the two most significant factors must be used. For example, to relate the effect of record discriminants on the attribute of early error detection, the most intuitive measure would be the number of discriminants types defined. The number by itself is meaningless. The impact of the definition of discriminant types depends on the number of user defined types.

A more appropriate measure for assessing the impact of record discriminants on early error detection is the number of discriminant types defined relative to the total number of user defined types. Since this indicator is identified as having a positive effect only, the final metric uses the measurement approach described above and scales it between five and ten. A value of five for the metric indicates no impact on the attribute and a value of ten indicates the maximum beneficial impact on the attribute.

The application of similar reasoning with respect to the attribute of complexity results in the same measure as early error detection. "Measures and Metrics for Ada Specific Indicators" on page 135 provides a complete analysis of the rationale, measurement approach, and final metric for the discriminant property/attribute indicators.
Statement Level Control Structures: Block Statement

The discussion of statement level control structures in Chapter 2 reveals a crucial Ada component not currently integrated in the OPA framework, the block statement which promotes higher code cohesion and improved code readability while reducing complexity. The indicators for this factor are:

- Use of Block Statements
  - Cohesion (+),
  - Complexity (+), and
  - Readability (+).

The measurement approach for these indicators utilizes the work by Danekar [DANER87] by extending his notion of "control structures" to include block statements. The term "control structures" was renamed to "blocking structures" in Chapter 2. The measurement approaches for these indicators vary slightly among attributes, but the general flavor is the same. The entire program is analyzed to determine an average number of blocking structures. The average can then be used to determine an expected number of blocking structures per subprogram. The deviation from the expected number of blocking structures is used to measure the impact of blocking structures on the three attributes. "Measures and Metrics for Ada Specific Indicators" on page 135 contains a detailed analysis of the measurement approaches and supporting metrics for each of the blocking structure indicators.
Unit Level Control Structures

The analysis in Chapter 2 of unit level control structures notes six areas for study. The first two areas, subprograms and parameters, contain additional capabilities or components of Ada not found in Pascal that are discussed. Additionally, four distinct components with no Pascal equivalents are also found in Ada. These are packages, generics, exception handling, and tasks. The discussion for the components will give the property/attribute indicators derived from the factors presented in Chapter 2. In addition to the property/attribute indicators, the resulting measures and supporting metrics for these indicators will be described.

Subprogram Issue: Operator Overloading

The overloading of subprogram names hampers readability in direct proportion to the number of overloaded subprogram names. The property/attribute indicator resulting from this factor is:

- Overloading of Subprogram Names
  - Readability (-).

The number of unique overloaded subprogram names relative to the total number of unique subprogram names measures the impact of this indicator. However, this does not fully capture the adverse effect of overloading subprogram names. Because having only half of the subprogram names overloaded creates tremendous readability problems, the ratio of unique overloaded subprogram names divided by the total number of unique subprogram
names is multiplied by a factor of two. The supporting metric for this indicator is in "Measures and Metrics for Ada Specific Indicators" on page 135.

Parameters

The analysis on parameters in Chapter 2 reveals five factors relating to default parameters and name parameter notation. All five factors affect the same three attributes: complexity, readability, and well-defined interface. The resulting indicators from these five factors are enumerated in Figure 9 on page 67.

The measurement approaches for these indicators are similar in design except for the three indicators associated with "Use of Parameters with Name Notation." A discussion of the measurement approach for these three indicators follows the discussion of the four factors with similar measurement approaches. The measurement approaches relate either a use or abuse of default parameters and name parameter notation. For example, the measurement approach for the indicators associated with "Definition and Use of Default Parameters for Stable Values" is the number of subprogram calls where defaults were defined and used, relative to the number of subprogram calls where default values were defined. Generalized, the measurement approach is number of subprograms calls that exhibit the use (or abuse) relative to the number of subprograms that could exhibit the use (or abuse). The generalized measurement approach applied to each of the factors is provided in "Measures and Metrics for Ada Specific Indicators" on page 135.

All five of the factors relating to default parameters and name parameter notation have a commonality: the measurement approaches do not differ among a particular factor's attributes. The rationale for using the same measurement approach relies upon the justification of the linkages between the factor and the attributes. Recall from Chapter 2
1. Use of Default Parameters with Positional Notation
   - Complexity (-)
   - Readability (-)
   - Well-Defined Interface (-)

2. Definition and Use of Default Parameters for Stable Values
   - Complexity (+)
   - Readability (+)
   - Well-Defined Interface (+)

3. Use of Parameters with Name Notation
   - Complexity (+)
   - Readability (+)
   - Well-Defined Interface (+)

4. Mixing the Order of Parameter Lists
   - Complexity (-)
   - Readability (-)
   - Well-Defined Interface (-)

5. Use of Both Positional and Name Notation in a Single Subprogram Call
   - Complexity (-)
   - Readability (-)
   - Well-Defined Interface (-)

Figure 9. Property/Attribute Indicators for Ada Parameters
that the rationale for each attribute is closely tied to the other two attributes. Since the positive (negative) effect of the factor on the attributes is similar, so are the measurement approaches.

The factor, "use of parameters with name notation," utilizes a measurement approach different from the others. As noted above, the measurement approaches for this factor do not differ among attributes. The measurement approach for these three indicators is the average number of parameters from subprogram calls using name notation with more than five parameters. Name notation is included in Ada to improve calls with large numbers of parameters, hence the measurement approach only notes calls with more than five parameters [FAIRR85]. The final metric for these indicators scales the measure between ten and zero. As the number of parameters grows, the clarity of the call will decrease irrespective of the parameter notation used. Name notation simply allows more parameters to be used than positional notation without loss of clarity.

Packages

Packages are an important Ada component for achieving modularity and applying accepted software engineering principles. The importance of packages is also reflected in the number of factors, and eventual indicators, developed for packages. The analysis in Chapter 2 reveals five factors relating to the use of packages and identifies the impact (if any) of the attributes on each. Figure 10 on page 69 contains the formal property/attribute indicators for Ada packages.

The definition of declaration packages has a positive impact on both ease of change and cohesion. The measurement approaches for these two property/attribute indicators is quite
1. Definition of Declaration Packages
   - Cohesion (+)
   - Ease of Change (+)

2. Insufficient Decomposition of Declaration Packages
   - Ease of Change (-)

3. Definition of Packages that export Subprograms
   - Cohesion (+)
   - Ease of Change (+)
   - Well-Defined Interface (+)

4. Units that "with" Packages that export Subprograms
   - Readability (+)
   - Complexity (+)

5. Definition of Packages that are never "withed"
   - Complexity (-)

Figure 10. Property/Attribute Indicators for Ada Packages
different. Program units that "with" a declaration package actually reference a percentage
of the declarations. A high percentage (or utilization) is indicative of related items because
they are necessary and used in the same program unit. The measure for the cohesiveness
of a package is the average utilization of "withing" units. The impact on ease of change is
quite different. The measurement approach for this indicator is to determine the reduction
of scope (percentage of units that must be checked during maintenance) and weight the
reduction by the significance of the scope reduction (percentage of the lines of code that
must be checked).

Insufficient decomposition of declaration packages results in excessive maintenance
activity, hence, hindering ease of change. Declaration package utilization is a measure for
the degree that declarations in a package are utilized. A low package utilization implies that
a level of abstraction or decomposition is missing. Package utilization is also a measure for
the percentage of items that must be checked for possible changes that actually need to be
checked during maintenance. Temper this measure by the significance of the package (total
number of declarations in the package relative to the number of declarations in all
declaration packages) to obtain a measure for ease of change. The supporting metric for
this indicator and all package indicators can be found in "Property/Attribute Pairs for
Identified Indicators" on page 110.

The second significant type of packages are those that export subprograms. The definition
of subprogram packages, a factor relating to packages that export subprograms, induces a
desirable effect on the attributes of cohesion, ease of change, and well-defined interface.
Since the measurement approaches for the attributes of cohesion and ease of change are
similar to those used for the indicators under definition of declaration packages, the
measurement approach for the attribute well-defined interface is discussed. Another factor
relating to packages that export subprograms is units that "with" packages that export
subprograms. This factor aids the attributes of complexity and readability. The measurement approach for the attribute complexity will be discussed to illustrate this factor.

As previously noted, the measurement approaches for the definition of packages that export subprograms relative to cohesion and ease of change are similar to those under definition of declaration packages. The measurement approach for the property/attribute indicator definition of packages that export subprograms relative to well-defined interface is quite distinct. This measurement approach considers the important elements concerning package interfaces.

The number of objects being exported is an element in assessing the quality of a package interface. The key objects that will be factored in are variables and subprogram specifications. The clarity of the interface is related to the Hrair limit. From the Hrair limit [MILLG56], five subprograms is the optimal number to be exported. Due to the damaging effect of global variables, zero is the optimal number of variables in the package specification. The second element is global variable references external to the package. It is through these references that information flow occurs that is not documented in the package specification. Due to the major impact of these references, their weight will be twice that of other element. The final metric utilizes these measures to assess package interfaces. Again, see "Measures and Metrics for Ada Specific Indicators" on page 135 for the supporting metric.

Units that "with" packages that export subprograms receive beneficial effects on the attributes of complexity and readability. The readability measurement approach considers the number of distinct calls to subprograms defined in packages relative to the total number of distinct subprogram calls. The measurement approach for the attribute complexity is different in that a unit need only consider the code in the package specification to use the
entire package. The goal for package design is that the code in the package body need never be considered by the "withing" unit. The measurement approach for this indicator calculates a reduced work of a unit factor by dividing the total lines of code of a package specification by the total lines of code of a package body. The lower this ratio, the larger the reduction of work. The exact metric formulation can be found in "Measures and Metrics for Ada Specific Indicators" on page 135.

The final factor relating to packages is the definition of packages that are never "withed." This factor has a negative impact on complexity. The measure for this indicator is the number of packages that are never "withed" relative to the total number of packages. Intuitively, a high value for this measure implies additional complexity or work with a program. As can be seen by the number and variety of indicators for packages, packages are an extremely important component in the Ada language.

Generics

Generics are another important Ada component that provide significant support of software engineering principles. Through generics, levels of abstraction can be made that are not possible in traditional block-structured languages. Chapter 2 reveals four factors relating to the use of generics. Synthesizing the four factors and the attributes identified as having an impact produces the property/attribute indicators shown in Figure 11 on page 73.

The indicators relating to definition of parameterless generics and use of subprograms as generic parameters result in a negative impact. The two uses of generics are legitimate and necessary, but do have a negative impact on the code. Definition of generic units relates the positive impact on complexity and readability achieved by creating an additional layer of abstraction. The last indicator, multiple instantiations of generic units relative to ease of
1. Definition of Parameterless Generics
   - Complexity (-)

2. Use of Subprograms as Generic Parameters
   - Complexity (-)
   - Readability (-)

3. Definition of Generic units
   - Complexity (+)
   - Readability (+)

4. Instantiation of Generic Units
   - Ease of Change (+)

Figure 11. Property/Attribute Indicators for Ada Generics
change, relates the benefit to maintenance activities that generic units can provide. Each of these indicators is discussed in the following text.

The definition of parameterless generics adversely affects complexity. Multiple instantiations of a parameterless generic results in duplicate copies of the objects exported by the generic unit. The measure for the increased complexity is the number of defined parameterless generics relative to the total number of generics defined. See "Measures and Metrics for Ada Specific Indicators" on page 135 for the supporting metric and notes on the application of this metric.

The next factor, use of subprograms as generic parameters, results in the identification of two indicators relative to complexity and readability. The adverse effect of the two indicators stems from the same source, i.e. code with a formal parameter of a subprogram name. The measurement approaches for the use of subprograms as generic parameters relative to both complexity and readability are the same. The measure for these two indicators is the number of generic units that have subprograms as parameters relative to the total number of units defined. The metric and special cases for these indicators are in "Measures and Metrics for Ada Specific Indicators" on page 135.

The definition of generic units relative to complexity and readability represents a beneficial impact. Defining generic units reduces complexity by adding a layer of abstraction. The measurement approaches do not differ between the indicators for complexity and readability because the beneficial impact is the same on both attributes. The measure for these indicators is the ratio of subprograms defined within a generic unit relative to the total number of subprograms defined. The ratio is multiplied by a factor of two because it is intuitive that if half of the subprograms are defined within generic units, the program has much of its complexity reduced due to abstractions made through the definition of generic
units. "Measures and Metrics for Ada Specific Indicators" on page 135 contains the supporting metrics for these two indicators.

The final generic indicator is multiple instantiations of a generic unit relative to its positive impact on ease of change. One instantiation of a generic unit yields no benefit on ease of change. The benefit on ease of change with respect to generics occurs as the number of instantiations of a generic unit increases. Therefore, the measure for this indicator is the average number of instantiations over all defined generics. The supporting metric (see "Measures and Metrics for Ada Specific Indicators" on page 135) scales this measure to reflect the positive impact of multiple generic instantiations on ease of change.

Tasking

As evident from the quantity and quality of indicators relating to packages and generics, both are important Ada components from a software engineering point of view. Tasking is another important Ada component. A goal of the design of Ada's concurrency component (tasking) is to develop an efficient and easily understood concurrency model. Experts in the real-time field are currently debating the success of the venture [HASSJ90]. However, given the current state of Ada tasking, the evaluation and analysis of tasking in Chapter 2 reveals the property/attribute indicators shown in Figure 12 on page 76.

The first tasking indicator is the use of tasking relative to its negative impact on complexity. This indicator relates the negative impact that the use of any concurrency mechanism has on complexity. The complexity increases as the number of program units dealing with concurrency increases. The smallest Ada program unit is a subprogram. Tasking visibly impacts a subprogram through the tasking communication mechanism, rendezvous. A less desirable communication method, global variables, is assessed with two other indicators.
1. Use of Tasking
   • Complexity (-)
2. Definition of Tasks
   • Readability (-)
3. Use of Rendezvous with In Out or In and Out Parameters
   • Coupling (-)
4. Use of Unprotected Global Data Areas
   • Coupling (-)
   • Ease of Change (-)

Figure 12. Property/Attribute Indicators for Ada Tasking Constructs
Therefore, an intuitive measure for this indicator is the percentage of subprograms that have a rendezvous. This measurement approach and supporting metric for this indicator is in "Measures and Metrics for Ada Specific Indicators" on page 135.

The definition of tasks relative to the negative impact on readability is the next tasking indicator. Although this indicator is similar to the previous one, the two are distinctly different. The first indicator related the negative impact of concurrency on complexity. The definition of tasks relative to readability relates the potential adverse effect resulting from the definition of tasks. A task can be written so as to not have a negative impact on readability. In fact, the negative impact stems from the number of accept statements, or communicational entry points to a task. Intuitively, a large number of entry points is less readable than a few entry points. The measure for this indicator is the number of accept statements in the task body. The optimal value for this measure is three or less accept statements with an increase in the number of accept statements corresponding to a decrease in the measure for readability.

The third tasking indicator is use of rendezvous with In Out or In and Out Parameters relative to its negative impact on coupling. The optimal communication method for a rendezvous is one directional, In or Out only. Rendezvous with In Out or In and Out Parameters provide two way communication. An intuitive measure for this indicator is the number of entries defined with In Out or In and Out parameters relative to all defined entries. The supporting metric for this indicator can be found in "Measures and Metrics for Ada Specific Indicators" on page 135.

The final two tasking indicators relate to one factor, use of unprotected global data areas which negatively impacts both coupling and ease of change. Units are coupled through the global data areas and ease of change is impacted by the non-deterministic nature of Ada
tasks. Because the negative impact of this factor affects both attributes similarly, the measurement approaches for the two indicators are the same. The measure for the two indicators relating to use of unprotected global data areas is the average number of units potentially concurrently coupled by a global data area.

Exception Handling

The analysis of exception handling reveals two significant factors: (1) the use of exception handlers that have a positive effect on complexity and (2) relying on upper level procedures to handle user raised exceptions which have a negative effect on coupling. This leads to the two indicators:

- Use of Exception Handlers
  - Complexity (+) and
- Relying on Upper Level Subprograms to Handle User Raised Exceptions
  - Coupling (-).

The use of exception handlers separates the algorithm from the error handling code, thus benefiting complexity. The measurement approach for this indicator assesses the number of subprograms that benefit from exception handlers. The measure for this indicator is the number of subprograms with an exception handling section relative to the total number of subprograms. This number is weighted by a factor of two because exception handlers in half of the subprograms are extremely beneficial.

The second exception handling indicator, relying on upper level subprograms to handle user raised exceptions relative to coupling, takes a dramatically different measurement approach than any previously discussed indicators. Relying on upper level subprograms to
handle user raised exceptions needlessly couples the subprograms together. Therefore, the measurement approach for this indicator is the number of subprograms that raise errors, but do not handle them. This measure, unlike many previous measures, is not made relative to the total number of subprograms. The presence of this indicator is inherently detrimental.

Summary

The discussion of measures for the unit level control structure exception handlers completes the development of metrics for Ada components within the OPA framework. As described in the chapter, the measurement approaches to the metrics are quite varied and are extremely dependent on the property/attribute pair being measured. One goal of the metric set is automatability to ensure that the metrics can be objectively applied using an automated assessment tool. The metrics described in this chapter meet this criteria of automatability and objectiveness. Chapter 4 discusses future and ongoing work related to developing an automated analyzer utilizing the metrics described in this thesis.
Chapter 4

Analysis and Conclusion

The study of Ada relative to the OPA framework through the seven step metric development process produced a set of intuitive and simple metrics that assess the impact of Ada code properties on the software engineering attributes. However, the metrics developed by Dandekar [DANDA87] need to be reviewed to ensure consistency among the entire set of metrics. Included in this chapter is a discussion of the review of existing OPA framework metrics and a summary of the entire set of metrics for the OPA framework. In addition, a current research effort is progressing toward the automation of the set of OPA framework metrics. A summary of the current results and future goals is described.

Review of Existing OPA Framework Metrics

The research project on which this thesis is based produced numerous lessons with respect to metric development and software quality indicators. A review of the existing OPA framework metrics was necessary to apply some of these lessons and to make the existing metrics consistent with the metrics developed for crucial Ada components. During the
original OPA research effort, the metric value of five was not emphasized as a median value, indicative of no discernable impact. As a result of this emphasis in the current research effort, the scale was expanded to be symmetric above and below the value of five. The adjustment to the scale required several of the existing metrics to be tempered to fit the new scale.

Additionally, several of the metrics assessed the absence of an undesirable product property as having the maximum beneficial impact on an attribute. For example, does the absence of global variables indicate low coupling? No, the absence of global variables indicates no undesirable impact on coupling by global variables, not a positive impact on coupling. Hence, the appropriate metric value for the property of no global variables relative to the attribute of coupling is a five.

Finally, a close scrutiny of a few metrics indicated that the property/attribute indicator relationship was not accurately reflected by the supporting metric. Some cases simply required a clarification of the metric's representation and some metrics necessitated a redesign of the measurement approach and supporting metric. One metric that was changed relates the length of subprograms relative to the attribute of complexity. The original metric used the average subprogram length for the entire program. This metric yields one value that may be normal despite large deviations among the subprograms. The revised metric compares each subprogram length to the optimal subprogram length. The end results are a thoroughly studied group of metrics that are consistent with and complementary to the metrics developed for crucial Ada components. See "Metrics for Ada Constructs by Software Engineering Attribute" on page 175 for the entire set of metrics.
Summary of the Entire Metric Set

The set of code metrics developed by Dandekar [DANDA87] for the OPA framework contains 31 metrics. To understand the magnitude of additional capabilities and components provided by Ada, note that the crucial components result in 36 additional metrics. The final metric set for assessing the quality of Ada-based products contains 67 metrics.

The break-down of the number of metrics proves quite interesting. Figure 13 on page 83 shows that some attributes simply cannot be measured at this time. One attribute, traceability, regards the ability to trace a software component from its inception to its current status. Code indicators simply do not support such a measure, but hopefully documentation and process indicators can capture this deficit.

The design of the metrics typically falls into one of three forms: (1) values increase from five to ten, (2) values decrease from five to zero, and (3) values decrease from ten to zero. When metric values increase from five to ten, the metric value increases as product properties indicate a positive effect on the attribute; however, only positive effects result from the product property. Metrics with value ranges from five to zero are the opposite. Product properties indicate a negative effect on the attribute only. Several examples of metrics with the previous two forms can be found in "Measures and Metrics for Ada Specific Indicators" on page 135. Metrics that range from ten to zero are quite different. The product properties that are indicative of the attribute can have positive effects, but if overdone, have negative effects. An example is the use of name parameter notation relative to the attribute complexity. Name parameter notation is beneficial, but as the number of parameters increases, the benefits decrease until additional parameters produce a negative impact.
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Metrics for Crucial Ada Components</th>
<th>Original OPA Framework Code Metrics</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cohesion</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Complexity</td>
<td>13</td>
<td>9</td>
<td>22</td>
</tr>
<tr>
<td>Coupling</td>
<td>3</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>Early Error Detection</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Ease of Change</td>
<td>5</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>Readability</td>
<td>10</td>
<td>7</td>
<td>17</td>
</tr>
<tr>
<td>Traceability</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Visibility of Behavior</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Well-Defined Interface</td>
<td>2</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>36</strong></td>
<td><strong>31</strong></td>
<td><strong>67</strong></td>
</tr>
</tbody>
</table>

*Figure 13. Number of OPA Framework Metrics by Attribute*
Automation of Metrics

As previously noted in the Chapter 1, the manual assessment of software is both impractical and error prone. Therefore, automating the set of metrics for Ada constructs developed within the OPA framework is of paramount importance. The development of an automated analyzer, utilizing the set of metrics described in this thesis, is an ongoing two-part process. The design of an analyzer and an associated product is discussed below.

Analyzer

Current research plans include developing an automated analyzer. As Figure 14 on page 85 shows, the automated analyzer processes Ada code, documentation, and process data to create a "raw" data file. The portion of the analyzer that creates the "raw" data file is known as the "front-end" while the report generator is known as the "back-end." The design of the "raw" data file is to make it as language, document, and process independent as possible. Obviously all dependencies cannot be removed, but as the independence of the "raw" data file increases, so does the ability to use the report generator in environments other than originally planned. The "back-end" or report generator, represented by the bottom line of Figure 14 on page 85, processes the "raw" data to produce reports describing the achievement of objectives, use of principles, and presence of attributes. Since the work in this thesis primarily impacts the design of the Ada code "front-end," the design is discussed further in the following paragraph.

To extract the necessary data, the "front-end" will accept Ada code and use both syntactic and semantic analysis to generate a "raw" data file. The contents of the "raw" data file will be used to support metric computations. Examples of "raw" data include the number of global variables referenced by a subprogram, total lines of code, average lines of code per
Figure 14. The Automatic Analyzer
subprogram, and the average number of parameters passed per subprogram call. The format that the "raw" data is stored is directly proportional to the complexity of the "back-end." An efficient well-designed storage format for the intermediate "raw" data aids the design of the "back-end." Because of this importance, a discussion of the format and design rationale follows.

Intermediate Files

The output of the analyzer's "front-end," the input for the "back-end," is called the intermediate file. The design of the intermediate file reflects the structure of the metrics. For example, for each subprogram package, the necessary data items include total lines of code, number of global variable references external to the package, and number of subprogram specifications in the package specification. A logical design is to have a data file for all packages with a data section for each package and a data section for data relevant to all packages. Extended, files of similar design are created for subprograms and tasks. The fourth data file contains general information pertinent to the entire program. Since a generic is a generic subprogram or a generic package, information regarding generics is placed in the file of the generic unit, not a separate generic file. The four intermediate files are a logical storage unit for information needed by the "back-end" of the analyzer.

Future Work

One item of future work related to this thesis has already been discussed, namely the analyzer. However, another work of dire importance is to gain user acceptance of the OPA framework and related metrics for software quality assessment of Ada-based products. The necessary work effort is a validation of the OPA assessment framework by applying it to a current Ada-based development project. Validation of the OPA framework is currently
being addressed by a research project with a two-year planning horizon.

The goals of the current research project include integrating management [ROSSC88], process [HUMPW87], and documentation [STEVK88] indicators with the code indicators and related metrics. To validate the research results, a project will be monitored during development. The results of the analyzer can then be assessed relative to the actual product for validity. An important product of this study is the feedback concerning metric values and code properties. Any discrepancies can be studied to determine if adjustments need to be made to the metric set.

Conclusion

The author believes that the Objectives, Principles, Attributes framework is an intuitive approach for the software quality assessment of Ada-based products. While working toward the goal of automated assessment of Ada-based products through the Objectives, Principles, Attributes framework, the author has made numerous contributions to software engineering in two areas, indicators and the automated analyzer. The contributions are listed below.

- Because of the unstructured nature of the research problem, a seven step procedure for developing metrics for Ada relative to the OPA framework was developed. The procedure, which can be generalized for any language, focuses research efforts at each stage of the metric development process from the identification of crucial language constructs to the development of measures and supporting metrics.
- One product resulting from the application of the metric development procedure is 36 property/attribute indicators for crucial Ada components. The indicators represent Ada components having a significant impact on the software engineering attributes.
The distribution of new indicators implies that packages, generics, and tasks are among the most significant of the crucial Ada components.

- For each of the 36 property/attribute indicators, a supporting measure and metric were developed. Each of the metrics is automatable which facilitates the development of an analyzer for assessing Ada-based products.

- The "front-end" of the analyzer, which is a language specific tool using syntactic and semantic analysis to parse code data, was designed.

- The output of the "front-end" is a series of intermediate files for input to the analyzer's "back-end" (also known as the report generator). Because the analyzer is expected to be refined throughout the research project, the intermediate files are designed to facilitate modifications to the analyzer while providing efficient output and input data storage to the analyzer's "front-end" and "back-end."

Summarizing the contributions listed above, the crucial components of Ada that are not found in Pascal were integrated into OPA framework through the seven step procedure. The resulting measures and supporting metrics are totally automatable and an automated analyzer is currently being developed. Evaluating Ada code for the achievement of objectives, use of principles, and presence of attributes will be important to both the developers of Ada-based products and the purchasers of Ada-based products. Instrumental to this assessment will be the completion of the Ada analyzer to facilitate the automated assessment process.
Bibliography


[ZUSEH8?] Zuse, Horst and Peter Bollmann, "Software Metrics," ????, 198?:
Appendix A

Package Implementation of Abstract Data Type

Queue Example
--QUEUE Example
--Purpose: Contains the routines and type declarations
-- to implement the abstract data type QUEUE
-
--Implementation of type is hidden from user via the use of
-- limited private.
-
--User may declare instances of type QUEUE only.
-
Package QUEUE is

subtype ITEMTYPE is STRING(1..80);

procedure CREATE_Q ( Q : out QUEUE );
--Create an empty queue Q.

function ISEMPTY ( Q : QUEUE ) -- function parameters
assumed "in"
return BOOLEAN;
--Determine if queue Q is empty.

procedure ADD ( Q : in out QUEUE;
               E : in ITEMTYPE);
--Add item E to the queue Q.

procedure REMOVE ( Q : in out QUEUE;
                  E : out ITEMTYPE;
                  SUCCESS : out BOOLEAN);
--Remove item E from the queue Q, the item that was added
earliest.

private
type NODE;
type NODEPTR is access NODE;
type NODE is
record
  ITEM : ITEMTYPE;
  NEXT : NODEPTR;
end record;


Private
package implementation of abstract data type queue example
end QUEUE;

package body QUEUE is

procedure CREATE_Q ( Q : out QUEUE ) is
--Create an empty queue Q.
begnin
Q := null;
end CREATE_Q;

function ISEMMPTY ( Q : QUEUE ) -- function parameters
assumed "in"
return BOOLEAN is
--Determine if queue Q is empty.
begnin
return (Q=null);
end ISEMMPTY;

procedure ADD ( Q : in out QUEUE;
E : in ITEMTYPE) is
--Add item E to the queue Q.
P : NODEPTR;

begnin
--Create a new node.
P := new NODE;
P.ITEM := E;

--Insert the new node.
if Q = null then
P.NEXT := P;
else
P.NEXT := Q.NEXT;
Q.NEXT := P;
end if;

Q := P;
end ADD;
procedure REMOVE (Q: in out QUEUE;
E: out ITEMTYPE;
SUCCESS: out BOOLEAN) is
--Remove item E from the queue Q, the item that was added earliest.
FRONT: NODEPTR;

begin

if Q = null then
  SUCCESS := FALSE;
else
  FRONT := Q.NEXT;
  E := FRONT.ITEM;
  if FRONT = Q then
    Q := null;
  else
    Q.NEXT := FRONT.NEXT;
  end if;
  SUCCESS := TRUE;
end if;
end REMOVE;

end QUEUE;
Appendix B

Generic Package Implementation of Abstract

Data Type Queue Example
--QUEUE Example
--Purpose: Contains the routines and type declarations
-- to implement the abstract data type QUEUE
-
--Implementation of type is hidden from user via the use of
-- limited private.
-
--User may declare instances of type QUEUE only.
-
--Generic: This generic package allows the user to instantiate
-- a queue of any item type.

generic
type ITEMTYPE is private;

Package QUEUE is

type QUEUE is limited private;

procedure CREATE_Q ( Q : out QUEUE );
--Create an empty queue Q.

function ISEMPY ( Q : QUEUE ) -- function parameters
assumed "in"
return BOOLEAN;
--Determine if queue Q is empty.

procedure ADD ( Q : in out QUEUE;
E : in ITEMTYPE);
--Add item E to the queue Q.

procedure REMOVE ( Q : in out QUEUE;
E : out ITEMTYPE;
SUCCESS : out BOOLEAN);
--Remove item E from the queue Q, the item that was added
earliest.

private
type NODE;
type NODEPTR is access NODE;
type NODE is
record
ITEM : ITEMTYPE;
NEXT : NODEPTR;
   end record;

type QUEUE is new NODEPTR;

end QUEUE;

package body QUEUE is

procedure CREATE_Q ( Q : out QUEUE ) is
   --Create an empty queue Q.
begin
   Q := null;
end CREATE_Q;

function ISEMPY ( Q : QUEUE ) return BOOLEAN is
   -- function parameters assumed "in"
   --Determine if queue Q is empty.
begin
   return (Q=null);
end ISEMPY;

procedure ADD ( Q : in out QUEUE;
                E : in ITEMTYPE ) is
   --Add item E to the queue Q.
P : NODEPTR;

begin
   --Create a new node.
P := new NODE;
P.ITEM := E;

   --Insert the new node.
   if Q = null then
      P.NEXT := P;
   else
      P.NEXT := Q.NEXT;
      Q.NEXT := P;
   end if;
Q := P;

end ADD;

procedure REMOVE ( Q : in out QUEUE;
        E : out ITEMTYPE;
        SUCCESS : out BOOLEAN ) is
--Remove item E from the queue Q, the item that was added
--earliest.

    FRONT : NODEPTR;

begin

    if Q = null then
        SUCCESS := FALSE;
    else
        FRONT := Q.NEXT;
        E := FRONT.ITEM;
        if FRONT = Q then
            Q := null;
        else
            Q.NEXT := FRONT.NEXT;
        end if;
        SUCCESS := TRUE;
    end if;

end REMOVE;

end QUEUE;

--NOTES:
--To declare an instance of this generic package for integers,
--use the following statement.
package QUEUE_INTEGER is new QUEUE(INTEGER);
Appendix C

Tasking Demonstration to Find Prime Numbers
procedure Primes is --The main program

    task Feeder; --Task to feed the pipe with values

    type Checker; --Task template allows create of multiple

        --Tasks

    type Checker_Ptr is access Checker;

    task type Checker is

        entry Who_Am_I (My_Prime : in Positive);

        entry Check_It Value_to_Check : in Positive);

    end Checker;

    procedure Make_New_Checker (A_Prime_Number : in Positive;

        New_Checker : out Checker_Ptr);

        --This procedure provides the ability to create new tasks.

    Front : Checker_Ptr; --This is the front of the "pipe".

    task body Feeder is separate;

    task body Checker is separate;

    procedure Make_New_Checker (A_Prime_Number : in Positive;

        New_Checker : out Checker_Ptr)

        is separate;

    begin
        null;
    end Primes;

-----------------------------

with Text_IO;
separate (Primes)
procedure Make_New_Checker (A_Prime_Number : in Positive;

    New Checker : out Checker_Ptr) is

    Result : Checker_Ptr;

begin
begin

Tasking Demonstration to Find Prime Numbers
Result := new Checker; --Make a new prime # task.

exception
when Storage_Error =>
    Text_IO.Put_line (" Not enough room to make new
tasks.");
    raise;
end;

Result.Who_Am_I ( A_Prime_Number); --Tell the task its
prime #.

--Allow the task to be used in the "pipe".
New_Checker := Result;

--Display the prime number.
Text_IO.Put_Line ( Integer'Image ( A_Prime_Number ));

end Make_New_Checker;
-------------------------------------------
-------------------------------------------
with Text_IO;
separate(Primes)
task body Feeder is
    Upper_Limit : Positive;
    package Integer_IO is new Text_IO.Integer_IO(Integer);
begin

    Text_IO.Put("Upper limit for primes?");
    Integer_IO.Get(Upper_Limit);

    --Generate the first prime #:
    Make_New_Checker(2,Front);

    --Feed the "pipe".
    for Counter in 3 .. Upper_Limit loop
        Front.Check_It(Counter);
    end loop;

end Feeder;
-------------------------------------------
-------------------------------------------
separate (Primes)
task body Checker is
    My_Prime,
    Value_to_Check : Positive;
    Next_Checker : Checker_Ptr;
    Prime : Boolean;
begin

    accept Who_Am_I ( My_Prime : in Positive ) do
        Checker.My_Prime := My_Prime;
    end Who_Am_I;

    loop
        select
            accept Check_Ht (Value_to_Check : in Positive) do
                Checker.Value_to_Check := Value_to_Check;
            end Check_Ht;
            or
                terminate;
        end select;
        Prime := Value_to_Check mod My_Prime /= 0;

        if Prime then
            if Next_Check /= null then
                -- It's not divisible by my number, pass the value on.
                Next_Checker.Check_Ht (Value_to_Check);
            else
                -- It really is prime.
                Make_New_Checker (Value_to_Check, Next_Checker);
            end if;
        end if;
    end loop;
end Checker;

Tasking Demonstration to Find Prime Numbers 106
Appendix D

Examples of Methods of Using Ada Exception Handlers
Examples are from [BCOCG83] pp. 272-276.

Abandon the execution of the unit
No error handler included in module, error is propagated to
the calling module

Try the operation again

type RESPONSE is (UP, DOWN, LEFT, RIGHT);
USER_REQUEST : RESPONSE;

loop
  begin
    PUT(">");  
    GET(USER_RESPONSE);  
    exit;
  exception
    when DATA_ERROR =>
      PUT_LINE("Invalid response; enter only UP, DOWN, LEFT,
or RIGHT");
      end;
  end loop;

Use an alternative approach

begin
  SEND_MESSAGE_TO_PATH_1 ( CRITICAL_MESSAGE );
exception
  when TASKING_ERROR =>
    SEND_MESSAGE_TO_PATH_2 ( CRITICAL_MESSAGE );
end;
-- Repair the cause of the error
procedure MOVE_RUDDER ( AMOUNT : in out INTEGER ) is
  RUDDER_STRESSED : exception;
begin
  -- send command to rudder servomechanism here
exception
  when RUDDER_STRESSED =>
    AMOUNT := AMOUNT / 2;
    if AMOUNT /= 0 then
      MOVE_RUDDER ( AMOUNT );
    else
      raise;
    end if;
end MOVE_RUDDER;
Appendix E

Property/Attribute Pairs for Identified Indicators
Use of Record Discriminants

Affect:

- Complexity (+)

- Early Error Detection (+)

Note:

pp471/[PRATT84] Ada record discriminants allow data with dynamic requirements to be stored easily, thus reducing complexity. (Complexity)

Justification:

Discriminants aid early error detection by encouraging abstractions to be made earlier in the life cycle, thereby exposing potential weaknesses before implementation. (Early Error Detection)
Use of Block Statements

Affect:

- Cohesion (+)
- Complexity (+)
- Readability (+)

Notes:

pp74/[WATTD87] "It is undesirable to group all declarations together at the head of the program, as we have done up to now: this conflicts with the principle of localization. One way to avoid this is the block." {Cohesion}

Justification:

Block statements are an additional structuring mechanism provided by Ada. The use of this structuring mechanism will reduce program complexity and improve code readability by grouping code into smaller, easily understood units. {Complexity, Readability}
Use of Default Parameters with Positional Notation

Affect:

- Complexity (-)
- Readability (-)

Notes:

pp102/[WICHB84a] Understanding and programming with defaults is difficult without name parameter notation. {Complexity, Readability}

pp57/[HARDW76] Calls using defaults without name parameter notation are less readable than those name parameter notation. {Readability}
Definition and Use of Default Parameters for Stable Values

Affect:

- Complexity (+)
- Readability (+)

Notes:

pp139/BOOOG84] Default parameters are "most useful when a subprogram has parameters whose actual values do not change over most calls." This aids the readability of the program. {Readability}

pp102/WICH84a] "But with readability a keygoal of Ada, key word parameters, with defaults, help significantly." The use of defaults for stable values also reduces the overall complexity of the program. {Complexity, Readability}
Definition of Default Parameters for Stable Values

Affect:

- Well-Defined Interface (+)

Notes:

pp139/[BOOCG84] Default parameters are "most useful when a subprogram has parameters whose actual values do not change over most calls." Extra information is conveyed about the module's interface. (Well-Defined Interface)
Use of Parameters with Name Notation

Affect:

- Complexity (+)
- Readability (+)

Notes:

pp139/[BOOCG84] Named parameter association can be used to improve the readability of subprogram calls and also remove ambiguity about the subprogram's "actual" interface. {Readability}

pp102/[WICHB84a] The readability of Ada programs is improved by using keyword parameters. Key word parameters also reduce the overall program complexity. {Complexity, Readability}

pp57/[HARDW76] Subprogram calls are inherently more readable when keyword parameters are used. {Readability}
Mixing the Order of Parameter Lists

Affects:

- Complexity (-)
- Readability (-)

Justification:

Using name parameter to mix the order of parameter lists causes two major problems. First, mixing the order of the parameter lists causes confusion and additional work by forcing the programmer to understand multiple formats for one subprogram call. This hinders both readability and complexity. (Complexity, Readability) Second, mixing the order of the parameter lists impacts the clarity of the subprogram interface.
Use of Both Positional and Name Notation in a Single Subprogram Call

Affects:

- Complexity (-)
- Readability (-)

Justification:

Using both positional and name notation in one subprogram call potentially results in two major problems. First, deciphering the two notations in one call makes the programming task more difficult and increases the overall complexity of the program. {Complexity} Second, use of both parameter notations forces the reader to spend more time deciphering the call, reducing the overall readability of the program. {Readability}
Overloading of Subprogram Names

Affects:

- Readability (-)

Notes:

pp134/[BOOCG83] "The unrestricted use of overloading will decrease the understandability of a program." [Readability]

pp98/[GHEZC82] "Excessive use of overloading can generate programs that are hard to understand, because a unique name denotes completely different entities." [Readability]

pp337/[GHEZC82] "Overloading combined with the scope rules of the language can easily make programs difficult to read." [Readability]
Definition of Declaration Packages

Affect:

- Cohesion (+)

- Ease of Change (+)

Notes:

pp193/[BOOCG83] "One of the simplest uses of packages is for the logical grouping of objects and types. This application benefits maintainability by factoring out common data, objects, and types and placing their definition in one location." (Cohesion, Ease of Change)

pp4-53/[ROSSD86] The use of data structure design methodology to group similar declarations together as one cohesive unit is particularly good. This also improves the ease of change. (Cohesion, Ease of Change)
Insufficient Decomposition of Declaration Packages

Affect:

- Ease of Change (-)

Notes:

pp.194/[BOOC83] Declaration packages should be kept relatively small. A large package should be decomposed at least one more level. This will aid ease of change by decreasing unnecessary recompilation. (Ease of Change)
Definition of Packages that Export Subprograms

Affect:

- Well-Defined Interface (+)
- Ease of Change (+)
- Cohesion (+)

Notes:

pp195/[BOOCG83] Grouping related subprograms together provides a tight implementation independent interface while enhancing ease of change. (Ease of Change, Well-Defined Interface)

pp121/[YOURE78] A module can be composed of logically related functions grouped together in a cohesive unit. (Cohesion)

pp616/[GANNJ86] The modularity achieved by using packages improves ease of change by localizing effects. (Ease of Change)
Units which "with" Packages that Export Subprograms

Affects:

- Complexity (+)

- Readability (+)

Notes:

pp195/[BOOCG83] Since the user of the package need never see the body, abstractions have been made that improve code readability and reduce the complexity of the code. {Complexity, Readability}

pp616/[GANNJ86] The modularity achieved by using packages reduces the complexity of the code. {Complexity}
Definition of Packages that are Never "withed"

Affect:

- Complexity (-)

Justification:

Defining packages that are never "withed" adds to the amount of code in the program without adding any additional functionality. This adds to the work associated with understanding the program, or the program complexity.

{Complexity}
Definition of Parameterless Generics

Affect:

- Complexity (-)

Note:

pp346/WATTD87] Definition of parameterless generics will lead to multiple packages with no distinction between them. Non-differing duplications causes confusion and hence, increases complexity. (Complexity)
Use of Subprograms as Generic Parameters

Affect:

- Complexity (-)
- Readability (-)

Note:

pp215/[BOOCG83] Some forms of subprogram parameters reduce clarity. 

{Complexity, Readability}
Definition of Generic Units

Affects:

- Complexity (+)
- Readability (+)

Note:

pp360/[WATT87] The generic is a powerful tool in providing levels of abstraction. The abstraction allows the use of generics without detailed knowledge of their implementation. This aids both complexity and readability. {Complexity, Readability}

pp217/[BOO83] "Generic program units are powerful tools that can assist the programmer in managing the complexity of his or her software solution."

{Complexity}
Multiple Instantiations of Generic Units

Affect:

- Ease of Change (+)

Note:

pp345/[WATTD87] Rather than manually duplicating units, use the Ada generic facility to define a generic unit and then make multiple instantiations of the generic to aid ease of change. {Ease of Change}
Use of Tasking

Affects:

- Complexity (-)

Note:

pp109/[DEITH84] "Concurrent programs are normally more difficult to write, debug, and prove correct than sequential programs." (Complexity)
Definition of Tasks

Affect:

- Readability (-)

Note:

pp109/[DEIT84] "Concurrent programs are normally more difficult to write, debug, and prove correct than sequential programs." This hampers the readability of the program. {Readability}
Use of Rendezvous with In Out or In and Out Parameters

Affects:

- Coupling (-)

Note:

pp4-47/[NIELK86] The tightest coupling during a rendezvous occurs when an entry with "in out" or "in" and "out" parameters is called. {Coupling}
Use of Unprotected Global Data Areas

Affect:

- Coupling (-)

- Ease of Change (-)

Note:

pp78/[DEITH84] Shared data must be protected by some form of exclusion mechanism to prevent erroneous results. By the nature of shared data and concurrency, repeating the conditions that led to the erroneous results may be difficult, hindering ease of change. {Ease of Change}

pp98/[YOURE87] When two or more modules interact with a common data environment, those modules are said to be common-environment coupled. Each pair of modules that interacts with the common environment is coupled - regardless of the direction of communication or the form of reference. {Coupling}

pp111/[PAGEM80] A change to a global data area requires that each module within the scope of the data area be checked for possible impacts. {Ease of Change}
Use of Exception Handling

Affects:

- Complexity (+)

Note:

pp684/[GOODJ75] Use of exception handlers makes the programming task easier. This results in a reduced program complexity. {Complexity}

pp154/[LAWLP88] Exception handling makes tracking errors during maintenance easier, there by reducing complexity. {Complexity}
Relying on Upper Level Subprograms to Handle User Raised Exceptions

Affects:

- Coupling (-)

Note:

pp223/[GILPG86] Propagating errors outside of the block that they occurred couples the units together via flow of control. (Coupling)

pp269/[BOOCH83] While designing exception handlers, it is best to capture and handle at error at the same level to prevent coupling. (Coupling)
Appendix F

Measures and Metrics for Ada Specific Indicators
<table>
<thead>
<tr>
<th>Property:</th>
<th>Use of Record Discriminants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute:</td>
<td>Complexity (+)</td>
</tr>
<tr>
<td>Rationale:</td>
<td>A factor in reducing complexity is the ability to use abstractions. Record discriminants encourage the accurate modeling of data types and allow more succinct abstractions to be made than are possible using &quot;conventional&quot; records.</td>
</tr>
<tr>
<td>Measurement</td>
<td>The measure for this indicator is the number of discriminant types defined relative to the number of user defined types defined. (Per Unit)</td>
</tr>
<tr>
<td>Approach:</td>
<td></td>
</tr>
<tr>
<td>Metric:</td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{Ratio of Types} = \frac{\text{Number of Discriminant Types Defined}}{\text{Total Number of User Defined Types Defined}}
\]

Proposed Metric = 5 + 5 * Ratio of Types
Property: Use of Record Discriminants
Attribute: Early Error Detection (+)
Rationale: Implementing record discriminants requires detailed data-structures earlier in the life cycle, exposing weaknesses in previous documents.
Measurement Approach: The measure for this indicator is the number of discriminant types relative to all user defined types. (Per Unit)
Metric:

\[
\frac{\text{Number of Discriminant Types Defined}}{\text{Total Number of User Defined Types Defined}}
\]

Proposed Metric = 5 + 5 \times \text{Ratio of Types}
Property: Use of Block Statements

Attribute: Cohesion (+)

Rationale: Block statements are an additional control structure provided by Ada. By binding code into control structures (block statements are one such control structure), the code cohesion is improved because such bindings usually imply that statements are functionally related.

Measurement Approach: The measurement approach for this indicator utilizes a measure across the entire program as well as measures at the subprogram level. All subprograms are analyzed to determine an average lines of code per blocking structure across the entire program. From this measure, the expected number of blocking structures for a particular subprogram can be computed. The final measure, computed on a per subprogram basis, uses two ratios: (1) the expected number of blocking structures divided by the actual number of blocking structures tempered by (2) the total lines of code enclosed by blocking structures divided by the total lines of code of the subprogram.

Metric: From Dandekar, 1987

Note: BS refers to control structures and BS at level 0 refers to any blocking structure not nested within another blocking structure.

\[
\text{Ave Loc per BS} = \frac{\text{TOTAL LINES OF SOURCE CODE FOR ALL SUBPROGRAMS}}{\text{TOTAL BS AT LEVEL 0 FOR ALL SUBPROGRAMS}}
\]

\[
\text{Expected BS at Level 0} = \frac{\text{TLOC of the Subprogram}}{\text{Ave TLOC per BS}}
\]

\[
\text{Proposed Metric} = \frac{\text{TLOC Enclosed by BS at Level 0}}{\text{TLOC of the Subprogram}} \times 10 \times \frac{\text{Expected BS at Level 0}}{\text{Number of BS at Level 0}}
\]
Property: Use of Block Statements

Attribute: Complexity (+)

Rationale: Block statements are a control structure provided by Ada. By using block statements to break code into smaller units, the complexity of the code is reduced.

Measurement Approach: The measurement approach for this indicator utilizes a measure across the entire program as well as measures at the subprogram level. All subprograms are analyzed to determine an average lines of code per blocking structure across the entire program. From this measure, the expected number of blocking structures for a particular subprogram can be computed. The final measure, computed on a per subprogram basis, uses two ratios: (1) the expected number of blocking structures divided by the actual number of blocking structures tempered by (2) the total lines of code enclosed by blocking structures divided by the total lines of code of the subprogram.

Metric: From Dandekar, 1987

Note: BS refers to control structures and BS at level 0 refers to any blocking structure not nested within another blocking structure.

\[
\text{Ave Loc per BS} = \frac{\text{TOTAL LINES OF SOURCE CODE FOR ALL SUBPROGRAMS}}{\text{TOTAL BS AT LEVEL 0 FOR ALL SUBPROGRAMS}}
\]

\[
\text{Expected BS at Level 0} = \frac{\text{TLOC of the Subprogram}}{\text{Ave TLOC per BS}}
\]

\[
\text{Proposed Metric} = \frac{\text{TLOC Enclosed by BS at Level 0}}{\text{TLOC of the Subprogram}} \times 10 \times \frac{\text{Expected BS at Level 0}}{\text{Number of BS at Level 0}}
\]
Property: Use of Block Statements

Attribute: Readability (+)

Rationale: Block statements are an additional control structure provided in Ada. Block statements aid readability by partitioning code into smaller related units. This partitioning results in smaller blocks of code that need to be considered as a unit by the reader, hence improving the understandability and readability of the code.

Measurement Approach: The measurement approach for this indicator utilizes a measure across the entire program as well as measures at the subprogram level. All subprograms are analyzed to determine an average lines of code per blocking structure across the entire program. From this measure, the expected number of blocking structures for a particular subprogram can be computed. The final measure, computed on a per subprogram basis, uses two ratios: (1) the expected number of blocking structures divided by the actual number of blocking structures tempered by (2) the total lines of code enclosed by blocking structures divided by the total lines of code of the subprogram.

Metric: From Dandekar, 1987

Note: BS refers to control structures and BS at level 0 refers to any blocking structure not nested within another blocking structure.

\[
\text{Ave Loc per BS} = \frac{\text{TOTAL LINES OF SOURCE CODE FOR ALL SUBPROGRAMS}}{\text{TOTAL BS AT LEVEL 0 FOR ALL SUBPROGRAMS}}
\]

\[
\text{Expected BS at Level 0} = \frac{\text{TLOC of the Subprogram}}{\text{Ave TLOC per BS}}
\]

\[
\text{Proposed Metric} = \frac{\text{TLOC Enclosed by BS at Level 0}}{\text{TLOC of the Subprogram}} \times 10 \times \frac{\text{Expected BS at Level 0}}{\text{Number of BS at Level 0}}
\]
Property: Use of Both Default Parameters and Positional Notation

Attribute: Complexity (-)

Rationale: A parameter list in positional notation that has utilized a default parameter gives the reader the impression that something has been left out. This adds to confusion, and hence, complexity.

Measurement Approach: The measure for this indicator is the number of omitted parameters relative to the total number of parameters possible in all calls using positional notation. (Per Subprogram)

Metric:

\[
\text{Ratio of Omitted Parameters} = \frac{\text{Number of omitted parameters in all calls using positional notation}}{\text{Total number of parameters possible in all calls using positional notation}}
\]

Proposed Metric = 5 - 5 * Ratio of Omitted Parameters
**Property:** Use of Both Default Parameters and Positional Notation

**Attribute:** Readability (-)

**Rationale:** It is more difficult to read a program containing calls utilizing positional notation that have omitted parameters. It appears to the reader that something has been left out.

**Measurement Approach:** The measure for this indicator is the number of omitted parameters relative to the total number of parameters possible in all calls using positional notation. (Per Subprogram)

**Metric:**

<table>
<thead>
<tr>
<th>Ratio of Omitted Parameters</th>
<th>=</th>
<th>Number of omitted parameters in all calls using positional notation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Total number of parameters possible in all calls using positional notation</td>
</tr>
</tbody>
</table>

| Proposed Metric | = | 5 - 5 * Ratio of Omitted Parameters |
Property: Definition and Use of Default Parameters for Stable Values

Attribute: Complexity (+)

Rationale: Parameter values that remain relatively stable for the entire program can be given default values. By defining and using a default value for a parameter, there is less information for the programmer to keep up with, hence, reducing complexity.

Measurement Approach: The measure for this indicator is the number of subprogram calls where defaults are defined and used, relative to the number of subprogram calls where default values are possible. (Per Subprogram)

Metric:

\[
\text{Ratio of Defaults Used and Defined} = \frac{\text{Number of Subprogram calls with Defaults Used}}{\text{Number of Subprogram calls with Default Possible}}
\]

Proposed Metric \( = 10 \times \text{Ratio of Defaults Used and Defined} \)
Property: Definition and Use of Default Parameters for Stable Values
Attribute: Readability (+)
Rationale: Reducing the amount of superfluous information benefits the reader of a program. By using a default parameter value for stable values, the reader need only consider the parameter when necessary.
Measurement Approach: The measure for this indicator is the number of subprogram calls where defaults were defined and used relative to the number of subprogram calls where default values were possible. (Per Subprogram)

Metric:

\[
\text{Ratio of Defaults Used and Defined} = \frac{\text{Number of Subprogram calls with Defaults Used}}{\text{Number of Subprogram calls with Default Possible}}
\]

Proposed Metric = 10 * Ratio of Defaults Used and Defined
Property: Definition of Default Parameters
Attribute: Well-Defined Interface (+)
Rationale: When default values are defined, it conveys additional information about the module's interface. This improves the clarity of the interface.

Measurement Approach: The measurement approach for this indicator is the number of parameters defined with defaults relative to the total number of parameters. (Per Subprogram)

Metric:

\[
\text{Ratio of Defaults Defined} = \frac{\text{Number of Parameters With Defaults Defined}}{\text{Total Number of Parameters}}
\]

\[
\text{Proposed Metric} = 5 + 5 \times \text{Ratio of Defaults Defined}
\]
**Property:** Use of Parameters with Name Notation

**Attribute:** Complexity (+)

**Rationale:** Because of the self-documenting feature of subprogram calls using name notation, the overall complexity of the program is reduced.

**Measurement Approach:** The measure for this indicator is the average number of parameters from subprogram calls using name notation and having more than five parameters. (Per Subprogram)

**Metric:**

\[
X = \frac{\text{Total Number of Parameters from Calls using Name Notation with } > 5 \text{ Parameters}}{\text{Total Number of Subprogram Calls using Name Notation with } > 5 \text{ Parameters}}
\]

**Proposed Metric**

\[
\text{Max}(0, (10 - (X - 6)))
\]

**Note:** If Total Number of Subprogram Calls using Name Notation with > 5 Parameters = 0
Then Proposed Metric = 5
Property: Use of Parameters with Name Notation
Attribute: Readability (+)
Rationale: The self documenting feature of subprogram calls using name notation inherently improves the readability of the program.
Measurement Approach: The measure for this indicator is the average number of parameters from subprogram calls using name notation with more than five parameters. (Per Subprogram)
Metric:

\[
X = \frac{\text{Total Number of Parameters from Calls using Name Notation with } > 5 \text{ Parameters}}{\text{Total Number of Subprogram Calls using Name Notation with } > 5 \text{ Parameters}}
\]

Proposed Metric \(= \text{Max}(0, (10 - (X - 6)))\)

Note: If Total Number of Subprogram Calls using Name Notation with > 5 Parameters = 0
Then Proposed Metric = 5
Property: Mixing the Order of Parameter Lists

Attribute: Complexity (-)

Rationale: Mixing the order of the parameter lists contributes to incomprehensibility and confusion. This adds to complexity.

Measurement Approach: The measure for this indicator is the number of subprogram calls where the order differs from the subprogram specification relative to the total number of subprogram calls. (Per Subprogram)

Metric:

\[
\text{Percentage of Subprogram Calls with Mixed Notation} = \frac{\text{Total Number of Subprogram Calls with Ordering Different from Subprogram Specification}}{\text{Total Number of Subprogram Calls}}
\]

Proposed Metric = 5 - 5 * Percentage of Subprogram Calls with Mixed Notation
Property: Mixing the Order of Parameter Lists

Attribute: Readability (-)

Rationale: Mixing the order of the parameter lists is confusing to the reader and hence, reduces the readability of the program.

Measurement Approach: The measure for this indicator is the number of subprogram calls where the order differs from the subprogram specification relative to the total number of subprogram calls. (Per Subprogram)

Metric:

\[
\text{Percentage of Subprogram Calls with Mixed Notation} = \frac{\text{Total Number of Subprogram Calls with Ordering Different from Subprogram Specification}}{\text{Total Number of Subprogram Calls}}
\]

Proposed Metric = \(5 - 5 \times \text{Percentage of Subprogram Calls with Mixed Notation}\)
Property: Use of Both Positional and Name Notation in one Subprogram Call

Attribute: Complexity (\( \cdot \))

Rationale: Using both forms of parameter notation in one subprogram call is confusing and hence, increases the overall complexity of the program.

Measurement Approach: The measure for this indicator is the number subprogram calls using mixed notation relative to the total number of subprogram calls. (Per Subprogram)

Metric:

\[
\text{Percentage of Subprogram Calls Using Both Parameter Notations} \quad = \quad \frac{\text{Total Number of Subprogram Calls Using Both Parameter Notations}}{\text{Total Number of Subprogram Calls}}
\]

Proposed Metric

\[
5 - 5 \times \text{Percentage of Subprogram Calls using Both Parameter Notations}
\]
<table>
<thead>
<tr>
<th><strong>Property:</strong></th>
<th>Use of Both Positional and Name Notation in one Subprogram Call</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Attribute:</strong></td>
<td>Readability (-)</td>
</tr>
<tr>
<td><strong>Rationale:</strong></td>
<td>Using both forms of parameter notation in one subprogram call is confusing to the reader and hence, reduces the overall readability of the program.</td>
</tr>
<tr>
<td><strong>Measurement Approach:</strong></td>
<td>The measure for this indicator is the number subprogram calls using mixed notation relative to the total number of subprogram calls. (Per Subprogram)</td>
</tr>
</tbody>
</table>

**Metric:**

\[
\frac{\text{Percentage of Subprogram Calls Using Both Parameter Notations}}{\text{Total Number of Subprogram Calls Using Both Parameter Notations}} = \frac{\text{Total Number of Subprogram Calls}}{\text{Total Number of Subprogram Calls}}
\]

Proposed Metric $=$

\[
5 - 5 \times \text{Percentage of Subprogram Calls using Both Parameter Notations}
\]

*Measures and Metrics for Ada Specific Indicators* 151
Property: Overloading of Subprogram Names
(Does not Include Overloading through Generics)

Attribute: Readability (-)

Rationale: Overloading of subprogram names reduces readability as the number of overloaded names increases. One subprogram name that refers to two or more subprogram definitions causes understandability problems in two main areas: (1) differences in the parameters of the subprograms and (2) differences in the semantics and functionality of the subprograms.

Measurement Approach: The measurement approach for this indicator is the number of unique overloaded subprogram names relative to the total number of unique subprogram names. Intuitively, readability is extremely low if half of the subprogram names are overloaded. Therefore, a factor of two is used in the measure. (Global)

Metric:

\[
\text{Subprogram Name Overloading Factor} = \frac{2 \times \text{Number of Unique Overloaded Subprogram Names}}{\text{Total Number of Unique Subprogram Names}}
\]

Proposed Metric \[= 5 - 5 \times \text{Subprogram Name Overloading Factor}\]

Note: If Subprogram Name Overloading Factor > 1.0 then Proposed Metric = 0
Property: Definition of Declaration Packages
Attribute: Cohesion (+)
Rationale: One use of packages is to group related constants and types (i.e. declarations). This grouping is logically cohesive (related items) and procedurally cohesive (physically in one package).
Measurement Approach: Program units that "with" a declaration package actually reference a percentage of the declarations. A high percentage (or utilization) is indicative of very related items because they were necessary and used in the same program unit. The measure for cohesion is the average utilization for all declaration packages in the program. (Per Declaration Package)
Metric:

\[ \text{Dec Package Utilization} = \sum \frac{\text{number of referenced declarations in the package}}{\text{"Withs" to a Dec Package}} \]

\[ \text{Proposed Metric} = 5 + 5 \times \text{(Dec Package Utilization)} \]
Property: Definition of Declaration Packages
Attribute: Ease of Change (+)
Rationale: The isolation of declarations into one cohesive package aids ease of change because units that may be impacted are easily identified.
Measurement Approach: The measurement approach for this indicator is to measure the reduction of the scope (percentage of units that must be checked) and weight the reduction by the significance of the scope reduction (percentage of code that must be checked). (Per Declaration Package)
Metric:

Reduction of Scope = \frac{\text{# of "withing" units}}{\text{total # of units}}

Significance of Scope Reduction = \frac{\text{TLOC of "withing" units}}{\text{TLOC}}

X = (\text{Reduction of Scope}) \times (\text{Significance of Scope Reduction})

Proposed Metric = 5 + 5 \times X
Property: Insufficient Decomposition of a Declaration Package
Attribute: Ease of Change (-)
Rationale: Insufficient decomposition of declaration packages results in more work during maintenance activity. Program units must be unnecessarily checked for possible impacts caused by changes to declaration packages.

Measurement Approach: Declaration package utilization is a measure for the degree to which declarations in a package are utilized. It is also a measure for the percentage of items that must be checked for possible changes that actually need to be checked. Temper this measure by the significance of the package (total number of declarations relative to all declaration packages) to obtain a measure for ease of change. (Global)

Metric:

\[
\text{Dec Package Significance} = \frac{\text{Total number of declarations in the package}}{\text{Total number of declarations in all declaration packages}}
\]

\[
\text{Tempered Utilization for all Dec Packages} = \sum \left[ \left( \text{Dec Package Utilization} \right) \times \left( \text{Dec Package Significance} \right) \right]
\]

Proposed Metric = \[ 5 \times (\text{Tempered Utilization for all Dec Packages}) \]
Property: Definition of Packages that export Subprograms

Attribute: Cohesion (+)

Rationale: The isolation and localization of related subprograms into one package provides for logical, functional, and procedural cohesion. Functional cohesion is the strongest, and hence, the most important.

Measurement Approach: To measure the cohesiveness of packages that export subprograms, relate the utilization of the subprograms by “withing” units. Intuitively, if the subprograms are sufficiently related, the majority of “withing” units will need to use most of the subprograms. (Per Subprogram Package)

Metric:

Note: The abbreviation "sub package" refers to any package which exports one or more subprograms

Sub Package Utilization = \[ \sum \text{package subprograms referenced} \]
"Withs" to a Sub Package
(total # of "withs") * (# of subprograms in the package specification)

Proposed Metric = 5 + 5 * (Sub Package Utilization)
<table>
<thead>
<tr>
<th>Property:</th>
<th>Definition of Packages that export Subprograms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute:</td>
<td>Ease of Change (+)</td>
</tr>
<tr>
<td>Rationale:</td>
<td>By grouping subprograms together in packages, it is possible to localize code for a group of functionally related subprograms. This creates modularity, resulting in an improvement for ease of change. The placing of the logically related subprograms benefits ease of change by isolating implementation details, thus reducing the “ripple effect.”</td>
</tr>
</tbody>
</table>

**Measurement Approach:**

The measurement approach for ease of change is to measure the total number of subprograms defined in subprogram packages relative to the total number of subprograms in the program. The significance of these subprograms is weighted by the significance of the subprograms (TLOC of the subprogram packages relative to TLOC). (Global)

**Metric:**

\[
\text{Relative number of subprograms in packages} = \frac{\text{Total number of subprograms in package specifications}}{\text{Total number of subprograms in the program}}
\]

\[
\text{Significance of packages which export subprograms} = \frac{\text{TLOC in packages which export subprograms}}{\text{TLOC of all subprograms}}
\]

\[
X = \left( \frac{\text{Relative number of subprograms in packages}}{} \right) \times \left( \frac{\text{Significance of packages which export subprograms}}{} \right)
\]

\[
\text{Proposed Metric} = 5 + 5 \times X
\]
Property: Definition of Packages that export Subprograms

Attribute: Well-Defined Interface (+)

Rationale: A package provides an interface to the subprograms it exports through the package specification. All "withing" units must use this interface to use any of the subprograms. Other than subprogram specifications, the package specification may contain constants, type declarations, variables, and exceptions.

Measurement Approach: One factor in assessing the quality of a package interface is the number of objects being exported. The key objects that will be factored in are variables and subprogram specifications. The clarity of the interface is directly related to the Hair limit. From the Hair limit, five subprograms is the optimal number. Because of the damaging effect of global variables, zero is the optimal number of variables in the package specification. The second factor is global variable references external to the package. It is through these references that information flow occurs that is not documented in the package specification. Due to the major impact of these references, their weight will be twice that of the first factor. (Per Subprogram Package)

Metric:

Note: "Sub Package" refers to any package which exports subprograms

\[
\begin{align*}
\text{Sub Package} & \quad = \quad \text{Number of Global Variables Declared} \\
\text{Global Variable Declaration Factor} & \quad \text{in the Sub Package Specification} \\
\text{Sub Package Subprogram Specs Factor} & \quad = \quad \text{Maximum(0, (Number of Subprogram Specifications in the Package Specification - 5))} \\
\text{Sub Package External Global Variable Reference Factor} & \quad = \quad \text{Number of Distinct External Global Variable References Found in the Sub Package} \\
\text{Interface Rating for a Sub Package} & \quad = \quad \left[ \text{Sub Package Global Variable Declaration Factor} + \text{Sub Package Subprogram Specs Factor} \right] \\
\text{} & \quad \frac{1}{2} \times \text{Sub Package External Global Variable Reference Factor} \\
\text{Proposed Metric} & \quad = \quad \text{Maximum(0, (10 - Interface Rating for a Sub Packages))}
\end{align*}
\]
Property: Units which "with" Packages that export Subprograms

Attribute: Complexity (+)

Rationale: "Withing" packages that export subprograms reduces the overall complexity of the program by reducing the amount of work associated with the program. Instead of having to look at the body of the package, a user need only look at the package specification to use any subprogram exported by the package.

Measurement Approach: The measure for the reduced work associated with a unit is the total lines of code of the "withed" package specifications relative to the total lines of code of the "withed" package bodies. (Per Unit)

Metric:

Note: The abbreviation "sub package" refers to any package which exports one or more subprograms

Reduced Work of a Unit = \[ \sum_{\text{all sub packages "withed" by the unit}} \left( 1 - \frac{TLOC \text{ of the Package Spec}}{TLOC \text{ of the Package Body}} \right) \]

Total # of sub packages "withed" by the unit

Proposed Metric = 5 + 5 * (Reduced Work of the Unit)
Property: Units which “with” Packages that export Subprograms

Attribute: Readability (+)

Rationale: “Withing” packages that export subprograms benefits readability. This benefit occurs because there is simply a subprogram call rather than a subprogram definition and a subprogram call. Every distinct subprogram call to a subprogram from a package benefits the readability of the code.

Measurement Approach: The measurement approach for this indicator is the number of distinct subprogram calls to subprograms from packages relative to the total number of distinct subprogram calls. (Per Unit)

Metric:

Unit Readability Factor = \frac{\text{# of distinct calls to subprograms in "withed" packages}}{\text{total # of distinct calls to all subprograms}}

Proposed Metric = 5 + 5 \times \text{Unit Readability Factor}
Property: Definition of Packages that are never “withed”
Attribute: Complexity (-)
Rationale: Defining packages that are never “withed” adds to the amount of code in the program without adding utility. This adds to the work associated with the program, or the program complexity.
Measurement Approach: The measure for this indicator is the number of packages that are never “withed” relative to the total number of packages. (Global)

Metric:

\[
X = \frac{\text{Total number of packages which are never "withed"}}{\text{Total number of packages}}
\]

Proposed Metric \[= 5 - 5 \times X\]
Property: Definition of Parameterless Generics
Attribute: Complexity (-)
Rationale: A parameterless generic provides a method of copying a group of objects. Because the generic has no parameters; multiple, duplicate copies of the objects are available. This is confusing and increases complexity.
Measurement Approach: The measure for this indicator is gives a value of three if the generic unit is parameterless and a five otherwise. (Per Generic Unit)

Metric:

If Number of Parameterless Generics = 0 Then
   Proposed Metric = 3
Else
   Proposed Metric = 5
Property: Use of Subprograms as Generic Parameters
Attribute: Complexity (-)
Rationale: Ada permits subprograms to be passed as parameters to generic units. The actual generic unit is confusing because the programmer must remember that the formal subprogram name is really a parameter to be specified by the generic instantiation. The functionality of the subprogram is unknown and may have side effects. This makes programming the generic harder, hence, increasing complexity.

Measurement Approach: The measurement approach for this indicator uses the number of subprogram parameters relative to the total number of parameters to the generic. This ratio is multiplied by a factor of two because of the severe impact of this indicator. (Per Generic Unit)

Metric:

\[
\frac{\text{Number of Generic Subprogram Parameters}}{\text{Total Number of Generic Parameters}} = 2 \times \frac{\text{Ratio of Generic Subprogram Parameters}}{}
\]

Proposed Metric \[= 5 - 5 \times \text{Ratio of Generic Subprogram Parameters}\]

Measures and Metrics for Ada Specific Indicators
Property: Use of Subprograms as Generic Parameters

Attribute: Readability (-)

Rationale: Ada permits subprograms as parameters to generic units. The reader sees a subprogram name that is only a placeholder for the real subprogram. Because of the unknown functionality and nondescript name, it is difficult for the reader to understand.

Measurement Approach: The measurement approach for this indicator uses the number of subprogram parameters relative to the total number of parameters to the generic. This ratio is multiplied by a factor of two because of the severe impact of this indicator. (Per Generic Unit)

Metric:

\[
\frac{\text{Ratio of Generic Subprogram Parameters}}{\text{Total Number of Generic Parameters}} = \frac{2}{\text{Number of Generic Subprogram Parameters}}
\]

Proposed Metric = 5 - 5 * Ratio of Generic Subprogram Parameters
Property: Definition of a Generic Unit (One Subprogram or One Package)
Attribute: Complexity (+)
Rationale: When identical algorithms differ only by type, it is less complex to abstract out the types and create templates of program units. Defining Ada generic units creates less complex units due to an additional layer of abstraction being made.

Measurement Approach: The measure for this indicator is the ratio of subprograms defined within a generic unit relative to the total number of subprograms defined. This ratio is multiplied by a factor of two because of the following argument. It is intuitive that if half of the subprograms are defined within a generic unit, the program has much of its complexity reduced because of the abstractions made through the definition of generic units. (Global)

Metric:

Ratio of Generics Subprograms = \( \frac{\text{Total Number of Subprograms Defined within Generic Units}}{\text{Total Number of Subprograms Defined}} \)

Generic Factor = 2 * \( \frac{\text{Ratio of Generics Subprograms}}{\text{Ratio of Generics Subprograms}} \)

Proposed Metric = 5 + 5 * Generic Factor

Note: If Generic Factor > 5 Then Proposed Metric = 10
Property: Definition of a Generic Unit (One Subprogram or One Package)

Attribute: Readability (+)

Rationale: Readability of the program is improved by abstracting out types from algorithms, that creates type independent algorithms. For example, an exchange algorithm can be written independent of the type. Readability and clarity is improved by having only one algorithm instead of multiple copies of similar algorithms.

Measurement Approach: The measure for this indicator is the ratio of subprograms defined within a generic unit relative to the total number of subprograms defined. This ratio is multiplied by a factor of two because of the following argument. It is intuitive that if half of the subprograms are defined within a generic unit, the program’s readability has improved because of the abstractions made through the definition of generic units. (Global)

Metric:

\[
\frac{\text{Ratio of Generics}}{\text{Subprograms}} = \frac{\text{Total Number of Subprograms Defined within Generic Units}}{\text{Total Number of Subprograms Defined}}
\]

\[
\text{Generic Factor} = 2 \times \frac{\text{Ratio of Generics}}{\text{Subprograms}}
\]

Proposed Metric = \[5 + 5 \times \text{Generic Factor}\]

Note: If Generic Factor > 5 Then Proposed Metric = 10
Property:  Multiple Instantiations of a Generic Unit
          (One Subprogram or one Package)

Attribute:  Ease of Change (+)

Rationale:  The benefit to ease of change with regard to generics increases as the
            number of generic instantiations increases. A generic with only one
            instantiation has no benefit to ease of change, whereas a generic
            with five instantiations achieves beneficial results with respect to
            ease of change. Only one program unit need be changed instead of
            five.

Measurement Approach:  The measure for this indicator is the average number of
                      instantiations over all defined generics. (Global)

Metric:

\[
\text{Average Number of Instantiations per Generic} = \frac{\sum \text{Number of Instantiations}}{\text{Total Number of Generics Defined}}
\]

Proposed Metric  =  5 + \text{Min}(5, (\text{Average Number of Instantiations per Generic} - 1))

Note:  If Total Number of Generics Defined = 1 then Proposed Metric = 5
Property: Use of Tasking
Attribute: Complexity (-)

Rationale: Concurrent programs are more difficult to write and debug. This results in increased program complexity.

Measurement Approach: Each subprogram unit that communicates with a task will increase the overall program complexity. The measure for this indicator is the percentage of subprograms that have rendezvous. (Global)

Metric:

\[
\text{Percentage of Subprograms with Rendezvous} = \frac{\text{# of Subprograms with one or more Rendezvous}}{\text{Total # of Subprograms}}
\]

Proposed Metric: \( 5 - 5 \times \text{Percentage of Subprograms with Rendezvous} \)
Property: Definition of Tasks
Attribute: Readability (-)
Rationale: The readability of a task is directly related to its communication with subprograms and other tasks. The number of accept statements defined in a task gives the number of communicational entry points into the tasks. The more of these communicational entry points, the less readable the task becomes.
Measurement Approach: The measure for this indicator is the number of accept statements in the task body. The optimal value for this measure is three or less with an increase in the number of accept statements corresponding to a decrease in the measure for readability. (Global)

Metric:

\[ \sum \text{Number of Accept Statements} \]

\[ \frac{\text{Average Number of Accept Statements}}{\text{Total Number of Defined Tasks}} = \frac{\text{All Tasks}}{5 - \text{Min}((\text{Average Number of Accept Statements} - 3), 5)} \]

Proposed Metric = 5 - Min((Average Number of Accept Statements - 3), 5)

Note: If Total Number of Defined Tasks = 0 Then Proposed Metric = 5
If Average Number of Accept Statements < 3 Then Proposed Metric = 5
Property: Use of Rendezvous with In Out or In and Out Parameters

Attribute: Coupling (-)

Rationale: To achieve low coupling of modules, it is desirable to have minimal communication. The smallest feasible exchange of communication is in one direction per rendezvous. (Information is given to the task or information is retrieved from the task). When the information flow is in both directions during a rendezvous, the task and the unit requesting the rendezvous are excessively coupled.

Measurement Approach: The measure for this indicator is the number of entries defined with In Out or In and Out parameters relative to all defined entries. (Per Subprogram)

Metric:

\[
\text{Entry Ratio} \quad = \quad \frac{\text{Number of Entries with In Out or Both In and Out Parameters}}{\text{Total Number of Entries}}
\]

Proposed Metric \( = \quad 5 - 5 \times \text{Entry Ratio} \)
Property: Use of Unprotected Global Data Areas

Attribute: Coupling (-)

Rationale: A change to the data area by any one of the accessing units can potentially cause a "ripple effect" in any other accessing module. This is a commonly accepted form of common coupling. This is further aggravated by the fact that these units can access the global data area concurrently.

Measurement Approach: The measure for this indicator is the average number of units potentially concurrently coupled by a global data area. (Global)

Metric:

\[ \sum \text{Number of Concurrent Units Accessing the Global Data Area} \]

Average Number of Concurrent Units Using Global Data = \[ \frac{\text{All Global Data Areas}}{\text{Total Number of Global Data Areas}} \]

Proposed Metric = \[ 5 - \min((\text{Average Number of Concurrent Units} - 1), 5) \]

Note: If Total Number of Global Data Areas = 0 Then Proposed Metric = 5
Property: Use of Unprotected Global Data Areas
Attribute: Ease of Change (-)
Rationale: A global data area is one that can be updated by concurrent units. This concurrent aspect is another hindrance that must be dealt with during a change to any one of the concurrent units. With a sequential program, behavior does not deviate without changes to input, a concurrent program's execution is nondeterminant, and hence, repeatability may not be achievable.

Measurement Approach: The measure for this indicator is the average number of units with concurrent access to global data areas. (Global)

Metric:

\[
\sum \text{Number of Concurrent Units Accessing the Global Data Area} = \frac{\text{Average Number of Concurrent Units Using Global Data}}{\text{Total Number of Global Data Areas}}
\]

Proposed Metric = 5 - Min((Average Number of Concurrent Units - 1), 5)

Note: If Total Number of Global Data Areas = 0 Then Proposed Metric = 5
<table>
<thead>
<tr>
<th>Property:</th>
<th>Use of Exception Handlers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute:</td>
<td>Complexity (+)</td>
</tr>
<tr>
<td>Rationale:</td>
<td>The use of exception handlers separates the algorithm from error handling code. This produces two distinct code sections, an algorithm section and an error handling section. This clarity reduces the complexity of the product.</td>
</tr>
<tr>
<td>Measurement Approach:</td>
<td>The measure for this indicator is the number of subprograms with an exception handling section relative to the total number of subprograms. This number is weighted by a factor of two because of the following argument: using exception handlers with at least half of the subprograms is extremely beneficial. (Global)</td>
</tr>
<tr>
<td>Metric:</td>
<td></td>
</tr>
<tr>
<td>Weighted Ratio of Exception Handlers</td>
<td>[ 2 \times \frac{\text{Number of Subprograms with an Exception Handler}}{\text{Total Number of Subprograms}} ]</td>
</tr>
<tr>
<td>Proposed Metric</td>
<td>[ 5 + \min(5, (5 \times \text{Weighted Ratio of Exception Handlers})) ]</td>
</tr>
<tr>
<td>Property:</td>
<td>Relying on Upper Level Subprograms to Handle User Raised Exceptions</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>Attribute:</td>
<td>Coupling (-)</td>
</tr>
<tr>
<td>Rationale:</td>
<td>A user raised exception should have a corresponding exception handler in the same subprogram. If an exception handler does not exist, the exception is propagated to upper level subprograms. This couples the subprograms via the exception propagation.</td>
</tr>
<tr>
<td>Measurement Approach:</td>
<td>The measure for this indicator is the number of subprograms that raise errors, but do not handle them. This measure is not made relative to the total number of subprograms because the presence of the indicator is simply inherently detrimental. (Global)</td>
</tr>
<tr>
<td>Metric:</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>= Number of Subprograms which Raise Exceptions Not Handled in that Subprogram's Exception Handlers</td>
</tr>
<tr>
<td>Proposed Metric</td>
<td>= 5 - Min(5,X)</td>
</tr>
</tbody>
</table>
Appendix G

Metrics for Ada Constructs by Software

Engineering Attribute
Metric Definition Notes

A variable reference is global to a subprogram if it is NOT (a) a parameter and (b) a locally declared variable.

A variable reference is global to a package if it is NOT declared in either the package specification or package body.

A variable reference is global to a generic if it is NOT a parameter of the generic.

A variable reference is global to a task if it is NOT (a) a locally declared variable and (b) a parameter to the entry.

The term parameters refers to parameters of both subprogram calls and entry calls unless explicitly stated otherwise.

Blocking structures (BS) include if statements, case statements, loop statements, and block statements.

Package length includes both the package specification and the package body.

Lines Of Code (LOC) Limited Terminal Semicolon Definition: Exclude those semicolons in character strings and comments and those in formal parameter lists.

A parameter is a switch if (1) the mode is In or In Out and (2) it is used in a decision statement.

Metrics assessed on a per unit basis do not "double" count any items. For example, a discriminant defined within a subprogram within a package would be counted for the subprogram only.
Coupling

Goal: Reduce Coupling

1. **Number of global variables referenced**

   \[
   y = \frac{\text{number of unique global variables referenced}}{\text{number of unique global, unique local variables, and unique parameters referenced}}
   \]

   Proposed metric = 5 - 5y

2. **Number of parameters passed**

   \[
   a = \frac{\text{total number of parameters}}{\text{total number of calls}}
   \]

   \[
   y = (10 - a) \times \frac{\text{number of unique parameters passed}}{\text{number of unique global variables referenced} + \text{unique parameters passed}}
   \]

   if number of calls = 0 \quad \text{Proposed metric} = 5
   \]

   if number of calls ≠ 0 \quad \text{Proposed metric} = y

3. **Types of parameters passed**

   \[
   y = \text{number of calls that have switches}
   \]

   if y = 0 \quad \text{Proposed metric} = 10
   \]

   if 0 < y < 6 \quad \text{Proposed metric} = 6 - y
   \]

   if 6 ≤ y \quad \text{Proposed metric} = 0

4. **Number of entry points in a subprogram and a task**

   \[
   y = \text{number of unique entry points to a task or subprogram}
   \]

   Proposed metric = 6 - y

5. **Number of structured data types passed as parameters**

   Proposed metric = 5 - \frac{\text{number of structured data type parameters}}{\text{total number of parameters}} \times 5

---

\(^{\dagger} \dagger - \text{indicates metric is applied to each subprogram and then a program average is calculated}\)
Coupling, Continued

7. Use of rendezvous with in out or in and out parameters †

\[
\text{Entry Ratio} = \frac{\text{Number of Entries with In Out or Both In and Out Parameters}}{\text{Total Number of Entries}}
\]

Proposed Metric = 5 - 5 * Entry Ratio

8. Use of unprotected global data areas \( \mathcal{Y} \)

\[
\text{Average Number of Concurrent Units Using Global Data} = \frac{\sum \text{Number of Concurrent Units Accessing the Global Data Area}}{\text{All Global Data Areas}}
\]

Proposed Metric = 5 - \( \min((\text{Average Number of Concurrent Units} - 1), 5) \)

Note: If Total Number of Global Data Areas = 0 Then Proposed Metric = 5

9. Relying on upper level subprograms to handle user raised exceptions \( \mathcal{Y} \)

\[ X = \frac{\text{Number of Subprograms which Raise Exceptions Not Handled in that Subprogram's Exception Handlers}}{\text{All Subprograms}} \]

Proposed Metric = 5 - \( \min(5, X) \)

---

\( \mathcal{Y} \) - indicates metric is applied globally to the entire program

Metrics for Ada Constructs by Software Engineering Attribute
Cohesion

Goal: Enhance Cohesion

1. Number of blocking structures †

   Level 0 indicates a blocking structure that is not nested within another blocking structure

   \[ \text{ave loc per BS} = \frac{\text{TOTAL LINES OF SOURCE CODE FOR ALL PROGRAMS}}{\text{TOTAL BS AT LEVEL 0 FOR ALL PROGRAMS}} \]

   \[ \text{expected BS at level 0} = \frac{\text{total loc}}{\text{ave loc per BS}} \]

   \[ \text{proposed metric} = \frac{\text{total loc enclosed by BS at level 0}}{\text{total loc}} \times 10 \times \frac{\text{expected BS at level 0}}{\text{number of BS at level 0}} \]

2. Division of code into logical units which perform single specific functions †

   Proposed metric = 5 + \frac{\text{total loc enclosed by BS at level 0}}{\text{total loc}} \times 5

3. Definition of declaration packages §

   \[ \text{Dec Package Utilization} = \sum \text{number of referenced declarations in the package} \]

   \[ \text{"Withs" to a Dec Package} \]

   \[ \text{(total # of "withs") \times (total # of declarations)} \]

   Proposed Metric = 5 + 5 \times (\text{Dec Package Utilization})
Cohesion, Continued

4. Definition of packages that export subprograms §

Note: The abbreviation "sub package" refers to any package which exports one or more subprograms

\[
\text{Sub Package Utilization} = \sum \text{package subprograms referenced} \]

"Withs" to a Sub Package

(total # of "withs") * (# of subprograms in the package specification)

Proposed Metric = 5 + 5 * (Sub Package Utilization)

\(^\text{§} - \text{indicates metric is applied to each package and then a program average is calculated}\)
Well-Defined Interfaces

Goal: Improve software/software and software/hardware interfaces

1. Number of global variables †

\[ y = \frac{\text{number of unique global variables referenced}}{\text{number of unique global variables referenced} + \text{number of unique parameters}} \]

Proposed metric = 5 - 5y

2. Use of parameters in subprogram and entry calls †

\[ y = \frac{\text{number of unique parameters passed}}{\text{number of unique global variables referenced} + \text{number of unique parameters passed}} \]

Proposed metric = 5 + 5y

3. Use of parameterless procedure calls †

\[ y = \frac{\text{number of unique parameterless calls}}{\text{number of unique calls}} \]

Proposed metric = 5 - 5y

4. Number of data structures used with respect to intra-routine communication †

\[ y = \frac{\text{number of unique structured data type references}}{\text{number of unique variable references}} \]

if \( y \geq 0.8 \) scale = 10
if \( 0.5 < y < 0.8 \) scale = 5 + 6y
if \( 0.15 \leq y \leq 0.5 \) scale = 10 - 1.5/y
if \( y < 0.15 \) scale = 0

5. Use of “Excessive” Number of Parameters †

If (Number of Parameters > 3) Then
   Proposed Metric = 5 - (4 - Number of Parameters)
Else
   Proposed Metric = 5

Metrics for Ada Constructs by Software Engineering Attribute
Well-Defined Interfaces, Continued

6. Definition of default parameters ♦

\[
\frac{\text{Ratio of Defaults Defined}}{\text{Number of Parameters With Defaults Defined}} = \frac{\text{Total Number of Parameters}}{}
\]

Proposed Metric = 5 + 5 \times \text{Ratio of Defaults Defined}

7. Definition of packages that export subprograms §

Note: "Sub Package" refers to any package which exports subprograms

\[
\begin{align*}
\text{Sub Package Global Variable Declaration Factor} & = \text{Number of Global Variables Declared in the Sub Package Specification} \\
\text{Sub Package Subprogram Specs Factor} & = \text{Maximum}(0, \text{Number of Subprogram Specifications in the Package Specification}) \\
\text{Sub Package External Global Variable Reference Factor} & = \text{Number of Distinct External Global Variable References Found in the Sub Package} \\
\text{Interface Rating for a Sub Package} & = \left[ \text{Sub Package Global Variable Declaration Factor} + \text{Sub Package Subprogram Specs Factor} + \left( 2 \times \text{Sub Package External Global Variable Reference Factor} \right) \right]
\end{align*}
\]

Proposed Metric = Maximum(0, (10 - Interface Rating for a Sub Packages))
Complexity

Goal: Reduce complexity

1. **Length of subprograms †**
   
   \[ y = 40 / \text{subprogram length} \]
   
   Proposed metric = 5y

2. **Length of packages §**
   
   \[ y = 300 / \text{package length} \]
   
   Proposed metric = 5y

3. **Number of blocking structures used †**
   
   Proposed metric is computed as in Cohesion # 1

4. **Maximum nesting level †**
   
   Compute the following for each blocking structure at level 0:
   
   if maximum nesting level \( \leq 2 \)
   
   \[ y = 10 \]
   
   else
   
   \[ y = 10 - 10 \times (\text{maximum level} - 2) \times \frac{\text{loc enclosed at level 3}}{\text{loc enclosed at level 0}} \times \frac{\text{loc enclosed at level 0}}{\text{total loc}} \]
   
   Proposed metric = \( \frac{\text{sum of every } y}{\text{number of control structures at level 0}} \)

5. **Number of goto statements used †**
   
   \[ y = 5 - 2 \times \text{number of gotos} \]
   
   if number of gotos \( \neq 0 \)
   
   Proposed metric = y

6. **Number of block comments †**
   
   Proposed metric = number of block comments \times \frac{\text{number of block comment lines}}{\text{total number of comment lines}}

7. **Number of single line comments †**
   
   \[ y = \frac{\text{number of single line comments}}{\text{total loc}} \]
   
   Proposed metric = 10 - 20 \times (y - 0.2)
Complexity, Continued

8. Number of structured data types used †

Proposed metric = 5 + \textit{number of structured data type parameters} \times 5
\textit{total number of parameters}

9. Number of if statements †

\[ y = \frac{\text{TOTAL LINES OF SOURCE CODE FOR ALL PROGRAMS}}{\text{TOTAL NUMBER OF IF STATEMENTS}} \]
\[ z = \frac{\text{total loc}}{y} \]

if number of ifs \leq z

Proposed metric = 10

if number of ifs > z

Proposed metric = 10 - 3 \times (\text{number of ifs} - z)

10. Use of both default parameters and positional notation †

\textit{Ratio of Omitted Parameters} = \frac{\text{Number of omitted parameters in all calls using positional notation}}{\text{Total number of parameters possible in all calls using positional notation}}

Proposed Metric = 5 - 5 \times \text{Ratio of Omitted Parameters}

11. Definition and use of default parameters for stable values †

\textit{Ratio of Defaults Used and Defined} = \frac{\text{Number of Subprogram calls with Defaults Used}}{\text{Number of Subprogram calls with Default Possible}}

Proposed Metric = 10 \times \text{Ratio of Defaults Used and Defined}
Complexity, Continued

12. Use of parameters with name notation †

\[ X = \frac{\text{Total Number of Parameters from Calls using Name Notation with } > 5 \text{ Parameters}}{\text{Total Number of Subprogram Calls using Name Notation with } > 5 \text{ Parameters}} \]

Proposed Metric = \( \text{Max}(0, (10 - (X - 6))) \)

Note: If Total Number of Subprogram Calls using Name Notation with > 5 Parameters = 0
Then Proposed Metric = 5

13. Mixing the order of parameter lists †

\[ \text{Percentage of Subprogram Calls with Mixed Notation} = \frac{\text{Total Number of Subprogram Calls with Ordering Different from Subprogram Specification}}{\text{Total Number of Subprogram Calls}} \]

Proposed Metric = \( 5 - 5 \times \text{Percentage of Subprogram Calls with Mixed Notation} \)

14. Use of both positional and name notation in a single subprogram call †

\[ \text{Percentage of Subprogram Calls Using Both Parameter Notations} = \frac{\text{Total Number of Subprogram Calls Using Both Parameter Notations}}{\text{Total Number of Subprogram Calls}} \]

Proposed Metric = \( 5 - 5 \times \text{Percentage of Subprogram Calls using Both Parameter Notations} \)
Complexity, Continued

15. Units which "with" packages that export subprograms $^4$

Note: The abbreviation "sub package" refers to any package which exports one or more subprograms

\[
\text{Reduced Work of a Unit} = \sum_{\text{all sub packages "withed" by the unit}} \left(1 - \frac{\text{TLOC of the Package Spec}}{\text{TLOC of the Package Body}}\right)
\]

Total # of sub packages "withed" by the unit

Proposed Metric $= 5 + 5 \times (\text{Reduced Work of the Unit})$

16. Definition of packages that are never "withed" $^5$

\[
X = \frac{\text{Total number of packages which are never "withed"}}{\text{Total number of packages}}
\]

Final Metric $= 5 - 5 \times X$

17. Definition of parameterless generics $^\ddagger$

If (Number of Parameterless Generics $\neq 0$) Then
Proposed Metric $= 3$

Else
Proposed Metric $= 5$

18. Use of record discriminants $^\wedge$

\[
\text{Ratio of Types} = \frac{\text{Number of Discriminant Types Defined}}{\text{Total Number of User Defined Types Defined}}
\]

Proposed Metric $= 5 + 5 \times \text{Ratio of Types}$

$^4$ $^\wedge$ indicates metric is applied on a per unit basis

$^5$ $^\ddagger$ indicates metric is applied on a per generic unit basis
Complexity, Continued

19. Use of subprograms as generic parameters ‡

\[
\text{Ratio of Generic Subprogram Parameters} = 2 \times \frac{\text{Number of Generic Subprogram Parameters}}{\text{Total Number of Generic Parameters}}
\]

Proposed Metric = 5 - 5 * Ratio of Generic Subprogram Parameters

20. Definition of a generic unit ¥

\[
\text{Ratio of Generics Subprograms} = \frac{\text{Total Number of Subprograms Defined within Generic Units}}{\text{Total Number of Subprograms Defined}}
\]

Generic Factor = 2 * Ratio of Generics Subprograms

Proposed Metric = 5 + 5 * Generic Factor

Note: If Generic Factor > 5 Then Proposed Metric = 10

21. Use of tasking ¥

\[
\text{Percentage of Subprograms with Rendezvous} = \frac{\# \text{ of Subprograms with one or more Rendezvous}}{\text{Total # of Subprograms}}
\]

Proposed Metric = 5 - 5 * Percentage of Subprograms with Rendezvous

22. Use of exception handlers ¥

\[
\text{Weighted Ratio of Exception Handlers} = 2 \times \frac{\text{Number of Subprograms with an Exception Handler}}{\text{Total Number of Subprograms}}
\]

Proposed Metric = 5 + Min(5, (5 * Weighted Ratio of Exception Handlers))
Readability

Goal: Enhance readability

1. Use of blocking structures †

\[ z = \frac{\text{TOTAL LINES OF SOURCE CODE FOR ALL PROGRAMS}}{\text{TOTAL NUMBER OF BLOCKING STRUCTURES}} \]

expected number of blocking structures = \(\frac{\text{total loc}}{z}\)

\[ y = \frac{\text{expected number of blocking structures}}{\text{actual number of blocking structures}} \]

\begin{align*}
\text{if } y & \leq 1.0 & \text{proposed metric} &= 10 \\
\text{if } 1.0 < y \leq 1.5 & \text{proposed metric} &= 10 - (y - 1) \times 20 \\
\text{if } y > 1.5 & \text{proposed metric} &= 0 
\end{align*}

2. Use of goto statements †

Proposed metric is computed as in Complexity #5

3. Use of block header comments †

Proposed metric is computed as in Complexity #6

4. Use of single line comments †

Proposed metric is computed as in Complexity #7

5. Subprogram length †

Proposed metric is computed as in Complexity #1

6. Package length §

Proposed metric is computed as in Complexity #2

7. Use of symbolic constants (literals) †

\[ y = 8 - \frac{\text{number of non-symbolic constants referenced}}{2} \]

\begin{align*}
\text{if number of non-symbolic constants referenced} &= 0 & \text{Proposed metric} &= 10 \\
\text{if number of non-symbolic constants referenced} & \neq 0 & \text{Proposed metric} &= y 
\end{align*}

8. Use of both default parameters and positional notation †

Proposed metric is computed as in Complexity #11
Readability, Continued

9. Definition and use of default parameters for stable values †
   Proposed metric is computed as in Complexity #12

10. Use of parameters with name notation †
    Proposed metric is computed as in Complexity #13

11. Mixing the order of parameter lists †
    Proposed metric is computed as Complexity #14

12. Use of both positional and name notation in a single subprogram call †
    Proposed metric is computed as in Complexity #15

13. Overloading of subprogram names ‡

\[
\text{Subprogram Name Overloading Factor} = \frac{2 \ast \text{Number of Unique Overloaded Subprogram Names}}{\text{Total Number of Unique Subprogram Names}}
\]

Proposed Metric \(= 5 - 5 \ast \text{Subprogram Name Overloading Factor} \)

Note: If Subprogram Name Overloading Factor > 1.0 then Proposed Metric = 0

14. Units which “with” packages that export subprograms ‡

\[
\text{Unit Readability Factor} = \frac{\# \text{ of distinct calls to subprograms in "withed" packages}}{\text{total \# of distinct calls to all subprograms}}
\]

Proposed Metric \(= 5 + 5 \ast \text{Unit Readability Factor} \)

15. Use of subprograms as generic parameters ‡
    Proposed metric is computed as in Complexity #19

16. Definition of a generic unit ‡
    Proposed metric is computed as in Complexity #20
Readability, Continued

17. Definition of tasks $\gamma$

\[ \sum \text{Number of Accept Statements} \]
\[ \frac{\text{Average Number of Accept Statements}}{\text{All Tasks}} \]
\[ \frac{\text{Total Number of Defined Tasks}}{} \]

Proposed Metric \[ = \]

\[ 5 - \text{Min}((\text{Average Number of Accept Statements} - 3), 5) \]

Note: If Total Number of Defined Tasks = 0 Then Proposed Metric = 5
If Average Number of Accept Statements < 3 Then Proposed Metric = 5
Ease of Change

Goal: Enhance extensibility

1. Use of global variables †

   Proposed Metric as in Coupling # 1

2. Use of distinct functions within a single module †

   Proposed Metric is computed as in Cohesion # 2

3. Definition of declaration packages §

   Reduction of Scope = \frac{\text{# of "withing" units}}{\text{total # of units}}

   Significance of Scope Reduction = \frac{\text{TLOC of "withing" units}}{\text{TLOC}}

   \[ X = (\text{Reduction of Scope}) \times (\text{Significance of Scope Reduction}) \]

   Proposed Metric = 5 + 5 \times X
Ease of Change, Continued

4. Insufficient decomposition of a declaration package $\gamma$

$$\text{Dec Package Significance} = \frac{\text{Total number of declarations in the package}}{\text{Total number of declarations in all declaration packages}}$$

$$\text{Tempered Utilization} = \sum_{\text{All Dec Packages}} \left[ \left( \text{Dec Package Utilization} \right) \times \left( \text{Dec Package Significance} \right) \right]$$

Proposed Metric $= 5 \times (\text{Tempered Utilization for all Dec Packages})$

5. Definition of packages that export subprograms $\gamma$

$$\text{Relative number of subprograms in packages} = \frac{\text{Total number of subprograms in package specifications}}{\text{Total number of subprograms in the program}}$$

$$\text{Significance of packages which export subprograms} = \frac{\text{TLOC in packages which export subprograms}}{\text{TLOC of all subprograms}}$$

$$X = \left( \frac{\text{Relative number of subprograms in packages}}{\text{Significance of packages which export subprograms}} \right)$$

Proposed Metric $= 5 + 5 \times X$
Ease of Change, Continued

6. Multiple instantiations of a generic unit

\[
\text{Average Number of Instantiations per Generic} = \frac{\sum \text{Number of Instantiations of all generics}}{\text{Total Number of Genetics Defined}}
\]

Proposed Metric = 5 + \( \text{Min}(5, (\text{Average Number of Instantiations per Generic} - 1)) \)

Note: If Total Number of Generics Defined = 1 then Proposed Metric = 5

7. Use of unprotected global data areas

Proposed metric is computed as in Coupling #8
Early Error Detection

Goal: Find errors as early as possible in the life cycle

1. Use of record discriminants

   Proposed Metric is computed as in Complexity # 18
Vita

Gary Neal Bundy was born on October 17, 1965 in Wilson, North Carolina. After graduating salutatorian of North Lenoir High School's 1984 graduating class, Gary entered North Carolina State University with computer science as his major. While at North Carolina State University, Gary was a cooperative education student with Northern Telecom, Inc. in the Research Triangle Park, North Carolina and a college preprofessional with International Business Machines, Inc., also in the Research Triangle Park. In May, 1988, Gary graduated magna cum laude from North Carolina State University with a B.S. in computer science. In August, 1988, Gary entered graduate school at Virginia Polytechnic Institute and State University. While at Virginia Tech, Gary was a Graduate Project Assistant with the Systems Research Center. He received his Master's Degree in July, 1990. Currently, Gary is a Member of the Technical Staff with the MITRE Corporation in McLean, Virginia.

[Signature]

Gary Neal Bundy